Solving classification problems with boosting

Tak Cheung

BWI-paper

September 2001

vrije Universiteit
Faculty of Sciences
Division of Mathematics and Computer Science
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands

Preface

The “BWI-paper” is one of the last part of the study program BWI (short for: “Business Mathematics and Computer Science”). The student is obliged to search for a subject individually, look for the related literature and write a scientific paper.

The study program BWI consists of about one third of each of the components Mathematics, Computer Science and Economics/Econometrics. BWI has three focus areas, namely “Computational Intelligence”, “Financial Mathematics” and “Optimization of Business Processes”.

During my study, I followed the course “Data Mining Techniques” which was taught by Dr. W.J. Kowalczyk. Because the subjects of the course were very interesting, I decided to write my BWI-paper about a subject in the field of data mining. I have chosen to study a special kind of algorithms, called boosting algorithms. These algorithms are chosen for the following reasons. The boosting algorithms are based on very interesting mathematical theory. According to the literature, the boosting algorithms outperform benchmark machine learning problems. The boosting algorithms have some nice applications, such as the credit card fraud detection problem. Thus, in my opinion the subject of this paper fits perfectly in the so-called “triangle of BWI”, where each of the three above mentioned components plays an important role, with “Computational Intelligence” as focus area.

I like to thank my supervisor Wojtek Kowalczyk for giving me some useful answers to my questions and for his patience, because I invested quite some time to finish this paper. Further, I like to thank Uty Pang-Atjok for giving me some hints concerning the typewriting program LaTeX, which was used to write this paper.

Tak Cheung.
Amsterdam, September 2001.

Summary

"Boosting" is one of the most important recent development in classification methodology. It is a general method for improving the performance of any learning algorithm.

In this paper the main question of interest was whether boosting algorithms are useful for solving classification problems. To answer this question several recent boosting algorithms were studied in depth. First, a theory associated with the boosting algorithms was given and second, the boosting algorithms were tested on small real world data for different classification problems.

Besides the fact that the boosting algorithms are easy to understand, have no parameters to tune (except for the number of rounds) and have nice theoretical properties, the great advantages of the boosting strategy for small classification problems are:

- The boosting strategy often outperforms "standard" classifciation algorithms, like Decision trees, k-Nearest Neighbors and Naïve Bayes Classifiers.
- The model seldom suffers from overtraining when boosting algorithms are used.

These two advantages follow not only from the results of the experiments in the literature but also from our own research. The disadvantage of the boosting strategy for small classification problems we experienced during our research is:

- The boosting algorithms can require much computation time to receive satisfying results.

In practice, one of the problem could be that the data is so large that it cannot fit into main memory. Another problem could be that the data has a very skewed distribution. Both problems play an important role in the credit card fraud detection domain. One approach to overcome these problems is to use the concept of "meta-learning". Meta-learning is loosely defined as learning from learned knowledge. Another approach is to use boosting algorithms.

Experiments from the literature have shown that meta-learning show a significant improvement over the results obtained by using standard commercial fraud detection systems. One of the main disadvantage is the need to run preliminary experiments to determine the desired distribution of the data and then to transform the data into the desired distribution, which is needed because data with a very skew distribution might not yield the most effective classifiers.

There has not been much experiments of applying boosting algorithms on real world credit data. Experiments have only shown that using a cost-sensitive boosting algorithm result in lower misclassification cost than using a regular boosting algorithm. Further research is necessary to asses the practical value of boosting for the credit card fraud detection problem.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

“Boosting” is one of the most important recent development in classification methodology. It is a general method for improving the performance of any learning algorithm. Boosting has its roots in a theoretical framework in machine learning called the ’probably approximately correct’ (PAC) learning model.

In 1984 Kearns and Valiant were the first to pose the question of whether a “weak” learning algorithm which performs just slightly better than random guessing in the PAC learning model can be “boosted” into an arbitrarily accurate “strong” learning algorithm. Schapire came up with the first provable polynomial-time boosting algorithm in 1989. A year later, Freund developed a much more efficient boosting algorithm which, although optimal in a certain sense, nevertheless suffered from practical drawbacks. In 1995 Freund and Schapire introduced the AdaBoost algorithm, which solved many of the practical difficulties of the earlier boosting algorithms. Since then, boosting has received much attention.

Boosting encompasses a family of methods. The focus of these methods is to produce a series of classifiers. The training set used for each member of the series is chosen based on the performance of the earlier classifier(s) in the series. In boosting, examples that are incorrectly predicted by previous classifiers in the series are chosen more often than examples that are correctly predicted. Thus, boosting attempts to produce new classifiers that are better able to predict examples for which the current ensemble’s performance is poor.

An interesting question we would like to answer is:
Are boosting algorithms useful for solving classification problems?

In this paper we shall try to answer this question. This is done as follows.

Since the introduction of the AdaBoost algorithm, many new boosting algorithms have been introduced. In Chapter 2 the most popular boosting algorithms since the introduction of the AdaBoost algorithm are described in detail. Besides the pseudo-code, also some error bounds and theory associated with the boosting algorithms are given. Because boosting algorithms work by sequentially applying a learning algorithm, the so-called “weak learning algorithms”, we shall also describe some of these “weak learning algorithms”. This is done in Chapter 3. After describing some boosting and weak learning algorithms, the boosting algorithms are tested on real-world data and some remarks about the experiments are made. This is done in Chapter 4.

A very interesting classification problem is the credit card fraud detection problem. Recently, this classification problem has been approached by a concept, called “meta-learning”, which led to better results than standard commercial fraud detection systems. Meta-learning is loosely defined as learning from learned knowledge. Another approach is to use boosting algorithms. In Chapter 5 both approaches are described.

Finally, some conclusions, appendixes and a bibliography close the paper.

Chapter 2

Description of various boosting algorithms

2.1 Introduction

In this chapter several boosting algorithms, which are known in the literature, are described in detail. The following categorization is made:

- Boosting algorithms for binary classification problems,
- Boosting algorithms for general classification problems.

2.2 Boosting algorithms for binary classification problems

2.2.1 Introduction

In 1995 Freund & Schapire proposed the *'AdaBoost'* procedure. Later, in 1998, Schapire & Singer presented a more generalized version of AdaBoost. When authors refer to the *'AdaBoost'* algorithm they are referring to the generalized version.

In the same year, Friedman, Hastie & Tibshirani showed that both the AdaBoost algorithms can be interpreted as stage-wise estimation procedures for fitting an additive *logistic* regression model, using a criterion similar to, but not the same as, the binomial log-likelihood. They derived a new boosting procedure that directly optimizes the binomial log-likelihood. Based on this algorithm, they also derived a "gentler" version of the generalized version of AdaBoost, which uses Newton stepping rather than exact optimization at each step.

In the literature, different authors use different names for the same boosting algorithm. The boosting algorithms described in this chapter have the same name as in Friedman, Hastie & Tibshirani [10]. We will call the algorithm AdaBoost that was proposed by Freund & Shapire, *'Discrete AdaBoost'* [8]. The generalized version of AdaBoost that was proposed by Schapire & Singer will be called *'Real AdaBoost'* [18]. The two algorithms that was proposed by Friedman, Hastie & Tibshirani will be called respectively *'LogitBoost'* and *'Gentle AdaBoost'* [10][11].

2.2.2 Discrete Adaboost

The boosting algorithm takes as input a training set of N examples $S = \{(x_1, y_1), \dots, (x_N, y_N)\}$ where each *instance* x_i is a vector of attribute values that belongs to a *domain* or *instance space* X, and each *label* y_i is the class label associated with x_i that belongs to a finite *label space* Y. We only focus on binary classification problems in which $Y = \{-1, 1\}$.

In addition, the boosting algorithm has access to another unspecified learning algorithm, called the weak learning algorithm, which is denoted generically as *weak learner*. The boosting algorithm

calls the weak learner repeatedly in a series of rounds. On round t, the algorithm provides the weak learner with a distribution D_t over the training set. In response, the weak learner computes a classifier or *hypothesis* $h_t : X \mapsto Y$ which should correctly classify a fraction of the training set that has large probability with respect to D_t. That is, the weak learner's goal is to find a hypothesis h_t which minimizes the (training) error ϵ_t.
The error is defined by $\epsilon_t = \mathbb{E}_{D_t}[1_{(y \neq h_t(x))}] = \mathbb{P}_{D_t}(y \neq h_t(x)) = \sum_{i:y_i \neq h_t(x_i)} D_t(i)$, where $\mathbb{E}_{D_t}$ and $\mathbb{P}_{D_t}$ represents respectively the expectation and the probability over the training data with respect to the distribution D_t and 1_S is the indicator of the set S, which takes the value 1 if S is true and takes the value 0 if S is false. Notice that the error is measured with respect to the distribution D_t on which the weak learner was trained.

At each iteration the algorithm increases weights of the observations misclassified by $h_t(x)$ by a factor that depends on the weighted training error, and otherwise the weight is left unchanged. The weights are then normalized by dividing by a normalization constant Z_t. Effectively, "easy" examples that are correctly classified by many of the previous weak hypotheses get lower weight, and "hard" examples which tend often to be misclassified get higher weight, so the weak learner is forced to focus on the hard examples in the training set.

In the last step the algorithm algorithm combines the *weak hypotheses* $h_1, \dots, h_t$ by a weighted sum into a single final hypothesis $H(x)$. Weak hypotheses with a higher error receive a lower weight.

The pseudocode of Discrete AdaBoost is given in Figure 2.1.

Analyzing the training error

An important theoretical property about Discrete AdaBoost is stated in the following theorem. This theorem [8] shows that if the weak hypotheses consistently have error only slightly better than 1/2, then the training error of the final hypotheses H drops to zero exponentially fast. For binary classification problems this means that the weak hypotheses need to outperform only slightly better than random.

Theorem 1 *Suppose the weak learning algorithm, when called by Discrete AdaBoost, generates hypotheses with errors $\epsilon_1, \dots, \epsilon_T$, where ϵ_t is defined in Figure 2.1. Assume each $\epsilon_t \leq 1/2$, and let $\gamma_t = 1/2 - \epsilon_t$. Then the following upper bound holds the error of the final hypothesis H:*

$$\frac{|\{i : H(x_i) \neq y_i\}|}{N} \leq \prod_{t=1}^{T} \sqrt{1 - 4\gamma_t^2} \leq \exp\Big(-2 \sum_{t=1}^{T} \gamma_t^2\Big).$$

Theorem 1 implies that the *training error* of the final hypothesis generated by Discrete AdaBoost is small. This does not necessarily imply that the *test* error is small.

2.2.3 Real AdaBoost

Let $S = \{(x_1, y_1), \dots, (x_N, y_N)\}$ be a sequence of training examples where each *instance* x_i belongs to a *domain* or *instance space* X, and each *label* y_i belongs to a finite *label space* Y. We focus on binary classification problems in which $Y = \{-1, 1\}$.

We assume access to a *weak* or *base* learning algorithm which accepts as input a sequence of training examples S along with a distribution D over $\{1, \dots, N\}$, i.e., over the indices of S. Given such input, the weak learner computes a *weak* (or *base*) *hypothesis* h. In general, h has the form $h : X \mapsto \mathbb{R}$. We interpret the sign of $h(x)$ as the predicted label (-1 of +1) to be assigned to instance x, and the magnitude $|h(x)|$ as the "confidence" in this prediction. Thus, if $h(x)$ is close to or far from zero, it is interpreted as a low or high confidence prediction.
This slightly generalized version of Discrete AdaBoost algorithm is shown in Figure 2.2.

When $\alpha_t > 0$, the main effect of Real AdaBoost's update rule is to decrease or increase the weight of the training examples classified correctly or incorrectly by h_t (i.e., examples i for which

Discrete AdaBoost (Freund & Schapire 1995)

- Given: $S = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, where $x_i \in X, y_i \in Y = \{-1, 1\}$.
- Initialize $D_1(i) = 1/N$ $(i = 1, \ldots, N)$.
- For $t = 1, 2, \ldots, T$:
  1. Train weak learner using distribution D_t.
  2. Get weak hypothesis $h_t(x) : X \mapsto \{-1, 1\}$ with error

  $$\epsilon_t = \mathbb{E}_{D_t}[1_{(y \neq h_t(x))}] = \mathbb{P}_{D_t}(y \neq h_t(x)).$$

  3. Choose $\alpha_t = \ln(\frac{1-\epsilon_t}{\epsilon_t})$.
  4. Update:

  $$D_{t+1}(i) = \frac{D_t(i)\exp(\alpha_t \cdot 1_{(y \neq h_t(x_i))})}{Z_t}, (i = 1, 2, \ldots, N),$$

  where Z_t is a normalization fator (chosen so that D_{t+1} will be a distribution).
- Output the final hypothesis:

$$H(x) = \text{sign}(F(x)) \text{ , where } F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

Figure 2.1: Discrete AdaBoost Algorithm.

Real AdaBoost (Schapire & Singer 1998)

- Given: $S = \{(x_1, y_1), \dots, (x_N, y_N)\}$, where $x_i \in X, y_i \in Y = \{-1, 1\}$.
- Initialize $D_1(i) = 1/N$ $(i = 1, \dots, N)$.
- For $t = 1, 2, \dots, T$:
  1. Train weak learner using distribution D_t.
  2. Get weak hypothesis $h_t(x) : X \mapsto \mathbb{R}$ with error
  $$\epsilon_t = \mathbb{E}_{D_t}[1_{(y \neq h_t(x))}] = \mathbb{P}_{D_t}(y \neq h_t(x)).$$
  3. Choose $\alpha_t \in \mathbb{R}$.
  4. Update:
  $$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}, (i = 1, 2, \dots, N),$$
  where Z_t is a normalization fator (chosen so that D_{t+1} will be a distribution).
- Output the final hypothesis:
$$H(x) = \text{sign}(F(x)) \text{ , where } F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

Figure 2.2: Real AdaBoost Algorithm.

y_i and $h_t(x_i)$ agree or disagree is sign). This version differs from Discrete Adaboost in that weak hypotheses can have range over all of $\mathbb{R}$ rather than the restricted values -1 and +1. A second difference is that the choice of α_t is unspecified.

Analyzing the training error

The most basic property of Real AdaBoost concerns its ability to reduce the training error. It can be shown that Real AdaBoost is indeed a generalization of Discrete AdaBoost when h_t takes only the values of +1 and -1 [17] (see Appendix A).

Schapire & Singer [18] derived the choice of α_t in the case that h_t has range [-1,+1] (rather than the values $\{-1, +1\}$) and proved the following corollary. (See Appendix A for the derivations).

Corollary 1 *Using the notation of Figure 2.2, assume each h_t has range $[-1,+1]$ and that we choose*

$$\alpha_t = \frac{1}{2}\ln\Big(\frac{1+r_t}{1-r_t}\Big),$$

where

$$r_t = \sum_{i=1}^{N} D_t(i) y_i h_t(x_i) = \mathbb{E}_{D_t}[y_i h_t(x_i)].$$

Then the training error of H is at most

$$\prod_{t=1}^{T} \sqrt{1-r_t^2}.$$

2.2.4 LogitBoost

Estimating probabilities

Roughly speaking, classification is the problem of predicting the label y of an example x with the intention of minimizing the probability of an incorrect prediction. However, it is often useful to estimate the *probabilty* of a particular label. Friedman, Hastie & Tibshirani [10] suggested a method for using the output of Real AdaBoost to make reasonable estimates of such probabilities. Specifically, they suggest using a logistic function, and estimating

$$\mathbb{P}_F(y = +1|x) = \frac{e^{F(x)}}{e^{F(x)} + e^{-F(x)}} \tag{2.1}$$

where, as usual, $F(x)$ is the weighted average of weak hypotheses produced by Real AdaBoost. The rationale for this choice is the close connection between the log loss (negative log likelihood) of such a model, namely,

$$\sum_{i=1}^{N} \ln\Big(1 + e^{-2y_i F(x_i)}\Big) \tag{2.2}$$

and the function which Real AdaBoost attempts to minimize:

$$\sum_{i=1}^{N} e^{-y_i F(x_i)}. \tag{2.3}$$

Specifically, it can be varified that Eq. (2.2) is upper bounded by Eq. (2.3). In addition, if the constant $(1 - \ln 2)$ is added to Eq. (2.2) (which does not affect its minimization), then it can be verified that the resulting function and the one in Eq. (2.3) have identical Taylor expansions around zero up to second order; thus, their behaviour near zero is very similar. Finally, it can be shown that, for any distribution over pairs (x, y), the expectations

$$\mathbb{E}[\ln(1 + e^{-2yF(x)})]$$

and

$$\mathbb{E}[e^{-yF(x)}]$$

are minimized by the same function F, namely,

$$F(x) = \frac{1}{2}\ln\Big(\frac{\mathbb{P}(y = +1|x)}{\mathbb{P}(y = -1|x)}\Big).$$

Thus, for all these reasons, minimizing Eq. (2.3), as is done by Real AdaBoost, can be viewed as a method of approximately minimizing the negative log likelihood given in Eq. (2.2). Therefore, we may expect Eq. (2.1) give a reasonable estimate.

Additive model

Friedman, Hastie & Tibshirani [10] showed that Discrete AdaBoost and Real AdaBoost can be interpreted as stagewise estimation procedures for fitting an additive logistic regression model. (In the Real AdaBoost case, when taking

$$h_t(x) = \frac{1}{2}\ln\frac{p_t(x)}{1 - p_t(x)} \text{ , where } p_t(x) = \hat{\mathbb{P}}(y = 1|x) \in [0, 1].$$

When this choice is made for h_t, we will call the Real AdaBoost algorithm the *population version* of Real AdaBoost).
They optimize an exponential criterion which to second order is equivalent to the binomial log-likelihood criterion. (See Appendix B for the derivation). They also derived a new boosting procedure "LogitBoost" that directly optimizes the binomial log-likelihood. (See Appendix B for the derivation).
A detailed description of LogitBoost is given in Figure 2.3.
Note that y_i^* is taken, which has values 0 and 1 instead of y_i, which has values -1 and +1.

Numerical note

Sometimes the $w(x)$ get very small when $p(x)$ is close to 0 or 1. This can cause numerical problems in the construction of z. The avoid this, the following implementation protections can be used:

- If $y^* = 1$, then compute $z = \frac{y^* - p}{p(1-p)}$ as $\frac{1}{p}$. Since this number can get large if p is small, treshold this ratio at $zmax$. The particular value choses for $zmax$ is not crucial; Friedman, Hastie & Tibshirani have found emperically that $zmax \in [2, 4]$ works well. Likewise, if $y^* = 0$, compute $z = \frac{-1}{(1-p)}$ with a lower treshold of $-zmax$.
- Enforce a lower threshold on the weights: $w = \max(w, 2 * \text{machine-zero})$.

LogitBoost - 2 classes (Friedman, Hastie & Tibshirani 1998)

- Start with weights $w_i = 1/N$ $(i = 1, \ldots, N)$, $F(x) = 0$ and probability estimates $p(x_i) = \frac{1}{2}$.
- Repeat for $t = 1, 2, \ldots, T$:
  1. Compute the working response and weights

     $$z_i = \frac{y_i^* - p(x_i)}{p(x_i)(1-p(x_i))}$$
     $$w_i = p(x_i)(1 - p(x_i))$$

  2. Estimate $h_t(x)$ by the weighted least-squares fitting of z_i to x_i using weights w_i.
  3. Update $F(x) \leftarrow F(x) + \frac{1}{2}h_t(x)$ and $p(x) \leftarrow \frac{e^{F(x)}}{e^{F(x)}+e^{-F(x)}}$.
- Output the classifier: $H(x) = \text{sign}[F(x)] = \text{sign}[\sum_{t=1}^{T} h_t(x)]$.

Figure 2.3: LogitBoost Algorithm.

2.2.5 Gentle AdaBoost

The population version of the Real AdaBoost procedure optimizes $\mathbb{E}e^{y(F(x)+h(x))}$ exactly with respect to f at each iteration. The "Gentle AdaBoost" procedure instead takes adaptive Newton steps much like the LogitBoost algorithm just described (See Appendix B for the derivation). A desription of the Gentle AdaBoost algorithm is given in Figure 2.4.

In the experiments we will use in all the algorithms the following estimate of the weighted class probabilities to update the functions

$$h_t(x) = \mathbb{P}_{D_t(i)}(y = 1|x) - \mathbb{P}_{D_t(i)}(y = -1|x), \tag{2.4}$$

rather than half the log-ratio:

$$h_t(x) = \frac{1}{2}\ln\frac{\mathbb{P}_{D_t(i)}(y = 1|x)}{\mathbb{P}_{D_t(i)}(y = -1|x)}. \tag{2.5}$$

The first reason for using Eq. (2.4) instead of Eq. (2.5) is that log-ratios can be numerically unstable, leading to very large updates in pure regions, while the update in Eq. (2.4) lies in the range $[-1, 1]$. Empirical evidence by Friedman, Hastie & Tibshirani suggests that using Eq. (2.4) instead of Eq. (2.5) also works very well and is often even better, especially when stability is an issue.
The second reason for using Eq. (2.4) instead of Eq. (2.5) is that Theorem 3 can now be applied. Thus, the main difference between the Gentle AdaBoost algorithm and the Real AdaBoost algorithm is that the first algorithm does not have a factor α_t in the exponent of the weight-updating rule.

Gentle AdaBoost (Friedman, Hastie & Tibshirani 1998)

- Start with weights $D_1(i) = 1/N$ $(i = 1, \dots, N)$, $F(x) = 0$.
- Repeat for $t = 1, 2, \dots, T$:
  1. Estimate $h_t(x)$ by weighted least-squares of y_i to x_i with weights $D_t(i)$.
  2. Update: $F(x) \leftarrow F(x) + h_t(x)$.
  3. Update:

$$D_{t+1}(i) = \frac{D_t(i)\exp(y_i h_t(x_i))}{Z_t}, (i = 1, 2, \dots, N),$$

  where Z_t is a normalization fator (chosen so that D_{t+1} will be a distribution).

- Output the classifier: $H(x) = \text{sign}[F(x)] = \text{sign}[\sum_{t=1}^{T} h_t(x)]$.

Figure 2.4: Gentle AdaBoost Algorithm.

2.3 Boosting algorithms for general classification problems

2.3.1 AdaBoost - J classes

There are several methods of extending AdaBoost to the multiclass case. The most straightforward generalization [9], called AdaBoost.M1, is adequate when the weak learner is strong enough to achieve reasonably high accuracy, even on the hard distributions created by AdaBoost. However this method fails if the weak learner cannot achieve at least 50% accuracy when run on these hard distributions.

For the latter case, several more sophisticated methods have been developed. These generally work by reducing the multiclass problem to a larger binary problem. Schapire and Singer's [18] algorithm AdaBoost.MH works by creating a set of binary problems, for each example x and each possible label y, of the form: “For example x, is the correct label y or is it one of the others?”. Freund and Schapire's [8] algorithm AdaBoost.M2 (which is a special case of Schapire each example x with correct label y and each *incorrect* label y' of the form: “For example x, is the correct label y or y'?”.

These methods require additional effort in the design of the weak learning algorithm. We shall therefore only give the AdaBoost.MH algorithm which is shown in Figure 2.5. We will consider the more general case *multi-label* case in which a single example may belong to any number of classes.

AdaBoost.MH (Schapire & Singer 1998)

- Expand the original N observations into N x J pairs $((x_i,1),y_{i1})),((x_i,2),y_{i2})),\ldots,((x_i,J),y_{iJ}))$, $i=1,\ldots,N$. Here y_{ij} is the $\{-1,1\}$ response for class j and observation i.
- Start with weights $D_1(i,j)=1/(N*J)$ $(i=1,\ldots,N, j=1,\ldots,J)$.
- Apply Real AdaBoost to the augmented dataset producing a function $F: X$ x $(1,\ldots,J)\mapsto \mathbb{R}$.
- Output the classifier: $\text{argmax}_j F(x,j)$, where $F(x,j)=\sum_{t=1}^{T} h_t(x,j)$.

Figure 2.5: AdaBoost.MH Algorithm.

2.3.2 LogitBoost - J classes

We start off by a natural generalization of the two-class symmetric logistic transformation.

Definition 1 *For a J class problem, let $p_j(x)=\mathbb{P}(y_j=1|x)$. We define the symmetric multiple logistic transformation*

$$F_j(x)=\ln(p_j(x))-\frac{1}{J}\sum_{k=1}^{J}\ln(p_k(x)).$$

Equivalently,

$$p_j(x)=\frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}} \quad , \quad \sum_{k=1}^{J} F_k(x)=0. \tag{2.6}$$

The centering condition in Eq. (2.6) is for numerical stability only; it simply pins the F_j down, else we could add an arbitrary constant to each F_j and the probabilities remain the same. The equivalence of these two definitions is easily established, as well as the equivalence with the two-class case.

In Figure 2.6 a natural generalization of the LogitBoost algorithm for 2 classes is given. It can be shown that the LogitBoost algorithm (J classes) uses quasi-Newton steps for fitting an additive symmetric logistic model by maximum-likelyhood. This is done in Appendix B.

LogitBoost - J classes (Friedman, Hastie & Tibshirani 1998)

- Start with weights $w_{ij} = 1/N$ $(i = 1, \ldots, N, j = 1, \ldots, J)$, $F_j(x) = 0$ and $p_j(x) = \frac{1}{J}, \forall j$.
- Repeat for $t = 1, 2, \ldots, T$:
  - Repeat for $j = 1, \ldots, J$:
    1. Compute working responses and weights in the jth class
    $$z_{ij} = \frac{y^*_{ij} - p_j(x_i)}{p_j(x_i)(1 - p_j(x_i))}$$
    $$w_{ij} = p_j(x_i)(1 - p_j(x_i))$$
    2. Fit the function $h_{tj}(x)$ by a weighted least-squares regression of z_{ij} to x_i with weights w_{ij}.
  - Set $h_{tj}(x) \leftarrow \frac{J-1}{J}(h_{tj}(x) - \frac{1}{J}\sum_{k=1}^{J} h_{tk}(x))$, and $F_j(x) \leftarrow F_j(x) + h_{tj}(x)$.
  - Update $p_j(x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_j(x)}}$.
- Output the classifier: $\text{argmax}_j F_j(x)$.

Figure 2.6: LogitBoost Algorithm (J classes).

Chapter 3

Choices for the weak learning algorithm

3.1 Introduction

In this chapter several choices for the weak learning algorithm or weak learner, are presented. The following choices have been made for the weak learner, which shall later be used in the experiments:

- Decision stumps
- k-Nearest Neighbor Algorithm
- Naïve Bayes Classifier

There are many choices for weak learner and the list is far from complete. However, the three algorithms are chosen for special reasons.

The first algorithm is chosen because many authors use this algorithm for the weak learning algorithm to evaluate the performance of the boosting algorithms.

The last two algorithms are also very popular in the field of machine learning. For this reason they are used for the weak learning algorithm. It is not known if there exists papers which uses these two algorithms for the weak learning algorithm.

Each of the listed weak learners will be described and will be provided with an example. A more detailed description of the k-Nearest Neigbor Algorithm and the Naïve Bayes Classifier can be found in [14]. For very detailed descriptions of tree classification in general, see [1].

Note that for the "weak" learning algorithm also a strong learning algorithm can be used. The k-Nearest Neigbor Algorithm and the Naïve Bayes Classifier are quite strong algorithms. Another quite strong algorithm that is used by many authors is the C4.5 algorithm, which constructs decision trees.

3.2 Decision stumps

"Decision stumps" or simply "stumps" is a word used by Friedman, Hastie & Tibshirani [10] for a "two-terminal node decision tree". Decision stumps have the form shown in Figure 3.1.

In this figure the t_j's (j is the attribute number) are called *nonterminal nodes* and the $t_{j,L}$'s and $t_{j,R}$'s are called *terminal nodes*.
With stumps as base classifier, each component function has the form

$$h_t(x) = c_t^L \cdot 1_{[x_j \leq s_t]} + c_t^R \cdot 1_{[x_j > s_t]} = h_t(x_j)$$

if the t-th stump chose to split on coordinate j. Here s_t is the split-point, and c_t^L and c_t^R are the weighted means of the response in the left and reight terminal nodes. Thus the model produced by boosting stumps is additive in the *original* features

$$F(x) = \sum_{j=1}^{p} g_j(x_j),$$

where $g_j(x_j)$ adds together all those stumps involving x_j (and is 0 if none exist).

In the experiments all h_t's are found by optimizing the mean squared error on a weighted version of the training data.

3.3 k-Nearest Neighbor Algorithm

The k-Nearest Neighbor Algorithm is a simple but powerful classification tool. The main idea behind this algorithm is that cases that are close to each other should have the same labels. When classifying a case it looks at k of its closest neighbors and classifies the case according to the majority of the classifications of its neighbors. For determining the closest neighbors, the algorithm can use several distance measures. The most frequently used measures are the Euclidean and the Hamming distance.

Given two vectors v and w, with

$$v = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \quad \text{and} \quad w = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix},$$

the Hamming distance between v and w is defined by

$$H = \sum_{i=1}^{n} |v_i - w_i|,$$

and the Euclidean distance between v and w is defined by

$$E = \sqrt{\sum_{i=1}^{n} (v_i - w_i)^2}.$$

The algorithm uses a training set as a collection of "reference points" and estimates the labels of cases with unknown classification in the training set. We will give an example.

Consider the situation in Figure 3.2, where each data point has two attribute values and the Euclidean distance measure is used for classification.
Let the dot in the center of the plot be the case to be classified, the black dots be cases with classification label 0 and let the grey dots be cases with classification label 1. The results of the k-Nearest Neighbor Algorithm is shown in Table 3.1.
When a k is used that has an even value, it could occure that the number of neighbors with classification 0 equals the number of neighbors with classification 1. When this occurs, the classification label is undecided. In practice, this is solved by assigning a classification label at random.

Next, consider the situation where the points in Figure 3.2 are provided with the weights given in Table 3.2 (The point closest to the point to be classified is "data point number 1"), then the result of the new situation is shown Figure 3.3.
In the new situation the distances between the point to be classified and the data points 2 and 3 have *relatively* increased by a factor 2 and the distances between the point to be classified and the data points 4 and 6 have *relatively* decreased by a factor 2. The results of the k-Nearest Neighbor Algorithm for the new situation is shown in Table 3.3.

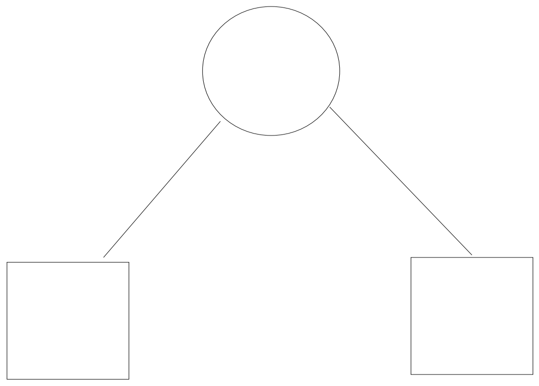

Figure 3.1: Decision stump: a two-terminal node decision tree.

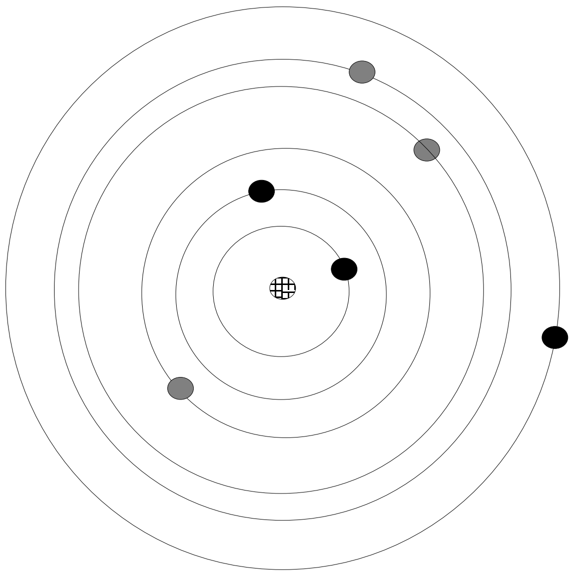

Figure 3.2: An illustration of the k-Nearest Neigbor Algorithm.

k	number of 0's	number of 1's	prediction
1	1	0	0
2	2	0	0
3	2	1	0
4	2	2	undecided
5	2	3	1
6	3	3	undecided

Table 3.1: Results of k-Nearest Neigbor Algorithm for Figure 3.2.

data point number	**weight**
1	$\frac{1}{6}$
2	$\frac{1}{3}$
3	$\frac{1}{3}$
4	$\frac{1}{12}$
5	$\frac{1}{6}$
6	$\frac{1}{12}$

Table 3.2: Assigning weights to the examples in Figure 3.2

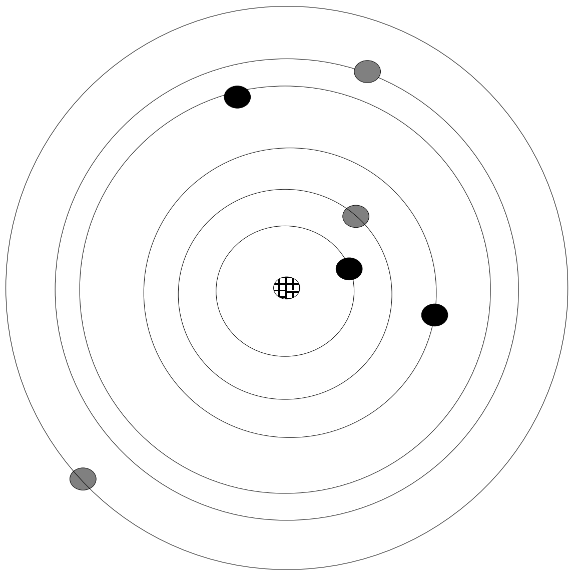

Figure 3.3: Situation of Figure 3.2 with weights from Table 3.2.

k	**number of 0's**	**number of 1's**	**prediction**
1	1	0	0
2	1	1	undecided
3	2	1	0
4	3	1	0
5	3	2	0
6	3	3	undecided

Table 3.3: Results of k-Nearest Neigbor Algorithm for Figure 3.3.

Attribute X	Attribute Y	Classification label
1	0	0
0	1	1
0	1	0
0	1	1
1	1	0
0	1	?

Table 3.4: Example of the data set for Naïve Bayes Classifier.

In the experiments, the weak learner considers each training example as a test example with the remaining cases in the training set as the training set. Further, the weak hypotheses $h(x_i)$ $(i = 1, \ldots, N)$ is defined by

$$\begin{cases} h_t(x_i) = \frac{\text{(number of examples in class 0) * -1 + (number of examples in class 1) * +1}}{k}, \\ \text{If } h_t(x_i) = 0, \text{ then } h_t(x_i) := \pm\frac{1}{k}. \end{cases}$$

(When Discrete AdaBoost is used, the sign of $h_t(x_i)$ is returned).

3.4 Naïve Bayes Classifier

The Naïve Bayes Classifier is a classification technique based on conditional probabilies. These conditional probabilities are estimated using historical data and based on these probabilities new cases are classified. An important rule is the following:

$$\mathbb{P}(\mathrm{A} \cap \mathrm{B}) = \mathbb{P}(\mathrm{A}|\mathrm{B})\mathbb{P}(\mathrm{B}) \quad \Longleftrightarrow \quad \mathbb{P}(\mathrm{A}|\mathrm{B}) = \frac{\mathbb{P}(\mathrm{A} \cap \mathrm{B})}{\mathbb{P}(\mathrm{B})}.$$

An important assumption is that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance (v_j), the probability of observing the conjunction $a_1, a_2, \ldots, a_n$ is just the product of the probabilities for the individual attributes: $\mathbb{P}(a_1, a_2, \ldots, a_n|v_j) = \prod_i \mathbb{P}(a_i|v_j)$.
The Naïve Bayes classifier uses this rule to obtain the conditional probabilities, which are needed for classifying new cases. These probabilities are estimated using a historical data set. The idea behind this classifier is to classify cases based on conditional probabilities of each value in the classification label given a certain combination of explanatory variables. We will give an example to illustrate the algorithm.

Consider the data set with one test case given in Table 3.4. In the training stage, the conditional probabilities are calculated according to the product rule, for example:

$$\mathbb{P}(\text{Classification} = 0|X = 1; Y = 0) = \frac{\mathbb{P}(\text{Classification} = 0; X = 1; Y = 0)}{\mathbb{P}(X = 1; Y = 0)} = \frac{1/5}{1/5} = 1.$$

This results in the probability matrix given in Table 3.5.
In the test stage we have the following probabitities for our test case:

$$\mathbb{P}(\text{Classification} = 0|X = 0; Y = 1) = \frac{1}{3} \quad , \quad \mathbb{P}(\text{Classification} = 1|X = 0; Y = 1) = \frac{2}{3}.$$

	X=0;Y=0	**X=0;Y=1**	**X=1;Y=0**	**X=1;Y=1**
Classification = 0	0	1/3	1	1
Classification = 1	0	2/3	0	0

Table 3.5: Results of Naïve Bayes Classifier for Table 3.4.

The test case will be classified according to the maximum of these probabilities; so the value of test label will be 1.

When weights are used, each of the conditional probabilities must be multiplied with the corresponding weight.

In the experiments, the weak learner considers each training example as a test example with the remaining cases in the training set as the training set. Further, the weak hypotheses $h(x_i)$ $(i = 1, \dots, N)$ is defined by

- $h(x_i) = \tilde{\mathbb{P}}$(Classification=0 | X=$m$;Y=$n$) * -1 + $\tilde{\mathbb{P}}$(Classification=1 | X=m;Y=n) * +1,

when the training case that is considered as a test case has value m for attribute X and value n for attribute Y and where $\tilde{\mathbb{P}}$ denotes the weighted probability.

(When Discrete AdaBoost is used, the sign of $h_t(x_i)$ is returned).

Chapter 4

Experiments with boosting algorithms

4.1 Introduction

In this chapter several data sets are used to evaluate the performance of the boosting algoithms. The datasets are from the UC-Irvine machine learning archive. The datasets used in this chapter are:

- Pima Indians Diabetes Database
- German Credit Data
- Heart Disease
- Waveform Data

(For a detailed description of the data sets, see Appendix C).

In the experiments we are mainly interested in the performance on the classification rate. We use *5-fold cross validation* to determine the classification rate. In Figure 4.1 a scheme for k-fold cross validation is given.
Unless stated otherwise, in all the experiments decision stumps are used for the weak learning algorithm. Decision stumps require less computation time, so that the number of iterations in boosting algorithm can be chosen very high. When using decision stumps or Naïve Bayes Classifiers the attributes which take continuous values need to be discretized.
Discretization is transforming continuous variables into a number of intervals.
For example, in the German Credit Data the variable 'age' may be discretized into three classes:
$0 = [0, 29], 1 = [30, 64], 2 = [65, \text{infinity}]$.
and 'loan duration (in month)' may be discretized into three classes:
$0 = [0, 12], 1 = [13, 24], 2 = [25, \text{infinity}]$.
All the other continous variables of the German Credit Data are discretized into 20 intervals, where each interval has the same length. This also holds for the continuous variables of Pima Indians Diabetes Database and Waveform Data Generator. In Heart Disease the continous variables are discretized into 10 intervals.

It is important to note that the objective of the experiments is to give some impressions of the results obtained by using the boosting algorithms. The results are not the optimal results.

k-fold cross validation

- Split the training set into k disjoint subsets of equal size.
- Treat every subset as a test set for a classifier which is trained on the remaining cases (k classifiers).
  Find their classification rates.
- Find the average classification rate.

Figure 4.1: k-fold cross-validation.

4.2 Experiments with boosting algorithms for binary classes

4.2.1 Pima Indians Diabetes Database

The results of running the four fitting methods: Discrete AdaBoost, RealAdaBoost, LogitBoost and Gentle AdaBoost are shown in Figure 4.2.
After 3000 iterations the following classification rate is obtained:
Discrete AdaBoost - training set: 83.01%, test set: 74.77%
Real AdaBoost - training set: 82.16%, test set: 73.59%
LogitBoost - training set: 84.25%, test set: 73.46%
Gentle AdaBoost - training set: 82.45%, test set: 74.12%

Thus, for this dataset the highest classification rate of the test set is obtained by using the Discrete AdaBoost algorithm. The figure also shows that Gentle AdaBoost gives good results when the number of iterations is relatively small. Futhermore, Gentle AdaBoost gives more stable results compared with Real AdaBoost. Further, we see that there is no overtraining.

We further notice that the maximal classification rate for the test set is 76.08% (Discrete AdaBoost - 700th iteration).

Results of other algorithms

In 1988 Smith, J.W., Everhart,J.E., Dickson,W.C., Knowler,W.C., & Johannes,R.S. used the *"ADAP learning algorithm"* to forecast the oneset of diabetes mellitus. ADAP algorithm makes a real-valued prediction between 0 and 1. This was transformed into a binary decision using a cutoff of 0.448. Using 576 training instances, the sensitivity and specificity of their algorithm was 76% on the remaining 192 instances. So, in their case three-forth of the data is used for training and one-forth of the data is used for testing.
If 4-fold cross-validation is used instead of 5-fold cross-validation in the experiments, the maximal classification rate is 75.52% instead of 76.08%. The results after 3000 iterations are almost identical. Thus, the boosting algorithms give reasonable results compared with the ADAP learning algorithm.

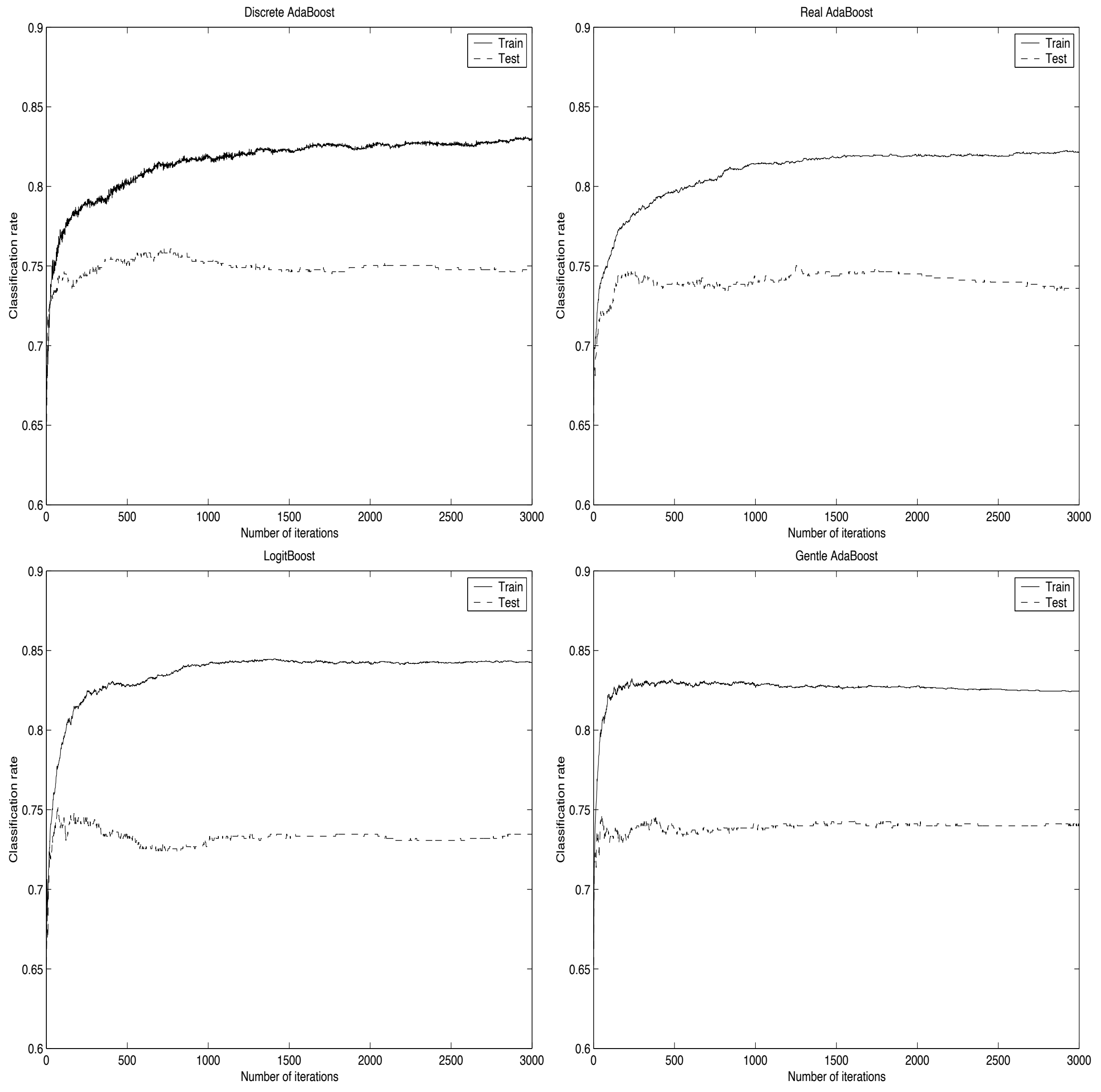

Figure 4.2: Classification rate of Pima Indians Diabetes Database.

4.2.2 German Credit Data

For now, we are only interested in the classification rate. Therefore, the cost matrix is omitted in the analyses.
The results of running the four fitting methods: Discrete AdaBoost, RealAdaBoost, LogitBoost and Gentle AdaBoost are shown in Figure 4.3.
After 3000 iterations the following classification rate is obtained:
Discrete AdaBoost - training set: 79.05%, test set: 71.50%
Real AdaBoost - training set: 78.93%, test set: 74.90%
LogitBoost - training set: 80.25%, test set: 75.40%
Gentle AdaBoost - training set: 78.90%, test set: 75.70%

Thus, for this dataset the highest classification rate of the test set is obtained by using the Gentle AdaBoost algorithm. This algorithm also gives very stable results compared with Real AdaBoost, which was also the case in "Pima Indians Diabetes Database" (see section 4.2.1). And again, there is no overtraining.

We further notice that the maximum classification rate for the test set is 75.90%
(LogitBoost - 642th iteration & Gentle AdaBoost - 86th iteration).

Results of other algorithms

A suitable algorithm for this data set is *Rough Data Models* [13] [12]. After experimenting with this algorithm the maximal classification rate obtained is 74.3%. It is not known if other algorithms can achieve a higher classification rate. Thus, we can only say that for this data set the boosting algorithms can give slightly better results than Rough Data Models.

4.2.3 Heart Disease

For now, we are only interested in the classification rate. Therefore, the cost matrix is omitted in the analyses.
The results of running the four fitting methods: Discrete AdaBoost, RealAdaBoost, LogitBoost and Gentle AdaBoost are shown in Figure 4.2.
After 3000 iterations the following classification rate is obtained:
Discrete AdaBoost - training set: 90.09%, test set: 80.00%
Real AdaBoost - training set: 90.37%, test set: 80.74%
LogitBoost - training set: 92.13%, test set: 79.26%
Gentle AdaBoost - training set: 89.72%, test set: 76.30%

From Figure 4.4 it is clear that the models are overtrained. It is not necessary to run 3000 iterations. A lower number of iterations, for example 50, is sufficient. After 50 iterations the following classification rate is obtained:
Discrete AdaBoost - training set: 87.96%, test set: 82.22%
Real AdaBoost - training set: 87.69%, test set: 81.85%
LogitBoost - training set: 89.54%, test set: 85.93%
Gentle AdaBoost - training set: 89.17%, test set: 82.59%

We further notice that the maximum classification rate for the test set is 87.41% (Gentle AdaBoost - 12th iteration).

Results of other algorithms

For this data set a very high classification rate can obtained by using *Rough Data Models*. Kowalczyk [13] [12] shows that with this method a classification rate of 84.93% can be obtained, which is more than the classification rate of 82.59% known in other literature.
When LogitBoost or Gentle AdaBoost is used, we get even higher classification rates. So, for this data set the boosting algorithms can give very good results.

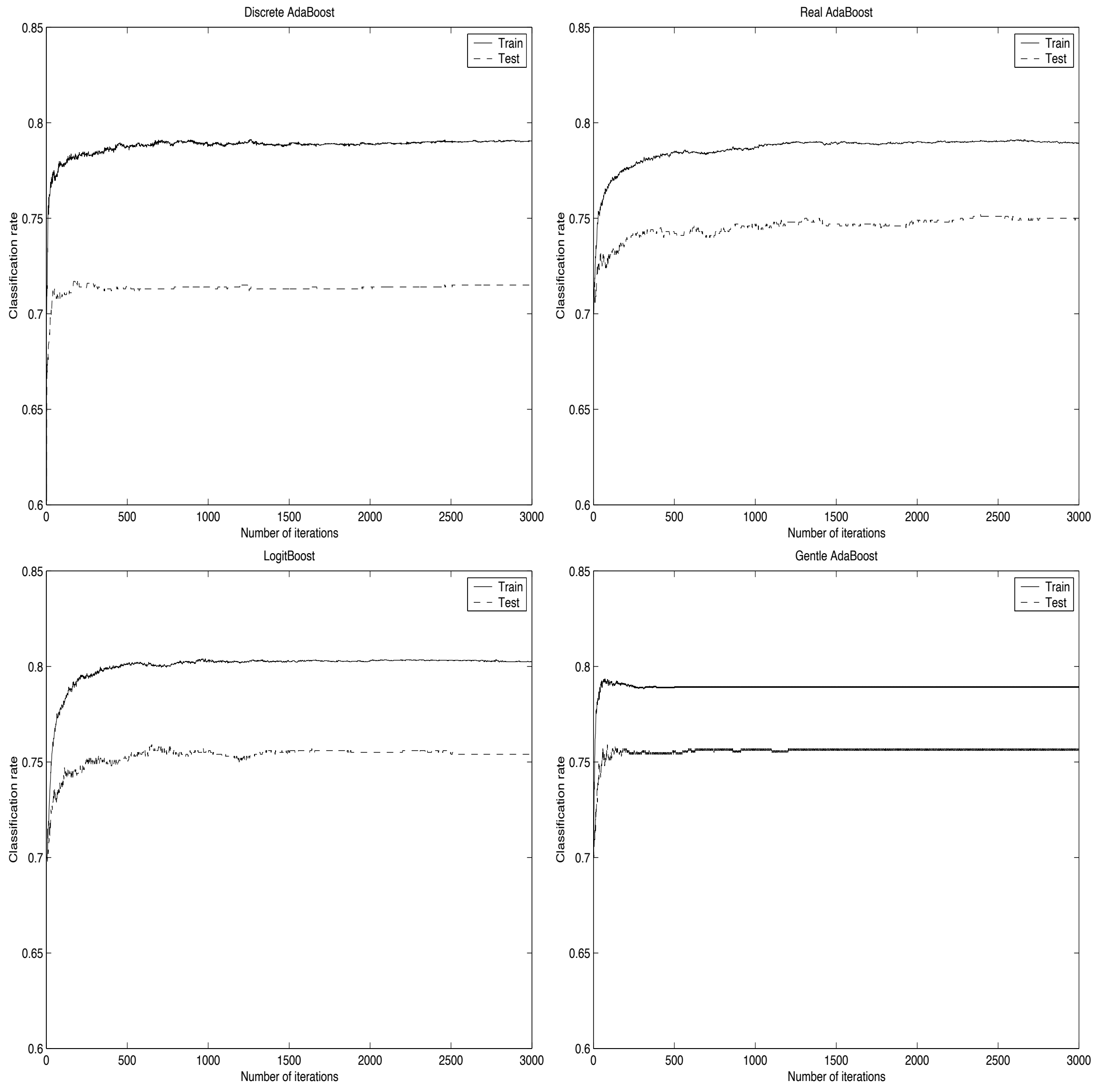

Figure 4.3: Classification rate of German Credit Data.

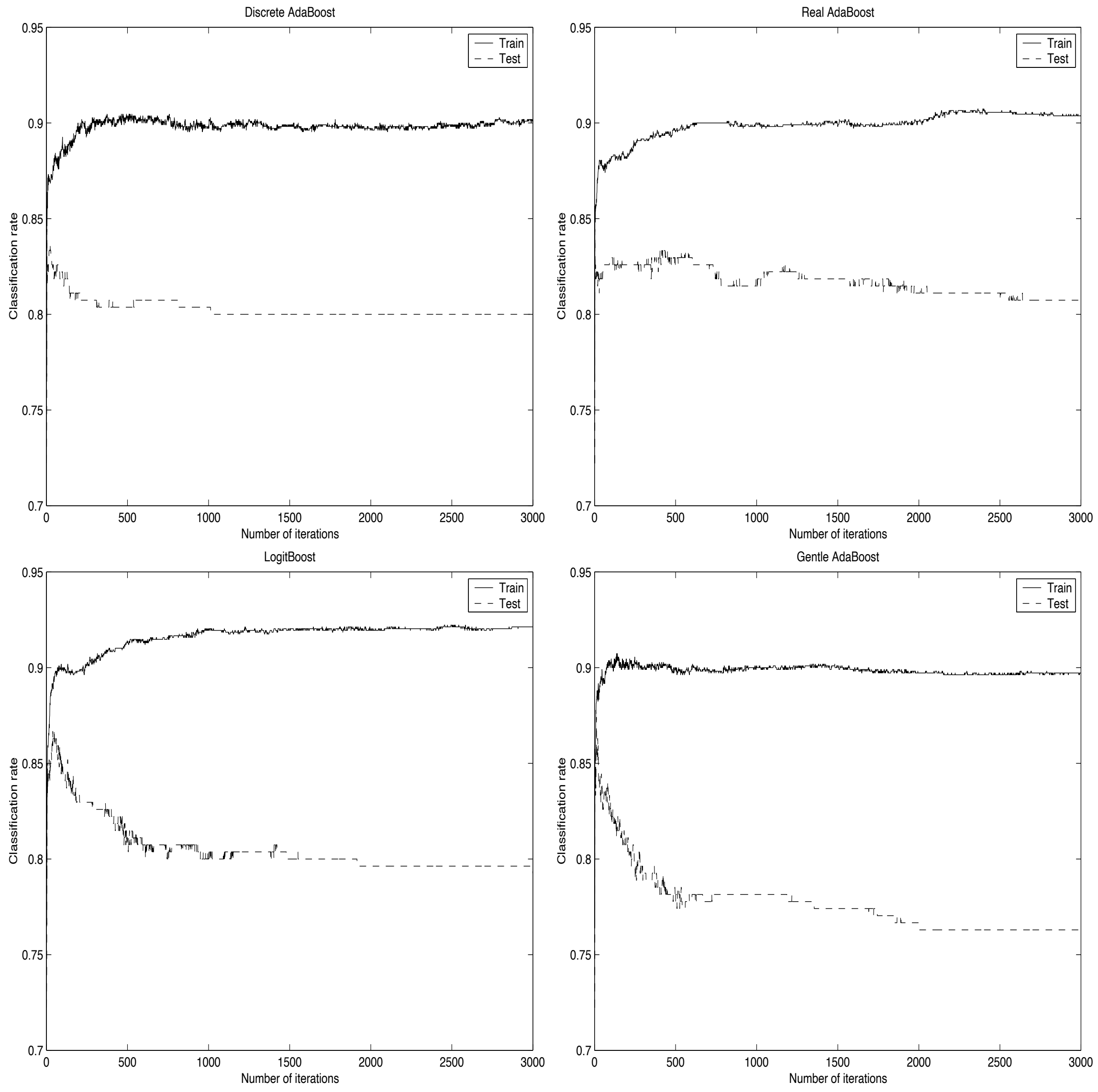

Figure 4.4: Classification rate of Heart Disease.

4.3 Experiments with boosting algorithms for multiple classes

4.3.1 Waveform Data

The results of running the two fitting methods: AdaBoost (J classes) & LogitBoost (J classes) are shown in Figure 4.5.
After 1000 iterations the following classification rate is obtained:
AdaBoost (J classes) - training set: 87.49%, test set: 85.22%
LogitBoost (J classes) - training set: 88.03%, test set: 85.34%

Results of other algorithms

In the past the following results were obtained:

- The optimal Bayes classification rate is 86%.
- With the CART decision tree algorithm a classification rate of 72% is achieved.
- With the Nearest Neighbor Algorithm a classification rate of 78% is achieved.

It is important to note that in all of these cases only 300 instances were used for training and 4700 instances were used for testing. If these numbers are used for the algorithms AdaBoost (J classes) and LogitBoost (J classes) the following classification rate is obtained after 1000 iterations:
AdaBoost (J classes) - training set: 100%, test set: 77.06%
LogitBoost (J classes) - training set: 96.33%, test set: 76.09%

Using 17-fold cross-validation, we obtain after 500 iterations:
AdaBoost (J classes) - training set: 99.88%, test set: 77.43%
LogitBoost (J classes) - training set: 98.66%, test set: 78.39%
When 17-fold cross-validation is used, each fold has $\frac{5000}{17} \approx 300$ instances. The results for this situation is shown in Figure 4.6. In this case boosting a simple decision tree gives better results than running a more sophisticated decision tree such as the CART algorithm.

When using 1-Nearest Neighbor as weak hypothesis, the maximal classification rate for the test set when using AdaBoost (J classes) is 78.89%. (See Figure 4.7). This is a little improvement over the 78% obtained by running the Nearest Neighbor Algorithm with the optimal k. Notice that running the boosting algorithm one round with 1-Nearest Neighbor Algorithm as weak learning algorithm is the same as running the Nearest Neighbor Algorithm with k=1, which will result in a classification rate of 76.79%.
When using Naive Bayes Classifier as weak hypothesis, the maximal classification rate for the test set when using AdaBoost (J classes) is also 78.89%. (See Figure 4.8). Running the boosting algorithm one round with Naive Bayes Classifier as weak learning algorithm is the same as running the Naive Bayes Algorithm, which will result in a classification rate of 78.34%.

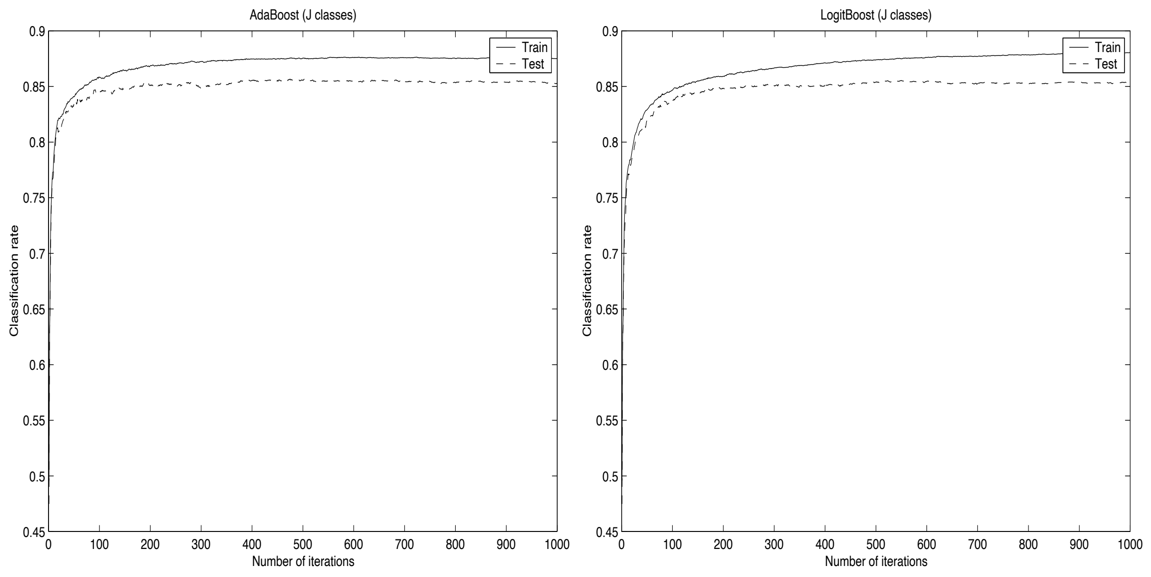

Figure 4.5: Classification rate of Waveform Data.

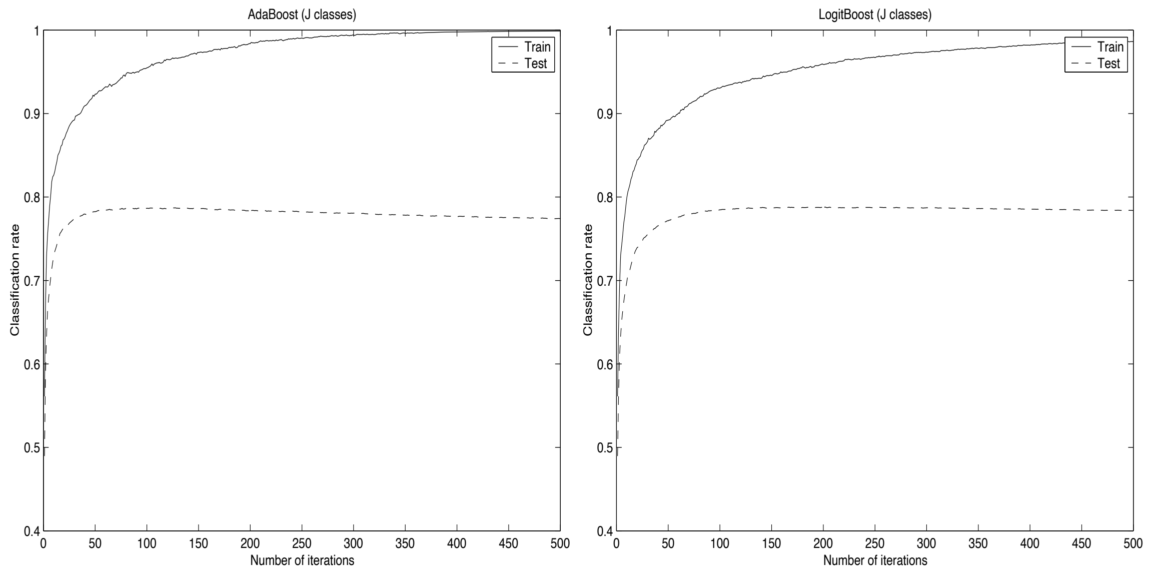

Figure 4.6: Classification rate of Waveform Data with small training dataset.

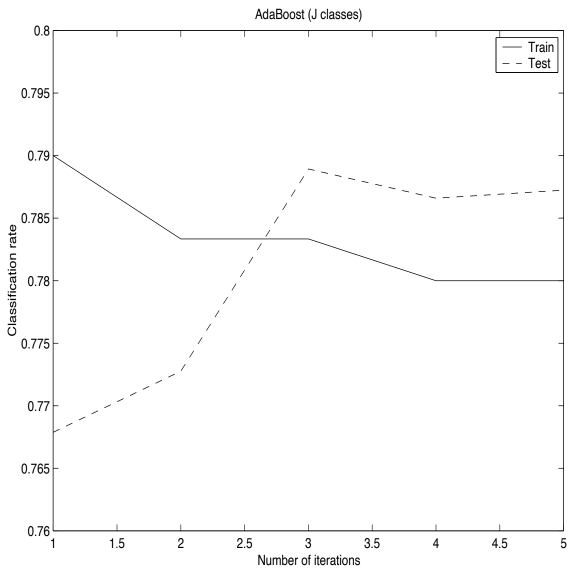

Figure 4.7: Classification rate of Waveform Data with 1-Nearest Neighbor Algorithm as weak learner.

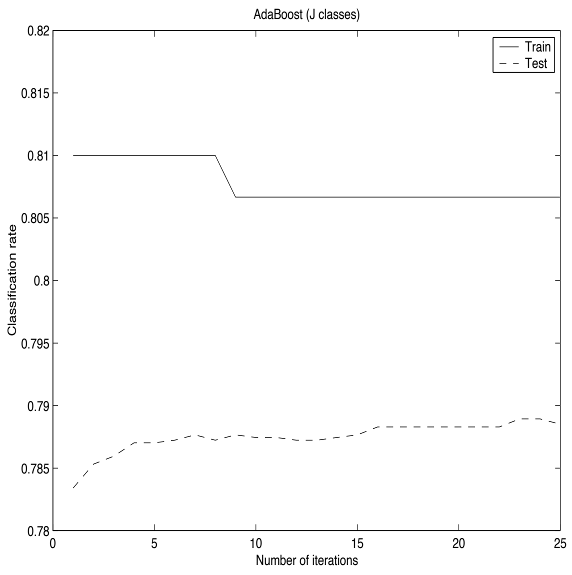

Figure 4.8: Classification rate of Waveform Data with Naive Bayes Classifier as weak learner.

4.4 Concluding remarks

Based on the experiments we make the following conclusions:

- The four different boosting algorithms does give different results on the classification rates. Which of the boosting algorithms gives the best result is dependent on the data set.

- If an algorithm is suitable for a specific classification problem, then it is worthwile to implement a weighted version of the algorithm and use this algorithm for the weak learning algorithm of the boosting algorithms.

- The disadvantage of using the boosting algorithms is that it can require much computation time to receive satisfying results.

  For example, Running 3000 iterations of Real AdaBoost with decision stumps on the German Credit Data and using 5-fold cross-validation requires about 2 hours on the Vrije Universiteit computer network. However, for most data sets it is not required to run so many iterations. Futher, the computation time can be reduced by using only the most important attributes. The importance of an attribute can for example be measured by considering the "Coefficient of Concordance" and the "Information Gain" [13][14]. For the "German Credit Data", selecting only the 4 attributes with the highest information gain, gives almost the same results but requires less computation time.

We conclude this chapter by giving some reasons of why and when boosting works ([8]).

Why does boosting work?

Two seperate reasons for the improvement in performance that is achieved by boosting are the following two effects that boosting has.

The first effect of boosting is that it generates a hypothesis whose error on the training set is small by combining many weak hypotheses whose eror may be large (but still better than random guessing).

The second effect of boosting has to do with variance reduction. Intuitively, taking a weighted majority over many hypotheses, all of which were trained on different samples taken out of the same training set, has the effect of reducing the random variability of the combined hypothesis. Boosting may have the effect of producing a combined hypothesis whose variance is significantly lower than those produced by the weak learner.

When does boosting work?

Boosting may be helpful on learning problems when they have either one of the following two properties.

The first property, which holds for many real-world problems, is that the observed examples tend to have varying degrees of hardness. For such problems the boosting algorithm tends to generate distributions that concentrate on the harder examples, thus challenging the weak learning algorithm to perform well on these harder parts of the sample space.

The second property is that the learning algorithm is sensitive to changes in the training examples so that significantly different hypotheses are generated for different training sets. Boosting tries actively to force the weak learning algorithm to change its hypotheses by changing the distribution over the training examples as a function of the errors made by previously generated hypotheses.

4.5 Experimental results of boosting in the literature

A lot of research on boosting has been done by several people over the years. The research includes experimenting with the boosting algorithms on real world data. In this section some results of the research will be mentioned.

Friedman, Hastie, & Tibshirani [10] have tested the four algorithms Discrete AdaBoost, Real AdaBoost, LogitBoost and Gentle AdaBoost on a collection of data sets from the UC-Irvine machine learning archive. They showed that, when they use the algorithms with a 2 or 8 terminal node decision tree as (base) classifier, the four algorithms give a lower misclassification error rate than the CART algorithm. Of all four algorithms, LogitBoost gives the most satisfying results. (In the experiments they used only 20 iterations and applied 5-fold cross validation if there were no pre-specified test set).

Freund & Schapire [8] compared boosting with “bagging”, a method introduced by Breiman in 1994 . Briefly, the method works by training each copy of the algorithm on a booststrap sample, i.e., a sample of size N chosen uniformly at random with replacement from the original training set S (of size N). The multiple hypotheses that are computed are then combined using simple voting; that is, the final composite hypothesis classifies an example x to the class most often assigned by the underlying “weak” hypotheses.
The differences between bagging and boosting can be summarized as follows: (1) bagging always uses resampling rather than reweighting; (2) bagging does not modify the distribution over examples or mislabels, but instead always uses the uniform distribution; (3) in forming the final hypothesis, bagging gives equal weight to each of the weak hypotheses.
They showed that for simple weak learning algorithms, boosting did significantly and uniformly better than bagging. When using C4.5 as weak learning algorithm, boosting and bagging seem more evenly matched, although boosting still seems to have a slight advantage. (In the experiments they used 100 iterations for both bagging and boosting and applied 10-fold cross validation if there were no pre-specified test set).

Chapter 5

Credit card fraud detection

5.1 Introduction

In today's increasingly electronic society and with the rapid advances of electronic commerce on the Internet, the use of credit cards for purchases has become convenient and neccessary. Credit card transactions have become the de facto standard for Internet and Web-based e-commerce. The US government estimates that credit cards accounted for approximately 13 billion US dollar in Internet sales during 1998. This figure is expected to grow rapidly each year. However, the growing number of credit card transactions provides more opportunity for thieves to steal credit card numbers and subsequently commit fraud. When banks lose money because of credit card fraud, cardholders pay for all of that loss through higher interest rates, higher fees and reduced benefits. Hence, it is in both the banks' and the cardholders' interest to reduce illegimate use of credit cards by early fraud detection.
The credit card fraud detection domain presents a number of challenging issues for data mining [5]:

- There are millions of credit card transactions processed each day. Mining such massive amounts of data requires highly efficient techniques that scale.
- The data are highly skewed: many more transactions are legitmate than fraudulent. Typical accuracy-based mining techniques can generate highly accurate fraud detectors by simply predicting that all transactions are legitimate, although this is equivalent to not detecting fraud at all.
- Each transaction record has a different dollar amount and thus has a variable potential loss, rather than a fixed misclassification cost per error type, as is commonly assumed in cost-based mining techniques.

In practice, the items mentioned above will give difficulties in detecting credit card fraud. The problems that arise are the following.

Because millions of credit card transactions are processed each day, it is possible that the whole data set cannot fit into main memory. But when the boosting algorithms, as decribed in chapter 2, are used, the whole data at a central site must be small enough to fit into main memory. When the data are highly skewed, classification algorithms might not yield the most effective classifiers. When each transaction record has a different dollar amount, a classification algorithm, for example the boosting algorithms described in Chapter 2, might classify many low amount transactions correct and some high amount transaction incorrect, so although the classification rate can be very high, the cost reduction may be low.

Fan et al. [6][7] propose adapted versions of the AdaBoost algorithm for the above mentioned problems. Before we give a decription of these algorithms, we first describe the concept of *meta-learning* and how to use this to solve the credit card fraud detection problem, which was done by Chan & Stolfo [2] [3] [4].

Chan & Stolfo use the following approach to handle massive amounts of data. A large data set is partitioned into subsets and the learning algorithm is run on each of the subsets. This will result in a set of base classifiers. These base classifiers will be combined into a single final classifier. The general strategy that seeks to learn how to combine a number of seperate learning processes in an intelligent fashion, is called *meta-learning*. One of the great benefits of introducing the meta-learning concept is that the problem of highly skewed data can be solved. Namely, the whole data set can be used to create data subsets with a desired distribution, generate classifiers from the subsets and finally integrate the classifiers by learning (meta-learning) from their classification behaviour.

This chapter is based mainly on [2] [3] [4], from which we make frequent quotes.

5.2 Meta-learning

5.2.1 Introduction

Meta-learning is loosely defined as learning from learned knowledge. In this case, we concentrate on learning from the output of concept learning systems. This is achieved by learning from the *predictions* of these classifiers on a common *validation data set*. Thus, we are interested in the output of the classifiers, not the internal structure and strategies of the learning algorithms themselves. Moreover, in some of the schemes defined, the data presented to the learning algorithms may also be available to the *meta-learner*.

In meta-learning a learning algorithm is used how to integrate the learned classifiers. That is, rather than having a predetermined and fixed integration rule, the integration rule is learned based on the behaviour of the trained classifiers. In the following subsections some of the different techniques used in the meta-learning study of Chan & Stolfo is presented.
The techniques fall into two general categories: the *arbiter* and *combiner* schemes.

The following distinguish between *base classifiers* and *arbiters/combiners* is made. A base classifier is the outcome of applying a leaning algorithm directly to 'raw' training data. The base classifier is a program that given a test datum provides a prediction of its unknown class. An arbiter or combiner, as detailed below, is a program generated by a learning algorithm that is trained on the predictions produced by a set of base classifiers and the raw training data. The arbiter/combiner is also a classifier, and hence other arbiters or combiners can be computed from the set of predictions of other arbiters/combiners.

5.2.2 Arbiter strategies

An *arbiter* is learned by some learning algorithm to arbitrate among predictions generated by different base classifiers. That is, its purpose is to provide an alternate and more educated prediction when the base classifiers present diverse predictions. This arbiter, together with an *arbitration rule*, decides a final classification outcome based upon the base predictions. Figure 5.1 depicts how the final prediction is made with the predictions from the two base classifiers and a single arbiter.
Let x be an instance whose classification we seek, $C_1(x), C_2(x), \ldots, C_k(x)$ are the predicted classifications of x from k base classifiers $C_1(x), C_2(x), \ldots, C_k(x)$ and $A(x)$ is the classification of x predicted by an arbiter. One example of an arbitration rule is as follows:

- Return the class with a plurality of votes in $C_1(x), C_2(x), \ldots, C_k(x)$ and $A(x)$, with preference given to the arbiter's choice in case of a tie.

Now, a detailed description of how an arbiter learns is given.
The training set of an arbiter is generated in a way that it contains the raw training examples whose classifications the base classifiers cannot predict consistently. Formally, a training set T

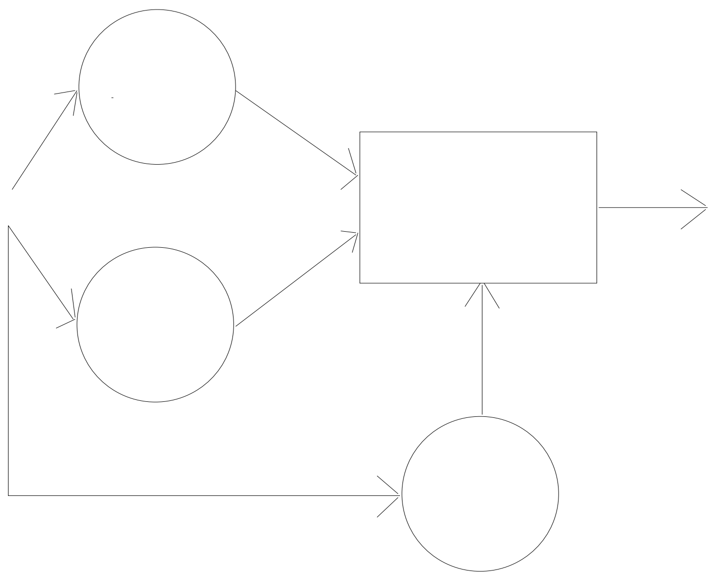

Figure 5.1: An arbiter with two classifiers.

for the arbiter is generated by picking examples from the validation set E. The validation set E is *randomly selected from all available data* prior to the onset of arbiter training. The choice of examples is dictated by a *selection rule*. One version of a selection rule (in the case there are more than two possible output labels) is as follows:

- An instance is selected if none of the classes in the k base predictions gathers a majority vote ($> k/2$ votes).

The purpose of this rule is to choose data that are in some sense "confusing"; i.e. the majority of classifiers do not agree on how the data should be classified. Figure 5.2 presents a sample training set for the arbiter strategy. Once the training set is formed, an arbiter is generated by the same learning algorithm used to train the base classifiers. Together with an arbitration rule, the learned arbiter resolves conflicts among the classifiers when necessary.

5.2.3 Combiner strategies

In the *combiner* strategy, the predictions of the learned base classifiers on the training set form the basis of the meta-learner's training set. A *composition rule*, which varies in different schemes, determines the content of training examples for the meta-learner. From these examples, the meta-learner generates a meta-classifier, that is called *combiner*. In classifying an instance, the base classifiers first generate their predictions. Based on the same composition rule, a new instance is generated from the predictions, which is then classified by the combiner (see Figure 5.3).

The aim of the *combiner* strategy is to coalesce the predictions from the base classifiers by learning the relationship between these predictions and the correct prediction. For example, a base classifier might consistently make the correct predictions for class c; i.e. when this base classifier predicts class c, it is probably correct regardless of the predictions made by the other base classifiers.
Note that a combiner computes a prediction that may be entirely different from any proposed by a base classifier, whereas an arbiter chooses one of the predictions from the base classifiers and the arbiter itself.

Chan & Stolfo experimented with two schemes for the composition rule. First, the predictions, $C_1(x), C_2(x), \ldots, C_k(x)$, for each example x in the validation set of examples, E, are generated

Class	Attribute vector	Example	Base classifiers' predictions		
$class(x)$	$attribute\_vector(x)$	x	$C_1(x)$	$C_2(x)$	$C_3(x)$
table	$attrvec_1$	x_1	table	table	table
chair	$attrvec_2$	x_2	table	chair	lamp
lamp	$attrvec_3$	x_3	lamp	chair	table

Training set for the *arbiter* scheme		
Instance	Class	Atribute vector
1	chair	$attrvec_2$
2	lamp	$attrvec_3$

Training set for the *class-combiner* scheme		
Instance	Class	Atribute vector
1	table	(table,table,table)
2	chair	(table,chair,lamp)
3	lamp	(lamp,chair,table)

Training set for the *class-attribute-combiner* scheme		
Instance	Class	Atribute vector
1	table	(table,table,table,$attrvec_1$)
2	chair	(table,chair,lamp,$attrvec_2$)
3	lamp	(lamp,chair,table,$attrvec_3$)

Figure 5.2: Sample training sets generated by the *arbiter* and *combiner* strategies with three base classifiers.

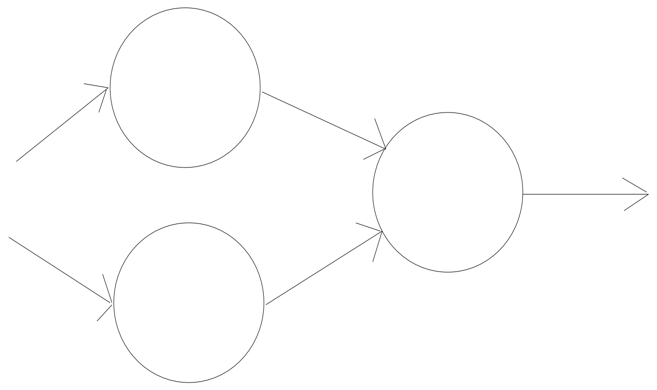

Figure 5.3: A combiner with two classifiers.

by the k base classifiers. These predicted classifications are used to form a new set of "meta-level training instances" T, which is used as input a learning algorithm that computes a combiner. The manner in which t is computed varies as defined below. In the following definitions, $class(x)$ and $attribute\_vector(x)$ denote the correct classification and attribute vector of example x as specified in the validation set E.

1. Return meta-level training instances with the correct classification and the predictions; i.e. $T = \{(class(x), C_1(x), C_2(x), \ldots, C_k(x)) | x \in E\}$. This scheme is denoted as *class-combiner*.

2. Return meta-level training instances as in *class-combiner* with the addition of the attribute vectors; i.e. $T = \{(class(x), C_1(x), C_2(x), \ldots, C_k(x), attribute\_vector(x)) | x \in E\}$. This scheme is denoted as *class-attribute-combiner*.

Figure 5.2 presents sample training sets for these two combiner schemes.

5.2.4 Arbiter trees

An *arbiter tree* is a hierarchical structure composed of arbiters that are computed in a bottom-up, binary-tree fashion. (The choice of a binary tree is to simplify the discussion). An arbiter is learned from the output of a pair of learned classifiers and recursively, an arbiter is learned from the output of two arbiters. A binary tree of arbiters (called an *arbiter tree*) is generated with the initially learned base classifiers at the leaves.

When an instance is classified by the arbiter tree, predictions flow from the leaves to the root. First, each of the leaf classifiers produces an initial prediction; i.e. a classification of the test instance. From a pair of predictions and the parent arbiter's prediction, a prediction is produced by an *arbitration rule*. This process is applied at each level until a final prediction is produced at the root of the tree.

Now, a detailed description of how to build an arbiter tree is given.
Suppose there are initially four training data subsets ($T_1 - T_4$), processed by some learning algorithm, L. First, four classifiers ($C_1 - C_4$) are generated from four instances of L applied to $T_1 - T_4$. The union of the subsets T_1 and T_2, U_{12}, is then classified by C_1 and C_2, which generates two sets of predictions, P_1 and P_2. A *selection rule* as detailed earlier generates a training set (T_{12}) for the arbiter from the predictions P_1 and P_2, and the subset U_{12}. The arbiter (A_{12}) is then trained from the set T_{12} by algorithm L. Similarly, arbiter A_{34} is generated from T_3 and T_4 and hence all the first-level arbiters are produced. Then U_{14} is formed by the union of the subsets T_1 through T_4 and is classified by the arbiter trees rooted with A_{12} and A_{34}. Similarly, T_{14} and A_{14} (root arbiter) are generated and the arbiter tree is complete. The resultant tree is depicted in Figure 5.4.

This process can be generalized to arbiter trees of higher order. The higher the order is, the shallower the tree becomes. In a parallel environment this translates to faster execution. However, there will logically be an increase in the number of disagreements (and hence data items selected for training) and higer order communication overhead at each level in the tree due to the arbitration of many more predictions at a single site.

5.2.5 Combiner trees

The way combiner trees are learned and used is very similar to arbiter trees. A combiner tree is trained bottom-up. A combiner, instead of an arbiter, is computed at each non-leaf node of a combiner tree. To simplify the discussion here, a description of how a *binary* combiner tree is used and trained is given.

To classify an instance, each of the leaf classifiers produces an initial prediction. From a pair of predictions, the composition rule is used to generate a meta-level instance, which is then classified

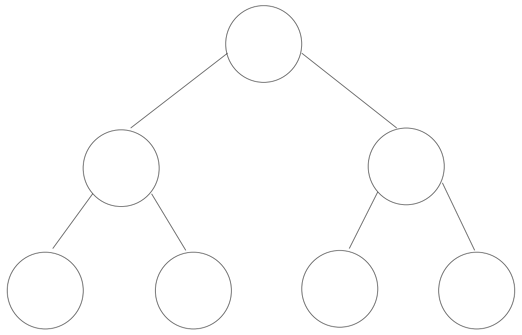

Figure 5.4: Sample binary arbiter tree.

by the parent combiner. This process is applied at each level until a final prediction is produced at the root of the tree.

5.2.6 Benefits of meta-learning

Meta-learning improves efficiency by executing in parallel the base-learning processes (each implemented as a distinct serial program) on (possibly disjoint) subsets of the training data set (a *data reduction technique*). This approach has the advantage, first, of using the same serial code without time-consuming process of parallelizing it, and second, of learning from small subsets of data that fit in main memory.

Meta-learning improves predictive performance by combining different learning systems each having different *inductive bias* (e.g. representation, search heuristics, search space). By combining seperately learned concepts, meta-learning is expected to derive a higher level learned model that explains a large database more accurately than any of the individual learners. Futhermore, meta-learning constitutes a scalable machine learning method since it can be generalized to hierarchical multi-level meta-learning.

5.2.7 Experiments

Attacking the scaling problem by data reduction has a negative impact on accuracy. However, among the combining schemes, empirical results in [2][3] show that *meta-learning* strategies can outperform *simple* or *weighted* voting. When using simple voting a final prediction, based on the predictions of different base classifiers, is chosen as the classification with a plurality of votes. A variation of simple voting is weighted voting. Each classifier is associated with a weight, which is determined by how accurate the classifier performs on a validation set.

The results also demonstrate that training on randomly sampled subsets of examples, without combining, is definitely not sufficient to maintain high accuracy.

The one-level meta-learning schemes, as specified in subsection 5.2.2 and 5.2.3, cannot generally achieve the goal of maintaining the accuracy across different numbers of subsets. However, with the hierarchical meta-learning approach as specified in subsection 5.2.4 and 5.2.5, this problem can be solved. Results form their investigation show that this approach is viable in sustaining the same level of accuracy as a base classifier given the entire data set. Furthermore, the techniques are also data and algorithm-independent, which enable any learning algorithm to train on large data sets.

Further, the combiner tree strategy can consistently boost the predictive accuracy of a global classifier under certain circumstances. This suggests that a properly configured meta-learning strategy combining multiple knowledge sources provides a more accurate view of all available data than any one learning algorithm alone can achieve.

5.3 Solving the credit card fraud detection problem with meta-learning

5.3.1 The fraud learning task

The learning task is quite straightforward. Given a set of "labeled transactions", $T = \{t|t =< f_1, \ldots, f_n >\}$, compute a model or classifier, C, by some learning algorithm L, that predicts from the features $< f_1, \ldots, f_{n-1} >$ the target class label f_n, "fraud" or "legitimate". Hence, $C = L(T)$, where L is a learning algorithm. Each element $t \in T$ is a vector of features, where we denote f_1 as the "transaction amount" ($tranamt$), and f_n as the target class label, denoted $fraud(t) = 0$ (legitimate transaction) or 1 (a fraudulent transaction). Given a "new unseen" transaction, x, with unknown class label, we compute $f_n(x) = C(x)$. C serves as our fraud detector.

In meta-learning, we first seek to compute a set of base classifiers, $\{C_i, i = 1, \ldots, m\}$, where $C_i = L_j(T_k)$, $\cup_k T_k = T$, varying the distributions of the training data (T_k) and using a variety of different machine learning algorithms (L_j) in order to determine the "best" strategies for building good fraud detectors. The "best" base classifiers are then combined by a variety of techniques in order to boost performance. One of the combining algorithms is the "class-combiner". A separate hold out training data set, V, is used to generate a meta-level training data to learn a new "classifier" M. M is computed by learning a classifier from training data composed of the predictions of a set of base classifiers generated over a set of validation data (V) along with the true class label. Hence, $M = L(< C_1(v), \ldots, C_m(v), f_n(v) >), v \in V$. The resultant meta-classifier works by inputing the predictions for some unknown into its constituent base classifiers, and then generating its own final class prediction from these base classifier predictions. Thus, for unknown x, $f_n(x) = M(C_1(x), \ldots, C_m(x))$. Notice, M is as well a classifier, or fraud detector.

5.3.2 Credit card datasets

Chase Bank and First Union Bank, members of the Financial Services Technology Consortium (FSTC), provided their researchers with real credit data for their study. The two data sets contain credit card transactions labeled as fraudulent or legitimate. Each bank supplied 500 thousand records spanning one year with 20% fraud and 80% nonfraud distribution for Chase Bank and 15% versus 85% for First Union Bank. In practice, fraudulent transactions are much less frequent than the 15% to 20% observed in the data given to them. A ratio of 1:100 or 1:1000 between fraudulent and legitimate transactions is not unusual. Therefore to evaluate the effectiveness of their techniques under more extreme conditions, they deliberately create more skewed distributions in some of their experiments.

Bank personnel developed the schemata of the databases over years of experience and continuous analyses to capture important information for fraud detection. The researchers didn't reveal the details of the schema. The records of one schema have a fixed length of 137 bytes each and about 30 attributes, including the binary class label (fraudulent/legitimate transaction). Some field are numeric and the rest categorical, i.e. numbers were used to represent a few discrete categories. Because account identification is not present in the data, transactions cannot be grouped into accounts. Therefore, instead of learning behaviour models of individual customer accounts, overall models are built that try to differentiate legitimate transactions from fraudulent ones. The models are customer-independent and can serve as a second line of defense, the first being customer-dependent models.

	Actual Positive (fraudulent)	**Actual Negative (legitimate)**
Predicted Positive	True Positive (Hit)	False Positive ($FalseAlarm$)
Predicted Negative	False Negative ($Miss$)	True Negative ($Normal$)

Table 5.1: Outcome of a transaction.

Outcome	**Cost**
Miss (FN)	$tranamt$
False Alarm (FP)	$overhead$ if $tranamt > overhead$ or 0 if $tranamt \leq overhead$
Hit (TP)	$overhead$ if $tranamt > overhead$ or $tranamt$ if $tranamt \leq overhead$
Normal (TN)	0

Table 5.2: Cost model assuming a fixed overhead.

5.3.3 Cost-based models for fraud detection

Most machine-learning literature concentrates on model accuracy (either training error or generalization error on hold-out test data computed as overall accuracy, true-positive or false-positive rates, or return-on-cost analyses). This domain provides a considerably different metric to evaluate the learned models' performance: models are evaluated and rated by a cost model. Due to the different dollar amount of each transaction and other factors, the cost of failing to detect a fraud varies with each transaction. Hence, the cost model for this domain relies on the sum and average loss caused by fraud. Hence define:

$$AverageAggregateCost = \frac{1}{n}\sum_{i=1}^{N} Cost(i),$$

where $Cost(i)$ is the cost associated with instance i and N is the total number of instances. Since it takes time and personnel to investigate a potentail fraudulent transaction, an *overhead* is incurred for each investigation. That is, if the amount of a transaction is smaller than the overhead, the transaction is not investigated even if it is suspicious. Therefore, assuming a fixed *overhead* and considering Table 5.1, they devised the cost model for each transaction given in Table 5.2, where *tranamt* is the amount of the credit card transaction.

5.3.4 A multi-classifier and meta-learning approach to skewed distributions

As mentioned earlier, using the natural class distribution might not yield the most effective classifiers (particularly when the distribution is highly skewed). Given a skewed distribution, it is possible to generate the desired distribution without removing any data. One approach is to create data subsets with the desired distribution, generate classifiers from the subsets, and integratre them by learning (meta-learning) from their classification behaviour. In their fraud domain, the original skewed distribution is 20:80 and the desired distribution is 50:50 (which was empirically shown by them). The majority instances are divided into 4 partitions and 4 data subsets are formed by merging the minority instances with each replicated across 4 data subsets to generate the desired 50:50 distribution. Figure 5.5 depicts this process.
Formally, let n be the size of the data set with a distribution $x : y$ (x is the percentage of the minority class) and $u : v$ be the desired distribution. Then the number of minority instances is n x x and the desired number of majority instances in a subset is nx x $\frac{v}{u}$. The number of subsets is the number of majority instances (n x y) divided by the number of desired majority instances in each subset, which is $\frac{ny}{\frac{nxv}{u}}$ or $\frac{y}{x}$ x $\frac{u}{v}$. (When it is not a whole number, take the ceiling ($\lceil \frac{y}{x}$ x $\frac{u}{v} \rceil$) and replicate some majority instances to ensure all of the majority instances are in the subsets).

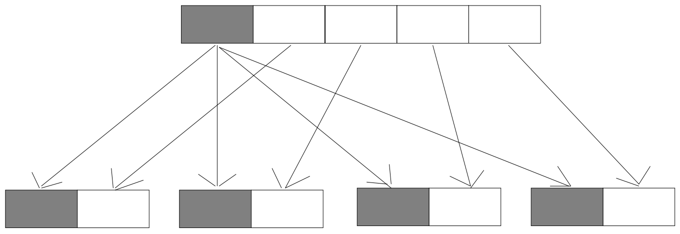

Figure 5.5: Generating four 50:50 data subsets from a 20:80 data set.

That is, we have $\frac{y}{x}$ x $\frac{u}{v}$ subsets, each of which has nx minority instances and $\frac{nxv}{u}$ majority instances.

The next step is to apply a learning algorithm(s) to each of the subsets. Since the subsets are independent, the learning process for each subset can be run in parallel on different processors. For massive amounts of data, substantial improvement in speed can be achieved for super-linear-time learning algorithms.

5.3.5 Experiments

Chan & Stolfo experimented with the credit card fraud detection data as described in subsection 5.3.2 and used the cost model as described in section subsection 5.3.3. For the transactions, they used the first 8 months (10/95 - 5/96) for training, the ninth month (6/96) for validating and the twelfth month (9/96) for testing (the two-month gap is chosen according to the amount of time needed to completely determine the legitimacy of transactions). Since the natural distribution is 20:80 and the desired distribution is 50:50, four subsets are generated from each month for a total of 32 subsets. They applied the four learning algorithms C4.5, CART, RIPPER and BAYES to each subset and generated 128 base classifiers. BAYES was used to train the meta-classifier. They experimented with different amounts of overhead and based on the experiments they made the following conclusions:

- The training class distribution affects the performance of the learned classifiers.
- Removing transactions with amounts lower than the overhead can shorten training time, but not without sacrificing some cost performance.
- The multi-classifier meta-learning approach using a 50:50 distribution in the data subsets for training can significantly reduce the amount of loss due to illegimate transactions. The subsets are independent and can be processed in parallel. Training time can furter be reduced by also using a 50:50 distribution in the *validation* set without degrading the cost performance. That is, this approach provides a means for efficiently handling learning tasks with skewed class distributions, non-uniform cost per error, and large amounts of data. Not only is the method efficient, it is also scalable to larger amounts of data.

5.3.6 JAM project

The *main objective* of Chan & Stolfo's research is to take advantage of the inherent parallellism and distributed nature of meta-learning and design and implement a powerful and practical distributed data mining system. Assuming that a system consists of several databases interconnected through an intranet or internet, the goal is to provide the means for each data site to utilize its own local data and, at the same time, benefit from the data that is available at other data sites without transferring or directly accessing that data. In this context, this can be materialized by learning agents that execute at remote data sites and generate classifier agents that can subsequently be transfered among the sites. This goal can be achieved through the implementation of the system called JAM (Java Agents for Meta-Learning). The JAM system is the result of the JAM project which includes researchers at *Columbia University, Florida Tech* and project management and technical experts at several banks who are members of the *Financial Services Technology Consortium (FSTC)* including Chase, First Union and Citibank.

JAM has been used in several experiments dealing with real-world learning tasks, such as solving crucial problems in fraud detection in financial information systems. The objective is to employ pattern-directed inference systems using models of fraudulent transaction behaviours to detect fraud. This approach requires analysis of large and inherently (e.g. from different banks) distributed databases of information about transaction behaviours to produce models of "probably fraudulent" transactions.

JAM is used to compute local fraud detection agents that learn how to detect fraud and provide intrusion detection services within a single information system; JAM provides an integrated meta-learning system that combines the collective knowledge acquired by individual local agents from among participating bank sites. Once derived local classifier agents or models are produced at some datasite(s), two or more such agents may be composed into a new classifier agent by JAM's meta-learning agents. JAM allows financial institutions to share their models of fraudulent transactions by exchanging classifer agents in a secured agent infrastructure. But they will not need to disclosure their proprietary data. In this way their competive and legal restrictions can be met, but they can still share information. The meta-classifiers then act as sentries forewarning of possibly fraudulent transactions and threats by inspecting, classifying and labeling each incoming transaction.

The main result of the JAM project is that combining multiple fraud models by distributed data mining over different transaction record sources (including multiple banks) results in better performance. The key difficulty is that financial companies avoid sharing their data for a number of (competitive and legal) reasons [19].

The JAM software can freely be downloaded from:

`http://www.cs.columbia.edu/~sal/JAM/PROJECT`

For a description of the architecture of JAM, see [15].

5.4 Solving the credit card fraud detection problem with boosting

5.4.1 Introduction

Fan et al. [7] propose adapted versions of the AdaBoost algorithm for scalable and distributed learning, where each weak classifier is not trained from the same whole data set at each round but only from a small portion of the training set. Further, they propose an on-line learning AdaBoost algorithm, which can be handy when new data become available periodically.

For classification problems with no fixed misclassification cost per error type, such as the credit card fraud detection problem, Fan et al. [6] proposes a cost-sensitive boosting algorithm, which they call the AdaCost algorithm. AdaCost is a variant of AdaBoost that modifies its "weight updating rule" by a "cost based factor". The purpose of AdaCost is to reduce the cumulative misclassification cost more than AdaBoost.

5.4.2 AdaBoost for scalable, distributed and on-line learning

Scalable and distributed learning

In AdaBoost, the weak learner is treated as a "black-box". There is not much control over it except for observing its accuracy and providing it with a different training sample at each round according to its accuracy in previous rounds. Each weak learner may freely choose examples in the sample given to it. Conversely, the weak learner can also be given a sample which is just a small "portion" of the complete training set. The weak classifier produced from the small portion of the training set is very likely "weaker" than one generated from the entire weighted training set, but the overall accuracy of the voted ensemble can still be boosted.

In order to increase accuracy, the small portion should not be a highly skewed sample of the complete training set. Two methods are proposed.

In r-sampling (Figure 5.6, r is $r$andom), a fixed number (n) of examples are randomly picked from the weighted training set (without replacement) together with their assigned weights. All examples have equal chance of being selected, and D_t is not taken into account in this selection. At different rounds, a new r-sample is picked.

In d-sampling (Figure 5.7, d is $d$isjoint), the weighted training set is partitioned into p disjoint subsets. Each subset is a d-sample. At each round, a different d-sample is given to the weak learner.

The weights of the chosen examples of both r- and d-sample are re-normalized to make them a distribution, D_t^*. A weak classifier is generated from these examples with distribution D_t^*. The update of weights and calculation of α_t are performed on the entire data set with distribution D_t. Both methods can be used for learning over very large data sets, but d-sampling is more suitable for distributed learning where data at each site cannot be culled together to a single site. The data at each site are taken as a d-sample. Weak learning at each round is carried out independently of all the local data of each site. On the other hand, r-sampling would choose data from different sites and transfer that to a single site. This is not possible for some applications and inefficient in general.

On-line learning

In on-line learning, there is a steady flow of new instances generated periodically or in real-time that ought to be incorporated to correct or update the model previously learned. This can be done by an on-line learning scheme using AdaBoost as a means to re-weight classifiers in an ensemble, and thus to reuse previously computed classifiers along with new classifier computed on a new increment of data.

The basic idea is to reuse previous weak classifiers $\{h_1, \ldots, h_{T-1}\}$ and learn a new classifier from the weighted new data (Figure 5.8). When the increment S_T arrives, the weight updating rule to both "re-weight" previous classifiers based on their accuracy on S_T is used and a weighted training set is generated. Data items that are not accurately predicted by these hypotheses receive higher weights. A new classifier h_T is trained from this weighted increment. At the end of this procedure, we have $\alpha_1, \ldots, \alpha_{T-1}$ for $h_1, \ldots, h_{T-1}$, a new classifier h_T and its weight α_T calculated from the weighted training set S_T during successive updates of the distribution. The interesting observation is that AdaBoost provides a means of assigning weights to classifiers based on validation set that these classifiers are not necessarily trained from. It also identifies data items that these classifiers perform poorly on and generates a weigted increment accordingly.

One problem with the above on-line learning method is that all the classifiers previously learned must be retained. This naturally increases the memory requirements and slows the learning and classifying procedures. To solve this problem, a "window" of a fixed number of classifiers can be used. Instead of retaining all classifiers, only the k most recent classifiers that reflect the most recent on-line data are used and stored. the value of k can either be fixed or change according to the amount of resource and accuracy requirement. The number of new data items accumulated on-line before on-line learning starts can be fixed or change at runtime as well.

r-sampling AdaBoost

- Given: $S = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, $x_i \in X$, $y_i \in \{-1, +1\}$ and sample size n.
- Initialize $D_1(i)$ (such as $D_1(i) = \frac{1}{N}$).
- For $t = 1, 2, \ldots, T$:
  1. Randomly choose n examples from S without replacement.
  2. Re-normalize their weights into a distribution, D_t^*.
  3. Train weak learner using distribution D_t^* for the chosen n examples.
  4. Compute weak hypothesis $h_t : X \mapsto \mathbb{R}$.
  5. Update

     $$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_t h_t(x_i))}{Z_t}$$

     for the complete training set S.
- Output the final hypothesis:

$$H(x) = \mathrm{sign}(F(x)) \text{ , where } F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

Figure 5.6: r-sampling AdaBoost Algorithm.

d-sampling AdaBoost

- Given: $S = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, $x_i \in X$, $y_i \in \{-1, +1\}$, sample size n and partition number p.
- Initialize $D_1(i)$ (such as $D_1(i) = \frac{1}{N}$).
- Divide S into p partitions $\{S_0, \ldots, S_{p-1}\}$.
- For $t = 1, 2, \ldots, T$:
  1. Re-normalize the weight of partition $S_{t \mathrm{mod} p}$ to make it a distribution, D_t^*.
  2. Train weak learner using distribution D_t^*.
  3. Compute weak hypothesis $h_t : X \mapsto \mathbb{R}$.
  4. Update

     $$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_t h_t(x_i))}{Z_t}$$

     for the complete training set S.
- Output the final hypothesis:

$$H(x) = \mathrm{sign}(F(x)) \text{ , where } F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

Figure 5.7: d-sampling AdaBoost Algorithm.

On-line AdaBoost

- Given: $S = \{(x_1, y_1), \dots, (x_N, y_N)\}$, $x_i \in X$, $y_i \in \{-1, +1\}$, sample size n, partition number p, $\{h_1, \dots, h_{T-1}\}$ and $S_T(|S_T| = N_T)$.
- For $t = 1, 2, \dots, T-1$:
  1. Choose $\alpha_t \in \mathbb{R}$.
  2. Update

$$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_t h_t(x_i))}{Z_t}$$

  the new increment S_T.

- Train weak learner using distribution D_T.
- Compute weak hypothesis $h_T : X \mapsto \mathbb{R}$.
- Output the final hypothesis:

$$H(x) = \text{sign}(F(x)) \text{ , where } F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

Figure 5.8: On-line AdaBoost Algorithm.

This on-line learning scheme with window size of k is efficient. The extra overhead involves k classifications of the increment with weight updating and $k+1$ computations of α. The predicting and weight updating procedure is linear in the size of the data increment. The computation of α has a bounded number of linear computations.

Experiments

Fan et al. tested the algorithms on 4 data sets ADULT, WHIRL, BOOLEAN and CHASE. ADULT is a population census data from the UCI machine learning database. WHIRL is an information extract data set that trains wrappers to correctly extract relevant information from a web page. BOOLEAN is an artificial boolean data set that has 15 boolean variables. (The data item is positive if 4 or 5 variables are true or otherwise negative). CHASE is a credit card fraud detection data set, courtesy of Chase Bank.

In the experiments they used Cohen's RIPPER as the "weak" learner. Based on the experiments with the data set, they made some conclusions. The conclusions are as follows.

- When d- and r-sampling are used for scalable and distributed learning, the results are in most cases comparable to or better than learning a global classifier from the complete training set and in many cases comparable to boosting the global classifier on the complete data set. However, the cost of learning and the requirements for memory are significantly lower.
- When using on-line AdaBoost there is a significant improvement from learning a single classifier on the new increment of data itself and in many caes even better than learning the global classifier where all data participates in learning. But its cost is similar to learning over the new increment data. The storage overhead for all these methods is bounded and can be pre-allocated before runtime. The computation overhead is also limited.

5.4.3 AdaCost: A cost-sensitive boosting algorithm

In 1999 Fan et al. [6] proprosed a cost-sensitive version of the (Real) AdaBoost algorithm: the AdaCost algorithm. The purpose is to reduce the cumulative misclassification cost (rate) more than (Real) AdaBoost. The cumulative misclassification cost rate is defined by:

$$\text{cumulative misclassification cost rate} = \frac{\text{misclassification cost}}{\text{maximal misclassification cost}},$$

where maximal misclassification cost is the cost of misclassifying all instances.
For the analyses of the AdaCost algorithm, the generalized analyses of RealAdaBoost will be followed.

Let $S = \{(x_1, c_1, y_1), \ldots, (x_N, c_N, y_N)\}$ be a sequence of training examples where each *instance* x_i belongs to a *domain* or *instance space* X, each *cost factor* c_i belongs to the *non-negative real domain* $\mathbb{R}_+$ and each *label* y_i belongs to a finite *label space* Y. We focus on binary classification problems in which $Y = \{-1, 1\}$. h is a weak hypothesis. It has the form $h : X \mapsto \mathbb{R}$. The sign of $h(x)$ is interpreted as the predicted label and the magnitude $|h(x)|$ is the "confidence" in this prediction. Let t be index to show the round of boosting and $D_t(i)$ be the weight given to (x_i, c_i, y_i) at the t-th round. $0 \leq D_t(i) \leq 1$ and $\sum_i D_t(i) = 1$. α_t is a chosen parameter as weight for weak hypothesis h_t at the t-th round. We assume $\alpha_t > 0$. $\beta(i) = \beta(\text{sign}(y_i, h_t(x_i)), c_i)$ is a cost adjustment function with two arguments: $\text{sign}(y_i, h_t(x_i))$ to show if $h_t(x_i)$ is correct, and the cost factor c_i.

A detailed description of the AdaCost algorithm is shown in Figure 5.9.

Cost function

The difference between AdaCost and AdaBoost is the additional cost adjustment function $\beta(\text{sign}(y_i, h_t(x_i)), c_i)$ in the weight updating rule. We will use β_i as a shorthand for

AdaCost (Fan et al. 1999)

- Given: $S = \{(x_1, c_1, y_1), \ldots, (x_N, c_N, y_N)\}$, where $x_i \in X, c_i \in \mathbb{R}^+, y_i \in Y = \{-1, 1\}$.
- Initialize $D_1(i)$ such as $D_1(i) = c_i / \sum_{j=1}^{N} c_j \; (i = 1, \ldots, N)$.
- For $t = 1, 2, \ldots, T$:
  1. Train weak learner using distribution D_t.
  2. Compute weak hypothesis $h_t(x) : X \mapsto \mathbb{R}$.
  3. Choose $\alpha_t \in \mathbb{R}$ and $\beta(i) \in \mathbb{R}^+$.
  4. Update:

$$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i)\beta(i))}{Z_t}, (i = 1, 2, \ldots, N),$$

  where $\beta(i) = \beta(\text{ sign}(y_i h_t(x_i)) \text{ , } c_i \text{)}$ is a cost-adjustment function.
  Z_t is a normalization fator (chosen so that D_{t+1} will be a distribution).

- Output the final hypothesis:

$$H(x) = \text{sign}(F(x)) \text{ , where } F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

Figure 5.9: AdaCost Algorithm.

$\beta(\text{sign}(y_i, h_t(x_i)), c_i)$. Furthermore we use β_+ when $\text{sign}(y_i, h_t(x_i)) = +1$ and β_- when $\text{sign}(y_i, h_t(x_i)) = -1$.

Fan et al. uses the following intuitive approach for the cost function $\beta(i)$: for an instance with a higher cost factor, $\beta(i)$ increases its weights "more" if the instance is misclassified, but decreases its weights "less" otherwise. Therefore, the following requirements must be met:

$\beta_-(c_i)$ is non-decreasing with respect to c_i ,
$\beta_+(c_i)$ is non-increasing with respect to c_i ,
$\beta_-(c_i)$, $\beta_+(c_i) \geq 0$.

For example, the following cost-function meets the requirements:
$\beta_-(c_i) = +0.5 \cdot c_i + 0.5$,
$\beta_+(c_i) = -0.5 \cdot c_i + 0.5$,
where each c_i is normalized to $[0, 1]$.

Training misclassification cost upper bound

Fan et al. derived the choice of α_t in the case that h_t has range [-1,+1] and $\beta(i)$ has range [0,+1] and proved the following corollary. (See Appendix A for the derivations).

Corollary 2 *Assuming h_t has range $[-1, +1]$, $\beta(i)$ has range $[0, +1]$ and α_t is chosen as*

$$\alpha_t = \frac{1}{2}\ln\Big(\frac{1+r_t}{1-r_t}\Big), \quad \text{where} \quad r_t = \sum_{i=1}^{N} D_t(i) y_i h_t(x_i) \beta(i).$$

The training cumulative cost of H is at most:

$$d \prod_{t=1}^{T} \sqrt{1 - r_t^2},$$

where $\quad d = \sum_{j=1}^{N} c_j$.

The upper bound on misclassification cost is actually reduced when $\sqrt{1 - r_t^2} < 1$.

Experiments with AdaCost

We would like to know what the results of the AdaCost algorithm are compared with the results of the AdaBoost algorithm in the performance on the cumulative misclassification cost rate. The comparison can be made for "German Credit Data" and "Heart Disease" as described in subsection 4.2.2 and 4.2.3, because they contain a cost matrix. In the experiments we shall also give the results of AdaCost with a uniform initial distribution ($D_1(i) = 1/N$ instead of $D_1(i) = c_i / \sum_{j=1}^{N} c_j$) and AdaBoost with cost-sensitive initial weights ($D_1(i) = c_i / \sum_{j=1}^{N} c_j$ instead of $D_1(i) = 1/N$). We have chosen the cost function $\beta(i)$ the same as the example cost function in the heading "Cost function" (see above). We shall again use 5-fold cross-validation.
It is amazing that in the experiments it is not necessary to run more than 50 rounds for the algorithms. For the German Credit Data the computation time is only 2 minutes and for the Heart Disease data set even less.

German Credit Data

In Figure 5.10 the results of the experiments are shown. Note that classifying all clients as good client will result in a cumulative misclassifcation cost rate of 0.7386 and classifying all clients as good client will result in a cumulative misclassifcation cost rate of 0.2614 . The "base cost" is the mimimum of these two values and is therefore equal to 0.2616. The minimal misclassification cost rate is:

AdaCost - 0.3196 (9th iteration)
AdaCost with uniform initial weights - 0.3142 (7th iteration)
AdaBoost with cost-sensitive initial weights - 0.3266 (14th iteration)
AdaBoost - 0.3294 (34th iteration)

so the AdaCost algorithm reduces cumulative misclassification cost more than AdaBoost. But the minimal misclassification cost rate we could find was 0.3142 and this is still higer than the base cost. So, for this dataset it is better not to use any of the algorithms.

Heart Disease

In Figure 5.11 the results of the experiments are shown. Note that classifying all persons with presence of a heart disease will result in a cumulative misclassifcation cost rate of 0.20 and classifying all persons with absence of a heart disease will result in a cumulative misclassifcation cost rate of 0.80 . So, the base cost is 0.20 . From Figure 5.11 we see that AdaCost has a minimal cumulative misclassification cost rate of 0.1112 and AdaBoost has a minimal cumulative misclassification cost rate of 0.1407. So, for this data set AdaCost reduces cumulative misclassification cost more than AdaBoost.

Experimental results of boosting in the literature

Fan et al. [6] showed empirically that for the real world credit card fraud detection data set from Chase Bank as described in subsection 5.3.2 with the cost model in subsection 5.3.3, the AdaCost algorithm show significant reduction in the cumulative misclassification cost over the AdaBoost algorithm without consuming additional computing power. In the experiments they used various values for the overhead. Further, they used Cohen's RIPPER as the weak learning algorithm. For the boosting algorithms they used 50 iterations and applied 10-fold cross validation to average the results.

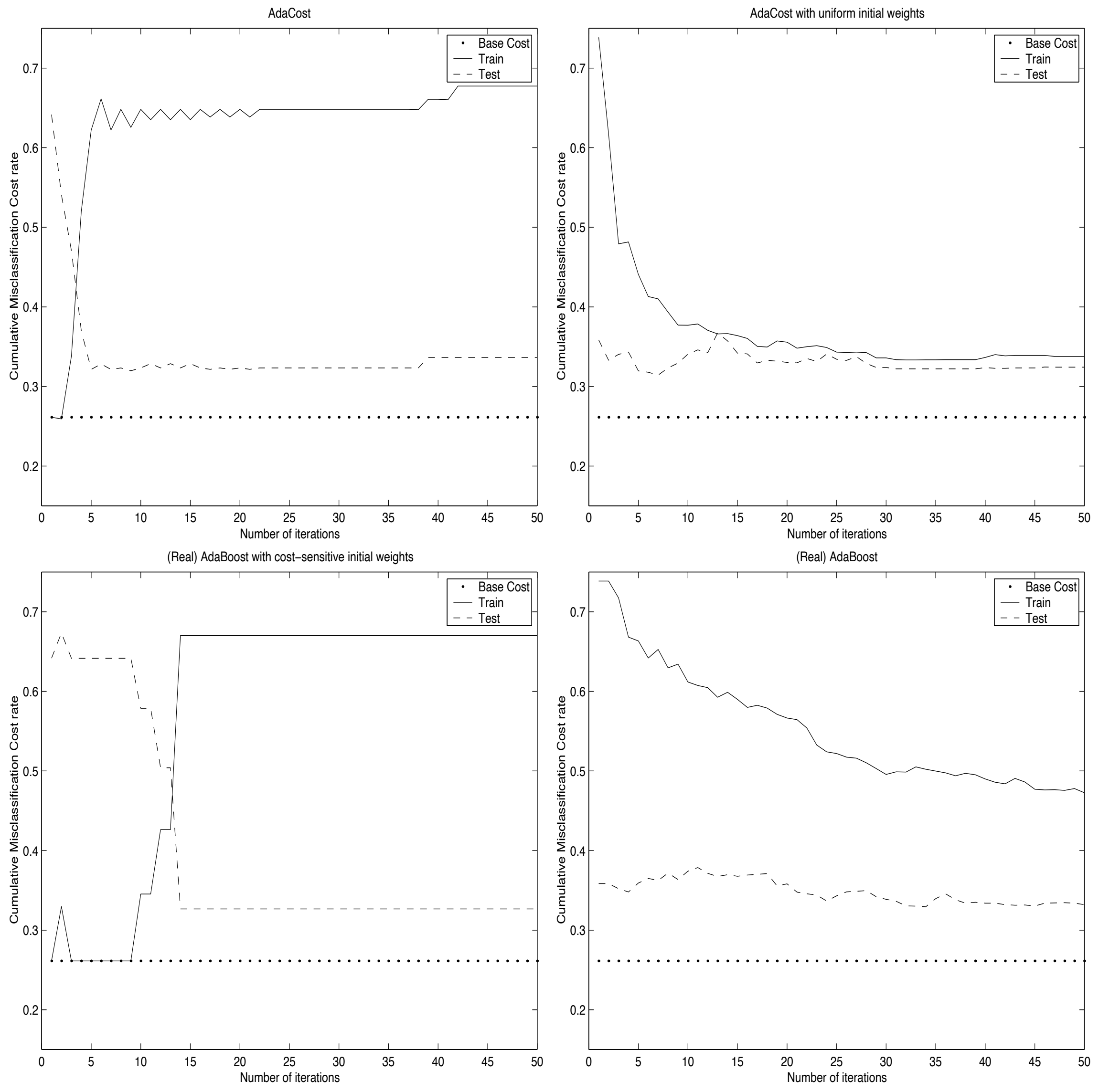

Figure 5.10: Cumulative Misclassification Cost rate with (Real) AdaBoost and AdaCost of German Credit Data.

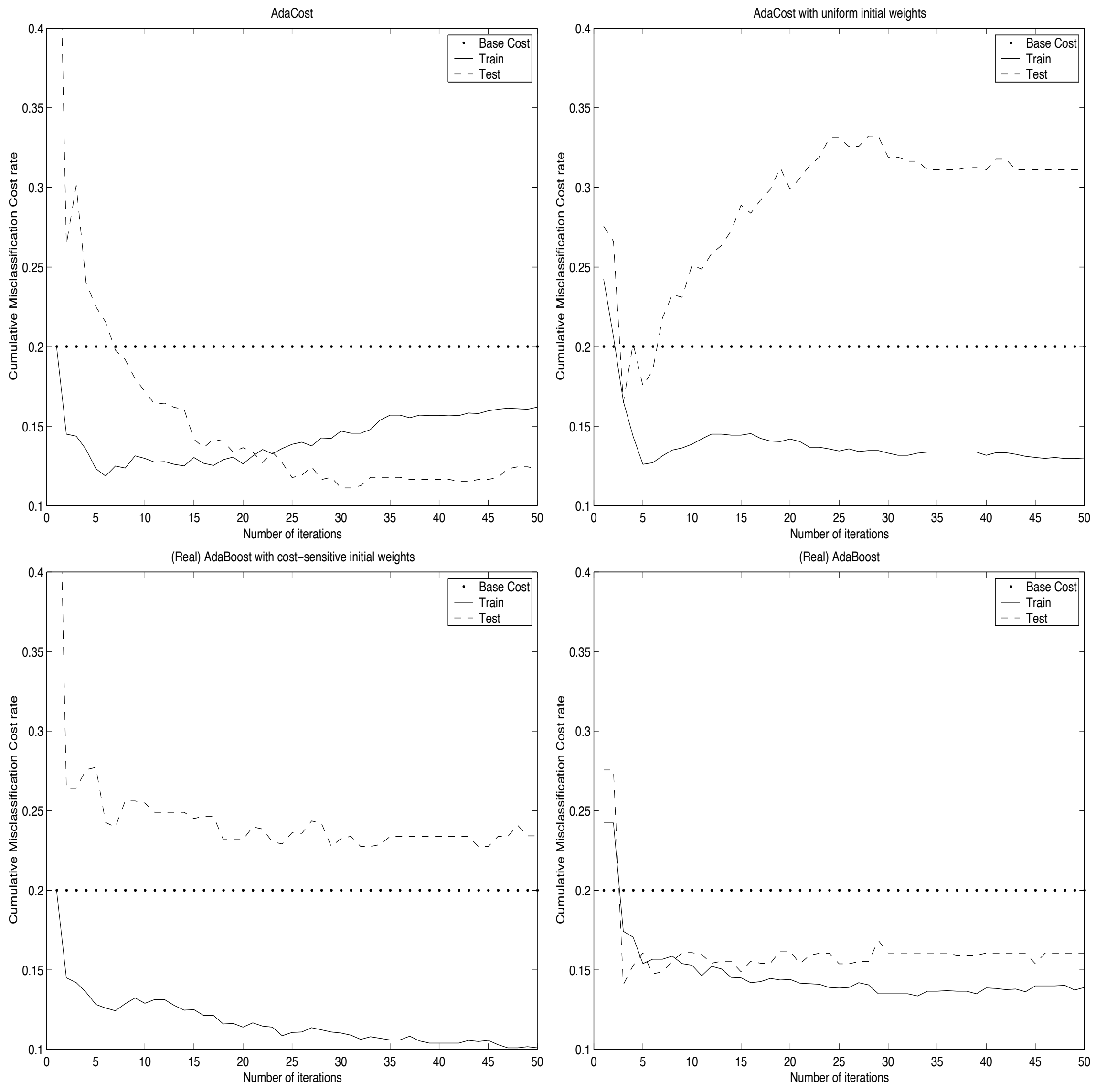

Figure 5.11: Cumulative Misclassification Cost rate with (Real) AdaBoost and AdaCost of Heart Disease.

Chapter 6

Conclusions

“Boosting” is one of the most important recent developments in classification methodology. Boosting works by sequentially applying a classification algorithm to reweighted versions of the training data, and then taking a weighted majority vote of the sequence of classifiers thus produced.

The most popular boosting algorithm in the literature is the AdaBoost algorithm. According to the literature, AdaBoost has many advantages [16]. The basic idea of it is easy to understand. It has no parameters to tune (except for the number of rounds). It requires no prior knowledge about the weak learner and so can be flexibly combined with *any* method for finding weak hypotheses. Finally, it comes with a set of theoretical guarantees given sufficient data and a weak learner that can reliably provide only moderately accurate weak hypotheses. So, instead of trying to design a learning algorithm that is accurate over the entire space, the focus can be on finding weak learning algorithms that only need to be better than random.

A nice property of AdaBoost is its ability to identify *outliers*, i.e., examples that are either mislabeled in the training data, or which are inherently ambiguous and hard to categorize. Because AdaBoost focuses its weight on the hardest examples, the examples with the highest weight often turn out to be outliers. When the number of outliers is very large, the emphasis placed on the hard examples can become detrimental to the performance of AdaBoost. In this case, a variant of AdaBoost, called “Gentle AdaBoost” which puts less emphasis on outliers, is a better option.

The disadvantage of AdaBoost is that the actual performance on a particular problem is clearly dependent on the data and the weak learner. Consistent with the theory, boosting can fail to perform well given insufficient data, overlay complex weak hypotheses or weak hypotheses which are too weak.

Logitboost, which is a result of applying statistical principles on AdaBoost, generally gives slightly better results than AdaBoost, but it is harder to understand.

In the credit card fraud detection domain, experiments by Chan & Stolfo have shown that their meta-learning approach show a significant improvement over the results obtained by using standard commercial fraud detection systems. One of the disadvantage of their approach is the need to run preliminary experiments to determine the desired distribution of the data, which is needed because data with a very skew distribution might not yield the most effective classifiers. This process can be automated but it is unavoidable since the desired distribution is highly dependent on the cost model and the learning algorithm. Another disadvantage is that in their experiments they use a cost model where lower amount transactions are not investigated. Problems arise when many fraudulent transactions appear to be lower amount transactions.

An alternative approach for the credit fraud detection problem is to use a cost sensitive learning algorithm, such as the “AdaCost” boosting algorithm, instead of an ordinary learning algorithm. Experiments reported in the literature have shown that this algorithm can show significant improvement over the “standard” boosting algorithms when *minimizing misclassification cost* is the performance measure for the problem. Because this approach uses the same cost model as the meta-learning approach, problems arise when many fraudulent transactions appear to be lower amount transactions.

The main question we were interested in, was whether boosting algorithms are useful for solving classification problems.

For small classification problems the boosting algorithms are very useful. Besides the advantages mentioned above, the great advantages are that the results of the boosting algorithms often often outperforms "standard" classifciation algorithms, like Decision trees, k-Nearest Neighbors, Naïve Bayes Classifiers etcetera and that the model seldom suffer from overtraining when boosting algorithms are used. This follows not only from the results of the experiments reported in the literature but also from our own research. Besides the disadvantages mentioned above, a great disadvantage we experienced during our research is that boosting algorithms can require much computation time to receive satisfying results.

For the credit card fraud detection problem, which is considered as a large classification problem, further research is needed because of the following reasons:

- It is not known which of the two approaches results has the most savings in cost. For the meta-learning approach several papers [4][19] mention the cost savings but for the boosting approach the are no papers which mention the actual cost savings.
- There hasn't been much research on using boosting algorithms for scalable, distributed or on-line learning. Fan et al.[7] only experimented with the AdaBoost algorithm and not with the AdaCost algorithm.

Other large classification problems where boosting algorithms can be used (which are not included in this paper), are hand-written digit recognition and text categorization, which is the problem of classifying text documents into categories or classes. Several papers have been written about the experiments with the AdaBoost algorithm (binary and multiple classes) on real data in these domains. The results are very promising.

We conclude this paper by noting that the true practical value of boosting can only be assesed by testing the method on real machine learning problems.

Appendix A

Theoretic results of the boosting algorithms

Real AdaBoost - Analyzing the training error

The most basic property of Real AdaBoost concerns its ability to reduce the training error. To show that Real AdaBoost is indeed a generalization of Discrete AdaBoost when h_t takes only the values of +1 and -1 [17], we must find an α_t that will result in receiving the same (or stronger) upper bound as found in Theorem 1. We shall first give an upper bound for the training error of the final hypothesis:

Theorem 2 *Assuming the notation of Figure 2.2, the following bound holds on the training error of H*

$$\frac{|\{i : H(x_i) \neq y_i\}|}{N} \leq \frac{1}{N} \sum_{i=1}^{N} \exp(-y_i F(x_i)) = \prod_{t=1}^{T} Z_t, \tag{A.1}$$

where

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

Eq. (A.1) suggests that he training error can be reduced most rapidly (in a greedy way) by choosing α_t and h_t on each round to minimize

$$Z_t = \sum_{i=1}^{N} D_t(i) \exp(-\alpha_t y_i h_t(x_i)).$$

In the case of binary hypotheses, this leads to the following choice of α_t:

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

and gives a bound on the training error of

$$\prod_{t=1}^{T} \Big(2\sqrt{\epsilon_t(1-\epsilon_t)}\Big) = \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} \leq \exp\Big(-2\sum_{t=1}^{T} \gamma_t^2\Big),$$

where $\epsilon_t = 1/2 - \gamma_t$. This bound was also found in Theorem 1.

Schapire & Singer [18] derived the choice of α_t in the case that h_t has range [-1,+1] (rather than the values $\{-1, +1\}$). They have done this as follows:
Let $u_i = y_i h_t(x_i)$. For weak hypotheses h_t with range [-1,+1], the choice of α_t can be obtained by approximating Z_t as follows:

$$Z_t = \sum_{i=1}^{N} D_t(i) e^{-\alpha_t u_i} \leq \sum_{i=1}^{N} D_t(i) \Big(\frac{1+u_i}{2} e^{-\alpha_t} + \frac{1-u_i}{2} e^{\alpha_t} \Big). \tag{A.2}$$

This upper bound is valid since $u_i \in [-1, +1]$, and is in fact exact if h_t has range $\{-1, +1\}$ (so that $u_i \in \{-1, +1\}$. Next, α_t can be analytically chosen to minimize the right hand side of Eq. (A.2) giving

$$\alpha_t = \frac{1}{2} \ln \Big(\frac{1+r_t}{1-r_t} \Big), \qquad \text{where} \qquad r_t = \sum_{i=1}^{N} D_t(i) u_i. \tag{A.3}$$

Plugging into Eq. (A.2), this choice gives the upper bound:

$$Z_t \leq \sqrt{1 - r_t^2}.$$

We have thus proved the following corollary of Theorem 2.

Corollary 1 *Using the notation of Figure 2.2, assume each h_t has range $[-1, +1]$ and that we choose*

$$\alpha_t = \frac{1}{2} \ln \Big(\frac{1+r_t}{1-r_t} \Big),$$

where

$$r_t = \sum_{i=1}^{N} D_t(i) y_i h_t(x_i) = \mathbb{E}_{D_t}[y_i h_t(x_i)].$$

Then the training error of H is at most

$$\prod_{t=1}^{T} \sqrt{1 - r_t^2}.$$

AdaCost - Training misclassification cost upper bound

Theorem 3 *The following holds for the upper bound of the training cumulative misclassification cost.* $[[\pi]]$ *returns 1 if predicate* π = *true or 0 otherwise.*

$$\sum_{i=1}^{N} c_i[[H(x_i) \neq y_i]] \leq d \prod_{t=1}^{T} Z_t, \qquad \textit{where} \qquad d = \sum_{j=1}^{N} c_j.$$

Proof See [6].

One consequence of Theorem 3 is that, to reduce training cost, we should seek to minimize Z_t at each round of boosting in the choice of α_t. For weak hypothesis h_t with range [-1,+1] and cost adjustment function $\beta(i)$ in the range [0,+1], the choice of α_t is:

$$\alpha_t = \frac{1}{2}\ln\Big(\frac{1+r_t}{1-r_t}\Big), \qquad \text{where} \qquad r_t = \sum_{i=1}^{N} D_t(i)u_i \ \ , \ \ u_i = y_i h_t(x_i)\beta(i). \tag{A.4}$$

Since $u_i \in [-1,+1]$, the following inequality holds:

$$Z_t = \sum_{i=1}^{N} D_t(i)e^{-\alpha_t u_i} \leq \sum_{i=1}^{N} D_t(i)\Big(\frac{1+u_i}{2}e^{-\alpha_t} + \frac{1-u_i}{2}e^{\alpha_t}\Big).$$

By zeroing the first derivative of the right hand side, we have the choice of α_t in formula (A.4). For this choice of α_t: $Z_t \leq \sqrt{1-r_t^2} \leq 1$.

Corrollary 2 *Assuming* h_t *has range* $[-1,+1]$*,* $\beta(i)$ *has range* $[0,+1]$ *and* α_t *is chosen as in formula A.4. The training cumulative cost of* H *is at most:*

$$d \prod_{t=1}^{T} \sqrt{1-r_t^2},$$

where $\quad r_t = \sum_{i=1}^{N} D_t(i) y_i h_t(x_i)\beta(i) \quad$ *and* $\quad d = \sum_{j=1}^{N} c_j$.

The upper bound on misclassification cost is actually reduced when $Z_t = \sqrt{1-r_t^2} < 1$.

Appendix B

Advanced theoretic results of the boosting algorithms

Lemma 1 $\mathbb{E}(e^{-yF(x)})$ is minimized at

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

Hence

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

Proof
While $\mathbb{E}$ entails expectation over the joint distribution of y and x, it is sufficient to minimize the criterion conditional on x.

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

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

□

Result 1 *The Discrete AdaBoost algorithm builds an additive logistic regression model via Newton-like updates for minimizing* $\mathbb{E}(e^{-yF(x)})$.

Derivation The proof of this result is very lenghty. See [11] for the complete proof.

□

Result 2 *The Real AdaBoost algorithm (population version) fits an additive logistic regression model by stage-wise and approximate optimization of* $J(F) = \mathbb{E}(e^{-yF(x)})$.

Derivation
Suppose we have a current estimate $F(x)$ and seek an improved estimate $F(x)+h(x)$ by minimizing $J(F(x)+h(x))$ at each x.

$$J(F(x)+h(x)) = \mathbb{E}[e^{-yF(x)}e^{-yh(x)}|x)$$

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

Dividing through $\mathbb{E}[e^{-yF(x)}|x]$ and setting the derivative w.r.t. $h(x)$ to zero we get

$$h(x) = \frac{1}{2}\ln\frac{\mathbb{E}_D[1_{[y=1]}|x]}{\mathbb{E}_D[1_{[y=-1]}|x]} = \frac{1}{2}\ln\frac{\mathbb{P}_D(y=1|x)}{\mathbb{P}_D(y=-1|x)},$$

where $D(x,y) = \exp(-yF(x))$. The weights get updated by

$$D(x,y) \leftarrow D(x,y) \cdot e^{-yh(x)}.$$

Careful examination of Schapire & Singer [18] shows that this update matches theirs (and the α_t's in the algorithm are redundant.) The algorithm as presented would stop after one iteration. In practice crude approximation to conditional expectation, such as decision trees or other constrained models is used, and hence many steps are required.

$\square$

Result 3 *The LogitBoost algorithm (2 classes) uses Newton steps for fitting an additive logistic model by maximum likelihood.*

Derivation

Let $y^* = (y+1)/2$, taking values 0 and 1, and parametrize the binomial probabilities by

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

The binomial log-likelihood is

$$l(y^*, p(x)) = y^*\ln(p(x)) + (1-y^*)\ln(1-p(x)) = -\ln(1+e^{-2yF(x)}).$$

Consider the update $F(x) + h(x)$ and the expected log-likelihood

$$\mathbb{E}l(F+h) = \mathbb{E}[2y^*(F(x)+h(x)) - \ln(1+e^{2(F(x)+h(x))})].$$

Conditioning on x, we compute the first and second derivative at $h(x) = 0$:

$$s(x) = \frac{\partial \mathbb{E}l(F(x)+h(x))}{\partial h(x)}|_{h(x)=0} = 2\mathbb{E}(y^* - p(x)|x),$$

$$B(x) = \frac{\partial^2 \mathbb{E}l(F(x)+h(x))}{\partial h(x)^2}|_{h(x)=0} = -4\mathbb{E}(p(x)(1-p(x))|x),$$

where $p(x)$ is defined in terms of $F(x)$. The Newton update is then

$$F(x) \leftarrow F(x) - B(x)^{-1}s(x)$$

$$= F(x) + \frac{1}{2}\frac{\mathbb{E}(y^* - p(x)|x)}{\mathbb{E}(p(x)(1-p(x))|x)}$$

$$= F(x) + \frac{1}{2}\mathbb{E}_D\Big(\frac{y^* - p(x)}{p(x)(1-p(x))}|x\Big),$$

where $D(x) = p(x)(1-p(x))$. Equivalently, the Newton update $h(x)$ solves the weighted lest-squares approximation (about$(F(x))$ to the log-likelihood

$$\min_{h(x)} \mathbb{E}_{D(x)}\Big(H(x) + \frac{1}{2}\frac{y^* - p(x)}{p(x)(1-p(x))} - (F(x)+h(x))\Big)^2.$$

$\square$

Result 4 *The Gentle AdaBoost algorithm uses Newton steps for minimizing* $\mathbb{E}e^{-yF(x)}$.
Derivation

$$\frac{\partial J(F(x)+h(x))}{\partial h(x)}|_{h(x)=0} = -\mathbb{E}(e^{-yF(x)}y|x),$$

$$\frac{\partial^2 J(F(x)+h(x))}{\partial h(x)^2} = \mathbb{E}(e^{-yF(x)}|x) \text{ since } y^2 = 1.$$

Hence the Newton update is

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

where $D(x,y) = e^{-yF(x)}$.

□

Result 5 *The AdaBoost algorithm for a J-class problem fits J uncoupled additive logistic models,* $G_j(x) = \frac{1}{2}\ln[p_j(x)/(1-p_j(x))]$, *each class against the rest.*

Result 6 *The LogitBoost algorithm (J classes) uses quasi-Newton steps for fitting an additive symmetric logistic model by maximum-likelihood.*
Derivation
We first give the score ("s") and Hessian ("B") for the Newton algorithm corresponding to a standard multi-logit parametrization

$$G_j(x) = \ln\frac{\mathbb{P}(y_j^* = 1|x)}{\mathbb{P}(y_J^* = 1|x)},$$

with $G_J(x) = 0$ (and the choice of J for the base class is arbitrary).
The expected conditional log-likelihood is

$$\mathbb{E}(l(G+g)|x) = \sum_{j=1}^{J-1} \mathbb{E}(y_j^*|x)(G_j(x)+g_j(x)) - \ln(1 + \sum_{k=1}^{J-1} e^{G_k(x)+g_k(x)})$$

and

$$s_j(x) = \mathbb{E}(y_j^* - p_j(x)|x), \qquad j = 1,\ldots,J-1,$$

$$B_{j,k}(x) = -p_j(x)(\delta_{jk} - p_k(x)), \qquad j,k = 1,\ldots,J-1.$$

Our quasi-Newton update amounts to using a diagonal approximation to the Hessian, producing udates:

$$g_j(x) \leftarrow \frac{\mathbb{E}(y_j^* - p_j(x)|x)}{p_j(x)(1-p_j(x))}, \qquad j = 1,\ldots,J-1.$$

To convert to the symmetric parametrization, we would note that $g_J = 0$, and set $h_j(x) = g_j(x) - \frac{1}{J}\sum_{k=1}^{J} g_k(x)$. However, this procedure could be applied using any class as the base, not just the Jth. By averiging over all choices for the base class, we get the update

$$h_j(x) = \Big(\frac{J-1}{J}\Big)\Big(\frac{\mathbb{E}(y_j^* - p_j(x)|x)}{p_j(x)(1-p_j(x))} - \frac{1}{J}\sum_{k=1}^{j} \frac{\mathbb{E}(y_k^* - p_k(x)|x)}{p_k(x)(1-p_k(x))}\Big).$$

□

Appendix C

Description of the data sets used in the experiments

C.1 Pima Indians Diabetes Database

The original owners of the "Pima Indians Diabetes Database" are *The Institute of Diabetes and Digestive and Kidney Diseases*. The provided data set contains the following information:

- The diagnostic, a binary-valued variable, investigated is whether the patient shows signs of diabetes according to World Health Organization criteria (i.e., if the 2 hour post-load plasma glucose was at least 200 mg/dl at any survey examination or if found during routine medical care). The population lives near Phoenix, Arizona, USA.

- Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

- The dataset contains 8 attributes plus a class variable.

- The number of instances is 768.

- The attributes are (all numeric-valued):

  1. Number of times pregnant
  2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
  3. Diastolic blood pressure (mm Hg)
  4. Triceps skin fold thickness (mm)
  5. 2-Hour serum insulin (mu U/ml)
  6. Body mass index (weight in kg/(height in m)^2)
  7. Diabetes pedigree function
  8. Age (years)
  9. Class variable (0 or 1)

- There are no missing attribute values.

- The class distribution is:

Class value	Number of instances
0	500
1	268

  (Class value 1 is interpreted as "tested positive for diabetes".)
  Predicting that all instances have class value 0 will result in a classification rate of 65.1%.

- Brief statistical analysis:

Attribute number	Mean	Standard Deviation
1.	3.8	3.4
2.	120.9	32.0
3.	69.1	19.4
4.	20.5	16.0
5.	79.8	115.2
6.	32.0	7.9
7.	0.5	0.3
8.	33.2	11.8

C.2 German Credit Data

The provider of the "German Credit Data" is Professor Dr. Hans Hofmann from *Institut für Statistik und Ökonometrie, Universität Hamburg*. The data set contains the following information:

- The dataset contains 20 attributes plus a class variable.
  7 attributes are numerical and 13 attributes are categorical.
- The number of instances is 1000.
- Description of the attributes:

  Attribute 1: (qualitative). Status of existing checking account.
  - A11 : ... $<$ 0 DM
  - A12 : 0 $\leq$ ... $<$ 200 DM
  - A13 : ... $\geq$ 200 DM / salary assignments for at least 1 year
  - A14 : no checking account

  Attribute 2: (numerical). Duration in month.

  Attribute 3: (qualitative). Credit history.
  - A30 : no credits taken / all credits paid back duly
  - A31 : all credits at this bank paid back duly
  - A32 : existing credits paid back duly till now
  - A33 : delay in paying off in the past
  - A34 : critical account / other credits existing (not at this bank)

  Attribute 4: (qualitative). Purpose.
  - A40 : car (new)
  - A41 : car (used)
  - A42 : furniture/equipment
  - A43 : radio/television
  - A44 : domestic appliances
  - A45 : repairs
  - A46 : education
  - A47 : (vacation - does not exist?)
  - A48 : retraining
  - A49 : business
  - A410: others

  Attribute 5: (numerical). Credit amount.

Attribute 6: (qualitative). Savings account / bonds.
- A61 : ... $<$ 100 DM
- A62 : 100 $\leq$ ... $<$ 500 DM
- A63 : 500 $\leq$ ... $<$ 1000 DM
- A64 : ... $\geq$ 1000 DM
- A65 : unknown / no savings account

Attribute 7: (qualitative). Present employment since:
- A71 : unemployed
- A72 : ... $<$ 1 year
- A73 : 1 $\leq$ ... $<$ 4 years
- A74 : 4 $\leq$ ... $<$ 7 years
- A75 : ... $\geq$ 7 years

Attribute 8: (numerical). Installment rate in percentage of disposable income.

Attribute 9: (qualitative). Personal status and sex.
- A91 : male - divorced / separated
- A92 : female - divorced / separated / married
- A93 : male - single
- A94 : male - married / widowed
- A95 : female - single

Attribute 10: (qualitative). Other debtors / guarantors
- A101 : none
- A102 : co-applicant
- A103 : guarantor

Attribute 11: (numerical). Present residence since:

Attribute 12: (qualitative). Property.
- A121 : real estate
- A122 : if not A121 : building society savings agreement / life insurance
- A123 : if not A121 / A122 : car or other, not in attribute 6
- A124 : unknown / no property

Attribute 13: (numerical). Age in years.

Attribute 14: (qualitative). Other installment plans.
- A141 : bank
- A142 : stores
- A143 : none

Attribute 15: (qualitative). Housing.
- A151 : rent
- A152 : own
- A153 : for free

Attribute 16: (numerical). Number of existing credits at this bank.

Attribute 17: (qualitative). Job.
- A171 : unemployed / unskilled - non-resident
- A172 : unskilled - resident
- A173 : skilled employee / official
- A174 : management / self-employed / highly qualified employee / officer

Attribute 18: (numerical). Number of people being liable to provide maintenance for.

Attribute 19: (qualitative). Telephone.
- A191 : none
- A192 : yes, registered under the customers name

Attribute 20: (qualitative). Foreign worker.
A201 : yes
A202 : no

- The class distribution is:

Class value	Number of instances
1	700
2	300

(Class value 1 is interpreted as "good customer" and class value 2 is interpreted as "bad customer"). Predicting that all instances have class value 1 will result in a classification rate of 70%.

- Cost Matrix
The following cost matrix is given:

	1	2
1	0	1
2	5	0

The rows represent the actual classification and the columns the predicted classification.
It is worse to class a customer as good when they are bad (5), than it is to class a customer as bad when they are good (1).

C.3 Heart Disease

The Heart Disease data set contains a selection of The Heart Disease Database. from original owners of the "Pima Indians Diabetes Database" are *The Institute of Diabetes and Digestive and Kidney Diseases*. The provided data set contains the following information:

- This database contains 13 attributes (which have been extracted from a larger set of 75).

- Attribute Information:
– 1. age.
– 2. sex.
– 3. chest pain type (4 values).
– 4. resting blood pressure.
– 5. serum cholestoral in mg/dl.
– 6. fasting blood sugar > 120 mg/dl.
– 7. resting electrocardiographic results (values 0,1,2).
– 8. maximum heart rate achieved.
– 9. exercise induced angina.
– 10. oldpeak = ST depression induced by exercise relative to rest.
– 11. the slope of the peak exercise ST segment.
– 12. number of major vessels (0-3) colored by flourosopy.
– 13. thal: 3 = normal; 6 = fixed defect; 7 = reversable defect.

- Attributes types:

Real: 1, 4, 5, 8, 10, 12.
Ordered: 11.
Binary: 2, 6, 9.
Nominal: 7, 3, 13.

- Variable to be predicted:

  Absence (1) or presence (2) of heart disease.

- Cost Matrix

	absence	presence
absence	0	1
presence	5	0

  where the rows represent the true values and the columns the predicted.

- There are no missing values.

- There are 270 observations: 150 observations with class value 1 and 120 observations with class value 2.
  Predicting that all instances have class value 1 will result in a classification rate of 55.6%.

C.4 Waveform Data

The providers of the "Waveform Data" are Breiman, et al. . More information can be found in [1](pages 43-49). The provided data set contains the following information:

- Relevant information:
  3 classes of waves.
  21 attributes, all of which include noise.

- The 21 attributes have continuous values.

- The number of instances is 5000.

- Attribute Information:
  Each class is generated from a combination of 2 of 3 "base" waves.
  Each instance is generated added noise (mean 0, variance 1) in each attribute.

- There are no missing values.

- The class distribution is as follows: 33% for each of 3 classes.

Appendix D

Some remarks on the experiments

For the experiments Matlab version 5.3 is used. The results of all the boosting algorithms are contained in a zip file. The implementation of all the boosting algorithms together with the implementation of the weak learning algorithms are also contained in a zip file.

Writing out the whole implementation of the boosting algorithms and the weak learning algorithms would require too much space. It is therefore ommitted. People who want to receive the above mentioned results and/or implementation must contact me.

Bibliography

[1] Breiman, L., Friedman, J., Olshen, R. & Stone, C. (1984), *'Classification and Regression Trees'*, Wadsworth International Group, Belmont, California.

[2] Chan, P., Stolfo, S.J. (1995). A Comparative Evaluation of Voting and Meta-learning on Partitioned Data, in *'Proceedings of the Twelfth International Conference on Machine Learning'*.

[3] Chan, P., Stolfo, S.J. (1998). On the Accuracy of Meta-learning for Scalable Data Mining, in *'Journal of Intelligent Integration of Information'*, L. Kerschberg (editor), 1998.

[4] Chan, P., Stolfo, S.J. (1998b). Toward Scalable Learning with Non-uniform Class and Cost Distributions: A Case Study in Credit Card Fraud Detection.

[5] Chan, P., Fan W., Prodromidis A. & Stolfo S. (1999). Distributed Data Mining in Credit Card Fraud Detection, in *'IEEE Intelligent Systems November/December 1999 Volume 14, Number 6'* , page 67-74.

[6] Fan, W., Stolfo, S.J., Zhang, J. & Chan, P.K. (1999). AdaCost: Misclassification Cost-Sensitive Boosting, in *'Machine Learning: Proceedings of the Sixteenth International Conference'*, page 97-105.

[7] Fan, W., Stolfo, S.J., Zhang (1999b). The Application of AdaBoost for Distributed, Scalable and On-line Learning.

[8] Freund, Y. & Schapire, R.E. (1996). Experiments with a new boosting algorithm, in *'Machine Learning: Proceedings of the Thirteenth International Conference'*, page 148-156.

[9] Freund, Y. & Schapire, R.E. (1997). A decision-theoretic generalization of the on-line learning and an application to boosting, in *'Journal of Computer and System Sciences'*, 55(1): 119-139.

[10] Friedman, J., Hastie, T. & Tibshirani R. (1998, August 17). Additive Logistic Regression: a Statistical View of Boosting, Technical report, Department of Statistics, Stanford University.

[11] Friedman, J., Hastie, T. & Tibshirani R. (1999, November 30). Additive Logistic Regression: a Statistical View of Boosting (Second Revision), Technical report, Department of Statistics, Stanford University.

[12] Kowalczyk, W. (1998). Rough Data Modeling: a new technique for analyzing data, in: *'Rough Sets in Knowledge Discovery'*, L. Polkowski, A. Skowron (editors), Physica–Verlag, 1998, page 400-421.

[13] Kowalczyk, W. (1998). An emperical Evaluation of the Accuracy of Rough Data Models, in *'Proceedings of the Seventh International Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems'*, IPMU '98, Paris, La Sorbonne, page 1534-1538.

[14] Mitchell, T.M. (1997). *'Machine Learning'*, The McGraw-Hill Companies, Inc., United States of America.

[15] Prodromidis, A., Chan, P. & Stolfo S. (2000). Meta-learning in distributed data mining systems: Issues and approaches, in *'In Advances in Distributed and Parallel Knowledge Discovery'*, H. Kargupta and P. Chan (editors), Chapter 3, AAAI/MIT Press.

[16] Schapire, R.E. (1999). A brief introduction to boosting, in *'Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence'*.

[17] Schapire, R.E. (1999b). Theoretical views of boosting and applications, in *'Tenth International Conference on Algorithmic Learning Theory'*.

[18] Schapire, R. & Singer, Y. (1998). Improved boosting algorithms using confidence-rated predictions, in *'Proceedings of the Eleventh Annual Conference on Computational Learning Theory'*, page 80-91.

[19] Stolfo, S., Fan W., Lee, W., Prodromidis, A. & Chan, P. (2000). Cost-based Modeling for Fraud and Instrusion Detection: Results from the JAM Project, in *'Proc. DARPA Information Survivability Conference and Exposition, IEEE Computer Press'*, p. II 130-144.