Adaboost and its application using classification trees

by

Nishay Vithal

Submitted in partial fulfillment of the requirements

for the degree of

Master of Science: Mathematical Statistics

In the Faculty of Natural & Agricultural Sciences

University of Pretoria

Pretoria

October 2013

Supervisor: Dr FHJ Kanfer
Co-supervisor: Mr SM Millard

[For further request for information please contact nvithal@iburst.co.za]

Abstract

This mini-dissertation seeks to provide the reader with an understanding of one of the most popular boosting methods in use today called Adaboost and its first extension Adaboost.M1. Boosting, as the name suggests, is an ensemble and machine learning method created to improve or "boost" prediction accuracy via repeated Monte-Carlo type simulations. Due to the methods flexibility to be applied over any learning algorithm, in this dissertation we have chosen to make use of decision trees, or more specifically classification trees constructed by the CART method, as a base predictor. The reason for boosting classification trees include the learning algorithms lack of accuracy when applied on a stand-alone basis in many settings, its practical real world application and the ability for classification trees to perform natural internal feature selection. The core topics covered include where the Adaboost method arose from, how and why it works, possible issues with the method and examples using classification trees as the base predictor to demonstrate and assess the methods performance. Although no formal mathematical derivation of the method was provided at the time the method was created, a statistical justification was put forward several years later which explained Adaboost in terms of well known additive modelling when minimizing a specific exponential loss function or criterion. This justification is provided along

with real and simulated examples demonstrating Adaboost's performance using two types of classification trees i.e. *stumps* (classification trees with two terminal nodes) and optimized or pruned full trees. What is shown empirically is that when boosting tree stumps the performance enhancements achieved by Adaboost in many cases meets or exceeds the single or boosted larger tree structures. This finding has benefits such as simplified model structures and lower computational time. Lastly we provide a cursory review of new developments within the field of boosting such as margin theory which seeks to provide an explanation as to the methods seemingly mysterious test and training error performance; optimized tree boosting procedures such as gradient boosted methods and combinatorial ensemble methods using bagging and boosting.

Acknowledgments

I would like to thank my mum and dad for their continued support and words of encouragement in motivating me to complete this dissertation. I would also like to thank my fiance for being the bed-rock in my life who has taken care of me and provided me with a warm environment which has allowed me to complete this dissertation.

Contents

List of terms and notational conventions

- **Monte-Carlo simulations**: a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results
- **Hypothesis**: refers to a learning function and should not be confused with "hypothesis" used within the context of *hypothesis testing*
- **Dendogram**: visual representation of a decision tree
- **Split point**: a decision criteria based on split variables which splits the data accordingly into various sub-sets in order to construct a decision tree
- **Split variable(s)**: the input variable which is selected to form part of the decision criteria in order to construct a decision tree
- **Tree node**: a subset of data represented in a dendogram which has been created by an earlier split point and split variable
- $I(\cdot)$: denotes an indicator function with the argument defining an output of 1 if true or 0 if false
- **Neural Nets:** abbreviation for *neural networks* which collectively defines or encompasses a large class of non-linear statistical learning methods (see Hastie et al. [4] p 389-416)

- **SVM**: acronym for *support vector machines* which is a method of separating a feature space where there are non-separable or overlapping classes (see Hastie et al. [4] p 417-438)

- **MARS**: acronym for *multivariate adaptive regression splines* which is a procedure using regression to cater for higher dimensional or multivariate problems (see Hastie et al. [4] p 423-437)

- **k-Kernels**: similar to SVM, a method of simplifying learning methods by mapping a feature space into a higher dimension in order to produce non-linear models using linear models derived in higher dimensional space (see Hastie et al. [4] p 670)

- **Machine learning**: a branch of artificial intelligence which concerns with the construction and study of systems that can learn from data

- **Signal processing**: an area of systems engineering, electrical engineering and applied mathematics that deals with operations on or analysis of signals, or measurements of time-varying or spatially varying physical quantities (see Hastie et al. [4] p 139-189)

- **Bagging**: an abbreviation for *bootstrap aggregating* is a machine learning ensemble meta-algorithm designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression. It has also been known to reduce variance and helps to avoid overfitting

- Note, the input instance variable $\mathbf{x}$ where $\mathbf{x}_i = \langle x_{1i}, x_{2i}, ..., x_{pi} \rangle'$, $i = 1, ...N$ and N being the number of observations, is a p dimensional vector and the domain space $\mathbf{X}$ where $\mathbf{x}_i \in \mathbf{X}$ will be kept consistent throughout this dissertation

- Reference to any term raised to the power t or m such as x^t or x^m will denote a reference to a index, usually time, and should not be confused as an exponent unless explicitly stated otherwise

Chapter 1
Introduction

Throughout the history of statistics a core focus and effort has related to predictive modelling due to its numerous real-life applications. Many advanced and complex models have been developed over the years in a bid to ultimately improve accuracy or prediction performance. The concepts of *supervised* and *machine learning* emerged as strong contenders in producing such predictive models which was based on defined algorithms using historical or a given set of information to find and model relationships between variables. However these individual models eventually suffered from accuracy limitations and a new method was required if accuracy was to be improved. It was this desire to improve predication accuracy using a wide range of available learning algorithms which triggered the creation of what is known today as *ensemble methods*. This class of learning methods was underpinned by the simple idea of combining several individual predictor models, known as base predictors, to create a more powerful overall model. Its application appeared to be useful in cases where such individual predictors which suffered from inaccuracy, known as *weak learners*, could be improved by combining them into a *committee* of predictors and therefore improving the overall result and ultimately producing what is known as a *strong learner*. **Boosting** is one such method of constructing an ensemble of

base predictors to produce a powerful predictive model. Other ensemble methods include Bayes optimal classifier (see Mitchell [1]), bagging (see Breiman [2]) and Bayesian model averaging (see Hoeting et al [3]) with each possessing their own respective strengths and weaknesses in terms of accuracy and applicability. Whilst Bayesian methods are often challenging to implement, boosting and bagging are relatively straightforward. The difference between the latter two methods stems from accuracy whereby boosting has proven to be superior (see Figure 1.1). Although there are many boosting techniques currently available this dissertation focuses on the most popular method called **Adaboost** due to its ease (straightforward to program) and flexibility of use (works across multiple settings i.e. binary-classification, multi-classification and regression). In addition since ensemble methods can effectively be applied using any base predictor this dissertation assumes the use of *decision trees* as such a base predictor. The primary reason for selecting decision trees is that this particular learning method has generally shown to perform poorly on an individual basis and therefore presents itself as an ideal candidate for boosting. Further reasons for selecting decision trees as a base predictor is provided in Section 3.1. Within the field of decision trees there are typically two sub-methods which exist being *regression trees* for quantitative responses and *classification trees* typically used for qualitative or binary responses. As the topic of regression type problems generally receives more research attention than classification type problems, this dissertation focuses on the latter less studied classification trees.

In order to demonstrate the practical application of Adaboost a real-life example will now be presented. In this example the aim is to illustrate the power of boosting using the Adaboost technique with classification trees as a base predictor. The results of the Adaboost method will then be compared to its primary competitor method, bagging. Since boosting is by construction an iterative process and improves with the number of iterations, diagrams shown within this dissertation will typically plot the error rates as a function of the boosting iterations. Figure 1.1 uses the **SPAM** dataset from Hastie et al.[4] (p 305-317) which seeks to develop a model which can filter spam e-mails from non-spam e-mails using several predictor variables e.g. counting words containing unusual characters like $%#@, the number of words associated with spam e-mail such as lucky, money, winner etc. Further details of this dataset are provided in Section 3.4. Under the subject of classification trees two specific sub-tree types are encountered and made use of exclusively within this dissertation:

- *stumps* i.e. trees containing a single split and therefore possessing only two terminal nodes and
- *pruned trees* which are trees generated by creating very large trees which are then *pruned* i.e. collapsing of non-terminal internal nodes, to produce a smaller and more optimized tree structure.

As a note, all trees constructed and used in this dissertation are based on binary split points to create simpler tree structures and facilitate quicker implementation.

As can be seen from Figure 1.1 based on the **SPAM** dataset, when using stumps as a base predictor, boosting produces far superior results to bagging (compare the solid red line to the solid blue line). In this case boosting produces an error rate of less than half that of bagging. In addition, for less than 10 boosting iterations, the performance of the boosted stumps matches that of the single pruned tree and actually begins to outperform the pruned tree as the number of boosting iterations continue. Similarly when using the pruned tree as a base classifier, boosting produces superior results to bagging (compare the dotted red line to the dotted blue line). What is also evident from Figure 1.1 is that the performance of the boosted stumps is comparable to the performance of the bagged pruned tree in terms of accuracy (compare the solid red line to the dotted blue line). The benefit of boosting the stumps however is that the process is far less computationally intensive than bagging the pruned tree and therefore boosting, in this context, would be a significantly more efficient method to use.

The question as to how and why Adaboost improves prediction performance provides the underlying motivation for this dissertation. Initially when the first boosting methods where developed by Robert E. Schapire in 1989 the performance of the method was somewhat of a mystery and could only be understood at an empirical level. A firm theoretical foundation at that time had yet to be provided. Simi-

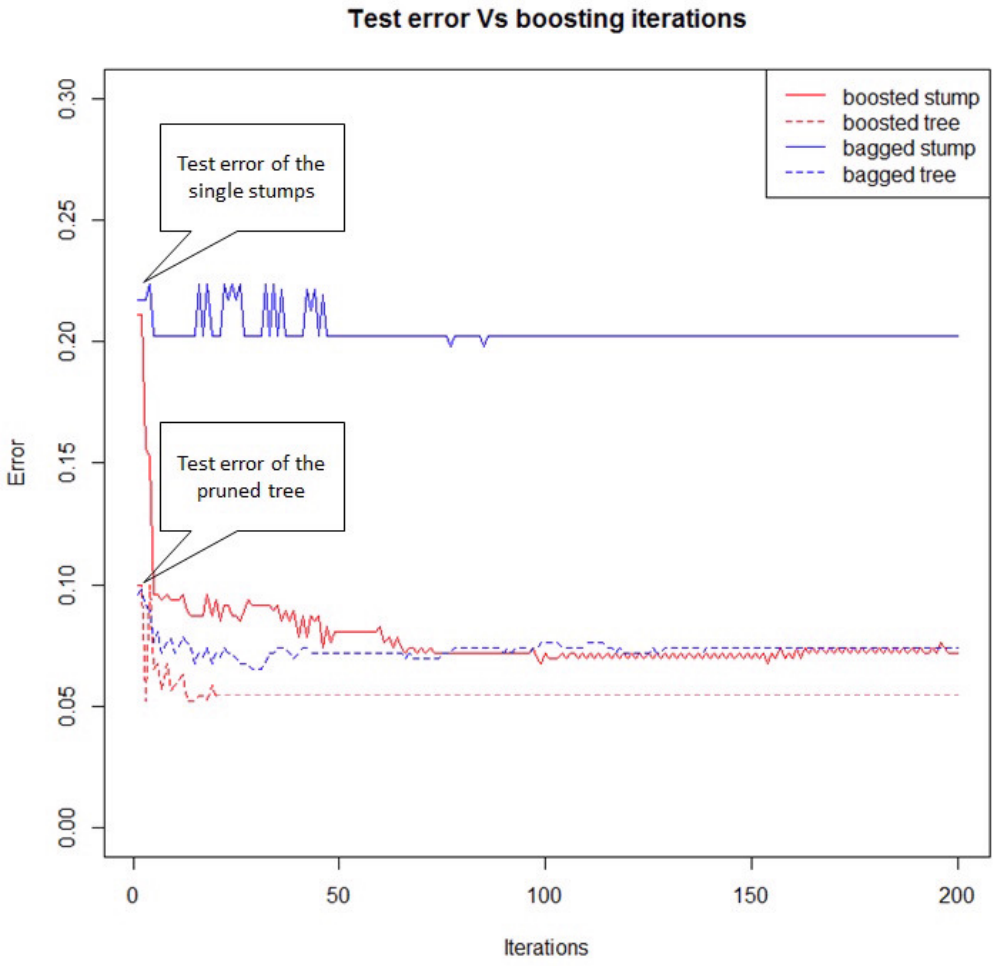


Figure 1.1: Graph showing the test error rate of the **SPAM** dataset as a function of the boosting and bagging iterations (re-created as per Hastie et al.[4] (p 305-317))

larly the development of Adaboost stemmed from an extension of another predefined method known as Hedge(β) which was developed to optimally allocate weights over 'choices' or strategies with the objective of minimizing some loss function over these 'choices'. Thereafter the first multi-class extension to Adaboost called Adaboost.M1 was formulated by Freund et al. [5] which as the name suggested allowed Adaboost to be applied in non-binary multi-classification settings. In Section 2.4 an examination of the genesis of Adaboost and consequently Adaboost.M1 is provided. The Adaboost.M1 method is compared to the original Adaboost method to understand the similarities and differences, if any, and lastly Adaboost is compared to its founding method Hedge(β).

Although the derivation of Adaboost is only explained through a comparative analysis, its mathematical justification emerged several years later after its development. Such a justification took the form of well-known additive modelling using a specific exponential loss function and criterion. In Section 2.5 the mathematics of additive modeling and the exponential loss function and criterion are explored and compared to each step of the Adaboost.M1 method to reaffirm equivalence.

In supplying the above information an answer to the 'how' question is provided whilst the 'why' question remains unexplained. The question as to why Adaboost improves prediction accuracy is reduced in Section 2.6 to understanding the theoretical error rate of the method and corresponding error bounds. If the Adaboost method re-

sults in producing narrow or low error bounds then a theoretical explanation as to its performance can be obtained.

Given that classification trees have been selected as the base predictor it would be of interest to understand *why* it has been selected, *how* classification trees work and *test* if it produces the desired results. The answers to these questions will be covered in Chapter 3.

Although the example in Figure 1.1 demonstrates the superior performance of Adaboost using classification trees as a base predictor it would be of interest to see the results of the method applied to different datasets. In Chapter 4 such an analysis is performed by applying the Adaboost.M1 method using classification trees as a base predictor to both simulated and real datasets and comparing the test error rates at different boosting iterations, in essence by re-creating and analyzing Figure 1.1 for each dataset. In addition, for each example in Chapter 4, Adaboost is performed using both the stump and pruned tree as the underlying base predictor with the output analyzed and compared to the results obtained in the preceding Chapter 3.

The layout of this dissertation is simple and assumes a reader with a moderate understanding of statistics but with no prior knowledge of boosting, bagging or any other ensemble method for that matter. There are five main chapters which comprise this dissertation including the *Introduction* which consist of the *Theoretical Framework of Adaboost, Classification Trees as a Base Predictor, Application of Adaboost*

and the *Conclusion*. The chapter on the *Theoretical Framework of Adaboost* explains the origins of boosting, the Adaboost methodology and algorithm, the derivation of the method, its mathematical justification and theoretical error bounds relating to its performance. The chapter on *Classification Trees as a Base Predictor* provides an analysis on the choice to use decision trees as a base predictor, gives a theoretical and practical explanation of classification trees and lastly shows examples using classification trees on a combination of simulated, real, binary and multi-classification datasets. The penultimate chapter *Application of Adaboost* analyzes the same datasets as given in the previous chapter for comparative purposes and seeks to determine the effectiveness of the method with the overall objective of illustrating the performance of the Adaboost technique. Lastly the *Conclusion* chapter presents some of the latest developments in the field of boosting and ends with some concluding remarks and arising questions.

Chapter 2
Theoretical Framework of Adaboost

In this chapter the origin of boosting is described, which shows early emergence of the method under PAC ("Probably Approximately Correct") theory, and the appearance of the first boosting algorithms. The Adaboost methodology is then analyzed which includes understanding the methods underlying principles and processes followed by the description and analysis of the actual **Adaboost.M1** algorithm.

The derivation of Adaboost is then explained by first articulating the on-line allocation problem which gave rise to the corresponding solution **Hedge(β)** algorithm and which subsequently lead to the creation of the first Adaboost method. In this chapter the **Hedge**(β) method is analyzed from an algorithm and performance point of view whilst the **Original Adaboost** algorithm is compared to both the **Adaboost.M1** algorithm to reaffirm consistency as well as the **Hedge(β)** algorithm to assess the possible similarities and differences.

Although no formal derivation is provided for Adaboost, a mathematical justification was provided by Hastie et al. [9] several years after the first publication of the method which sought to firmly explain Adaboost using well understood statistical principles. What these authors where able to show is that the process of Adaboost was equivalent to building an additive model using a specific exponential loss function or criterion. In this chapter we will explore the concept of additive modelling, the expo-

nential loss function and criterion and lastly how these two well known principals are combined to give rise to Adaboost, more specifically its discrete case **Adaboot.M1**.

Lastly the chapter concludes with an analysis on the final error of the Adaboost prediction and attempts to find and prove the bounds on such an error. The bounds are then analyzed to determine the conditions under which they tighten or loosen in order to ultimately determine the theoretical performance of Adaboost.

2.1 The Origins of Boosting

The origin of the boosting technique can be traced back to an early application of PAC machine learning which was proposed in 1984 by Leslie Valiant. PAC was developed as a framework which allowed a learner to select the "optimal" function from a class of possible functions. This was accomplished by selecting such a function that would result in the lowest error. The field of boosting then emerged as a possible method of enhancing PAC and was articulated and framed by Kearns et al.[6] through the following question,

> "can a *weak* learning algorithm which performs slightly better than random guessing be boosted into an arbitrarily *strong* learning algorithm."

The formal definition of PAC *weak* and *strong* learning algorithms are given below and are taken from Freund et al. [5].

Definition 1 *A strong PAC-learning algorithm is an algorithm that given* $\varepsilon, \delta > 0$ *and access to random examples, outputs with probability* $1 - \delta$ *a hypothesis with error at most* ε.

Definition 2 *A weak PAC-learning algorithm is an algorithm that satisfies the same conditions as a strong PAC-learning algorithm but only for* $\varepsilon \geq \frac{1}{2} - \gamma$ *where* $\gamma > 0$ *is either a constant or decreases as* $1/p$ *where* p *is a polynomial in the relevant parameters*

In other words Definition 1 can be explained as, given some small non-negative value for ε and δ and the ability to draw random samples from some input space distribution, the PAC-learning algorithm will select the learning function from a class of possible functions with a $1 - \delta$ chance or high probability and which algorithm produces an error of at most ε, a measure of the selected learning function's accuracy. Definition 2 on the other hand imposes a condition on the accuracy of the selected learning function to be only slightly better than random guessing and hence the reference to the term *weak learner*. Note in both Definition 1 and Definition 2 it is implied that $\varepsilon < \frac{1}{2}$ as if this condition is not true we may as well flip a coin as the chosen learner.

The first boosting algorithm according to Freund et al. [7] was developed by Robert E. Schapire in 1989. A year later the first enhancements emerged from Yoav Freund with the *boost by majority* algorithm that according to Freund et al. [5] was

considerably more effective. The first application of the boosting technique then appeared in 1993 within the field of Optical Character Recognition ("OCR") i.e. converting scanned hand-written text into displayable and editable computer text. Thereafter many authors began to developed new boosting methods and algorithms with numerous practical applications.

2.2 Adaboost Methodology

This thesis will focus on the most popular booting algorithm in use today called **Adaboost** which was first developed and published by Freund et al.[5] in 1995. In Freund et al.[5] the first extension of PAC learning methods to boosting was established along with the formulation of the first basic Adaboost algorithms. The authors derived a method of optimally allocating weights to observations or strategies with the objective being to minimize some loss function over the distribution of weights. It was the extension of this basic idea which lead to the creation of the first Adaboost method.

Intuitively the Adaboost methodology makes sense when understanding its underlying concept and process. In essence the Adaboost method works by constructing a *strong* learner by iteratively applying a *weak* learner to re-weighted observations. It is in the re-weighting of the observations that lends to the power of the method. At each iteration the Adaboost method has the effect of placing relatively more weight on observations leading to incorrect predictions whilst minimizing or decreasing, on

a relative basis, the weights placed on observations leading to correct predictions. The effect of this dynamic weighting process is that those observations which lead to incorrect predictions are given more emphasis in the next iteration to be correctly predict by the weak learner. The series of predictors generated, which are essentially the individual weak learners trained on the same continually re-weighted dataset, at each iteration are then combined to create the final model. It is in *how* these individual predictors are combined which lends to the strength of the Adaboost method. Such a combination occurs in a way as to ensure those predictors with higher overall accuracy are given more influence or weighting in the overall or combined model than those with less accuracy.

An illustration is now shown as to how the principal above is applied to a simple classification problem. Consider a two class problem with dependent variable $Y \in \{-1, 1\}$ and a given set of samples/examples from a population i.e. training set $(\mathbf{x}_1, y_1), .., (\mathbf{x}_N, y_N)$ where each observation vector $\mathbf{x}_i$ vector belongs to some domain or instance space $\mathbf{X}$ consisting of p inputs such that $\mathbf{x}_i = \langle x_{1i}, x_{2i}, ..., x_{pi} \rangle'$, $i = 1, ..., N$. Next define a weak learner h satisfying the properties in Definition 2 such that

$$h : \mathbf{X} \to \{-1, 1\} \tag{2.1}$$

and let $w_i > 0$ be the weight applied to each observation in $\mathbf{X}$. The estimated error rate on the training set is then

$$\widehat{\epsilon} = \frac{1}{N} \sum_{i=1}^{N} I(y_i \neq h(\mathbf{x}_i)) \tag{2.2}$$

with the generalized error rate on all future predictions being

$$\epsilon = E_{XY} I(Y \neq h(\mathbf{X})) = P[Y \neq h(\mathbf{X})]. \tag{2.3}$$

For h to be defined as a weak leaner as per Definition 2, or classifier in this case, its error rate ϵ needs to be only slightly better than random guessing, i.e. where $\epsilon < \frac{1}{2}$. Aside from this requirement being intuitive in that it does not make sense to boost a weak learner that performs less than random guessing, in the case of a two class problem as we have here, if we did have a weak learner h that performed worse than random guessing $\epsilon > \frac{1}{2}$ one such strategy could be to simply replace h with its compliment h'. The requirement for weak learners to perform better than random guessing is an important one within the context of Adaboost and the justification for this criteria to hold true will be shown in Section.2.3. Continuing with our example, the Adaboost method then proceeds to fit the weak classifier h using weights w_i to the training set $(\mathbf{x}_i, y_i), i = 1, ..., N$ and updates the w_i's using the resultant classifier's training error. The weak classifier is once again fitted to the training set producing a second weak classifier h however on this occasion using the updated weights from the earlier iteration. This process continues for a number of pre-determined iterations, which are also known in literature as boosting rounds, and

are usually parametrized by $t = 1, ..., T$ or $m = 1, ..., M$. The result is a sequence of weak classifiers $h_t(\mathbf{x})$ trained on observations using weights $\mathbf{w}^t = \langle w_1^t, w_2^t, ..., w_N^t \rangle'$ calculated at each booting round $t = 1, ..., T$. The prediction generated from each weak classifier is then combined through a weighted majority vote to produce the final output or model,

$$h_f(\mathbf{x}) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})\right). \tag{2.4}$$

In equation (2.4) the $\alpha_1, \alpha_2, ..., \alpha_T$ are calculated by the Adaboost algorithm so as to weight the contribution of each individual weak classifier $h_t(\mathbf{x})$ in such a way that higher accuracy weak classifiers receive more influence than lower accuracy classifiers in determining the final prediction. The sign function used here is due to the fact that $h_f(\mathbf{x})$ can take on any real value between and outside of -1 and 1 and that $Y \in \{-1, 1\}$. It is at this juncture that it becomes evident as to how the name *Adaboost* was arrived at in that the method continuously adapts to the error rates of the individual weak learners - "Ada" is therefore short for "adaptive".

In the description of the Adaboost process given by Figure 2.1 one assumes that the weak learner, or classifier in this case, can be trained on the data using weights w_i^t however when this is not possible Freund et al.[7] suggests an alternate method using random (re)sampling. Using the weights w_i^t one simply samples with replacement from the training set at each boosting iteration according the distribution of $\mathbf{w}^t$. The classifier is then trained on the sample of un-weighted observations drawn according to $\mathbf{w}^t$ and the process continues as before. We have also assumed here that h maps

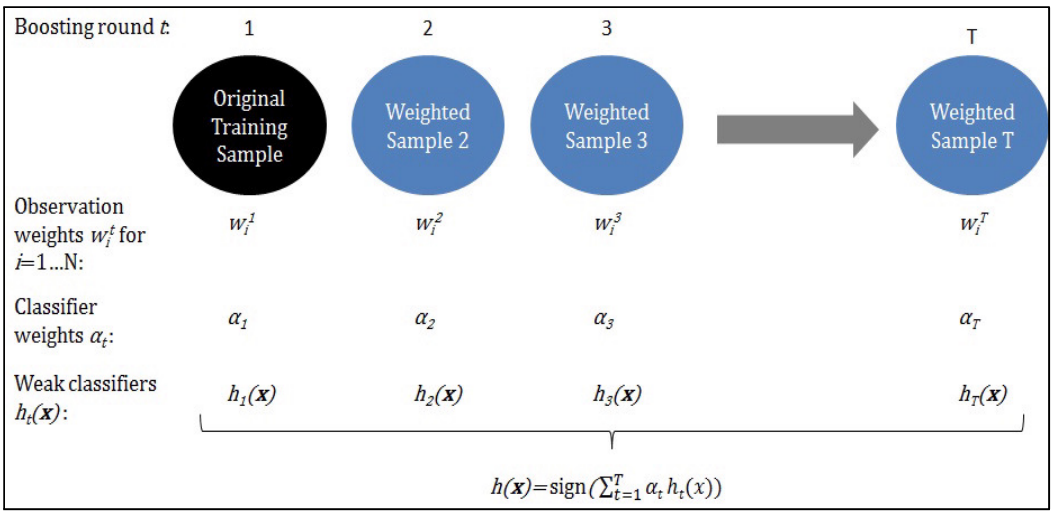


Figure 2.1: Visual descrpition of the Adaboost method expanding on the diagram given in Hastie et al. [4] (p 338). A series of weak -1/1 classifiers are generated and trained on iteratively reweighted versions of the training set to produce a final prediction.

the $\mathbf{x}_i$'s discretely onto the set $\{-1, 1\}$ however as shown in Schapire et al.[8], Adaboost can also be extended to handle weak hypothesis which output real-valued or *confidence rated* predictions. In this case h outputs a prediction $h \in \mathbb{R}$ whose sign for example dictates the predicted label for -1 or 1 and whose magnitude $|h_f(\mathbf{x})|$ gives a measure of "confidence" in the prediction. The use of such real-valued outputs become particularly useful in hybrid ensemble methods (see Kotsiantis et al. [15]) which we shall touch on in Chapter 5. It is also easy to see that such predictions can be normalised to produce class probability outputs (see Freund et al.[7]) and interestingly from this one can also begin to see a relationship between the weak learner h and class probabilities $P[y = 1]$. Such a relationship will be clarified in Section 2.5 within the context of the Adaboost algorithm. The Adaboost method described ear-

lier within the binary classification context can also be extended seamlessly to handle multi-classification problems however in these cases restrictions on the accuracy of the weak learner begin to apply which will be discussed in Section.2.3.

A question which arises is *why* use Adaboost as a method of optimization when there are numerous alternative boosting methods available. If one was to simply use the availability of literature on competing boosting methodologies as a gauge, Adaboost would selected without contest. Aside from this rather anecdotal evidence, the popularity of Adaboost has grown rapidly over the years when compared to other boosting methods attributable mainly to the following reasons:

- The algorithm is simple and easy to understand - even to someone without extensive statistical knowledge.
- The process is quick and easy to implement making its application to real-life problems appealing.
- Adaboost can be applied to both classification and regression type problems and therefore is more versatile and allows for practical flexibility.

More specifically, and which will be demonstrated through the course of this dissertation, the Adaboost method:

- Does not require prior knowledge of the accuracy of the weak learners unlike some of its competitors.

- Has appealing error bounds.
- Reduces both training error and test error and in some cases continues to reduce the test error long after the training error drops to zero even though theory suggests otherwise.
- In most cases does not succumb to the problem of over-fitting.
- Reduces both variance and bias where other ensemble methods such as *Bagging* reduces only variance.

2.3 Adaboost.M1 Algorithm

In this section we will introduce and analyze an algorithm for the Adaboost method called **Adaboost.M1** first described in Freund et al. [5] where the "**M**" stands for multi-class and "**1**" the first extension. As the name suggests the **Adaboost.M1** algorithm was created to cater for multi-class problems however for the purposes of describing the **Adaboost.M1** algorithm we will continue to assume $y_i \in \{-1, 1\}$ and therefore restrict the number of classes to $k = 2$, but will show how the algorithm can by adjusted to accommodate for situations where $k > 2$. **Adaboost.M1** has also been referred to as *"Discrete Adaboost"* by Hastie et al. [9], since the weak learner h returns a discrete class label, however the method is not limited by this and can also accommodate situations where the weak learner h outputs *real-valued* predic-

tions. More precisely when these predictions are in the form of class probabilities, Hastie et al. [9] showed that **Adaboost.M1** can be successfully adapted to handle real-valued outputs and named this extended method *"Real Adaboost"*. The authors in Hastie et al. [9] also showed that *Real Adaboost* in some instances outperformed **Adaboost.M1** due to additional information being captured at each boosting round**.** The **Adaboost.M1** algorithm shown here has been tailored to be more 'user-friendly' when compared to descriptions given by other authors. In doing so an attempt has been made to simplify and elaborate on the algorithms provided by Freund et al. [7] and Hastie et al.[4] (p 337-387). We will also show why there is an accuracy restriction placed on the weak learners in Adaboost and why such a restriction increases as the number of classes increase within the context of multi-classification.

Analyzing the **Adaboost.M1** algorithm set out in the following page, we see that after the weights in step 1 are initialized, we call the classifier for the first time creating h_1 in step 2a and calculate its corresponding weighted error rate $\widehat{\varepsilon_1}$ in step 2b. At step 2c the weight α_1 for the first classifier's contribution to the final prediction is calculated. More generally $\alpha_t \propto 1/\widehat{\varepsilon_t}$ i.e.α_t gets larger as $\widehat{\varepsilon_t}$ gets smaller and therefore step 2c has the effect of placing greater importance, by using a higher α_t weighting, on those classifiers h_t which have lower error rates and by implication higher accuracy, and less importance, through a lower α_t weighting, on those classifiers with higher error rates and by implication poorer accuracy. In addition when the training error rate of each classifier $\widehat{\epsilon_t}$ is much less than $\frac{1}{2}$ i.e. the weak classi-

Algorithm 1 Adaboost.M1 algorithm applied to a binary case

Algorithm **Adaboost.M1**
Given the training data $(y_1, \mathbf{x}_1), ..., (y_N, \mathbf{x}_N)$ where $\mathbf{x}_i \in \mathbf{X}$, $y_i = \{-1, 1\}$

1. Initialize the observation weights ${w_i}^1 = 1/N$, $i = 1, 2, ..., N$

2. For $t = 1$ to T:

   (a) Fit/train the weak classifier h_t on the training data using weights w_i^t (in the case of the classifier being unable to use these weights, resample with replacement from the training data according to the distribution $\mathbf{w}^t$)

   (b) Compute

$$\widehat{\varepsilon}_t = \frac{\sum_{i=1}^{N} {w_i}^t I(y_i \neq h_t(\mathbf{x}_i))}{\sum_{i=1}^{N} {w_i}^t}$$

   the weighted error rate of $h_t(\mathbf{x}_i)$

   (c) Calculate $\alpha_t = \ln((1 - \widehat{\varepsilon}_t)/\widehat{\varepsilon}_t)$

   (d) Update the observation weights $w_i^{t+1} = w_i^t \cdot \exp[\alpha_t \cdot I(y_i \neq h_t(\mathbf{x}_i))]$, for $i = 1, 2, ..., N$

3. Output $h_f(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}_i)\right)$

fiers is significantly better than random guessing, we get $\alpha_t \geq 0$ and this inequality becomes larger as $\widehat{\epsilon_t}$ decreases and therefore more weight is placed on those classifiers with better accuracy in the final prediction. In the next step 2d. the weights are then updated for each individual observation in the training set to produce the next set of observation weights w_i^{t+1} for $i = 1...N$. The equation in step 2d has the opposite effect to step 2c in that miss-classified observations have their weights increased by a factor of $exp(\alpha_t)$ where $exp(\alpha_t) \geq 1$ because $\alpha_t \geq 0$, while the weights of correctly classified observations are left unchanged. The effect of weighting observation in this manner is that in the next boosting round, h_{t+1} is then forced to train and correctly classify those observations that where incorrectly classified in the preceding boosting round t and thus illustrates the power of the Adaboost method of iteratively improving prediction accuracy. We also note here that if $\widehat{\epsilon_t} > \frac{1}{2}$ the algorithm has the effect of decreasing the observation weights which in this context is counterintuitive and therefore reinforces the requirement for base predictor accuracy which is better than random guessing. In such a case the **Adaboost.M1** algorithm is halted or the inverse classification applied however the latter being only applicable when faced with binary classification problems. Step 3 of the **Adaboost.M1** algorithm simply aggregates all the individual predictors h_t for $t = 1...T$ according to a weighted majority vote based on α_t weights to create the final prediction h_f. As noted earlier, in step 2c the α_t's assign more importance i.e. weight to higher accuracy classifiers which en-

sures that only the most accurate classifiers in the set of all classifiers account for the majority of the final prediction.

The application of **Adaboost.M1** to multi-class problems, where $k > 2$ and the weak learner h_t at each boosting round t outputs a discrete class label, is straightforward. If we assume, for example, the class labels are numeric $y \in Y = \{1, ..., k\}$ such that $h_t : \mathbf{X} \rightarrow Y$ the only resultant change in the **Adaboost.M1** algorithm occurs at step 3 i.e. in the creation of the final prediction h_f. Instead of h_f being the sign of a weighted summation of predictions outputting -1 or 1 as shown in step 3 and in equation 2.4, h_f is simply adapted to:

$$h_f = \arg\max_{y \in Y} \sum_{t:h_t(\mathbf{x})=y}^{T} \alpha_t \tag{2.5}$$

The final prediction is therefore the class y with the largest sum of α_t weights over the correct predictions. In other words, the final prediction h_f now outputs the class label y that maximizes the sum of weights of the weak learners predicting that label. Since α_t measures the accuracy of the weak learner h_t, the effect of equation (2.5) is that the final class label predicted is generated from the most accurate and correctly predicted weak learners.

With respect to the multi-class case we now analyze the drawback of **Adaboost.M1** being the restriction placed on the accuracy of the individual weak classifiers h_t. If during any boosting round whilst in the multi-class application of **Adaboost.M1** the weak classifier generates a prediction with accuracy less than $\frac{1}{2}$ i.e.

$\widehat{\varepsilon_t} > \frac{1}{2}$, the algorithm is halted and the final prediction is given using the weak classifiers generated up until that point. The reason for this can be explained as follows. In the binary case $(k = 2)$, a random guess will be correct with probability $\frac{1}{2}$ and the **Adaboost.M1** algorithm can be adjusted 'on-the-fly' to cater for situations where h_t performs worse than random guessing for example by replacing h_t with its complement h'_t. However in the case where $k > 2$ the probability of a correct random prediction is only $1/k < 1/2$, in which case there is no straightforward adjustment to the algorithm as before and therefore our requirement for the accuracy of the weak learner to perform better than random guessing is significantly stronger. Assuming the requirements of the accuracy of h_t are met we encounter yet another problem in respect of the multi-class application of the **Adaboost.M1** algorithm. We shall show the practical implication of this problem using an informal example as given in Freund et al. [5]. Consider a learning problem where $k = 3$ and $Y = \{0, 1, 2\}$. Suppose that it is "easy" to predict whether the class label is 2 but "hard" to predict whether the label is 0 or 1. Then a weak learner which predicts correctly whenever the label is 2 and otherwise guesses randomly between 0 and 1 is guaranteed to be correct at least half the time, significantly beating the $1/3$ accuracy achieved by guessing entirely at random. In this case the multi-class **Adaboost.M1** algorithm can still be boosted into arbitrary accuracy however becomes infeasible since we have assumed it is hard to distinguish between 0 and 1 labelled instances. The authors Freund et al. [5] along with Schapire et al. [8] developed further boosting algorithms to overcome this prob-

lem by essentially creating a set of binary problems for each class label which will be discussed in Section 4.4 and Section 4.5.

2.4 "Derivation" of Adaboost

The Adaboost method was first put forward and described by Freund et al. [5] and was originally formulated within the context of the *on-line allocation problem*, which will be described in this section. It is interesting to note that Freund et al. [5] provided no formal derivation of Adaboost and instead showed how this then "new" boosting method arose as an extension of the solution to the on-line allocation problem, however developed with a different goal in mind of "strengthening" weak learners as apposed to selecting optimal strategies. This section will cover the on-line allocation problem, show and analyze the **Hedge**(β) algorithm developed by Freund et al. [5] to solve such a problem and finally describe the original Adaboost algorithm. Since Freund et al. [5] also articulates an extension to their original Adaboost algorithm, i.e. Adaboost.M1, we will also seek to compare their original Adaboost algorithm to the **Adaboost.M1** algorithm shown earlier in Section 2.3. Lastly the solution for the on-line allocation problem **Hedge**(β) will be compared to the original Adaboost algorithm to highlight any similarities and differences, if any, between the two algorithms.

2.4.1 On-line allocation problem

The on-line allocation problem is best understood using a practical example taken from Freund et al.[5]. Assume you are a horse racing gambler and you have come into a losing streak. To try and change your luck you decide to let your fellow gamblers place wages on your behalf hoping their picks will be the winners. The problem you face is which gambler(s) do you choose (or *allocate*) to place bets on your behalf in order to maximize your winnings. Naturally you would choose the gambler who would win the most, unfortunately you do not know this ahead of time when placing the bets (*on-line*). Instead you attempt to select a combination of gamblers placing different size bets with each in order to produce a result as close to placing all your bets with that single expert gambler, if somehow you knew beforehand that he/she would win the most.

What we have described above is specific occurrence of the on-line allocation problem, more generally we can reduce the problem to the following statement: *how do we select the options or strategies from a selection of options or strategies which would maximize our chances of success?*. The authors Freund et al.[5] go on to formalize the on-line allocation model as follows. The allocation agent A has N options or strategies to choose from and we number these options or strategies using integers $1, ..., N$. At each timestep $t = 1, 2, ..., T$ the allocator A decides on a distribution $\mathbf{p}^t = \langle p_1^t, p_2^t, ..., p_N^t \rangle$ over the strategies, that is each component $p_i^t \geq 0$ is the weight allocated to strategy i at time step t, such that $\sum_{i=1}^{N} p_i^t = 1$. Each strategy i then

suffers some loss ℓ_i^t which is determined by the "environment". In the horse racing example this would be the size of the loss, if any, incurred for a particular horse race event t by placing your bet with gambler i. The loss suffered by A can then be calculated as $\sum_{i=1}^{N} p_i^t \ell_i^t = \mathbf{p}^t \cdot \ell^t$, i.e. the combined or average loss of strategies with respect to A's chosen allocation rule at time t (note ℓ^t denotes a loss vector over all N strategies at time t i.e. $\ell^t = \langle \ell_1^t, \ell_2^t, ..., \ell_N^t \rangle'$, $t = 1, ..., T$). Once again if we revert back to our horse racing example this would be your combined loss for a race t having placed bets with each gambler i. The loss function as described here is known as the *mixture loss* as it is a "mixture" of losses over all possible strategies or options. Using the converse argument to describe our goal of maximizing our winnings, the goal of A is to produce a loss as close to the loss suffered by the best strategy. In other words A must minimize its total loss relative to the loss suffered by the best strategy across all time steps t. That is A attempts to minimize its net loss

$$L_A - \min_i L_i \tag{2.6}$$

where,

$$L_A = \sum_{t=1}^{T} \mathbf{p}^t \cdot \ell^t \tag{2.7}$$

is the total/cumulative loss suffered by algorithm A over T time steps or trials, and

$$L_i = \sum_{t=1}^{T} \ell_i^t \tag{2.8}$$

is strategy $i's$ cumulative loss over all T time steps. Now that we have sufficiently articulated the on-line allocation problem we are in a position to analyze one such possible solution called Hedge(β).

2.4.2 Hedge β algorithm

The authors Freund et al. [5] go on to formulate an algorithm that solves the on-line allocation problem using the principals of minimization as applied to expression (2.6). They named this algorithm **Hedge(**β) presumably because of its analogy to, "hedging" one's bets. The **Hedge(**β) algorithm and its corresponding analysis are in effect direct generalizations of Littlestone and Warmuth's weighted majority algorithm as described in Littlestone et al. [10].

Algorithm 2 The on-line allocation algorithm

Algorithm **Hedge(**β)
Select parameter $\beta \in [0, 1]$
Initialize the first weight vector $\mathbf{w}^1 \in [0, 1]^N$ with $\sum_{i=1}^{N} w_i^1 = 1$
For $t = 1$ to T

1. Choose allocation

$$\mathbf{p}^t = \left(\frac{w_1^t}{\sum_{i=1}^{N} w_i^t}, \frac{w_2^t}{\sum_{i=1}^{N} w_i^t}, ..., \frac{w_N^t}{\sum_{i=1}^{N} w_i^t} \right)' = \frac{\mathbf{w}^t}{\sum_{i=1}^{N} w_i^t}$$

2. Receive loss vector $\ell^t \in [0, 1]^N$ from the environment

3. Suffer loss $\mathbf{p}^t \cdot \ell^t$

4. Set/update the new weight vector to be

$$w_i^{t+1} = w_i^t \beta^{\ell_i^t}$$

Analyzing the **Hedge(**β) algorithm we see that a vector of non-negative weights denoted $\mathbf{w}^t = \langle w_1^t, w_2^t, ..., w_N^t \rangle', t = 1, ..., T$ are maintained and sequentially updated throughout the time step iterations. The only restriction placed on the weight vector is that the initial weight vector $\mathbf{w}^1$ must be non-negative and sum to one, so that $\sum_{i=1}^{N} w_i^1 = 1$. Beside this rather loose restriction, the initial weight vector may be arbitrary and set to $\mathbf{w}^1 = \langle 1/N, 1/N, ..., 1/N \rangle'$, where $w_i^1 = 1/N$ for $i = 1, .., N$ without loss of generality. The initialization of the first weight vector $\mathbf{w}^1$ can be viewed as a "priori" over the set of strategies however without the need of being highly accurate. If we had some initial knowledge about which strategies to favour we could just as easily modify the initial weight vector to accommodate such knowledge. Note that the weights on future trials need not sum to one as they are continually normalized in step 1.

As the name indicates **Hedge(**β) is parametrized by the parameter β. The choice of β, as we will later see, has an impact on the bounds of the total cumulative loss of the **Hedge(**β) algorithm over all time steps T called $L_{\mathbf{Hedge}(\beta)}$, as defined generally in equation (2.7), and therefore the overall effectiveness of the algorithm. Running through the steps of the **Hedge(**β) algorithm we see that in Step 1 the choice of strategies are weighted by the normalized current weight vector. More specifically **Hedge(**β) chooses the strategies according to the distribution vector

$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^{N} w_i^t}. \tag{2.9}$$

In Step 2 the loss vector ℓ^t is received. Later we will see that for a specific choice of ℓ, **Hedge**(β) appears to give rise to the **Adaboost.M1** algorithm. Step 3 simply outputs the loss incurred over all the strategies at each time step t which ultimately are aggregated to create the total loss $L_{\mathbf{Hedge}(\beta)}$ over all available time steps. Note Step 3 is analogous to calculating an expected loss function over the probability distribution $\mathbf{p}^t$. Using the loss vector in Step 2, Step 4 updates the weight vector $\mathbf{w}^t$ using the multiplicative rule

$$w_i^{t+1} = w_i^t \beta^{\ell_i^t}. \tag{2.10}$$

As a convention note in equation (2.10), β is raised to the power of ℓ_i^t whereas w_i^t relates to the weight of observation i at time step t. Analyzing equation (2.10) we see that for strategies producing *greater losses* i.e. higher values of ℓ_i^t, the next set of successive weights w_i^{t+1} are *decreased* relative to the current weights w_i^t since $\ell_i^t \in [0, 1]$ and $\beta \in [0, 1]$. This update rule intuitively makes sense as one would generally stop or decrease the emphasis placed on choosing strategies which produce high losses. We can also understand the overall effectiveness of the **Hedge**(β) in terms of the algorithm's total loss $L_{\mathbf{Hedge}(\beta)}$. The importance of understanding the bounds on the error of **Hedge(**β) i.e. $L_{\mathbf{Hedge}(\beta)}$, will become evident as we seek to establish error bounds on the Adaboost method in Section 2.6. Since at the end of the algorithm a final optimal weight vector $\mathbf{w}^{T+1}$, which by construction is a function of all the previous weight vectors, is produced, we can use the same analysis given in Littlestone et al. [10] and apply it to find bounds on $\sum_{i=1}^{N} w_i^{T+1}$ which will then lead

to bounds on $L_{\mathbf{Hedge}(\beta)}$. We start by proposing the following lemma by expanding on the summarized proof as given in Freund et al.[5].

Lemma 1 *For any sequence of loss vectors* $\ell^1, ..., \ell^T$

$$\ln\left(\sum_{i=1}^{N} w_i^{T+1}\right) \leq -(1-\beta)L_{\mathbf{Hedge}(\beta)} \tag{2.11}$$

Proof. We can show using the convexity argument

$$\alpha^r \leq 1-(1-\alpha)r \text{ as given in Freund et al.[5]} \tag{2.12}$$

for $\alpha \geq 0$ and $r \in [0,1]$ that,

$$\begin{aligned}
\sum_{i=1}^{N} w_i^{t+1} &= \sum_{i=1}^{N} w_i^t \beta^{\ell_i^t}, \text{ from (2.10)} \\
&\leq \sum_{i=1}^{N} w_i^t (1-(1-\beta)\ell_i^t), \text{ using the convexity argument (2.12) applied to each } \beta^{\ell_i^t} \text{ term} \\
&= \left(\sum_{i=1}^{N} w_i^t\right)\left(1-(1-\beta)\frac{\sum_{i=1}^{N} w_i^t \ell_i^t}{\left(\sum_{i=1}^{N} w_i^t\right)}\right) \\
&= \left(\sum_{i=1}^{N} w_i^t\right)\left(1-(1-\beta)\sum_{i=1}^{N}\left(\frac{w_i^t}{\left(\sum_{i=1}^{N} w_i^t\right)}\ell_i^t\right)\right) \\
&= \left(\sum_{i=1}^{N} w_i^t\right)\left(1-(1-\beta)\mathbf{p}^t \cdot \ell^t\right), \text{ using the definition in equation (2.9).}
\end{aligned}$$

Applying repeatedly for $t = 1, ..., T$ yields

For $t = 1$:

$$\begin{aligned}\sum_{i=1}^{N} w_i^2 &\leq \left(\sum_{i=1}^{N} w_i^1\right)\left(1-(1-\beta)\mathbf{p}^1\cdot\ell^1\right)\\ &= \left(1-(1-\beta)\mathbf{p}^1\cdot\ell^1\right), \text{since}\sum_{i=1}^{N} w_i^1 = 1 \qquad (2.13)\end{aligned}$$

For $t = 2$:

$$\begin{aligned}\sum_{i=1}^{N} w_i^3 &\leq \left(\sum_{i=1}^{N} w_i^2\right)\left(1-(1-\beta)\mathbf{p}^2\cdot\ell^2\right)\\ &\leq \left(1-(1-\beta)\mathbf{p}^1\cdot\ell^1\right)\times\left(1-(1-\beta)\mathbf{p}^2\cdot\ell^2\right), \text{using (2.13)}\\ &= \prod_{t=1}^{2}\left(1-(1-\beta)\mathbf{p}^t\cdot\ell^t\right)\end{aligned}$$

Therefore $t = T$:

$$\sum_{i=1}^{N} w_i^{T+1} \leq \prod_{t=1}^{T}\left(1-(1-\beta)\mathbf{p}^t\cdot\ell^t\right).$$

Using the exponential identity inequality $1+x \leq e^x$ where $x = -(1-\beta)\mathbf{p}^t\cdot\ell^t$ we get

$$\begin{aligned}\sum_{i=1}^{N} w_i^{T+1} &\leq \prod_{t=1}^{T}\left(1-(1-\beta)\mathbf{p}^t\cdot\ell^t\right)\\ &\leq \prod_{t=1}^{T}\exp(-(1-\beta)\mathbf{p}^t\cdot\ell^t)\\ &= \exp\left(-(1-\beta)\sum_{t=1}^{T}\mathbf{p}^t\cdot\ell^t\right). \qquad (2.14)\end{aligned}$$

Recalling that $\sum_{t=1}^{T}\mathbf{p}^t\cdot\ell^t = L_{\mathbf{Hedge}(\beta)}$ is the total loss incurred for **Hedge(**β) and after taking the natural logs on either side of (2.14) we get

$$\ln\left(\sum_{i=1}^{N} w_i^{T+1}\right) \leq -(1-\beta)L_{\mathbf{Hedge}(\boldsymbol{\beta})}.$$

■

Rearranging Lemma 2.11 we can produce the upper bound on $L_{\mathbf{Hedge}(\beta)}$

$$L_{\mathbf{Hedge}(\beta)} \leq \frac{-\ln\left(\sum_{i=1}^{N} w_i^{T+1}\right)}{(1-\beta)}. \tag{2.15}$$

We see from the inequality (2.15) that the upper error limit of **Hedge(**β) is a function of the number of strategies N which intuitively makes sense as the more choices we have the harder it becomes to limit our losses. Note the w_i^t's will always be less than one due to the scaling effect in step 4 of the **Hedge(**β) algorithm and therefore the numerator in the right-hand-side of inequality (2.15) will always be positive. The effect of the choice of β on the error bounds is shown later. We note using equation (2.10) we can express the final weights as a function of the previous weights. *Please note the exponent convention with regards to* w_i *and* β *below.*

$$w_i^{T+1} = w_i^1 \prod_{t=1}^{T} \beta^{\ell_i^t} = w_i^1 \beta^{\left(\sum_{t=1}^{T} \ell_i^t\right)} = w_i^1 \beta^{L_i} \tag{2.16}$$

with the last equation in (2.16) following from the definition given in equation (2.8). The combination of inequality (2.15) and equation (2.16) gives rise to the $L_{\mathbf{Hedge}(\beta)}$ or error bound theorem in Freund et al.[5].

Theorem 2 *For any sequence of loss vectors* $\ell^1, ..., \ell^T$ *we have*

$$L_{\mathbf{Hedge}(\beta)} \leq \frac{-\ln\left(w_i^1\right) - L_i \ln \beta}{1-\beta} \tag{2.17}$$

Proof. The proof follows naturally by replacing w_i^{T+1} in inequality (2.15) with equation (2.16) and noting that $-\ln\left(\sum_{i=1}^{N} w_i^1\right) \leq -\ln\left(w_i^1\right)$ since $\sum_{i=1}^{N} w_i^1 = 1$

$$\begin{aligned} L_{\mathbf{Hedge}(\beta)} &\leq \frac{-\ln\left(\sum_{i=1}^{N} w_i^{T+1}\right)}{(1-\beta)} = \frac{-\ln\left(\sum_{i=1}^{N} w_i^1 \beta^{L_i}\right)}{(1-\beta)} \\ &\leq \frac{-\ln\left(w_i^1 \beta^{L_i}\right)}{(1-\beta)} \\ &= \frac{-\ln(w_i^1) - L_i \ln \beta}{(1-\beta)} \end{aligned}$$

■

Assuming arbitrarily that strategy i is the best, Theorem 2.17 tells us that **Hedge(**β) does not perform "too much worse" than the best strategy i since the numerator in inequality (2.17) is defined in terms of the loss incurred by the best strategy L_i where w_i^1 and β are known values or constants. Therefore **Hedge(**β) accomplishes the criteria required to solve the on-line allocation problem set out in expression (2.6). We are now in a position to understand the importance of β in that Theorem 2.17 and Figure 2.2 tells us that for large values of β the upper bound of $L_{\mathbf{Hedge}(\beta)}$ tightens and approaches the loss incurred on the optimal strategy i. The trade-off of large β values however is that the choice of good strategies becomes harder to distinguish as higher values of β do not penalize losing strategies as severely as smaller values of β and therefore either more time steps or a better priori for the initial weights are required. More generally the bound (2.17) shows the difference in loss between **Hedge(**β) and the best strategy i is dependent on the choice of β and on the initial weight w_i^1 of the best strategy i. Interestingly when we set the initial weights $w_i^1 = 1/N$ the bound

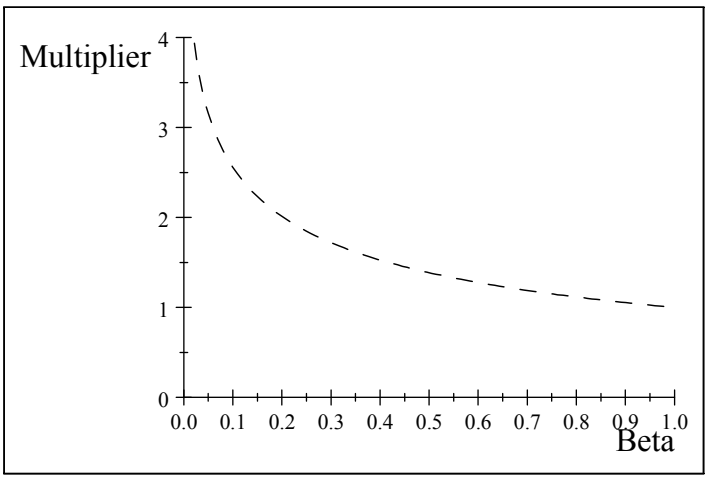


Figure 2.2: Shows the relationship between β and the multiplier $\frac{-\ln(\beta)}{1-\beta}$ of the loss of the best strategy L_i which constitutes a component of the upper bound of $L_{\mathbf{Hedge}(\beta)}$

becomes

$$L_{\mathbf{Hedge}(\boldsymbol{\beta})} \leq \frac{-\ln(1/N) - L_i \ln \beta}{(1-\beta)} = \frac{\ln N - L_i \ln \beta}{(1-\beta)}$$

and hence is dependent on the logarithm of N which is reasonable even for very large number of strategies.

2.4.3 Original Adaboost algorithm

It now becomes quite evident as to the possible extension of **Hedge(**β) to Adaboost in that **Hedge(**β) seeks to optimally select the best strategies by sequentially reweighting the importance of higher accuracy strategies while Adaboost, in its simplest form, seeks to enhance prediction by weighting a series of individual predictors based on their *accuracy* whilst sequentially reweighting observations based on their impor-

tance in terms of *inaccuracy*. It is this connection that formed the basis the authors Freund et al. [5] used to formulate the first Adaboost algorithm,

> "...there is an obvious similarity between the algorithms **Hedge(**β) and **Adaboost** [what we have called the **Original Adaboost**]. This similarity reflects a surprising "dual" relationship between the on-line allocation model and the problem of boosting. Put another way there is a direct mapping or reduction of the boosting problem to the on-line allocation problem"

In Section 2.4.4 we shall also explore the differences between these two algorithms in further detail.

We shall now describe the original Adaboost algorithm as given in Freund et al. [5]. The authors first articulate the Adaboost algorithm for the binary case however with the class labels defined as $Y \in \{0, 1\}$ instead of $Y \in \{-1, 1\}$. In addition the weak learner is not a discrete mapping as before but a real function mapping onto the range of the new class labels such that $h : \mathbf{X} \rightarrow [0, 1]$.

We shall now analyze the **Original Adaboost** algorithm by comparing it against **Adaboost.M1** noting the following similarities and differences. *As a convention note in Step 2(e) of the Original Adaboost algorithm,* β *is raised to the power of* $1 - |h_t(x_i) - y_i|$ *whereas* w_i^t *relates to the weight of observation* i *at time step* t:

Similarities

- The error $\widehat{\varepsilon_t}$ in both algorithms has the effect of calculating the "average" training error of the classifier at a particular iteration h_t with respect to the observation weights at each boosting round t.

Algorithm 3 The original Adaboost algorithm applied to a binary case

Algorithm **Original Adaboost**

Given the training data $(y_{1,}\mathbf{x}_1), (y_{2,}\mathbf{x}_2), ..., (y_{N,}\mathbf{x}_N)$ where $\mathbf{x}_i \in X$, $y_i = \{0, 1\}$

1. Initialize the observation weights w_i^1 for $i = 1, 2, .., N$ such that $\mathbf{w}^1 \sim D$ (some arbitary distribution)

2. For $t = 1$ to T :

   (a) Generate the weight distribution

   $$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^{N} w_i^t}$$

   (b) Fit/train the weak learner h_t on the training data using the weight distribution $\mathbf{p}^t$

   (c) Calculate the estimated weighted error of h_t

   $$\widehat{\varepsilon_t} = \sum_{i=1}^{N} p_i^t \left| h_t(\mathbf{x}_i) - y_i \right|$$

   (d) Calculate $\beta_t = \widehat{\varepsilon_t}/(1 - \widehat{\varepsilon_t})$

   (e) Update the observation weights

   $$w_i^{t+1} = w_i^t \beta_t^{1-|h_t(\mathbf{x}_i)-y_i|}$$

3. Output the final prediction

$$h_f = \begin{matrix} 1, \text{ if } \sum_{t=1}^{T}(\ln(1/\beta_t)h_t(\mathbf{x})) \geq \frac{1}{2}\sum_{t=1}^{T}(\ln(1/\beta_t) \\ 0, \text{ otherwise} \end{matrix}$$

- The weight update steps of both algorithms accomplish the same effect by increasing the weights of incorrectly predicted observations relative to correctly predicted observations (see Figure 2.3 - all beta's are monotone increasing. Note *Weight Update* refers to $\beta_t^{1-|h_t(x_i)-y_i|}$ and *Observation Error* $|h_t(x_i) - y_i|$).

- The classifier weights of $\ln(1/\beta_t)$ in the **Original Adaboost** algorithm has the same effect of increasing the weights of higher accuracy classifiers h_t in the final prediction as $\beta_t \propto \widehat{\varepsilon}_t$ and therefore $\ln(1/\beta_t) \propto 1/\widehat{\varepsilon}_t$.

- The construction of the final hypotheses although different for the **Original Adaboost** algorithm accomplishes the same objective by increasing the weights of those weak learners with greater accuracy and correspondingly decreasing the weights of low accuracy weak learners. Further to this the **Original Adaboost** algorithm has the effect of outputting the class label 1 if most of the weak learners outputs values greater than $\frac{1}{2}$ and 0 when less than $\frac{1}{2}$ which is analogous to the sign function used in **Adaboost.M1** when $Y \in \{-1, 1\}$.

Differences

- Step 1 in the **Original Adaboost** algorithm allows the initial weight vector to be any distribution however by setting $\mathbf{w}^1$ to $w_i^1 = 1/N$ for $i = 1, .., N$ we can get equality between the two algorithms.

- In **Adaboost.M1** $\alpha_t \propto 1/\widehat{\varepsilon}_t$ whilst $\beta_t \propto \widehat{\varepsilon}_t$ for **Original Adaboost** however this difference is nullified in Step 3 of the **Original Adaboost** algorithm by taking the inverse natural log of β_t.

- Although as mentioned above the weight update steps accomplishes the same objective it differs for both algorithms - **Adaboost.M1** *increases* the weights of *incorrectly* predicted observations and leaves the weights unchanged for correctly predicted observation whilst **Original Adaboost** decreases all the weights but *decreases* the weights of *correctly* predicted observations by a greater margin. (see Figure 2.3 - observations with low error (good predictions) have their weights scaled/decreased more than observations with high error (bad predictions)).

- The construction of the final hypotheses differ however as mentioned above, the final prediction in the **Original Adaboost** algorithm is analogous to using the sign function in the final prediction of **Adaboost.M1** for the binary case. This difference makes sense given the h_t's are real valued functions and the class labels are $0/1$ in the **Original Adaboost** algorithm as apposed to discrete classifiers with class labels $-1/1$ in the **Adaboost.M1** algorithm for the binary case.

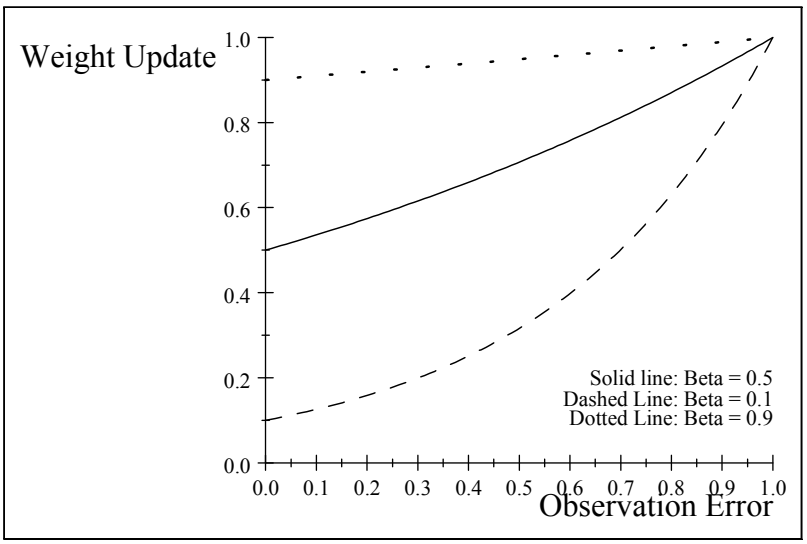


Figure 2.3: Shows the relationship between the weight update and observation error as defined in the **Original Adaboost** algorithm for β = 0.1, 0.5 and 0.9

We can conclude from the above analysis that although the **Original Adaboost** algorithm differs from **Adaboost.M1** algorithm in form, in substance the two algorithms are virtually alike and accomplish the same objective.

2.4.4 Hedge β vs. the Original Adaboost algorithm

Although no formal derivation of the Adaboost method was ever provider the paper, which outlines the first Adaboost algorithm Freund et al.[5], used principals contained in the **Hedge(β)** algorithm to assist in the formulation of the **Original Adaboost** algorithm. In addition although the two algorithms make use of similar principals they are fundamentally different in their construction and output with such differences best articulated using extracts taken from Freund et al.[5].

Difference 1 The implied extension or reduction of **Hedge(β)** to the **Original Adaboost** is reversed in that,

"in such a reduction [from **Hedge(β)** to the **Original Adaboost],** one might naturally expect a correspondence relating the strategies to the weak hypotheses and trials (and associated loss vectors) to examples in the training set. However the reduction we have used is reversed: the "strategies" correspond to the examples, and the trials are associated with the weak hypotheses."

Difference 2 The definition of loss is reversed in that,

"in **Hedge(β)** the loss ℓ_i^t is small if the i^{th} strategy suggests a *good* action on the t^{th} trial while in **Original Adaboost** the "loss" $\ell_i^t = 1 - |h_t(\mathbf{x}_i) - y_i|$ appearing in the weight-update rule (Step 2e.) is small if the t^{th} hypothesis suggests a *bad* prediction on the i^{th} example."

The above quotes make sense as the goal in **Hedge(β)**, is to *decrease* the weight of those strategies which results in greater *losses*, whilst the goal in the Adaboost method and more specifically the **Original Adaboost** is to *increase* the weights of those observations which results in *incorrect* predictions in order to "force" the weak learner to "concentrate" on correctly predicting those "hard" observation in the next boosting round.

The main technical difference between the two algorithms is related to the parameter β which is no longer fixed ahead of time as in the case of **Hedge(β)** but

rather changes at each time step according to the error. The two benefits of this is that we obtain a significantly superior error bounds on the individual weak learners which contributes to a lower **final** boosting error ϵ_f and that we do not require prior knowledge of the accuracy of the weak learners as β adapts dynamically to the error rates of the weak learners at each boosting round.

2.5 The mathematical justification of Adaboost

In 1998, almost a decade after the first papers where published on the general concept of boosting (see Kearns et al. [6]) Jerome Friedman, Trevor Hastie and Robert Tibshirani put forward a paper (see Hastie et al. [9]) which sought to provide the statistical community with the first theoretical framework explaining the Adaboost methodology, which up until that point had been regarded as somewhat of a phenomenon in terms of its effectiveness. In essence they where able to successfully explain the mystery surrounding Adaboost using the well known statistical principal of additive modelling and maximum likelihood making use of a specific exponential loss function and criterion. More specifically they where able to show how **Adaboost.M1** is equivalent to forward stagewise additive modelling minimizing an exponential loss function as well as additive logistic regression minimizing the exponential criterion.

In this section we will start by understanding the basic construct of additive models and how it can be extended to handle more complex model structures. We shall also examine possible methods to construct additive models such as *backfit-*

ting or using the *forward stagewise process.* Next a description and analysis of the exponential loss function and criterion is provided and lastly we show that when these processes and functions are combined they can be proved, on a step-by-step basis, to being equivalent to Adaboost**.** In other words we will show in detail how **Adaboost.M1** is *equivalent to forward stagewise additive modelling using an exponential loss function* as well as *additive logistic regression using the exponential criterion*.

2.5.1 Additive Regression Models

We start our analysis by understanding the basic principles of additive modeling using one of its most simplest forms, the additive regression model. In this case the response variable Y is quantitative and correspondingly so is the vector of predictor variables $\mathbf{X} = \langle X_{1,}X_{2,...,}X_p\rangle'$, where p denotes the number of individual predictor variables. The goal of additive regression is to therefore model $E(Y|X_1, X_2, .., X_p) = F(\mathbf{X})$ such that

$$F(\mathbf{X}) = \alpha + f_1(X_1) + f_2(X_2) + ... + f_p(X_p) \tag{2.18}$$

where the f_j's are separate functions fitted using each predictor variable X_i and α is a constant usually estimated using the arithmetic average of the y_i's. Many authors have suggested different ways to generate the f_j's, Hastie et al. [4] (p 295-336) suggests the use of unspecified smooth ("non-parametric") functions whilst Hastie et al. [9] makes use of the modular backfitting algorithm as given in Buja et al. [11] to

achieve stability or convergence. We shall now describe the *backfitting* algorithm as given in Hastie et al.[9] due to its simplicity and ease of implementation, as well as for its suitable practical application.

Algorithm 4 Backfitting algorithm for additive models

Algorithm **Additive Backfitting**

1. Initialize $\widehat{\alpha} = \frac{1}{N}\sum_{i=1}^{N} y_i$, and use any method to fit $\widehat{f}_j$ or set $\widehat{f}_j \equiv 0$, for all $j = 1$ to p

2. Repeat until the $\widehat{f}_j$'s stabilize or converge i.e. the sucessive iterations of the $\widehat{f}_j$'s differ less than some pre-specified tolerance

   (a) For $j = 1$ to p

      i. calculate the vector $y - \widehat{\alpha} - \sum_{k \neq j} \widehat{f}_k(x_k)$

      ii. Fit the updated responses in step i above to x_j such that $\widehat{f}_j$ becomes

$$\widehat{f}_j(x_j) \leftarrow E\left[y - \widehat{\alpha} - \sum_{k \neq j} \widehat{f}_k(x_k) \mid x_j\right] \tag{2.19}$$

We can see from the right-hand-side of the update rule in (2.19) of the **Additive Backfitting** algorithm that all the latest versions of the functions f_k are used to form the partial residuals and hence f_j is the model fitted which best explains the partial residuals. Buja et al.[11] then showed that under fairly general conditions the **Additive Backfitting** algorithm, as shown here, will converge to a closed form solution of the minimizer of $E(y - F(\mathbf{x}))^2$.

2.5.2 Extended Additive Models

Additive models described in the previous Section 2.5.1 can be extended to situations where we are not limited to p, f_j functions or elements with each a function of a single predictor variables X_j, but rather where we have as many f_j elements as we like with each a function of any subset of the predictor variables including possibly the full set $\mathbf{X}$. Within this context we can define a set of M functions $\{f_m(\mathbf{x})\}_1^M$ each consisting potentially of all the predictors. Using the general basis function expansion as given by Hastie et al.[4] (p 139-189)

$$f_m(\mathbf{x}) = \beta_m b(\mathbf{x}; \boldsymbol{\gamma}_m), \tag{2.20}$$

where the definition in equation (2.20) is given in terms of a "basis function" $b(\cdot)$ i.e. linear or non-linear combinations of the multivariate vector $\mathbf{x}$, and is characterized by the multiplier β_m and vector of parameter(s) $\boldsymbol{\gamma}_m$. Note, we have also dropped the use of α as in equation (2.18) which we have assumed is modelled in the f_m's in equation (2.20). The extended additive model then becomes

$$F_M(\mathbf{x}) = \sum_{m=1}^{M} f_m(\mathbf{x}) = \sum_{m=1}^{M} \beta_m b(\mathbf{x}; \boldsymbol{\gamma}_m). \tag{2.21}$$

As noted above, in this extended additive model the "basis functions" $b(\mathbf{x}; \boldsymbol{\gamma}_m) \in \mathbb{R}$ can take on numerous forms in addition to linear and quadratic combinations of the predictor variables and can be used for example to model single-layer neural networks (see Hastie et al. [4] (p 389-416)), signal processing (see Hastie et al. [4]

(p 139-189)) and multivariate adaptive regression splines (see Hastie et al. [4] (p 295-336)).

As the authors Hastie et al. [4] (p 337-387) note, typically these extended additive models are fitted by minimizing a loss function averaged or aggregated over the training data

$$\min_{\{\beta_m,\gamma_m\}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(\mathbf{x}_i;\boldsymbol{\gamma}_m)\right) \tag{2.22}$$

where the loss function L can be squared-error or a likelihood-based loss function. If a least-squares loss function is used, Hastie et al.[9] suggest one can solve for an optimal set of parameters $\{\beta_m, \boldsymbol{\gamma}_m\}_1^M$ through a generalized version of the **Additive Backfitting** algorithm with updates

$$\{\beta_m, \boldsymbol{\gamma}_m\} \leftarrow \arg\min_{\beta,\boldsymbol{\gamma}} E\left[y - \sum_{k\neq m}^{M} \beta_k b(\mathbf{x};\boldsymbol{\gamma}_k) - \beta b(\mathbf{x};\boldsymbol{\gamma})\right]^2 \tag{2.23}$$

for $m = 1, 2, ..., M$ until convergence is reached. Both expressions (2.22) and (2.23) are formidable minimization problems that require intensive numerical optimization techniques. However Hastie et al. [4] (p 337-387) suggests a simple alternative when it is feasible to rapidly solve the problem of (2.22) using a single basis function

$$\min_{\beta,\gamma} \sum_{i=1}^{N} L\left(y_i, \beta b(\mathbf{x};\boldsymbol{\gamma})\right).$$

Similarly Hastie et al.[9] suggests a "greedy" forward stepwise approach to solve problem (2.23) as

$$\{\beta_m, \boldsymbol{\gamma}_m\} \leftarrow \arg\min_{\beta,\boldsymbol{\gamma}} E\left[y - F_{m-1}(\mathbf{x}) - \beta b(\mathbf{x};\boldsymbol{\gamma})\right]^2 \tag{2.24}$$

for $m = 1, 2, ..., M$, where $F_{m-1}(\mathbf{x})$ is as defined in equation (2.21) and it's component parameters $\{\beta_k, \boldsymbol{\gamma}_k\}_1^{m-1}$ are fixed at their corresponding solution values at earlier iterations. At this point we begin to notice the similarities of additive modelling to boosting, and more specifically to Adaboost. By equating the multiplier and basis functions to the weak learners $\beta_m b(\mathbf{x}; \boldsymbol{\gamma}_m) = h_t(\mathbf{x})$ where $t = m$ the final boosting predictor $h_f(\mathbf{x})$ then mirrors the form of $F_M(\mathbf{x})$, where $T = M$. In addition when analyzing the backfitting updates of (2.23) and its greedy cousin given in expression (2.24), both methods only require an algorithm to fit a *single weak learner* to modified versions of the response data

$$y_m \leftarrow y - \sum_{k \neq m} f_k(\mathbf{x}) \tag{2.25}$$

for backfitting updates in (2.23) and

$$y_m \leftarrow y_{m-1} - F_{m-1}(\mathbf{x}) \tag{2.26}$$

for backfitting updates in (2.24). The effect of the update in (2.26) is that the previous solutions at earlier iterations have no effect on explanatory power on the new outputs y_m. This observation along with the update rules combined with the additive form of the final predictor is analogous to the Adaboost process which calls the same weak learner multiple time on different instances or weightings of the training data and then pools or adds together the individual weak learners to create a powerful "committee" prediction.

At this point we now introduce the use of decision tress or more specifically classification trees as a weak learner. In the context of extended additive modeling we can define the parameter $\boldsymbol{\gamma} = \langle s, j \rangle'$, where s is the split point and j the split variable. We shall describe why we have chosen to use decision trees as our base classifier in Section 3.1.

2.5.3 Forward Stagewise Additive Modelling

What we have described previously in the backfitting update of (2.24) is a particular aspect of forward stagewise additive modelling, albeit unknowingly. The purpose of forward stagewise additive modelling, as given in Hastie et al. [4] (p 337-387) is,

> "to produce an efficient solution to the minimization problem [given in (2.22)] by sequentially adding new basis functions to the expansion without adjusting the parameters and coefficients of those that have already been added."

We shall now show the forward stagewise additive modeling algorithm as given in Hastie et al. [4] (p 337-387).

At each iteration in step 2a we solve for the optimal parameter $\boldsymbol{\gamma}_m$, to find the basis function $b(\mathbf{x}; \boldsymbol{\gamma}_m)$, and corresponding coefficient β_m to add to the current expansion $f_{m-1}(\mathbf{x})$ in step 2b. Once the term $f_m(\mathbf{x})$ is produced the process is repeated eventually growing the final expansion to M basis terms with M corresponding β coefficients. Note the term "forward stagewise" is used because we start the algorithm with a single term then successively "move forward" (note reference to the word "forward" is because we cannot go back and change $f_m(\mathbf{x})'s$ after they have been added

Algorithm 5 Algorithm showing the forward stagewise additive modeling process

Algorithm **Forward Stagewise Additive Modeling**

1. Initialize $f_0(\mathbf{x}) = \mathbf{0}$

2. For $m = 1$ to M

   (a) Calculate

   $$(\beta_m, \boldsymbol{\gamma}_m) = \arg\min_{\beta,\boldsymbol{\gamma}} \sum_{i=1}^{N} L\left(y_i, f_{m-1} + \beta b(\mathbf{x}_i; \boldsymbol{\gamma})\right)$$

   (b) Set $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \beta_m b(\mathbf{x}; \boldsymbol{\gamma}_m)$

to the expansion) to grow it term by term "in stages". Such a process differs from traditional additive modelling which typically starts with a full set of M additive terms, albeit arbitrary, and enhances *all* M terms in an iterative fashion until convergence is reached.

2.5.4 Exponential Loss Criterion and Function

As with most of the expressions used previously we have defined a generic loss function L and used squared-error loss only for illustrative purposes to show how back-fitting using squared-error loss can create additive models. However we have yet to specify a specific loss function that would lead to the Adaboost method. In this section we define such a loss function using the exponential function

$$L(y, f(x)) = \exp(-y\, f(x)). \tag{2.27}$$

Similarly Hastie et al.[9] define the exponential criterion as

$$J(F) = E(e^{-yF(x)}). \tag{2.28}$$

The choice and use of these measures will become apparent when we show:

1. the equivalence of forward stagewise additive modeling using the exponential loss function (2.27) to **Adaboost.M1**

2. and the equivalence of additive logistic regression modeling using the exponential criterion (2.28) to **Adaboost.M1.**

specifically for the binary case where the response variable is $Y \in \{-1, 1\}$**.**

We shall now set some groundwork that will be required to prove the second point above in the next Section 2.5.5. The lemma below and accompanying proof is taken from Hastie et al.[9] however in doing so we have expanded on the proof which shows the function $F(x)$ that minimizes $J(F)$ is the closed form symmetric logistic transform of $P(y = 1|x)$.

Lemma 3 *$E(e^{-yF(x)})$ is minimized at*

$$F(x) = \frac{1}{2} \ln \frac{P(y = 1|x)}{P(y = -1|x)} \tag{2.29}$$

Hence

$$P(y = 1|x) = \frac{e^{F(x)}}{e^{-F(x)} + e^{F(x)}} \tag{2.30}$$

$$P(y=-1|x)=\frac{e^{-F(x)}}{e^{-F(x)}+e^{F(x)}} \tag{2.31}$$

Proof. While E entails expectations over the joint distribution of y and x, it is sufficient to minimize the criterion of $J(F)$ conditional on x as we have assumed x is the training predictor variable(s) which are given. Expanding the conditional criterion we get

$$\begin{aligned} E(e^{-yF(x)}|x) &= P(y=1|x)e^{-(1)\times F(x)}+P(y=-1|x)e^{-(-1)\times F(x)} \\ &= P(y=1|x)e^{-F(x)}+P(y=-1|x)e^{F(x)}. \end{aligned}$$

Minimizing with respect to $F(x)$ requires finding the partial derivative of the criterion

$$\frac{\partial E(e^{-yF(x)}|x)}{\partial F(x)}=-P(y=1|x)e^{-F(x)}+P(y=-1|x)e^{F(x)}$$

and setting the above result to zero. Therefore,

$$\begin{aligned} -P(y &= 1|x)e^{-F(x)}+P(y=-1|x)e^{F(x)}=0 \\ P(y &= -1|x)e^{F(x)}=P(y=1|x)e^{-F(x)} \\ e^{F(x)} &= \frac{P(y=1|x)e^{-F(x)}}{P(y=-1|x)} \\ e^{2F(x)} &= \frac{P(y=1|x)}{P(y=-1|x)} \text{...divided through by } e^{-F(x)} \\ 2F(x) &= \ln\left(\frac{P(y=1|x)}{P(y=-1|x)}\right) \\ F(x) &= \frac{1}{2}\ln\left(\frac{P(y=1|x)}{P(y=-1|x)}\right). \end{aligned} \tag{2.32}$$

Which proves equation (2.29) of the lemma. The corollaries are then found by rearranging equation (2.32)

$$\begin{aligned} P(y &= 1|x)e^{-F(x)} = P(y=-1|x)e^{F(x)} \\ &= (1-P(y=1|x))e^{F(x)} \text{...because } P(y=1|x) = 1-P(y=-1|x) \\ &= e^{F(x)} - e^{F(x)}P(y=1|x) \\ P(y &= 1|x)e^{-F(x)} + e^{F(x)}P(y=1|x) = e^{F(x)} \\ P(y &= 1|x)(e^{-F(x)} + e^{F(x)}) = e^{F(x)} \end{aligned}$$

$$P(y=1|x) = \frac{e^{F(x)}}{e^{-F(x)} + e^{F(x)}}.$$

Similarly using the same arguments and solving for $P(y=-1|x)$ we get

$$P(y=-1|x) = \frac{e^{-F(x)}}{e^{-F(x)} + e^{F(x)}}.$$

■

We notice here that the usual logistic transform does not have the factor $\frac{1}{2}$ as in equation (2.29) however by multiplying the numerator and denominator in equation (2.30) by $e^{F(x)}$ we get the usual logistic model

$$p(x) = \frac{e^{2F(x)}}{1+e^{2F(x)}}$$

and hence the two models i.e. the model given in equation (2.29) and the traditional logistic model are equivalent up to a factor of two.

Further to our choice of the exponential loss criterion, we can compare this measure to another well-known loss function, squared error loss. In Figure 2.4 the

quantity yF on the x-axis is a measure of classification accuracy, where $sign(F)$ is analogous to the final predictor $h_f(x)$ as per the **Adaboost.M1** algorithm and therefore F is not a probability mass function and can take on values less than 0 and greater than 1. Negative yF values indicate the degree of incorrect classification and conversely positive values indicate the degree of correct classification. We see from Figure 2.4 that exponential loss is monotone decreasing while squared error is not. The non-monotonicity of the squared-error limits the practical application of this loss function, in our binary case, as it begins to penalize classifications where yF is greater than 1, or in other words classifications which are "too correct". Therefore exponential loss appears to be the more appropriate choice due to its decreasing monotonicity as well as being able to be continuously differentiable (smooth) which is an important requirement in proving both points (1) and (2) above.

2.5.5 Equivalence of Adaboost to Additive Modelling using Exponential Loss or the Exponential Criterion

We are now in a position to show that Adaboost actually fits:

1. an additive forward stagewise model using the exponential loss function given in equation (2.27) and

2. an additive logistic regression model using Newton updates for minimizing the exponential criterion given in equation (2.28).

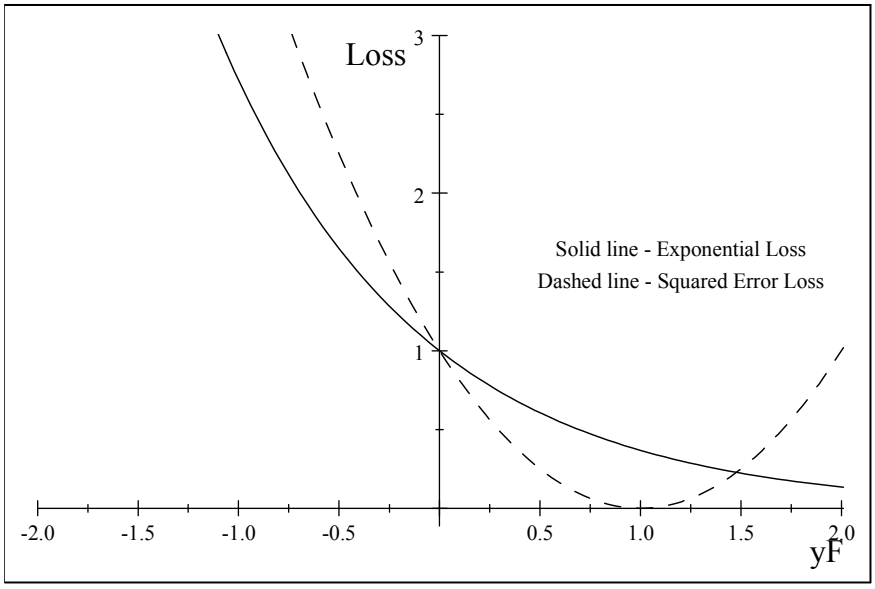


Figure 2.4: Graph, as per Hastie et al. [4] (p 347), showing how the exponential loss function continues to decrease as yF increases i.e. classification accuracy improves. Conversely with squared error loss the graph also shows how such a function begins to penalize classifications which are "too correct".

Once again we consider the binary case with $y \in \{-1, 1\}$ and $\mathbf{x}_i = \langle x_{1i}, x_{2i}, .., x_{pi} \rangle'$, $i = 1, ...N$ a vector of p inputs. We will first present a proposition and proof using elements from Hastie et al. [4] (p 337-387) to show point 1 above, followed by the proposition and proof from Hastie et al.[9] to show point 2 above.

Proposition 4 ***Adaboost.M1*** *is equivalent to* ***Forward Stagewise Additive Modeling*** *using the exponential loss function.*

Proof. Let the basis function be the weak classifier defined in (2.1) such that $h_m \in \{-1, 1\}$. We start by replacing the loss function in the minimization problem (2.22) with the exponential loss function of (2.27) and use the **Forward Stagewise Additive Modeling** algorithm to generate the $\{\beta_k, h_k\}_1^M$ to develop the additive model. Therefore we need to solve Step 2a of the **Forward Stagewise Additive Modeling** algorithm in the form

$$(\beta_m, h_m) = \arg\min_{\beta,h} \sum_{i=1}^{N} \exp[-y_i(f_{m-1}(\mathbf{x}_i) + \beta h(\mathbf{x}_i))] \tag{2.33}$$

for the weak classifier h_m and corresponding coefficient β_m to be added at each step. We can re-write (2.33) as

$$(\beta_m, h_m) = \arg\min_{\beta,h} \sum_{i=1}^{N} w_i^m \exp(-y_i \beta h(\mathbf{x}_i)) \tag{2.34}$$

where

$$w_i^m = \exp(-y_i f_{m-1}(\mathbf{x}_i)). \tag{2.35}$$

Since each w_i^m does not depend on β nor h it can be regarded as a weight that is applied to each observation at each time step m. We also see from equation (2.35)

that the weights change at each iteration since w_i^m is a function of f_{m-1}. Noting that

$$-y_i h(\mathbf{x}_i) = \begin{cases} 1, \text{ when } y_i \neq h(\mathbf{x}_i) \\ -1, \text{ when } y_i = h(\mathbf{x}_i) \end{cases}$$

we can now express the right-hand-side of the minimization argument in equation (2.34) as

$$e^{-\beta} \cdot \sum_{i:y_i=h(\mathbf{x}_i)} w_i^m + e^{\beta} \cdot \sum_{i:y_i \neq h(\mathbf{x}_i)} w_i^m \tag{2.36}$$

Note, reference here to e^{β} or e raised to any variable, number or parameter should be interpreted in the usual exponent way and does not denote a timestep such as in w_i^m. Expression (2.36) can then be written using indicator functions as

$$\left\{ e^{-\beta} \cdot \sum_{i=1}^{N} w_i^m - e^{-\beta} \cdot \sum_{i=1}^{N} w_i^m I(y_i \neq h(\mathbf{x}_i)) \right\} + e^{\beta} \cdot \sum_{i=1}^{N} w_i^m I(y_i \neq h(\mathbf{x}_i)) =$$
$$e^{-\beta} \cdot \sum_{i=1}^{N} w_i^m + (e^{\beta} - e^{-\beta}) \cdot \sum_{i=1}^{N} w_i^m I(y_i \neq h(\mathbf{x}_i)). \tag{2.37}$$

Therefore solving for h_m for some given $\beta > 0$, and because the left-hand-side term in expression (2.37) i.e. $e^{-\beta} \cdot \sum_{i=1}^{N} w_i^m$ and the coefficient of the right-hand-side term i.e. $(e^{\beta} - e^{-\beta})$ can be viewed as positive constants as they made up of known quantities,.the solution to equation (2.34) becomes

$$h_m = \arg\min_h \sum_{i=1}^{N} w_i^m I(y_i \neq h(\mathbf{x}_i)). \tag{2.38}$$

It is reaffirming to note that our reduction in equation (2.38) results in a solution h_m which minimizes the weighted error rate. We now need to solve for β by

differentiating expression (2.37) with respect to β and setting the result equal to zero

$$\frac{\partial}{\partial\beta}\left[e^{-\beta}\cdot\sum_{i=1}^{N}w_i^m+(e^{\beta}-e^{-\beta})\cdot\sum_{i=1}^{N}w_i^m I(y_i\neq h(\mathbf{x}_i))\right]=0$$

$$\text{Therefore, }\frac{\partial}{\partial\beta}\left[e^{-\beta}\cdot\sum_{i=1}^{N}w_i^m+(e^{\beta}-e^{-\beta})\cdot\sum_{i=1}^{N}w_i^m I(y_i\neq h(\mathbf{x}_i))\right]=$$

$$-e^{-\beta}\cdot\sum_{i=1}^{N}w_i^m+e^{\beta}\cdot\sum_{i=1}^{N}w_i^m I(y_i\neq h(\mathbf{x}_i))+e^{-\beta}\cdot\sum_{i=1}^{N}w_i^m I(y_i\neq h(\mathbf{x}_i))=$$

$$-e^{-\beta}+e^{\beta}\cdot\varepsilon_m+e^{-\beta}\cdot\varepsilon_m\text{ (divided through by }\sum_{i=1}^{N}w_i^m)$$

where $\varepsilon_m=\dfrac{\sum_{i=1}^{N}w_i^m I(y_i\neq h(\mathbf{x}_i))}{\sum_{i=1}^{N}w_i^m}$ as defined in step 2b of the **Adaboost.M1** algorithm.

Then

$$-e^{-\beta}+e^{\beta}\cdot\varepsilon_m+e^{-\beta}\cdot\varepsilon_m=-1+e^{2\beta}\cdot\varepsilon_m+\varepsilon_m=0\text{ (multiplied through by }e^{\beta})$$

and

$$e^{2\beta}=\frac{1-\varepsilon_m}{\varepsilon_m}.$$

Therefore,

$$\beta=\frac{1}{2}\ln\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right). \tag{2.39}$$

Now that we have found both β_m and h_m we can proceed to the next step in the **Forward Stagewise Additive Modeling** algorithm (step 2b) and update the approximation

$$f_m(\mathbf{x})=f_{m-1}(\mathbf{x})+\beta_m h_m(\mathbf{x})$$

which causes the weights for the next iteration to be

$$\begin{aligned} w_i^{m+1} &= \exp(-y_i f_m(\mathbf{x}_i)) \text{ ...using equation (2.35)} \\ &= \exp(-y_i f_{m-1}(\mathbf{x}) - y_i \beta_m h_m(\mathbf{x})) \end{aligned}$$

and finally

$$w_i^{m+1} = w_i^m \exp\left(-y_i \beta_m h_m(\mathbf{x})\right) \tag{2.40}$$

Using the well-known identity $-y_i h_m(\mathbf{x}_i) = 2 \cdot I(y_i \neq h_m(\mathbf{x}_i)) - 1$ we can re-write equation (2.40) as

$$w_i^{m+1} = w_i^m \cdot \exp\left[2\beta_m \cdot I(y_i \neq h_m(\mathbf{x}_i)) - \beta_m\right] = w_i^m \cdot \exp\left[\alpha_m I(y_i \neq h_m(\mathbf{x}_i))\right] \cdot e^{-\beta_m} \tag{2.41}$$

where $\alpha_m = 2\beta_m$ is the quantity defined at step 2c in the **Adaboost.M1** algorithm. The factor $e^{-\beta_m}$ in equation (2.41) does not have an effect as it multiplies all the weights by the same value and therefore equation (2.41) is equivalent the weight update step 2d in the **Adaboost.M1** algorithm. Note up until this point we have shown equivalence to steps 2b, 2c and 2d of the **Adaboost.M1** algorithm. Step 2a can be viewed as an approximate solution to equation (2.38), by fitting a weak learner, in this case a base classifier, which minimizes the weighted error rate. The final step 3 in the **Adaboost.M1** algorithm is equivalent to equation 2.21 i.e. the final or full additive model, since $\alpha_m = 2\beta_m$ all the terms in equation (2.21) are multiplied by a constant of two, and combined with the sign function in step 3 of the

Adaboost.M1 algorithm results in the desired output. Setting $M = T$ and $m = t$ completes the proof. ■

Proposition 5 ***Adaboost.M1*** *is equivalent to additive logistic regression by using adaptive Newton updates for minimizing the exponential criterion.*

Proof. Define the exponential loss criterion as in equation (2.28) and suppose we have a current estimate $F(\mathbf{x})$ and seek an improved estimate $F_I(\mathbf{x}) = F(\mathbf{x}) + cf(\mathbf{x})$. We start by fixing c (and $\mathbf{x}$) and expanding $J(F(\mathbf{x}) + cf(\mathbf{x}))$ using regular Taylor series to second order about $f(\mathbf{x}) = 0$. Also let the function $f(\mathbf{x}) \in \{-1, 1\}$ (discrete case).

$$\begin{aligned} J(F + cf) &= E[e^{-y(F(\mathbf{x})+cf(\mathbf{x}))}] = E[e^{-yF(\mathbf{x})-ycf(\mathbf{x})}] \\ &\approx E[e^{-yF(\mathbf{x})}(1 - ycf(\mathbf{x}) + c^2 f(\mathbf{x})^2/2)] \text{ ...where } y^2 = 1 \text{ because} \\ y &\in \{-1, 1\} \end{aligned} \tag{2.42}$$

Note, reference here to e^{β} or e raised to any variable, number, parameter or combination thereof should be interpreted in the usual exponent way. Using the same argument as in the previous proof the quantity $e^{-yF(\mathbf{x})}$ is fixed at the current step (as $F(\mathbf{x})$ is known) and can therefore be regarded as a weight

$$w = w(\mathbf{x}, y) = e^{-yF(\mathbf{x})}. \tag{2.43}$$

Using the definition of weighted conditional expectation taken from Hastie et al.[9]

$$E_w[g(\mathbf{x},y)|\mathbf{x}] \stackrel{def}{=} \frac{E[w(\mathbf{x}, y)g(\mathbf{x},y)|\mathbf{x}]}{E[w(\mathbf{x}, y)|\mathbf{x}]} \tag{2.44}$$

or re-written as

$$E[w(\mathbf{x},y)|\mathbf{x}]\cdot E_w[g(\mathbf{x},y)|\mathbf{x}] = E[w(\mathbf{x},y)g(\mathbf{x},y)|\mathbf{x}], \tag{2.45}$$

we can minimize the approximation in equation (2.42) at each point of $f(\mathbf{x}) \in \{-1,1\}$ i.e. point-wise

$$\begin{aligned}
\widehat{f}(\mathbf{x}) &= \arg\min_f E_w(1 - ycf(\mathbf{x}) + c^2 f(\mathbf{x})^2/2|\mathbf{x})\cdot E(e^{-yF(\mathbf{x})}|\mathbf{x}), \text{ (using definition (2.45))} \\
&= \arg\min_f E_w(1 - ycf(\mathbf{x}) + c^2 f(\mathbf{x})^2/2|\mathbf{x}), \text{ (since } E(e^{-yF(\mathbf{x})}|\mathbf{x}) \text{ is fixed/known)} \\
&= \arg\min_f E_w[1/2 + 1/2(y^2 - 2ycf(\mathbf{x}) + c^2 f(\mathbf{x})^2)|\mathbf{x})], \text{ (replacing 1 with } y^2) \\
&= \arg\min_f E_w[1/2 + 1/2(y - cf(\mathbf{x}))^2|\mathbf{x})] \\
&= \arg\min_f E_w[(y - cf(\mathbf{x}))^2|\mathbf{x})] \\
&\quad \text{, (since } E_w(1/2) \text{ and the coefficient } 1/2 \text{ of } (y - cf(\mathbf{x}))^2 \text{ are constants which} \\
&\quad \text{do not effect the minimization argument)}
\end{aligned}$$

and therefore

$$\widehat{f}(\mathbf{x}) = \arg\min_f E_w[(y - f(\mathbf{x}))^2|\mathbf{x})]. \tag{2.46}$$

Equation (2.46) arises when considering the two possible choice for $f(\mathbf{x}) \in \{-1,1\}$ and which does not effect the minimization argument as c is fixed. Thus minimizing the quadratic approximation (using Taylor series) to the exponential loss criterion leads to a weighted least-squares choice of $f(\mathbf{x}) \in \{-1,1\}$ and this therefore constitutes the Newton step. Given $\widehat{f}(\mathbf{x}) \in \{-1,1\}$, we can now find an estimate for c i.e.

$\widehat{c}$ by directly minimizing $J(F + c\widehat{f})$ using the results of the Lemma 2.29.

$$\begin{aligned}
\widehat{c} &= \arg\min_c J(F + c\widehat{f}) = \arg\min_c E[e^{-yF(\mathbf{x})}e^{-yc\widehat{f}(\mathbf{x})}] \\
&= \arg\min_c E_w[e^{-yc\widehat{f}(\mathbf{x})}], \text{ (using the definition of weighted conditional} \\
&\quad \text{expectation in equation (2.45) and the fact that } E(e^{-yF(\mathbf{x})}) \text{ is fixed/known)} \\
&= \frac{1}{2}\ln\frac{P_w(y\widehat{f}(\mathbf{x}) = 1|\mathbf{x})}{P_w(y\widehat{f}(\mathbf{x}) = -1|\mathbf{x})}, \text{ (using result of the Lemma 2.29)} \\
&\quad \text{where } c = F(x),\ y = y\widehat{f}(\mathbf{x}) \text{ and } P_w(\cdot) \text{ is the weighted probability.}
\end{aligned}$$

Noting $P_w(y\widehat{f}(\mathbf{x}) = 1|\mathbf{x}) = P_w(y = \widehat{f}(\mathbf{x})|\mathbf{x})$ and $P_w(y\widehat{f}(\mathbf{x}) = -1|\mathbf{x}) = P_w(y \neq \widehat{f}(\mathbf{x})|\mathbf{x}) = q$, therefore

$$\widehat{c} = \frac{1}{2}\ln\frac{1-q}{q} \tag{2.47}$$

$$\text{, since } P_w(y\widehat{f}(\mathbf{x}) = 1|\mathbf{x}) = P_w(y = \widehat{f}(\mathbf{x})|\mathbf{x}) = 1 - q$$

$$\text{where } q \text{ can also be written as } q = E_w[I(y \neq \widehat{f}(\mathbf{x})] \tag{2.48}$$

i.e. the weighted expected error rate of incorrect predictions

since,

$$\begin{aligned}
E_w[I(y &\neq \widehat{f}(\mathbf{x})] = \frac{E[wI(y \neq \widehat{f}(\mathbf{x}))|\mathbf{x}]}{E[w|\mathbf{x}]}\text{.(using definition (2.44))} \\
&= \frac{w_{y\neq\widehat{f}(\mathbf{x})}1 \cdot P(y \neq \widehat{f}(\mathbf{x}))}{w} + \frac{w_{y=\widehat{f}(\mathbf{x})}0 \cdot P(y = \widehat{f}(\mathbf{x}))}{w} \\
&= \frac{w_{y\neq\widehat{f}(\mathbf{x})}1 \cdot P(y \neq \widehat{f}(\mathbf{x}))}{w} = P_w(y \neq \widehat{f}(\mathbf{x})).
\end{aligned} \tag{2.49}$$

Note from expression (2.47) c can be negative if $q > \frac{1}{2}$ i.e. the weak learner does worse than 50% which automatically reverses the polarity of f and therefore once again highlighting the requirement for better than random performance of the weak learner. Combining the steps above and the estimates $\widehat{f}(\mathbf{x})$ in (2.46) and $\widehat{c}$ in (2.47) we can produce the update for $F(\mathbf{x})$ and hence find $F_I(\mathbf{x})$ i.e. the improved estimate.

$$F_I(\mathbf{x}) = F(\mathbf{x}) + \widehat{c}\widehat{f}(\mathbf{x}) \tag{2.50}$$

and therefore the update becomes

$$F(\mathbf{x}) \leftarrow F(\mathbf{x}) + \frac{1}{2}\ln\frac{1-q}{q} \times \widehat{f}(\mathbf{x}). \tag{2.51}$$

The implication of update (2.51) is that it also updates the weights. Using equation (2.43)

$$w_u(\mathbf{x}, y) = e^{-yF_I(\mathbf{x})} = e^{-y(F(\mathbf{x})+\widehat{c}\widehat{f}(\mathbf{x}))} = e^{-y(F(\mathbf{x})+\widehat{c}\widehat{f}(\mathbf{x}))} = w(\mathbf{x}, y)e^{-y\widehat{c}\widehat{f}(\mathbf{x})}$$

the weight updates becomes

$$w(\mathbf{x}, y) \leftarrow w(\mathbf{x}, y)e^{-y\widehat{c}\widehat{f}(\mathbf{x})}. \tag{2.52}$$

Using the identity once again $-y\widehat{f}(\mathbf{x}) = 2 \times I(y \neq \widehat{f}(\mathbf{x})) - 1$ we see that the update is equivalent to

$$\begin{aligned} w(\mathbf{x}, y) &\leftarrow w(\mathbf{x}, y) \cdot \exp\left[\frac{1}{2}\ln\left(\frac{1-q}{q}\right)(2 \times I(y \neq \widehat{f}(\mathbf{x})) - 1)\right] \\ &= w(\mathbf{x}, y) \cdot \exp\left[\ln\left(\frac{1-q}{q}\right) I(y \neq \widehat{f}(\mathbf{x})) - \frac{1}{2}\ln\left(\frac{1-q}{q}\right)\right] \end{aligned} \tag{2.53}$$

and therefore the weight update is

$$w(\mathbf{x}, y) \leftarrow w(\mathbf{x}, y) \cdot \exp\left[\ln\left(\frac{1-q}{q}\right) I(y \neq \widehat{f}(\mathbf{x}))\right] \tag{2.54}$$

since the last term on the right-hand-side in (2.53) $-\frac{1}{2}\ln\left(\frac{1-q}{q}\right) = -\widehat{c}$ multiplies all the weights by the same value and therefore has no effect. In conclusion we have shown in equation (2.48) that $q = \widehat{\varepsilon}$, that $\alpha = 2\widehat{c}$, and that the weight update rule (2.54) is equivalent to step 2d of **Adaboost.M1.** Using the same arguments in the previous proof we can view equation (2.46) as being a least squares approximation to step 2a of **Adaboost.M1.** Lastly we note the final prediction in step 3 of **Adaboost.M1** is equivalent by way of construction to equation (2.50). ■

2.6 Adaboost Error

One of the central ideas underpinning the development of Adaboost is the method's ability to improve prediction accuracy and hence reduce the final prediction error of h_f. In this section we shall analyze the theoretical effect that Adaboost has on the training error by understanding the training error bounds and in Chapter 4 we will analyze this effect empirically. Given the definition of error in equation (2.3) we can use this to define the training error at each boosting round t using the distribution of (2.9) as

$$\epsilon_t = P_{i\sim\mathbf{p}^t}[h_t(\mathbf{x}_i) \neq y_i] = \sum_{i:h_t(\mathbf{x}_i)\neq y_i} p_i^t \tag{2.55}$$

where the p_i^t's are normalized weights for each observation at each boosting round t. In our analysis to follow we start by writing the above training error (2.55) of a weak learner h_t on the t^{th} boosting round as

$$\epsilon_t = \frac{1}{2} - \gamma_t, \tag{2.56}$$

where $\gamma_t \geq 0$ is a measure of how much better the weak learner h_t is than random guessing since we have assumed we remain in the binary case. The authors Freund et al. [5] then proved that the training error of the final prediction has an upper bound of

$$\epsilon_f \leq \prod_t^T \left[2\sqrt{\epsilon_t(1-\epsilon_t)}\right] = \prod_t^T \sqrt{1-4\gamma_t^2} \leq \exp\left(-2\sum_t^T \gamma_t^2\right). \tag{2.57}$$

Therefore the training error of the final prediction ϵ_f is given in terms of the accuracy of the individual component predictions and hence if the $\gamma_t's$ increase, the error drops exponentially fast. Pre-Adaboost algorithms required some knowledge of the error bound or accuracy of h_t, however in practice this information is generally not known or difficult to obtain. It is the ability of Adaboost to continue in the absence of this prior information and dynamically adapt to the error of the individual weak predictors which has set it apart from other boosting methods. We shall now formally state the theorem and proof that the training error of h_f i.e. ϵ_f is bounded above as per inequality (2.57). The following theorem and proof are both taken from Freund et al. [5] however they have been expanded on for completeness.

Theorem 6 *Suppose the weak learners h_t generate training errors $\epsilon_1, \epsilon_2, ..., \epsilon_T$ for each boosting round. Then the final prediction error $\epsilon_f = P_{i\sim\mathbf{p}}[h_f(\mathbf{x}_i) \neq y_i]$, where $\mathbf{p}$ is the population distribution, of the final prediction h_f is bounded above by*

$$\epsilon_f \leq \prod_t^T \left[2\sqrt{\epsilon_t(1-\epsilon_t)}\right]. \tag{2.58}$$

which can also be written as

$$\epsilon_f \leq \prod_t^T \sqrt{1-4\gamma_t^2} \tag{2.59}$$

or,

$$\epsilon_f \leq \exp\left(-2\sum_t^T \gamma_t^2\right) \tag{2.60}$$

Proof. Let y, $\mathbf{w}^t$ and $\mathbf{p}^t$ be defined as in the **Original Adaboost** algorithm. We start by using the update rule given in the **Original Adaboost** algorithm Step 2e and the

main arguments from Lemma 2.11 and Theorem 2.17.

$$\begin{aligned}
\sum_{i=1}^{N} w_i^{t+1} &= \sum_{i=1}^{N} w_i^t \beta_t^{1-|h_t(\mathbf{x}_i)-y_i|} \\
&\leq \sum_{i=1}^{N} w_i^t [1-(1-\beta_t)(1-|h_t(\mathbf{x}_i)-y_i|)], \text{ using} \\
&\quad \text{the Convexity Argument in 2.12} \\
&\quad \text{and where } \alpha = \beta_t \text{ and } r = 1-|h_t(\mathbf{x}_i)-y_i| \\
&= \left(\sum_{i=1}^{N} w_i^t\right)[1-(1-\beta_t)(1-\epsilon_t)], \\
\text{where } \frac{\sum_{i=1}^{N} w_i^t (1-|h_t(\mathbf{x}_i)-y_i|)}{\sum_{i=1}^{N} w_i^t} &= \frac{\sum_{i:h_t(\mathbf{x}_i)\neq y_i}^{N} w_i^t}{\sum_{i=1}^{N} w_i^t} \\
&= \sum_{i:h_t(\mathbf{x}_i)\neq y_i} p_i^t \\
&= \epsilon_t \text{ as per equation 2.55.}
\end{aligned}$$

Using the same argument as in Lemma 2.11 for repeated application where $t = 1, ..., T$ and noting that $\sum_{i=1}^{N} w_i^1 = 1$ is by definition a distribution as per Step 1 of the **Original Adaboost**, we get

$$\sum_{i=1}^{N} w_i^{T+1} \leq \prod_{t=1}^{T} [1-(1-\beta_t)(1-\epsilon_t)]. \tag{2.61}$$

We note in the **Original Adaboost** algorithm that the final prediction h_f only make a mistake on observation i when

$$\prod_{t=1}^{T} \beta_t^{-|h_t(\mathbf{x}_i)-y_i|} \geq \left(\prod_{t=1}^{T} \beta_t\right)^{-1/2} \tag{2.62}$$

where the derivation of (2.62) is shown below and follows from step 3 of the **Original Adaboost** algorithm and noting that h_f will only make a mistake when $|h_t(\mathbf{x}_i) - y_i| = 1$ for most boosting rounds,

$$\begin{aligned}
\sum_{t=1}^{T}(\ln 1/\beta_t) \times |h_t(\mathbf{x}_i) - y_i| &\geq \frac{1}{2}\sum_{t=1}^{T}\ln 1/\beta_t \text{ from step 3 of the } \textbf{Original Adaboost} \\
\sum_{t=1}^{T}(\ln \beta_t) \times - |h_t(\mathbf{x}_i) - y_i| &\geq \frac{1}{2}\sum_{t=1}^{T} - \ln \beta_t \\
\sum_{t=1}^{T}\ln \beta_t^{-|h_t(\mathbf{x}_i)-y_i|} &\geq -\frac{1}{2}\sum_{t=1}^{T}\ln \beta_t \\
\ln\left(\prod_{t=1}^{T}\beta_t^{-|h_t(\mathbf{x}_i)-y_i|}\right) &\geq \ln\left(\prod_{t=1}^{T}\beta_t\right)^{-\frac{1}{2}} \\
\prod_{t=1}^{T}\beta_t^{-|h_t(\mathbf{x}_i)-y_i|} &\geq \left(\prod_{t=1}^{T}\beta_t\right)^{-\frac{1}{2}} \text{ as per (2.62).}
\end{aligned}$$

The final weight of any observation i is then

$$\begin{aligned}
w_i^{T+1} &= w_i^T \beta_T^{1-|h_T(\mathbf{x}_i)-y_i|} \\
&= w_i^{T-1} \beta_{T-1}^{1-|h_{T-1}(\mathbf{x}_i)-y_i|} \beta_T^{1-|h_T(\mathbf{x}_i)-y_i|} \\
&= \ldots \\
&= D(i)\prod_{t=1}^{T}\beta_t^{1-|h_t(\mathbf{x}_i)-y_i|} \qquad (2.63)
\end{aligned}$$

where $D(i) = w_i^1$ are the chosen initial weights as per step 1 of the **Original Adaboost** algorithm. Using expressions (2.62) and (2.63) we can derive the lower bound for the sum of the final weights by using the sum of the final weights where

h_f is incorrect,

$$\begin{aligned}
\sum_{i=1}^{N} w_i^{T+1} &\geq \sum_{i:h_f(\mathbf{x}_i)\neq y_i}^{N} w_i^{T+1}, \text{ because } i : h_f(\mathbf{x}_i) \neq y_i \subseteq N \\
&= \sum_{i:h_f(\mathbf{x}_i)\neq y_i}^{N} \left[D(i) \prod_{t=1}^{T} \beta_t^{1-|h_t(\mathbf{x}_i)-y_i|} \right], \text{ using equation (2.63)} \\
&\geq \sum_{i:h_f(\mathbf{x}_i)\neq y_i}^{N} \left[D(i) \left(\prod_{t=1}^{T} \beta_t \right)^{1/2} \right], \text{ using the inequality (2.62) multiplied by } \beta^T \\
&= \left(\sum_{i:h_f(\mathbf{x}_i)\neq y_i}^{N} D(i) \right) \left(\prod_{t=1}^{T} \beta_t \right)^{1/2} \\
&= \epsilon_f \cdot \left(\prod_{t=1}^{T} \beta_t \right)^{1/2}, \text{ using the definition of training error in (2.55) applied to the final error.} \qquad (2.64)
\end{aligned}$$

Using equations (2.61) and (2.64) we get

$$\begin{aligned}
\epsilon_f &\leq \frac{\sum_{i=1}^{N} w_i^{T+1}}{\left(\prod_{t=1}^{T} \beta_t \right)^{\frac{1}{2}}}, \text{ using inequality (2.64)} \\
&\leq \frac{\prod_{t=1}^{T} [1 - (1-\beta_t)(1-\epsilon_t)]}{\left(\prod_{t=1}^{T} \beta_t \right)^{\frac{1}{2}}}, \text{ using inequality (2.61)} \\
&= \prod_{t=1}^{T} \left[\frac{[1 - (1-\beta_t)(1-\epsilon_t)]}{\sqrt{\beta_t}} \right] \qquad (2.65)
\end{aligned}$$

Since we note that all the factors in the product of equation (2.65) are positive we can minimize the right-hand-side by minimizing each factor with respect to β_t

$$
\begin{aligned}
& \frac{\partial}{\partial \beta_t}\left\{\frac{[1-(1-\beta_t)(1-\epsilon_t)]}{\sqrt{\beta_t}}\right\} \\
= & \frac{\partial}{\partial \beta_t}\left\{\frac{\beta_t+\epsilon_t-\beta_t\epsilon_t}{\sqrt{\beta_t}}\right\} \\
= & \frac{\partial}{\partial \beta_t}\left\{\beta_t^{\frac{1}{2}}+\beta_t^{-\frac{1}{2}}\epsilon_t-\beta_t^{\frac{1}{2}}\epsilon_t\right\} \\
= & \frac{1}{2}\beta_t^{-\frac{1}{2}}-\frac{1}{2}\beta_t^{-\frac{3}{2}}\epsilon_t-\frac{1}{2}\beta_t^{-\frac{1}{2}}\epsilon_t \\
= & 1-\beta_t^{-1}\epsilon_t-\epsilon_t, \text{ by dividing through by } \frac{1}{2}\beta_t^{-\frac{1}{2}} \\
\beta_t = & \frac{\epsilon_t}{1-\epsilon_t}, \text{ by setting the above result equal to zero} \qquad (2.66)
\end{aligned}
$$

Therefore equation (2.66) minimizes equation (2.65) and plugging equation (2.66) into equation (2.65) we get the final expression for the upper bound.

$$
\begin{aligned}
\epsilon_f \leq & \prod_{t=1}^{T}\left[\frac{[1-(1-\frac{\epsilon_t}{1-\epsilon_t})(1-\epsilon_t)]}{\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}}\right] \\
= & \prod_{t=1}^{T}\left[\frac{[1-(1-\epsilon_t-\frac{\epsilon_t}{1-\epsilon_t}+\frac{\epsilon_t^2}{1-\epsilon_t})]}{\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}}\right] \\
= & \prod_{t=1}^{T}\left[\frac{[\epsilon_t+\frac{\epsilon_t-\epsilon_t^2}{1-\epsilon_t}]}{\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}}\right] \\
= & \prod_{t=1}^{T}\left[\frac{[\epsilon_t+\frac{\epsilon_t(1-\epsilon_t)}{1-\epsilon_t}]}{\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}}\right] \\
= & \prod_{t=1}^{T}\left[\frac{2\epsilon_t}{\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}}\right] \\
= & \prod_{t=1}^{T}\left[2\epsilon_t\times\sqrt{\frac{1-\epsilon_t}{\epsilon_t}}\right] \\
= & \prod_{t=1}^{T}\left[2\sqrt{\epsilon_t(1-\epsilon_t)}\right]. \qquad (2.67)
\end{aligned}
$$

To show the alternate expression of inequality (2.59) we simply use equation (2.56) in place of ϵ_t in equation (2.67)

$$\begin{aligned}\epsilon_f &\leq \prod_{t=1}^{T}\left[2\sqrt{\epsilon_t(1-\epsilon_t)}\right] = \prod_{t=1}^{T}\left[2\sqrt{\left(\frac{1}{2}-\gamma_t\right)\left(1-\frac{1}{2}+\gamma_t\right)}\right] \\ &= \prod_{t=1}^{T}\left[2\sqrt{\frac{1}{4}-\gamma_t^2}\right] \\ &= \prod_{t=1}^{T}\left[\sqrt{1-4\gamma_t^2}\right].\end{aligned}$$

To show the final inequality (2.60) we use the Kullback-Leibler divergence

$$KL(a \parallel b) = a\ln(a/b) + (1-a)\ln((1-a)/(1-b))$$

where $a = 1-a = \frac{1}{2}$ and $b = \epsilon_t = \frac{1}{2}-\gamma_t$ we get

$$\begin{aligned}KL\left(\frac{1}{2} \parallel \frac{1}{2}-\gamma_t\right) &= \frac{1}{2}\ln\left(\frac{1}{2}/\left[\frac{1}{2}-\gamma_t\right]\right) + \frac{1}{2}\ln\left(\frac{1}{2}/\left[\frac{1}{2}+\gamma_t\right]\right) \\ &= \frac{1}{2}\left\{\ln\left(\frac{1}{2}\right) - \ln\left(\frac{1}{2}-\gamma_t\right)\right\} + \frac{1}{2}\left\{\ln\left(\frac{1}{2}\right) - \ln\left(\frac{1}{2}+\gamma_t\right)\right\} \\ &= \ln\left(\frac{1}{2}\right) - \frac{1}{2}\ln\left(\frac{1}{2}-\gamma_t\right) - \frac{1}{2}\ln\left(\frac{1}{2}+\gamma_t\right). \qquad (2.68)\end{aligned}$$

Using the inequality (2.59) we get,

$$
\begin{aligned}
\epsilon_f &\leq \prod_{t=1}^{T}\left[\sqrt{1-4\gamma_t^2}\right] \\
&= \exp\left\{\ln\left(\prod_{t=1}^{T}\left[\sqrt{1-4\gamma_t^2}\right]\right)\right\} \\
&= \exp\left\{\sum_{t=1}^{T}\ln\left[\sqrt{1-4\gamma_t^2}\right]\right\} \\
&= \exp\left\{\frac{1}{2}\sum_{t=1}^{T}\ln\left(1-4\gamma_t^2\right)\right\} \\
&= \exp\left\{\frac{1}{2}\sum_{t=1}^{T}\ln\left[(1-2\gamma_t)(1+2\gamma_t)\right]\right\} \\
&= \exp\left\{\frac{1}{2}\sum_{t=1}^{T}\left[\ln 2\left(\frac{1}{2}-\gamma_t\right)+\ln 2\left(\frac{1}{2}+\gamma_t\right)\right]\right\} \\
&= \exp\left\{\sum_{t=1}^{T}\left[\frac{1}{2}\ln 2+\frac{1}{2}\ln\left(\frac{1}{2}-\gamma_t\right)+\frac{1}{2}\ln 2+\frac{1}{2}\ln\left(\frac{1}{2}+\gamma_t\right)\right]\right\} \\
&= \exp\left\{-\sum_{t=1}^{T}\left[\ln\frac{1}{2}-\frac{1}{2}\ln\left(\frac{1}{2}-\gamma_t\right)-\frac{1}{2}\ln\left(\frac{1}{2}+\gamma_t\right)\right]\right\} \\
&= \exp\left\{-\sum_{t=1}^{T}KL\left(\frac{1}{2}\parallel\frac{1}{2}-\gamma_t\right)\right\}\text{,...as per (2.68)} \qquad (2.69) \\
&\leq \exp\left\{-2\sum_{t=1}^{T}\gamma_t^2\right\}
\end{aligned}
$$

Where the last step follows from the fact that $KL(a \parallel b)$ is an asymmetric measure of distance of b from a and therefore the "distance" is γ_t. Following on from this the smaller the inner term $KL(\bullet)$ of equation (2.69) is, and therefore distance, the larger the whole quantity on the right-hand side of inequality (2.69) and since $0 \leq \gamma_t \leq \frac{1}{2}$, therefore $2\gamma_t^2 \leq \gamma_t$. ■

We note that we proved the above theorem for the discrete binary case where $y \in \{0, 1\}$. In this case the condition of $\epsilon_t \geq \frac{1}{2}$ is acceptable however as noted earlier, in the multi-class case we require a stricter condition such that $\epsilon_t \leq \frac{1}{2}$.

We conclude this sections with a brief view of the generalization error of Adaboost. The authors in Freund et al. [5] showed that the generalization error of the Adaboost method, with high probability, is at most

$$\widehat{P}[h(\mathbf{x}) \neq y] + \widetilde{O}\left(\sqrt{\frac{Td}{N}}\right) \tag{2.70}$$

where $\widehat{P}[\cdot]$ is the empirical probability on the training set and d is the number of VC-dimensions which Freund et al. [7] quotes as "a measure of the "complexity" of a space of hypotheses". For further information on VC dimensions see Blumer et al. [12]. What is interesting in expression (2.70) is that it suggests that boosting will overfit if run for too many rounds i.e. the upper generalization error bound becomes large as T increases. However, as we will see in the Chapter 4, in most cases overfitting does not occur when run for hundreds of boosting rounds and that Adaboost continues to drive down the test error long after the training error nears or is at zero. It is this somewhat positive phenomenon that lends to Adaboost's practical and attractive application as an effective boosting method.

Chapter 3
Classification Trees as a Base Predictor

In this chapter we will provide the reasons for selecting classification trees as our chosen base predictor which will be made use of within the Adaboost examples in the following Chapter 4. An overview of the **C**lassification **A**nd **R**egression **T**ree ("CART") method created by Breiman et al.[16] is provided which is used to construct or grow our classification trees. In addition we will also show and analyze the method of *cost-complexity pruning* used to prune or optimize the outputted tree structure. Lastly examples showing how classification trees are applied to actual datasets and a review of the resulting performance is provided. To ensure a complete analysis both simulated and real data-sets are made use of and within each we analyze a binary classification and multi-classification problem.

3.1 The choice of decision trees as a base predictor

As we have seen in Chapter 2, Adaboost is a generalized method of enhancing the predictive accuracy of weak learners or base predictors. A question which then follows is *which* weak learner or base predictor should we select for boosting and *why*. In most cases it is difficult to select the "best" weak learning method as the performance of these methods tend to vary depending on the data in terms of size, complexity and completeness. In addition the methods should also satisfy important practical

considerations to be regarded as useful such as exhibiting quick computational time, being easy to understand as well as allowing an interpretation of the results. The authors Hastie et al. [4] (p 351) attempt to rate various popular learning methods along different criteria. We have added to their comparison by including a relative score calculated using an equal weighting of each characteristic and applying the following score where 1 = good, 0 = fair, -1 = poor. The results of the comparison and corresponding scores are shown below.

Characteristic	Neural Nets	SVM	Trees	MARS	k-Kernels
Natural handling of data of "mixed" type	$\Downarrow -1$	$\Downarrow -1$	$\Uparrow +1$	$\Uparrow +1$	$\Downarrow -1$
Handling of missing values	$\Downarrow -1$	$\Downarrow -1$	$\Uparrow +1$	$\Uparrow +1$	$\Uparrow +1$
Robustness to outliers in input space	$\Downarrow -1$	$\Downarrow -1$	$\Uparrow +1$	$\Downarrow -1$	$\Uparrow +1$
Insensitive to monotone transformations of inputs	$\Downarrow -1$	$\Downarrow -1$	$\Uparrow +1$	$\Downarrow -1$	$\Downarrow -1$
Computational scalability (large N)	$\Downarrow -1$	$\Downarrow -1$	$\Uparrow +1$	$\Uparrow +1$	$\Downarrow -1$
Ability to deal with irrelevant inputs	$\Downarrow -1$	$\Downarrow -1$	$\Uparrow +1$	$\Uparrow +1$	$\Downarrow -1$
Ability to extract linear combinations of features	$\Uparrow +1$	$\Uparrow +1$	$\Downarrow -1$	$\Downarrow -1$	$\Longleftrightarrow 0$
Interpretability	$\Downarrow -1$	$\Downarrow -1$	$\Longleftrightarrow 0$	$\Uparrow +1$	$\Downarrow -1$
Predictive power	$\Uparrow +1$	$\Uparrow +1$	$\Downarrow -1$	$\Longleftrightarrow 0$	$\Uparrow +1$
Score	-5	-5	+4	+2	-2

It is clear from the table that Trees as a learning method appears to be the most attractive and scores higher than the other methods by a significant margin aside from MARS. Trees however fail on only two criteria being,"Ability to extract linear combinations of features", as it is constructed by a series of indicator functions, and "Predictive power" which we shall demonstrate in Section 3.3 to Section 3.6 . It is the failure of Tree methods on the latter criteria that is of interest as boosting has been specifically designed to enhance weak learners or put in other words, learners which have poor predictive power. The authors Hastie et al. [4] (p 351) then go on to refer to an "off-the-shelf" method. They define "off-the-shelf" as a method that can

be directly applied to the data without requiring a great deal of time-consuming data preprocessing or careful tuning of the learning procedure. Referring back to the table we can now see that Trees appear to meet most of the requirements to be called an "off-the-shelf" learning method primarily because they are:

- Fast to construct
- Easy to implement
- Produce interpretable models (when the Trees are small)
- Can accommodate mixed variables - categorical and numeric
- Not sensitive to missing values
- Invariant to monotone transformations of the predictors and therefore immune to scaling and outlier effects
- By design perform internal feature selection i.e. select the most appropriate variables to be used, and can therefore deal with data comprising of large domains.

It is because of the analysis and reasons given above that Trees has been selected as the base learner to be used in this dissertation with classification trees as a specific tree based method. We will show in Chapter 4 that classification trees also happen to benefit from an accuracy point of view when used with Adaboost.

3.2 Classification tree methodology

We start by understanding the basic methodology of classification trees in that classification trees work by recursively splitting the domain of $\mathbf{x}_i \in \mathbf{X}$ where $\mathbf{x}_i = \langle x_{1i}, x_{2i}, ..., x_{pi} \rangle'$ is a vector consisting of p input variables, into a set of separate multi-dimensional "cubes", called nodes or regions, with each node defining a class label. For a K class problem where the class labels are denoted $k = \{1, 2, ..., K\}$ and where we have created M such regions or nodes $R_1, R_2, ..., R_M$, classification trees seek to find a mapping $f(X) : \mathbf{X} \rightarrow \{1, 2, ..., K\}$ using the model,

$$f(\mathbf{x}) = \sum_{m=1}^{M} k(m) I(\mathbf{x} \in R_m), \tag{3.1}$$

where $k(m) : \{1, 2, ..., M\} \rightarrow \{1, 2, ..., K\}$ is the mapping of the M regions to the K classes.

As noted we shall describe one of the most popular tree-based method known as CART, however focusing on the first letter of the acronym i.e. the "C" part of CART. The primary goal in constructing classification trees using CART and hence the creation of the partitions and resultant regions R_m, is to determine which of the input variables $j = 1, 2, ..., p$ to select for splitting and at what points s, known as split-points, these variables need to be split. Once these regions are defined we also need to determine which classification to make within each region in order to create the best fit. This process is an iterative one as splitting, or the formation of multiple regions, is continued until some predefined stopping rule is triggered or we have as

many classes as there are observations. Usually binary trees i.e. trees with a single variable j and split point s which result in only two child nodes, are used in the CART method however creation of trees comprising of more than two child nodes is possible but not shown here due to added structural complexity. The CART method for constructing binary trees can best be understood using a simple example taken from Hastie et al.[4] (p 305-317) however applied to the case of a two class $K = 2$ classification problem.

Example 1 *Consider a classification problem where* $\mathbf{X}$ *consists of two input variables such that* $\mathbf{X} = \langle X_1, X_2 \rangle'$ *and can only take on values in the unit interval i.e.* $X_1 \in [0, 1]$ *and* $X_2 \in [0, 1]$. *Let* Y *be the binary dependent variable such that* $K = 2$ *and* $Y \in \{-1, 1\}$. *Then* X_1 *is first chosen as the variable to be split,* $j = 1$ *and is thereafter split at point* t_1, $s = t_1$. *Next, the region* $X_1 \leq t_1$ *is split at* $X_2 = t_2$ *and the region* $X_1 > t_1$ *split at* $X_1 = t_3$. *Finally the region* $X_1 > t_3$ *is split at* $X_2 = t_4$. *The result of this process is a partition into five regions or terminal nodes* R_1, R_2, R_3, R_4 *and* R_5 *shown in Figure 3.1 with each of the regions defining a class label either* 1 *or* 2 .

The same tree growing process can be depicted using a binary tree or dendogram, see Figure 3.2. The full data-set sits on the top of the tree. Observations satisfying the condition at each split are assigned to the left branch, and conversely observations failing to satisfy the condition are assigned to the right branch. The terminal nodes correspond to the regions R_1, R_2, R_3, R_4 *and* R_5. *A key advantage of*

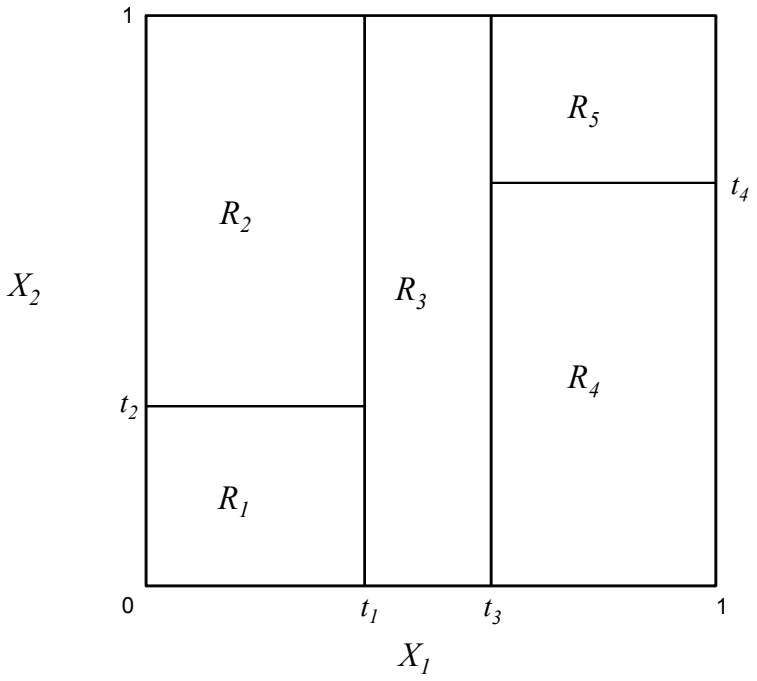


Figure 3.1: Graphical representation of how the CART methodology works using Example 1 as shown in Hastie et al.[4] (pg 306).

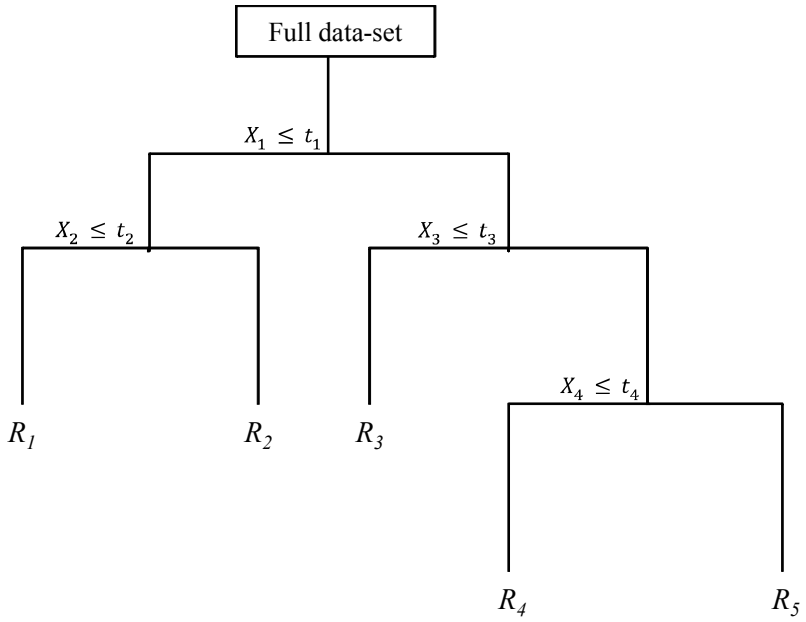


Figure 3.2: Figure showing the graphical respresentation of the CART method using a binary tree generated from Example 1 as also shown in Hastie et al.[4] (pg 306).

this recursive binary tree representation is interpretability as the entire domain space can be described using a single tree. Additionally as the number of input variables exceed two dimensions, representations as in Figure 3.1 become difficult whilst representations as in Figure 3.2 can quite easily accommodate higher dimension domain spaces.

The next questions which arises is how does one go about partitioning the domain to find the regions or nodes, in other words we need to find split variable j and split-point s. In addition we also need to determine what classification needs to be made within each region or node i.e. in our example this would be equivalent to deciding whether to classify the response variable Y in each region as 1 or -1. We

shall use the CART method as described by Hastie et al.[4] (p 305-317) for regression trees but adapted for classification trees. We start by defining the first pair of half-planes

$$R_1(j,s) = \{X \mid X_j \leq s\} \text{ and } R_2(j,s) = \{X \mid X_j > s\}.$$

We then need to solve the minimization problem

$$\min_{j,s} \left[N_1 Q_1(T) + N_2 Q_2(T)\right] \tag{3.2}$$

where $Q_m(T)$ is a measure of node impurity in node m of tree T with $|T|$ terminal nodes (at first $|T| = 2$) i.e. in our case we have chosen to use the miss-classification rate as an impurity measure in each node,

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m(j,s)} I(y_i \neq k(m)) \tag{3.3}$$

and where N_m (also known as node size) is the number of observations falling in region R_m and $k(m) = \arg\max_k \widehat{p}_{mk}$ is the majority or modal class in node m where

$$\widehat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m(j,s)} I(y_i = k) \tag{3.4}$$

is the proportion of class k observations in node m. Note combining equation (3.3) and equation (3.4) Q_m can be expressed as

$$\begin{aligned} Q_m(T) &= \frac{1}{N_m} \sum_{x_i \in R_m(j,s)} I(y_i \neq k(m)) \\ &= \frac{1}{N_m} \sum_{x_i \in R_m(j,s)} [1 - I(y_i = k(m))] \\ &= \frac{1}{N_m} \left[N_m - \sum_{x_i \in R_m(j,s)} I(y_i = k(m)) \right] \\ &= 1 - \frac{1}{N_m} \sum_{x_i \in R_m(j,s)} I(y_i = k(m)) \\ &= 1 - \widehat{p}_{mk(m)} \end{aligned} \tag{3.5}$$

Therefore the inner term of expression (3.2) has the effect of finding split variable j and split point s which creates the regions with the smallest number of miss-classified observations based on the modal class or, more generally, the smallest impurity measure. Once j and s are found the process is then repeat on the resulting regions R_1 and R_2.

Now that we known how to grow the tree we need to be able to determine when to stop growing it. This is usually done using some predefined stopping rule such as choosing not to split a node if it contains less than some predefined number of observations since continuing to build the full tree i.e. $|T| = N$ can become computationally infeasible as well as potentially resulting in over-fitting. Conversely if we stop too early and thereby build small trees i.e. trees with a small number of terminal nodes $|T|$, we run the risk of losing prediction accuracy. Therefore one

needs to find an optimum tree size which exists somewhere in-between the full tree and the tree stump.

One such method to find an optimum tree size is known as *cost-complexity pruning* as suggested by the authors in Hastie et al.[4] (p 295-366). In summary the method works by growing a tree to size T_0 stopping the process only when some predefined minimum node size (say $N_m = 5$) is reached. This large tree T_0 is then pruned by collapsing any number of its non-terminal nodes resulting in a sub-tree $T_\alpha \subseteq T_0$ where $\alpha \geq 0$ is a tuning parameter. We define the cost complexity criterion as

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T| \tag{3.6}$$

with the idea being to find a sub-tree T_α for some α which minimizes equation (3.6). The effect of the tuning parameter is that for large values of $|T|$, α needs to be small in order to minimize equation (3.6) and therefore smaller pruned trees are produced and similarly for small values of α, $|T|$ can be larger resulting in larger pruned trees. T_α is then found by collapsing the internal (non-terminal) nodes which produce the smallest per-node increase in $\sum_{m=1}^{|T|} N_m Q_m(T) = \sum_{m=1}^{|T|} \left[\sum_{x_i \in R_m(j,s)} I(y_i \neq k(m))\right]$ which in our case is the total number of miss-classified observations across all the terminal nodes. Intuitively this makes sense as we are successively collapsing the nodes which results in the smallest increase in miss-classification and hence preserving accuracy to a degree. We continue this process until we reach the single-node (root) tree and in doing so generate a finite series of trees with diminishing number

of nodes. Then according to Hastie et al.[4] (p 295-366) this series of trees must contain T_α. To find an estimate for α we select the value $\widehat{\alpha}$ which minimizes the x-fold cross-validated miss-classification rate to produce the final tree $T_{\widehat{\alpha}}$.

We now turn to understanding the effectiveness of using miss-classification as a node impurity measure Q_m as defined in equation (3.3). As we grow the tree we can potentially end up with a large tree particularly if we have multiple inputs variables and a large number of observations. In this case numerical optimization becomes an appropriate choice as apposed to the manual and time consuming process of scanning through the input variables and its values. It is within this context that the authors in Hastie et al.[4] (p 295-366) describe another impurity measures for Q_m known as the *Gini index* and defined as

$$Q_m(T) = \sum_{k=1}^{K} \widehat{p}_{mk}(1 - \widehat{p}_{mk}). \tag{3.7}$$

Comparing the two impurity measures when faced with a two class problem where $K = 2$ and p is the proportion in the second class, equation (3.5) then becomes $1 - \max(p, 1-p)$ whilst equation (3.7) becomes $(1-p)p + p(1-p) = 2p(1-p)$. Plotting both these expressions as a function of p we get Figure 3.3 where we can see that immediately the Miss-classification index can become problematic as it is not continuously differentiable and therefore limits numerical optimization. Additionally from Figure 3.3 we note the superior performance of the Gini index from the parabolic shape of its graph which shows that it is more sensitive to changes in p i.e. the Gini index drops faster on either side of $p = 0.5$ than the miss-classification index

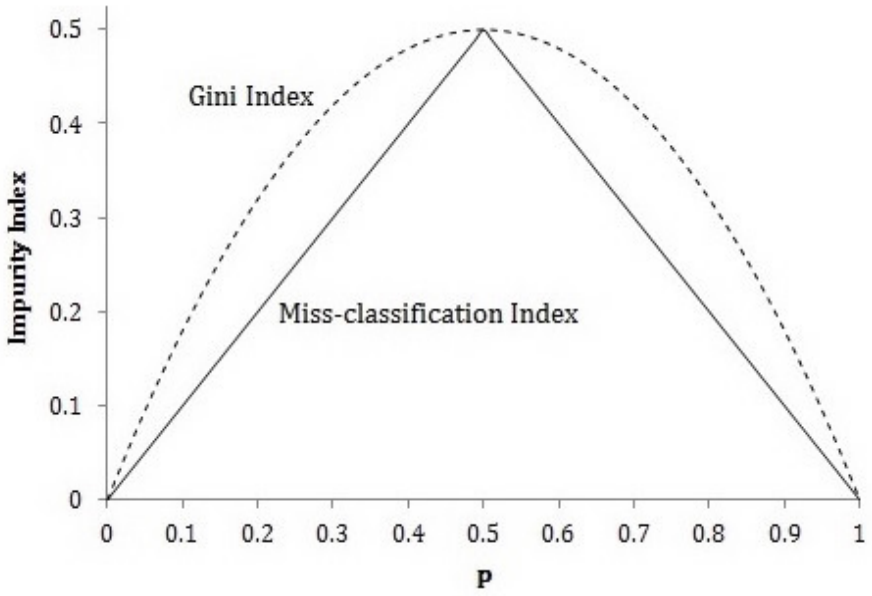


Figure 3.3: As per Hastie et al.[4] (p 309) showing node impurity measures for two–class classification being Miss-classification and Gini Index, as a function of the proportion p in the second class.

and is therefore potentially a better measurement of node impurity. The difference in effectiveness can also be demonstrated by expanding on the example as described in Hastie et al.[4] (p 295-366).

Example 2 *Assume we are faced with a two-class problem (denoted class* (A, B)*) and a learning set of data which comprises 400 observations in each class (denoted as* $(400, 400)$*). Suppose we then construct a tree using the method described above which results in the first split creating a left child node of* $(300, 100)$ *and a right child node of* $(100, 300)$*. Therefore based on the method above the left child node classifies observations falling in this region to class* A *(based on the modal class i.e.* 300

observations of class A *verses* 100 *observations of class* B*) and similarly the right child node classifies observations to class* B*. Therefore the total miss-classification rate for this split is* $(100 + 100)/(400 + 400) = 200/800 = 0.25$*. Next suppose we repeat the tree growing process on the full set of learning data which results in a left child node of* $(200, 400)$ *and a right child node of* $(200, 0)$ *then this split also creates a miss-classification rate* $(200 + 0)/(400 + 400) = 200/800 = 0.25$ *however is intuitively preferable over the previous split as the right child node is now a pure node. If we calculate the Gini index for both these splits remembering that* p *is the proportion in the second class (in our case class* B*) we get* $100/400 * (1 - 100/400) + 300/400 * (1 - 300/400) = 0.375$ *for the first split and* $400/600 * (1 - 400/600) + 0/200 * (1 - 0/200) = 0.222$ *for the second split and therefore if we where to use the Gini index as the chosen impurity measure we would have certainly produced the tree with the second split with the better structure.*

We can therefore produce a simple tree growing rule using both impurity measures. When growing a tree one should use the Gini index as the impurity measure or more specifically use equation (3.7) in expression (3.2) and when pruning the tree using cost-complexity one should use miss-classification error or more specifically use equation (3.3) in equation (3.6). The choice of the miss-classification index for pruning is because it is an easy and quick measure to calculate however the Gini index could just as equally be used without any loss in accuracy.

3.3 Simulated binary classification example

We shall now generate the dataset that was used in Example 1 but with specific classification values $\{-1, 1\}$ for each region such that R_1 and R_3 classify the response variable as 1 and similarly R_2, R_4 and R_5 make a -1 classification. Following the creation of this dataset we will then build the classification tree using the process described in Section 3.2 above and subsequently use *cost-complexity pruning* to prune the tree. As a note we have elected to stop the tree growing procedure for a particular node when that node contains at most 5 observations. We shall use 10-fold cross validation to examine the error. The code shown in Appendix A.1, and more generally for all examples used in this dissertation, can be run on the open source and free statistical computer package R (*http://cran.r-project.org/*) by simply copying and pasting the text directly into the programme.

As we can see from Figure 3.5 the full tree does not resemble the structure as shown in Figure 3.2 and is too cumbersome to be effective practically. Figure 3.4 is generated using the *printcp()* command in R as detailed in Appendix A.1. The *"root node error"* in this case is the number of $1's$ relative to the full data set. The column labeled *"CP"* is known as the *complexity parameter* and is analogous to α in equation (3.6) however is scaled to the unit interval such that $\alpha = 1$ generates a tree with no splits and conversely for small values of α large trees are generated evidenced by the 10^{th} row in Figure 3.4 where $nsplit = 61$. This is also the reason why we specified in the R code that *cp=0* as the tree growing process would have

```
Classification tree:
rpart(formula = Y ~ X1 + X2, data = Data, method = "class", parms = list(split = "gini"),
    control = stoppingrule)

Variables actually used in tree construction:
[1] X1 X2

Root node error: 443/1000 = 0.443

n= 1000

           CP nsplit rel error  xerror     xstd
1  0.11173815      0   1.00000 1.00000 0.035459
2  0.00451467      3   0.65688 0.71106 0.033159
3  0.00300978      5   0.64786 0.79684 0.034114
4  0.00290229     18   0.60045 0.80813 0.034222
5  0.00282167     29   0.56433 0.81716 0.034305
6  0.00225734     37   0.53499 0.81716 0.034305
7  0.00150489     44   0.51919 0.84876 0.034577
8  0.00112867     54   0.50339 0.86005 0.034666
9  0.00075245     58   0.49887 0.88262 0.034833
10 0.00000000     61   0.49661 0.90293 0.034971
> |
```

Figure 3.4: Screenshot of the R output using the printcp() command of the full tree generated using Example 1 applied to a binary classification problem

stopped once the scaled complexity parameter exceeded the given input number. As the name suggests the column labelled *"nsplit"* is the number of split points in the tree from which the number of terminal nodes can easily be calculated as $nsplit + 1$. Therefore the full tree shown in Figure 3.5 has 62 terminal nodes i.e. $|T_0| = 62$. The next two columns in Figure 3.4 named *"rel error"* and *"xerror"* are scaled (on the unit interval) miss-classification rates and 10-fold cross validated miss-classification rates respectively with the last column being the standard deviation of the cross-validated miss-classification rates. It is straightforward to calculate the actual (un-scaled) miss-classification rates by simply multiplying the *"rel error"* and *"xerror"* columns by the *"root node error"* as these measurements have been scaled to the zero split tree.

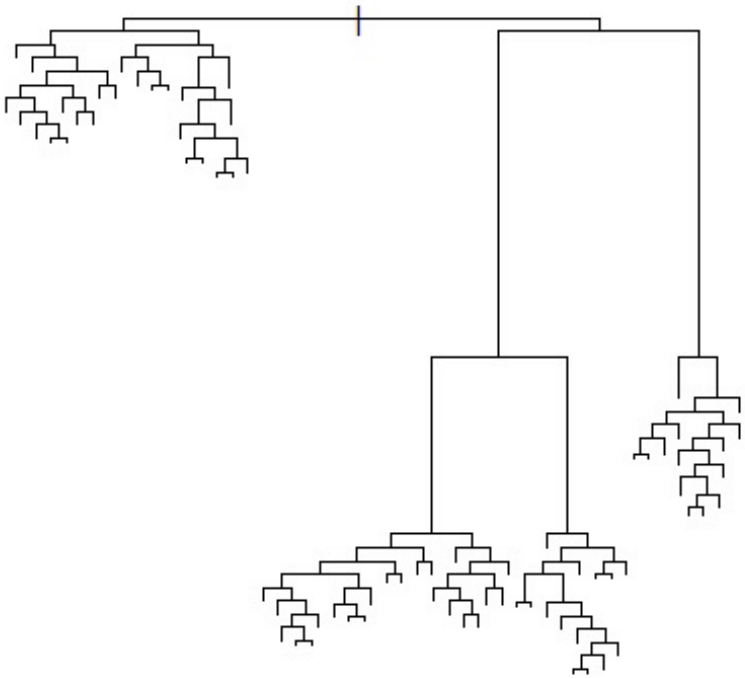


Figure 3.5: Dendogram showing the full tree T_0 based on the Example 1 applied to a binary classification problem

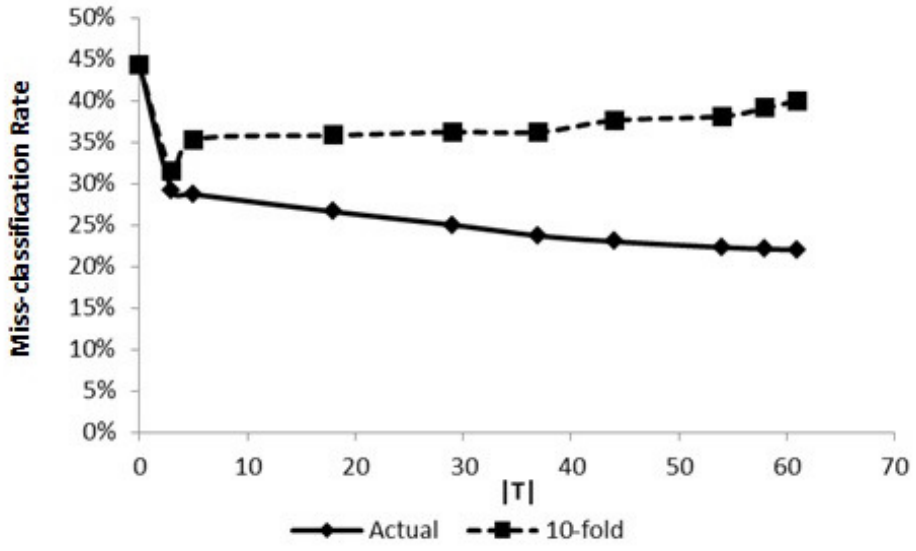


Figure 3.6: Graph showing the miss-classification rates for the simulated data in Example 1 applied to a binary classification problem as a function of tree size

Figure 3.6 shows the un-scaled miss-classification rates for the training or actual tree as well as using 10-fold cross validation. What is interesting to note is that the actual or training miss-classification rates begin to fall as the tree size increases and in fact drops below the 30% miss-classification level which if we recall was the pre-determined quantity of introduced error. When analyzing 10-fold cross-validated miss-classification rate Figure 3.6 shows that for tree sizes where $|T| > 4$ the cross-validated error begins to increase which can also be verified by the increase in Figure 3.4 under the "*xerror*" column. The combination of these two findings indicate that for trees larger than $|T| > 4$ the problem of over-fitting occurs. More importantly what this information tells us is at which point the tree should be pruned which in this

```
Classification tree:
rpart(formula = Y ~ X1 + X2, data = Data, method = "class", parms = list(split = "gini"),
    control = stoppingrule)

Variables actually used in tree construction:
[1] X1 X2

Root node error: 443/1000 = 0.443

n= 1000

         CP nsplit rel error  xerror     xstd
1 0.1117381      0   1.00000 1.00000 0.035459
2 0.0045147      3   0.65688 0.71106 0.033159
> |
```

Figure 3.7: R output using the *printcp()* command showing the pruned tree on simulated data as per Example 1 applied to a binary classification problem

case is where $nsplit = 3$. The pruning can be done in R by appending the following code below to Appendix A.1.

```
#Prune the full tree0 to the selected cp value where xerror is minimized

treealpha<-prune(tree0,cp=0.0045147)

#Plot and print the pruned tree

printcp(treealpha)

plot(treealpha,compress=TRUE,margin=0.2)

text(treealpha,all=TRUE,use.n=TRUE)
```

The R code to prune the tree requires the specific *cp* value at which to prune and hence the input of 0.0045147 in the *prune()* command. As we can see from Figure 3.7 we have now produced the pruned tree with $nsplit = 3$ (4 terminal nodes). We also note that the actual training miss-classification error (*rel error*) is $0.433 * 0.65688 = 0.28443$ and the 10-fold cross validated miss-classification error (*xerror*) is $0.433 *$

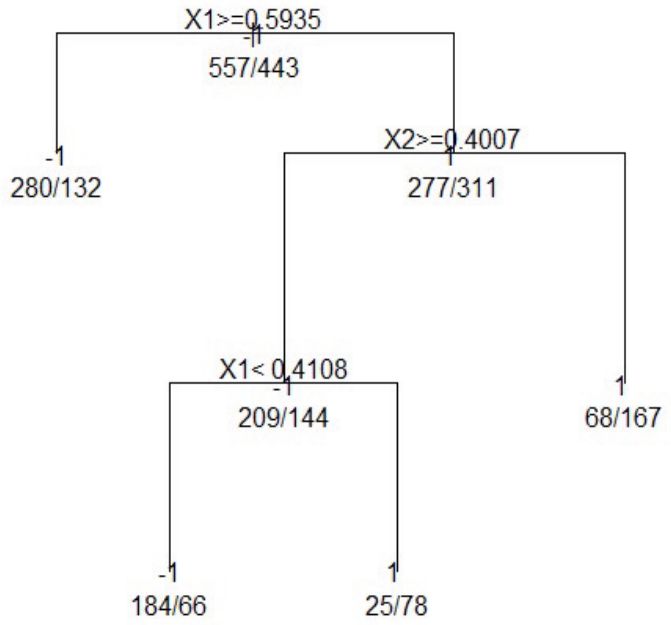


Figure 3.8: Dendogram of the pruned tree generated as per Example 1 applied to a binary classification problem

$0.71106 = 0.30789$ are both close to the original constructed error of 0.3 providing us with a degree of comfort that the tree is right-sized. Analyzing the tree in Figure 3.8 we first notice that structurally it resembles Figure 3.2. Extending the analysis we notice that the estimated split points are very near to those of the actual constructed split points of 0.4 and 0.6. Lastly and most importantly the classifications made within each region as shown in Figure 3.8 are compared to the original construct of the simulated binary data:

- R1:$\{Y = -1 : X_1 \geq 0.5935 \approx 0.6\}$ - this is comparable to R_5 and R_4 which both make a -1 classification
- R2:$\{Y = -1 : X_1 < 0.4108 \approx 0.4$ and $X_2 \geq 0.4007 \approx 0.4\}$ - this is comparable to R_2 which makes a -1 classification
- R3:$\{Y = 1 : 0.4 \approx 0.4108 \leq X_1 < 0.5935 \approx 0.6$ and $X_2 \geq 0.4007 \approx 0.4\}$ - this is comparable to R_3 which makes 1 classification
- R4:$\{Y = 1 : X_1 < 0.5935 \approx 0.6$ and $X_2 < 0.4007 \approx 0.4\}$ - this is comparable to R_1 which makes 1 classification

Therefore the regions generated by the pruned tree in Figure 3.8 map exactly back to the regions constructed in the simulated data set and in fact has gone a step further and simplified the original constructed tree by reducing the number of terminal nodes from 5 to 4 without any loss in accuracy.

3.4 Real binary classification example

We now apply the CART method to an actual real-life data set called **SPAM** as used in Hastie et al.[4] (p 295-366) which can be found at *www-stat.stanford.edu*. The data set comprises of 4601 observations or e-mails, which was part of a study to develop an automated machine learning method to screen for "spam" or "junk" e-mails. The dependent variable is binary with "spam" being coded as a 1 and 0 representing actual e-mails. The data was compiled by Hewlett-Packard laboratories and used 57 predictor variables to in a bid to develop a predictive model which where made up of:

- 48 predictor variables each measuring the percentage of times a particular word occurred in an e-mail such as **business, address, internet, free** etc.
- 6 predictor variables each measuring the percentage of times a particular non-alphanumeric character occurred in an e-mail i.e. **ch; , ch(, ch[, ch! , ch$** and **ch#**
- 1 predictor variable called **CAPAVE** measuring the average length of uninterrupted sequences of capital letters in an e-mail
- 1 predictor variable called **CAPMAX** measuring the length of the longest uninterrupted sequence of capital letters in an e-mail

- 1 predictor variable called **CAPTOT** measuring the sum of the length of all uninterrupted sequences of capital letters in an e-mail

The full tree T_0 can then be created by running the R code as given in Appendix A.2 resulting in the output shown in Figure.3.9. As before the tree growing procedure for a particular node is halted when a minimum node size of 5 is reached.

Analyzing Figure 3.9 we notice on of the salient benefits of the CART method in that it performs natural internal feature selection which even in the full tree T_0 reduced the number of predictor variables from 57 to 28. The table output in Figure 3.9 and the miss-classification graph in Figure 3.10 shows that the optimum tree size is reached where $|T| = 21$ as beyond this the cross-validated error rate no longer improves remaining at more or less this level. The results shown in Figure 3.10 also closely resembles the graph as shown in Figure 9.4 in Hastie et al.[4] (p 314) which showed the 10-fold cross validated error rate flattening out at between 8% and 9% hence the results shown here are to a degree consistent. However where Hastie et al.[4] (p 295-366) elected to prune the tree at $|T| = 17$ we will prune it at $|T| = 21$,as per our reason above, and therefore we will use the *prune()* command where $nsplit = 20$ and $cp = 0.0027579$. The pruned tree can be created and displayed by appending the R code below to Appendix A.2 which results in a 10-fold cross validated error rate of $0.39404 * 0.22173 = 8.737\%$ which is a fairly good result.

#Prune the full tree0 to the selected cp value where xerror is minimized

```
Classification tree:
rpart(formula = Y ~ ., data = spamdata, method = "class", parms = list(split = "gini"),
    control = stoppingrule)

Variables actually used in tree construction:
 [1] capital_run_length_average capital_run_length_longest capital_run_length_total
 [4] char_freq_dollar           char_freq_exclamation      char_freq_leftbracket
 [7] char_freq_semicolon        word_freq_1999             word_freq_650
[10] word_freq_business         word_freq_edu              word_freq_email
[13] word_freq_font             word_freq_free             word_freq_george
[16] word_freq_hp               word_freq_hpl              word_freq_internet
[19] word_freq_mail             word_freq_money            word_freq_our
[22] word_freq_over             word_freq_re               word_freq_remove
[25] word_freq_technology       word_freq_will             word_freq_you
[28] word_freq_your

Root node error: 1813/4601 = 0.39404

n= 4601

           CP nsplit rel error  xerror      xstd
1  4.7656e-01      0   1.00000 1.00000 0.0182819
2  1.4892e-01      1   0.52344 0.54992 0.0154140
3  4.3023e-02      2   0.37452 0.46111 0.0144265
4  3.0888e-02      4   0.28847 0.31715 0.0123722
5  1.0480e-02      5   0.25758 0.28682 0.0118457
6  8.2736e-03      6   0.24710 0.27689 0.0116645
7  7.1704e-03      7   0.23883 0.26255 0.0113944
8  5.2951e-03      8   0.23166 0.25152 0.0111795
9  4.4126e-03     14   0.19581 0.23938 0.0109354
10 3.5852e-03     15   0.19140 0.22780 0.0106944
11 3.3094e-03     19   0.17705 0.22118 0.0105529
12 2.7579e-03     20   0.17375 0.21732 0.0104691
13 2.2063e-03     24   0.16271 0.21566 0.0104329
14 1.6547e-03     34   0.14065 0.20960 0.0102985
15 1.3789e-03     37   0.13569 0.20684 0.0102366
16 1.2870e-03     39   0.13293 0.20574 0.0102117
17 1.1031e-03     42   0.12907 0.20463 0.0101867
18 8.2736e-04     51   0.11914 0.20132 0.0101111
19 7.3543e-04     53   0.11748 0.19581 0.0099834
20 6.8946e-04     59   0.11307 0.19746 0.0100220
21 5.5157e-04     63   0.11031 0.19746 0.0100220
22 3.3094e-04     65   0.10921 0.20243 0.0101364
23 2.7579e-04     75   0.10535 0.20243 0.0101364
24 6.1286e-05     77   0.10480 0.20408 0.0101742
25 0.0000e+00     86   0.10425 0.20574 0.0102117
```

Figure 3.9: R output showing the full tree T_0 trained on the **SPAM** data using 10-fold cross-validation

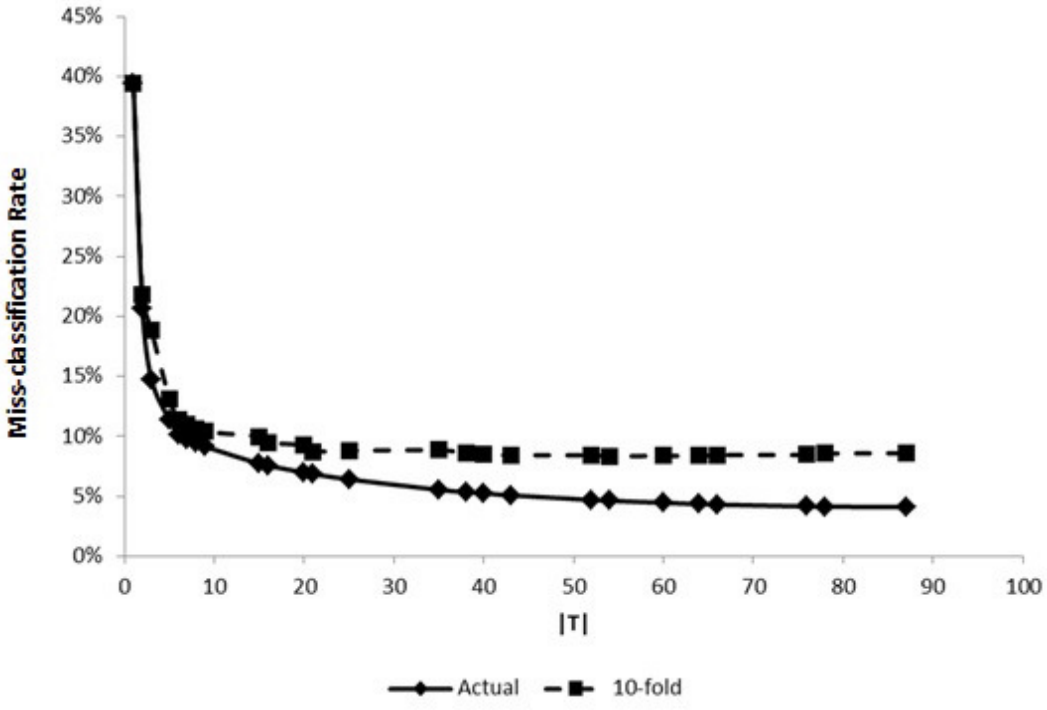


Figure 3.10: Graph showing the actual or training miss-classification rate and 10-fold cross-validation error rate for the **SPAM** dataset as a function of the tree size $|T|$

```
Variables actually used in tree construction:
 [1] capital_run_length_average capital_run_length_longest
 [3] capital_run_length_total   char_freq_dollar
 [5] char_freq_exclamation      char_freq_semicolon
 [7] word_freq_1999             word_freq_edu
 [9] word_freq_font             word_freq_free
[11] word_freq_george           word_freq_hp
[13] word_freq_money            word_freq_our
[15] word_freq_remove           word_freq_you

Root node error: 1813/4601 = 0.39404

n= 4601

          CP nsplit rel error  xerror     xstd
1  0.4765582      0   1.00000 1.00000 0.018282
2  0.1489244      1   0.52344 0.55378 0.015453
3  0.0430226      2   0.37452 0.47876 0.014637
4  0.0308880      4   0.28847 0.33315 0.012635
5  0.0104799      5   0.25758 0.28902 0.011885
6  0.0082736      6   0.24710 0.27799 0.011685
7  0.0071704      7   0.23883 0.26806 0.011500
8  0.0052951      8   0.23166 0.26310 0.011405
9  0.0044126     14   0.19581 0.25317 0.011212
10 0.0035852     15   0.19140 0.24104 0.010969
11 0.0033094     19   0.17705 0.23442 0.010833
12 0.0027579     20   0.17375 0.22173 0.010565
> |
```

Figure 3.11: R output showing the prunned tree T_α

```
#in this case where cp=0.0027579

treealpha<-prune(tree0,cp=0.0027579)

#Plot and print the pruned tree

printcp(treealpha)

plot(treealpha,uniform=TRUE,branch=0,margin=0.015)

text(treealpha,all=TRUE,use.n=TRUE)
```

From Figure 3.12 we notice the similarity in shape to the dendogram in Figure 9.5 shown in Hastie et al.[4] (p 315). Analyzing the tree in Figure 3.12 from the

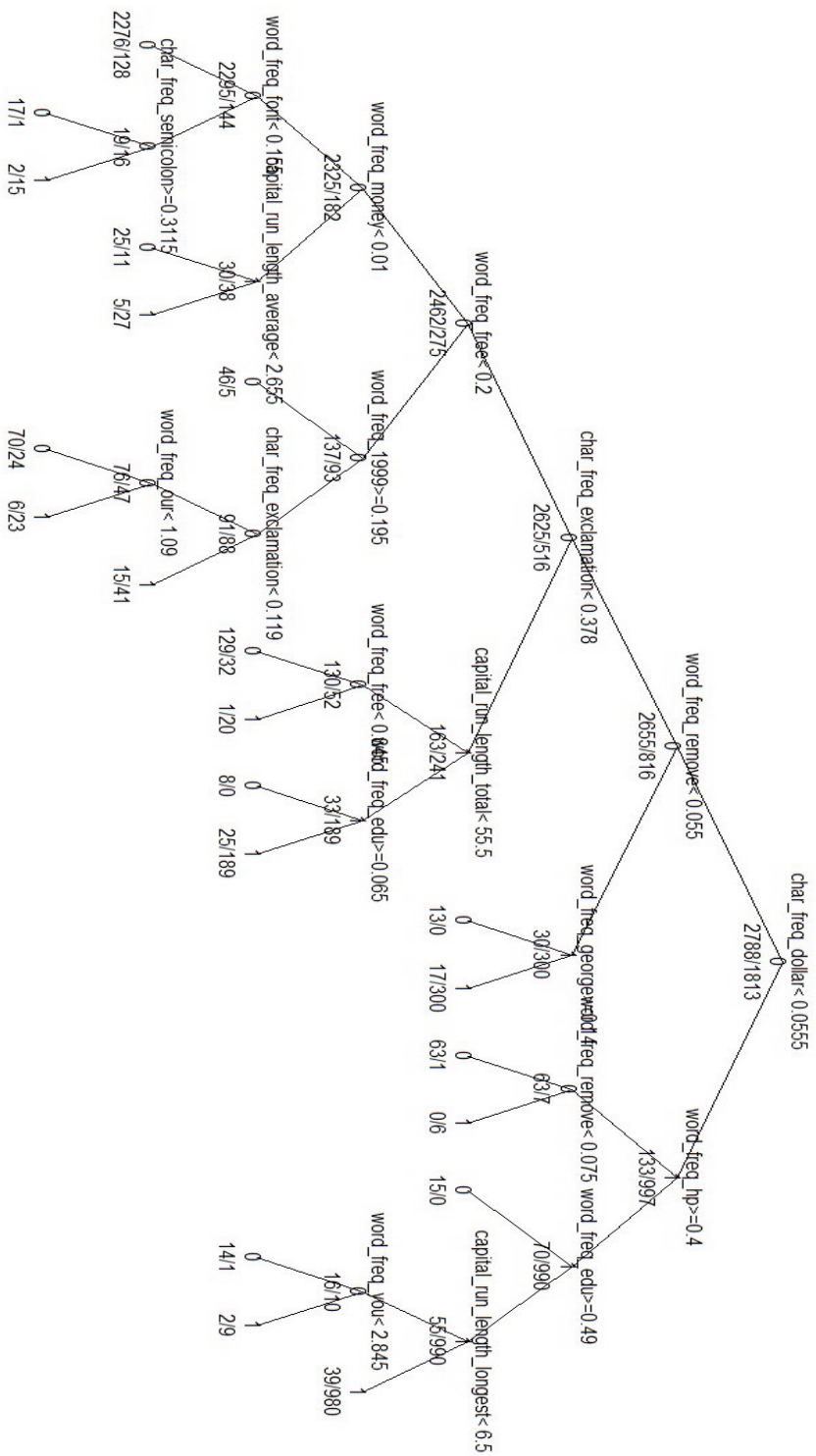


Figure 3.12: Dendogram of the prunned tree trained on the **SPAM** data set

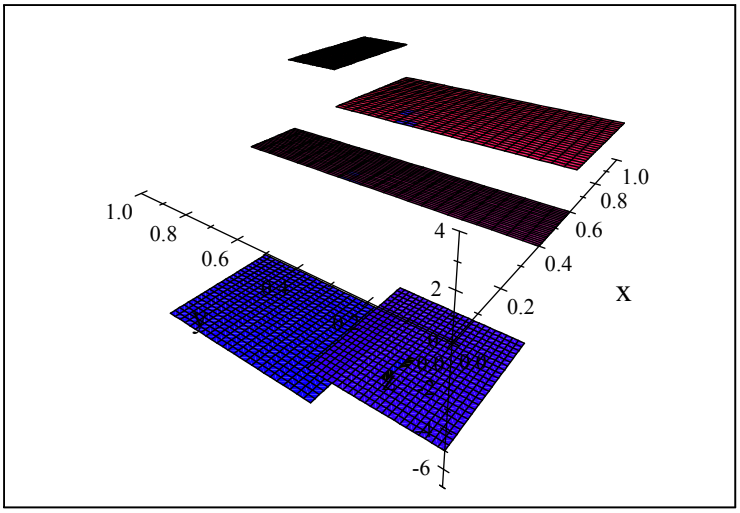
top-down we notice both trees use the dollar character **char_freq_dollar** as the first split variable with the same corresponding split point and likewise uses the variables **REMOVE** and **hp** with approximately the same split points in the next level down. The trend of similarities continues as one moves down the tree dendogram and therefore we are relatively comfortable that we have created the optimum classification tree for the **SPAM** data. A possible reason for the slight deviance between the two trees could be attributed to different impurity measures and/or tree optimization procedures being employed however this would require further analysis which is beyond the scope of this dissertation.

3.5 Simulated multi-classification example

The extension of CART to miss-classification problems i.e. when $K > 2$ is straightforward as we have set out in Section 3.2. However one does need to be cognizant that certain draw-backs could arise such as for large number of classes $K \gg$ the computational time can increase by a factor of K where numerical optimization is not possible. In addition the requirement for a larger training data sets becomes increasingly more important as there are now N/K possible observations per class and this decreases as K increases. A simulated example using CART on a multi-classification problem will now be shown.

We shall once again make use of the earlier simulated dataset as in Section 3.3 however instead of coding the regions R as either 1 or -1 we shall use the same

region coding as given in Figure 9.2 of Hastie et al.[4] (p 306) or more specifically set $K = 5$ with $k(1) = -5, k(2) = -7, k(3) = 0, k(4) = 2, k(5) = 4$. A visual representation of the dataset structure is shown below. At this point we note that these classes are numeric whereas the primary purpose of classification trees is to deal with non-numeric response variables however we have chosen to use numeric values for graphical representation and could have just as easily assigned each region with alphanumeric or non-alphanumeric characters.



Graph showing the multi-classification structure of the simulated dataset as per Example 1 and Hastie et al.[4] (pg 306).

The R code shown in Appendix A.3 generates the simulated multi-classification dataset with random error added to 30% of the data. We also generate the full tree

```
Variables actually used in tree construction:
[1] X1 X2

Root node error: 728/1000 = 0.728

n= 1000

           CP nsplit rel error  xerror     xstd
1  0.16552198      0   1.00000 1.00000 0.019329
2  0.15521978      2   0.66896 0.78434 0.021499
3  0.10164835      3   0.51374 0.51374 0.021018
4  0.00343407      4   0.41209 0.41209 0.019906
5  0.00320513      6   0.40522 0.44093 0.020279
6  0.00274725     11   0.38462 0.47115 0.020620
7  0.00240385     13   0.37912 0.47940 0.020705
8  0.00228938     22   0.35302 0.48901 0.020799
9  0.00192308     25   0.34615 0.50275 0.020924
10 0.00137363     33   0.32692 0.51923 0.021062
11 0.00103022     47   0.30082 0.52060 0.021073
12 0.00091575     51   0.29670 0.51511 0.021029
13 0.00068681     58   0.28434 0.51511 0.021029
14 0.00045788     62   0.28159 0.52885 0.021137
15 0.00034341     70   0.27747 0.52885 0.021137
16 0.00000000     74   0.27610 0.52885 0.021137
>
```

Figure 3.13: R output showing the the full tree T_0 trained on simulated multi-classification data as per Example 1

T_0 based on the minimum node size rule of 5 and assess at which point based on the 10-fold cross-validated error rate the tree would need to be pruned.

From Figure 3.13 and Figure 3.14 we notice that the relative or scaled 10-fold cross validated error is minimized at $nsplit = 4$ and therefore $|T| = 5$ which is the number of terminal nodes by design. Note the *plotcp()* command in R is used to generate Figure 3.14. The actual 10-fold cross validated error at $nsplit = 4$ can be

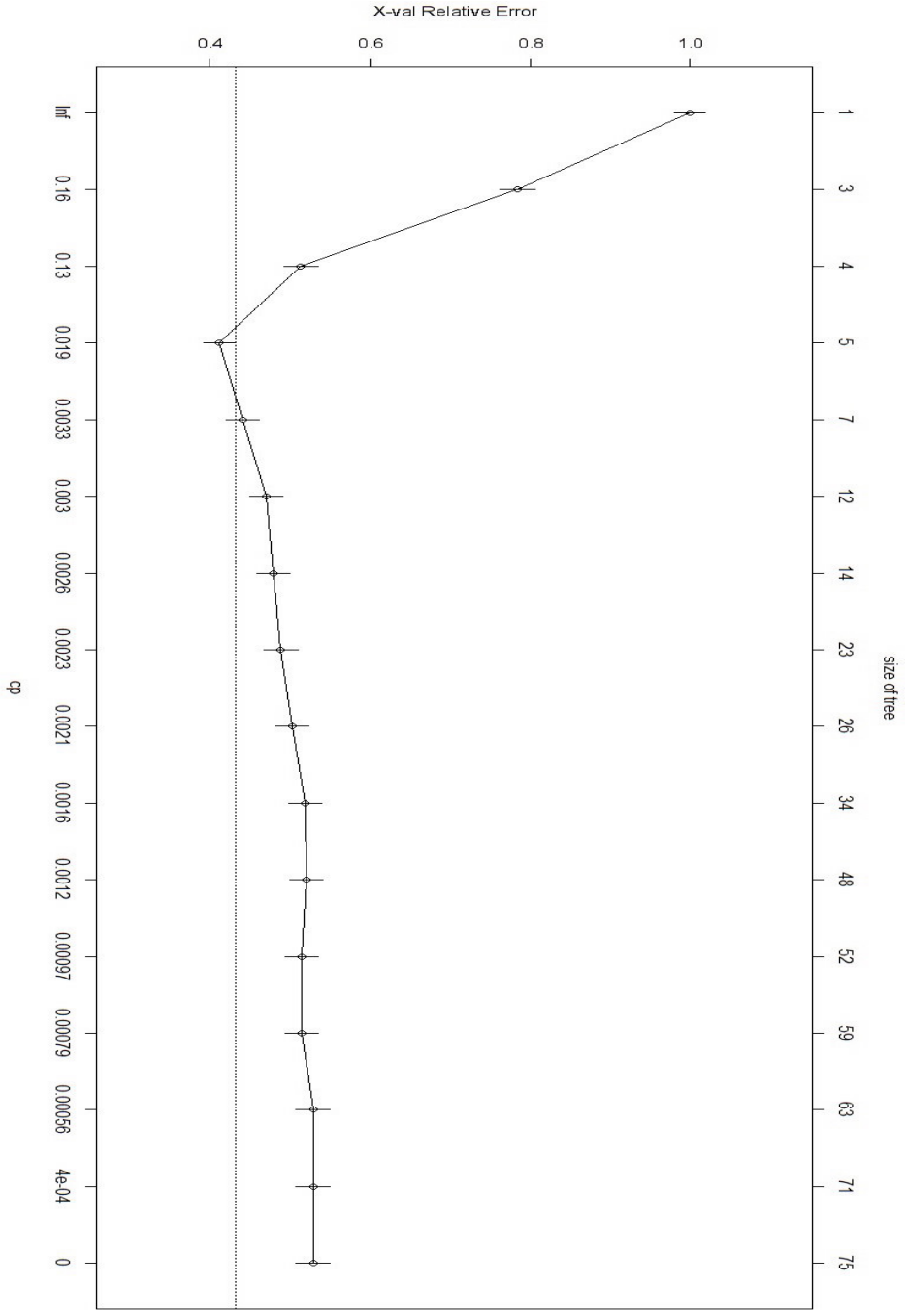


Figure 3.14: Graph showing releative or scaled 10-fold cross validated error as a function of the tree size $|T|$ and complexity parameter α of the simulated multi-classification problem based on Example 1

obtained by multiplying *xerror* with the *"root node error"*, which in this case would be the aggregate of all the other class observations excluding observations belonging to the modal class of the entire data set. We therefore calculate the 10-fold cross validated error rate $0.41209 * 0.728 = 0.3$ which as we know is the simulated pre-determined miss-classification rate. We then prune the tree at the appropriate *cp* value and plot the resultant dendogram by appending the R code below to Appendix A.3.

```
#Prune the full tree0 to the selected cp value where xerror is minimized
treealpha<-prune(tree0,cp=0.00343407)
#Plot the pruned tree
plot(treealpha,compress=TRUE,margin=0.2)
text(treealpha,all=TRUE,use.n=TRUE)
```

From Figure 3.15 we notice the terminal nodes provide classifications for all five predefined classes. As a note the number of observations per class in Figure 3.15 are given in increasing sequential order i.e. $-7/-5/0/2/4$. Analyzing the regions further we get:

- R_1:$\{Y = -5 : X_1 < 0.4$ and $X_2 < 0.4031 \approx 0.4\}$
- R_2:$\{Y = -7 : X_1 < 0.4$ and $X_2 \geq 0.4031 \approx 0.4\}$
- R_3:$\{Y = 0 : X_1 \geq 0.4$ and $X_1 < 0.6002 \approx 0.6\}$
- R_4:$\{Y = 2 : X_1 > 0.6$ and $X_2 < 0.8002 \approx 0.8\}$

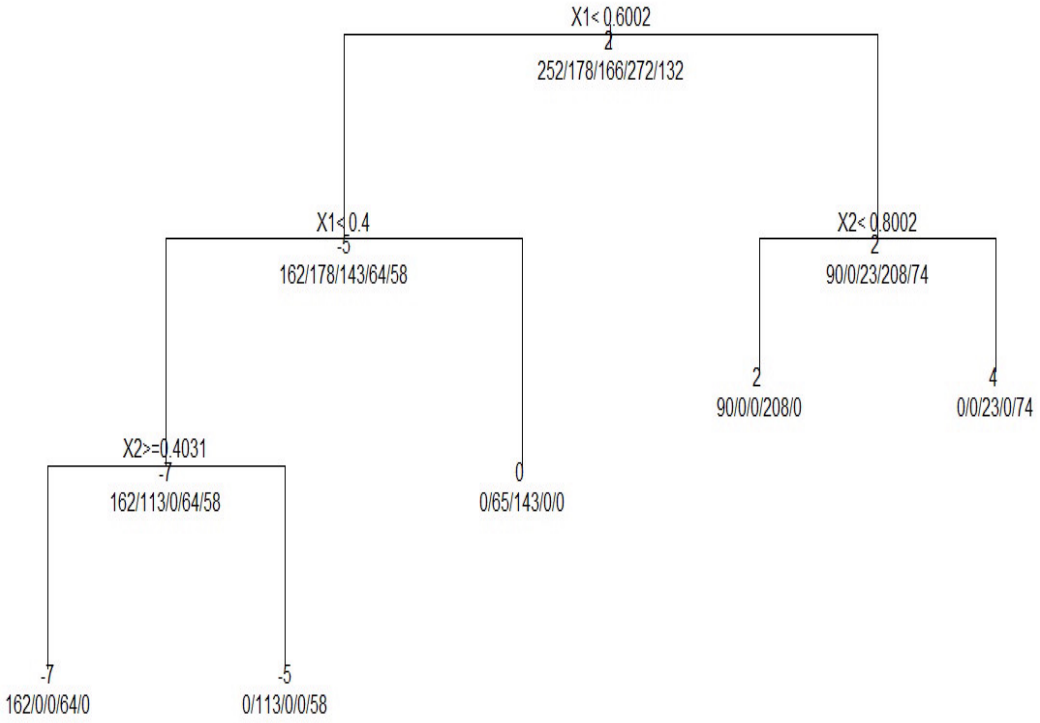


Figure 3.15: Dendogram of the prunned tree based on simulated multi-classification data as per Example 1

- R_5:$\{Y = 4 : X_1 < 0.4$ and $X_2 \geq 0.8002 \approx 0.8\}$

All the regions shown above are almost an exact match to the original structure of the simulated data aside from the boundary classifications and therefore we can conclude that CART, in addition to solving binary problems, also works well when faced with multi-classification problems.

3.6 Real multi-classification example

We now turn to a real-life example of a multi-classification problem similar to the OCR text recognition problem mentioned earlier. The data set chosen is taken from the US Postal services as in Hastie et al. [4] (p 295-366) which consist of scanned images of handwritten ZIP codes with each data point or observation representing a single number. Each scanned digit is a represented using a 16x16 bit greyscale image. The idea being to be able to develop an automated model which can quickly and accurately classify each scanned handwritten digit as a number (0,1,2...,9) and therefore negate the need for manual sorting. The training dataset comprises of 7291 observations or scanned digits with 256 predictor variables representing the 16x16 bit image and coded on a scale from -1 to 1 with -1 values representing "white space" pixels and 1 representing "dark space" or "marked" pixels. The full tree T_0 is built and pruned to find T_α by running the R code in Appendix A.4.

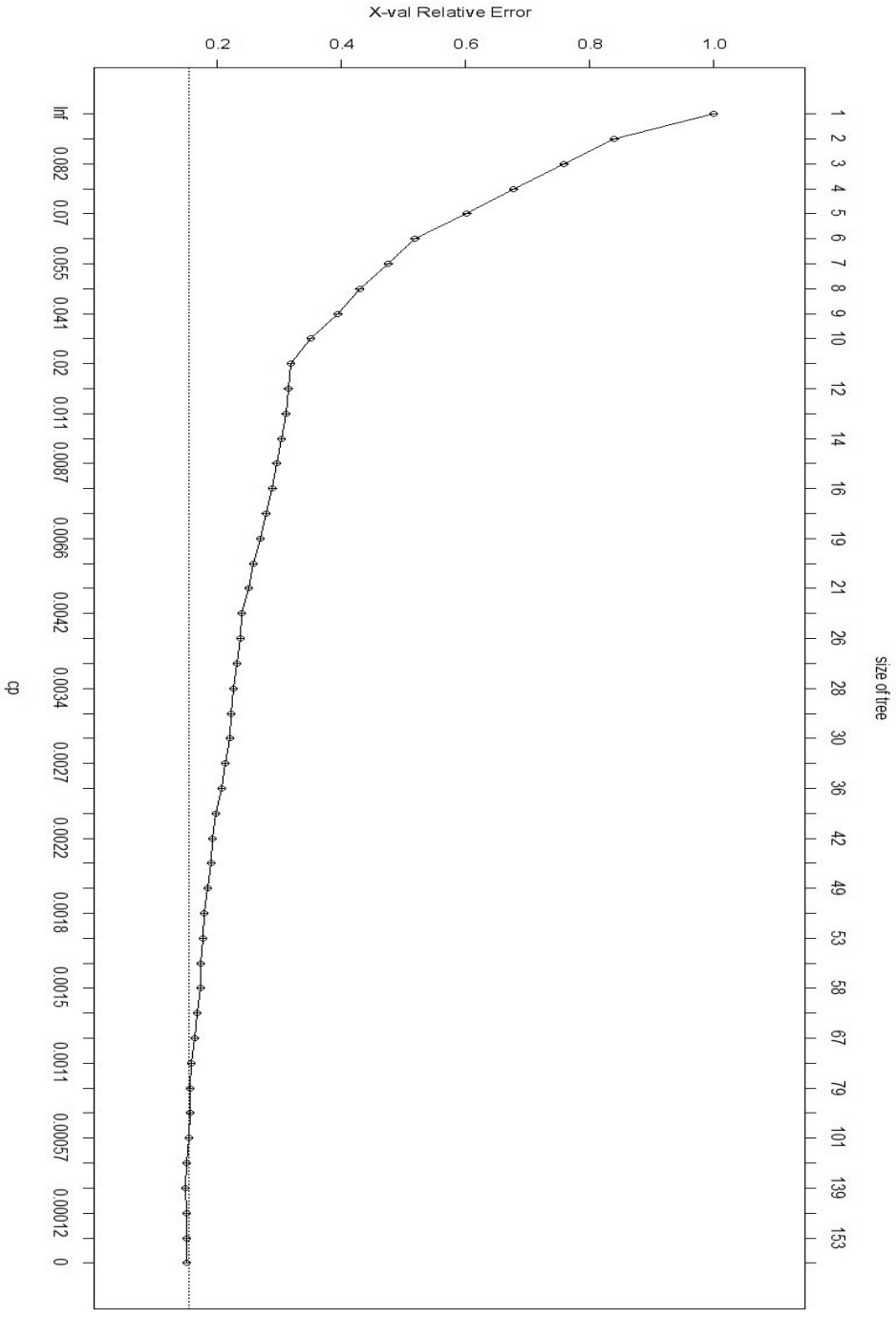


Figure 3.16: *CP* plot of T_0 trained on the **digits** data

```
Classification tree:
rpart(formula = Y ~ ., data = digitsdata, method = "class", parms = list(split = "gini"),
    control = stoppingrule)

Variables actually used in tree construction:
  [1] X10  X100 X101 X102 X103 X105 X109 X116 X118 X120 X121 X122 X123 X13  X130
 [16] X131 X132 X133 X134 X136 X137 X139 X14  X140 X142 X145 X149 X151 X154 X157
 [31] X158 X161 X162 X164 X165 X166 X167 X168 X169 X171 X173 X177 X178 X179 X180
 [46] X181 X182 X183 X184 X186 X188 X198 X199 X200 X204 X208 X21  X210 X211 X213
 [61] X214 X216 X218 X222 X228 X23  X230 X233 X234 X237 X238 X239 X24  X246 X248
 [76] X249 X25  X27  X28  X36  X37  X40  X43  X5   X56  X58  X6   X60  X62  X69
 [91] X7   X71  X72  X74  X76  X78  X79  X8   X84  X85  X87  X88  X89  X9   X90
[106] X91  X92  X93  X99

Root node error: 6097/7291 = 0.83624

n= 7291
```

Figure 3.17: R output of the full tree T_0 trained on the **digits** data

Based on Figure 3.16, Figure 3.17 and Figure 3.18 we see that the 10-fold relative cross validated error flattens out around $|T| = 79$ at an error rate of $0.83624 * 0.15598 = 13.044\%$. We therefore elect to prune the tree at this point calling the R code below and appending it to Appendix A.4.

#Prune the full tree0 to the selected cp value where xerror is minimized or in this case where the error rate flattens i.e. where cp=0.00082008

treealpha<-prune(tree0,cp=0.00082008)

printcp(treealpha)

plot(treealpha,uniform=TRUE)

text(treealpha,splits=FALSE,all=FALSE)

From a cursory analysis of Figure 3.19 we note that all the digits (0,1,2,...,9) have a classification mapping and therefore combined with the low cross-validated

	CP	nsplit	rel error	xerror	xstd
1	1.5926e-01	0	1.000000	1.00000	0.0051826
2	8.7092e-02	1	0.840741	0.84074	0.0063989
3	7.7579e-02	2	0.753649	0.75939	0.0067422
4	7.5283e-02	3	0.676070	0.67804	0.0069393
5	6.5278e-02	4	0.600787	0.60259	0.0070022
6	6.3966e-02	5	0.535509	0.51894	0.0069411
7	4.6908e-02	6	0.471543	0.47614	0.0068556
8	4.1824e-02	7	0.424635	0.42939	0.0067185
9	3.9528e-02	8	0.382811	0.39495	0.0065866
10	3.0999e-02	9	0.343284	0.35083	0.0063765
11	1.2957e-02	10	0.312285	0.31901	0.0061939
12	1.1481e-02	11	0.299328	0.31442	0.0061652
13	1.1153e-02	12	0.287846	0.31032	0.0061391
14	9.0208e-03	13	0.276693	0.30343	0.0060942
15	8.3648e-03	14	0.267673	0.29539	0.0060399
16	7.3807e-03	15	0.259308	0.28916	0.0059965
17	7.2167e-03	16	0.251927	0.27833	0.0059182
18	6.0686e-03	18	0.237494	0.27030	0.0058577
19	5.4125e-03	19	0.231425	0.25750	0.0057567
20	4.4284e-03	20	0.226013	0.25045	0.0056986
21	3.9364e-03	21	0.221584	0.23913	0.0056016
22	3.7723e-03	25	0.205839	0.23766	0.0055886
23	3.6083e-03	26	0.202067	0.23274	0.0055447
24	3.1163e-03	27	0.198458	0.22650	0.0054876
25	2.9523e-03	28	0.195342	0.22322	0.0054569
26	2.7883e-03	29	0.192390	0.21994	0.0054258
27	2.6242e-03	30	0.189601	0.21355	0.0053638
28	2.2962e-03	35	0.176480	0.20748	0.0053033
29	2.2142e-03	39	0.167295	0.19813	0.0052069
30	2.1322e-03	41	0.162867	0.19272	0.0051492
31	1.9682e-03	44	0.156470	0.18993	0.0051190
32	1.8042e-03	48	0.148598	0.18452	0.0050590
33	1.7222e-03	50	0.144989	0.17845	0.0049901
34	1.6402e-03	52	0.141545	0.17763	0.0049806
35	1.5581e-03	55	0.136625	0.17320	0.0049288
36	1.4761e-03	57	0.133508	0.17238	0.0049191
37	1.3121e-03	62	0.126128	0.16812	0.0048680
38	1.1481e-03	66	0.120879	0.16402	0.0048178
39	9.8409e-04	73	0.112842	0.15811	0.0047438
40	8.2008e-04	78	0.107922	0.15598	0.0047166
41	6.5606e-04	85	0.102181	0.15631	0.0047208
42	4.9205e-04	100	0.092340	0.15434	0.0046954
43	3.2803e-04	121	0.082008	0.15024	0.0046417
44	1.6402e-04	138	0.076431	0.14925	0.0046287
45	8.2008e-05	150	0.074463	0.15024	0.0046417

Figure 3.18: Second part of the R output of the full tree T_0 trained on the **digits** data

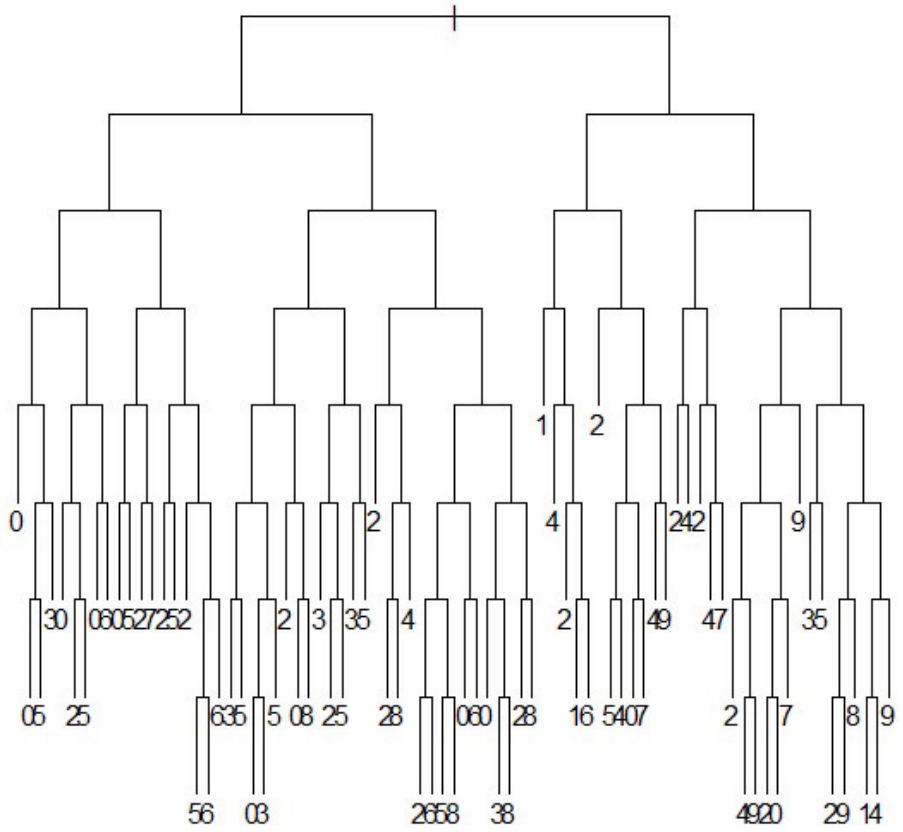


Figure 3.19: Dendogram showing the prunned tree T_α trained on the **digits** data

error rate of 13.04% provides us with a degree of comfort that this model may be appropriate. The creators of the **digits** data also provided a test set which comprised of a further 2007 observations or digits. The R code below, which should be appended in sequential order to the earlier code, calculates the test error which produced a slightly higher error rate of 17.04% verses the 10-fold cross validated error rate based on the training set of 13.04%.

```
#calculate the test error
errordata<-data.frame(Actual= digitstestdata[,1],Predicted=testprediction)
temp<-ifelse(errordata[,1]!=errordata[,2],1,0)
testerror=sum(temp)/nrow(errordata)
print(testerror)
#output shown as
[1] 0.1704036
```

What we have shown in this chapter is that although classification trees can produce relatively acceptable results from an accuracy point of view they tend to suffer from the problem of over-fitting for large trees. In the next chapter we will use the same four datasets as used in this chapter i.e. simulated binary classification, real binary classification, simulated multi-classification and real multi-classification, to determine if boosting using Adaboost produces better results than the single classification trees constructed here.

Chapter 4
Application of Adaboost

In this chapter we will focus our attention on the application of Adaboost using classification trees as a base predictor. As in Chapter 3 the first set of applications will be in the simulated case exploring both a binary and multi-classification problem. The next set of applications is where the datasets are real and once again where we are faced with a binary and multi-classification problem. We will also test to see if boosting using *stumps* produces results comparable or better than large trees and what happens when these large trees themselves are boosted. The metric of interest that we will be analyzing will be the test and training error rates of the final predictions as a function of boosting iterations to determine to what extent boosting using different base classifiers improves the previous results as seen in Chapter 3.

4.1 Illustration of the effectiveness of Adaboost

The simulated example given in Hastie et al. [4] (p 339-340) will be used to demonstrate the effectiveness of Adaboost. The simulated data is structured such that the input variables $X_1, X_2, ..., X_{10}$ are generated from a standard independent Gaussian $N(0,1)$ distribution and the output variable Y is defined as

$$Y = \begin{cases} 1 \text{ if } \sum_{j=1}^{10} X_j^2 > \chi_{10}^2(0.5) \\ -1 \text{ otherwise} \end{cases} \tag{4.1}$$

where $\chi^2_{10}(0.5)$ is the median of a chi-squared random variable with 10 degrees of freedom and is selected because of how the independent variable are pooled together i.e. sum of squares of 10 standard Gaussian variables. The choice of the median in the construction of this simulated dataset has the effect that any single learner trained on this data will be at most regarded as a *weak learner* possessing an accuracy rate only marginally better than random guessing. Therefore for a large enough number of simulated observations, due to the fact that we have 10 predictor variables, this dataset should be an ideal candidate for boosting. Based on structure as given in equation (4.1) we shall generate a training data set consisting of 2000 observations and a test data set consisting of 10000 observations. We will then use the *CART* method to grow the full tree T_0 and compare the test error against the test error produced using *boosted stumps*. The R code shown in Appendix B.1 uses a relatively new Adaboost package in R called *Ada* which we customize to run the **Adaboost.M1** algorithm.

From Figure 4.1 we see that the full tree T_0 has 233 terminal nodes with a 10-fold cross validated error rate of $0.56557 * 0.4995 = 28.25\%$. The R code in Appendix B.1 outputs the test error rate of 26% for the full tree T_0 and 46% for the stump. Both these test error rates are shown as the solid lines in Figure 4.2. We notice from Figure 4.2 that after around the 50^{th} boosting iteration the **Adaboost.M1** method begins to outperform the full tree T_0 and at $M = 400$ produces a test error of $1 - 0.882 = 11.8\%$ as shown in Figure 4.3 by calling the *summary()* command

```
Classification tree:
rpart(formula = Y ~ ., data = datatrain, method = "class", parms = list(split = "gini"),
    control = stoppingrule)

Variables actually used in tree construction:
 [1] X1  X10 X2  X3  X4  X5  X6  X7  X8  X9

Root node error: 999/2000 = 0.4995

n= 2000

           CP nsplit rel error  xerror     xstd
1  0.11061061      0  1.000000 1.05606 0.022349
2  0.08508509      2  0.778779 0.83784 0.022084
3  0.05705706      3  0.693694 0.74474 0.021637
4  0.03903904      4  0.636637 0.69870 0.021338
5  0.02602603      6  0.558559 0.65165 0.020976
6  0.02502503      7  0.532533 0.62563 0.020750
7  0.02002002      8  0.507508 0.60460 0.020553
8  0.01601602      9  0.487487 0.58659 0.020375
9  0.01501502     10  0.471471 0.58458 0.020354
10 0.01101101     12  0.441441 0.57858 0.020292
11 0.00633967     14  0.419419 0.53954 0.019863
12 0.00600601     17  0.400400 0.53453 0.019804
13 0.00533867     18  0.394394 0.52553 0.019697
14 0.00500501     21  0.378378 0.52553 0.019697
15 0.00467134     24  0.363363 0.52853 0.019733
16 0.00420420     27  0.349349 0.52553 0.019697
17 0.00400400     32  0.328328 0.51752 0.019599
18 0.00350350     38  0.304304 0.51852 0.019611
19 0.00325325     40  0.297297 0.51451 0.019562
20 0.00300300     46  0.275275 0.50851 0.019487
21 0.00250250     59  0.235235 0.50551 0.019448
22 0.00200200     68  0.211211 0.50551 0.019448
23 0.00166834     90  0.166166 0.50450 0.019436
24 0.00160160     93  0.161161 0.51552 0.019574
25 0.00150150    101  0.147147 0.51451 0.019562
26 0.00125125    116  0.118118 0.52553 0.019697
27 0.00100100    120  0.113113 0.52653 0.019709
28 0.00075075    204  0.025025 0.55856 0.020078
29 0.00050050    209  0.021021 0.55756 0.020067
30 0.00033367    229  0.011011 0.56156 0.020111
31 0.00000000    232  0.010010 0.56557 0.020154
> |
```

Figure 4.1: Output showing the full tree T_0 grown using the simulated gaussian data as per equation 4.1

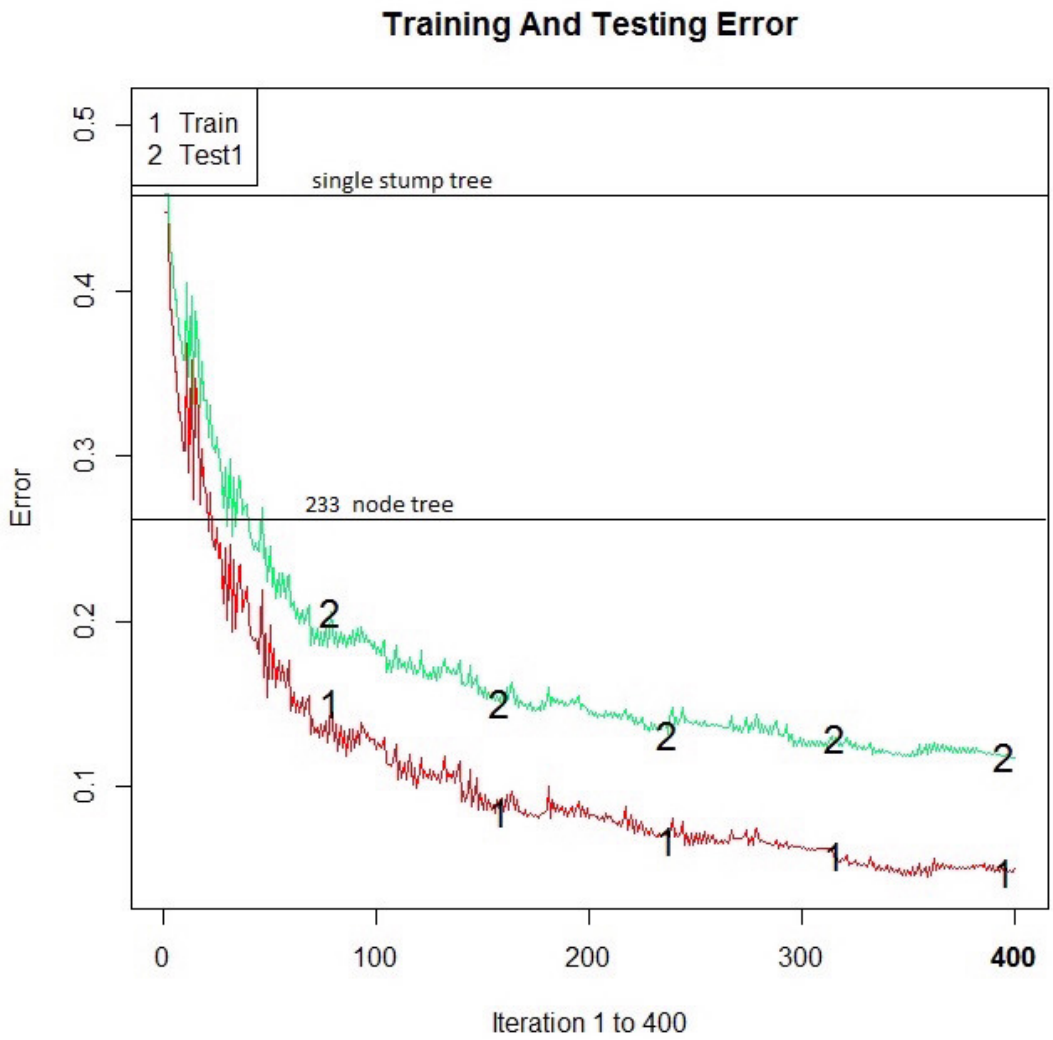


Figure 4.2: Graph showing the training error and test error of tree stumps boosted using Adaboost on the simulated data as per equation 4.1. The lines entitled *single stump* and *232 node tree* represent the test errors

```
> summary(adaboost,n.iter=M)
Call:
ada(Y ~ ., data = datatrain, iter = M, loss = "e", type = "discrete",
    control = stumprule, bag.frac = 1, nu = 1, test.x = datatest[,
        -1], test.y = datatest[, 1])

Loss: exponential Method: discrete   Iteration: 400

Training Results

Accuracy: 0.95 Kappa: 0.899

Testing Results

Accuracy: 0.882 Kappa: 0.765
```

Figure 4.3: R output showing the error rates for the boosted stumps on simulated data as per Equation 4.1 at $M = 400$

in R. Hence Adaboost using stumps has provided a four times improvement over the single tree stump and a two times improvement when compared to the full tree T_0. It is reaffirming to note that the results shown here correlate to the findings as given in Hastie et al. [4] (p 339-340).

4.2 Simulated boosted binary classification example

We now analyze the first example given in Section 3.3 whereby we simulated a two variable input space on the unit interval with a binary response as given in Hastie et al. [4] (p 305-317). In our analysis we will compare the effectiveness of Adaboost using boosted stumps against the optimized full tree or *pruned tree* shown in Section 3.3. The R code used to perform the comparison is shown in Appendix B.2. Note the simulated data is re-generated to ensure consistency in comparing the results.

As a note the full tree T_0 is pruned based on the 10-fold cross validated output of *printcp()* under the column *xerror* shown in Figure 4.4 After the full tree is pruned at $CP = 0.00434783$ the resultant pruned tree is then run on the 10000 simulated observations of test data which produces a test error of 30.71% (the variable *fulltreetesterror* outputs this error rate in R as shown in Appendix B.2). The same process is run for the tree stump with a test error of 39.4% (the variable *stumptreetesterror* outputs this error rate in R as shown in Appendix B.2). It is at this point we note that boosting the tree stump is unlikely to beat the performance of the pruned tree with an error rate already close to the introduced population error of 30% however what we are interested in is to what extent boosting will enhance the accuracy of the single stump tree. We can see from Figure 4.5 that after less than 10 boosting iterations the test error rate drops from the single stump error rate of 39.4% to less than 34% however remains above the error rate of the pruned tree (note this is not shown in Figure 4.5 as it falls below the x-axis). We also notice that the test error rate jumps upwards marginally between the 10^{th} and 80^{th} boosting iterations however stabilizes for iterations greater than $m = 80$ which reaffirms to a degree the anomaly that boosting when run for numerous rounds does not suffer from the problem of over-fitting as we have seen with CART.

Following the improvements seen using boosted stumps it would be of interest to test what performance enhancement could result if we elected to boost the pruned tree T_α. This analysis can quite easily be performed by appending the R code below

```
Classification tree:
rpart(formula = Y ~ X1 + X2, data = traindata, method = "class",
    parms = list(split = "gini"), control = stoppingrule)

Variables actually used in tree construction:
[1] X1 X2

Root node error: 460/1000 = 0.46

n= 1000

           CP nsplit rel error  xerror     xstd
1  0.10652174      0   1.00000 1.00000 0.034262
2  0.00652174      3   0.68043 0.73261 0.032495
3  0.00434783      5   0.66739 0.72826 0.032447
4  0.00326087     13   0.63043 0.76304 0.032811
5  0.00304348     19   0.61087 0.78696 0.033037
6  0.00289855     39   0.51087 0.79348 0.033096
7  0.00217391     42   0.50217 0.79348 0.033096
8  0.00130435     45   0.49565 0.83696 0.033451
9  0.00108696     50   0.48913 0.89783 0.033848
10 0.00072464     64   0.46957 0.90217 0.033872
11 0.00036232     67   0.46739 0.90217 0.033872
12 0.00000000     73   0.46522 0.90217 0.033872
```

Figure 4.4: R output of the full tree grown on simulated binary data as given in Section 3.3

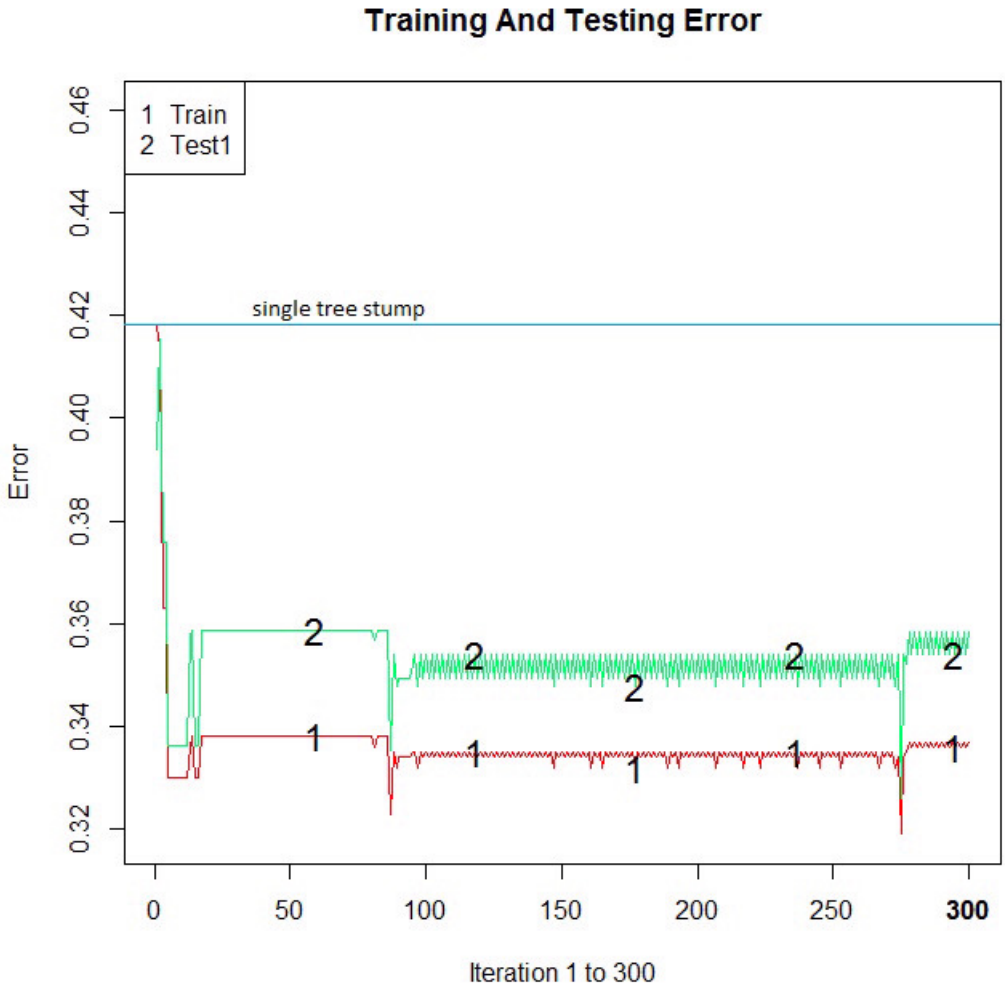


Figure 4.5: Test and training error of the boosted stump generated on simulated binary data as per Section 3.3

to Appendix B.2. Note the *cp* value selected for the pruned tree is based on Figure 4.4 .

```
#calculate the boosted alpha tree
stoprule=rpart.control(cp=0.005)
adaboost<-ada(Y~.,data=traindata,iter=M,loss="e",type="discrete",
control=stoprule,bag.frac=1,nu=1, test.x=testdata[,-1],test.y=testdata[,1])
print(adaboost)
plot(adaboost,FALSE,TRUE)
```

Figure 4.6 shows that after the first few boosting iterations the training error of the boosted pruned tree drops below the true population error rate. This can be explained by the fundamental principal of boosting in that after each successive boosting iterations miss-classified observations are given more training weight in subsequent classification models and therefore the boosted model additively caters for those miss-classified observations. When looking at the test error in Figure 4.6 we notice that after the 50^{th} boosting iteration the test error flattens out at 40% which is above the test error rate we obtained earlier for boosted stumps. A possible explanation for the superior performance of the boosted stumps over the boosted pruned trees is that the simple two node tree structure of each term in the boosted stump model is more appropriate for additive modeling when using this dataset.

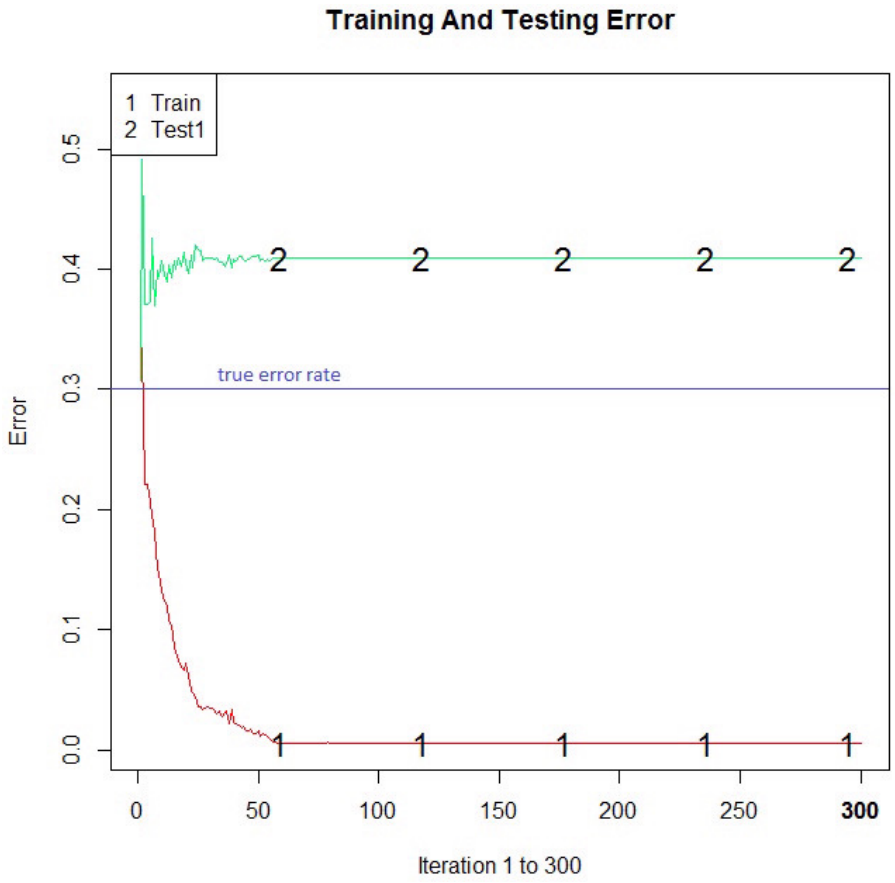


Figure 4.6: Error rates of the boosted pruned tree on simulated binary classification data as per Section 3.3

4.3 Real boosted binary classification example

We now turn to the real life **SPAM** data to assess if applying Adaboost will result in performance enhancements over the pruned classification tree T_α as per Section 3.4 . Unlike before where we used 10-fold cross validation as a proxy for test error we shall now split the 4601 observations into training and test data into the ratio $90\%/10\%$ respectively to make the comparison more meaningful. The R code in Appendix B.3 can be used to generate the comparison.

From Figure 4.7 we notice a marked improvement in accuracy when using Adaboost with stumps on the **SPAM** data. The test error rate of the pruned tree here was 7% (in the earlier example as per Section 3.4 we had a 10-fold cross validated error rate of between $8\% - 9\%$) however after boosting using stumps the test error rate reduces to around 5% which represents almost a 50% improvement. What is also interesting to note from Figure 4.7 is that both the training error and test error continues to drop after the 300^{th} boosting iterations which indicates we could boost to some arbitrary accuracy without suffering from over-fitting. Analyzing the performance of the boosted pruned tree T_α, which can be performed by appending the R code below to the R code given in Appendix B.3, Figure 4.8 shows that the boosted pruned tree outperforms the boosted stumps. The boosted pruned tree only required roughly 50 boosting rounds before the test error rate stabilized at less than 3% which represented almost a 50% improvement over the boosted stump and a 75% improvement over the single pruned tree T_α. Figure 4.9 shows Figure 4.8 however zoomed in

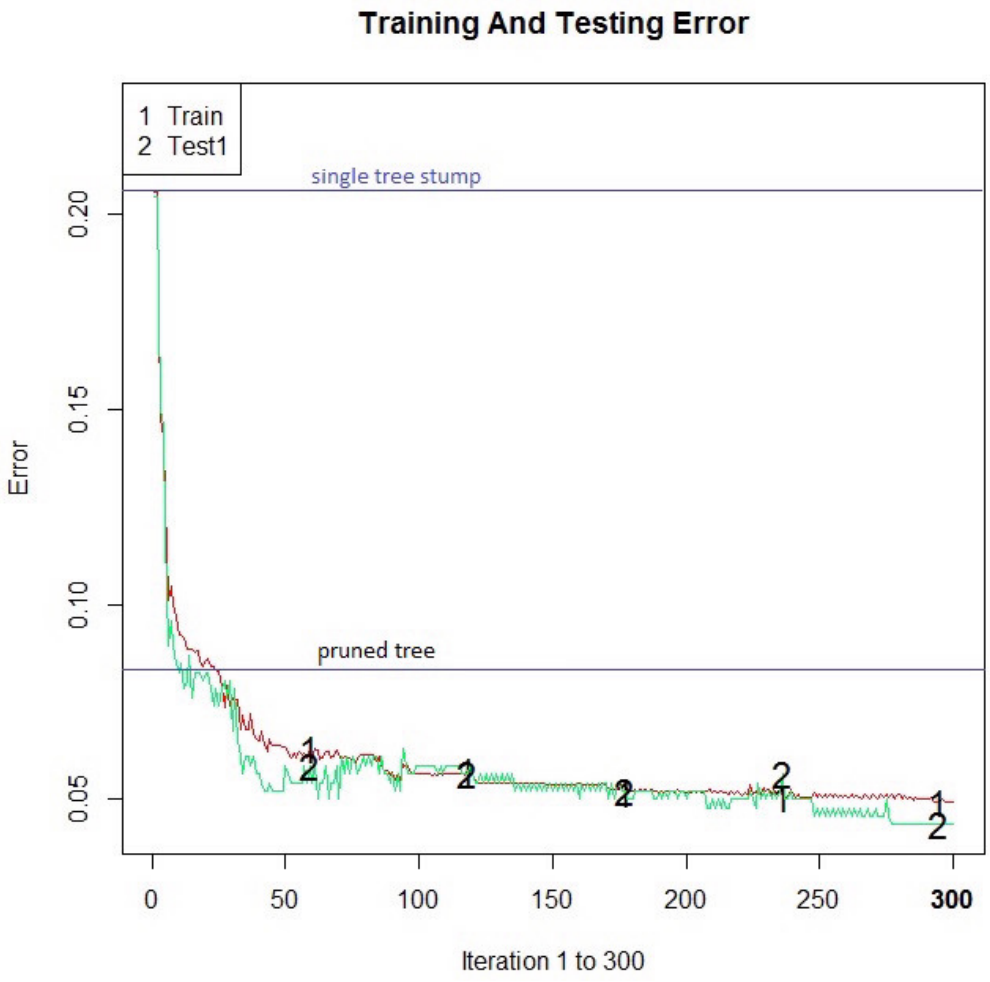


Figure 4.7: Error rates of boosted tree stump on the **SPAM** data as per Secion 3.4

on the x-axis for boosting iterations ranging from 1-50. What this shows is that the test error continues to fall after the 10^{th} boosting iterations where the training error is near or at zero which, as noted in Chapter 2, is a one of the positive anomalies of Adaboost.

```
#calculate the boosted alpha tree
stoprule=rpart.control(cp=0.003)
adaboost<-ada(Y~.,data=traindata,iter=M,loss="e",type="discrete",
control=stoprule,bag.frac=1,nu=1, test.x=testdata[,-1],test.y=testdata[,1])
print(adaboost)
plot(adaboost,FALSE,TRUE)
```

4.4 Simulated boosted multi-classification example

Adaboost.M1 has a distinct drawback within the multi-classification setting as the method will only work for weak learners with an accuracy rate $> \frac{1}{2}$. In addition this criteria becomes increasingly more difficult to hold true for $K > 2$ particularly when the base learner is a tree stump and therefore two solutions become available:

1. The response variable is re-coded to binary and therefore the **Adaboost.M1** algorithm is run using stumps K times for each binary class 0-1 outputting K responses for each observation. The predicted class is then selected based on the highest class value as calculated as in equation (2.5)

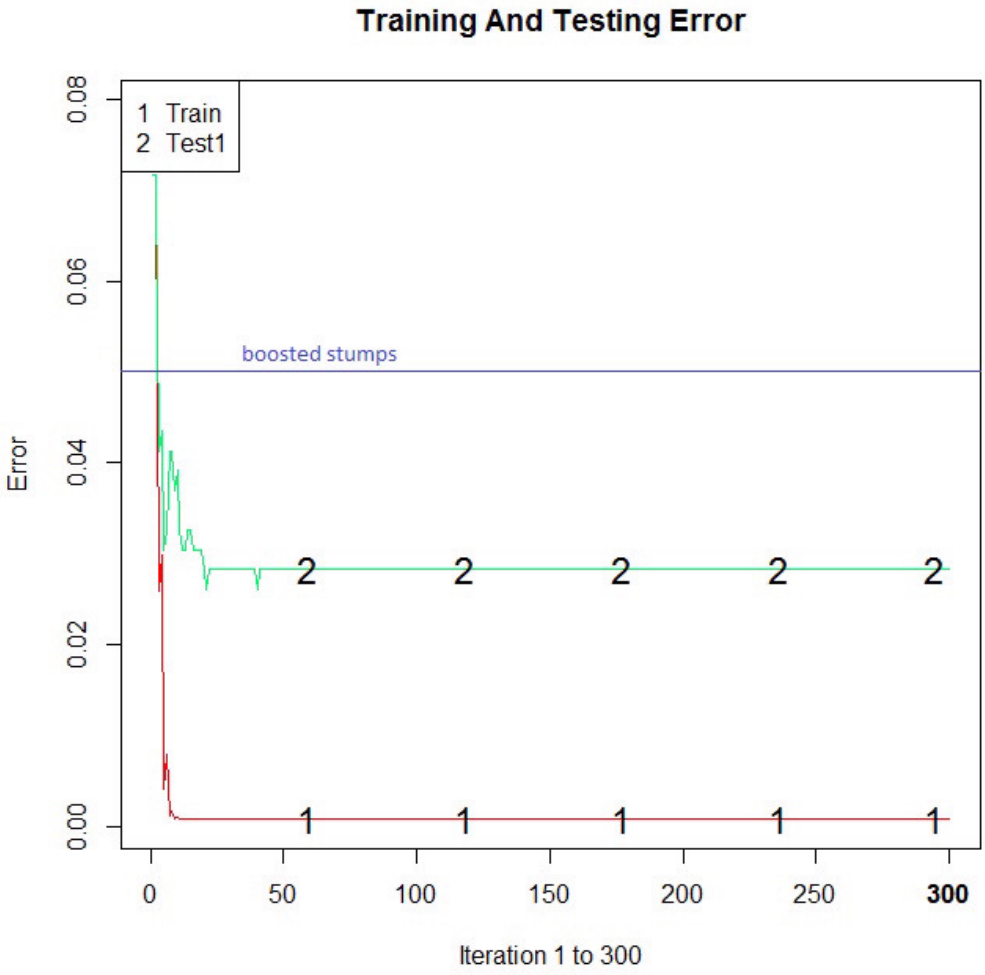


Figure 4.8: Error rates of the boosted prunned tree based on **SPAM** data

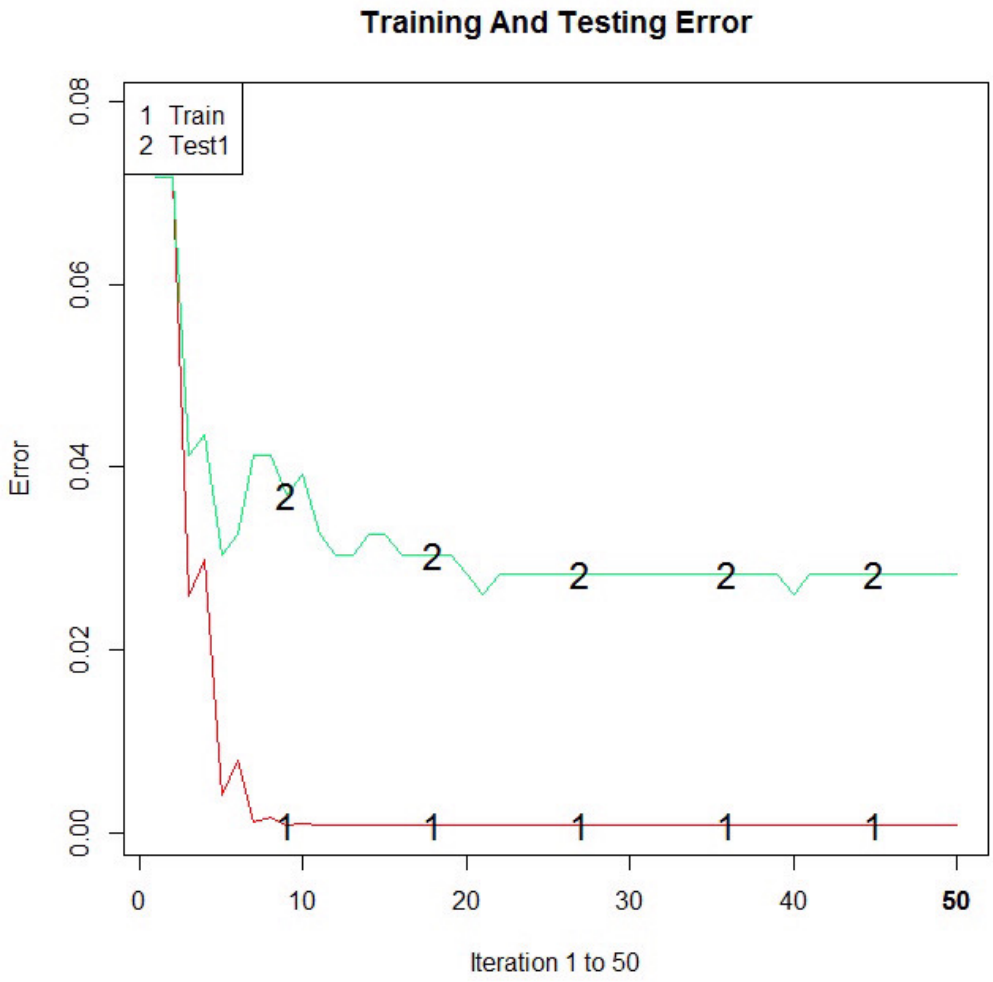


Figure 4.9: Shows Figure 4.8 zoomed in to boosting iterations from 1-50

2. The pruned tree is used as a base learner and boosted using **Adaboost.M1**

The R code in Appendix B.4 compares the accuracy of these two approaches using the simulated multi-classification data as per Section 3.5. The *"ada"* package for R we have been using up until now is only capable of handling binary responses and therefore its application to the first point above is appropriate. However when using the second approach we need to use a package capable of handling multi-class responses and therefore turn to the package *"adabag"*. Both of these packages can be found in the online R library including related help files.

As like before in order to ensure consistency we regenerate the simulated multi-classification dataset and pruned tree using the code in Appendix B.4.1. The pruned tree built as per Section 3.5 results in a test error rate of 26% which is lower than the predefined population error rate of 30%. Note, this is quite plausible as the test sample could have a lesser proportion of error observations due to the random sampling. The R code given in Appendix B.4.2, which must be run after the code in Appendix B.4.1 is called, re-codes the simulated data into binary responses and runs boosted stumps on the resultant "binarized" dataset as per the method in point 1 above. *As a note the nested loop in Appendix B.4.2 is computationally intensive and therefore may take a few minutes to compute even when run on a high powered PC.*

From Figure 4.10 we see that using boosted stumps on the response variables in binary format produces somewhat satisfactory results as the test error of 35% at

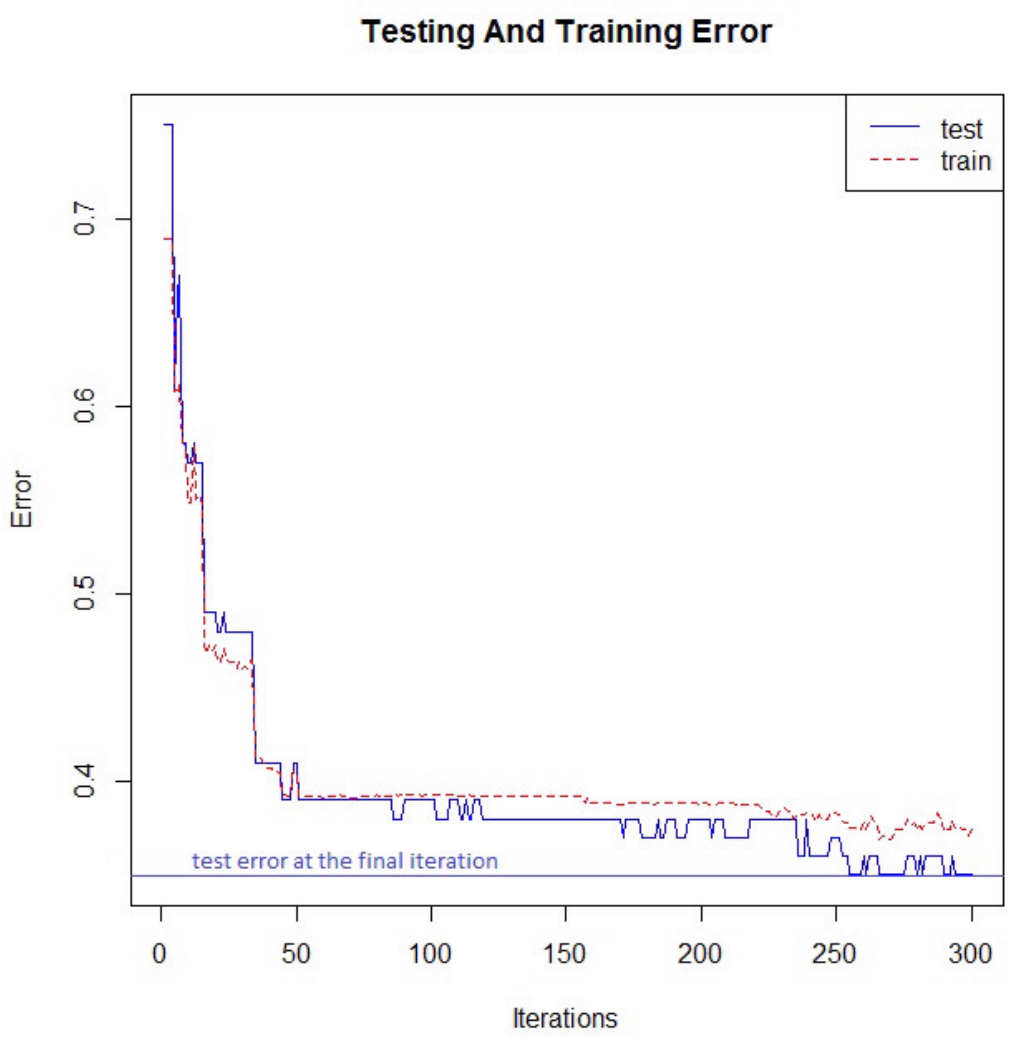


Figure 4.10: Graph showing the boosted training and test error rates using stumps based on simulated multi-classification data as per Section 3.5

the final boosting iteration $M = 300$ is near the actual population error of 30% and hence is also comparable in performance to the pruned multi-classification tree shown in Section 3.5. We also notice that the test error is lower than the training error most likely due to the larger size of the training set relative to test set and the fact that cross-validation is not used here and therefore the test set could very well contain a lower error rate than the training set. Interestingly on further analysis of Figure 4.10 we note that the stumps generated from about iterations 1 to 25 produce error rates above 50% indicating that one could have applied random guessing initially, for at least the first 25 boosting rounds, and thereafter boosted the stumps. Next we shall use the second approach and boost the pruned tree to ascertain if we can produce better results than the boosted stumps. The R code shown in Appendix B.4.3 must be run after both sets of code in Appendix B.4.1 and Appendix B.4.2 have been compiled.

Figure 4.11 shows the error rate as a function of the boosting iterations when using the pruned tree as the base classifier. Earlier we noted that the test error rate for the pruned tree was 26%, which happened to be lower than the pre-specified population error rate of 30%. As a result, boosting the pruned tree classifier resulted in those miss-classified training observations in the first iteration being given additional weighting relative to the correctly classified observations in subsequent boosting iterations, which adversely affected the performance when the boosted models at each iteration where run on the test data. This problem is analogous to over-fitting and

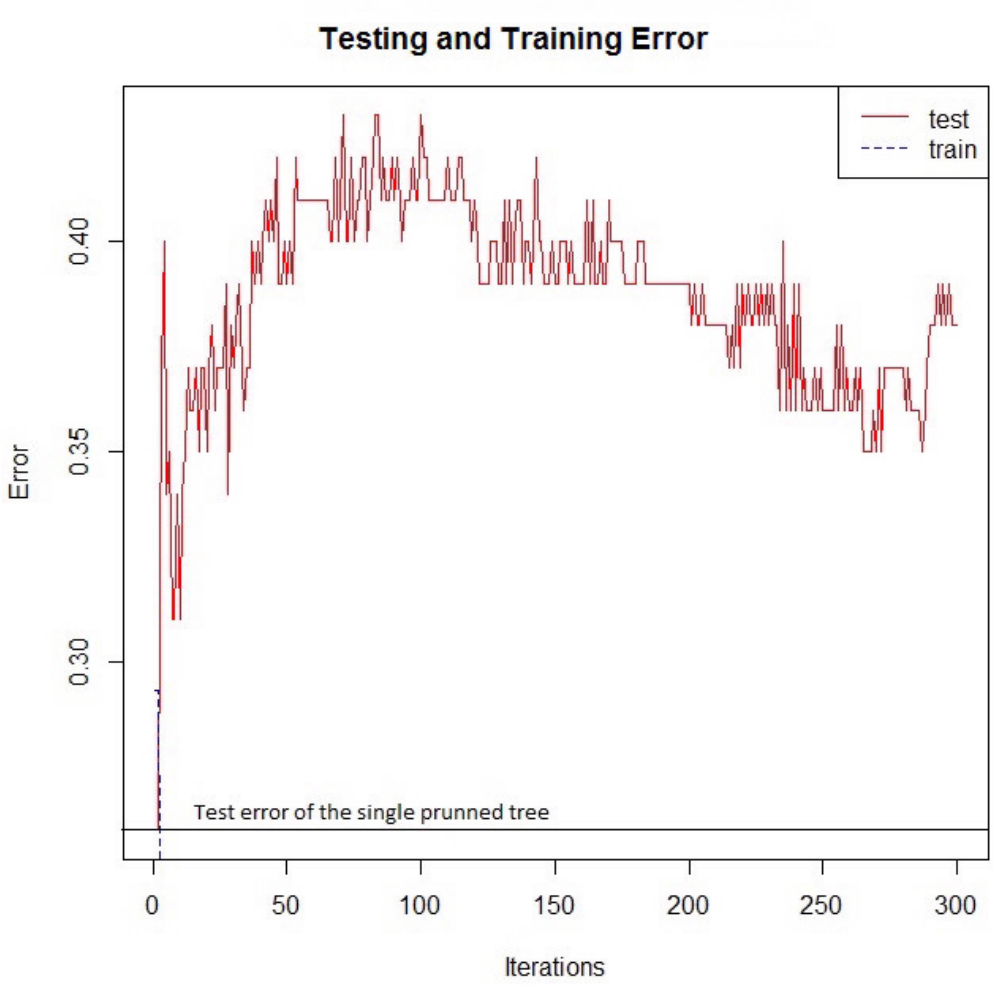


Figure 4.11: Error rates of the pruned tree boosted using simulated multi-classification data as per Section 3.5

therefore the result obtained here reaffirms the "somewhat" in the earlier statement that "*boosting was **somewhat** immune to over-fitting*" and hence we have discovered a case where over-fitting within the context of boosting has indeed resulted in loss of accuracy. However we do notice a downward trend in the test error from around the 100^{th} boosting iteration indicating some recovery in accuracy as the component models which pushed up the test error are increasingly outnumbered by component models which push down the test error as more classifiers are added to the boosted model. The final boosted test error rate using the pruned tree as a base classifier in our case is 38% which is relatively close to the test error rate of 35% generated by the boosted stumps however on careful inspection of Figure 4.11 we notice that we do attain a test error of 35% between iterations 250 to 300.

4.5 Real boosted multi-classification example

Turning back to our earlier real multi-classification problem of OCR using US postal zip codes as described in Section 3.6 we shall again make use of the two approaches of (1) running boosted stumps on a "binarized" version of the response variable and (2) boosting the pruned tree. The R-code in Appendix B.5.1 "binarizes" the response data and produces the boosted stump. As noted earlier the algorithm as coded here is computationally intensive and will take a significant amount of time to output the results. Alternatively one can reduce the number of boosting iterations however the trade-off will be less accurate results.

The output shown in Appendix B.5.1 is a test error rate of 16.692% which is lower, albeit marginally, than the test error of the pruned decision tree of 17.04% as calculated in Section 3.6 and similarly the training error here of 12.207% is lower than the 10-fold cross-validated training error of the pruned decision tree of 13.04% also calculated in Section 3.6. It is interesting to note from Figure 4.12 that error curves of both the training and test data appear to continue to fall after the 100^{th} boosting round suggesting that if one where to boost beyond 100 rounds superior results could be obtained. *Note the reason we have chosen to boost to 100 rounds is due to the fact that the algorithm is computationally intensive and boosting upwards of 200 rounds would take a significant amount of time to compile even on a high powered PC*. Once again we note from Figure 4.12 that stumps generated from about iterations 1 to 10 produce error rates above 50% indicating that one could have applied random guessing initially, for at least the first 10 boosting rounds, and thereafter boosted the stumps.

We shall now conclude our analysis by boosting the pruned tree using the R-code shown in Appendix B.5.2 to ascertain if we obtain improved results. From the output shown in Appendix B.5.2 we get a test error rate of 5.923% which is significantly lower than both the single pruned tree error rate (17.04%) and boosted stumps error rate (16.692%) both shown in Figure 4.13. We also notice in Figure 4.13 that the training error of the pruned tree drops to zero after a few (<10) boosting rounds yet the test error continues to fall well after this, demonstrating once again

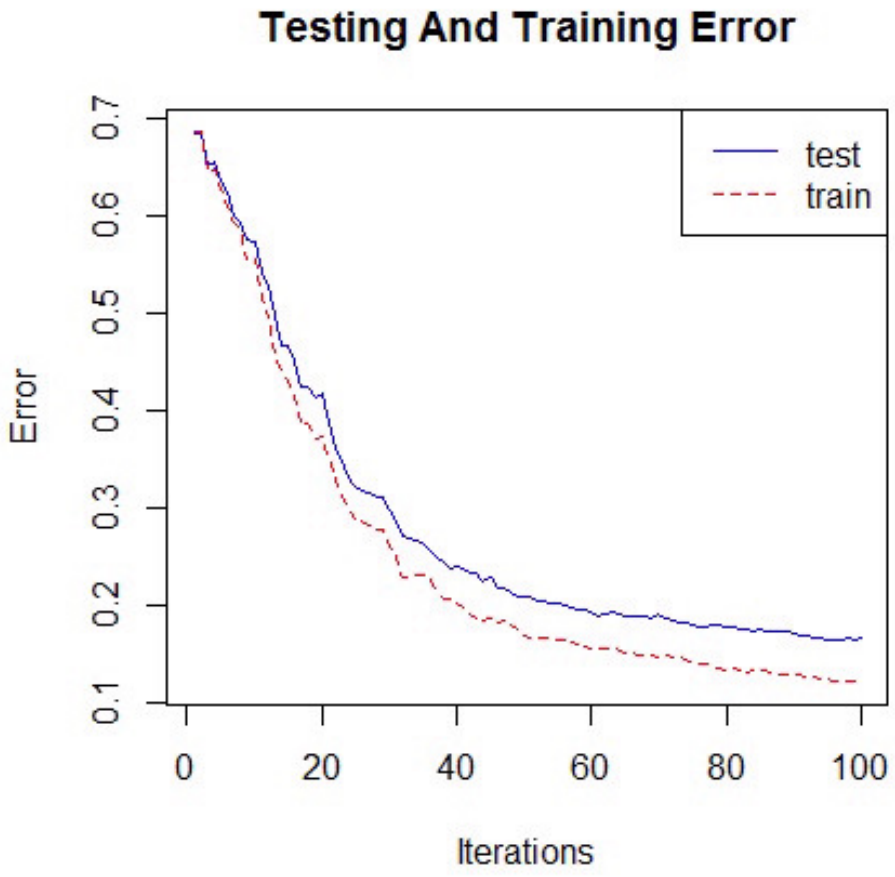


Figure 4.12: Graph showing the error rates of the boosted stumps using the **Digits** data

Adaboost's unexplained performance. Figure 4.13 also shows that boosting after approximately 40 rounds yield marginal gains and therefore the boosting procedure should be stopped at this stage to reduce computational time. Although we have attained significant improvement in prediction accuracy it has come at the expense of added computational time as well as created a final model which is highly complex in structure i.e. a hundred 79 node trees.

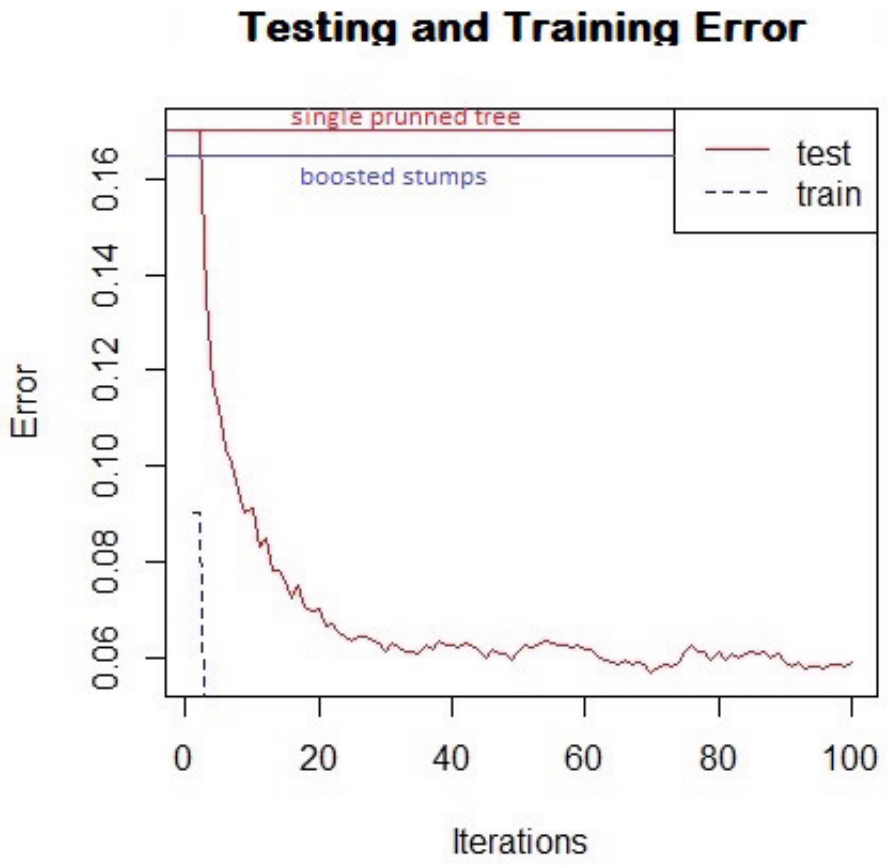


Figure 4.13: Graph showing the error rates of the boosted prunned tree using the **Digits** data as per Section 3.6

Chapter 5
Conclusion

5.1 Latest Developments

The section below presents some of the latest developments in the field of boosting. The first new development explored is the topic of *Margin Theory* which seeks to provide an understanding of boosting's continued performance particularly in light of situations where the training error has reached zero and the test error continues to decline. The second topic is the development of combinatorial techniques using both boosting and bagging. The purpose being to leverage the best each method has to offer in order to create more accurate and robust predicitive power when compared to the individual methods. The last development discussed is the topic of *Gradient Boosted Trees* which seeks to optimize the boosting algorithim when the base predictos are trees.

5.1.1 Margin theory

We have seen that Adaboost displays somewhat of a positive phenomenon in that the test error rate continues to decrease even after the training error reaches zero for an increasing number of boosting iterations. It was not until the authors Schapire et al. [14] that a firm reason for this positive anomaly was provided in terms of a defined

quantity called a *margin.* The margin was constructed to measure the "confidence of prediction" i.e. $yh(\mathbf{x})$ where in the binary case $y \in \{-1, 1\}$ with a real valued base predictor $h(\mathbf{x}) \in [-1, 1]$ is a number in the range $[-1, 1]$. It is easy to see that a positive margin indicates a correct classification and conversely a negative margin indicates a incorrect classification. In addition a margin closer to 1 indicates a "more confident" prediction whilst a positive number closer to 0 indicates a "less confident" prediction. The authors also showed that the margin could be represented graphically as a cumulative distribution function where the x-axis is the margin value say θ and the y-axis the cumulative percentage of observations with a margin less than θ i.e. $P[yh(\mathbf{x}) \leq \theta]$. Figure 5.1 is taken from Schapire et al. [14] with the bottom two graphs showing the margin distribution of training dataset for the bagging and boosting methods. From Figure 5.1 we see that the training error rate using boosting is zero at 5 iterations whilst the test error continues to come down after 5. Similarly the same result is seen using bagging however only after 100 iterations which demonstrates the superiority of boosting over bagging. The dotted lines in the bottom two graphs of Figure 5.1 is the margin distribution taken after 5 iterations whilst the solid line is the margin distribution taken after 1000 iterations. Since we would want as many observations with a margin close to or equal to 1 we would ideally like to see the graph of the cumulative distribution to be concentrated towards the extreme right. As noted earlier for boosting that the training error rate was zero after 5 iterations, we now notice that the dotted line (which represents the margin distribution after 5

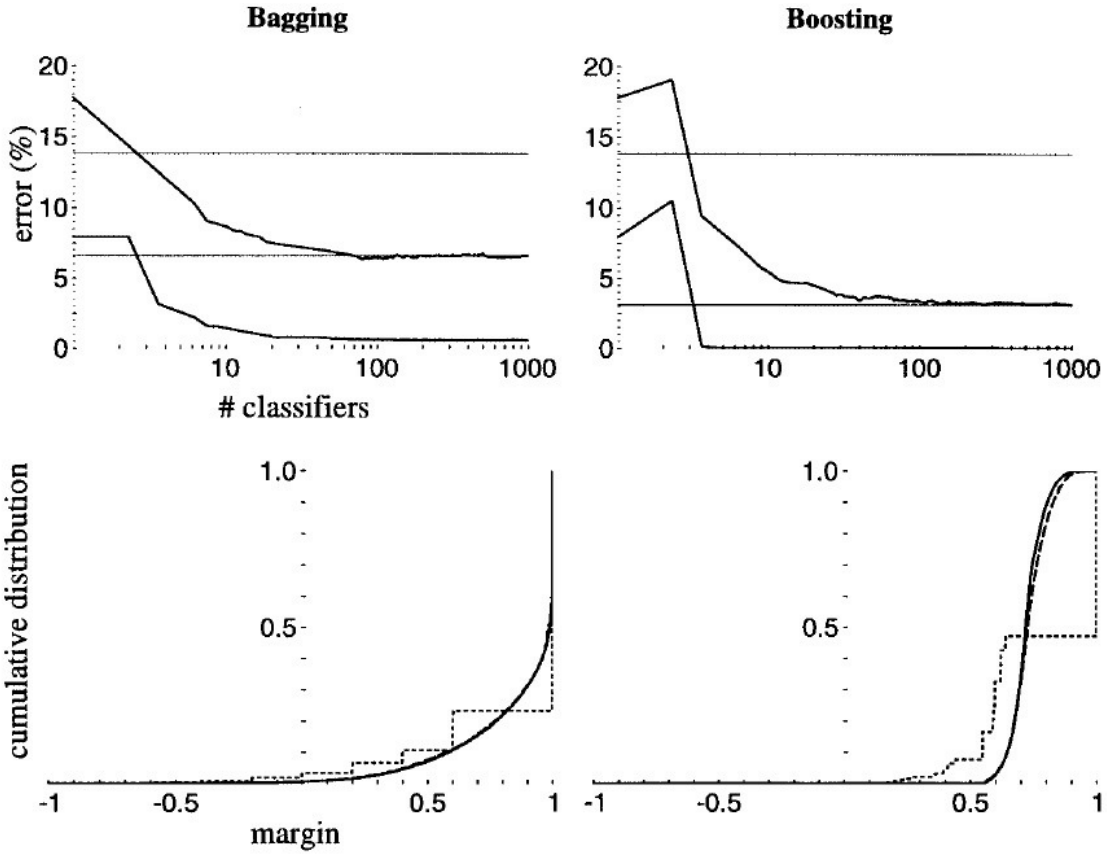


Figure 5.1: Top two graphs showing the training error rates and test error rates as a function of bagging and boosting iterations.Bottoms two graphs showing the margin distributions of the training dataset of the bagging and boosting examples above

iterations) is seen to be relatively spread out over margin values greater than 0. When looking at the solid line for boosting (which represents the margin distribution after 1000 iterations) we notice that the curve is more concentrated to the right beyond a margin value of 0.5. It is this improvement, remembering that the margin distribution is in terms of the training dataset, that translates into improved performance in the test error rate of boosting.

The authors Schapire et al.[14] provided an explanation and showed that the Adaboost algorithm has the effect of increasing the weights of those observations where the margin was small (or negative) and therefore Adaboost sought not only to boost prediction accuracy but also the margin quantity. Because the margin is a real valued function even after the training error had reached zero the margin, due to observations with small positive margins which would have irrespectively been correctly classified, continued to be "boosted" by the algorithm which therefore improved the test error rate performance. The authors Schapire et al. [14] went on to formally prove that if most of the margins of the examples where large the chances of the total margin being less than zero (negative) decreased.

5.1.2 Combining boosting and bagging

A question which arises under the theme of improving prediction accuracy is "what would result if multiple ensemble methods where combined" and specifically within this dissertation where we have compared boosting to bagging, what would the result be if these two methods where merged somehow. The authors Kotsiantis et al. [15] performed such a combination noting that there are conditions where bagging outperforms boosting which occurs namely within noisy data settings. The reason being is that bagging, by way of construction, is primarily a bias reducing method whilst boosting reduces both bias and variance. The authors Kotsiantis et al. [15] in effect create a model by running the two methods (boosting using Adaboost and

bagging) simultaneously with each producing a prediction with an associated "confidence value". The confidence value is calculated by summing up the individual prediction probabilities i.e. the base predictors are able to produce confidence rated predictions, of each method. The method with the highest number is then selected. This method seeks to improve the robustness of both the boosting and bagging algorithm in order to accommodate various settings i.e. noisy data and or noise-less data.

The authors go onto to compare their new method against bagging and a variety of boosting methods including Adaboost using different base predictors. The results, compiled over multiple datasets, prove positive, albeit not by a significant margin improving performance only between 9%-16%. However the combinatorial method did consistently outperforming competitor ensemble methods by this margin. The authors also note that although the results where positive, drawbacks of the method included lengthy computation times (only boosted for 25 iterations and 10 bagging iterations) and added model complexity.

5.1.3 Gradient boosted trees

Adaboost method using classification trees, as described in this thesis, is underpinned by minimizing the exponential loss function as in equation (2.27). However there are multiple other loss function available which serve to be more robust but which unfortunately result in increased complexity and computational intensity. Faced with

this problem the authors in Hastie et al. [4] (p 337-387) proposed an alternate or enhanced technique to Adaboost for trees using numerical optimization called *Gradient Boosting*. The method, in summary, works by using a more robust differentiable loss function, for classification trees the authors propose using the *multinomial deviance loss function* which has a closed formed solution, which is then partially differentiated at the discrete training data set observations which targets are subsequently fitted to the independent training data. The effect of differentiating the loss function is known as *Steepest Descent* as one aims to minimize the loss function optimally using its gradient. The process is repeated or boosted as many times as required and thereby builds an additive model similar to forward stagewise additive modelling described earlier. What has been outlined here is merely a high level summary of the Gradient Boosting method and further reading on this method may be found in Hastie et al. [4] (p 337-387).

5.2 Concluding Remarks

We have shown that although the field of boosting has been around since the 80's, Adaboost, boosting's most popular sub-method, was only developed in the mid 90's and is a relatively young and evolving development in the field of statistics. The method has many practical and appealing applications such as improving prediction accuracy whilst remaining simple to implement however also exhibits certain drawbacks such as the blackbox nature of the final model and being computationally intensive.

We have also shown the origin of Adaboost does not derive from a neat linear mathematical proof and instead was discovered somewhat by chance through an extension of a solution to the very popular on-line allocation problem. Thereafter for a period of time the statistical community continued to be astounded by the methods unexplained performance and it was only after a few years that more rigorous and robust mathematical justifications were put forward. Such justifications were based on using well known statistical frameworks, namely by employing specific loss functions under forward stagewise additive and additive logistic modelling processes which resulted in the exact formulation and match to the Adaboost algorithm.

We also described the methodology of classification trees and studied the application of this predictive model on simulated and real data. The output in summary was that although the results where satisfactory for larger trees, stumps generally where not accurate enough to be of any use in practice and hence presented itself as an ideal candidate for boosting.

Lastly we analyzed the application of Adaboost on the same real and simulated datasets used for classification trees in order to compare the results. What we found was that Adaboost using stumps generally performed as well or in some cases better than a full or optimized pruned tree. A possible explanation for this could be attributed to the boosted stumps simpler additive model structure and which also presents itself as topic for further exploration. We also discovered that when applying Adaboost using a pruned tree the overall results where somewhat mixed. Where

underperformance was noted, and not by a large margin, this could be corrected by increasing the number of boosting rounds. Conversely where superior performance was noted the achievement in prediction accuracy appeared significant. In addition we also noted situations where the test error continued to fall long after the training error had reached zero which empirically reaffirmed the positive anomaly of boosting. Therefore although Adaboost as a prediction enhancing technique generally appeared to perform well it suffered at times under certain situations, however which underperformance could be corrected by changing the choice of the base predictor or increasing the number of boosting iterations with the latter coming at the cost of increased computational time.

By no means are we suggesting Adaboost as a "silver bullet" method to increase prediction accuracy when faced with poor performing individual learning methods. As with improved accuracy emanating from the application of Adaboost comes increased model complexity and interpretability attributed to the additive nature of the method as well as increased computational time as we have experienced in the boosting examples. The issue of model complexity and interpretability can be argued that in the real world their are cases where complexity can be ignored i.e. black-box models are acceptable, because one is more interested in the prediction output rather than the underlying methodology or casual relationships. Similarly the issue of prolonged computational time becomes mute when projecting the trend of ever increasing computer power and therefore we can be confident that over a relatively short period of

time this issue will fall away completely. In summary the combination of the arguments above can not help but lead one to believe that Adaboost is very likely to become increasingly more popular in the future as a generic method of overcoming modeling inaccuracy problems.

References

[1] T.M Mitchell, Machine Learning: *An Artificial Intelligence Approach,* Morgan Kaufmann Publisher Inc., California, 1986

[2] L.Breiman, *Bagging Predictors,* Machine Learning, 24, (1996), Kluwer Academic Publishers, Boston, 123-140

[3] J.A. Hoeting, D Madigan, A.E. Raftery and C.T Volinsky, *Bayesian Model Averaging*, Statistical Science Vol 14, No 4, (1999), 382-417

[4] Trevor Hastie, Robert Tibshirani and Jerome Friedman, *The Elements of Statistical Learning - Data Mining, Inference, and Prediction,* Second Edition, Springer Science + Business Media, LLC, New York, 2008

[5] Yoav Freund and Robert E. Schapire, *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting,* Journal of Computer and System Sciences, 55, (1997), 119-139

[6] Michael Kearns and Leslie G. Valiant, *Learning Boolean formulae or finite automata is as hard as factoring.* Technical Report TR-14-88, Harvard University Aiken Corporation Laboratory, August 1988

[7] Yoav Freund and Robert E. Schapire, *A short introduction to boosting,* Journal of Japanese Society for Artificial Intelligence, 14(5), (1999), 771-780

[8] Robert E. Schapire and Yoram Singer, *Improved boosting algorithms using confidence-rated predictions,* Proceedings on the Eleventh Annual Conference on Computational Learning Theory, (1998), 80-91

[9] Trevor Hastie, Robert Tibshirani and Jerome Friedman, *Additive Logistic Regression: a Statistical View of Boosting (with discussion),* Annals of Statistics 28, 307-337

[10] N. Littlestone and M.K. Warmuth, *The weighted majority algorithm*, Inform and Comput, 108, (1994), 212-261

[11] A. Buja, T. Hastie, R. Tibshirani, *Linear smoothers and additive models (with discussion)*, Annals of Statistics, 17, 435-555 (1989)

[12] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth, *Learnability and the Vapnik-Chervonenkis dimension*, Journal of the Association for Computing Machinery, 36(4), October 1989, 929–965

[13] L. Brieman and R Ihaka, *Nonlinear discriminant analysis via scaling and ACE,* Technical report for the University of California, Berkeley, (1984)

[14] Robert E. Schapire, Yoav Freund, Peter Bartlett and Wee Sun Lee, *Booting the Margin: A new Explanation for the Effectiveness of Voting Methods*,The Annals of Statistics, Vol 26 No. 5, 1651-1686, 1998

[15] S.B. Kotsiantis, P.E. Pintelas, *Combining Bagging and Boosting,* International Journal of Computational Intelligence, Vol 1, No. 5, 1304-2386, 2004

[16] Leo Breiman, J. H.Friedman, R. A. Olshen and , C. J. Stone. *Classification and regression trees*. Monterey, CA: Wadsworth & Brooks/Cole Advanced Books & Software, 1984

Appendix A
R Code used for Classification Tree Examples

A.1 R code for the simulated binary classification example

```
#Load the rpart package

    library(rpart)

    # Generate the input variables X1 and X2 on the unit interval

    Y=0

    N=1000

    X1=runif(N)

    X2=runif(N)

    # Select values for t1, t2, t3 and t4

    t1=0.4

    t2=0.4

    t3=0.6

    t4=0.8

    # Generate the binary dependent variable which takes on values of -1 and 1

    for (i in 1:N) {

    #define region R1 to be Y=1
```

```
if (X1[i]<=t1 & X2[i]<=t2){Y[i]=1}

#define region R2 to be Y=-1

if (X1[i]<=t1 & X2[i]>t2){Y[i]=-1}

#define region R3 to be Y=1

if (X1[i]<=t3 & X1[i]>t1){Y[i]=1}

#define region R4 to be Y=-1

if (X1[i]>t3 & X2[i]<=t4){Y[i]=-1}

#define region R5 to be Y=-1

if (X1[i]>t3 & X2[i]>t4){Y[i]=-1}

}

# add random error to say 30% of the observations

maxerror=round(0.3*N,0)

index=0

for (count in 1:maxerror){

    isin=1

    while (isin==1){

        isin=0

        temp=round(runif(1,min=1,max=N),0)

        for (i in 1:length(index)) {

            if(index[i]==temp) {isin=1}

        }
```

```
        }
    index[count]=temp
    if (Y[index[count]]==1) {Y[index[count]]=-1} else {Y[index[count]]=1}
    }
    # create the data set/frame
    Data <- data.frame(Y=Y,X1=X1,X2=X2)
    # Grow the full tree using the Gini index and stop when a minimum node size of 5
    is reached
    #first set the tree stopping criteria i.e. minimum node size of 5. Also set
    cross-validation to 10-fold
    stoppingrule=rpart.control(minbucket=5,xval=10,cp=0)
    tree0 <-rpart(Y ~X1 + X2,data=Data,method="class",control=stoppingrule,
    parms=list(split="gini"))
    printcp(tree0)
    plot(tree0,compress=TRUE,margin=0.2)
```

A.2 R code for the real binary classification example

```
#Load the rpart and XLconnect packages
    library(rpart)
    library(XLconnect)
    #XLconnect allows reading of excel files into R
```

```
# Specify the file location

spamfile <- system.file("demoFiles/spamdata.xlsx", package = "XLConnect")

# Load the workbook

spamwb <- loadWorkbook(spamfile)

# Read worksheet “spam”

spamdata <- readWorksheet(spamwb, sheet = "spam")

# Grow the full tree using the Gini index and stop when a minimum node size

#of 5 is reached

#first set the tree stopping criteria i.e. minimum node size of 5.

#Also set cross-validation to 10-fold

stoppingrule=rpart.control(minbucket=5,xval=10,cp=0)

tree0<-rpart(Y~.,data=spamdata,method="class",control=stoppingrule,

parms=list(split="gini"))

printcp(tree0)
```

A.3 R code for the simulated multi-classification example

```
#Load the rpart package

# Generate the input variables X1 and X2 on the unit interval

Y=0

N=1000

X1=runif(N)
```

```
X2=runif(N)

# Select values for t1, t2, t3 and t4

t1=0.4

t2=0.4

t3=0.6

t4=0.8

# Generate the binary dependent variable which takes on values of -1 and 1

for (i in 1:N) {

#define region R1 to be Y=-5

if (X1[i]<=t1 & X2[i]<=t2){Y[i]=-5}

#define region R2 to be Y=-7

if (X1[i]<=t1 & X2[i]>t2){Y[i]=-7}

#define region R3 to be Y=0

if (X1[i]<=t3 & X1[i]>t1){Y[i]=0}

#define region R4 to be Y=2

if (X1[i]>t3 & X2[i]<=t4){Y[i]=2}

#define region R5 to be Y=4

if (X1[i]>t3 & X2[i]>t4){Y[i]=4}

}

# add random error to say 30% of the observations

maxerror=round(0.3*N,0)
```

```
index=0
for (count in 1:maxerror){
isin=1
while (isin==1){
isin=0
temp=round(runif(1,min=1,max=N),0)
for (i in 1:length(index)) {
 if(index[i]==temp) {isin=1}
 }
}
index[count]=temp
if (Y[index[count]]==-5) {Y[index[count]]=4} else
if (Y[index[count]]==-7) {Y[index[count]]=2} else
if (Y[index[count]]==0) {Y[index[count]]=-5} else
if (Y[index[count]]==2) {Y[index[count]]=-7} else
if (Y[index[count]]==4) {Y[index[count]]=0}
}
# Create the data set/frame
Data <- data.frame(Y=Y,X1=X1,X2=X2)
# Grow the full tree using the Gini index and stop when a minimum node size
of 5 is reached
```

```
#first set the tree stopping criteria i.e. minimum node size of 5.
Also set cross-validation to 10-fold
stoppingrule=rpart.control(minbucket=5,xval=10,cp=0)
tree0 <-rpart(Y ~X1 + X2,data=Data,method="class",control=stoppingrule,
parms=list(split="gini"))
printcp(tree0)
plotcp(tree0)
```

A.4 R code for the real multi-classification example

#Due to the size of the data-set this code should be run on R as soon as it has been loaded prior to running any other commands

```
options(java.parameters = "-Xmx4g" ) #increases the amount of memory to use
# Load the XLConnect and rpart packages
library(XLConnect)
library(rpart)
#XLconnect allows reading of excel files into R
# Specify the file location
digitsfile <- system.file("demoFiles/digitsdata.xlsx", package = "XLConnect")
# Load the workbook
digitswb <- loadWorkbook(digitsfile)
# Read worksheet “zip”
```

```
digitsdata <- readWorksheet(digitswb, sheet = "zip")

stoppingrule=rpart.control(minbucket=5,xval=10,cp=0)

tree0<-rpart(Y~.,data=digitsdata,method="class",

control=stoppingrule,parms=list(split="gini"))

printcp(tree0)

plotcp(tree0)
```

Appendix B
R code used for Adaboost Examples

B.1 R code used for the Adaboost illustration example

```
#Load the rpart package

    library(rpart)

    #Generate the training and test data

    # Generate the independent standard normal input variables X1 to X10

    Y=0

    N=12000

    X1=rnorm(N)

    X2=rnorm(N)

    X3=rnorm(N)

    X4=rnorm(N)

    X5=rnorm(N)

    X6=rnorm(N)

    X7=rnorm(N)

    X8=rnorm(N)

    X9=rnorm(N)

    X10=rnorm(N)
```

```
#calculate the chi squared value at p=0.5

chisq= qchisq(0.5, df=10)

summation=0

for(i in 1:N){

summation=X1[i]^2+X2[i]^2+X3[i]^2+X4[i]^2+X5[i]^2+

X6[i]^2+X7[i]^2+X8[i]^2+X9[i]^2+X10[i]^2

      ifelse(summation>chisq,Y[i]<-1,Y[i]<–1)

}

trainindex=sample(1:N,2000,FALSE)

testindex=setdiff(1:N, trainindex)

datatrain <- data.frame(Y=Y[trainindex],X1=X1[trainindex],X2=X2[trainindex],

X3=X3[trainindex], X4=X4[trainindex], X5=X5[trainindex], X6=X6[trainindex],

X7=X7[trainindex], X8=X8[trainindex], X9=X9[trainindex], X10=X10[trainindex])

datatest <- data.frame(Y=Y[testindex],X1=X1[testindex],X2=X2[testindex],

X3=X3[testindex], X4=X4[testindex], X5=X5[testindex], X6=X6[testindex],

X7=X7[testindex], X8=X8[testindex], X9=X9[testindex], X10=X10[testindex])

#train the full tree

stoppingrule=rpart.control(minbucket=1,xval=10,cp=0)

tree0 <-rpart(Y~.,data=datatrain,method="class",control=stoppingrule,

parms=list(split="gini"))

printcp(tree0)
```

```
#test the full tree

testprediction<-predict(tree0, newdata=datatest, type = "class")

#calculate the test error of the full tree

errordata<-data.frame(Acutal= datatest[,1],Predicted=testprediction)

temp<-ifelse(errordata[,1]!=errordata[,2],1,0)

fulltreetesterror=sum(temp)/nrow(errordata)

print(fulltreetesterror)

#train the stump

stumprule=rpart.control(cp=-1,maxdepth=1,minsplit=0)

stump <-rpart(Y ~.,data=datatrain,method="class",control=stumprule,

parms=list(split="gini"))

#test the stump tree

testprediction<-predict(stump, newdata=datatest, type = "class")

#calculate the test error of the stump

errordata<-data.frame(acutal= datatest[,1],predicted=testprediction)

temp<-ifelse(errordata[,1]!=errordata[,2],1,0)

stumptreetesterror=sum(temp)/nrow(errordata)

print(stumptreetesterror)

#calculate the boosted stumps

library(ada)

M=400
```

```
adaboost<-ada(Y~.,data=datatrain,iter=M,loss="e",type="discrete",

control=stumprule,bag.frac=1,nu=1, test.x=datatest[,-1],test.y=datatest[,1])

print(adaboost)

plot(adaboost,FALSE,TRUE)

summary(adaboost,n.iter=M)
```

B.2 R code used for the simulated boosted binary example

```
#Load the requisite packages

library(rpart)

library(ada)

# Generate the input variables X1 and X2 on the unit interval

Y=0

N=11000

X1=runif(N)

X2=runif(N)

# Select values for t1, t2, t3 and t4

t1=0.4

t2=0.4

t3=0.6

t4=0.8

# Generate the binary dependent variable which takes on values of -1 and 1
```

```
for (i in 1:N) {

#define region R1 to be Y=1

if (X1[i]<=t1 & X2[i]<=t2){Y[i]=1}

#define region R2 to be Y=-1

if (X1[i]<=t1 & X2[i]>t2){Y[i]=-1}

#define region R3 to be Y=1

if (X1[i]<=t3 & X1[i]>t1){Y[i]=1}

#define region R4 to be Y=-1

if (X1[i]>t3 & X2[i]<=t4){Y[i]=-1}

#define region R5 to be Y=-1

if (X1[i]>t3 & X2[i]>t4){Y[i]=-1}

}

# add random error to say 30% of the observations

maxerror=round(0.3*N,0)

index=0

for (count in 1:maxerror){

isin=1

while (isin==1){

isin=0

temp=round(runif(1,min=1,max=N),0)

for (i in 1:length(index)) {
```

```
            if(index[i]==temp) {isin=1}
        }
        }
        index[count]=temp
        if (Y[index[count]]==1) {Y[index[count]]=-1} else {Y[index[count]]=1}
        }
        # create the training and test data sets
        trainindex=sample(1:N,1000,FALSE)
        testindex=setdiff(1:N, trainindex)
        traindata <- data.frame(Y=Y[trainindex],X1=X1[trainindex],X2=X2[trainindex])
        testdata <- data.frame(Y=Y[testindex],X1=X1[testindex],X2=X2[testindex])
        # Grow the full tree using the Gini index and stop when a minimum node size of 5 is reached
        #first set the tree stopping criteria i.e.  minimum node size of 5.  Also set cross-validation to
10-fold
        stoppingrule=rpart.control(minbucket=5,xval=10,cp=0)
        tree0 <-rpart(Y ~X1 + X2,data=traindata,method="class",control=stoppingrule,parms=list(split="gini"))
        printcp(tree0)
        #Prune the full tree0 to the selected cp value where xerror is minimized
        treealpha<-prune(tree0,cp=0.00434783)
        printcp(treealpha)
        #test the full tree
```

```
testprediction<-predict(treealpha, newdata=testdata, type = "class")

#calculate the test error of the prunned tree

errordata<-data.frame(Acutal= testdata[,1],Predicted=testprediction)

temp<-ifelse(errordata[,1]!=errordata[,2],1,0)

fulltreetesterror=sum(temp)/nrow(errordata)

print(fulltreetesterror)

#train the stump

stumprule=rpart.control(cp=-1,maxdepth=1,minsplit=0)

stump <-rpart(Y ~.,data=traindata,method="class",control=stumprule,parms=list(split="gini"))

#test the stump tree

testprediction<-predict(stump, newdata=testdata, type = "class")

#calculate the test error of the stump

errordata<-data.frame(acutal= testdata[,1],predicted=testprediction)

temp<-ifelse(errordata[,1]!=errordata[,2],1,0)

stumptreetesterror=sum(temp)/nrow(errordata)

print(stumptreetesterror)

#calculate the boosted stumps

M=300

adaboost<-ada(Y~.,data=traindata,iter=M,loss="e",type="discrete",control=stumprule,bag.frac=1,nu=1,
test.x=testdata[,-1],test.y=testdata[,1])

print(adaboost)
```

```
plot(adaboost,FALSE,TRUE)
```

B.3 R code used for the real boosted binary example

```
# Load the rpart, ada and XLconnect packages

        library(rpart)

        library(ada)

        library(XLConnect)

        #XLconnect allows reading of excel files into R

        # Specify the file location

        spamfile <- system.file("demoFiles/spamdata.xlsx", package = "XLConnect")

        # Load the workbook

        spamwb <- loadWorkbook(spamfile)

        # Read worksheet “spam”

        spamdata <- readWorksheet(spamwb, sheet = "spam")

        # create the training and test data sets based on 10-fold comparision

        N=nrow(spamdata)

        trainN=round(0.9*N,0)

        testN=round(0.1*N,0)

        trainindex=sample(1:N,trainN,FALSE)

        testindex=setdiff(1:N, trainindex)

        traindata <- data.frame(Y=spamdata[trainindex,ncol(spamdata)],X=spamdata[trainindex,- ncol(spamdata)])
```

```
testdata <- data.frame(Y=spamdata[testindex,ncol(spamdata)],X=spamdata[testindex,- ncol(spamdata)])
# Grow the full tree using the Gini index and stop when a minimum node size of 5 is reached
#first set the tree stopping criteria i.e.  minimum node size of 5.  Also set cross-validation to 10-fold
stoppingrule=rpart.control(minbucket=5,xval=10,cp=0)
tree0 <-rpart(Y~.,data=traindata,method="class",control=stoppingrule,parms=list(split="gini"))
printcp(tree0)
#Prune the full tree0 to the selected cp value where xerror is minimized
#in this case where cp=0.003
treealpha<-prune(tree0,cp=0.003)
#test the pruned tree
testprediction<-predict(treealpha, newdata=testdata, type = "class")
#calculate the test error of tree alpha
errordata<-data.frame(acutal= testdata[,1],predicted=testprediction)
temp<-ifelse(errordata[,1]!=errordata[,2],1,0)
prunnedtreetesterror=sum(temp)/nrow(errordata)
print(prunnedtreetesterror)
#output: [1] 0.07173913
#train the stump
stumprule=rpart.control(cp=-1,maxdepth=1,minsplit=0)
stump <-rpart(Y ~.,data=traindata,method="class",control=stumprule,parms=list(split="gini"))
```

```
#test the stump tree

testprediction<-predict(stump, newdata=testdata, type = "class")

#calculate the test error of the stump

errordata<-data.frame(acutal= testdata[,1],predicted=testprediction)

temp<-ifelse(errordata[,1]!=errordata[,2],1,0)

stumptreetesterror=sum(temp)/nrow(errordata)

print(stumptreetesterror)

#output: [1] 0.2043478

#calculate the boosted stumps

M=300

adaboost<-ada(Y~.,data=traindata,iter=M,loss="e",type="discrete",control=stumprule,bag.frac=1,nu=1,
test.x=testdata[,-1],test.y=testdata[,1])

print(adaboost)

plot(adaboost,FALSE,TRUE)
```

B.4 R code used for the simulated boosted multi-classification example

B.4.1 Part 1

```
# Load the rpart, ada and XLconnect packages

library(rpart)
```

```
library(ada)

library(adabag)

library(XLConnect)

# Generate the input variables X1 and X2 on the unit interval

Y=0

N=1000

X1=runif(N)

X2=runif(N)

# Select values for t1, t2, t3 and t4

t1=0.4

t2=0.4

t3=0.6

t4=0.8

# Generate the binary dependent variable which takes on values of -1 and 1

for (i in 1:N) {

#define region R1 to be Y=-5

if (X1[i]<=t1 & X2[i]<=t2){Y[i]=-5}

#define region R2 to be Y=-7

if (X1[i]<=t1 & X2[i]>t2){Y[i]=-7}

#define region R3 to be Y=0

if (X1[i]<=t3 & X1[i]>t1){Y[i]=0}
```

```
#define region R4 to be Y=2

if (X1[i]>t3 & X2[i]<=t4){Y[i]=2}

#define region R5 to be Y=4

if (X1[i]>t3 & X2[i]>t4){Y[i]=4}

}

# add random error to say 30% of the observations

maxerror=round(0.3*N,0)

index=0

for (count in 1:maxerror){

isin=1

while (isin==1){

isin=0

temp=round(runif(1,min=1,max=N),0)

for (i in 1:length(index)) {

        if(index[i]==temp) {isin=1}

}

}

index[count]=temp

if (Y[index[count]]==-5) {Y[index[count]]=4} else

if (Y[index[count]]==-7) {Y[index[count]]=2} else

if (Y[index[count]]==0) {Y[index[count]]=-5} else
```

```
if (Y[index[count]]==2) {Y[index[count]]=-7} else
if (Y[index[count]]==4) {Y[index[count]]=0}
}
# create the training and test data sets comparable to 10-fold cross validation
trainN=round(0.9*N,0)
testN=round(0.1*N,0)
trainindex=sample(1:N,trainN,FALSE)
testindex=setdiff(1:N, trainindex)
traindata <- data.frame(Y=Y[trainindex],X1=X1[trainindex],X2=X2[trainindex])
testdata <- data.frame(Y=Y[testindex],X1=X1[testindex],X2=X2[testindex])
# Grow the full tree using the Gini index and stop when a minimum node size of
#5 is reached
#first set the tree stopping criteria i.e. minimum node size of 5.
#Also set cross-validation to 10-fold
stoppingrule=rpart.control(minbucket=5,xval=10,cp=0)
tree0 <-rpart(Y~.,data=traindata,method="class",control=stoppingrule,
parms=list(split="gini"))
printcp(tree0)
#Prune the full tree0 at the selected cp value where xerror is minimized based on
#the output above.
#Note the cp value chosen here should be simlar to the value chosen in the
```

```
#earlier example. In this case where cp=0.003

treealpha<-prune(tree0,cp=0.003)

printcp(treealpha)

#test the pruned tree

testprediction<-predict(treealpha, newdata=testdata, type = "class")

#calculate the test error of tree alpha

errordata<-data.frame(acutal= testdata[,1],predicted=testprediction)

temp<-ifelse(errordata[,1]!=errordata[,2],1,0)

prunnedtreetesterror=sum(temp)/nrow(errordata)

print(prunnedtreetesterror)

#[1] 0.26
```

B.4.2 Part 2

```
#re-code the response variable into binary for the training data

bin1<-as.numeric(traindata[,1]==-7)

bin2<- as.numeric(traindata[,1]==-5)

bin3<- as.numeric(traindata[,1]==0)

bin4<- as.numeric(traindata[,1]==2)

bin5<- as.numeric(traindata[,1]==4)

Iytrain<-cbind(bin1,bin2,bin3,bin4,bin5)

#re-code the response variable into binary for the test data
```

```
bin1<-as.numeric(testdata[,1]==-7)

bin2<- as.numeric(testdata[,1]==-5)

bin3<- as.numeric(testdata[,1]==0)

bin4<- as.numeric(testdata[,1]==2)

bin5<- as.numeric(testdata[,1]==4)

Iytest<-cbind(bin1,bin2,bin3,bin4,bin5)

#specify the number of classes and the rule to grow the stump

K=5

stumprule=rpart.control(cp=-1,maxdepth=1,minsplit=0)

#creates the empty list to generate K boosting models for each class

Fs<-list()

#apply Adaboost.M1 using stumps and 300 boosting iterations

#the code below generates the K models where $model$F[[2]] extracts the

#final test sum and $model$F[[1]] extracts the final training sum

M=300

for(i in 1:K){

 Fs[[i]]<-ada(Y~.,data=data.frame(Y=Iytrain[,i],X1=traindata[,2],X2=traindata[,3]),

 control=stumprule, iter=M,test.x=testdata[,-1],test.y=Iytest[,i])

}

temptestpred<-list()

temptrainpred<-list()
```

```
boostumptesterror=0

boostumptrainerror=0

#codes the response variable in binary form of the test data so that

#it is easier to manipulate

# i.e column 1=-7, 2=-5, 3=0, 4=2, 5=4

ytest<-apply(Iytest,1,which.max)

ytrain<-apply(Iytrain,1,which.max)

#note that the code before this point will compute relatively quickly

#however the code below will require some time to compute as noted before

for(j in 1:M){

for(ki in 1:K){

 #type="F" below outputs the tree (alpha) weights

 temptestpred[[ki]]=predict(Fs[[ki]],newdata=

 data.frame(Y=Iytest[,ki],X1= testdata[,2], X2= testdata[,3]),type="F",n.iter=j)

 temptrainpred[[ki]]=predict(Fs[[ki]],newdata=

 data.frame(Y=Iytrain[,ki],X1= traindata[,2], X2= traindata[,3]),type="F",n.iter=j)

 }

#selects the column index with the highest alpha weight

predstest<- sapply(1:testN,function(i)which.max(c(temptestpred[[1]][i],

temptestpred[[2]][i], temptestpred[[3]][i], temptestpred[[4]][i], temptestpred[[5]][i])))

predstrain<- sapply(1:trainN,function(i)which.max(c(temptrainpred[[1]][i],
```

```
temptrainpred[[2]][i], temptrainpred[[3]][i], temptrainpred[[4]][i], temptrainpred[[5]][i])))

#calculate the error rates

temptest<-ifelse(ytest!=predstest,1,0)

temptrain<-ifelse(ytrain!=predstrain,1,0)

boostumptesterror[j]=sum(temptest)/testN

boostumptrainerror[j]=sum(temptrain)/trainN

}

#plot the test and training error

boosting_iterations<-1:M

matplot(boosting_iterations,cbind(boostumptesterror, boostumptrainerror),

type="l",col=c("blue","red"), main="Testing And Training Error",xlab="Iterations", ylab="Error")

legend("topright", c("test","train"), col = c("blue","red"), lty=1:2)

#print the test error rate at the final boosting iteration

boostumptesterror[M]

#Output - [1] 0.35
```

B.4.3 Part 3

```
#alternative method using Adabag package

#convert the response variable to string and create the test and training data

ytrainchar=as.character(traindata[,1])

datatrainchar=data.frame(Y=ytrainchar, X1=traindata[,2],X2=traindata[,3])
```

```
ytestchar=as.character(testdata[,1])
datatestchar=data.frame(Y=ytestchar, X1=testdata[,2],X2=testdata[,3])
#select the tree growing rule to produce the alpha tree at the cp value shown earlier
alpharule=rpart.control(cp=0.003)
#call the boosting routine with M=300 boosting iterations as before
#note "Freund" calculates the tree weights alpha as we have shown above and
#boos=TRUE specifies whether resampling is performed using the iterative weights and
#therefore boos=FALSE means all observations are included
boostwotrain <- boosting(Y ~.,data=datatrainchar,mfinal=M, coeflearn="Freund", boos=FALSE,
control=alpharule)
boostwotest <- predict.boosting(boostwotrain,newdata=datatestchar)
boostwotest$error
#Output: [1] 0.38
#graph of the average error rate at each boosting iteration
errorevol(boostwotrain,newdata=datatrainchar)->evol.train
errorevol(boostwotrain,newdata=datatestchar)->evol.test
#comparing error evolution in training and test set
plot(evol.test$error, type="l", main="Testing and Training Error",
xlab="Iterations", ylab="Error", col = "red")
lines(evol.train$error, cex = .5 ,col="blue", lty=2)
legend("topright", c("test","train"), col = c("red", "blue"), lty=1:2)
```

B.5 R code used for the real boosted multi-classification example

B.5.1 Part 1

#Due to the size of the data-set this code should be run on R as soon as it has been loaded prior to running any other commands

```
options(java.parameters = "-Xmx4g" ) #increases the amount of memory to use

# Load the rpart, ada and XLconnect packages

library(XLConnect)

library(rpart)

library(ada)

# Specify the file location for the training and test data

digitsfile <- system.file("demoFiles/digitsdata.xlsx", package = "XLConnect")

digitsfiletest <- system.file("demoFiles/digitstestdata.xlsx", package = "XLConnect")

# Load the workbooks

digitswb <- loadWorkbook(digitsfile)

digitswbtest <- loadWorkbook(digitsfiletest)

# Read worksheet “zip”

digitsdata <- readWorksheet(digitswb, sheet = "zip")

digitsdatatest <- readWorksheet(digitswbtest, sheet = "zip")

#re-code the response variable into binary for the training data
```

```
bin0<-as.numeric(digitsdata[,1]==0)

bin1<-as.numeric(digitsdata[,1]==1)

bin2<-as.numeric(digitsdata[,1]==2)

bin3<-as.numeric(digitsdata[,1]==3)

bin4<-as.numeric(digitsdata[,1]==4)

bin5<-as.numeric(digitsdata[,1]==5)

bin6<-as.numeric(digitsdata[,1]==6)

bin7<-as.numeric(digitsdata[,1]==7)

bin8<-as.numeric(digitsdata[,1]==8)

bin9<-as.numeric(digitsdata[,1]==9)

Iytrain<-cbind(bin0,bin1,bin2,bin3,bin4,bin5,bin6,bin7,bin8,bin9)

#re-code the response variable into binary for the test data

bin0<-as.numeric(digitsdatatest[,1]==0)

bin1<-as.numeric(digitsdatatest[,1]==1)

bin2<-as.numeric(digitsdatatest[,1]==2)

bin3<-as.numeric(digitsdatatest[,1]==3)

bin4<-as.numeric(digitsdatatest[,1]==4)

bin5<-as.numeric(digitsdatatest[,1]==5)

bin6<-as.numeric(digitsdatatest[,1]==6)

bin7<-as.numeric(digitsdatatest[,1]==7)

bin8<-as.numeric(digitsdatatest[,1]==8)
```

```
bin9<-as.numeric(digitsdatatest[,1]==9)

Iytest<-cbind(bin0,bin1,bin2,bin3,bin4,bin5,bin6,bin7,bin8,bin9)

#specify the number of classes and the rule to grow the stump

K=10

stumprule=rpart.control(cp=-1,maxdepth=1,minsplit=0)

#creates the empty list to generate K boosting models for each class

Fs<-list()

#extracts the predictor variable dataset excluding the response variable

xtest=data.frame(digitsdatatest[,-1])

testN=nrow(xtest)

xtrain=data.frame(digitsdata[,-1])

trainN=nrow(xtrain)

M=100 #set the number of boosting iterations

#create the Adaboost.M1 models using stumps

#the code below generates the K models

for(i in 1:K){

Fs[[i]]<-ada(Y~.,data=data.frame(Y=Iytrain[,i],xtrain),control=stumprule,iter=M,test.x=xtest,test.y=Iytest[,i])

}

#create and initialize the temporary variables and error variables

temptestpred<-list()

temptrainpred<-list()
```

```
boostumptesterror=0

boostumptrainerror=0

#codes the response variable in binary form of the test data so that it is easier to manipulate

# i.e column 1=0, 2=1, 3=2, 4=3, 5=4, 6=5, 7=6, 8=7, 9=8, 10=9

ytest<-apply(Iytest,1,which.max)

ytrain<-apply(Iytrain,1,which.max)

#the nested loops below calculates the weighted predictions for the K responses and selects the highest as the final prediction

for(j in 1:M){

for(ki in 1:K){

#type="F" below outputs the tree (alpha) weights multiplied by the response variable (ensemble average)

temptestpred[[ki]]=predict(Fs[[ki]],newdata=data.frame(Y=Iytest[,ki],xtest),type="F",n.iter=j)

temptrainpred[[ki]]=predict(Fs[[ki]],newdata=data.frame(Y=Iytrain[,ki],xtrain),type="F",n.iter=j)

}

#selects the column index with the highest ensemble average

predstest<- sapply(1:testN,function(i)which.max(c(temptestpred[[1]][i], temptestpred[[2]][i], temptestpred[[3]][i], temptestpred[[4]][i], temptestpred[[5]][i], temptestpred[[6]][i], temptestpred[[7]][i], temptestpred[[8]][i], temptestpred[[9]][i], temptestpred[[10]][i])))
```

```
predstrain<- sapply(1:trainN,function(i)which.max(c(temptrainpred[[1]][i], temptrainpred[[2]][i],
temptrainpred[[3]][i], temptrainpred[[4]][i], temptrainpred[[5]][i], temptrainpred[[6]][i], temptrainpred[[7]][i],
temptrainpred[[8]][i], temptrainpred[[9]][i], temptrainpred[[10]][i])))
#calculates the error at each boosting iteration
temptest<-ifelse(ytest!=predstest,1,0)
temptrain<-ifelse(ytrain!=predstrain,1,0)
boostumptesterror[j]=sum(temptest)/testN
boostumptrainerror[j]=sum(temptrain)/trainN
}
#plot the test and training error
boosting_iterations<-1:M
matplot(boosting_iterations,cbind(boostumptesterror, boostumptrainerror),type="l",col=c("blue","red"),
main="Testing And Training Error",xlab="Iterations", ylab="Error")
legend("topright", c("test","train"), col = c("blue","red"), lty=1:2)
#print the test and training error rate at the final boosting iteration
boostumptesterror[M]
#Output: [1] 0.1669158
boostumptrainerror[M]
#Output: [1] 0.1220683
```

B.5.2 Part 2

```
#alternative method using Adabag
        library(adabag)
        #convert the response variable to string and create the test and training data
        ytrainchar=as.character(digitsdata[,1])
        datatrainchar=data.frame(Y=ytrainchar,xtrain)
        ytestchar=as.character(digitsdatatest[,1])
        datatestchar=data.frame(Y=ytestchar,xtest)
        #select the tree growing rule to produce the alpha tree at the cp value shown earlier
        alpharule=rpart.control(cp=0.00082008)
        boostwotrain <- boosting(Y ~.,data=datatrainchar,mfinal=M, coeflearn="Freund", boos=FALSE,
control=alpharule)
        boostwotest <- predict.boosting(boostwotrain,newdata=datatestchar)
        boostwotest$error
        #Output [1] 0.05929248
        #graph of the average error rate at each boosting iteration
        errorevol(boostwotrain,newdata=datatrainchar)->evol.train
        errorevol(boostwotrain,newdata=datatestchar)->evol.test
        #comparing error evolution in training and test set
        plot(evol.test$error, type="l", main="Testing and Training Error",
        xlab="Iterations", ylab="Error", col = "red")
```

```
lines(evol.train$error, cex = .5 ,col="blue", lty=2)

legend("topright", c("test","train"), col = c("red", "blue"), lty=1:2)
```