Signature of Professor

TECHNISCHE UNIVERSITÄT WIEN
Vienna | Austria

Master Thesis

Boosting Classifications with Imbalanced Data

Submitted at the TU Wien,
Institute of Statistics and Mathematical Methods in Economics

under the supervision of
Univ. Prof. Dipl.-Ing. Dr.techn. Peter Filzmoser

by
Philipp Rudolf Bauer

Date

Signature

Acknowledgements

Firstly, I would like to express my gratitude to Prof. Filzmoser for his kind support and helpful advice in writing this thesis.

I would also like thank my proofreaders, Prof. Filzmoser, my mother and my father, who spent a significant amount of time to guarantee that this thesis is almost free of spelling errors.

A special thanks to my father who not only provided critical input but who also encouraged and motivated me in my studies.

Also, I would like to thank my friends and colleagues for their support and friendship during my time at university.

Lastly, I would like to thank both my parents for supporting me in all my endeavours, being unnaturally patient with me at times and being there for me whenever I needed them.

Abstract

Boosting is an ensemble method which uses a *"weak"* classifier to create a *"strong"* one, based on the theory of Robert Schapire's work in 1990 (see Schapire 1990). It appears similar to bagging yet is fundamentally different.

This thesis will start with a short introduction followed by a chapter describing the theory and methodology behind boosting. This is followed by a chapter presenting a set of boosting algorithms, applicable to binary, multi-class and regression problems.

The major focus of this thesis is to examine the performance of boosting algorithms on imbalanced data sets. The issue with these data sets is that classifiers tend to emphasize the larger classes, which leads to significant class distribution skews. An established general solution to this issue is to apply sampling methods. After introducing these, the simulations chapter demonstrates that boosting algorithms work well with minority sampling in binary classification, whereas majority sampling appears to be preferable in the multi-class problem. However, it will be shown that in the multi-class setting the inbuilt re-weighting of hard to classify problems of the boosting algorithms AdaBoost.M1 and SAMME, is sufficient to handle imbalances in the data set, without any sampling necessary.

Contents

Chapter 1

Introduction

This introductory chapter establishes the basics behind classification and regression problems. It introduces ensemble methods and the basics behind boosting while comparing it's modus operandi with other ensemble methods such as bagging or random forests.

1.1 Function Estimation

1.1.1 Basic Principle of Regression and Classification

A basic prediction problem can be described as a system consisting of an input $x \in X$ and an output $y \in Y$, where x and y are realisations of random variables X and Y respectively. Using a training sample $\{(x_i, y_i)\}_{i=1,\dots,n}$ of size n, the goal is to find a function, an estimation, $F(X)$ to predict Y as good as possible given values of the input X. This approximation must be chosen such that it minimizes the expected value of a specified loss function $L(Y, F(X))$ over their joint distribution:

$$\hat{F} = \arg\min_F E_{X,Y}(L(Y, F(X))) = \arg\min_F E_X \left[E_{Y|X}(L(Y, F(X)))|X \right]. \tag{1.1.1}$$

Some of the most well known and convenient loss functions are squared error $L(Y, F(X)) = (Y - F(X))^2$ and absolute loss $L(Y, F(X)) = |Y - F(X)|$, both frequently used in regression and exponential loss $L(Y, F(X)) = e^{-YF(X)}$ and binomial log-likelihood $L(Y, F(X)) = log(1 + e^{-2YF(x)})$ used in classification.

Depending on the output set Y, the individual outputs vary and can be a quantitative value $Y \subset \mathbb{R}$, such as a measurement in height or weight, or a qualitative distinction $Y = \{1, 2, \dots, k\}$, such as blood-type. Here k denotes the number of possible realisations of Y, referred to as *classes* or *labels*. If for example $k = 2$, then the classification problem at hand is referred to as a *binary* problem. These classes are often descriptive labels rather and are often denoted as numbers for sake of simplicity. Due to their distinctiveness, they are called *categorical* or *discrete* variables. The prediction is termed a *regression* if dealing with quantitative outputs and a *classification* when dealing with categorical outputs.

For now let $(x_1, \dots, x_p)$ denote the independent input and $y \in \mathbb{R}^p$ as the dependent output, both of dimension p. Then the classical *linear regression model* would be

$$y = F(x_1, \dots, x_p; \beta) + \epsilon = \beta_0 + \sum_{j=1}^{p} x_i \beta_i \tag{1.1.2}$$

with an error of ϵ and the parameters, or coefficients, $\beta = (\beta_0, \dots, \beta_p)$ which describe the position of the function F in the p-dimensional space. The goal now is to find an estimate $\hat{y} = F(x_1, \dots, x_p; \hat{\beta})$ to approximate the true y. A good estimate $\hat{y}$ would have to have a small error ϵ. Hence, a linear model can be solved by minimizing the error term with respect to the coefficients β.
Given a data set $\{y_i, x_{i1}, \dots, x_{ip}\}_{i=1}^{n}$ of size n and using matrix notation $X = (x_1, \dots, x_n)^\top$, $y = (y_1, \dots, y_n)^\top$ and $\epsilon = (\epsilon_1, \dots, \epsilon_n)^\top$, with $x_i = (x_{i1}, \dots, x_{ip})^\top$ for $i = 1 \dots, n$ the model can be written as

$$y = X\beta + \epsilon.$$

Using the *method of least squares*, where those coefficients are chosen that minimize the residual sum of squares

$$RSS(\beta) = \sum_{i=1}^{n} (y_i - x_i^\top \beta)^2 = (y - X\beta)^\top (y - X\beta),$$

the following solution for β is calculated as

$$\hat{\beta} = (X^\top X)^{-1} X^\top y.$$

In the case of classification, the problem is identifying to which of a set of categories, referred to $j = 1, \dots, k$ classes, the observations $(y_i)_{i=1,\dots,n}$ belong to. Observations within a class should have similar properties, while those of other classes should differ. Therefore, if the estimator $F(\cdot)$ will take values in the discrete set $Y = \{1, 2, \dots, k\}$, then the input space X can be categorized into k regions accordingly. For example, given a binary $k = 2$ problem, one way is to treat the target $Y = \{-1, 1\}$ as a quantitative output. As the estimates will generally lie within the interval $[-1, 1]$, the class assigned to an observation x will be determined whether or not $\hat{y} > 0$. Using more than one boarder, this approach can be generalized to k classes.

1.1.2 Additive Models

Focusing on the regression problem, where the response y is quantitative, with a number of n observations, x and y have some joint distribution, and the objective is to try to model the mean $E(y|x) = F(x)$. The *additive model* (see Friedman 2001) has the form

$$F(x) = \sum_{i=1}^{n} f_i(x_i), \tag{1.1.3}$$

where each input variable x_i has a separate function $f_i(x_i)$ for every $i = 1, \dots, n$.

A more general procedure is to restrict the additive model $F(x)$ to be a member of a parametrized class of functions $F(x; P)$, where P is a finite $(t = 1, \ldots, T)$ set of parameters whose joint values identify individual class members. The $f_i(\cdot)$ are to be simple parametrized functions $f_t(x) = \alpha_t h(x, \beta_t)$ of parameters β_t and multipliers α_t. Then the additive model in (1.1.3) becomes

$$F_T(x) = \sum_{t=1}^{T} \alpha_t h(x, \beta_t). \tag{1.1.4}$$

Generally, these sets of functions $\{h(x, \beta_t)\}_{t=1}^{T}$ are called "*basis functions*" since they span a function subspace.
The individual terms differ in the joint values β_t chosen for these parameters. Such functions (1.1.4) are the foundations of many function approximations.

1.2 Ensemble Methods

In statistics, enemble methods use a combination of multiple base estimators or classification algorithms in order to achieve a better predictive performance compared to that of single algorithms. While there is an increase in computation time for ensemble methods as compared to using a single algorithm, they compensate the weak performance of a single algorithm with additional computations. Basically all common types of ensembles can be separated into two categories (see Pedregosa et al. 2011):

1. The first and straight-forward approach to ensemble methods are **averaging methods**, where a set of various estimators are built independently from one another and then the average of their final predictions is used. This procedure leads to a combined estimator with a reduced variance. For example, probably the two most well known and established averaging methods are:
   Bagging, derived from bootstrap aggregating, is an ensemble learning method whereby a multitude of different regression or classification models are evaluated and their results aggregated repeatedly using weights depending on the individual models.
   Random forests are an ensemble learning method for classification and regression that construct a multitude of decision trees and output the class that is either the mode of the classes or mean prediction of the individual trees.

2. Instead of building the estimators independently as was done in bagging, **boosting** creates an ensemble by sequentially training each new estimator to emphasize the training instances that previous models misclassified.

The general principal behind boosting, is starting with a *base procedure*, also referred to as a *weak learner*, to generate multiple predictions from re-weighted data, which are then aggregated (linear/convex combination, majority voting) to obtain the final prediction (see Bühlmann and Geer 2011).

The first step is to choose which *weak learner* to use for the construction of a first estimate or prediction $\hat{f}(\cdot)$ based on the input of data $(X_i, Y_i)_{i=1,\ldots,n}$, with covariates X_i and responses Y_i (independent and dependant variables, respectively).

This crucial choice of *base procedure*, which for example could be linear regression, component wise smoothing splines or regression trees, source of which will be discussed in detail in a later chapter (refer to Section 2.4.1).

The chosen procedure is then applied to data $(x_i, y_i)_{i=1,\ldots,n}$ to calculate the predicition $\hat{f}(\cdot)$ and use $\hat{f}(\cdot)$ to obtain weights for the n sample points. Thus, creating re-weighted data, upon which the *base procedure* is used again to create new weights (see Bühlmann and Geer 2011).

$$\begin{tikzcd}
(x_i, y_i)_{i=1,\ldots,n} \arrow[r, "\text{base procedure}"] & \hat{f}(\cdot) \\
\text{re weighted data 1} \arrow[r, "\text{base procedure}"] & \hat{f}_1(\cdot) \\
\text{re weighted data 2} \arrow[r, "\text{base procedure}"] & \hat{f}_2(\cdot) \\
\vdots \arrow[r, "\text{base procedure}"] & \vdots \\
\text{re weighted data T} \arrow[r, "\text{base procedure}"] & \hat{f}_T(\cdot)
\end{tikzcd}$$

After this cycle is repeated a number of $t = 1, \ldots, T$ (iterations) times the T predictors are aggregated using suitable coefficients $\alpha_1, \ldots, \alpha_T$

$$\hat{F}(\cdot) = \sum_{t=1}^{T} \alpha_t \hat{f}_t(\cdot) \tag{1.2.1}$$

to obtain the boosting predictor $\hat{F}(\cdot)$.

The specification of the re-weighting mechanism, the combination of linear coefficients $(\alpha_t)_{t=1,\ldots,T}$, choice of the *base procedure* and other various choices characterize different boosting schemes. What all of them have in common is the fact, that the data weights in step t only depend on the results of the previous iteration $t-1$.

Chapter 2

Boosting

This chapter starts off by discussing some general ideas on boosting as an ensemble method and introducing the first algorithm, AdaBoost. Its properties, such as bounds of its training and generalization error will be presented. Following that, the core concept, examples of weak learners and the importance of regularization are discussed. Furthermore, two different interpretations of boosting will be shown, to merit the approach of boosting it will demonstrated, that the AdaBoost algorithm can be represented as a steepest decent algorithm in a function space.

2.1 PAC Models

To further elaborate on some basic ideas of boosting and ensemble methods this section will focus on binary classification as opposed to multiple classification and regression, which will be elaborated in Section 3.2.

In a binary classification setting, given a set of observations the goal is to find a way of assigning an object to one of two different classes. Using the notation of X (often $X \subseteq \mathbb{R}^p$) being the input space of observations $x_i \in X$ and denoting $Y = \{-1, +1\}$ as the possible outputs of a hypothesis. The objective is to estimate a function $f : X \to Y$, using the training data pairs of size n of the input and output generated independently at random from an unknown probability distribution $P(x, y)$, $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^p \times \{-1, +1\}$, such that f will correctly predict future unseen samples (x, y). Thus, a label for an input x will be assigned as $f(x)$. Meir and Rätsch (see Meir and Rätsch 2003) defined, in the case of $Y = \{-1, +1\}$, this as a *hard classifier.* On the other hand, the case if $Y = \mathbb{R}$, where the label assigned corresponds to $sign(f(x))$, this is referred to as a *soft classifier.* The true performance of a classifier f is assessed by the risk function

$$R(f) = E_P L(f) = \int L((f(x), y))dP(x, y) \ , \tag{2.1.1}$$

where L denotes an appropriately chosen loss function (see Section 1.1). An appropriate choice of loss function $L(\cdot)$ in the case of binary classification is the **0/1-loss** defined as $L(f(x), y) = \mathbb{1}(yf(x) \leq 0)$, where $\mathbb{1}(E) = 1$ if the event E occurs and zero otherwise. As the risk $R(f)$ measures the expected loss with respect to examples which were not observed in the training set, it is termed as the **generalization error**. In machine learning and statistical learning theory the generalization error (also called out-of-sample error) is a measure of how accurately an algorithm is able to predict outcome values for previously unseen data.

Since the probability distribution $P(x, y)$ in (2.1.1) is unknown, the risk cannot be minimized directly to obtain the optimal function f. Thus, f has to be approximated based on the properties of the function class $\mathcal{F}$ and the available information $\{(x_i, y_i)\}_{i=1,\ldots,n}$. A simple solution to this problem would be using empirical risk

$$\hat{R}_n(f) = \frac{1}{n}\sum_{i=1}^{n} L(f(x_i), y_i) \tag{2.1.2}$$

and calculating the minimum of it, instead of using the minimum of the risk function (2.1.1). As the law of large numbers states, that the average of the results obtained from a large number of trials should be close to the expected value, one can infer that the empirical risk (2.1.2) converges to the risk (2.1.1): $\hat{R}(f) \to R(f)$ as $n \to \infty$. This however, does not guarantee that the function which satisfies $\tilde{f}_n = \arg\min_{f\in\mathcal{F}} \hat{R}_n(f)$ is the asymptotic minimum of $R(f)$. A sufficient condition to achieve convergence is the uniform convergence of $\hat{R}_n$ over $\mathcal{F}$ to $R(f)$. If not, a function f with arbitrarily small error on the sample set can be chosen, which leads to poor generalization. Even providing sufficient conditions such that the function which minimizes the empirical risk will perform optimally, for small sample sizes large deviations are possible and over-fitting might occur and a small generalization error cannot be obtained by simply minimizing the training error (2.1.2).

Therefore, to avoid the dilemma of over-fitting mentioned above, the complexity of the class of functions $\mathcal{F}$ needs to be controlled which is called **regularization**. The intent is to restrict the size of function class $\mathcal{F}$, such that the complexity of the chosen f is reduced. Regularization is justified as it attempts to impose Occam's razor[1] on the solution. It will be shown in Section 2.3 that a complex function is less preferable than a "simple" function that explains less of the data.

It seems that boosting algorithms, such as (1.2.1), when using an ensemble of many weak learners, produce an increasingly more complex function. Surprisingly this is not the case under certain conditions, as will be shown in Theorem 2.3.3. Again one has to be careful, as this observation could lead to thinking, that due to the limitation of the complexity of a boosting algorithm, over-fitting does not occur, which is not the case and *regularization* will be necessary (see Section 2.5).

Before introducing the first boosting algorithm AdaBoost, **PAC**-models, which stands for *"Probably Approximately Correct"*, are introduced. Here a solution should be close enough (*approximately*) in most of the cases (*probably*). The definitions taken from Meir and Rätsch (see Meir and Rätsch 2003) of a strong and weak PAC algorithm are as follows:

Definition 2.1.1 *Given a sample of n data points $\{(x_i, y_i)\}_{i=1}^{n}$, with x_i random and independently generated from some distribution $P(x)$, $y_i = f(x_i)$ and f is from a*

[1] Occam's razor states that among competing hypotheses, the one with the fewest assumptions should be selected (*"The simplest solution is the most likely"*). This principle is attributed to William of Ockham (c. 1287–1347)

class of binary functions $\mathcal{F}$. Now if for every distribution P, every $f \in \mathcal{F}$ and every $0 \leq \epsilon$, $\delta \leq 1/2$ with probability larger than $1 - \delta$, the algorithm outputs a hypothesis h such that $P[h(x) \neq f(x)] \leq \epsilon$, then the algorithm is a ***strong PAC learning algorithm****.*

Definition 2.1.2 *A* ***weak PAC learning algorithm*** *is essentially the same as a strong one and is defined analogously, except that it is only required to satisfy the conditions for particular ϵ and δ, rather than all pairs.*

2.2 AdaBoost

The AdaBoost algorithm was formulated 1996 by Yoav Freund and Robert Schapire in their publication *"A Decision-Theoretic Generalization of On-Line Learning and an Application to boosting"* (see Freund and Schapire 1997) and they received the Gödel Prize[2] in 2003 for their work. They called it such, short for *ada*ptive *boost*ing, as unlike previous algorithms it adjusts adaptively to the errors of the weak hypothesis returned by the *weak learner*. It is arguably the most well known boosting algorithm and is generally considered as a first step towards more practical boosting algorithms.

The algorithm considers the input of data $(x_1, y_1), \ldots, (x_n, y_n)$, with covariates x_i and labels y_i chosen randomly according to a fixed but unknown distribution P on $X \times Y$, where X is a p-dimensional space and Y is the label set. In the case of AdaBoost the label set will be $Y = \{-1, 1\}$. The extensions to the multi class and real valued case will be discussed at a later stage (see Chapter 3). The goal is to learn to predict the correct label y given an observation x, by finding a hypothesis H which is consistent with most of the sample. The case refereed to as "overfitting", where a hypothesis is accurate within the training set but inaccurate with data outside of the training sample, can occur. This can be avoided by restricting the hypothesis to be simple (see Section 2.3).

One of the main ideas of the algorithm is to maintain a distribution or a set of weights over the training set. The weight of this distribution on training example $i = 1, \ldots, n$ on round $t = 1, \ldots, T$ is denoted by $w_t(i)$. Initially, all weights are set equally, but on each round, the weights of incorrectly classified examples are increased so that the weak learner is forced to focus on the hard to classify examples in the training set.

AdaBoost calls a given *weak learner* repeatedly in a series of $t = 1, \ldots, T$ iterations. At each step t of the iteration, the weak learner's job is to produce a weak hypothesis $h_t : X \to \{-1, 1\}$, appropriate for the distribution w_t, with which the individual hypothesis $(h_t(x_i))_{i=1,\ldots,n}$ for each sample in the training set are created. Then the weights α_t for each individual h_t are chosen. This is done by minimizing the sum of the training error

[2]The Gödel Prize is an annual prize for outstanding papers in the area of theoretical computer science.

$$\epsilon_t(h_t, w_t) = P_{i \sim w_t}[h_t(x_i) \neq y_i] = \sum_{i:h_t(x_i) \neq y_i} w_t(i) \tag{2.2.1}$$

which measures the goodness of a base procedure with respect to the distribution w_t (see Freund and Schapire 1999). Generally this can be written as

$$\epsilon_t = \sum_i L[H_{t-1}(x_i) + \alpha_t h(x_i)], \tag{2.2.2}$$

where L is a loss/error function, $H_{t-1}(x)$ the boosted classifier up to the previous iteration and $\alpha_t h(x)$ is the weighted weak learner which is considered to be added to the final classifier.

Assuming exponential loss, the weights α_t, which minimize the training error (2.2.2), can be derived. As the boosted classifier of iteration $t-1$ will be of the form

$$H_{t-1}(x_i) = \alpha_1 h_1(x_i) + \ldots + \alpha_{t-1} h_{t-1}(x_i),$$

it follows that the classifier of the t'th iteration can then be written as

$$H_t(x_i) = H_{t-1}(x_i) + \alpha_t h_t(x_i). \tag{2.2.3}$$

Thus, the total error ϵ of H_t will be the sum of its exponential loss on each data point and from (2.2.2) one obtains

$$\epsilon = \sum_{i=1}^{n} e^{-y_i H_t(x_i)} \tag{2.2.4}$$

where

$$-y_i H_t(x_i) = \begin{cases} -1 & \text{if correctly classified } y_i = H_t(x_i) \\ 1 & \text{if incorrectly classified } y_i \neq H_t(x_i) \end{cases}$$

determines a true or a false classification. While using the equality of (2.2.3) and substituting $w_1(i) = 1$, $w_t(i) = e^{-y_i H_{t-1}(x_i)}$ into the total error (2.2.4) and then splitting the sum into the correctly and the incorrectly classified cases by h_t the following is obtained

$$\epsilon = \sum_{i=1}^{n} w_t(i) e^{-y_i \alpha_t h_t(x_i)} = \sum_{i:h_t(x_i) \neq y_i} w_t(i) e^{\alpha_t} + \sum_{i:h_t(x_i) = y_i} w_t(i) e^{-\alpha_t}. \tag{2.2.5}$$

Differentiating those two sums and setting the differential to zero

$$\frac{d\epsilon}{d\alpha_t} = 0 = \sum_{i:h_t(x_i) = y_i} w_t(i) e^{\alpha_t} - \sum_{i:h_t(x_i) \neq y_i} w_t(i) e^{-\alpha_t}.$$

and solving this for α_t yields the following result for the alphas

$$\alpha_t = \frac{1}{2} ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$$

Algorithm 1 AdaBoost

```
1: Initialize the weights for the individual sample points as $w_1(i) = 1/n$ for all $i = 1, \dots, n$.
2: for $t = 1$ to $T$ do
3:     Using a weighted fitting, train the classifier with respect to the weighted data and obtain the classifier $h_t$
4:     Calculate the misclassification rate of $h_t$
           $$\epsilon_t = \sum_{i:h_t(x_i)\neq y_i} w_t(i) \ / \ \sum_{i=1}^{n} w_t(i)$$
5:     Set
           $$\alpha_t = \frac{1}{2} ln \left( \frac{1-\epsilon_t}{\epsilon_t} \right)$$
6:     Update the weights
           $$w_{t+1} = \frac{w_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if correctly classified } y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if incorrectly classified} y_i \neq h_t(x_i) \end{cases} = \frac{w_t(i) e^{-\alpha_t y_i h_t(x_i)}}{Z_t},$$
       where $Z_t$ is a normalization factor, chosen such that $w_t$ will be a distribution $\sum_{t=1}^{N} w_{t+1}(i) = 1$
7: end for
8: Output the final hypothesis
           $$H(x) = sign \left( \sum_{t=1}^{T} \alpha_t h_t(x) \right)$$
```

as the negative *logit* function of ϵ_t multiplied by $\frac{1}{2}$.

Therefore, given a set of data $(x_1, y_1), \dots, (x_n, y_n)$, where $x_i \in X$ and $y_i \in Y = \{-1, +1\}$, AdaBoost works as shown in Algorithm 1.

To choose the classifier h_t in step 3 one has to consider the following. The training error sum in (2.2.5) can also be written as

$$\epsilon_t = \sum_{i=1}^{n} w_t(i) e^{-\alpha_t} + \sum_{i:h_t(x_i)\neq y_i} w_t(i)(e^{\alpha_t} - e^{-\alpha_t}).$$

Here only the second sum is dependent on h_t. Thus, it follows that the h_t which minimizes $\sum_{i:h_t(x_i)\neq y_i} w_t(i)$ also minimizes the training error ϵ. In other words, the weak classifier with the lowest weighted error is chosen.

After the first step of initializing the weights of the sample points and receiving the weak hypothesis h_t in step 3, AdaBoost chooses a parameter α_t as in the pseudo code provided above. Intuitively, α_t measures the importance that is assigned to h_t. It is to note, that $\alpha_t \geq 0$ if $\epsilon_t \leq \frac{1}{2}$ and α_t gets smaller the larger ϵ_t gets and vice versa.

In step 6 the distribution of w_{t+1} is next updated as shown in the code. The effect of this rule is to increase the weight of examples misclassified by h_t, and to decrease the weight of correctly classified examples. Hence, this forces the following classifiers to focus on those x_i which are "hard" to classify.

Lastly, the final or combined hypothesis H computes the sign of a weighted combination of weak hypotheses

$$F(x) = \sum_{t=1}^{T} \alpha_t h_t(x) \tag{2.2.6}$$

where α_t is the weight assigned to h_t. This is equivalent to saying that H is computed as a weighted majority vote of the weak hypotheses h_t where each is assigned weight α_t.

2.3 Boosting Properties

2.3.1 Training Error Bounds

If one considers a hypothesis which randomly guesses each observation's class, it is intuitively obvious that the hypothesis has an error rate of 1/2 (see Definition 2.4.1). Hence, we can write the error ϵ_t of h_t as $\frac{1}{2} - \eta_t$. We call η_t the *edge*, which measures how much better than random are h_t's predictions. Freund and Schapire (see Freund and Schapire 1997) proved the following theorem, concerning the training error (the fraction of mistakes on the training set) of the final hypothesis H:

Theorem 2.3.1 *Assuming the weak learner generates hypotheses with errors $\epsilon_1, \dots, \epsilon_T$ according to the AdaBoost (Algorithm 1) in step 4. Then the final hypothesis H outputed by the algorithm has an error bounded by*

$$\epsilon_T = P_{i\sim w_t}[H_T(x_i) \neq y_i] \leq 2^T \prod_{t=1}^{T} \sqrt{\epsilon_t(1-\epsilon_t)}. \tag{2.3.1}$$

Thus, if each weak hypothesis is slightly better than random such that $\eta_t > 0$, then the training error drops exponentially fast.

Replacing ϵ_t with $1/2-\eta_t$, the bound of the error term in (2.3.1) can be rewritten as follows,

$$2^T \prod_{t=1}^{T} \sqrt{\epsilon_t(1-\epsilon_t)} = \prod_{t=1}^{T} \sqrt{1-4\eta_t^2} = exp\big(-\sum_{t=1}^{T} KL(1/2||1/2-\eta_t)\big) \leq exp\big(-2\sum_{t=1}^{T} \eta_t^2\big), \tag{2.3.2}$$

where $KL(a||b) = aln(a/b) + (1-a)ln\big((1-a)/(1-b)\big)$ is the *Kullback-Leibler divergence*[3]. In the case where the errors of all the hypotheses are equal to $\epsilon_t = 1/2 - \eta$, for a set $\eta > 0$, the Equation (2.3.2) is further simplified as

[3]Also referred to as *relative entropy*, it is a measure of how one probability distribution diverges from a second expected probability distribution.

$$\epsilon_T \leq (1-4\eta^2)^{T/2} = exp(-T \cdot KL(1/2||1/2-\eta)) \leq e^{-2T\eta^2}, \qquad (2.3.3)$$

where the last inequality is achieved by a form of the Chernoff bound for the probability that less than $T/2$ coin flips turn out "heads" in T tosses of a random coin whose probability for "heads" is $1/2-\eta$.
With this last Equation (2.3.3) a **number of iterations** sufficient for AdaBoost to achieve error ϵ of H_T can be obtained as (see Freund and Schapire 1997)

$$T = \left\lceil \frac{1}{KL(1/2||1/2-\eta)} ln\frac{1}{\epsilon} \right\rceil \leq \left\lceil \frac{1}{2\eta^2} ln\frac{1}{\epsilon} \right\rceil. \qquad (2.3.4)$$

2.3.2 Generalization Error Bounds

Theorem 2.3.1 guarantees that the error of the final hypothesis H on the given training sample is small. This however often does not suffice, as the generalization error of H, which is the error of H over the whole instance space X instead of just the input data $(x_1, y_1), \ldots, (x_n, y_n)$, has to be taken into consideration. The generalisation error will be denoted as ϵ_g.

Freund and Schapire (see Freund and Schapire 1997) used the training error ϵ, the sample size n, the VC-dimension[4] d of the weak hypothesis space and the number of boosting rounds T to bound the generalization error of the final hypothesis. To get the ϵ_g close to the training error ϵ, the first approach is to restrict the weak learner to choose its hypothesis from some simple class of functions. Freund and Schapire argue that (see Freund and Schapire 1997) this restriction/choice is specific to the concurrent learning problem and should reflect the knowledge about the properties of the unknown concept. Secondly to restrict the number of iterations T, and therefore the number of weak hypothesis h_t that are combined to form H. One method is to use an upper bound on the VC-dimension of the concept class.

There exist different types of VC-dimension bounds. One version, which is an improved boundary from the classic result of Vampnik and Chervonenkis[5], taken from Bartlett and Mendelson (see Bartlett and Mendelson 2002) is:

Theorem 2.3.2 *Let $\mathcal{F}$ be a class of $\{-1,+1\}$-valued functions defined on a set X, let P be a probability distribution on $X \times \{-1,+1\}$, and suppose that n samples $S = \{(x_i, y_i)\}_{i=1,\ldots,n}$ are chosen independently according to P. Then, there is an absolute constant c such that for any integer n, with probability at least $1-\delta$ over samples of length n, every function $f \in \mathcal{F}$ satisfies*

$$P(y \neq f(x)) \leq \hat{P}_n(y \neq f(x)) + c\sqrt{\frac{VCdim(\mathcal{F}) + log(1/\delta)}{n}}, \qquad (2.3.5)$$

[4]The VC-dimension (for Vapnik–Chervonenkis dimension) is a standard measure of the "complexity" of a space of functions that can be learned by a statistical classification algorithm. It was originally introduced 1971 by V. Vapnik and A. Chervonenkis in their collaboration: *"On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities"*.

[5]V. Vapnik and A. Y. Chervonenkis. *"On the uniform convergence of relative frequencies of events to their probabilities. Theory of Probability and its Applications"*, 16 (2): pages 264-280, 1971.

where

$$\hat{P}_n(S) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_S(x_i, y_i).$$

It is important to note, that this bound is distribution free, namely it holds independently of the underlying probability measure P.

Generalization Error as a function of Margin Distribution

Taking into account that a sample (x, y) was correctly classified if its margin was positive, Schapire et al. (see Schapire et al. 1998) argued that if that margin was large, then minor changes to the weights were unlikely to change the label and used the margins of the training sample to create a better generalization error bound. The margin of sample (x, y) is

$$margin_H(x, y) := \frac{y\sum_t \alpha_t h_t(x)}{\sum_t |\alpha_t|} \in [-1, +1]. \tag{2.3.6}$$

In their work, Schapire et al. (see Schapire et al. 1998) interpreted the magnitude of the margin as a measure of confidence in the classification and proved that larger margins on the training set translate into a better upper bound on the generalization error.

Theorem 2.3.3 *Given samples $S = \{(x_i, y_i)\}_{i=1,\dots,n}$ of size n chosen independently at random according to P. Assuming the base hypothesis space $\mathcal{H}$ is finite, and let $\delta > 0$. Then with probability greater or equal $1 - \delta$ over the random choice of the training set S and every weighted average function f of the convex hull of $\mathcal{H}$ the following bound (taken from Schapire et al. 1998) holds for all $\theta > 0$:*

$$P(yf(x) \leq 0) \;\leq\; \hat{P}_n(yf(x) \leq 0) + O\left(\frac{1}{\sqrt{n}}\left(\frac{ln(n)ln(|\mathcal{H}|)}{\theta^2} + ln(1/\delta)\right)^{1/2}\right).$$

For finite $\mathcal{H}$, with VC-dimension d the following, less complex, bound holds:

$$P(yf(x) \leq 0) \;\leq\; \hat{P}_n(yf(x) \leq 0) + O\left(\sqrt{\frac{d}{n\theta^2}}\right). \tag{2.3.7}$$

What is important to note, is that these bounds are entirely independent of the number of iterations T used in the boosting algorithm.

2.3.3 Exponential Loss Function

A primary benefit of exponential loss in additive modeling is computational as it leads to the simple modular re-weighting of the AdaBoost algorithm. AdaBoost minimizes the exponential loss,

$$\frac{1}{n}\sum_{i=1}^{n} e^{y_i F(x_i)} \tag{2.3.8}$$

where $F(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$ was already given in Equation (2.2.6). Hence, the choices of α_t and h_t would be chosen such, as to minimize Equation (2.3.8). This was first observed by Breiman (see Breiman 1999). It is therefore relevant to study the statistical properties of the exponential loss.

It seems clear that, using exponential loss as the loss function $L(\cdot)$ prioritises the choice of a function F for which the sign of $H(x_i) = sign(F(x_i))$ is likely to agree with the correct label y_i. In other words, it achieves the desired effect of trying to minimize the number of mis-classifications. Friedman et al. (see Friedman, Hastie, and Tibshirani 2000) showed that

$$\tilde{f}(x) = \arg\min_{f(x)} E_{Y|x} = \frac{1}{2} ln \frac{P(Y=1|x)}{P(Y=-1|x)}, \tag{2.3.9}$$

by trying to improve an estimate $F(x)$ with a new one $F(x) + f(x)$ by minimizing $E(e^{-yF(x)}e^{-yf(x)}|x)$ for every x. This can be rewritten as

$$E(e^{-yF(x)}e^{-yf(x)}|x) = e^{-f(x)} E\left(e^{-yF(x)} \mathbb{1}_{[y=1]}|x\right) + e^{f(x)} E\left(e^{-yF(x)} \mathbb{1}_{[y=-1]}|x\right).$$

Dividing this by $E(e^{-yF(x)}|x)$ and setting the derivative with respect to $f(x)$ to zero and solving for $f(x)$ yields the desired result in Equation (2.3.9), which is equivalent to

$$P(Y=1|x) = \frac{1}{1 + e^{-2\tilde{f}(x)}}. \tag{2.3.10}$$

With this it becomes clear, *what* the exponential loss is estimating: Namely one half of the log-odds of $P(Y=1|x)$. Which justifies the initial intuition of using its sign as the classification rule in step 8 of Algorithm 1.

A different possible loss function which could be used, that has the same population minimizer is the *negative binomial log-likelihood* (referred to as *deviance*).

Setting $\tilde{y} = \frac{y+1}{2} \in \{0, 1\}$ and parametrize the binomial probabilities as

$$p(x) = \frac{1}{1 + e^{-2f(x)}} = \frac{e^{f(x)}}{e^{f(x)} + e^{-f(x)}},$$

the binomial log-likelihood loss function and the *deviance* are then respectively given as

$$\begin{aligned} l(\tilde{y}, p(x)) &= \tilde{y} ln(p(x)) + (1-\tilde{y}) ln(1-p(x)) \quad and \\ -l(\tilde{y}, p(x)) &= ln(1 + e^{-2yF(x)}) \end{aligned} \tag{2.3.11}$$

Therefore, the population minimizers of the exponential loss $E(e^{-yF(X)})$ and of the deviance $-El(\tilde{y}, p(x))$ are the same, because the expected log-likelihood is maximized at the true probabilities $p(x) = P(\tilde{y} = 1|x)$, which defined the *logit* $F(x)$, which is excactly the same as the minimizer of the exponential loss due to Equation (2.3.10). Hence, it doesn't matter which criterion is chosen as they both lead to the same solution at the population level.

A palpable suggestion might arise, as to estimate H using the squared error $E(y - F(x))^2$. Friedman et al. (see Friedman, Hastie, and Tibshirani 2000) have found that non monotonicity of the squared error is inferior to monotone loss criteria. It yields correct classifications, but with $yF(x) > 1$ the loss increases for increasing values of $|F(x)|$. Hence, correct classifications that are *"too clear"* are penalized as much as the misclassification errors.

2.4 Weak Learners

An important property of boosting is the fact that all boosting algorithms have the **weak learning assumption** in common. This means, that a weak learning algorithm, used by boosting as a subroutine, can always do better than random guessing on the distributions presented to it. Without this condition, boosting would fail to work both in theory and in practice and it is therefore critical to ensure that the boosting algorithm is supplied by a "*good*" weak learner. It has already been shown that AdaBoost (Algorithm 1) works by forming a re-weighting w of the data at each step $t = 1 \ldots, T$ and constructs a base learner based on the weight distribution. Before properly defining a weak learner and its assumption, the definition of a *baseline* learner (as defined by Meir and Rätsch 2003) is given:

Definition 2.4.1 *Let $w = (w(1), \ldots, w(n))$ be a probability weighting of the data points $S = \{(x_i, y_i)\}_{i=1,\ldots,n}$ and let W_+ be the subset of the positively labeled points, and analogue for W_-. Set $W_+ = \sum_{i:y_i=+1} w(n)$ and similarly for $W_- = \sum_{i:y_i=-1} w(n)$. The* ***baseline classifier*** *f_{BL} is defined as $f_{BL}(x) := sign(W_+ - W_-)$ for all x. In other words, the baseline classifier predicts $+1$ if $W_+ \geq W_-$ and -1 otherwise. It becomes clear, that for any weighting w, the error of the baseline classifier is at most $1/2$.*

In contrast to the strong and weak PAC algorithms defined in section 2.1 (Definition 2.1.1 and 2.1.2 respectively) in this review, the definition of a weak learner is not as limited as the PAC definition for most applications. It is defined by Meir and Rätsch as follows (see Meir and Rätsch 2003):

Definition 2.4.2 *A learner is a* ***weak learner*** *for sample S if, given any weighting w on S, it is able to achieve a weighted classification error (see (2.2.5)) of $\epsilon_t(h_t, w_t) < 1/2 - \eta$, where $0 < \eta < 1/2$ is a fixed value called the edge.*

Therefore, a *weak learner* achieves an accuracy which is slightly but strictly better than that of random guessing of a baseline classifier. This property is what is meant by the *weak learner assumption.* The edge value η in Definition 2.4.2 quantifies this difference of the performance of the weak learner from the baseline classifier.

In his earlier work *"The Strength of Weak Learnability"* (see Schapire 1990) Schapire proved that any weak learning algorithm can be efficiently converted, termed *"boosted"*, into a strong one (see Definition 2.1.1). The main result of his paper is:

Theorem 2.4.1 *A problem is weakly learnable if and only if it is strongly learnable.*

The first statement of this equivalence is trivial, as a strong learner is automatically a weak learner for the same problem. However, to prove the reverse of the statement is more difficult. Schapire's approach was to use an algorithm for transforming a weak learner into a strong one. Starting off with a class of weak learners, a mechanism is used to boost the accuracy by a small amount and this mechanism is then applied recursively to make the error arbitrarily small (for the full proof refer to Schapire's work (Schapire 1990)).

A prerequisite for a boosting algorithm to work well, is the existence of a weak learner. In the case of binary classification, as in Algorithm 1, the condition is that the weighted empirical error of every weak learner is strictly smaller than $\epsilon_t(h_t, w_t) < \frac{1}{2} - \eta$, where η is an *edge* parameter first introduced in Definition 2.4.1. A weak learner constructs a binary classifier H based on a data set, $S = \{(x_i, y_i)\}_{i=1,\dots,n}$, where each pair (x_n, y_n) is weighted by a non-negative weight $w(i)$. Then the following must hold,

$$\epsilon_t(h, w) = \sum_{i:h(x_i)\neq y_i} w(i) \leq \frac{1}{2} - \eta \ , \quad (\eta > 0). \tag{2.4.1}$$

Without making restrictions about the data set S, it may be impossible to find a strictly positive value of $\eta > 0$ for which the Equation (2.4.1) is true, given a simple weak learner. To illustrate this, Meir and Rätsch (see Meir and Rätsch 2003), considered the following example:

A two-dimensional problem with $n = 4$ observations, $x_1 = (-1, -1)$, $x_2 = (+1, +1)$, $x_3 = (-1, +1)$, $x_4 = (+1, -1)$, and corresponding labels $\{-1, -1, +1, +1\}$. Now if the weak h is restricted to be an axis-parallel half-space, then it is obvious that no such h can achieve an error smaller than 1/2 with a uniform weighting over the examples.

Yoav Freund shows, in his work (see Freund 1995), that if a class $\mathcal{H}$ of binary hypothesis is highly correlated with the target function f, then f can be exactly represented as a convex combination of a small number of functions from $\mathcal{H}$. In other words this means, that the empirical error can be expected to approach zero, after a sufficiently large number of boosting iterations. Let f be denote a Boolean function, from the binary p-dimensional cube $\{-1, +1\}^p$ to $\{-1, +1\}$ and $\mathcal{H}$ be a set of binary hypotheses h_t to approximate f. $\mathcal{D}$ shall be the distribution over the cube $\{-1, +1\}^p$, and defining the correlation between f and $\mathcal{H}$ with respect to $\mathcal{D}$ as

$$Corr_{\mathcal{H}}^{\mathcal{D}}(f) = \sup_{h\in\mathcal{H}} E_{\mathcal{D}}(f(x)h(x)),$$

and the distribution free correlation as

$$Corr_{\mathcal{H}}(f) = \inf_{h\in\mathcal{H}} Corr_{\mathcal{H}}^{\mathcal{D}}(f).$$

Using this notation, Yoav Freund then stated the following theorem

Theorem 2.4.2 *Let f be a Boolean function over $\{-1, +1\}^p$ and $\mathcal{H}$ be a set of functions over the same domain. Then if the number of iterations T is*

$$T > 2ln(2)nCorr_{\mathcal{H}}(f)^{-2}$$

then f can be represented as

$$f(x) = sign\left(\sum_{t=1}^{T} h_t(x)\right).$$

2.4.1 Choice of Weak Learner

As the concept of weak learners is an essential part of boosting, it is important for every algorithm to specify a weak learner. The decision of what weak learner to choose is mostly driven by the consideration of the structural properties of the boosting estimate H or by optimizing the predictive capacity. Even though weak learners/base procedures are central in boosting, the emphasis of the theory of choosing an adequate weak learner, is considerably less compared to the magnitude of work put in surrounding boosting's other facets. In most practical experiments the choice was mostly based on the evaluation of the error rate and not based on the mathematical properties of the weak learner. The first analysis concerning the choice of a weak learner was done by Lev Reyzin in his paper (see Reyzin 2014).

If one considers the workings of the weak learning algorithm as doing empirical risk minimization, over some hypothesis class $\mathcal{H}$, then the algorithm depending on a labelled sample $(\mathbf{x}, \mathbf{y})$ returns an hypothesis $h \in \mathcal{H}$ such that

$$h = \arg\min_{h \in \mathcal{H}} P_x[h(x) \neq y].$$

A more lenient criterion would be to find an approximate empirical risk minimization hypothesis. With this in mind, Lev Reyzin in his paper (see Reyzin 2014) argued that the following properties are important for an effective weak learner:

Coverage: All parts of the hypothesis space should be "covered" by a weak learner. Thus, no areas that are sparsely covered should exist. With high coverage, diversity is easier to achieve.

Diversity: A weak learner should have many hypotheses that disagree on a large fraction of examples in their predictions. A reason for this is AdaBoost's update rule which makes the distriubtion w_{t+1} be such that

$$P_{(x,y)\sim w_{t+1}}[h_t(x) = y] = P_{(x,y)\sim w_{t+1}}[h_t(x) \neq y] = 1/2.$$

Therefore, hypotheses highly correlated with h_t will likely have a small edge as well, making the weak learning condition harder to satisfy.

Error: As mentioned before, the weak learner assumption must hold.

Evaluability: The weak learner should be efficiently evaluable for it to be usable in practice.

Richness: It is advantageous if a linear combination of hypotheses from $\mathcal{H}$ is able to represent a rich set of functions.

Optimizability: Finding an approximate empirical risk minimization over the weak learners should be easy to control.

Simplicity: For a finite H, its cardinality $\#H$ should be as small as possible, while an for infinite one it's VC dimension (or similar measure) should be small.

It is to note that some of the conditions reinforce each other, while others are in contest with one another. For example, a way to achieve diversity is to ensure coverage, whereas diversity is in direct conflict with simplicity. Therefore, a good weak learner must balance out among these. For instance, the set of all Boolean functions, while achieving great diversity, coverage, richness, and efficiency fails the error and simplicity requirements, making for an ineffective weak learner. As the final boosting estimator is a sum of base procedure estimates (see for example Algorithm 1 step 8), the structural properties of the boosting function estimator are induced by a linear combination of structural characteristics of the base procedure.

Some important examples of weak learners/base procedures yielding desirable structures for the boosting estimator are presented:

Decision stumps

A natural class for weak learners to look at are *decision stumps*. They are simple weak learners that examine only one (binary) feature value. They are commonly used as base classifiers in boosting and all have the same complexity by most any measure. Decision stumps can be viewed as single node decision trees. Meaning, it is a decision tree with one internal node (the root) which is immediately connected to the terminal nodes. Therefore, the connotation of a decision "stump" as it only contains the root and no further branches or leaves of a whole *"tree"*. A decision stump makes a prediction based on the value of just a single input feature and thus examines only one binary value and predicts either its occurrence or its absence.

Due to the vote in AdaBoost, and other ensemble methods, the sum over the iterations T can be rewritten as a sum over the weak learners in finite space

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right) = sign\left(\sum_{i} \hat{\alpha}_i h_i(x)\right)$$

with appropriate coefficients $\hat{\alpha}_i > 0$. Since this is a linear function when stumps are used and many functions cannot be predicted with sufficient accuracy via a linear combination, decision stumps can be rather limited in their usage.

Despite the simplicity of decision stumps and their afore mentioned downside, they have proven useful in practice, early on in the original work of Freund and Schapire (1995) and probably most notably in the state-of-the-art Viola–Jones face detection algorithm, which employs AdaBoost with decision stumps as weak learners (see Viola and Jones 2003).

Trees

As a follow up of simple stumps are *classification and regression trees*, in short **CART**. The term CART was first introduced by Breiman et al. (see Breiman et al. 1984). CART is a non-parametric tree learning technique that produces regression or classification trees, depending on whether the outcome variable is numeric or categorical. As compared to stumps they present a more complex form of weak learner and are perhaps the most popular general weak learner.

Although the usage of trees as weak learners in boosting has proven to be competitive, generally outperforming stumps, for example shown by Caruana and Niculescu-Mizil in 2006 (see Caruana and Niculescu-Mizil 2006), they are also prone to overfitting. To avoid over complex trees and as a consequence of over-fitting, the decision trees are needed to be pruned by removing nodes, thus reducing their size and their complexity. Another drawback of trees is their non-robustness in some cases, wherein small changes in the training data can result in large changes in the final predictions.

Sparse parities

Due to the afore mentioned drawbacks of trees, Lev Reyzin (see Reyzin 2014) proposed using *sparse parity functions*, as weak learners in boosting.

Considering that any Boolean function $f : \{0,1\}^n \to \{-1,1\}^n$ using Fourier analysis can be uniquely written as

$$f(x) = \sum_{S \in \{0,1\}^n} \hat{f}_S \chi_S(x) \tag{2.4.2}$$

with the *characters*

$$\chi_S(x) = (-1)^{S \cdot x},$$

over the Galois field consisting of two elements. This function can simply be seen as the changing with the parity of $S \cdot x$.
If all 2^n of the *Fourier coefficients* $\hat{f}_S$, of the corresponding characters of f, are specified, then f can be reconstructed. The *degree* of $\hat{f}_S$ is equal to $||S||_1$.

Reyzin's proposed technique can approximately recover targets whose *Fourier-weight* is concentrated on the low degree characters. These functions f therefore satisfy

$$W^{\leq d}(f) := \sum_{S \in \{0,1\}^n \,:\, ||S|| \leq d} \hat{f}_S^2 \geq 1 - \epsilon_0,$$

for a suitable choice of ϵ_0 as a function of the target error rate.

Reyzin noticed the similiarities between Equation (2.4.2) and the final hypothesis H of boosting

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right).$$

Now setting the class of weak learners $\mathcal{H}_d$ to be the characters of degree $\leq d$ (d-parities), including their negations, one gets a correspondence between the α's and the $\hat{f}$'s, as well as between the h's and the χ_S. Therefore, boosting will try to find a low degree Fourier approximation of the target function, if it exists.

In Reyzin 2014 it is shown that the sparse parities, as a weak learner, meet the diversity, coverage, simplicity, efficiency, error and richness properties. To satisfy the optimizability criterion, all the $\binom{n}{d}$ elements need to be tried. An extension to the parity function to the multi-class case, with k classes, is using the function $x \cdot c(mod\ k)$. Then x will take integer values, splitting the numerical values of x into k classes.

Componentwise linear least squares for generalized linear models

Another weak learner is mentioned by Bühlmann and van de Geer (2011). When trying to fit a high dimensional generalized linear model (dimension p and $x^{(j)}$ denotes the column j of x and $x_i^{(j)}$ the i'th element of $x^{(j)}$)

$$g(E[Y_i|x]) = \sum_{j=1}^{p} \beta_j x^{(j)} \quad , \quad with\ Y_1, \ldots, Y_n\ independent,$$

boosting can be useful when using the weak learner

$$\hat{g}(x) = \hat{y}_{\hat{j}} x^{(\hat{j})},$$

with

$$\hat{y}_j = \frac{\sum_{i=1}^n x_i^{(j)} \tilde{y}_i}{\sum_{i=1}^n (x_i^{(j)})^2} \ , \ \hat{j} = \underset{j=1,\ldots,p}{\arg\min} \left(\tilde{y} - \hat{y}_j x_i^{(j)} \right) = \underset{j=1,\ldots,p}{\arg\max} \frac{|\sum_{i=1}^n x_i^{(j)} \tilde{y}_i|^2}{\sum_{i=1}^n (x_i^{(j)})^2}$$

and the current negative gradient vector $\tilde{y}$ of the loss. In the case of centred variables $\bar{x} = 0$, the $\hat{j}$ is choosen that maximizes the absolute correlation with the residual vector. The weak learner selects the best variable according to ordinary least squares (refer to Section 1.1.1).

Componentwise smoothing splines for additive models

As a non-parametric weak learner for function estimation, least squares *cubic smoothing spline* estimates can be used (see Bühlmann and Geer 2011). The cubic smoothing spline estimate $\hat{g}$ of the function g is the minimizer of

$$\hat{g}_j(\cdot) = \underset{f}{\arg\min} \left(\sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \int (f'')^2 dx \right),$$

where $\lambda \geq 0$ is a smoothing parameter, which is often estimated by cross validation and f'' is the second derivative of f. The weak learner is

$$\hat{g}(x) = \hat{g}_j(x^{(\hat{j})}) \quad , \quad with \ \ \hat{j} = \underset{j=1,\ldots,p}{\arg\min} \sum_{i=1}^{n} \left(\tilde{y}_i - \hat{g}_j(x_i^{(j)}) \right).$$

2.5 Regularization

As has already been mentioned, a common issue when minimizing a loss function is overfitting the model on the training set. To avoid this problem, it has become standard to *regularize* the problem. In other words, to constrain or modify the size of the function class, thereby reducing complexity and emphasizing smoothness.

As was already seen in Section 2.2, the AdaBoost algorithm's final hypothesis is constructed by a linear combination F of weak hypotheses

$$F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

This $F(x)$ is chosen over all such linear combinations, as to minimize the exponential loss. Schapire proposed (see Schapire 2013), that one way to regularize is to choose the objective to be the minimization of this same loss, but subject it to the constraint that the weak hypotheses weights appearing in F, when viewed collectively as a vector, have l_1-norm bounded by some pre-set parameter $B > 0$. As opposed to the l_1-norm, which is sometimes called the *"lasso"*[6], a different norm could be used as well.
This constraint to those weights is referred to as *shrinkage*. For example, while using a boosting algorithm, with trees as weak learners, the contribution of each tree is scaled by a factor $v \in (0, 1)$. Friedman (see Friedman 2001) empirically showed that smaller values of v produce better test errors, but require a larger number of iterations T respectively.

AdaBoost doesn't explicitly regularize, as there are no limitations to the weights on the weak hypothesis. However, Schapire (see Schapire 2013) argues that the AdaBoost has an implicit form of regularization, as he shows how a simple variant of the algorithm, when stopped after any number of rounds t, can often be viewed as providing an approximate solution to l_1-regularized minimization of exponential loss. Thus, overfitting is avoided implicitly.
The two downsides of this are, that firstly this only applies to a variant of Adaboost in which the weights are set to a constant and secondly that AdaBoost is stopped after a relatively small number of rounds.

To summarize, a few ways to achieve regularization are: The modification of the function class, the limitation of boosting iterations T (as it can be related to a l_1-regularization) or by shrinkage.

2.6 Boosting Interpretations

2.6.1 A Bayesian Interpretation

In one of their earlier papers Freud and Schapire (see Freund and Schapire 1997) showed that the final hypothesis generated by AdaBoost is similar to one suggested by Bayesian analysis.

In the binary setting of AdaBoost, given some examples (x, y) that have been generated by the distribution on $X \times \{-1, 1\}$, and a set of $\{-1, 1\}$-valued hypotheses

[6]Robert Tibshirani introduced in 1996 the *lasso* (short for *least absolute shrinkage and selection operator*) in his work *"Regression Shrinkage and Selection via the lasso"*. It is a regression method that performs both variable selection and regularization.

$h_1, \ldots, h_T$. The objective is to combine these hypotheses in an optimal way to obtain the final hypothesis H.

After having achieved this and then given a new observation x and the hypothesis prediction $h_t(x)$, the Bayes optimal decision rule says that the label, which has the highest probability/likelihood should be chosen, given the hypothesis values. Therefore, according to the Bayes optimal decision rule it should predict 1 if

$$P[y = 1|h_1(x), \ldots, h_T(x)] > P[y = -1|h_1(x), \ldots, h_T(x)], \qquad (2.6.1)$$

and -1 otherwise.

If the event $h_t(x) \neq y$ is conditionally independent of the actual true label y and of the predictions of all the other hypotheses $h_1(x), \ldots, h_{t-1}(x), h_{t+1}(x), \ldots, h_T(x)$, then the errors of the different hypotheses are independent of one another and of the target concept. Under those conditions, by applying Bayes rule, the Bayes optimal decision rule (Equation 2.6.1) will be relatively easy to compute and can be expressed in the simple form in which 1 is predicted if

$$P[y = 1] \prod_{t:h_t(x)=-1} \epsilon_t \prod_{t:h_t(x)=1} (1 - \epsilon_t) > P[y = -1] \prod_{t:h_t(x)=-1} (1 - \epsilon_t) \prod_{t:h_t(x)=1} \epsilon_t$$

and -1 otherwise. Here, the error term ϵ_t refers to the probability $\epsilon_t = P[h_t(x) \neq y]$. Adding to the set of hypotheses the trivial hypothesis h_0 which always predicts the value 1, the probability $P[y = 0]$ can be replaced by ϵ_0. Then taking the logarithm of both sides of this inequality and rearranging the terms, the Bayes optimal decision rule is revealed to be identical to the combination rule that is generated by AdaBoost. In the opposite case, if the errors of the different hypotheses are dependent, then the Bayes optimal decision rule becomes a lot more complex. Even though the prerequisites of independence are not met, the above mentioned simple rule is used. In statistics and machine learning this is referred to as the *"naive Bayes"* (or simple Bayes). Freud and Schapire offered an alternative to the practice of *"naive Bayes"*, as to use AdaBoost to find a combination rule which has a guaranteed non-trivial accuracy due to Theorem 2.3.1.

2.6.2 A Game-theoretical Interpretation

The behavior of AdaBoost, or boosting in general, can also be understood in a game-theoretic setting. Freund and Schapire presented in their paper (see Freund and Schapire 1996b) the close connection between boosting and *game theory*[7]).

A two person game in normal form, is defined by a matrix $\mathbf{M}$. The first player, called row-player, chooses a row i and the other player, the column-player, simultaneously chooses a column j. The result or outcome of the game is given in the selected entry M_{ij}, which describes the *loss* of the row-player. The goal of the row-player is to minimize the loss, whereas the column-player tries to maximize it, which

[7]Roger Myerson defined game theory as: *"the study of mathematical models of conflict and cooperation between intelligent rational decision-makers"* (Game Theory: Analysis of Conflict, Harvard University Press, 1991

is called a *"zero-sum"* game. An easy to understand and well known example would be "Rock, Papers, Scissors" where the loss matrix is

$$\mathbf{M} = \begin{array}{c} \\ R \\ P \\ S \end{array} \!\!\begin{array}{c} \begin{array}{ccc} R & P & S \end{array} \\ \begin{pmatrix} \frac{1}{2} & 1 & 0 \\ 0 & \frac{1}{2} & 1 \\ 1 & 0 & \frac{1}{2} \end{pmatrix} \end{array}.$$

The two players are also allowed to play randomly. That is, the row-player chooses a distribution $\mathbf{P}$ and (simultaneously) the column-player chooses another distribution $\mathbf{Q}$, over the loss matrix $\mathbf{M}$. Hence, the row-player's expected loss is

$$\mathbf{M(P,Q)} \;\; = \mathbf{P}^T\mathbf{MQ} = \sum_{i,j} \mathbf{P}_i M_{ij} \mathbf{Q}_j,$$

where $\mathbf{P}_i$ is the row-players probability of choosing of row i and $\mathbf{Q}_j$ that of the column-player's of choosing column j. With sequential play, where the column-player can make his choice after the row-player has made his, the column-player will want to maximize the loss by his choice of $\mathbf{Q}$ as $\max_{\mathbf{Q}} \mathbf{M}(\mathbf{P},\mathbf{Q})$. Since the row-player knows this, he will choose $\mathbf{P}$ accordingly to minimize the loss $\min_{\mathbf{P}} \max_{\mathbf{Q}} \mathbf{M}(\mathbf{P},\mathbf{Q})$. A strategy $\mathbf{P}^*$ which realizes this is called a *minmax*-strategy. Analogues if the column-player where to choose first, one'd arrive at a *maxmin*-strategy $\mathbf{Q}^*$ which satisfies $\max_{\mathbf{Q}} \min_{\mathbf{P}} \mathbf{M}(\mathbf{P},\mathbf{Q})$.
Although one would assume that the player who chooses last is at a disadvantage, Von Neumann proved in his *minmax Theorem*[8] that this is not the case, as

$$\min_{\mathbf{P}} \max_{\mathbf{Q}} \mathbf{M}(\mathbf{P},\mathbf{Q}) = \max_{\mathbf{Q}} \min_{\mathbf{P}} \mathbf{M}(\mathbf{P},\mathbf{Q}) = v, \tag{2.6.2}$$

where v is called the *value* of the game $\mathbf{M}$.

Boosting can be viewed as repeated play of $t = 1, \dots, T$ rounds of a particular game matrix. To apply the game-theoretical thinking to boosting, the row-player will be called a *learner*, representing the boosting algorithm, and the column-player the *environment*, representing the weak learner. Thus, the distribution $\mathbf{P}$ is the boosting algorithm's choice of a distribution D_t over training examples, while the choice of a column j ($\mathbf{Q}$) becomes the weak learner's choice of hypothesis h_t. Assuming a finite set of m binary weak hypotheses $h_1, \dots, h_m$ and a fixed training set $(x_i, y_i)_{i=1,\dots,n}$ the matrix $\mathbf{M}$ has n rows and m columns where

$$M_{ij} = \begin{cases} 1 & \text{if } h_j(x_i) = y_i \\ 0 & else \end{cases}.$$

Applying this matrix $\mathbf{M}$ to Von Neumanns theorem (2.6.2) and reinterpreting it in the boosting setting (see Freund and Schapire 1996b and Schapire 1999) lead to the following. If there is a weak hypothesis with error at most $1/2 - \gamma$, for any distribution over the examples, then there exists a convex combination of weak hypotheses with a margin of at least 2γ on all training examples. AdaBoost seeks to find such a

[8]John von Neumann *"Zur Theorie der Gesellschaftsspiele"*, Math. Annalen. 100, 1928, pages 295–320

hypothesis by combining many weak hypotheses; so in a sense, the minmax theorem tells us that AdaBoost at least has the potential for success since, given a “good” weak learner, there must exist a good combination of weak hypotheses. Going further, AdaBoost can be shown to be a special case of a more general algorithm for playing repeated games, or for approximately solving matrix games. This shows that, asymptotically, the distribution over training examples as well as the weights over weak hypotheses in the final hypothesis have game-theoretic intepretations as approximate minmax or maxmin strategies.

2.7 Gradient Boosting

In this section it will be shown that the AdaBoost algorithm can be represented as a steepest decent algorithm in a function space, which can be called *functional gradient decent* (short: FDG). This was first shown by Breiman (see Breiman 1999) and Friedman (see Friedman 2001) continued this by developing a more general, statistical framework, which shows that boosting can be interpreted as a method for estimating functions.

2.7.1 General Case

Continuing the notation where (x, y) denotes an example from $X \times Y$, where X is the space of measurements and Y denotes the space of classes/labels. F shall be a function $F : X \to Y$ mapping X to Y and $\langle \cdot, \cdot \rangle$ be an inner product on the set of all linear combinations of functions like F. The goal is to find a function F, which minimizes the function

$$\Phi(F) = E\left(L(F(x), y)\right) \tag{2.7.1}$$

where L is a chosen loss function ($\Phi(\cdot)$ is also referred to as *cost functional*, see Mason et al. 1999).

This minimization process can be done iteratively via a gradient decent procedure. Starting with a given function F_1, the objective is to find a new f to add to F_0 such that the cost $\Phi(F_1 + cf)$ is less than that of F_0, where c is some small constant. Hence, the incentive is to find the f, the “direction” such that $\Phi(F_0 + pf)$ decreases the fastest. The desired direction f will be obtained via *steepest-decent*, where to find a local minimum of a function one takes steps proportional to the negative of the gradient of the function at the current point. Let the m'th step be composed as $f_m = -c_m g_m(x)$, with the multipliers c_m and the gradient $g_m(x)$ is

$$g_m(x) = \nabla\Phi(F(x)) = \left[\frac{\partial\Phi(F(x))}{\partial F(x)}\right]_{F(x)=F_{m-1}}, \tag{2.7.2}$$

with

$$F_{m-1}(x) = \sum_{i=1}^{m-1} f_i(x).$$

The multipliers c_m of $f_m = -c_m g_m(x)$ can be calculated by a simple line search

$$c_m = \arg\min_c E_{y,x} L(y, F_{m-1}(x) - c g_m(x)). \tag{2.7.3}$$

Since the choice of a new function f is restricted in some cases, in general it can not be chosen as the negative gradient as mentioned above. Instead a function f will be chosen with the greatest inner product with $-\nabla\Phi(F(x))$ (here for sake of simplicity the multiplier c_m was omitted). Thus, a function f which maximizes $-\langle\nabla\Phi(F), f\rangle$ will be chosen to substitute as the negative gradient. As a motivation for this procedure, consider after an optimizing a c, one obtains $\Phi(F+cf) = \Phi(F) + c\langle\nabla\Phi(F), f\rangle$ and thus, the greatest reduction in cost will occur for the f which maximizes $-\langle\nabla\Phi(F), f\rangle$.

2.7.2 AdaBoost as Steepest Decent Algorithm

Restricting the functions f, the base hypotheses, to mappings to $Y = \{-1, +1\}$ and the inner product to

$$\langle F, G\rangle := \frac{1}{n}\sum_{i=1}^{n} F(x_i)G(x_i)$$

for two functions F and G and a training sample $(x_i, y_i)_{i=1,\dots,n}$ generated by some unknown distribution P on $X \times Y$. The unchanged goal is to find a function $\hat{F}$ such that the misclassification probability

$$P_{x,y}[sgn(F(x)) \neq y]$$

is minimal.
For this section, let the **margin** of F on an example (x, y) be defined as $\mathbf{yF(x)}$ (as opposed to Equation (2.3.6)).
Considering *margin cost-functionals* (such as exponential loss for AdaBoost, see Section 2.3.3) defined by

$$\Phi(F) := \frac{1}{n}\sum_{i=1}^{n} L(y_i F(x_i))$$

where $L : \mathbb{R} \to \mathbb{R}$ is a differentiable real-valued function of the margin. Using these definitions, the negative inner product of the gradient and f is calculated as

$$-\langle\nabla\Phi(F), f\rangle = -\frac{1}{n^2}\sum_{i=1}^{n} y_i f(x_i) L'(y_i F(x_i)).$$

Since positive margins correspond to examples being correctly labelled by $sgnF$ and negative margins to be incorrectly labelled examples, any sensible cost function of the margin will be monotonically decreasing. It follows that therefore $-L'(y_i F(x_i))$ will always be positive and dividing it by $-\sum_{i=1}^{n} L'(y_i F(x_i))$, it becomes clear that finding an f maximizing $-\langle\nabla\Phi(F), f\rangle$ is equivalent to finding an f which minimizes the weighted error term

$$\sum_{i: f(x_i)\neq y_i} w_t(i) \quad where \quad w_t(i) := \frac{L'(y_i F(x_i))}{\sum_{i=1}^{n} L'(y_i F(x_i))} \quad for\ i = 1, \dots, n.$$

This is the same/similar procedure used in AdaBoost, where the sum of the training error is minimized, see (2.2.1).

2.7.3 The generic FGD Algorithm

In the general case of finite data, wherein the non-parametric approach expectations such as $E_y[\cdot|x]$ cannot be estimated accurately by its data value at each x_i, and even if it could, one would like to estimate $F(x)$ at x values other than the training sample points.
This issue can be overcome by using nearby data points to impose smoothness on the solution. One way to do this is to assume a parametrized form such as

$$F(x; \{\beta_m, a_m\}) = \sum_{m=1}^{M} b_m h(x; a_m)$$

where the (generic) function $h(x; a)$ is usually a simple parameterized function of the input variables x, characterized by parameters $a = (a_1, a_2, \ldots)$. The individual terms differ in the joint values a_m chosen for these parameters. Then a parameter optimization is done to minimize the corresponding data based estimate of expected loss,

$$\{\beta_m, a_m\}_{m=1}^{M} = \underset{\{\beta'_m, a'_m\}_1^M}{\arg\min} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta'_m h(x_i; a'_m)\right).$$

In situations where this is applicable one can try a "greedy stagewise" approach. For $m = 1, \ldots, M$ the parameters are obtained as

$$(\beta_m, a_m) = \underset{\beta, a}{\arg\min} \sum_{i=1}^{N} L\big(y_i, F_{m-1}(x_i) + \beta h(x_i; a)\big) \tag{2.7.4}$$

and at the m'th step the function will be

$$F_m(x) = F_{m-1}(x) + \beta_m h(x; a_m). \tag{2.7.5}$$

Assuming now, that for a given loss function $L(y, F)$ and/or a particular base learner $h(x; a)$ the solution to (2.7.4) is difficult to obtain. For any approximation $F_{m-1}(x)$, the function $\beta_m(x; a_m)$ in (2.7.4) and (2.7.5) can be viewed as the best greedy step toward the data-based estimate of

$$F^* = \underset{F}{\arg\min}\, E_{y,x} L(y, F(x)) \tag{2.7.6}$$

under the condition that the step *"direction"* $h(x; a_m)$ be a member of the parameterized class of functions $h(x; a)$. Therefore, it can be regarded as a steepest descent-decent as in (2.7.2). Substituting the loss function $L(y_i, F(x_{xi}))$ into (2.7.2) and assuming sufficient regularity such that the differentiation and the integration become interchangeable one obtains

$$-g_m(x_i) = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}.$$

Although this gives the best steepest-decent step direction in the n-dimensional data space at $F_{m-1}(x)$, this gradient is only defined at the n training data points and cannot be generalized to other x values.
A way to overcome this problem, is to choose a member of the parameterized class $h(x; a_m)$ that produces $h_m = \{h(x_i, a_m)\}_{i=1}^{n}$ which is most parallel to $-g_m$, i.e. the

$h(x, a)$ most highly correlated with $-g_m$ over the data distribution. Specifically it can be obtained from the solution of

$$a_m = \arg\min_{a,\beta} \sum_{i=1}^{n} [\tilde{y}_i - \beta h(x_i; a)]^2 \tag{2.7.7}$$

where the constrained negative gradient $h(x, a_m)$ is used instead of the unconstrained on $-g_m$. The line search (2.7.3) is then preformed as

$$p_m = \arg\min_{p} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + ph(x_i; a_m) \tag{2.7.8}$$

and finally the approximation is updated

$$F_M(x) = F_{m-1}(x) + p_m h(x; a_m).$$

In essence, the idea is instead to obtain a solution under a smoothness constraint (2.7.4), the constraint is applied to the unconstrained solution first by fitting $h(x, a)$ to the *"pseudo-responses"* $\{\tilde{y}_i = -g_m(x_i)\}_{i=1}^{n}$. This allows the replacement of the difficult function minimization problem (2.7.4) by a least-squares function minimization (2.7.7) and only a single parameter optimization based on the original criterion (2.7.8). Thus, for any $h(x; a)$ for which a feasible least-squares algorithm exists for solving (2.7.7), one can use this approach to minimize any differentiable loss $L(y, F)$ in conjunction with forward stage-wise additive modeling.

Algorithm 2 Generic FGD Algorithm

```
1: Initialize $\hat{F}_0(x)$ with an offset value: $\hat{F}_0(x) = \arg\min_p \sum_{i=1}^{n} L(y_i, p)$ or $\hat{F}_0(x) \equiv 0$
2: for $m = 1$ to $M$ do
3:     Compute the negative gradient and evaluate at $\hat{F}_{m-1}(x)$:
           $\tilde{y}_i = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=\hat{F}_{m-1}(x)}, \ i = 1, \ldots, n$
4:     Calculate the parameters
           $a_m = \arg\min_{a,\beta} \sum_{i=1}^{n} [\tilde{y}_i - \beta h(x_i; a)]^2$
       and
           $p_m = \arg\min_{p} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + ph(x_i; a_m))$
5:     Update the function
           $F_m(x) = F_{m-1}(x) + p_m h(x; a_m)$
6: end for
7: Output
           $F_M$
```

Using the notation above, Friedman (see Friedman 2001) has given the following Algorithm 2 which will be referred to as the *FGD* (functional gradient decent)-algorithm.

The number of iterations m, which is a main tuning regularization parameter, can be determined via cross-validation. Depending on the choice of loss functions in the generic FDG-Algorithm 2, various boosting algorithms can be defined (refer to section 3.3.2).

Chapter 3

Algorithms and Expansions

The focus of this chapter is the introduction of various boosting algorithms, some of which will be used in the simulations of Chapter 4.
It will start with presenting alternative algorithms to the AdaBoost in a binary setting, thereby introducing the Real- and Gentle-AdaBoost, as well as the Logit Boost. After that, the expansion to the multi-class setting is given. This will introduce multiclass algorithms such as AdaBoost.M1, SAMME and Logit - k class. Lastly, the final expansion to the regression problem will be covered as well.

3.1 Binary Boosting

The introduction of the AdaBoost algorithm by Freund and Schapire, sparked a significant amount of further research on this type of learning technique (boosting), since the theoretical properties and practical performances of it were promising. Thus, a lot of different variants arose, which follow the basic structure of AdaBoost, but change several aspects, such as the weight updates or the choice of loss function.

In this section some variants besides AdaBoost (Algorithm 1) for binary classification will be shown and described in detail. In the binary setting, the possible output space will be a set of only two possible outcomes: $Y = \{-1, +1\}$. This could possibly translate to output variables such as $Y = \{\textit{“no”}, \textit{“yes”}\}$ or $Y = \{\textit{“positive”}, \textit{“negative”}\}$, for example.

3.1.1 Real AdaBoost - RealBoost

Starting off with the *Real AdaBoost* algorithm, which was formulated by Friedman, Hastie, and Tibshirani (see Friedman, Hastie, and Tibshirani 2000). Out of simplicity and to avoid confusion with the original AdaBoost, the Real AdaBoost will be referred to as **RealBoost** in this thesis. The RealBoost alorithm, taken from Friedman, Hastie, and Tibshirani 2000, is presented as Algorithm 3.

The name *“real”* refers to the fact that in RealBoost, the class probability estimate is converted, using half the log ratio, to a real valued scale (as seen in step 4 of Algorithm 3). This value or scale can be seen as the probability, that a given input y belongs to a class, considering the current weight distribution w for the training

Algorithm 3 Real AdaBoost

```
1: Initialize weights $w_i = 1/n$ for $i = 1, \dots, n$
2: for $t = 1$ to $T$ do
3:     Using weights $w_t$, fit the classifier to obtain a class probability estimate $p_t(x) = \hat{P}_w(y = 1) \in [0, 1]$
4:     Set $f_t(x)$ as
           $$f_t(x) = \frac{1}{2} log\left(\frac{p_t}{1 - p_t}\right) \in \mathbb{R}$$
5:     Update the weights as
           $$w_{t+1} = \frac{w_t e^{-yf_t(x)}}{Z_t}$$
       where $Z_t$ is a normalization factor, chosen such that $w_t$ will be a distribution $\sum_i^n w_{t+1}(i) = 1$.
6: end for
7: Output the final classifier as
           $$sign(F(x)) = sign(\sum_{t=1}^{T} f_t(x)).$$
```

set. This value $p_t(x)$ is then used to represent an observation's contribution to the final overall model.
Just like in AdaBoost, observation weights for subsequent iterations are updated in step 5 according to the exponential loss function of AdaBoost, thus trying to minimize the expectation of $e^{-yF(x)}$.

The major difference between AdaBoost (Algorithm 1) and RealBoost (Algorithm 3), are in the steps 3 and 4 of Algorithm 3. While standard AdaBoost classifies the input patterns and computes the weighted error rate, RealBoost uses weighted probability estimates to update the additive logistic model, rather than the classifications themselves.

Given a current estimate $F(x)$ and the objective is to obtain an improved estimate $F(x) + f(x)$ through minimization of $J(F(x) + f(x))$ at each x, with $J(F) = E[e^{-yF(x)}]$, the following can be done:

$$\begin{aligned} J(F(x) + f(x)) &= E[e^{-yF(x)} e^{-yf(x)} | x] \\ &= e^{-f(x)} E[e^{-yF(x)} I_{[y=1]} | x] E[e^{-yF(x)} I_{[y=-1]} | x] \end{aligned}$$

Dividing this result by $E[e^{-yF(x)}|x]$, then calculating the derivative with respect to $f(x)$ and setting it to zero, one obtains:

$$f(x) = \frac{1}{2} log\left(\frac{E_w[\mathbb{1}_{[y=1]}|x]}{E_w[\mathbb{1}_{[y=-1]}|x]}\right) = \frac{1}{2} log\left(\frac{P_w(y = 1|x)}{P_w(y = -1|x)}\right),$$

with weights $w(x, y) = e^{-yF(x)}$, which are updated as $w(x, y) \leftarrow w(x, y)e^{-yf(x)}$. With this, Friedman, Hastie, and Tibshirani proved the following result (see Friedman, Hastie, and Tibshirani 2000):

Theorem 3.1.1 *The Real AdaBoost algorithm fits an additive logistic regression model by stagewise and approximate optimization of* $J(F) = E[e^{-yF(x)}]$*.*
Furthermore, at the optimal $F(x)$*, the weighted conditional mean of* y *is* 0 *as*

$$\frac{\delta J(F(x))}{\delta F(x)} = -E(e^{-yF(x)})y = 0.$$

3.1.2 LogitBoost

LogitBoost is another boosting algorithm which was formulated by Hastie, Tibshirani and Friedman (2000) The Logit Boost variant consists of using adaptive **Newton steps** to fit an additive logistic model. As compared to gradient descent, Newton's method also uses curvature information to take a more direct route, therefore requiring the second derivative aswell. At each iteration one approximates the desired function $F(x)$ by a quadratic function and then takes a step towards the minimum of that quadratic function. Instead of minimizing the exponential loss, LogitBoost minimizes the logistic loss (negative conditional log-likelihood)

$$L(y, F) = ln(1 + e^{-yF(x)}).$$

In their paper Friedman et al. (see Friedman, Hastie, and Tibshirani 2000) consider a two-class case with response $y_i \in Y = \{0, 1\}$ and represent the probability of $y = 1$ by $p(x)$ where

$$p(x) = \frac{e^{F(x)}}{e^{F(x)} + e^{-F(x)}}.$$

Again considering an update to a function, as in the previous sections, as $F(x)+f(x)$ the expected log-likelihood is

$$El(F + f) = E[2y(F(x) + f(x)) - log(1 + e^{2(F(x)+f(x))})]$$

and the first and second derivatives at $f(x) = 0$ respectively are

$$s(x) = \frac{\partial El(F(x) + f(x))}{\partial f(x)}|_{f(x)=0} = 2E(y - p(x)|x)$$

and

$$H(x) = \frac{\partial^2 El(F(x) + f(x))}{\partial^2 f(x)}|_{f(x)=0} = -4E(p(x)(1 - p(x))|x).$$

The Newton Update then works as follows

$$\begin{aligned} F(x) \leftarrow F(x) - H(x)^{-1}s(x) &= F(x) + \frac{1}{2}\frac{E(y - p(x)|x)}{E(p(x)(1 - p(x)|x)} \\ &= \frac{1}{2}E_w\left(\frac{y - p(x)}{p(x)(1 - p(x))}|x\right) \end{aligned} \tag{3.1.1}$$

with $w(x) = p(x)(1 - p(x))$.

With this, one arrives at the boosting Algorithm 4, referred to as **LogitBoost**, which uses Newton steps for fitting an additive logistic model by maximum likelihood.

Algorithm 4 LogitBoost (2 classes)

```
1: Initialize weights $w_i = 1/n$ for $i = 1, \dots, n$
2: for $t = 1$ to $T$ and while $F \neq 0$ do
3:     Compute the working responses $z_i$ and the weights $w_i$
           $$z_i = \frac{y_i - p(x_i)}{p(x_i)(1 - p(x_i))} \quad , \quad w_i = p(x_i)(1 - p(x_i))$$
4:     Fit $f_t(x)$ by a weighted least-squares of $z_i$ to $x_i$ with weights $w_i$
5:     Update the function: $F(x) = F(x) + \frac{1}{2} f_m$ and set $p(x) = \frac{e^{F(x)}}{e^{F(x)} + e^{-F(x)}}$
6: end for
7: Output the final classifier as
           $$F(x) = sign(\sum_{m=1}^{M} f_m(x))$$
```

3.1.3 Gentle AdaBoost - GentleBoost

As a final addition, the *Gentle AdaBoost* algorithm will be shown, which was also formulated by Friedman, Hastie, and Tibshirani 2000. Continuing the abbreviation of Section 3.1.1 with the RealBoost, out of simplicity the Gentle AdaBoost algorithm will be referred to as **GentleBoost** in this thesis. The GentleBoost alorithm, taken from Friedman, Hastie, and Tibshirani 2000, is presented as Algorithm 5.

As opposed to proceeding like the RealBoost, by optimizing $E(e^{-y(F(x)+f(x))})$ with respect to f at each iteration t, the GentleBoost takes adaptive Newton steps just like the LogitBoost algorithm described in the previous Section 3.1.2. Therefore, instead of fitting a class probability estimate, GentleBoost in step 3 uses weighted least-squares regression at each iteration.

Even though RealBoost and GentleBoost are optimizing the same loss function and perform similarly on identical data sets (as will be seen later in Section 4.2.2), GentleBoost is numerically superior because it doesn't require the computation of log-ratios which can be numerically unstable, as they involve quotients, with the possibility of the denominator approaching zero. The algorithm GenleBoost is called *"gentle"* because it is considered to be both conservative and more stable as compared to the RealBoost algorithm.

In their paper, Friedman, Hastie, and Tibshirani (2000) showed that since,

$$\left.\frac{\delta J(F(x) + f(x))}{\delta f(x)}\right|_{f(x)=0} = -E(e^{-yF(x)} y|x) \quad and$$
$$\left.\frac{\delta^2 J(F(x) + f(x))}{\delta f(x)^2}\right|_{f(x)=0} = E(e^{-yF(x)}|x) \quad (as\ y^2 = 1)$$

Algorithm 5 GentleBoost

```
1: Initialize weights $w_i = 1/n$ for $i = 1, \dots, n$ and set $F_0(x) = 0$.
2: for $t = 1$ to $T$ do
3:     Using weights $w_t$, fit the regression function $f_t(x)$ by weighted least-squares of $y_i$ to $x_i$.
4:     Update the function $F(x)$ as
           $$F_t(x) = F_{t-1}(x) + f_t(x).$$
5:     Update the weights as
           $$w_{t+1} = \frac{w_t e^{-y f_t(x)}}{Z_t}$$
       where $Z_t$ is a normalization factor, chosen such that $w_t$ will be a distribution $\sum_i^n w_{t+1}(i) = 1$.
6: end for
7: Output the final classifier as
       $$sign(F(x)) = sign(\sum_{t=1}^{T} f_t(x)).$$
```

the Newton update will be

$$F(x) \leftarrow F(x) + \frac{E(e^{-yF(x)} y|x)}{E(e^{-yF(x)}|x)} = F(x) + E_w(y|x),$$

with weights $w(x, y) = e^{-yF(x)}$ and consequently they obtained the following result:

Theorem 3.1.2 *The GentleBoost algorithm uses Newton steps for minimizing* $J(F) = E[e^{-yF(x)}]$.

As one could imagine, since both algorithms are using Newton steps, there is a similarity between the updates for the GentleBoost and those for the LogitBoost. Using the following notation

$$P = P(y = 1|x) \quad and \quad p(x) = \frac{e^{F(x)}}{(e^{F(x)} + e^{-F(x)})},$$

the Newton step update of GentleBoost can be written as

$$\frac{E(e^{-yF(x)} y|x)}{E(e^{-yF(x)}|x)} = \frac{e^{-F(x)} P - e^{F(x)}(1-P)}{e^{-F(x)} P + e^{F(x)}(1-P)} = \frac{P - p(x)}{(1-p(x))P + p(x)(1-P)}, \tag{3.1.2}$$

and analogous those for LogitBoost (see Equation (3.1.1)) as

$$\frac{1}{2} \frac{P - p(x)}{(1-p(x))p(x)}. \tag{3.1.3}$$

If $p(x) \approx 1/2$, then the two expressions (3.1.2) and (3.1.3) are nearly the same. However, if $p(x)$ become extreme, such as $p(x) \approx 0$ and $P \approx 1$, then (3.1.3) can grow to enormous size, while (3.1.2) will always remain in its interval of $[-1, 1]$.

3.2 Multi-class Boosting

This section will focus on the extension of boosting to the multi-class problem, with emphasis to the expansions of AdaBoost.
In the previous sections only binary classification problems were considered, where the set of labels Y contained only two elements $\{-1, 1\}$. However, in most practical learning problems the set Y will be a finite set of size k, of $Y = \{l_1, \dots, l_k\}$ class labels or a real bounded interval $Y \in \mathbb{R}$. Thus, in general one will be faced with a multi-class or a regression problem respectively.
Although not a focus of this thesis, a third case called the multi-label case is possible, where each observation $x \in X$ may belong to multiple labels in Y. Hence, an observation x will be paired with a subset $Y_i \subseteq Y$ of all possible labels. If $\#Y_i = 1$ for all $i = 1, \dots, n$, then it returns to a single-label or simply called a multi-class problem.

The first presented multi-class extensions of AdaBoost (see Algorithm 1) are **AdaBoost.M1** and **AdaBoost.M2**, which were introduced by Schapire and Freud in 1997 (see Freund and Schapire 1997). The difference between them is the way they treat each class by using different loss functions. In AdaBooost.M1 the weight of a base classifier is a function of the error rate, while in AdaBoost.M2 the sampling weights are increased for instances for which the pseudoloss exceeds 1/2.

This is followed by an expansion of the LogitBoost to k classes - **Logit k-class**, which was presented by Friedman, Hastie and Tibshirani in their work (see Friedman, Hastie, and Tibshirani 2000). This boosting algorithm uses quasi-Newton steps for fitting an additive symmetric logistic model by maximum-likelihood.

Lastly, **SAMME** a new algorithm for multi-class boosting was proposed in 2006 by Zhu et. al. referred to as **SAMME** - Stagewise Additive Modeling using a Multi-class Exponential loss function - or simply called *Multi-class AdaBoost*. It directly extends the AdaBoost algorithm to the multi-class case without reducing it to multiple two-class problems.

3.2.1 AdaBoost.M1

The first, simpler expansion AdaBoost.M1 was presented by Freud and Schapire in 1996 (see Freund and Schapire 1996a).

Again the problem will consist of a training set of size n, $\{(x_i, y_i)\}_{i=1,\dots,n}$, with input values $x_i \in X$, output values $y_i \in Y$ and a *weak learner*. Here X denotes some space and Y is a finite set of cardinality k (number of classes).

In AdaBoost.M1 the initial weight distriubtion w_1 is uniform over the data. The α_t are computed similiarly to AdaBoost, as was done in Section 2.2, as a function of the error term ϵ_t. To compute the weights w_{t+1} from the previous weights w_t and the weak hypothesis h_t, the weight of sample i is multiplied by some number α_t if h_t was able to correctly classify, otherwise the weight is left unchanged. Therefore, samples which were "*easily*" classified by the previous hypothesis get a lower weight,

while those that were *"hard"* to classify get a higher weight. Hence, AdaBoost.M1 like the original AdaBoost focuses on those samples which were hardest for the weak learner in the previous iteration.
At each iteration $t = 1, \dots, T$, the chosen weak learner is provided with the distribution of the weights over the training set, and constructs a hypothesis h_t which minimizes the training error ϵ_t (which is measured with respect to the distriubtion of w_t). The final hypothesis $H(X)$ is again a weighted vote of the weak hypotheses, where the class y is chosen, which maximizes the sum of weights of weak hypotheses predicting the class.
As a result, the AdaBoost.M1 - Algorithm 6 - was stated by Freud and Schapire in their work *"Experiments with a New Boosting Algorithm"*, in 1996 (see Freund and Schapire 1996a).

Algorithm 6 AdaBoost.M1

```
1: Initialize the weights for the individual sample points as $w_1(i) = 1/n$ for all $i = 1, \dots, n$.
2: for $t = 1$ to $T$ do
3:     Using the weak learner with the distribution of $w_t$, train the classifier with respect to the weighted data and obtain the classifier $h_t : X \to Y$
4:     Calculate the error of $h_t$
           $$\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} w_t(i)$$
       if $\epsilon_t > \frac{1}{2}$, then set $T = t - 1$ and stop the loop.
5:     Set
           $$\alpha_t = \frac{1}{2} ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)$$
6:     Update the weights $w_t$:
           $$w_{t+1} = \frac{w_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if correctly classified } y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if incorrectly classified} y_i \neq h_t(x_i) \end{cases},$$
       where $Z_t$ is a normalization factor, chosen such that $w_t$ will be a distribution $\sum_{t=1}^{N} w_{t+1}(i) = 1$
7: end for
8: Output the final hypothesis
           $$H(x) = \arg\max_{y \in Y} \sum_{t:h_t(x)=y} \alpha_t$$
```

The major drawback of AdaBoost.M1 is, that the error ϵ_t of the weak hypothesis h_t has to be less or equal to 1/2. In the binary case $k = 2$, this can easily be achieved, as the weak hypothesis have to be slightly better than random guessing. It was shown in Section 2.3, that if the weak hypotheses have an error better than 1/2, then the training error of the final hypothesis H falls to zero exponentially fast (refer to Theorem 2.3.1). Hence for binary classification problems, this means that

the weak hypotheses need only be slightly better than random guessing.
For multi-class hypotheses with $k > 2$ the expected error of random guessing is $1 - \frac{1}{k}$, which increases with the number of possible classes k. Thus, the requirement of an error to be less than 1/2 is strong and often not met.

It is important to note, that while the training error of AdaBoost.M1 is small, this does not imply that the test error is small as well. However, Schapire and Freud showed that if the weak hypotheses are "*simple*" and the number of iterations T is "*small*", then the difference between the training and test errors can also be theoretically bounded (see Freund and Schapire 1997). In their work (see Freund and Schapire 1997), Schapire and Freund presented a proof to the following theorem:

Theorem 3.2.1 *Suppose a weak learner, when called by AdaBoost.M1 (Algorithm 6), generates hypotheses with errors $\epsilon_1, \dots, \epsilon_T$, where ϵ_t is defined as in Algorithm 6 and assuming $\epsilon_t \leq 1/2$, for all $t = 1, \dots, T$.*
Then the error ϵ of the final hypothesis $H(\cdot)$ is bounded by

$$\epsilon = P_{i \sim w_t}[H(x_i) \neq y_i] \leq 2^T \prod_{t=1}^{T} \sqrt{\epsilon_t(1 - \epsilon_t)}. \tag{3.2.1}$$

3.2.2 AdaBoost.M2

The motivation of Schapire's and Freund's (see Freund and Schapire 1997) second multi-class algorithm - AdaBoost.M2 - is to overcome the difficulty of an error term less than 1/2 for the weak learners which are required in AdaBoost.M1. This is done by extending the communication between the boosting algorithm and the weak learner. Therefore the difference between M1 and M2 is in the way they treat each class. In the M1 variant, the weight of a weak learner is a function of the error rate. In M2, the sampling weights are increased for instances for which the error exceeds 1/2.

The first step is to allow the weak learner to generate more expressive hypotheses, which, rather than identifying a single label in Y, instead choose a set of "*plausible*" labels. This can often be easier than simply choosing just one single label. The weak learner does this by creating a vector $[0, 1]^k$ to indicate a *"degree of plausibility"*, where values close to 1 represent high plausibility and those close to 0 low plausibility. It is important to note, that this vector is not a probability vector, as the sum of the components does not have to be equal to one.

In addition to the weak learner becoming more expressive, a more complex requirement is placed on the performance of the weak hypotheses. Instead of using the prediction error, the weak hypothesis h_t must do well with respect to a measure called the *pseudo-loss*. While the ordinary error is computed with respect to a distribution over examples, the pseudo-loss is computed with respect to a distribution over the set of all pairs of examples and incorrect labels. Therefore, the algorithm focuses not only on the samples which are hard to classify but especially on the incorrect classes that are hard to discriminate.
Analogous to the *weak learning assumption* (see Section 2.4), it will be shown that,

the boosting algorithm AdaBoost.M2, which is based on these ideas, achieves boosting if each weak hypothesis has pseudo-loss slightly better than random guessing.

$$\epsilon_t = \frac{1}{2} \sum_{(i,y)\in B} w(t)(i,y)\left(1 - h_t(x_i,y_i) + h_t(x_i,y)\right) \tag{3.2.2}$$

To define the pseudo-loss, one must first describe B as the set of all mislabels $B = \{(i,y) \ : \ i \in \{1,\dots,n\}, y \neq y_i\}$, while a *mislabel* is a pair (i,y) where y is an incorrect label/class associated with example/data point i. A *mislabel distribution* is a distribution defined over the set B of all mislabels.

One can interpret the mislabel (i,y) as representing a binary question. For a fixed training example (x_i,y_i), a given hypothesis h is used to answer $k-1$ binary questions. Each question asking if the class/label of sample x_i is the correct class y_i or one of the incorrect labels y. In other words, it is asked that the correct label y_i will be discriminated from the incorrect label y. Hence, the weight $w_t(i,y)$ assigned to this mislabel represents the importance of distinguishing the incorrect label y on example x_i.

Values of h_t in $(0,1)$ are to be interpreted as probabilities. Assuming for simplicity that h_t only takes values in $\{0,1\}$. Then, if it holds for the weak hypothesis h_t that $h_t(x_i,y_i) = 1$ and $h_t(x_i,y) = 0$, it means that h_t has correctly predicted x_i's class as y_i and not y. Conversely, if $h_t(x_i,y_i) = 0$ and $h_t(x_i,y) = 1$ then the weak hypothesis has predicted incorrectly. Lastly if $h_t(x_i,y_i) = h_t(x_i,y)$ then the weak hypothesis can be considered a random guess and be chosen uniformly at random.

However, if instead a more general case is considered, where the weak hypothesis h takes values in $[0,1]$, then $h(x,y)$ can be interpreted as a randomized decision for the procedure mentioned previously above. In other words, that means that, first a random bit $b(x,y)$ is chosen which is 1 with probability $h(x,y)$and 0 otherwise (see Freund and Schapire 1997),

$$P(b(x,y)=1) = h(x,y) \quad , \quad P(b(x,y)=0) = 1 - h(x,y).$$

Applying the reasoning above to a stochastically chosen binary function b. The probability of choosing the incorrect answer y to the question above is

$$\begin{aligned} P\big(b(x_i,y_i) = 0 \wedge b(x_i,y) = 1\big) + P\big(b(x_i,y_i) = b(x_i,y)\big) \\ = \frac{1}{2}(1 - h(x_i,y_i) + h(x_i,y)). \end{aligned} \tag{3.2.3}$$

Therefore, if considering all $k-1$ binary questions as equally important, one can define the loss of the weak hypothesis to be the average over all $k-1$ questions, of the probability of an incorrect answer (3.2.3):

$$\frac{1}{k-1} \sum_{(i,y)\in B} \frac{1}{2}(1 - h(x_i,y_i) + h(x_i,y)) = \frac{1}{2}\left(1 - h(x_i,y_i) + \frac{1}{k-1} \sum_{(i,y)\in B} h(x_i,y)\right). \tag{3.2.4}$$

As already mentioned previously, a motivation of introducing pseudo-loss is to make it easier to discriminate similar labels/classes, i.e. put an emphasis on those binary questions which are the most difficult to answer. This is achieved by attaching a weight to the different questions. Then for each instance x_i and mislabel (i, y) a weight $\tilde{w}(i, y)$ is assosiated with the question that discriminates the label y from the correct label y_i. Here $\tilde{w} : \{1, \ldots, n\} \times Y \to [0, 1]$ is a function, refered to as *label weighting function*, which assigns a probability distribution over the $k - 1$ discrimination problems, binary questions, mentioned above $\sum_{(i,y)\in B} \tilde{w}(i, y) = 1$. The average in (3.2.4) is then replaced with a $\tilde{w}(i, y)$ weighted average to obtain[1] :

Definition 3.2.1 *The pseudo-loss of h on training instance i with respect to the label weighting function q is defined as:*

$$ploss_t(h, i) := \frac{1}{2} \sum_{(i,y)\in B} \tilde{w}(i, y) \left(1 - h(x_i, y_i) + h(x_i, y)\right). \tag{3.2.5}$$

In AdaBoost.M2 the weak learner's goal is to minimize the expected pseudo-loss (3.2.5) for given distribution w and weighting function q.

Schapire and Freund showed (see Freund and Schapire 1997), that a weak learner can be boosted if it can consistently produce weak hypotheses with pseudo-losses smaller than 1/2. Note that pseudo-loss 1/2 can be achieved trivially by any uninformative hypothesis. Furthermore, a weak hypothesis h with pseudo-loss $\epsilon \geq 1/2$ is also beneficial to boosting since it can be replaced by the hypothesis $1 - h$ whose pseudo-loss $1 - \epsilon < 1/2$.

Analogous to their statement about the error bound of AdaBoost.M1, in their work (see Freund and Schapire 1997), Schapire and Freund presented a proof to the following theorem:

Theorem 3.2.2 *Suppose a weak learner, when called by AdaBoost.M2 (Algorithm 7), generates hypotheses with pseudo-losses $ploss_1, \ldots, ploss_T$, where $ploss_t$ is defined as in Algorithm 6.*
Then the error ϵ of the final hypothesis $H(\cdot)$ is bounded by

$$\epsilon = P_{i\sim w_t}[H(x_i) \neq y_i] \leq (k - 1)2^T \prod_{t=1}^{T} \sqrt{ploss_t(1 - ploss_t)}. \tag{3.2.6}$$

3.2.3 SAMME

In their 2006 paper, *"Multi-class AdaBoost"* (see Zhu et al. 2009), Zhu, Rosset, Zou and Hastie present *SAMME* - **S**tagewise **A**dditive **M**odeling using a **M**ulti-class **E**xponential loss function. The algorithm's purpose is to naturally extend the original AdaBoost to the multi-class case without reducing it to a multiple two-class problem.

[1]Definition taken from Freund and Schapire' works, see Freund and Schapire 1996a

Algorithm 7 AdaBoost.M2

```
1: Initialize the weights for the individual sample points as $\tilde{w}_1(i,y) = 1/|B|$ for all mislabels $(i,y) \in B$.
2: for $t = 1$ to $T$ do
3:     Using the *weak learner* with the distribution of $\tilde{w}_t$, train the classifier with respect to the weighted data and obtain the hypotheses $h_t : X \times Y \to [0,1]$.
4:     Calculate the pseudo-loss of $h_t$
           $$ploss_t(h,i) := \frac{1}{2} \sum_{(i,y)\in B} \tilde{w}(i,y)\left(1 - h(x_i,y_i) + h(x_i,y)\right).$$
5:     Set
           $$\alpha_t = \frac{1}{2} ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$
6:     Update the weights $\tilde{w}_t$:
           $$\tilde{w}t + 1(i,y) = \frac{\tilde{w}_t(i,y)}{Z_t} \cdot exp(-\alpha_t)^{\frac{1}{2}(1+h(x_i,y_i)+h(x_i,y))}$$
       where $Z_t$ is a normalization factor, chosen such that $\tilde{w}_{t+1}$ will be a distribution $\sum_{t=1}^{N} \tilde{w}_{t+1}(i) = 1$
7: end for
8: Output the final hypothesis
       $$H(x) = \arg\max_{y\in Y} \sum_{t=1}^{T} \alpha_t h_t(x,y)$$
```

The algorithm is equivalent to a forward stagewise additive modeling algorithm that minimizes a novel exponential loss for multi-class classification. The algorithm is highly competitive in terms of misclassification error rate. It is worth noting that if the number of classes is $k = 2$, SAMME reduces to AdaBoost.

A difference between the SAMME Algorithm 8 and the standard AdaBoost - Algorithm 1 - is in the weights

$$\alpha_t = log(\frac{1-\epsilon_t}{\epsilon_t}) + log(k-1). \tag{3.2.7}$$

The parameters α_ts are positive if either $\epsilon_t < \frac{k-1}{k}$ or the accuracy of the weak classifiers is better than random guessing. With this, the new algorithm puts more weight on the misclassified data points than AdaBoost, and the new algorithm also combines weak classifiers a little differently than AdaBoost.

With the addition of the term $log(k-1)$, the SAMME algorithm puts more weight on the misclassified data points than the previously mentioned variations of the AdaBoost Algorithm M1 and M2. Consequently it also combines weak classifiers a little differently to obtain the final classifier $C(x)$, by the addition of the term $log(k-1)\sum_{t=1}^{T} \mathbb{1}(h_t(x) = y)$.

Algorithm 8 SAMME

```
1: Initialize the weights of the observations $w_1(i) = \frac{1}{n}$, for $i = 1, \dots, n$
2: for $t = 1$ to T do
3:     Fit a classifier $h_t(x)$ to the training data using the weights $w_t(i)$
4:     Calculate the error of $h_t(x)$:
           $$\epsilon_t = \sum_{i:h_t(x_i)\neq y_i}^{n} w_t(i) \ / \ \sum_{i=1}^{n} w_t(i)$$
5:     Calculate
           $$\alpha_t = log(\frac{1-\epsilon_t}{\epsilon_t}) + log(k-1)$$
6:     Set the new weights as
           $$w_{t+1}(i) = \frac{w_t(i)}{Z_t} \times \begin{cases} 1 & \text{if correctly classified } y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if incorrectly classified} y_i \neq h_t(x_i) \end{cases}, \quad i = 1, \dots, n$$
       where $Z_t$ is a normalization factor, chosen such that $w_t$ will be a distribution $\sum_{i=1}^{n} w_{t+1}(i) = 1$
7: end for
8: Output
           $$C(x) = \arg\max_k \sum_{t=1}^{T} \alpha_t \mathbb{1}(h_t(x) = y)$$
```

This addition of the term $log(k-1)$ in (3.2.7) is not arbitrary, since it makes the SAMME algorithm equivalent to fitting a forward stagewise additive model using a multi-class exponential loss function. As was already shown, see (2.3.9), the population minimizer of the exponential loss function is one half of the logit transformation. In their work (see Zhu et al. 2009) they show that the Bayes optimal classification rule in terms of minimizing the misclassification error, is equivalent to the population minimizer.

3.2.4 Logit k Classes

In their work, Friedman, Hastie and Tibshirani (see Friedman, Hastie, and Tibshirani 2000) considered k mutually exclusive classes y_j for a k-class problem, each taking values in $\{-1, +1\}$. Then similar to the previous sections the probability of $y_j = 1$ is set as $p_j(x) = P(y_j = 1|x)$. Friedman et al. defined the *symmetric multiple logistic transformation* as

Definition 3.2.2 *Given a k class problem , let $p_j(x) = P(y_j = 1|x)$. The symmetric multiple logistic transformation is defined as*

$$F_j(x) = ln(p_j(x)) - \frac{1}{k}\sum_{j=1}^{k} ln(p_j(x)) \tag{3.2.8}$$

or equivalently,

$$p_j(x) = \frac{e^{F_j(x)}}{\sum_{j=1}^{k} e^{F_j(x)}} \quad where \quad \sum_{j=1}^{k} F_j(x) = 0 \tag{3.2.9}$$

By pinning down the F_j's with the centering condition in (3.2.9), numerical stability is to be achieved. If this was not done, then one could add an arbitrary constant to each F_j and the probabilities would remain the same.

The LogitBoost for k-classes uses quasi-Newton steps for fitting an additive symmetric logistic model by maximum-likelihood. As supposed to the "normal" Newton method, the quasi-Newton method can also be used if the Hessian or Jacobian are unavailable or too cumbersome to compute for every iteration. Instead of calculating the matrizes directly they are merely approximated.

Algorithm 9 LogitBoost (k classes)

```
 1: Initialize weights $w_{ij} = 1/n$ for $i = 1, \dots, n$ and $j = 1, \dots, k$
 2: for $t = 1$ to $T$ and while $F \neq 0$ do
 3:     for $j = 1$ to $k$ do
 4:         Compute the working responses $z_{ij}$ and the weights $w_{ij}$
                $z_{ij} = \frac{y_{ij} - p_j(x_i)}{p_j(x_i)(1 - p_j(x_i))}$ , $w_{ij} = p_j(x_i)(1 - p_j(x_i))$
 5:         Fit $f_{tj}(x)$ by a weighted least-squares of $z_{ij}$ to $x_i$ with weights $w_{ij}$
 6:     end for
 7:     Set $f_{tj}(x) = \frac{k-1}{k}(f_{tj}(x) - \frac{1}{k}\sum_{j=1}^{k} f_{tj}(x))$, and $F_j(x) = F_j(x) + f_{tj}(x)$
 8:     Update $p_j(x) = \frac{e^{F_j(x)}}{\sum_{j=1}^{k} e^{F_j(x)}}$ , where $\sum_{j=1}^{k} F_j(x) = 0$
 9: end for
10: Output the final classifier as
                $F(x) = \arg\max_j F_j(x)$
```

Algorithm 9 is a natural generalization of the LogitBoost Algorithm 4 using k-classes.

Friedman et al. derived this Algorithm 9 by fist presenting the score and Hessian for the Newton algorithm corresponding to a standard multi-logit parametrization

$$G_j(x) = log\frac{P(y_j = 1|x)}{P(y_k = 1)}$$

with $G_k(x) = 0$, where the choice of the base class k is arbitrary. The expected conditional log-likelihood is

$$E((l)(G + g)|x) = \sum_{j=1}^{k-1} E(y_j|x)(G_j(x) + g_j) - log(1 + \sum_{j=1}^{k-1} e^{G_j(x)+g_{jk}(x)})$$

with

$$s_j(x) = E(y_j - p_j(x)|x) \quad , \quad j = 1, \dots, k-1$$

and the Hessian as

$$H_{j,i}(x) = -p_j(x)(\delta_{ji} - p_i(x)) \quad , \quad j, i = 1, \dots, k-1.$$

As it is quasi-Newton, Friedman et al. used a diagonal approximation to the Hessian, producing the updates

$$g_j(x) = \frac{E(y_j - p_j(x)|x)}{p_j(x)(1 - p_j(x))} \quad , \quad j = 1, \dots, k-1.$$

Now to convert the symmetric parametrization, it is to note that $g_j = 0$ and set $f_j(x) = g_j - \frac{1}{k}\sum_{i=1}^{k} g_i(x)$. Again, any class could be used as a base for this procedure. By averaging over all choices for the base class the update will be

$$f_k(x) = \frac{k-1}{k}\left(\frac{E(y_j - p_j(x)|x)}{p_j(x)(1 - p_j(x))} - \frac{1}{k}\sum_{j=1}^{k}\frac{E(y_j - p_j(x)|x)}{p_j(x)(1 - p_j(x))}\right).$$

The advantage of quasi-Newton steps is the removing of the dependency of choosing a base class.

3.3 Regression Boosting

As the previous Section 3.2 introduced a few possible multi-class expansions to boosting, this chapter will show boosting in the setting of a regression problem. However, it is not a focus of this thesis and will not be presented in the experiments of Chapter 4.

In the regression case, given a set of data $\{(x_i, y_i)\}_{i=1,\dots,n}$, the label space Y will be a subset of the real numbers $\mathbb{R}$, as opposed to being restricted to a finite set of labels in the binary or multi-class scenario.

3.3.1 AdaBoost.R

One of, if not the first application of boosting to a regression problem was done by Freund and Schapire in their paper (see Freund and Schapire 1997) in 1996, where their idea was to convert the regression problem into a classification one.

If the label space is the closed interval $Y = [0, 1]$ and the sample (x, y) is chosen at random according to some distribution, then the goal is to find a function $H : X \to Y$ which predicts the value of y, for a given value x. This is done by minimizing a loss function L. Probably the most popular choice is to find an H with small *mean squared error* (MSE)

$$H = \underset{h \in \mathcal{H}}{\arg\min}\, E[(h(x) - y)^2] \tag{3.3.1}$$

which, given a set of data $S = \{(x_i, y_i)\}_{i=1,\dots,n}$ equates to minimizing the empircial MSE

$$H = \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \left(h(x_i) - y_i\right)^2 . \tag{3.3.2}$$

Freund and Schapire (see Freund and Schapire 1997) reduced the regression problem to a binary classification problem, and then applied their AdaBoost. The general informal idea is to map each example (x_i, y_i) to an infinite set of binary questions, one for each $y \in Y$, and each asking if the correct label y_i is bigger or smaller than y. Therefore, for every example (x_i, y_i) a set of examples are defined and indexed by pairs (i, y) for all $y \in [0, 1]$: the associated instance is $\tilde{x}_{i,y} = (x_i, y)$, and the label is $\tilde{y}_{i,y} = I_{y \geq y_i}(y)$. Even though one must maintain an infinitely large training set, this method can be implemented efficiently (see Freund and Schapire 1997), as the extension from finite sets to infinite training sets is straightforward. Every hypothesis $h : X \to Y$ is equally reduced to a binary valued hypothesis $\tilde{h} : X \times Y \to \{0, 1\}$,

$$\tilde{h}(x, y) = \begin{cases} 1 & \text{if } y \geq h(x) \\ 0 & \text{if } y < h(x) \end{cases} .$$

In this sense, $\tilde{h}$ attempts to answer these binary questions in a natural way using the estimated value $h(x)$.

As it was done with classification problems, a distribution w over the training set is assumed, which is often considered uniform $w(i) = 1/n$. In compliance with the previous reductions, this distribution is mapped to a density $\tilde{w}$ over pairs (i, y), such that the minimization of classification error in the reduced space is equivalent to minimization of the MSE for the original problem. This is done by defining

$$\tilde{w}(i, y) = \frac{w(i)|y - y_i|}{Z}$$

with Z as a normalization constant,

$$Z = \sum_{i=1}^{n} w(i) \int_0^1 |y - y_i| dy.$$

Now if calculating the binary error of $\tilde{h}$ with respect to the density $\tilde{w}$,

$$\begin{aligned} \sum_{i=1}^{n} \int_0^1 |\tilde{y}_(i, y) - \tilde{h}(\tilde{x}_{i,y})| \tilde{w}(i, y) dy &= \frac{1}{Z} \sum_{i=1}^{n} w(i) \left| \int_{y_i}^{h(x_i)} |y - y_i| dy \right| \\ &= \frac{1}{2Z} \sum_{i=1}^{n} w(i)(h(x_i) - y_i)^2, \end{aligned} \tag{3.3.3}$$

one can see that it is directly proportional to the mean squared error.

Using this principle of reduction, Freund and Schapire presented the boosting Algorithm *AdaBoost.R* (see Freund and Schapire 1997). There, for every pair (i, y), AdaBoost.R maintains a weight $w_t(i, y)$, which is initialized by the weight $\tilde{w}(i, y)$,

Algorithm 10 AdaBoost.R

```
1: Initialize the weight vector for the pair $(i, y)$ as
       $$\tilde{w}_1(i, y) = \frac{w(i)|y - y_i|}{Z}$$
   for all $i = 1, \ldots, n$, $y \in Y$ and with normalization constant $Z$
       $$Z = \sum_{i=1}^{n} w(i) \int_0^1 |y - y_i| dy.$$
2: for $t = 1$ to $T$ do
3:     Normalize the weights
           $$\bar{w}_t = \frac{w_t}{\sum_{i=1}^{n} \int_0^1 w_t(i, y) dy}$$
4:     Using the weak learner with the distribution of $\bar{w}_t$, train the classifier with
       respect to the weighted data and obtain the hypotheses $h_t : X \times Y$.
5:     Calculate the loss of $h_t$
           $$\epsilon_t = \sum_{i=1}^{n} \left| \int_{y_i}^{h_t(x_i)} \bar{w}_t(i, y) \right|$$
       if the error $\epsilon_t > 1/2$, then set $T = t - 1$ and abort the loop.
6:     Set
           $$\alpha_t = \frac{1}{2} ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)$$
7:     Update the weights $\tilde{w}_t$ for every pair $(i, y)$:
           $$w_{t+1}(i, y) = \begin{cases} w_t(i, y) & \text{if } y_i \leq y \leq h_t(x_i) \text{ or if } y_i \geq y \geq h_t(x_i) \\ w_t(i, y)\alpha_t & \text{else} \end{cases}.$$
8: end for
9: Output the final hypothesis
       $$H(x) = \inf \left\{ y \in Y \middle| \sum_{t: h_t \leq y} \alpha_t \geq \frac{1}{2} \sum_t \alpha_t \right\}.$$
```

defined above. Those weights are then normalized to obtain the $\bar{w}_t$, which are consequently used by the weak learner to obtain a weak hypothesis h_t, which minimizes the loss ϵ. Then the weights updated as described by the reduction.

Similar to the AdaBoost.M2 algorithm, AdaBoost.R not only varies the distribution over the examples (x_i, y_i), but also modifies in every round the definition of the loss suffered by a hypothesis on each example. Therefore, even though the goal is to minimize the squared error, the weak learner must be able to handle loss functions that are more complicated than MSE.

Since the reduced weak hypothesis $\tilde{h}(x, y)$ are non decreasing as a function of y, a final hypotheses $\tilde{h}$ as the threshold of a weighted sum of these hypothesis is also non-decreasing as a function of y. With a binary output of $\tilde{h}$, for every x there is one value of y for which $\tilde{f}(x, y) = 0$ for all $y' < y$ and $\tilde{f}(x, y) = 1$ for all $y' > y$. This is equivalent to the value of y given $H(x)$, which was defined in the AdaBoost.R (Algorithm 10).

There is still the conundrum of maintaining weights $w_t(i, y)$ over an infinity set of points y. If those weights are viewed as a function of y, the weights $w_t(i, y)$ become a piece-wise linear function. At the first iteration $t = 1$, $w_1(i, y)$ has two linear pieces, and each update at step 7 of Algorithm 10 has the potential to break one of the parts in two at the point $h_t(x_i)$. The storing and updating of such piece-wise linear functions are all straightforward operations. Furthermore, the integrals can be evaluated explicitly since these only involve integration of piece-wise linear functions.

Freund and Schapire stated the following theorem in their work (see Freund and Schapire 1997), which describes the performance of AdaBoost.R, by providing an upper bound for the error.

Theorem 3.3.1 *Suppose a weak learner, when called by AdaBoost.R (Algorithm 10), generates hypotheses with errors $\epsilon_1, \dots, \epsilon_T$, where ϵ_t is defined as in Algorithm 10*
Then the mean squared error ϵ of the final hypothesis $H(\cdot)$ is bounded by

$$\epsilon = E[(H(x_i) - y_i)^2] \leq 2^T \prod_{t=1}^{T} \sqrt{\epsilon_t(1 - \epsilon_t)}. \tag{3.3.4}$$

3.3.2 LS, LAD and M - Boosting

Several loss criteria can be applied to the Generic FDG Algorithm (refer to Algorithm 2 in Section 2.7.3) to obtain different regression boosting algorithms.
In his work Friedman presented a handful of algorithms some of which will be briefly mentioned here (for further details and actual presentations of the algorithms refer to Friedman 2001 or Bühlmann and Geer 2011):

LS Boost

The LS(least-sqaures) Boost, or sometimes referred to as *L2 Boost*(see Bühlmann and Geer 2011) is a functional gradient descent algorithm, using the squared error as a loss function $L(y, F) = (y - F)^2/2$.

LAD Boost

The LAD (least-absolute-deviation) Boost, analogue to the LS Boost uses the absolute error as a loss function $L(y, F) = |y - F|$.

M Boost

Lastly the M Boost, analogue to the other two uses the Huber's loss[2](M) as a loss function

$$L(y, F) = \begin{cases} \frac{1}{2}(y - F)^2 & |y - F| \leq \theta \\ \theta(|y - F| - \theta/2) & |y - F| > \theta. \end{cases}$$

[2]Defined by Peter Huber in 1964 in his work *“Robust Estimation of a Location Parameter”*.

Chapter 4

Simulations

In this section all calculations in the simulations will be done with the open source programming language for statistical computing **R** (see R-Core-Team 2013).

4.1 Imbalanced Data

To refer to a data set as *"imbalanced data"*, generally implies a classification problem where the classes are not represented equally. Suppose a given binary problem has a total of $n = n_1 + n_2$ observations, where n_1 and n_2 represent the number of observations which are part of the first and second class, respectively. In most cases these data sets will not exactly have the equal number of instances in each class, though a small difference $n_1 \approx n_2$ often does not matter and will have little effect. Now if one of the two is significantly greater than the other, $n_1 \gg n_2$, then the data of the classification problem will be referred to as *imbalanced*. Building models based on such data sets tend to emphasize these observations belonging to the bigger class (n_1). This imbalance can occur in multi-class classification problems as well as in binary/two-class problems.

In many situations, imbalanced data sets are not just exceptions, but occur more often than expected. This is especially the case in medical research, as the relevant data sets have two classes for the main outcome, of a feature being present or not. This feature, could be a positive or negative result of an examination to determine if the patient had signs of cancer or not. Hopefully, the majority of patients will be healthy, while the rest will be diagnosed positively. Another example, which will be discussed in detail in Section 4.2, would be data concerning the outcome of a sale effort of a subscription. The number of successful sales will make up only a small percentage of all attempts.

As was the case in the examples mentioned before, the goal of a certain classification problem might not just be to achieve an overall low error rate. It might be more crucial to rightly determine one class as opposed to the others, i.e. assigning different levels of importance to the classes. With such imbalanced problems, often more important than an overall low error rate is the correct classification of the minority class.

Since the usual classifier tends to be more sensitive to detecting the majority class and less so the minority class, it becomes necessary to pre-process imbalanced data before it is used in a model. Therefore, if nothing is done beforehand the classification will output a prediction which will be biased towards the majority class, which can be seen later in Section 4.2.2.

Chen, Liaw and Breiman (see Chen, Liaw, and Breiman 2004) suggested two common approaches to accommodate the problem of imbalanced data. The first solution is to assign a higher cost to misclassification of the minority class, and simultaneously trying to minimize the overall cost. In a way this is already indirectly done in boosting algorithms through the re-weighting of examples which were hard to classify in each iteration.
The second proposed solution is to use a pre-sampling method, thereby either reducing the sampling size of the majority class (*under-* or *down-sampling*), increasing the size of the minority class (*over-sampling*) or as a middle ground doing both at the same time.

It is important to note that, if such sampling/balancing methods are to be applied, then the sampling has the be done *after* the data was split into a training and a test data set. This is done because otherwise the independence of training and test set is not guaranteed.

4.1.1 Under-sampling

Maybe the most popular strategy to apply to imbalanced data sets is to *"under-sample"*. Here all classes are downsized by reducing the size of the majority classes down to a number of observations equal to the number of observations in the minority class. As a result, every class will have the same number of observations.

In detail, given a training set, denoted as X_{train}, a sample of size n_{min} (the size of the minority class) for each class will be drawn from X_{train} with replacement, thus, resulting in a new set $\tilde{X}_{train}$, with n_{min} observations for each class. As a consequence of the reduced sample size, the more imbalanced the dataset is the more samples will be discarded when under-sampling is applied, thus loosing potentially useful information.

4.1.2 Naive-sampling

"Naive-sampling", just like under-sampling, downsizes all classes to the size of the minority class n_{min}. However, instead of drawing with replacement from the training set X_{train}, it will be done *without* replacement.

4.1.3 Over-sampling

Conversely, over-sampling implies the increase of all classes, which are smaller than the majority class. This results in every class having as many observations as the majority class.

Analogous to under-sampling, a sample of size n_{max} (the size of the majority class) for each class will be drawn from X_{train} with replacement. The final set $\tilde{X}_{train}$ will contain n_{max} observations for each class. As each class will have as many observations as the majority class, this method can be very computationally intensive, especially compared to under-sampling.

4.1.4 Same-size-sampling

A combination of under- and over-sampling, *"same-size-sampling"* downsizes "big" classes, such as the majority class, and increases the "small" ones such as the minority class. The goal is to maintain the same size of the training data and to have the same number of observations for each class.

Assuming there are k different classes and a total of n observations in the training set X_{train}. Then, depending if the class has observations greater or less than $\lceil n/k \rceil$, it will either be downsized or increased to $\lceil n/k \rceil$ observations. Hence, each class will have the same number of observations and the total number will not have changed $\#\tilde{X}_{train} = \#X_{train}$, as it did in under- and over-sampling.

4.2 Binary Experiment

This section will show the application of boosting on an imbalanced data set, containing a binary response (*"no"* or *"yes"*). It will start off with an explanation of the chosen data set and will then proceed with the simulation of the boosting algorithms.

Three boosting algorithms **AdaBoost**, **RealBoost** and **GentleBoost**, which were described in Section 3.1, will be applied to the data set, to obtain different results. The reasoning behind the choice of those particular three is the following. AdaBoost is chosen because it is the first boosting algorithm conceived and is the foundation of all successors. As it computes the weighted error rate, it is of interest to compare it with an algorithm which uses something different, namely weighted probability estimates to update an additive logistic model. Therefore, the second choice will be RealBoost. Lastly, GentleBoost (just like LogitBoost) uses Newton steps and uses weighted least-squares regression and since it therefore doesn't require the computation of log-rations, which can be numerically unstable, it will be the third choice. As classification or regression trees (**CART**, refer to Section 2.4.1) are arguably the most popular they will be the choice of *weak-learner* for each of the chosen boosting methods.

The simulations of these three boosting algorithms will be done using the **R**-function"ada" which is provided by the **R**-package *ada* (see Culp, Johnson, and Michailidis 2016 and R-Core-Team 2013).

4.2.1 Data Set

The Bank dataset provided by Moro, Cortez and Rita, is publicly available[1] for research (for further details see Moro, Cortez, and Rita 2014 and Moro, Laureano, and Cortez 2011).

The data is the result of direct marketing campaigns of a Portuguese banking institution, which used direct marketing to promote their product. Within a campaign, the internal contact centre had human agents use the telephone, to contact clients to sell the bank term deposit and each time a contact was established, a set of attributes was stored. The sale was motivated by offering an attractive long-term deposit deal, with good interest rates. Occasionally, more than one contact to the same client was required, in order to assess if the bank term deposit would be (*'yes'*) or would not be (*'no'*) subscribed.
As this type of marketing has the drawback of possibly generating a negative attitude among customers towards banks due to the intrusion of privacy, it was of importance to improve efficiency: an approximate number of successes should be achieved with the fewest number of contacts possible. Therefore the classification goal is to predict whether or not the client will subscribe a term deposit (variable y), based on previous experiences.

The individual campaigns were integrated and outputted together, as single campaigns were merged together. The data used in this simulation, will be a subset of the 79354 contacts which were accumulated during 17 campaigns.
It consists of 45211 observations with no missing values, 16 variables (see Table 4.1), both categorical and numerical, and the binary result y of either (*'yes'*) or (*'no'*). This dataset is unbalanced, as less than 12% (11.7%) records are related with successes (see the Table 4.2).

Table 4.1: Variable description of bank data

Variable	Type	Description
age	numeric	age of participant
job	categorical	type of job
martial	categorical	marital status ("married","divorced","single")
education	categorical	type of education ("unknown","secondary","primary","tertiary")
default	categorical	has credit in default? (binary: "yes","no")
balance	numeric	average yearly balance, in euros
housing	categorical	has housing loan? (binary: "yes","no")
loan	categorical	has personal loan? (binary: "yes","no")
contact	categorical	contact communication type ("unknown","telephone","cellular")
day	numeric	last contact day of the month
month	categorical	last contact month of year (categorical: "jan",...,"nov", "dec")
duration	numeric	last contact duration, in seconds (numeric)
campaign	numeric	number of contacts performed during this campaign and for this client
pdays	numeric	number of days that passed after client was last contacted previously
previous	numeric	number of contacts performed before this campaign and for this client
poutcome	categorical	outcome of the previous marketing campaign ("unknown","other","failure","success")

[1]The dataset can be found at the UC Irvine Machine Learning Repository (UCI): https://archive.ics.uci.edu/ml/datasets/Bank+Marketing

Table 4.2: Bank responses

no	*yes*
39922	5289

4.2.2 Binary Simulation

The dataset is split (sed.seed(1308)) into a training set, consisting of 70% of the observations and into a test set, made out of the remaining 30% (which results in 31647 training-observations and 13564 test-observations). Each of the three boosting algorithms (AdaBoost, RealBoost and GentleBoost) is given the same training dataset to model a final hypothesis, which is then used on the test set.

Using the aforementioned separation into training and test set, the **AdaBoost**, after $T = 100$ iterations, returns the following result, illustrated in a confusion matrix:

Table 4.3: AdaBoost test set

		Actual class		
	class-labels	no	yes	total
Predicted class	no	11659	885	12544
		97.05%	57.06%	92.48%
	yes	354	666	1020
		2.95%	42.94%	7.52%
	total	12013	1551	12325
		88.57%	11.43%	90.87%

The *misclassification rate* of the training set[2] is 8.99% (2845 out of 31647) and using Table 4.3, the misclassification rate of the test set is 9.13% (1239 out of 13564). Here the *misclassification rate*[3] of the test result refers to the percentage of incorrectly classified observations: With 885 incorrectly labeled as “no” and 354 as “yes”, there were 1239 false labels out of 13564 predictions. As this difference between training and test error is very small, overfitting is not an issue in this first simulation.

There are 354 predictions in which AdaBoost incorrectly assumed a client of the bank to be interested in a subscription, thus possibly alienating or mildly annoying someone. This is often referred to as a *false positive*. Conversely, in 885 cases interested costumers where labelled as “not interested” in a subscription, even though they would have been. Therefore, missing the chance of obtaining new costumers, which is arguably economically worse than annoying someone. Analogous to the previous connotation, this is called a *false negative*.

Of the 885+666 = 1551 possible subscribers, AdaBoost achieved a correct *“yes”*-classification of 42.94%, which can also be called the *true positive rate*. Conversely,

[2]Out of simplicity, to save space and to focus on the essential, the tables of the training results will be omitted.

[3]This term will be used interchangeably with test (training) error in this thesis.

354 were incorrectly labelled, but that only equates to a misclassification of 2.95% of the $11659 + 354 = 12013$ observations not interested in a subscription. In other words, the *true negative rate* is low with 2.95%. Therefore, the goal should be to increase the percentage of the correct *"yes"*-classification, while possibly keeping the wrongly classified *"yes"* observations low.

The next attempt, continuing with the same training and test set and the same number of iterations ($T = 100$), the boosting algorithm **RealBoost** has the following results:

Table 4.4: RealBoost test set

		Actual class		
	class-labels	no	yes	total
Predicted class	no	11641	878	12519
		96.90%	56.61%	92.30%
	yes	372	673	1045
		3.10%	43.39%	7.70%
	total	12013	1551	12314
		88.57%	11.43%	90.78%

Using Table 4.4, the misclassification rate of the test set is 9.22% (1250 out of 13564), while that of the training is 9.01% (2851 out of 31647). Once again quite similar. It is to note, that the test error is slightly lower than the training error. Which is not unheard of and can be used to rule out overfitting.

Comparing this with the results of AdaBoost, the RealBoost model achieves a missclassifcation rate slightly worse. However, the percentage of a correct *"yes"*-classification, which is 43.39%, is slightly better than AdaBoost's 42.94%. In other words, having predicted 7 more possible subscriptions compared to AdaBoost.

Lastly, the final attempt, makes use of **GentleBoost**, applied once more on the same training and test set and the same number of iterations ($T = 100$), with the following confusion table:

Table 4.5: GentleBoost test set

		Actual class		
	class-labels	no	yes	total
Predicted class	no	11735	1008	12743
		97.69%	64.99%	93.95%
	yes	278	543	821
		2.31%	35.01%	6.05%
	total	12013	1551	12314
		88.57%	11.43%	90.52%

In this case the misclassification rate of the training error is 9.63% (3049 out of 31647) and using Table 4.5 that of the test error is 9.48% (1286 out of 13564). Once again, a similar error, with a slightly smaller test error.

Comparing this with the results of AdaBoost, an increase of roughly 0.5%-points in both training and test error, can be seen. Although, only 278 were incorrectly labelled as *"yes"*, as opposed to 354 (AdaBoost), GentleBoost only achieved a correct *"yes"*-classification of 35.01%, whereas RealBoost had 43.39%. Hence, GentleBoost predicted 130 less possible subscriptions as compared to RealBoost.

Summarizing after this first simulation, it seems choosing either AdaBoost or RealBoost over GentleBoost is preferable. This is due to the lowest misclassification rate (both training and test), as well as the clearly better *"yes"*-classification percentage.

The result of only 43.39% (RealBoost) correct prediction of possible subscribers is rather disappointing. As was mentioned already, the bank's goal is primarily to increase the percentage of the correct *"yes"*-classification and only secondarily to keep the wrongly classified *"yes"* observations low, as this is approach is more economical. In other words, it is more crucial to rightly determine the *"yes"* class as opposed to the *"no"*. Therefore the theory of Section 4.1 will be applied, to assign different levels of importance to the class, through the usage of pre-sampling methods.

All four sampling methods, under-, naive-, over- and same-size-sampling will be used on the same training sample as before ($70-30$ separation and set.seed(1308)), and then given to AdaBoost to compare the different results.

The first experiment used *under-sampling*, where 3738 observations are drawn from the training samples with replacement for each class, thus resulting in a new training set with 7476 observations. It's $n_{min} = 3738$, since the training sample contained as many observations assigned to the *"yes"* class.

Table 4.6: AdaBoost - under-sample - test set

		Actual class		
	class-labels	no	yes	total
Predicted class	no	10165	207	10372
		84.62%	13.35%	76.47%
	yes	1848	1344	3192
		15.38%	86.65%	23.53%
	total	12013	1551	11509
		88.57%	11.43%	84.85%

Comparing this new Table 4.6, with the tables without any pre-sampling, Table 4.3 for example, one can already see a clear improvement concerning the correct classification of the second class.
Even though there is an increase in training and test error to 11.46% and 15.15% respectively, one cannot argue against the vast improvement of a correct "yes" classification rate of 86.65% compared to the previous best 42.94% of AdaBoost. Hence, the number of possible new subscriptions for the bank more than doubled.
On the downside though, the false positves now represent 15.38% compared to the 2.95% of AdaBoost without special consideration of sampling.

Opposed to the first case of under-sampling, only the procedure of the other three pre-sampling methods will be described but their results will only shown in comparison with the others in Table 4.7.

The second experiment uses the pre-sampling method *naive-sampling*, which just like before has 3738 observations for each class drawn from the training samples but this time without replacement. Therefore it also has a total of 7476 observations for the training set which is then used by the algorithm.
The third experiment is *over-sampling*, where 27909 observations are drawn from the training samples with replacement for each class, thus resulting in a larger new training set with 55818 observations. This time, $n_{max} = 27909$, as the training sample contained as many observations assigned to the majority *"no"* class.
Lastly, the fourth experiment of *same-size-sampling* uses 15824 observations, drawn from the training samples with replacement for each class, as a new training set. Since the size of the original training sample is $n = 31647$ and there are $k = 2$ classes, each class will have $\lceil 31647/2 \rceil = 15824$ observations.

The comparison for AdaBoost of not using one (none) and using one of the four pre-sampling method is shown in Table 4.7. It shows the misclassification rate of the training and test set, as well as the correct *"yes"* classification percentage of the test set.

Table 4.7: AdaBoost results with different pre sampling methods

	Sampling method				
	none	under	naive	over	same-size
Training error	8.99%	11.46%	13.12%	13.11%	12.75%
Test error	9.13%	15.15%	14.78%	14.66%	14.72%
"yes"-percentage	42.94%	86.65%	86.53%	86.53%	86.78%

This comparison of AdaBoost, being applied four times to the same training set with different pre sampling method, doesn't show big differences between the sampling methods (except not using one). Additionally it was only done once to a single training and test set. Thus, it is clear, that it is necessary to chose one of the four pre-sampling methods but it isn't sufficient enough to choose a specific sampling method. Furthermore, since neither Real- nor GentleBoost were considered, it is necessary to preform repeated simulations with all possible combinations to provide more conclusive evidence and use it to come to a better conclusion.

Therefore, the next step will be to compare the three algorithms in repeated (100 runs) simulations for each of the four sampling methods. In every single run, a new split of training and test set (70/30%), is re sampled according to one of the four sampling methods and then given to AdaBoost, RealBoost and GentleBoost, who then produce a boosting model with $T = 100$ iterations. Hence, there will be twelve different combinations of boosting algorithms (three) and pre-sampling method (four).

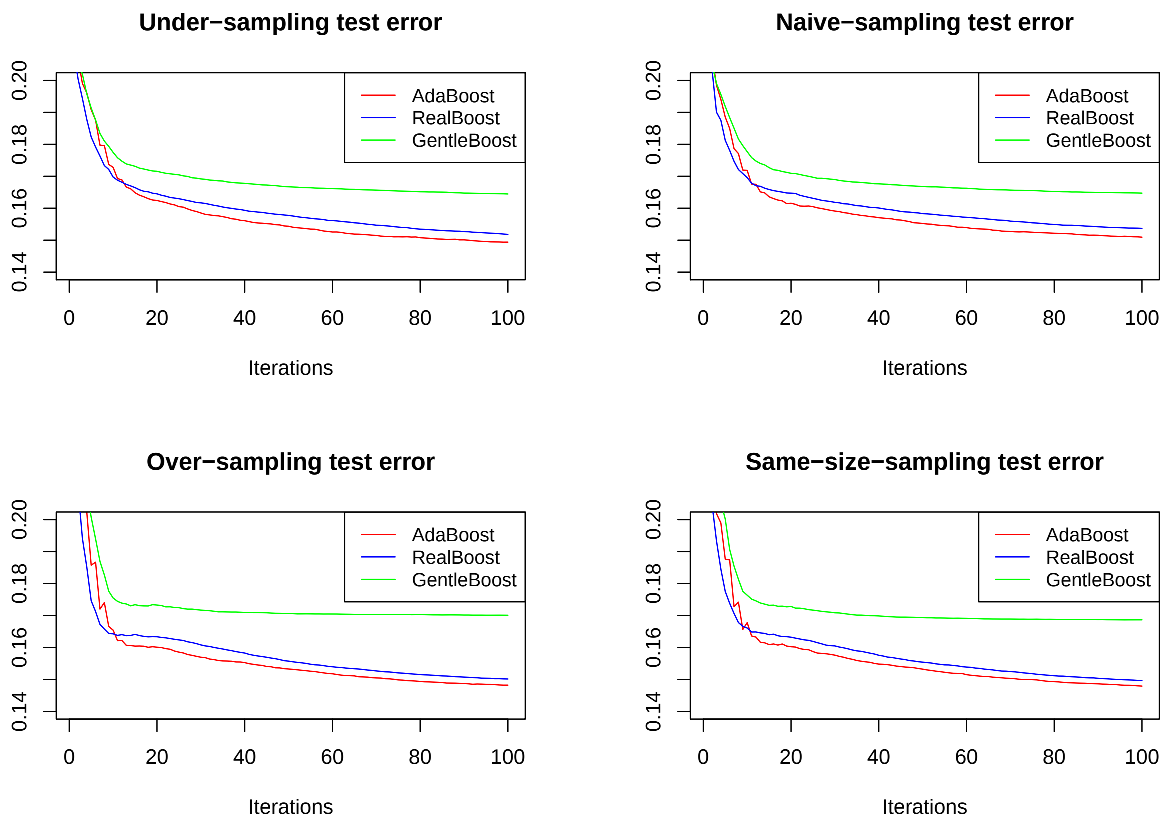


Figure 4.1: Test errors mean of the respective algorithms, as a function of the number of iterations $t = 1, \ldots, T$, for each sampling method, obtained from 100 different simulations.

In Figure 4.1 the result of the repeated simulation is shown for each sampling method in a separate line plot. In each plot the mean of the test error is shown as a function of the number of iterations ($t = 1, \ldots, 100$) for all three algorithms. The boosting algorithms are forthwith colour coded: AdaBoost as red, RealBoost as blue and GentleBoost as green. It can be seen, that the test error of GentleBoost is the worst by quite a margin and AdaBoost slightly outperforms RealBoost. This is true for every single pre-sampling method. Besides this, it also shows that GentleBoost does not improve as much as the other two algorithms with increasing iterations and it seems to stagnate after about 30 iterations. As a rule of thumb or lower bound, one could argue that for each sampling method and algorithm at least 20 iterations are necessary for each method to significantly reduce the test error.

Figure 4.2, which in the upper plot presents the results of the individual test errors in boxplots for each algorithm and sampling method. In the lower second plot the percentage of correct classification of the *"yes"* class is shown in a similar manner as above.
While AdaBoost outperforms the other two with respect to the test error, its *"yes"* classification performance is inferior to the other algorithms. Depending on the choice of pre sampling method either RealBoost or GentleBoost are shown to achieve higher curve of classification. Thus, a case for choosing either could be made. How-

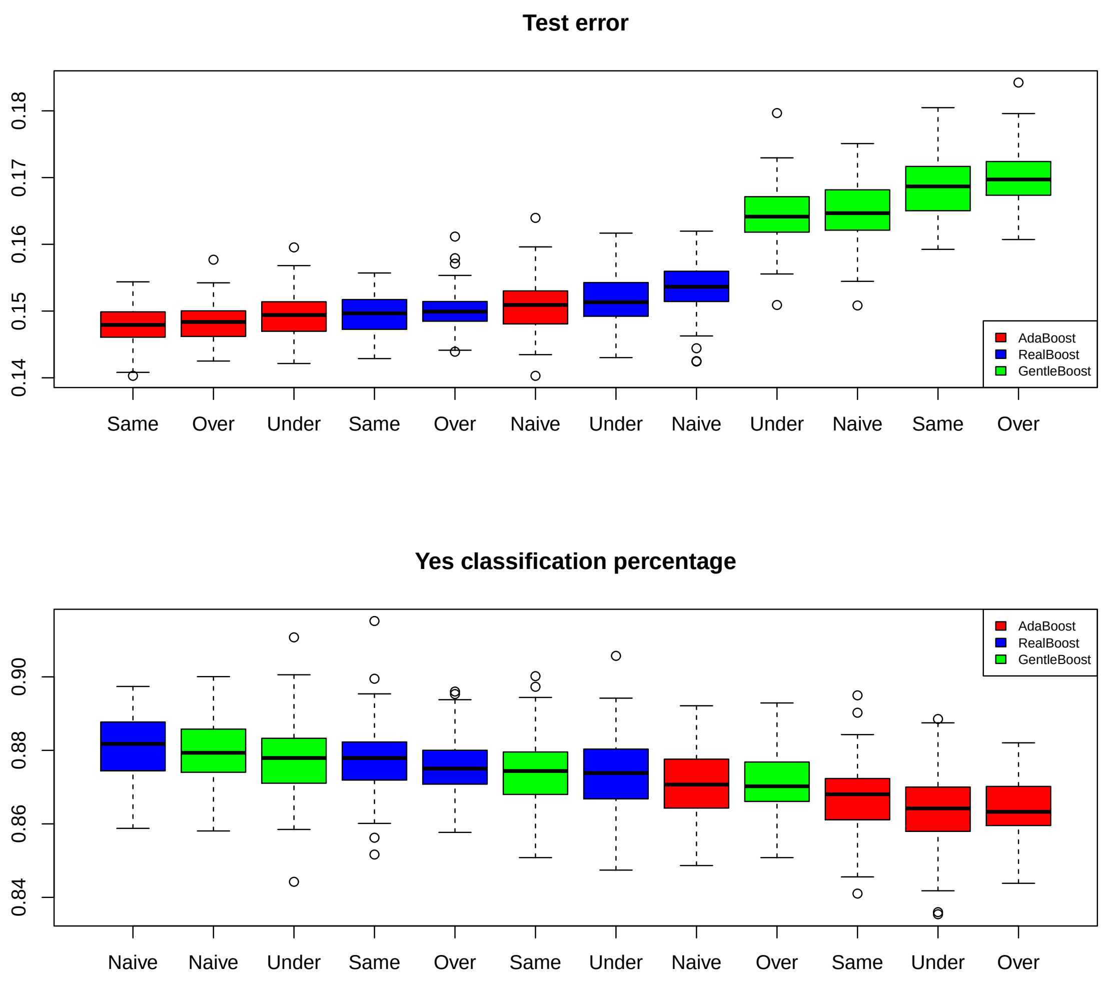


Figure 4.2: Boxplots of the test errors and the percentage of the correct classification of the *"yes"* class for each sampling method and each boosting algorithm, obtained from 100 different simulations.

ever, since the test error of GentleBoost is worse, RealBoost seems the best pragmatic choice out of the three.

The previous Figure 4.2 suggests some kind of pattern to govern the relationship between the different algorithms and sampling methods. This becomes more clear in Figure 4.3, where the medians of the percentage of correct "yes" classification is plotted against the medians of test errors.

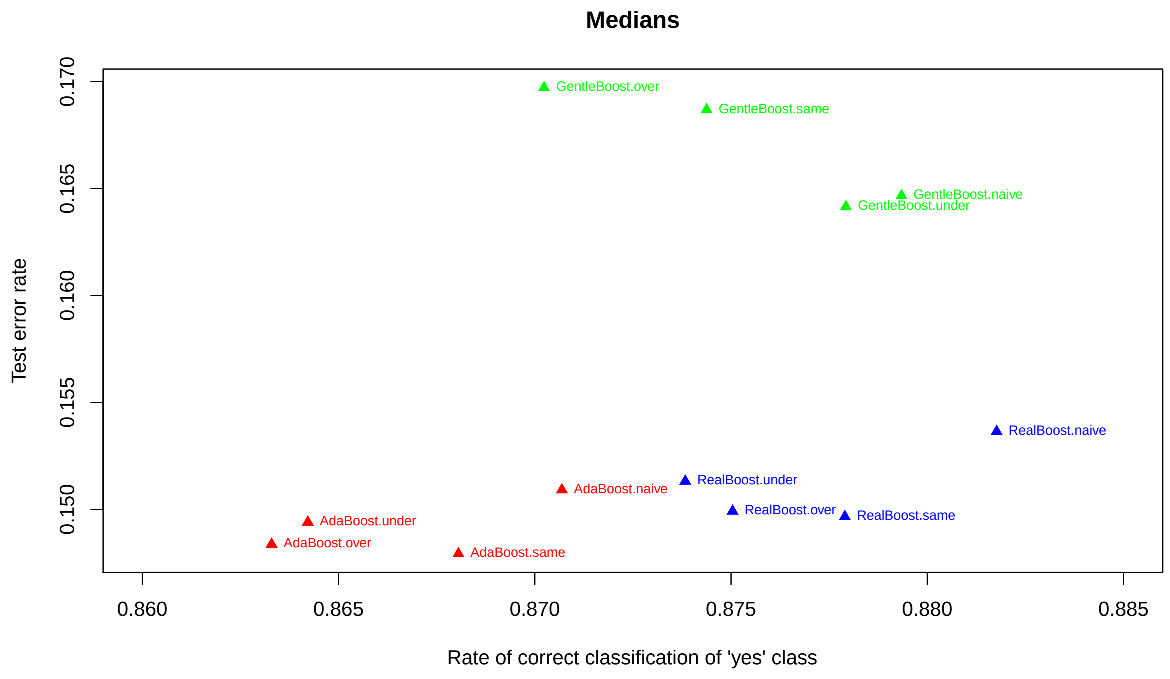


Figure 4.3: Comparison of the test errors and the percentage of the correct classification of the *"yes"* class for each sampling method and each boosting algorithm, obtained from 100 different simulations.

This plot can be used to identify the *"best"* and most appropriate sampling method. Since the primary goal of this classification is to obtain as many new subscriptions as possible, a higher percentage of correct "yes" prediction is imperative. The **naive**-sampling method appears best as it is furthest to the right (higher percentage) for each algorithm. While either **same-size** (AdaBoost and RealBoost) or **under** (GentleBoost) could be considered second depending on the choice of boosting algorithm.

In conclusion to the analysis done above on Figures 4.2 and 4.3, **RealBoost** pre sampled with the **naive**-method warrants to be the considered the best choice. Besides this combination the following three are worth considering as well: same-sized-sampled RealBoost, naive-sampled GentleBoost and under-sampled GentleBoost. Out of simplicity the four combinations will be subsequently referred to as naive-Realboost, same-size-Realboost, naive-GentleBoost and under-GentleBoost.

After having established the four most promising combinations of algorithms

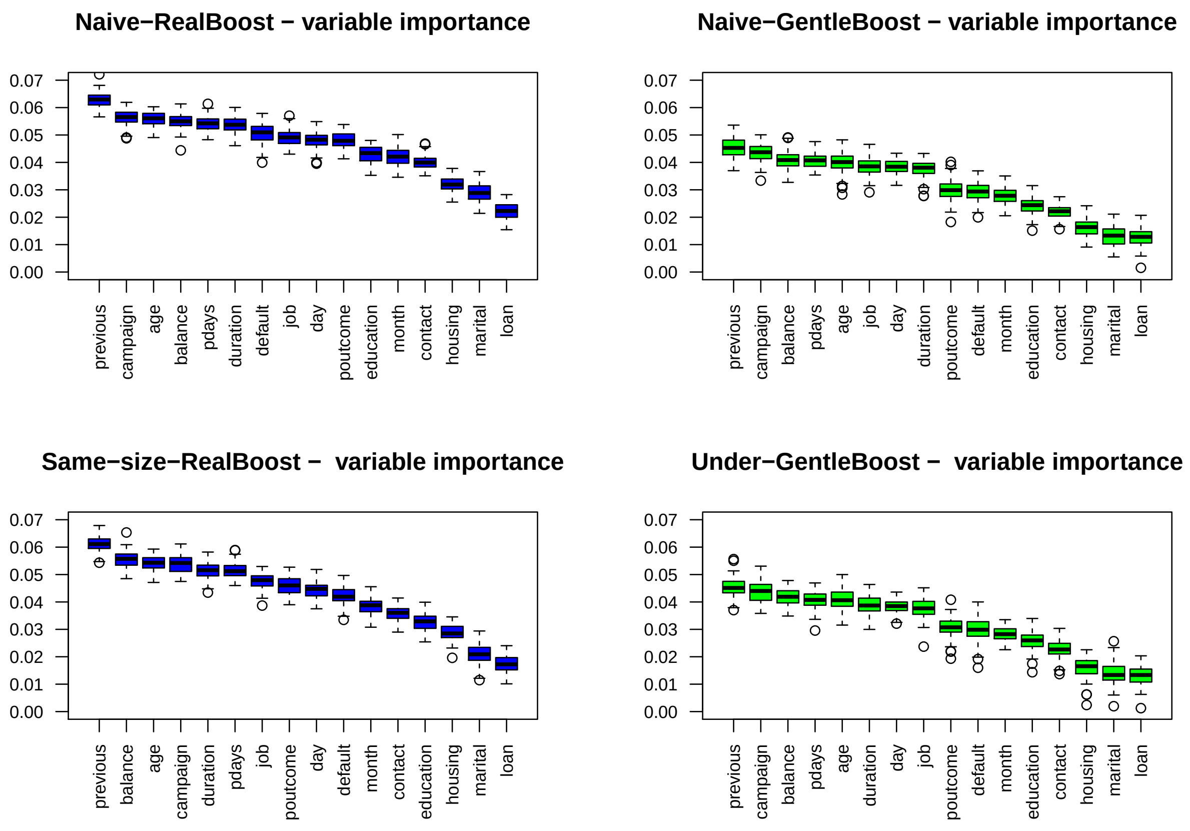
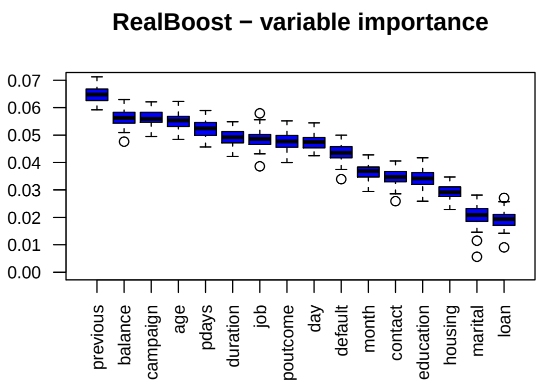


Figure 4.4: Boxplots of the variable importance scores of all variables for each afore mentioned combination of boosting algorithm and sampling method.

and sampling methods, it is interesting to illustrate the variable importance for each method. Figure 4.4 shows the box plots of scores for the variables for each method mentioned. The importance scores are defined by Hastie, Tibshirani and Friedman in their work *"The Elements of Statistical Learning" (see Hastie, Tibshirani, and Friedman 2008).* It is a rather standard measure for determining variable importance. The more often a variable is selected for boosting the more likely the variable contains useful information for classification.

In Figure 4.4 one can see that for all four methods, out of the 16 different variables, the variables "previous", "balance", "campaign", and "age" are always within the top five, albeit in slightly different sequence. However, a minor emphasis on "previous", describing the number of contacts performed *before* the campaign for this client (refer to Table 4.1 for an explanation of variables), can be seen as it is clearly a step above the others. This seems logical, as it might indicate if the person called had already been a ("previous") client and therefore might be more inclined to agree to another subscription. Similarly, "campaign" describes the number of contacts performed *during* the campaign. Thus, describing the attention given to a certain customer, to convince him of the possible benefits of a subscription. The importance of "balance", being the average yearly balance in Euro, is self explanatory as the contacted person should be able to afford the subscription. The importance of the variable "age", indicates the existence of certain age groups to be more or



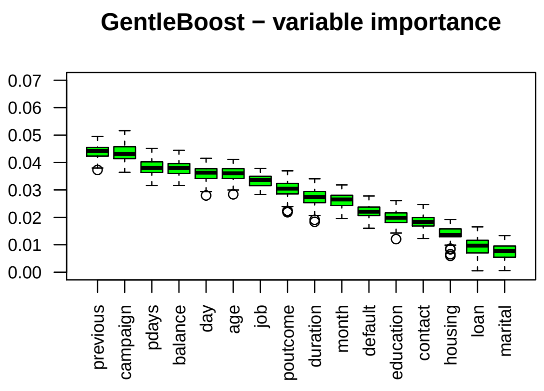


Figure 4.5: Boxplots of the variable importance scores of all variables for RealBoost and GentleBoost without any pre sampling method applied.

less interested in a new subscription.

Conversely, “loan”, “marital” and “housing” are always least relevant by a visible margin.
Thus, it doesn’t seem to matter much if a person has or hasn’t a personal (“loan”) or a housing loan(“housing”).
Lastly, the low importance of the variable “marital” shows that being either married, divorced/widowed or single doesn’t affect the targets willingness to buy a subscription.

It is to note, that since all four methods applied to data pre-sampled, these variable importance plots represent such models which placed a focus on the minority class (*“yes”*). An additional step is to compare these with the variable importance of a simulation of RealBoost and GentleBoost without any of the pre-sampling methods. The result of this (again 100 runs and $T = 100$ boosting iterations) can be seen in Figure 4.5.
However, there doesn’t seem to be that much of a difference between the two. This, can be seen more clearly in Table 4.8 where the ranking of each variable is displayed according to the method used.

4.2.3 Conclusions

Now to summarize the results of the three boosting algorithms which were applied to the imbalanced binary bank data set. With the analysis done in Section 4.2.2 it can be concluded, that RealBoost, sampled with the naive method, provided the best results in regards to the correct classification of the *“yes”* class and had a reasonably low test error. Three alternatives to this to consider are RealBoost with same-size or GentleBoost either naive-sampled or under-sampled. AdaBoost generally achieved the best test errors but had the lowest percentages of correct *“yes”* classification. Since the primary objective of this classification was to achieve a high rate for this percentage, the algorithm AdaBoost fails in comparison to the other two boosting algorithms.

The minority method naive-sampling proved to have the highest percentages of

Table 4.8: Ranking of variable importance

	Sampling - Algorithm Combination					
variable	Naive-Real	Same-Real	Naive-Gentle	Under-Gentle	None-Real	None-Gentle
age	3	3	5	5	4	6
job	8	7	6	8	7	7
marital	15	15	15	15	15	16
education	11	13	12	12	13	12
default	7	10	10	10	10	11
balance	4	2	3	3	2	4
housing	14	14	14	14	14	14
loan	16	16	16	16	16	15
contact	13	12	13	13	12	13
day	9	9	7	9	9	5
month	12	11	11	11	11	10
duration	6	5	8	6	6	9
campaign	2	4	2	2	3	2
pdays	5	6	4	4	5	3
previous	1	1	1	1	1	1

correct classification of the *"yes"* class of the four sample methods to overcome the issue of the imbalance in the present data set. It is quite interesting, that the majority method of over-sampling showed better results than under-sampling only in the case of RealBoost, whereas it was the worst for both GentleBoost and AdaBoost. In Section 4.3.2 it will be shown however, that the majority method will consistently lead to better results in multi-class simulations as compared to the minority method. Regardless of the existence of this special case of RealBoost, one could argue for the use of minority methods in binary boosting settings and majority methods in multi class boosting. However, the assumption of primarily using minority sampling on imbalanced binary data sets and the apparent anomaly of RealBoost would warrant further study outside of the scope of this thesis.
What can be said, is that same-size-sampling generally appears to be a safe choice of sampling method to deal with the issue of imbalanced data sets for all three boosting algorithms. As will also be shown in section 4.3.2, it does fairly well in a multi-class as-well. Hence, it is arguably a first go-to method for binary and multiclass boosting problems with imbalanced data sets.

The research done in Section 4.2.2 can be expanded upon by applying other variants of boosting algorithms and to compare the results. This can be done by using different boosting algorithms altogether, such as LogitBoost (refer to Section 3.1.2), or by evaluating the same three boosting algorithms with different weak learners than the classification trees that were used. For example, the tree's themselves could be pruned to produce different trees, the sparse parities (refer to Reyzin 2014) could be used or any other weak learner mentioned in Section 2.4.1.
Thereby possible combinations of boosting algorithm and weak learner might show promise. In other words, the goal is to determine what boosting algorithm works well with which weak learner, in the case of imbalanced binary data sets. This train of thought could be continued by applying this procedure to a variety of categorized

(size, number of variables, degree of imbalance, etc.) data sets. The final objective of this, is to be able to provide a recommendation of a boosting algorithm with a certain weak learner, depending on the properties of the data set and the focus of the classification at hand.

4.3 Multiclass Experiment

Just like the previous Section 4.2, this section will show the application of boosting on an imbalanced data set but unlike before it will contain multiple responses, i.e. it will be a multiclass experiment. First a new data set will be introduced, which has a response variable with elven different possible outcomes ($k = 11$), then two multi-class boosting algorithms will be applied to estimate a classification models.

The two boosting algorithms chosen for the multiclass experiment are: **AdaBoost.M1** and **SAMME**, which were described in Section 3.2. These algorithms will be applied to the data set and their results compared.

As was mentioned, AdaBoost.M1 is Freund and Schapire's first expansion of their original AdaBoost algorithm to the multi-class problem (see Freund and Schapire 1996a). Another later extension to the AdaBoost came with the SAMME algorithm (see Zhu et al. 2009), which just like AdaBoost.M1 reduces to AdaBoost in a two-class problem setting. Compared to AdaBoost.M1 it puts more weights on misclassified observations and thus, combines the weak classifiers differently (refer to Section 3.2.3).

The weak learners will once more be classification or regression trees (**CART**, refer to Section 2.4.1), as they already were used in the binary simulation of Section 4.2. The simulations of both algorithms AdaBoost.M1 and SAMME will be done using the **R**-function"boosting", which is provided by the **R**-package *adabag* (see Alfaro, Gamez, and Garcia 2013).

4.3.1 Meteorite Data

The meteorite data set originates from the CoMeCs Project 2017 (see CoMeCS-Project 2017), where it was referred to as *"comecs data set"*. It was provided by Brandstätter, Ferrière, and Koeberl from the Natural History Museum (Vienna, Austria). The samples were prepared by Engrand from the Centre de Sciences Nucléaires et de Sciences de la Matière (Orsay, France). Helchenbach from the Max Planck Institute for Solar System Research (Göttigen, Germany) took the **TOF-SIMS**[4] measurements. A formatted data set was provided by Anna Sofia Kircher (see Kircher 2016).

The dataset contains 1035 observations of 297 variables all of which are numerical containing normed spectra, mass numbers[5]. There are no mass numbers for

[4]Time-of-Flight Secondary Ion Mass Spectrometry (TOF-SIMS) is a surface-sensitive analytical technique that focuses a pulsed beam of primary ions onto a sample surface to remove molecules from its atomic monolayers (secondary ions). These particles are then accelerated into a "flight tube" and their mass is determined by measuring the exact time at which they reach the ctor detector (mass spectrum). Refer to Kircher 2016 for more detailed information.

[5]The mass number, nucleon number or also called atomic mass, is the total number of nuceleons (all protons and neutrons) in an atomic nucleus, which is different for each isotope of a chemical element. It is to note that, the mass number is not the same as the atomic number, which denotes the number of protons in a nucleus, and thus uniquely identifies an element.

$m23, m115$ and $m197$ as these spectra were removed for chemical reasons, therefore only 297 of 300 inorganic mass bins were considered.

The classes are made up by ten different meteorites[6] and the eleventh class is gold, referred to as "*substrate*" in the data set. Gold is added, because as an element its spectral composition is distinctive and therefore should be distinguishable from the others. A number of corns from each meteorite (or a gold corn) is placed on four different gold plates (presented as the *target* variable), each containing multiple corns from at least two meteorites. The targets with the corns are then used for **TOF-SIMS**-measurements resulting in the observations of mass numbers.
The spectra of the corns and the target are measured along a grid (see Figure 4.6). As the corns are smaller than the mesh size of the grid, it is a priori not clear whether the measurements are from the gold-plate or from the corn of the meteorite itself. In other words, it was necessary to perform a classification to determine if an observation was from said corn/meteorite or from the target and therefore belongs to the class substrate.

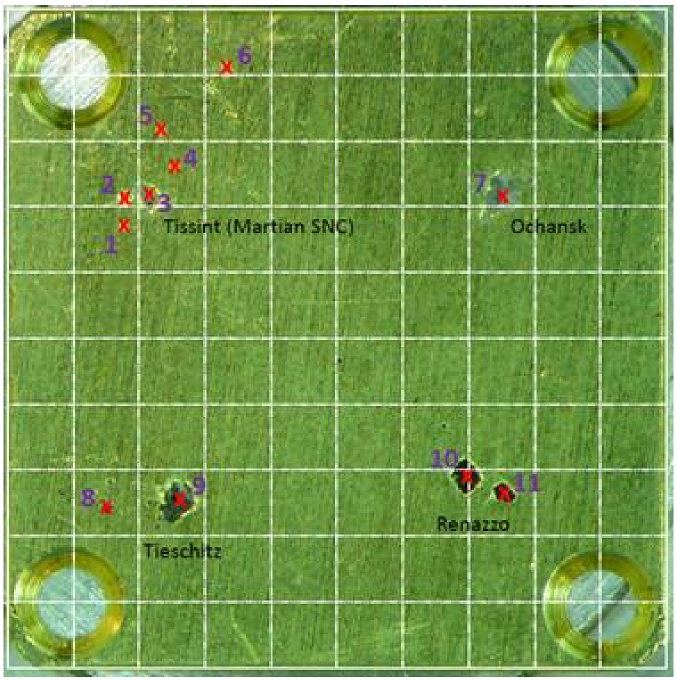


Figure 4.6: Image of Target 4E1 and eleven corns belonging to different meteorites placed upon it, taken from CoMeCS-Project 2017.

The data is imbalanced on different levels, due to the fact that not only does every meteorite have a different number of observations, which are unequally distributed on a different number of corns but these are also placed differently on a number of gold-plates. The imbalance of the data set is illustrated in Table 4.9.
As can be seen in Table 4.9, the data is imbalanced as *"substrate"* has almost nine times as many observations as *"tieschitz"*. Furthermore, another imbalance can be seen, as some meteorites such as *"renazzo"* only have two corns and one

[6]For example, *Tamdakht* was a meteorite which fell down on earth on the 20th of December in 2008 in Marokko, named after the village nearby.

target/gold-plate, while the other meteorites *"pultusk"* and *"tamdakht"* have elven and nine corns respectively.

Table 4.9: This table shows the number of total corns, the number of different gold plates they were placed on and the total observations for each of the elven classes.

	Nr. of corns	Nr. of targets	Total observations
allende	4	2	170
lance	3	2	77
mocs	6	2	66
murchison	2	2	85
ochansk	1	1	44
pultusk	11	2	78
renazzo	2	1	66
substrate		4	240
tamdakht	9	2	94
tieschitz	2	1	27
tissint	6	1	88

4.3.2 Multiclass Simulation

Applying the procedure used in Section 4.2.2, the meteorite dataset will be split (set.seed(1308)) into a training set, consisting of 70% of the observations and into a test set, made out of the remaining 30% (which results in 724 training-observations and 311 test-observations). Each of the boosting algorithms (AdaBoost.M1 and SAMME) is given the same training dataset to model a final hypothesis, which is then used on the test set.

Using the aforementioned separation into training and test set (without any pre-sampling methods applied), the **AdaBoost.M1**, after $T = 100$ iterations, returns the following result, illustrated with the help of a confusion matrix:

Table 4.10: AdaBoost.M1 test set

		Actual class										
	class-label	allende	lance	mocs	murchison	ochansk	pultusk	renazzo	substrate	tamdakht	tieschitz	tissint
Predicted class	allende	47	6	1	2	0	2	0	0	1	0	0
	lance	0	12	0	0	0	0	0	0	0	0	0
	mocs	1	2	13	1	0	0	0	0	0	0	0
	murchison	0	0	0	21	0	0	0	0	0	0	0
	ochansk	0	0	0	0	10	0	0	0	0	1	1
	pultusk	0	0	0	0	0	18	0	0	2	0	0
	renazzo	0	0	0	0	0	0	24	0	0	0	0
	substrate	0	0	0	0	0	0	0	91	0	0	1
	tamdakht	0	0	0	0	0	0	0	0	27	0	0
	tieschitz	0	0	0	0	0	0	0	0	0	4	0
	tissint	0	0	0	0	1	0	0	0	0	0	22
	ccp	97.9%	60.0%	92.9%	87.5%	90.9%	90.0%	100%	100%	90.0%	80.0%	91.7%

The *misclassification rate* of the training set is 0% (0 out of 724), meaning that there were no misclassifications within the training set. From Table 4.10 the misclassification rate of the test set can be seen as 7.1% (22 out of 311). As the training set was predicted *"perfectly"* and the test error is rather low considering the imbalance, one could consider this a good result.

However, the *correct class percentages*[7], in short **ccp**, vary substantially depending on the different classes. While *substrate* and *renazzo* achieve a perfect result of 100%, *lance* only has a ccp of 60%. However, the reason for the low ccp of *lance* does not lie within the imbalance of the data set, since it had 57 observations in the training set, while meteorites *tieschitz* and *ochansk* only had 22 and 33 respectively and achieved better results. It is to note, that *substrate* (i.e. the observations of the gold plates) was correctly identified in each instance of the test set. Hence, it stands to reason that its distinctive properties of its spectral composition held true to be easily distinguishable from the others.

Proceeding with the second experiment using the same training and test set and the same number of iterations ($T = 100$) and applying the the boosting algorithm **SAMME** leads to the following results.

Table 4.11: SAMME test set

		Actual class										
	class-label	allende	lance	mocs	murchison	ochansk	pultusk	renazzo	substrate	tamdakht	tieschitz	tissint
Predicted class	allende	47	5	1	2	0	4	0	0	1	0	0
	lance	0	12	0	0	0	0	0	0	0	0	0
	mocs	1	1	13	1	0	0	0	0	0	0	0
	murchison	0	1	0	21	0	0	0	0	0	0	0
	ochansk	0	0	0	0	11	0	0	0	0	1	1
	pultusk	0	1	0	0	0	16	0	0	1	0	0
	renazzo	0	0	0	0	0	0	24	0	0	0	0
	substrate	0	0	0	0	0	0	0	91	0	0	1
	tamdakht	0	0	0	0	0	0	0	0	28	0	0
	tieschitz	0	0	0	0	0	0	0	0	0	4	0
	tissint	0	0	0	0	0	0	0	0	0	0	22
	ccp	97.9%	60.0%	92.9%	87.5%	100%	80.0%	100%	100%	93.3%	80.0%	91.7%

Using Table 4.11, the misclassification rate of the test set is 7.1% (22 out of 311), while that of the training is 0% (0 out of 724). These errors are equal to the previous result of AdaBoost.M1. However, their results still differ, in regard to where the misclassifications occurred.

Comparing the two results of Table 4.11 and 4.10 the following can be observed. The two algorithms seem to have the biggest issue in predicting the *lance* meteorite, as in both instances the ccp of 60% is the lowest of all other classes. Therefore, the discrepancy of varying ccp's can be seen here as well. Although it had the lowest number of observations in the test set, *ochansk* was correctly classified everytime by SAMME. Just as before with AdaBoost.M1, the classes of *substrate* and *renazzo* were correctly identified for every observation of the test set.

As the two boosting algorithms performed almost identically in these first two simulation, a decision as to which algorithm is better can not be made without further investigation. Furthermore, the issue of the imbalance of the data set was not yet directly addressed with any pre-sampling methods. The idea of Section 4.1

[7]The correct class percentages represent the percentages of a certain class, which were correctly labelled. In the previous Section 4.2.2 this was presented as the percentage of correct *"yes"* classification

will be applied once more and different levels of importance will be assigned to the classes, through the usage of different sampling methods.

The four mentioned sampling methods, under-, naive-, over- and same-size-sampling will be used on the same training sample as before ($70-30$ separation and set.seed(1308)), and then given to AdaBoost.M1 and SAMME to compare the different results.

The first experiment used *under-sampling*, where 22 observations are drawn from the training samples with replacement for each class, thus resulting in a new training set with 242 observations. The class *tieschitz* had the lowest number of observations with $n_{min} = 22$ in the training set, which determined the number of samples drawn for each class while under-sampling.

Table 4.12: AdaBoost.M1 - under-sample - test set

		Actual class										
	class-label	allende	lance	mocs	murchison	ochansk	pultusk	renazzo	substrate	tamdakht	tieschitz	tissint
Predicted class	allende	41	0	1	2	0	3	0	0	1	0	0
	lance	4	18	0	0	0	0	1	0	1	0	0
	mocs	1	1	11	0	0	1	0	0	0	0	0
	murchison	0	0	0	21	0	2	0	0	0	0	0
	ochansk	0	0	0	0	10	0	0	0	1	0	2
	pultusk	0	1	2	1	0	13	0	0	1	0	0
	renazzo	0	0	0	0	0	0	23	0	0	1	0
	substrate	1	0	0	0	1	0	0	91	0	0	2
	tamdakht	0	0	0	0	0	1	0	0	26	0	0
	tieschitz	2	0	0	0	0	0	0	0	0	4	0
	tissint	0	0	0	0	0	0	0	0	0	0	20
	ccp	85.4%	90.0%	78.6%	87.5%	90.9%	65.0%	95.8%	100%	86.7%	80.0%	83.3%

Comparing this new Table 4.12 with the Table 4.10 without any pre-sampling, the previous imbalance of ccp seems to have been reduced a bit, as the class of *lance* went up from 60% to 78.6%. However, at the same time the ccp of *pultusk* went down from 80% to 65%.

As expected the error rates went up. The training and test error are 12.7% and 10.6% respectively, thus drastically increasing for the training set and slightly so for the test set. These two rates are now closer to each other and contrary to before (not using pre-sampling) one could arguably rule out overfitting.

The results of under-sampling with SAMME can be seen in Table 4.13.

Table 4.13: SAMME - under-sample - test set

		Actual class										
	class-label	allende	lance	mocs	murchison	ochansk	pultusk	renazzo	substrate	tamdakht	tieschitz	tissint
Predicted class	allende	42	2	4	3	0	3	0	0	1	0	0
	lance	3	15	0	0	0	1	1	0	1	0	0
	mocs	1	0	9	0	0	1	0	0	2	0	0
	murchison	0	0	0	21	0	2	0	0	0	0	0
	ochansk	0	0	0	0	9	0	0	0	1	0	4
	pultusk	0	2	1	0	0	13	0	0	2	0	0
	renazzo	0	0	0	0	1	0	24	0	0	1	1
	substrate	1	0	0	0	1	0	0	91	0	0	1
	tamdakht	0	0	0	0	0	1	0	0	23	0	0
	tieschitz	2	2	0	0	0	0	0	0	0	4	0
	tissint	0	0	0	0	0	0	0	0	0	0	18
	ccp	87.5%	75.0%	64.3%	87.5%	81.8%	65.0%	100%	100%	76.7%	80.0%	75.0%

Just like AdaBoost.M1 the ccp's seem to be more equally distributed. However, the training and test errors are slightly worse, with 15.3% and 13.6% respectively. It is worth mentioning, that in both under-sampling cases (AdaBoost.M1 and SAMME) the class *substrate* still was correctly classified in each instance. Thus, the special spectral composition of substrate (gold) still remains, even when pre-sampling is applied.

The remaining three sampling methods, had the following numbers of observations for each class in the training set for their respective experiment. The second experiment uses the pre-sampling method *naive-sampling*, which just like before has 22 observations for each class drawn from the training samples but this time without replacement. The third experiment with *over-sampling*, where 149 (*substrate* had 149 observations in the training set) observations are drawn from the training samples with replacement for each class, thus resulting in a larger new training set with 1639 observations. Lastly, the fourth experiment of *same-size-sampling* uses 66 observations, drawn from the training samples with replacement for each class, as a new training set. Since the size of the original training sample is $n = 724$ and there are $k = 11$ classes, each class will have $\lceil 724/11 \rceil = 66$ observations.

The comparison for AdaBoost.M1 and SAMME of not using a sampling method and using one of the four pre-sampling methods is shown in Table 4.14. The table shows the ccp of the test set for each class, the mean of the ccp and the respective test error.

Table 4.14: Test results of all boosting algorithms (with $T = 100$ iterations) with different pre-sampling methods, with the same training and test set set.seed(1308).

		Correct classification of individual class												
	sampl	allende	lance	mocs	murchison	ochansk	pultusk	renazzo	substrate	tamdakht	tieschitz	tissint	mean	error rate
M1	none	97.9%	60.0%	92.9%	87.5%	90.9%	90.0%	100%	100%	90.0%	80.0%	91.7%	89.2%	0.071
	under	85.4%	90.0%	78.6%	87.5%	90.9%	65.0%	95.8%	100%	86.7%	80.0%	83.3%	85.7%	0.106
	naive	89.6%	75.0%	57.1%	87.5%	90.9%	80.0%	79.2%	100%	86.7%	80.0%	91.7%	83.4%	0.116
	same	97.9%	60.0%	92.9%	87.5%	90.9%	90.0%	100%	100%	90.0%	80.0%	91.7%	89.2%	0.071
	over	93.8%	70.0%	100%	91.7%	100%	85.0%	95.8%	100%	90.0%	80.0%	91.7%	90.7%	0.068
SAMME	none	97.9%	60.0%	92.9%	87.5%	100%	80.0%	100%	100%	93.3%	80.0%	91.7%	89.4%	0.071
	under	83.3%	85.0%	78.6%	91.7%	90.9%	45.0%	100%	100%	80.0%	80.0%	70.8%	82.3%	0.135
	naive	87.5%	90.0%	71.4%	87.5%	90.9%	55.0%	95.8%	100%	80.0%	80.0%	91.7%	84.5%	0.113
	same	97.9%	60.0%	92.9%	87.5%	100%	80.0%	100%	100%	93.3%	80.0%	91.7%	89.4%	0.071
	over	95.8%	55.0%	92.9%	91.7%	90.9%	85.0%	100%	100%	90.0%	80.0%	95.8%	88.8%	0.074

Although a decision between AdaBoost.M1 and SAMME cannot be made as the differences between their respective correct classification rates don't look to be significant, a certain structure of the sampling methods can be seen. Under- and naive-sampling perform visibly worse than the other two sampling methods and arguably worse than "none"(not pre-sampling at all) as well. This discrepancy is shown in the error rates and in the means of the ccp's, as the under and naive have about five percent points less average correct classification.

The results from Table 4.14 only show the results of a single training and test set, which isn't sufficient enough to choose a specific sampling method and algorithm. It is therefore necessary to preform repeated simulations analogous to the binary simulations of Section 4.2.
Thus, the next step will be to compare AdaBoost.M1 and SAMME in repeated

(100 runs) simulations with and without one of the four sampling methods. Each run will use a new split of training and test set (70/30%), which is consequently re sampled (or not) according to one of the four sampling methods and then given to AdaBoost.M1 and SAMME, to produce a boosting model with $T = 100$ iterations. The result of these repeated simulations will be ten different combinations of boosting algorithm (two) and the option of pre sampling (four methods) or no pre sampling.

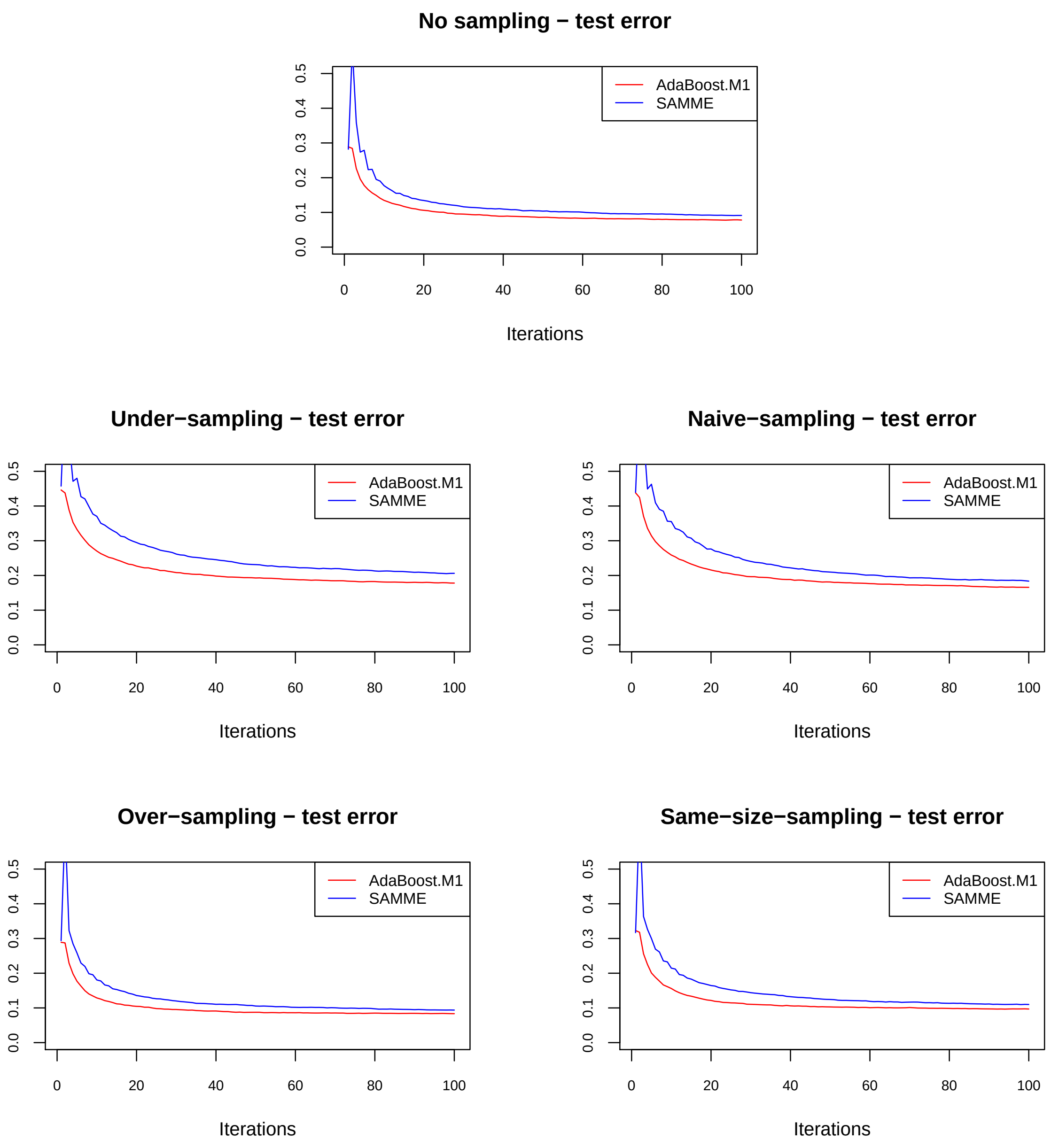


Figure 4.7: Test errors means of the respective algorithms, as a function of the number of iterations $t = 1, \ldots, T$, for each sampling method, obtained from 100 different simulations.

The result of the repeated simulation is shown for each sampling method in a separate line plot in Figure 4.7. In each plot the mean of the test error is shown for

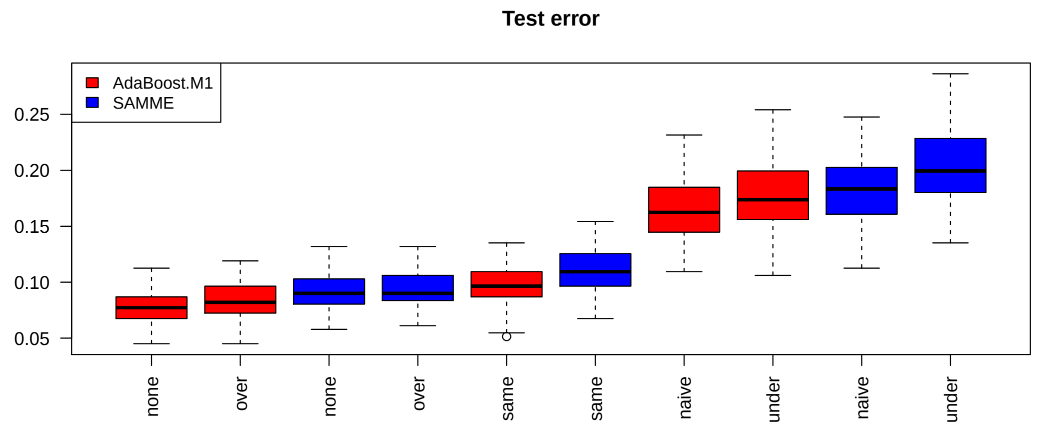


Figure 4.8: Boxplots of the test errors for each sampling method and boosting algorithm, obtained from 100 different simulations.

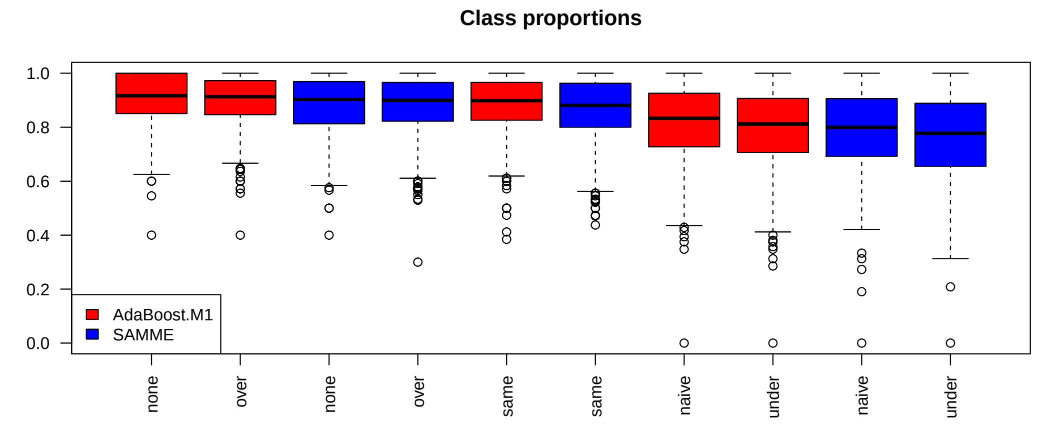


Figure 4.9: Boxplots of the class percentages for each sampling method and boosting algorithm, obtained from 100 different simulations.

AdaBoost.M1 and SAMME as a function of the number of iterations ($t = 1, \ldots, 100$). To ensure coherence, the two boosting algorithms are forthwith colour coded: AdaBoost.M1 as red and SAMME as blue.
In every instance/plot AdaBoost.M1 has a lower test error as compared to SAMME. Furthermore, the aforementioned three methods of not sampling (referred to as *"none"* forthwith), over and same maintain their lower error rates compared to under- or naive-sampling. Similiar to the binary plot of Figure 4.1 in Section 4.2.2, it can be seen, that SAMME (as RealBoost before) has an unstable start and needs at least 10 iterations before stabilizing. Further, one could argue that, the rule of thumb of a lower bound for the number of boosting iterations is $T = 20$. These number of iterations are necessary for each sampling method and boosting algorithm, to significantly reduce the test error.

Figure 4.8 presents the results of the individual test errors as boxplots for the two boosting algorithms at the final iteration ($t = T = 100$) and each sampling method. It is no surprise, that the test error is lowest if no sampling method ("none") was used, as there was no extra emphasis of the minority classes. The sampling methods applied can be separated in two groups. One comprising none, over and same the other under and naive.

In Figure 4.9 the boxplots of the combined ccp for each combination of boosting algorithm at the final iteration ($t = T = 100$) and sampling method is shown. From Figure 4.8 and 4.9 the dominance of group one over group two can be inferred. In both instances, test error and class percentages, AdaBoost.M1 outperforms SAMME by a small margin for every sampling method. Therefore it seems, that AdaBoost.M1 applied with a sampling method from group one is preferable.

This relationship between the different algorithms and sampling methods is illustrated in Figure 4.10, where the medians of the correct class percentages is plotted against the medians of test errors.

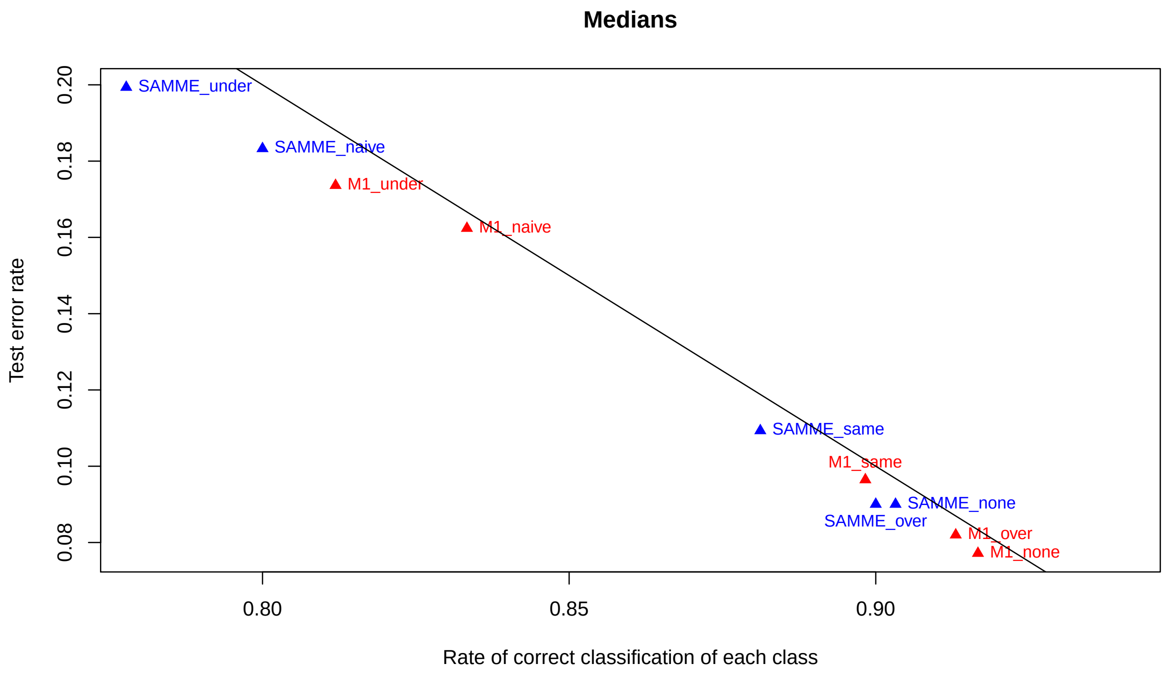


Figure 4.10: Comparison of the test errors and the percentages of the correct classification of the individual classes for each sampling method and each boosting algorithm, obtained from 100 different simulations.

This plot can be used to identify the *"best"* in terms of ccp and test error and thus the most appropriate sampling method. In ranking order, the sampling methods of **none**, **over** and **same** clearly outperform the other two(*under* and *naive*). Hence, providing further evidence to the previous conclusions made. In addition, the AdaBoost.M1 algorithm appears to provide a slightly but visibly better classification in comparison with SAMME.

To summarize the analysis done using Figures 4.8, 4.9 and 4.2, the **AdaBoost.M1** either **not sampled** at all or **over**-sampled, warrants to be considered the best possible combination. The results of the SAMME algorithm, with the same sampling methods applied, is only slightly worse and thus worth consideration as well. The sampling method *same* with either boosting algorithm, though worse than the other two(*over* and *none*) will also be examined as a means of comparison. Out of simplicity the six combinations will be subsequently referred to as *none-AdaBoost.M1, over-AdaBoost.M1, same-AdaBoost.M1, none-SAMME, over-SAMME and same-SAMME.*

It is interesting to note, that this order of sampling methods is almost the complete opposite of was observed in the simulations of the binary problem of Section 4.2.2. There, the *naive*-sampling method appeared best for each algorithm and either *under* or *same-size* were considered second depending on the choice of boosting algorithm.

The correct class percentages of none-AdaBoost.M1, over-AdaBoost.M1, same-AdaBoost.M1, none-SAMME, over-SAMME and same-SAMME are shown in Figure 4.11. Viewing the boxplot of the ccp's of the six combinations, no substantial differences can be seen.
The class of *substrate*, which represents observations made from the gold plate, was with a few exception classified correctly for each observation and chosen method. Thus, confirming that its spectral composition as an element is distinctive and therefore it's distinguishable from the meteorites. Similiarly, the algorithms appear to be effective with classifying the meteorites *renazzo* and *tissint*, as their ccp is quite high. Arguably the most difficult meteorite to classify is *pultusk*, as its median is lowest. These assumptions can be verified with Table 4.15, where the medians of the ccp's, their total means (of the medians) and the standard deviation (sd) of the individual ccp's is presented.

Table 4.15: Shows the medians of the correct classification percentage -ccp- of the six combinations of boosting algorithm and sampling method, for each of the elven classes, obtained from 100 different simulations.

		Standard Deviation of ccp's of individual class											Overall	
	Sampling	allende	lance	mocs	murchison	ochansk	pultusk	renazzo	substrate	tamdakht	tieschitz	tissint	mean	sd
M1	none	92.31%	85.71%	84.81%	89.83%	88.89%	83.60%	100%	100%	90.91%	87.50%	100%	91.23%	6.19
	over	90.70%	85.71%	85.45%	91.49%	88.56%	82.61%	100%	100%	90.00%	87.50%	96.30%	90.76%	5.81
	same	86.89%	83.94%	84.81%	91.30%	87.50%	80.00%	95.83%	100%	87.50%	87.50%	96.49%	89.25%	6.03
SAMME	none	92.94%	81.82%	83.33%	88.12%	83.33%	78.26%	98.33%	100%	89.09%	85.71%	100%	89.17%	7.66
	over	91.23%	84.21%	86.19%	88.68%	81.82%	77.53%	100%	100%	83.30%	87.50%	96.43%	89.17%	7.22
	same	86.54%	81.98%	80.95%	89.47%	81.82%	76.46%	96.23%	100%	85.19%	87.50%	96.43%	87.51%	7.41

The standard deviations of the ccp's of each class for every combination are shown in Table 4.16. In this table the standard deviation of the percentages (100%) and not of the percentage points (1.00) are calculated. Once more the means and standard deviation for each row (algorithm and sampling combination) are shown at the end.

With the two Tables 4.15 and 4.16 are clear decision can be made. Starting of with the choice of algorithm, **AdaBoost.M1** outperforms SAMME. It not only has higher average ccp's but they don't vary as much (sd) either. Only in a few

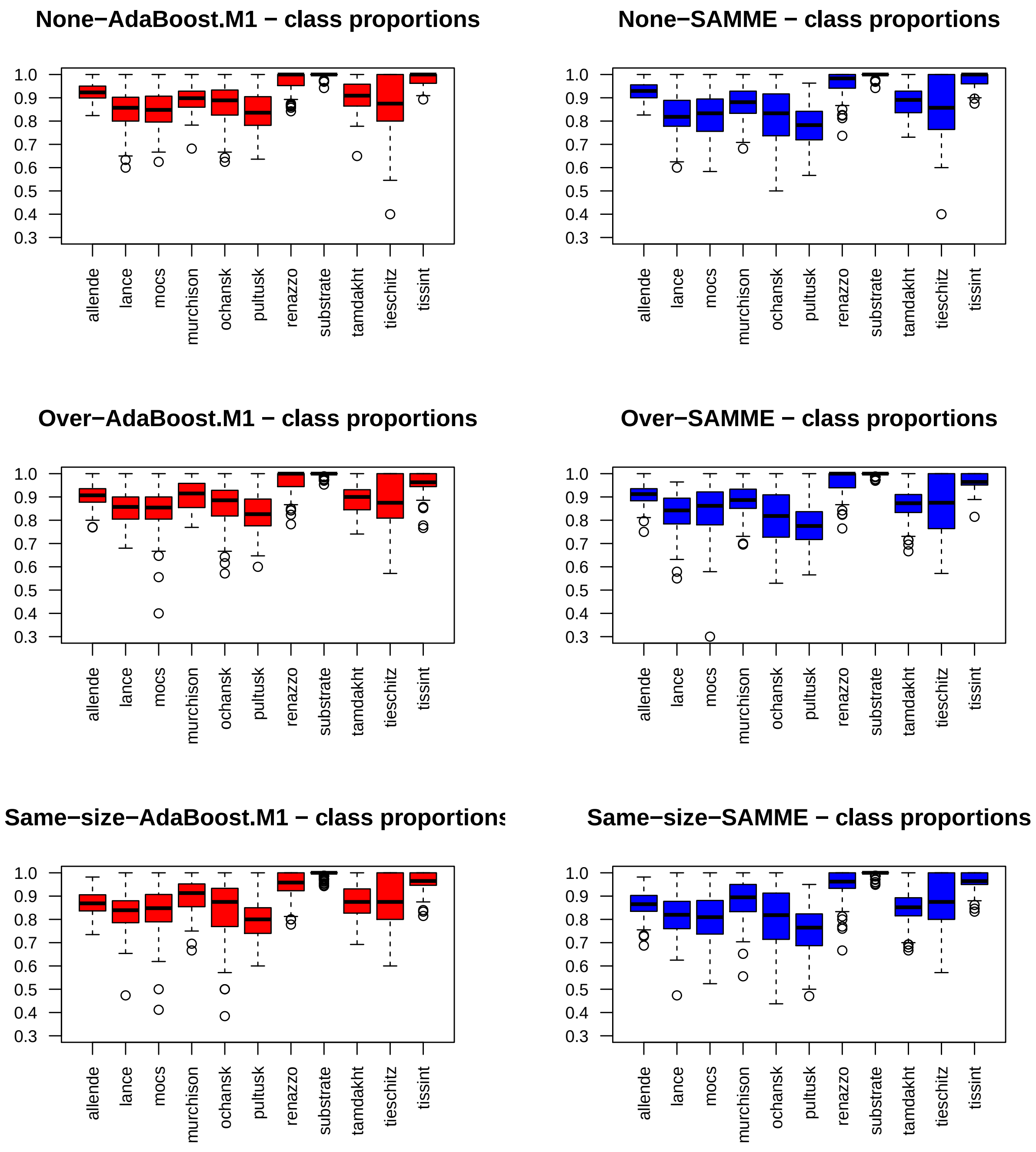


Figure 4.11: Boxplots of percentages of correct prediction of each class for the AdaBoost.M1 and SAMME algorithm.

cases, such as with class *tissint* and *substrate* in the “none” sampling scenario, does SAMME have slightly better results.
As for the choice of sampling method a case for either *none* or *over* can be made. While *none* produces better overall results, these vary more depending on the observed class. However, one could prefer not to sample at all, i.e. using the **none** method. The argument for this choice, is that even though the results vary, depending on the class *(*6.19 *vs* 5.81 *for the medians of Table 4.15 and* 3.2454 *vs* 2.9770 *for the sd of Table 4.16)*, this deviation is not significantly bigger than that of the

over-sampling method. This pay-off is arguably worth the slight increase of performance, as the mean of the ccp's is 91.23% (*none*) against that of 90.75% (*over*) and the mean of the standard deviations in each class is 6.3029 (*none*) opposed to 6.7375 of *over*. Thus, its not only more accurate in its classification but individual results do not vary as much either. Furthermore, one could argue that due to the re-weighting of boosting in each iteration to focus on the observations which were hard to classify, an additional step to combat the imbalance in the data set with pre-sampling is not necessary.

Table 4.16: Shows the standard deviation of the correct classification percentage - ccp- of the six combinations of boosting algorithms and sampling methods, for each of the elven classes, obtained from 100 different simulations.

		Standard Deviation of ccp's of individual class											Overall	
	Sampling	allende	lance	mocs	murchison	ochansk	pultusk	renazzo	substrate	tamdakht	tieschitz	tissint	mean	sd
M1	none	3.6641	7.9682	7.9751	5.8577	9.2390	8.2687	3.9437	0.8996	6.3461	12.1710	2.9993	6.3029	3.2454
	over	4.6254	7.4748	9.6293	6.5100	9.5949	8.5531	4.8876	0.9347	6.3662	11.2603	4.2765	6.7375	2.9770
	same	5.1965	8.0529	9.8796	6.5754	12.9269	8.6982	5.6354	1.2624	7.0302	10.6013	3.9543	7.2558	3.2734
SAMME	none	3.8022	8.8217	8.8633	6.7418	11.6718	8.8122	5.1929	0.8282	6.5134	12.3982	2.8981	6.9585	3.6019
	over	4.6247	8.1453	10.8458	6.8701	12.6412	9.0097	5.0233	0.8609	6.9360	11.5216	3.5076	7.2715	3.6201
	same	5.4454	8.9033	10.4545	7.7383	13.5219	10.0913	6.4194	1.0075	7.0748	10.7771	3.7667	7.7464	3.5429

After having established the most promising combination of algorithm and sampling method, being **AdaBoost.M1 without any sampling**, it is interesting to illustrate the variable importance. The **R**-package *adabag* (see **adabag**) quantifies the importance of the variables with consideration of the gain of the Gini index[8] given by a variable in a tree and the weight of this tree in the case of boosting.

In Figure 4.12 one can see the 15 out of 297 most important variables (spectra - mass numbers) for each of the four shown methods. Out of these, the most important variable is *m40* , which is followed exclusively by *m52* for every combination. These two are followed by *m27*, *m56* and *m58* in no particular order.

The most important variable *m40*, shown in Figure 4.13, could for example be used to distinguish the classes *murchison*, *ochansk*, *renazzo* and *tissint* from the remaining ones. Likewise *m52* adds to the classifcation of *ochansk* and *tissint* and *m68* to that of arguably *lance*. The spectras *m27* and especially *m56* can be used to differentiate the gold plate class *substrate* from the meteorites as can be seen in Figure 4.15.

Contrary to the important variables, for example *m284* (the least important variable for AdaBoost.M1 without sampling), hardly add anything to the classification. It can be seen in Figure 4.14, that the observations for each class are scattered and therefore no clear distinction can be made between them. The data set is quite noisy as this is the case for many variables.

[8]The Gini-index is a measure of impurity, as it measures how often a randomly chosen element from the set would be incorrectly labelled if it was randomly labelled according to the distribution of labels in the subset (see Kircher 2016).

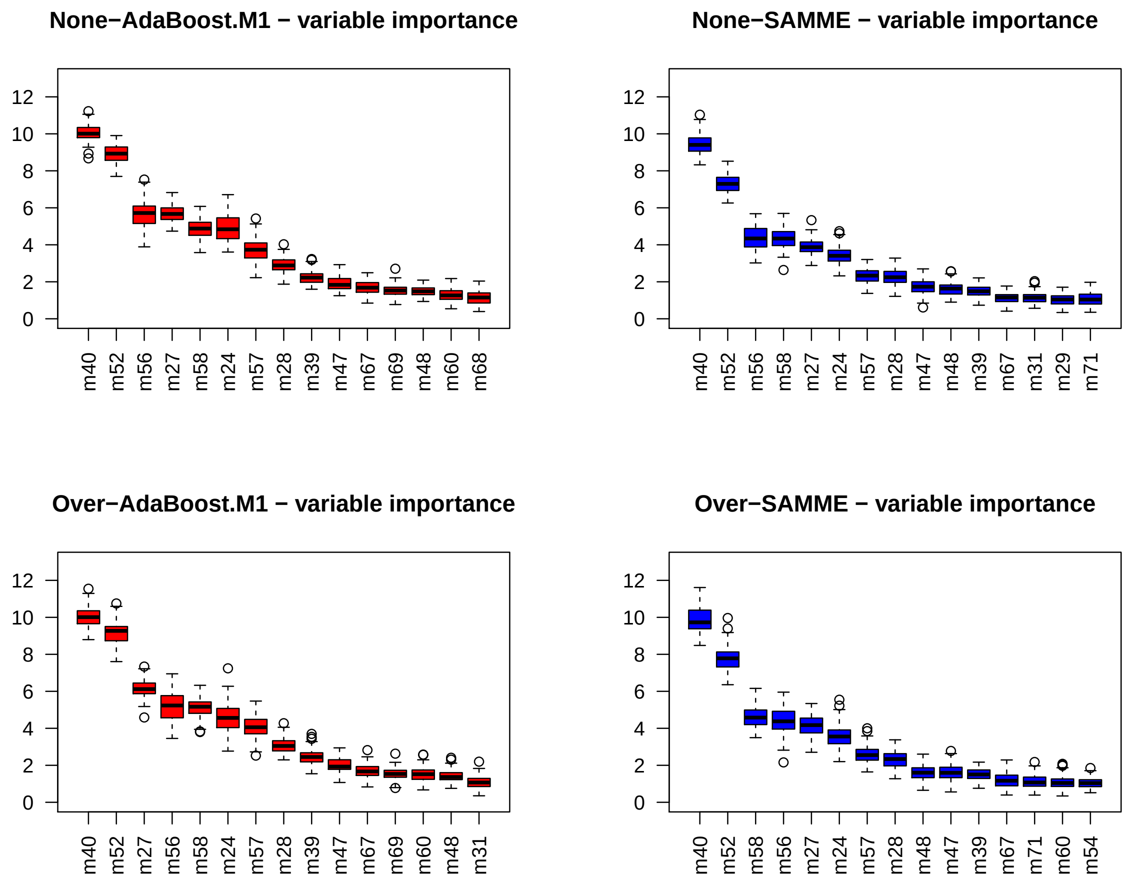


Figure 4.12: Variable Importance boxplots of AdaBoost.M1 and SAMME for the two relevant pre-sampling methods of the 15 most important variables.

4.3.3 Conclusions

The investigation of the performance of the two boosting algorithms, applied to the imbalanced multi-class data set *comecs* proved to be quite interesting. It was concluded in Section 4.3.2 that the AdaBoost.M1 algorithm, without any sampling method, led to the most favourable results concerning the test data. AdaBoost.M1 but with over-sampling the training set came in second. This can be interpreted such, that it was not necessary to use a sampling method to counter the imbalance, as "not sampling" was superior to over-sampling on this data set. It can be argued, that AdaBoost.M1's inbuilt re-weighting of hard to classify observations, was sufficient to overcome the issue of imbalance an no additional corrective pre sampling was necessary. Recalling the results from Section 4.2.2, this was not the case in the binary example, where the major focus was to increase the classification of the *"yes"* class. However, it was shown in Table 4.15, that the choice of AdaBoost.M1 without pre-sampling was superior for almost all eleven classes, except for a few exceptions. Therefore, even if the primary objective was to correctly classify a certain meteorite, the decision of choosing AdaBoost.M1 without sampling is still valid.

The boosting algorithm SAMME performed slightly worse than AdaBoost.M1 in this classification simulation of the imbalanced data set comecs. However, it has

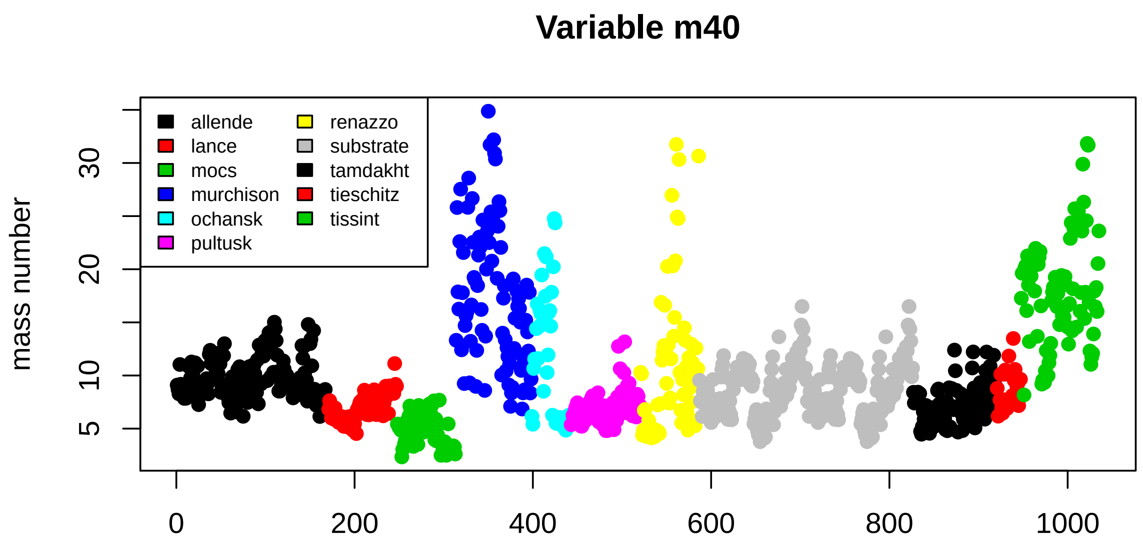


Figure 4.13: Variable plot of the important spectrum *m40* for all observations in the data set.

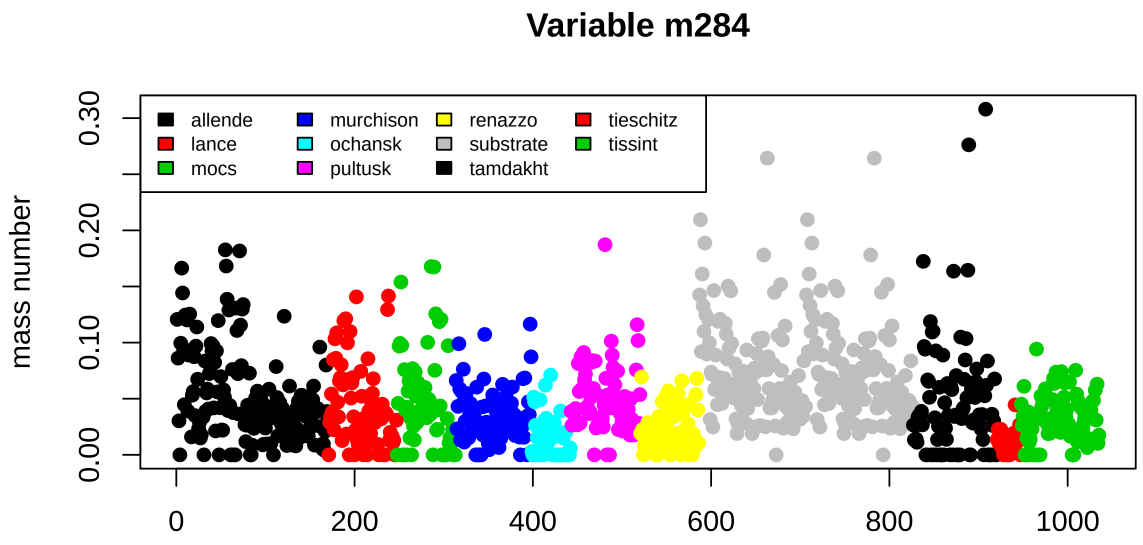
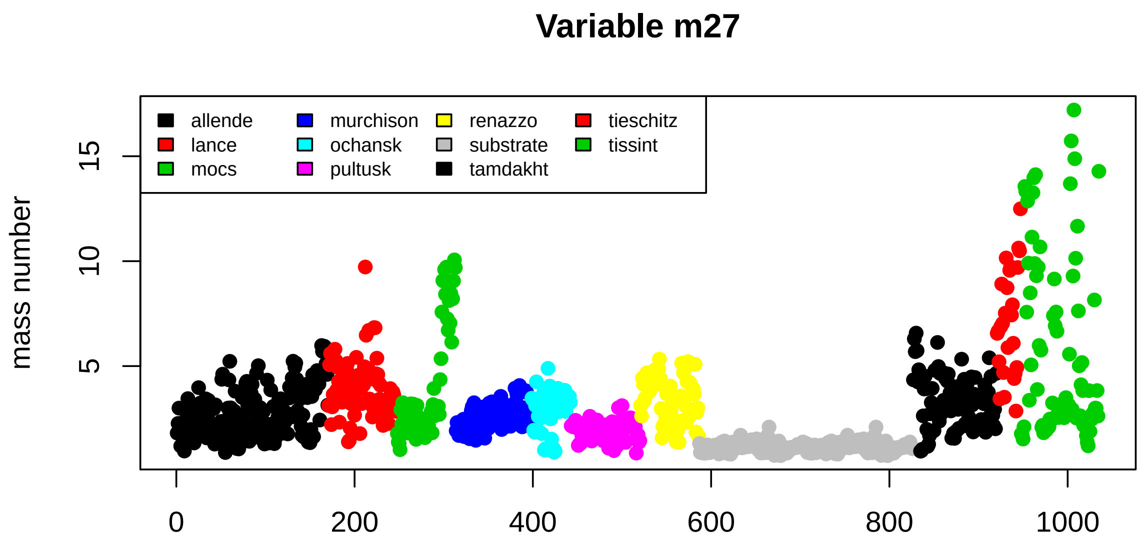


Figure 4.14: Variable plot of the least important spectrum *m284* for AdaBoost.M1 without sampling, for all observations in the data set.

a similar behaviour with regards to the choice of sampling method, as it was best to not sample, followed by over-sampling and then to use same-size.

The minority methods of under-and naive-sampling clearly had the worst results of the five considered methods. It is interesting to note, that this is the opposite of what occurred in the previous binary simulation of Section 4.2.2. There, the best option was to naive-sample RealBoost or GentleBoost, followed by either same-size or under-sampling depending on the chosen boosting algorithm. Same-size generally appears to a safe choice of sampling method to deal with the issue of imbalanced data sets, as it was only slightly worse than over-sampling and it likewise performed well in the binary simulation. Furthermore it is computationally superior and has less tendency to overfit compared to over-sampling.

Further studies could encompass the use of different weak learns with AdaBoost.m1 and other boosting algorithms, instead of the classification trees which were used. A quick way to test a different base procedure is by changing the tree's properties such as the maximum depth of any node of the final tree, maybe even reducing to stumps. Various other weak learners can used as well, preferably those that satisfy the recommended properties of Section 2.4.1.
Of course other multi-class boosting algorithms could be used to possibly achieve better results than with AdaBoost.M1 or SAMME.



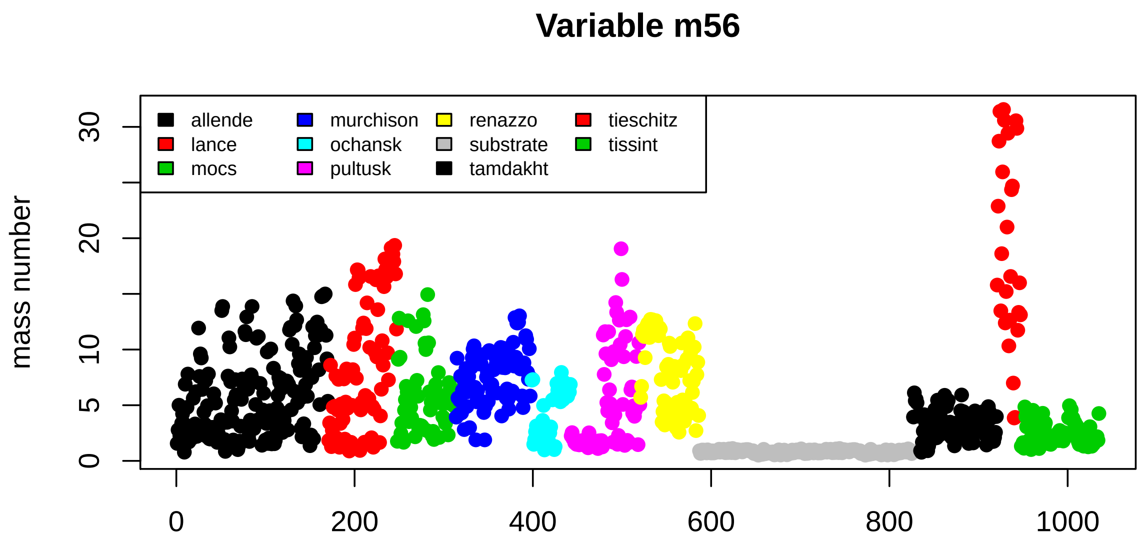


Figure 4.15: Variable plot of the important spectra *m27* and *m56* for all observations in the data set.

Appendix R Code

The custom R-function *sample_unb* is used to apply the different sampling methods of Section 4.1. In the case of the binary and multi-class simulations the input will be the training set and the output will be a vector of indices of the training set, which are then used by the boosting algorithm to build the model.

```
#### sample_unb - samples data according to the chosen method
## Input:
# data...data set which shall be sampled
# method..."under", "naive", "sss"(same size sampling) or "over"
## Output:
# fin_ind... indices of data which were sampled by method
sample_unb <- function (data , method ='naive') {
n  <- nrow(data)
data_ind   <- c(1:n)
repl <- TRUE
#  naive methode
if( method == 'naive'){
repl <- FALSE
n_s <- min(   table(  data[, ncol(data) ]  )    )
}
# under sample
if( method == 'under'){
n_s <- min(   table(  data[, ncol(data) ]  )    )
}
# over sample
if( method == 'over'){
n_s <- max(   table(  data[, ncol(data) ]  )    )
}
# same sample size
if( method == 'sss'){
n_class <- length(table(data[, ncol(data)]))
n_s  <- round ( n/n_class ,0)
}

fin_ind <- tapply(data_ind, data[, ncol(data) ],function (i){
sample (i, size = n_s , replace = repl ) })

return(unlist(fin_ind))
}
```

The custom R-function *Binary_Run* is used to achieve the binary simulations of Section 4.2.2. The input consists of the bank data of Section 4.2.1, the number of iterations($It = T = 100$), the type of sampling method used and a number indicating how many runs shall be done(100). The output includes the training and test errors, the percentages of correct *"yes"* classification and the variable importance for each algorithm.

```
#### Binary_Run - simulation of binary experiment
## Input:
# data...data set which shall be sampled
# It...number of iterations for each boosting algorithm
# runs...number of simulation runs
# p... split variable of training/test set.
## Output:
# ans... list containing all the relevant results
Binary_Run <- function (data, It = 100, runs = 100, method ='none',
 p=0.7,... ) {
## INITIALIZE
# Error Test/Training
Ada_errTe <- Ada_errTr <- Gen_errTe <- NULL
Gen_errTr <- Rea_errTe <- Rea_errTr <- NULL
Ada_yes <- Gen_yes <- Rea_yes <- NULL   # test "yes"-classification-rate
Ada_vip <- Gen_vip <- Rea_vip <- NULL   # variable importance

n <- dim(data)[1] # number of observation's
l <- ncol(data)   # number of variables + 1

###### Start runs
# TIME MANAGEMENT
ptm0 <- proc.time()

for(i in 1:runs){
## Training and Test Set
trind<-sample(1:n,floor(p*n),FALSE)
teind<-setdiff(1:n,trind)
##  Unbalanced Sampling method
if(method %in% c("over","under","naive","sss")){
data_tr  <- data[trind,]
sampl    <- sample_unb(data_tr, method=method)
}else{
data_tr <- data[trind,]
sampl   <- c(1:nrow(data_tr))
}

## Boosting
########### ADABOOST
BankAdaBoost <- ada(y~., data=data_tr[sampl,], iter=It, type="discrete",
    loss="e")
BankAdaBoost <- addtest(BankAdaBoost, data[teind,-l], data$y[teind])
########### Gentle BOOST
BankGentle <- ada(y~., data=data_tr[sampl,], iter=It, type="gentle",
    loss="e")
BankGentle <- addtest(BankGentle, data[teind,-l], data$y[teind])
########## Real BOOST
BankReal <- ada(y~., data=data_tr[sampl,], iter=It, type="real",
    loss="e")
BankReal <- addtest(BankReal, data[teind,-l], data$y[teind])

## TRAINING AND TEST ERROR  =   ## MISS-CLASSIFICATION %
# ADA
Ada_errTeTr <- BankAdaBoost$model
Ada_errTeTr <- Ada_errTeTr$errs
Ada_errTe <- cbind(Ada_errTe,Ada_errTeTr[,3])  # Test Error
Ada_errTr <- cbind(Ada_errTr,Ada_errTeTr[,1])  # Training Error
# Gentle
Gen_errTeTr <- BankGentle$model
Gen_errTeTr <- Gen_errTeTr$errs
Gen_errTe <- cbind(Gen_errTe,Gen_errTeTr[,3])  # Test Error
Gen_errTr <- cbind(Gen_errTr,Gen_errTeTr[,1])  # Training Error
# Real
Rea_errTeTr <- BankReal$model
Rea_errTeTr <- Rea_errTeTr$errs
Rea_errTe <- cbind(Rea_errTe,Rea_errTeTr[,3])  # Test Error
Rea_errTr <- cbind(Rea_errTr,Rea_errTeTr[,1])  # Training Error

## TEST ERROR of class "yes"
b_true     <- data[teind,l]
# ADA
AdaFitTest <- predict(BankAdaBoost, newdata=data[teind,-l])
d <- table(data.frame(AdaFitTest,b_true))
Ada_yes <- rbind(Ada_yes,d[4]/(d[3]+d[4]))
# GEN
AdaFitTest <- predict(BankGentle, newdata=data[teind,-l])
d <- table(data.frame(AdaFitTest,b_true))
Gen_yes <- rbind(Gen_yes,d[4]/(d[3]+d[4]))
# REA
AdaFitTest <- predict(BankReal, newdata=data[teind,-l])
d <- table(data.frame(AdaFitTest,b_true))
Rea_yes <- rbind(Rea_yes,d[4]/(d[3]+d[4]))

## Variable Importance
vip_tmp <-  varplot(BankAdaBoost, plot.it = FALSE, type = "scores")
vip_tmp <-  vip_tmp[order(names(vip_tmp))]
Ada_vip <- cbind(Ada_vip, vip_tmp)
vip_tmp <-   varplot(BankGentle, plot.it = FALSE, type = "scores")
vip_tmp <- vip_tmp[order(names(vip_tmp))]
Gen_vip <- cbind(Gen_vip, vip_tmp)
vip_tmp <-   varplot(BankReal, plot.it = FALSE, type = "scores")
vip_tmp <- vip_tmp[order(names(vip_tmp))]
Rea_vip <- cbind(Rea_vip, vip_tmp)

## ITERATION Number and time passed
ptm1=proc.time() - ptm0        # Time of individual step i
jnk=as.numeric(ptm1[3])
cat('Total Runtime:', jnk, "Loop at run", i,'\n')
}
#################### End of for-loop ####################

### Convert Information
# Test/Training Error, development
Ada_Tr <- apply(Ada_errTr,1,mean)
Ada_Te <- apply(Ada_errTe,1,mean)
Rea_Tr <- apply(Rea_errTr,1,mean)
Rea_Te <- apply(Rea_errTe,1,mean)
Gen_Tr <- apply(Gen_errTr,1,mean)
Gen_Te <- apply(Gen_errTe,1,mean)
# Test/Training Error, final iteration=It
Comb_Tr <- cbind(Ada_errTr[It,],Rea_errTr[It,],Gen_errTr[It,])
colnames(Comb_Tr) <- c("AdaBoost", "RealBoost","GentleBoost")
Comb_Te <- cbind(Ada_errTe[It,],Rea_errTe[It,],Gen_errTe[It,])
colnames(Comb_Te) <- c("AdaBoost", "RealBoost","GentleBoost")
# Yes-classification of training
Comb_Yes <- cbind(Ada_yes, Rea_yes, Gen_yes)
colnames(Comb_Yes) <- c("AdaBoost", "RealBoost","GentleBoost")

### Save
ans <- list(Ada_Tr = Ada_Tr,Ada_Te = Ada_Te,
Rea_Tr = Rea_Tr,Rea_Te = Rea_Te,
Gen_Tr = Gen_Tr,Gen_Te = Gen_Te,
Comb_Tr = Comb_Tr,Comb_Te = Comb_Te,
Comb_Yes = Comb_Yes,
Ada_vip = Ada_vip, Gen_vip = Gen_vip,
Rea_vip = Rea_vip
)

return(ans)
}
```

Analogous to *Binary_Run*, the custom R-function *Multi_Run* is used to achieve the multi class simulations of Section 4.3.2. The input consists of the meteorite data of Section 4.3.1, the number of iterations($It = T = 100$), the type of sampling method used and a number indicating how many runs shall be done(100). The output includes the training and test errors, the correct classification percentages and the variable importance for each algorithm.

```
#### Multi_Run - simulation of multi-class experiment
## Input:
# data...data set which shall be sampled
# It...number of iterations for each boosting algorithm
# runs...number of simulation runs
# p... split variable of training/test set.
## Output:
# ans... list containing all the relevant results
Multi_Run <- function (data, It = 100, runs = 100, method ='none',
p=0.7,... ) {
## INITIALIZE
# Variables
var_names <- colnames(data[,-298])
# Test/Training Error by Iterations
M1_errTr  <- M1_errTe  <- NULL
M2_errTr  <- M2_errTe  <- NULL
# Class Errors
M1_ClTe <- SAMME_ClTe <- NULL
# Variable Importance
M1_vip     <- SAMME_vip  <- NULL

n <- dim(data)[1] # number of observation's
l <- ncol(data)   # number of variables + 1

# TIME MANAGEMENT
ptm0 <- proc.time()

for(i in 1:runs){
## Training and Test Set
trind<-sample(1:n,floor(p*n),FALSE)
teind<-setdiff(1:n,trind)
## Imbalanced Sampling method
if(method %in% c("over","under","naive","sss")){
data_tr  <- data[trind,]
sampl    <- sample_unb(data_tr, method=method)
}else{
data_tr  <- data[trind,]
sampl   <- c(1:nrow(data_tr))
}
## Boosting
################### AdaBoost.M1
Met_M1 <-boosting(names ~., data=data_tr[sampl,] , mfinal = It,
 coeflearn = "Breiman")
################### SAMME
Met_SAMME  <-  boosting(names ~., data=data_tr[sampl,] , mfinal = It ,
 coeflearn = "Zhu" )
## TRAINING AND TEST ERROR =   ## MISS-CLASSIFICATION %
# M1
ErTr_M1  <- errorevol(Met_M1,newdata=data[trind, ])
ErTe_M1  <- errorevol(Met_M1,newdata=data[teind, ])
M1_errTr <- cbind(M1_errTr,ErTr_M1$error)
M1_errTe <- cbind(M1_errTe,ErTe_M1$error)
# SAMME
ErTr_SAMME  <- errorevol(Met_SAMME,newdata=data[trind, ])
ErTe_SAMME  <- errorevol(Met_SAMME,newdata=data[teind, ])
SAMME_errTr <- cbind(SAMME_errTr,ErTr_SAMME$error)
SAMME_errTe <- cbind(SAMME_errTe,ErTe_SAMME$error)

## Individual Class Test Errors
# M1
M1_pred  <- predict.boosting(Met_M1, newdata = data[teind, ])
M1_table  <- M1_pred$confusion
M1_ClTe   <- cbind(M1_ClTe,   diag(M1_table)/apply(M1_table, 2, sum))
# SAMME
SAMME_pred  <- predict.boosting(Met_SAMME, newdata = data[teind, ])
SAMME_table <- SAMME_pred$confusion
SAMME_ClTe  <- cbind(SAMME_ClTe,   diag(SAMME_table)/apply(SAMME_table,
2, sum))
## Variable Importance
# M1
tmp_1    <- Met_M1$importance
tmp_1    <- tmp_1[var_names]
M1_vip    <- cbind(M1_vip, tmp_1)
# SAMME
tmp_1    <- Met_SAMME$importance
tmp_1    <- tmp_1[var_names]
SAMME_vip <- cbind(SAMME_vip, tmp_1)

# ITERATION Number and time passed
ptm1=proc.time() - ptm0         # Time of individual step i
jnk=as.numeric(ptm1[3])
cat('Total Runtime:', jnk, "Loop at run", i,'\n')
}
#################### End of for-loop ###################

### Convert Information
# Test/Training Error, development
M1_Tr   <- apply(M1_errTr,1,mean)
M1_Te   <- apply(M1_errTe,1,mean)
SAMME_Tr <- apply(SAMME_errTr,1,mean)
SAMME_Te <- apply(SAMME_errTe,1,mean)

# Test/Training Error, final iteration=It
```

```
Comb_Tr <- cbind(M1_errTr[It,],SAMME_errTr[It,])
colnames(Comb_Tr) <- c("AdaBoost.M1", "SAMME")
Comb_Te <- cbind(M1_errTe[It,],SAMME_errTe[It,])
colnames(Comb_Te) <- c("AdaBoost.M1", "SAMME")

### Save
ans <- list(M1_Tr = M1_Tr,M1_Te = M1_Te,
SAMME_Tr = SAMME_Tr,SAMME_Te = SAMME_Te,
Comb_Tr = Comb_Tr,Comb_Te = Comb_Te,
M1_ClTe = M1_ClTe, SAMME_ClTe = SAMME_ClTe,
M1_vip = M1_vip,  SAMME_vip = SAMME_vip
)

return(ans)
}
```

Bibliography

Alfaro, E., M. Gamez, and N. Garcia (2013). *adabag: An R Package Ada for Classification with Boosting and Bagging.* R package version 4.1.

Bartlett, P. and S. Mendelson (2002). "Rademacher and Gaussian Complexities: Risk Bounds and Structural Results". *Journal of Machine Learning Research* 3, pp. 463–482.

Breiman, L. (1999). "Prediction games and arcing classifiers". *Neural Computation Vol 11(7)*, pp. 1493–1517.

Breiman, L., J. Friedman, R. Olshen, and C. Stone (1984). *Classification and regression trees.* Monterey, CA: Wadsworth, Brooks/Cole Advanced Books, and Software.

Bühlmann, P. and S. van de Geer (2011). *Statistics for High Dimensional Data.* Zürich: Springer.

Caruana, R. and A. Niculescu-Mizil (2006). "An empirical comparison of supervised learning algorithms". *In Proc. 23 rd Intl. Conf. Machine learning (ICML'06)*, pp. 161–168.

Chen, C., A. Liaw, and L. Breiman (2004). *Using Random Forest to Learn Imbalanced Data.* Tech. rep. Department of Statistics, Berkeley, University of California.

CoMeCS-Project (2017). *Comet and Meteorite Materials - Studied by Chemometrics of Spectroscopic Data.* URL: `http://www.lcm.tuwien.ac.at/comecs/` (visited on 12/09/2017).

Culp, M., K. Johnson, and G. Michailidis (2016). *ada: The R Package Ada for Stochastic Boosting.* R package version 2.0-5.

Freund, Y. (1995). "Boosting a weak learning algorithm by majority". *AT&T Bell Laboratories, New Jersey.*

Freund, Y. and R. Schapire (1996a). "Experiments with a New Boosting Algorithm". *Machine Learning: Proceedings of the Thirteenth International Conference.*

— (1996b). "Game Theory, On-line Prediction and Boosting". *Proceedings of the Ninth Annual Conference on Computational Learning Theory,* pp. 325–332.

— (1997). "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". *Journal of computer and system sciences 55*, pp. 119–139.

— (1999). "A Short Introduction to Boosting". *Journal of Japanese Society for Artificial Intelligence*, pp. 771–780.

Friedman, J. (2001). "Greedy Function Approximation: A Gradient Boosting Machine". *The Annals of Statistics Vol. 29*, pp. 1189–1232.

Friedman, J., T. Hastie, and R. Tibshirani (2000). "Additive Logistic Regression: a Statistical View of Boosting". *Annals of Statistics*, pp. 337–407.

Hastie, T., R. Tibshirani, and J. Friedman (2008). *The Elements of Statistical Learning.* Stanford California: Springer.

Kircher, A. (2016). "Random Forest for Unbalanced Multiple-Class Classification". MA thesis. Wien: Technischen Universität Wien.

Mason, L., P. Bartlett, J. Baxter, and M. Frean (1999). "Boosting Algorithms as Gradient Descent". *Proceedings of the 12th International Conference on Neural Information Processing Systems*, pp. 512–518.

Meir, R. and G. Rätsch (2003). "An Introduction to Boosting and Leveraging". *Advanced Lectures on Machine Learning.* Ed. by Mendelson and Smola. Berlin, Heidelberg: Springer.

Moro, S., P. Cortez, and P. Rita (2014). "A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems". *Decision Support Systems, 62:22-31.*

Moro, S., R. Laureano, and P. Cortez (2011). "Using Data Mining for Bank Direct Marketing: An Application of the CRISP-DM Methodology". *Proceedings of the European Simulation and Modelling Conference*, pp. 117–121.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". *Journal of Machine Learning Research* 12, pp. 2825–2830.

R-Core-Team (2013). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing. Vienna, Austria. URL: `http://www.R-project.org/` (visited on 05/08/2017).

Reyzin, L. (2014). "On Boosting Sparse Parities". *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pp. 2055–2061.

Schapire, R. (1990). "The Strength of Weak Learnability". *Machine Learning*, pp. 197–227.

— (1999). "Theoretical Views of Boosting". *Computational Learning Theory: Fourth European Conference, EuroCOLT 99,* pp. 1–10.

— (2013). "Explaining AdaBoost". *Empirical Inference.* Ed. by B Schölkopf, Z. Luo, and V. Vovk. Springer, Berlin, Heidelberg, pp. 37–52.

Schapire, R., Y. Freund, P. Bartlett, and W. Sun Lee (1998). "Boosting the margin: A new explanation for the effectiveness of voting methods". *Annals of Statistics Volume 26*, pp. 1651–1686.

Viola, P. and M. Jones (2003). "Robust Real Time Face Detection". *International Journal of Computer Vision 57*, pp. 137–154.

Zhu, J., S. Rosset, H. Zou, and T. Hastie (2009). "Multi-class AdaBoost". *Statistics and its Interface*, pp. 349–360.