RZ 3402 (#93465) 02/11/02
Computer Science 34 pages

Research Report

Some Theoretical and Practical Perspectives on Boosting Weak Predictors

Michel Cuendet and Abderrahim Labbi

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Some Theoretical and Practical Perspectives on Boosting Weak Predictors

Michel Cuendet Abderrahim Labbi

Intelligent Business Infrastructure
IBM Research, Zurich Research Laboratory
CH-8803 Rüschlikon
Switzerland

Email: abl@zurich.ibm.com

Abstract

In this paper, we present some theoretical and experimental results related to boosting weak decision trees. In the first section, we present a broad overview of boosting fundamentals and techniques used to build predictive classifiers and show different derivations of the AdaBoost algorithm. A theoretical explanation of the relationships between different derivations is presented. In section 2 we derive specific forms of boosted decision trees by introducing a more consistent minimization criterion. We focus in particular on "Stumps" which are one-level decision trees. The more complex Alternating Decision Trees (ADT) are also described. The important issue of overfitting and related regularization techniques are also addressed. We introduce a general formulation of the AdaBoost algorithm which extends it to the general case of cost-sensitive classification and regularization.

In section 3, we discuss implementation and computing performance issues and we present the results of the experiments conducted with the various boosting methods described in section 2 using a number of benchmark datasets as well as a project management database.

1 The fundamentals of boosting

1.1 Classification and Decision Trees

The problem is the following. We are given a set of N *examples* or *instances* described by a number of *attributes*. These attirbutes can be real valued numbers or nominal descriptors. The attribute values characterizing an instance are grouped in the input vector x. Each instance is affected to a class, described by the *label* y. The set of all possible attributes is noted X, and the set of labels Y. In the first part of this work, we only consider the two-class or binary problem, where the labels can only take values in $Y = \{-1; +1\}$. The examples are noted $z = (x, y)$, and they constitute the *training set* Z. We suppose that the examples in Z are drawn independently at random according to an unknown but fixed distribution D over $X \times Y$.

The goal of a classification algorithm is, given the training set, to be able to predict the labels y of unknown instances drawn from D when only x is known. For this, the algorithm builds an hypothesis $H(x)$ such that the *training error*

$$E^{train} = \frac{1}{N} \sum_{i=1}^{N} I(H(x_i) \neq y_i)$$

is minimum. Here $I(.)$ stands for the indicator function.

We assume that D is stationary, and that the training set is representative of this distribution. Then the hypothesis minimizing the training error should also minimize the prediction error on any set of examples not used in the learning process. To assess this in practice, a fraction of the available data is often kept out of the training set, to constitute a *test set* on which the *generalization error* or *test error* can be evaluated.

There are many types of classification algorithms, among which the neural networks, nearest neighbor algorithms, support vector machines, and decision trees. In this work, we only focus on decision trees, of which we give a short description.

In figure 1-1 is plotted a simple decision tree with two decision nodes (rectangles) and three prediction leaves (ellipses). The tree defines a binary classification rule which maps instances of the form $(a, b) \in \mathbb{R}^2$ into one of the two classes denoted by -1 or +1. According to the results of the tests performed at each decision node, an instance is mapped into a path along the tree from the root to one of the leaves.

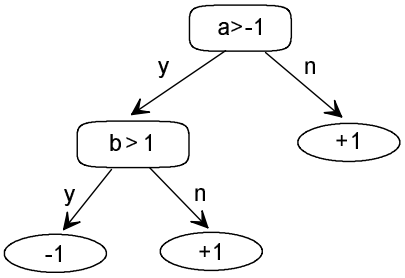

Figure 1-1 : Sketch of a basic decision tree for a binary problem.

The use of a decision tree to classify an instance is straightforward, but the construction of the optimal tree from the data is the challenging part. Usually, the algorithms proceed iteratively, starting at the root, and adding one decision node at a time. The optimal split is found by an extensive search

over all possible splits, and the decision rule minimizing or maximizing a given criterion is kept. We give a few examples of criterions that are commonly used to build decision trees.

1.1.1 The Gini index criterion

Let S be a set of examples representing m classes. Let also p_j be the relative frequency of class j. The *Gini measure of impurity* of S is defined as

$$gini(S) = 1 - \sum_{j=1}^{m} p_j^2$$

It reaches zero if only one class is present in S. The *Gini index* of a split between subsets S_1 and S_2 is a weighted average of the Gini measures of the two subsets:

$$gini[split] = \frac{N_1}{N} gini(S_1) + \frac{N_2}{N} gini(S_2)$$

where N_1 and N_2 are the size of the corresponding subsets, and N is the total number of examples. The split with the lowest value of the Gini index is chosen. The Gini index is used by the popular CART algorithm to grow trees. Note that this index does not allow to work with a weighted dataset.

1.1.2 The information gain criterion

Again, let p_j be the relative frequency of class j in a set S of N examples. The information needed to identify the class of an element of S is equivalent to the entropy of the class probability distribution $\{p_1, ..., p_m\}$. Accordingly, the *information* provided by S on the m classes is defined as

$$Info[S] = -\ [\ p_1 \log(p_1) + ... + p_m \log(p_m)\ ].$$

If the classes are uniformly represented in S, the information is $Info(S) = 1$. If all examples belong to the same class, $Info(S) = 0$. Now we define the *split information* for a split of S into subsets S_1 and S_2 containing N_1 and N_2 examples as

$$Info[S_1, S_2] = \frac{N_1}{N} Info(S_1) + \frac{N_2}{N} Info(S_2).$$

The *information gain* is defined by

$$Gain[S, S_1, S_2] = Info[S] - Info[S_1, S_2]$$

It represents the difference between the information needed to identify an element of S before and after the split. It is a measure of the information gain obtained by a the test on one attribute. The decision rule yielding the greatest information gain is adopted as the next node in the tree.This criterion is used in the famous C4.5 algorithm [25].

In this work, we don't use any of those two criteria to build decision trees. We use another construction rule, that optimizes the trees for use with boosting. Furthermore, we will in fact only build one-level decision trees (stumps), which relieves us from the problem of *pruning*.

Pruning refers to the limitation of the final size of the tree. It is often required that the iterative splitting stops when a minimum number of instances is reached in a leaf node. Even with this constraint, the trees obtained by repeating the Gini or information gain techniques are quite complex with long and uneven paths. The pruning of the decision tree is done by replacing a whole subtree by a

leaf node, thus canceling all further splits. The replacement takes place if the expected error rate in the subtree is greater than in the single leaf.

1.2 The Theory of Boosting

In the previous section, we have seen a way to build a tree classifier directly from the data. However, the generalization accuracy of such a classifier is limited. One can wonder if there is a way to improve this accuracy, for example by combining different decision trees. More generally, can a "weak" learning algorithm which performs just slightly better than random guessing be boosted into an arbitrarily accurate "strong" learning algorithm?

The technique of *boosting* brings an answer to this question. The term boosting was introduced by Freund and Schapire in 1995 in a milestone paper [12]. The concept of boosting relies on two main intuitive ideas :

- Averaging the predictions of different weak learners to get a more accurate hypothesis.
- Letting the weak learners focus on "hard to learn" examples

We will see shortly how these two benefits are efficiently combined in the AdaBoost algorithm. The sensitive point is how to compute weights for the averaging of the weak learners, and how to make the weak learners focus on hard examples. The boosting algorithm operates iteratively. An example is called "hard" if it has not been classified correctly by the previous weak learners. This can be specified to the next weak learner in two ways. One can use selective resampling of the training set. But the best way is to attribute weights to the examples, especially is their number is limited.

The first versions of AdaBoost were constructed based only on the two intuitive ideas above, and gave surprisingly good experimental results. For example, the generalization error was observed to continue to decrease even after the training error had reached zero. When used with decision trees, AdaBoost seemed to solve the problem of pruning by unraveling the unnecessary complexity. It also showed remarkable results with weak learners as simple as a single decision node.

Several justifications for this appeared later in the literature. First, the theory of margins gave some light on these outstanding capabilities. Theoretical convergence bounds on the training and generalization errors were derived. AdaBoost was proved to iteratively minimizes a loss functional over all possible weak hypotheses. It turned out to have also grounding in the game theory, as well as an independent probabilistic justification, as we will see in the following. Extensive work is still in progress in a number of research groups to settle the theoretical framework of AdaBoost, and to find even more powerful refinements.

1.2.1 The original AdaBoost algorithm

In this paragraph we reproduce the original AdaBoost algorithm as it was proposed by Freund and Schapire [7], and refined in subsequent papers from the same authors [5], [6] and [8]. The algorithm is given a training set of N examples $Z = \{z_i = (x_i, y_i), i = 1, ..., N\}$. On this set AdaBoost maintains a weight distribution $W = \left\{ w(z_i) \mid \sum_{i=1}^{N} w(z_i) = 1 \right\}$. It is assumed that we have at hand a weak learning algorithm that guarantees a weighted training error $\varepsilon < 1/2$ for any weights distribution.

AdaBoost

Initialize: $w(z_i) = 1/N$ for all $i = 1...N$.

Do for $t = 1...T$ **:**

1. Train the weak learner with the weighted training set $\{Z, W\}$.
   Obtain hypothesis $h_t(\mathrm{x}) \in \{-1, 1\}$.
2. Calculate the weighted training error of h_t
$$\varepsilon_t = \sum_{i=1}^{N} w_t(z_i)\, I(h_t(x_i) \neq y_i). \tag{1.2-1}$$
3. Set the voting weight
$$\alpha_t = \tfrac{1}{2} \ln \frac{1-\varepsilon_t}{\varepsilon_t}. \tag{1.2-2}$$
4. Update the weights:
$$w_{t+1}(z_i) = \frac{w_t(z_i) \exp\{-y_i\, \alpha_t h_t(x_i)\}}{Z_t}, \tag{1.2-3}$$
where Z_t is a normalization constant such that $\sum_{i=1}^{N} w_{t+1}(z_i) = 1$.

Output: Final Hypothesis

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

In the following, we will consider some improved versions of AdaBoost. Namely, this version only accepts boolean hypotheses from the weak learner, whereas versions dealing with real valued hypotheses show better performances.

The most general weak hypothesis that we will consider in this work is a rule of the following form

$$h_t : X \mapsto \{h_-; 0; h_+\}, \quad h_- < 0,\ h_+ > 0,$$

where the predicted label is the sign of h_t, and $|h_t|$ is a measure of the confidence of the prediction. In this case, the algorithm remain the same, except that the training error becomes

$$\varepsilon_t = \sum_{i=1}^{N} \omega_t(z_i)\, I(\mathrm{sign}[h_t(x_i)] \neq y_i). \tag{1.2-4}$$

1.3 The Classification Margin

Let's return to the case where the weak learner returns a hypothesis which is one of the classification labels. $h(x) \in \{y_1; ...; y_k\}$. We define the classification margin for an example as the difference between the total voting weight associated to the correct label and the maximal weight associated to any single incorrect label.

$$mg(z_i, a) = \sum_{t=1}^{T} \alpha_t I(h_t(x_i) = y_i) - \max_{y' \neq y_i} \sum_{t=1}^{T} \alpha_t I(h_t(x_i) = y')$$

An example is classified correctly if and only if its margin is positive. The margin is a measure of the consistence of the different weak hypotheses on one example. In the case where y can have only

two possible values $\{-1;+1\}$ the margin can be rewritten as

$$mg(z_i,a) = \sum_{t=1}^{T} a_t I(h_t(x_i) = y_i) - \sum_{t=1}^{T} a_t I(h_t(x_i) \neq y_i)$$
$$= y_i \sum_{t=1}^{T} a_t h_t(x_i)$$

and because $\sum_{t=1}^{T} a_t I(h_t(x_i) = y_i) + \sum_{t=1}^{T} a_t I(h_t(x_i) \neq y_i) = |a|$, the margin can also be formulated as

$$mg(z_i,a) = |a| - 2\sum_{t=1}^{T} a_t I(h_t(x_i) \neq y_i). \qquad (1.3\text{-}1)$$

In the last two paragraphs, we have introduced the standard AdaBoost algorithm, together with the margin, an essential parameter to follow its behavior. We focus now on some theoretical grounding and some justifications of why AdaBoost works.

1.4 Boosting and Game Theory

The Arcing Algorithm can be derived from very general assumptions in the framework of classical game theory. In basic game theory, a two-person game can be defined by a matrix M. There are two players called the row player and the column player. To play the game, the row player chooses a row i and the column player a column j. The loss suffered by the row player is $M_{i,j}$. The row player's goal is to minimize his loss. The game is said to be "zero-sum" if the goal of the column player is to maximize this loss. In the following, we restrict ourselves to the case where the number of choices available to each player, i.e. the dimension of M is finite.

Instead of choosing a single row or column (pure strategy), the players can make randomized choices (mixed strategy). That is, the row player chooses a distribution P over the rows of M, and simultaneously the column player chooses a distribution Q over the columns. The row player's expected loss is computed as:

$$\sum_{i,j} P(i)M(i,j)Q(j) = P^T MQ.$$

For ease of notation, we will denote this quantity by $M(P,Q)$. Von Neumann's well-known minmax theorem states that no matter which player plays first, the outcome is the same in either case so that

$$\min_P \max_Q M(P,Q) = \max_Q \min_P M(P,Q)$$

The common value of the two sides of the equality is called the value of the game.

In the case of the arcing algorithm, player 1 chooses $z_i \in Z$, and player 2 chooses one weak hypothesis h_m among all possible weak hypotheses. In terms of distributions, Player 1 chooses the weighting of the instances in Z and Player 2 chooses the voting weights over the weak hypotheses. The value of the game will be given by

$$\phi^* = \inf_a \max_i mg(z_i,a) = \sup_W \min_t E_W[l_t(z_i)].$$

where

$$E_W[l_t(z_i)] = \sum_{i=1}^{N} w_t(z_i) I(h_t(x_i) \neq y_i).$$

For the matrix game, the problem is to find the weak hypothesis h_t in order to minimize the error $E_W[l_t(z_i)]$ Breiman [2], as well as Freund et al. in [12] and [13] show that the arcing algorithm and hence AdaBoost can be derived as an optimal strategy for solving the game problem.

1.5 Boosting and Arcing Algorithms

Breiman [2] showed that AdaBoost is a special case of a larger class called *arcing algorithms*. The radial "arc" stands for "actively resample and combine". These algorithms apply to more general problems such as multiclass classification or regression. We focus only on the case of binary classification to reformulate the framework in the variables used here.

The starting point is to formulate the idea that the margin $mg(z, a)$ should be generally large by requiring uniform largeness. This means maximizing

$$\min_i \ mg_T(z_i, a).$$

We also define a loss function of the form

$$\ell(y_i, h_t(x_i)) = I(h_t(x_i) \neq y_i),$$

that we will also denote $\ell_t(z_i)$. From (1.3-1), we have that

$$mg_T(z_i, a) = |a| - 2 \sum_{t=1}^{T} a_t \ell(y_i, h_t(x_i))$$

The Arcing Algorithm works on a vector a of non-negative weights such that a_t is the weight assigned to the predictor h_t. It also maintains a weight distribution W, whose initial values are usually uniform. It updates a in the following steps:

1. Construct a a weight distribution W on Z, depending on the outcomes of the first t steps.
2. Choose h_{t+1} as the one that minimizes best $E_W \ell(y_i, h_t(x_i))$.
3. Calculate the voting weight a_{t+1} depending on the $(t+1)$ predictors already selected.
4. Repeat until satisfactory convergence.

Let $f(x)$ be a function such that $f(x) \to \infty$ as $x \to -\infty$ and $f(x) \to 0$ as $x \to \infty$ with everywhere negative first and positive second derivatives. We want to minimize

$$G_t(a) = \sum_{i=1}^{N} f(mg_t(z_i, a))$$

Essentially, G defines an error function over all margin distributions, which depends on the value of $|a|$. The larger the margins (i.e. the smaller the rate of incorrect classification), the smaller the value of G. To decrease G most efficiently, which pattern should increase its margin? The answer to this question lies in the gradient of G. Indeed, the arcing algorithm operates as follows: At the current value of a, update the weights according to

$$w_{t+1}(z_i) = \frac{f'(mg_t(z_i, a))}{\sum_i f'(mg_t(z_i, a))},$$

which is equivalent to

$$w_{t+1}(z_i) = \frac{\partial G_t(a)}{\partial m g_t(z_i, a)} \Big/ \sum_{i=1}^{N} \frac{\partial G_t(a)}{\partial m g_t(z_i, a)}. \tag{1.5-1}$$

At the next iteration $t = t + 1$, the algorithm finds the weak hypothesis h_t minimizing the weighted error $\sum_i w(z_i)\ell(z_i)$. The goal is then to integrate this new weak hypothesis in the final hypothesis in a way that minimizes the error function. In other words, we have to find the voting weight a_t that minimizes $G_t(a, a_t)$. This can be done by finding the solution of

$$\frac{\partial G_t(a, a_t)}{\partial a_t} = 0. \tag{1.5-2}$$

1.6 AdaBoost as an Arcing Algorithm

As it has been pointed out by Breiman [2] and other authors (Schapire & Singer [32], Friedman [14], Raetsch et al. [28], Allwein et al. [1]), AdaBoost is the particular arcing algorithm for which $f(x) = e^{-x}$. So, we have that

$$\begin{aligned} G_t(a) &= \sum_{i=1}^{N} \exp\{-m g_t(z_i, a)\} \\ &= \sum_{i=1}^{N} \exp\left\{-y_i \sum_{r=1}^{t} a_r h_r(x_i)\right\}. \end{aligned} \tag{1.6-1}$$

To find the weight updating rule, we apply formula (1.5-1) to get

$$w_{t+1}(z_i) = \frac{\exp\{-m g_t(z_i, a)\}}{\sum_i \exp\{-m g_t(z_i, a)\}}.$$

The denominator of this function plays the role of a normalizing factor insuring that the $w_{t+1}(z_i)$ are a distribution. To lighten the notations, we denote this factor ξ_t. The weight updating rule reads now

$$w_{t+1}(z_i) = \frac{1}{\xi_t} \exp\left\{-y_i \sum_{r=1}^{t} a_r h_r(x_i)\right\} \tag{1.6-2}$$

By taking the last time step out of the exponential,

$$w_{t+1}(z_i) = \frac{1}{\xi_t} \exp\left\{-y_i \sum_{r=1}^{t-1} a_r h_r(x_i)\right\} \exp\{-y_i a_t h_t(x_i)\}.$$

The first term is equivalent to the weight updating rule (1.6-2) to find $w_t(z_i)$ at the previous iteration, without its normalizing factor ξ_{t-1}. By substituting, we get

$$w_{t+1}(z_i) = \frac{\xi_{t-1}}{\xi_t} w_t(z_i) \exp\{-y_i a_t h_t(x_i)\}.$$

If we set $Z_t = \xi_t / \xi_{t-1}$, we obtain exactly the weight updating rule of AdaBoost, first stated in equation (1.2-3). We can summarize the relation between the two normalizing factors by using the iteration rule:

$$\xi_t = \prod_{r=1}^{t} Z_r. \tag{1.6-3}$$

Now, to find the voting weight of the weak hypothesis $h_t(x_i)$, we have to zero the derivative of $G_t(a)$ with respect to a_t, according to formula (1.5-2). First, we reformulate $G_t(a)$ by separating the last time step, and then using formula (1.6-3),

$$\begin{aligned} G_t(a, a_t) &= \sum_{i=1}^{N} \exp\left\{-y_i \sum_{r=1}^{t-1} a_r h_r(x_i)\right\} \exp\{-y_i a_t h_t(x_i)\} \\ &= \sum_{i=1}^{N} \xi_{t-1} w_t(z_i) \exp\{-y_i a_t h_t(x_i)\}. \end{aligned} \tag{1.6-3}$$

Then, we differentiate and split the sum in two parts depending on the correctness of the prediction of h_t.

$$\begin{aligned} \frac{\partial G_t}{\partial a_t} &= \sum_{i=1}^{N} -y_i h_t(x_i) \xi_{t-1} w_t(z_i) e^{-y_i a_t h_t(x_i)} \\ &= -\sum_{i:y_i = h_t(x_i)} \xi_{t-1} w_t(z_i) e^{-a_t} + \sum_{i:y_i \neq h_t(x_i)} \xi_{t-1} w_t(z_i) e^{a_t} \end{aligned}$$

Setting $\frac{\partial G_t}{\partial a_t} = 0$ leads to

$$a_t = \frac{1}{2} \ln\left\{ \frac{\sum_{i:y_i = h_t(x_i)} w_t(z_i)}{\sum_{i:y_i \neq h_t(x_i)} w_t(z_i)} \right\}.$$

The denominator in this expression is exactly the weighted training error ε_t as defined in (1.2-1). Because the labels $y_i \in \{-1; 1\}$ and the weights sum up to one, the numerator is equal to $1 - \varepsilon_t$. Thus,

$$a_t = \frac{1}{2} \ln\left\{ \frac{1 - \varepsilon_t}{\varepsilon_t} \right\}.$$

This is exactly the expression introduced in (1.2-4). Note that ε_t is required to be smaller than $1/2$, so a_t is bound to be positive.

1.7 The alternative approach of Schapire and Singer

Schapire and Singer [32] provide a simpler analysis of the AdaBoost algorithm in which the weak hypothesis h_t can take any real value instead of being restricted to $\{-1; 1\}$. The sign of $h_t(x)$ is interpreted as the predicted label, and the magnitude $|h_t(x)|$ as a measure of the confidence in the prediction. Thus, if $h_t(x)$ is close to zero, it is interpreted as a low confidence prediction.

With such a weak hypothesis, one can expect that the choice of the importance of a weak hypothesis in the process of boosting can be left to some extent to the weak learner. We will show that the value of $h_t(x)$ can be set by the weak learner in a way that its magnitude actually plays the role of the voting weight a_t in the determination of the final hypothesis. With this kind of *self-voted weak hypothesis*, AdaBoost just sums the h_t without reweighting them, which corresponds to $a_t = 1 \ \forall t$. But for now, let $h_t(x)$ be simply real valued and a_t be unspecified.

Let the final hypothesis of AdaBoost be

$$H_T(x) = \sum_{t=1}^{T} a_t h_t(x),$$

so that the predicted label is $\text{sign}(H(x))$. We introduce here a first bound on the training error of H, on which the determination of a_t is based.

Theorem 1 (Schapire and Singer [32]) : Assuming the previous notations, the following bound holds on the training error of H :

$$\frac{1}{N}|\{i : \text{sign}(H(x_i)) \neq y_i\}| \leq \prod_{t=1}^{T} Z_t,$$

where Z_t is the normalization factor of the weight update rule in AdaBoost (1.2-3).

Proof : By unraveling the update rule remembering that the weights are initialized to $1/N$, we have:

$$w_{T+1}(x_i) = \frac{\exp\{-y_i H_T(x_i)\}}{N \prod_{t=1}^{T} Z_t} \tag{1.7-1}$$

Moreover, if $\text{sign}(H(x_i)) \neq y_i$ then $\exp\{-y_i H(x_i)\} \geq 1$. Thus,

$$I(\text{sign}(H(x_i)) \neq y_i) \leq \exp\{-y_i H(x_i)\}. \tag{1.7-2}$$

Combining Equations (1.7-1) and (1.7-2), give the stated bound since

$$\begin{aligned} \frac{1}{N} \sum_{i=1}^{N} I[\text{sign}(H(x_i)) \neq y_i] &\leq \frac{1}{N} \sum_{i=1}^{N} \exp\{-y_i H(x_i)\} \\ &= \sum_{i=1}^{N} w_{T+1}(x_i) \prod_{t=1}^{T} Z_t \\ &= \prod_{t=1}^{T} Z_t. \end{aligned}$$

■

The important consequence of Theorem 1 is that, in order to minimize the training error, a reasonable approach might be to greedily minimize the bound given in the theorem, by minimizing Z_t on each round of boosting. We can apply this idea in the choice of a_t, or directly for the choice of a self-voting weak hypothesis h_t.

1.7.1 Choosing a_t like the original AdaBoost

To simplify notation, let us fix t and drop the corresponding indexes. We denote $w^i = w_t(x_i)$ and $u_i = y_i h_t(x_i)$. In the following discussion, we assume without loss of generality that $w^i \neq 0$ for all i. The original AdaBoost algorithm is a special case where $h \in \{-1; 1\}$. Then also $u_i \in \{-1; 1\}$, and

$$\begin{aligned} Z &= \sum_{i=1}^{N} w^i \, e^{-a u_i} \\ &= \sum_{i=1}^{N} w^i \left(\frac{1+u_i}{2} e^{-a} + \frac{1-u_i}{2} e^{a} \right). \end{aligned}$$

This expression can be analytically minimized to find

$$\alpha = \frac{1}{2} \ln\left(\frac{\sum_{h(x_i)=y_i} w^i}{\sum_{h(x_i)\neq y_i} w^i} \right).$$

Alternatively, given ε_t defined in (1.2-4) and that the weights sum up to one

$$\alpha = \frac{1}{2} \ln\left(\frac{\varepsilon_t}{1-\varepsilon_t} \right).$$

1.7.2 Choosing α_t in the general case

In the case where h_t can take any real value, we can numerically minimize $Z(\alpha)$ by solving

$$\frac{dZ}{d\alpha} = -\sum_{i=1}^{N} w^i u_i \, e^{-\alpha u_i} = 0.$$

It is interesting to note that by the definition of w_{t+1}^i, the equation above is equivalent to

$$-Z \sum_{i=1}^{N} w_{t+1}^i u_i = 0.$$

Thus, if w_{t+1}^i is formed using the value of α_t that minimizes Z_t (so that $Z'(\alpha) = 0$), then we have that

$$\sum_{i=1}^{N} w_{t+1}^i u_i = E_{W_{t+1}}[y_i h_t(x_i)] = 0.$$

In words, this means that , with respect to the distribution W_{t+1}, the weak hypothesis h_t will be exactly uncorrelated with the labels y_i. In the case where $h \in \{-1; 1\}$, this is equivalent to say that the w_{t+1}^i are chosen such that the previous weak hypothesis h_t has exactly a weighted training error ε of $1/2$. This is characteristic of a perpendicular search as in a gradient decent method.

It can be verified that $d^2Z/d\alpha^2$ is strictly positive for all $\alpha \in \mathbb{R}$ (ignoring the trivial case that all $h(x_i) = 0$, $\forall i$. Therefore, $Z'(\alpha)$ can have at most one zero. Moreover, if there exists at least one $h(x_i)$ of each sign, then $Z'(\alpha) \to \pm\infty$ as $\alpha \to \pm\infty$. This means that $Z'(\alpha)$ has at least one root, except in the degenerate case where all non-zero $h(x_i)$'s have the same sign. So, we can numerically find the unique minimum of $Z(\alpha)$.

1.8 The Z criterion for finding self-voted weak hypotheses

So far, we have only used theorem 1 to find an appropriate voting weight α, given a weak hypothesis. This theorem can be applied more broadly, to design weak learning algorithms that can be combined more powerfully with boosting. We assume that the weak learner can freely scale any weak hypothesis h by a constant factor. This factor can be chosen equivalent to α. In other words, we can fold α into h, and set the goal of the weak learner to minimize

$$Z = \sum_{i=1}^{N} w(x_i) \exp\{-y_i h(x_i)\}. \tag{1.8-1}$$

In the case of decision trees, the predictions are based on a partitioning of the attribute space. It is divided into disjoint blocks $X_1, ..., X_M$ for which $h(x) = h(x')$ for all $x, x' \in X_j$. In other words, the prediction of h depends only on which block X_j falls into.

Suppose that at time t we are given a partition $X_1, ..., X_M$. The goal is now to find a function h that minimizes Z. Let us denote $h^j = h(x)$ for $x \in X_j$. We have to find appropriate choices for h^j. For each j, define W_+ and W_- such that

$$W_{\pm}^j = \sum_{i: x_i \in X_j \wedge y_i = \pm 1} w(x_i) = P_W\left[x_i \in X_j \wedge y_i = \pm 1\right].$$

Note that the index $\pm$ refers to the labels y_i and not to the correlation between the y_i and the h^j. We can further admit that some attributes have missing values among the x_i. If these missing values prevent from evaluating h, the weak learner should abstain and output $h = 0$ for these examples. Let's denote X_0 the set of examples that have missing values for the attributes needed to determine h, and W_0 the corresponding weight. Then equation (1.8-1) can be rewritten

$$\begin{aligned} Z &= \sum_j \sum_{i: x_i \in X_j} w(x_i) \exp\{-y_i h^j\} \\ &= W_0 + \sum_j \left[W_+^j e^{-h^j} + W_-^j e^{h^j} \right]. \end{aligned} \tag{1.8-2}$$

This is minimized for

$$h^j = \frac{1}{2} \ln\left\{ \frac{W_+^j}{W_-^j} \right\}. \tag{1.8-3}$$

Note that the sign of h^j is equal to the (weighted) majority class within block j. Moreover, h^j is close to zero (a low confidence prediction) if there is a roughly equal split of positive and negative examples in block j. Likewise, h^j is far from zero if one label strongly predominates.

Plugged into the above expression of Z, this choice gives:

$$Z = W_0 + 2 \sum_j \sqrt{W_+^j W_-^j}\ . \tag{1.8-4}$$

This can be chosen as a splitting criterion for growing a decision tree, rather than the Gini index or an entropic function as described in section 1.1. One decision node is built by greedily choosing the split which causes the greatest drop in the value of Z. This is where the algorithm requires a lot of computing power: At each node, every possible split for every attribute has to be tested. We address this point in detail in section 3.1.

Let's come back for a moment on the output of a decision node, h^j as defined in equation (1.8-3). It may well happen that W_+^j or W_-^j is very small or even zero, in which case h^j will be very large or infinite in magnitude. In practice, such large predictions can cause numerical problems. In addition, there may be theoretical reasons to suspect that large, overly confident predictions will increase the tendency to overfit (see [32]).

To limit the magnitude of the predictions, a smoothed version of (1.8-3) can be used. For some small positive δ,

$$h^j = \frac{1}{2} \ln\left\{ \frac{W_+^j + \delta}{W_-^j + \delta} \right\}. \tag{1.8-5}$$

This has the effect of bounding $|h^j|$ by $\frac{1}{2} \ln\{(1+\delta)/\delta\}$. Moreover, this only slightly increases the value of Z, and doesn't degrade much the accuracy of the criterion. It can be shown [32] that for a partition in two blocks,

$$Z_\delta \leq Z + 2\sqrt{\delta}\ .$$

In practice, we choose δ on the order of $1/N$, where N is the number of examples.

1.8.1 The link to the G functional

We provide in this paragraph a demonstration of the link between the G functional framework introduced by Breiman [2] and used by Raetsch [31] and Fan et al. [4], and the approach of Schapire and Singer [32] based on Z_t.

Proposition 2 : With the notations introduced in this paragraph,

$$G_t(z,h) = \prod_{r=1}^{t} Z_r .$$

Proof : We rewrite equation (1.6-3) in the case where h_t is a self-voting weak hypothesis.

$$\begin{aligned} G_t(z,h) &= \sum_{i=1}^{N} \exp\left\{-y_i \sum_{r=1}^{t} h_r(x_i)\right\} \\ &= \sum_{i=1}^{N} \exp\left\{-y_i \sum_{r=1}^{t-1} h_r(x_i)\right\} \exp\{-y_i h_t(x_i)\} \\ &= \sum_{i=1}^{N} \xi_{t-1} w_t(z_i) \exp\{-y_i h_t(x_i)\}. \end{aligned}$$

Remembering that $\xi_t = \prod_{r=1}^{t} Z_r$ from equation (1.6-2),

$$G_t(z,h) = \prod_{r=1}^{t-1} Z_r \sum_{i=1}^{N} w_t(z_i) \exp\{-y_i h_t(x_i)\}.$$

The sum term is exactly the expression of Z_t (1.8-1). Hence the result. ■

At step t, the only contribution of h_t in $G_t(z,h)$ is through the term Z_t. So when we differentiate $G_t(z,h)$ with respect to a_t, or equivalently here by $|h_t|$, to find its optimal magnitude, we get

$$\frac{\partial G_t}{\partial |h_t|} = \prod_{r=1}^{t-1} Z_r \ \frac{\partial Z_t}{\partial |h_t|}.$$

This shows that minimizing Z_t is equivalent to minimizing $G_t(z,h)$.

1.9 An original Z criterion

There is an alternative approach to Schapire and Singer's way to build a weak hypothesis minimizing Z_t as exposed in paragraph 1.8. Returning to equation (1.8-2), we can group the weights of the instances belonging to X_j in a different manner. Instead of summing the weights according to the label of the examples, we can focus on the correlation between the y_i and the h^j. We group in $\overline{W}_+^j$ all instances for which h^j makes the correct prediction for y_i, i.e. h^j and y_i have the same sign.

$$\overline{W}_{\pm}^{j} = \sum_{i:x_i \in X_j \wedge \text{sign}(h_j)=y_i} w(x_i) \ . \qquad (1.9\text{-}1)$$

As before, we denote W_0 the total weight of the examples for which h_t cannot be determined. According to this new weight grouping, the test value Z is computed as follows

$$\overline{Z} = \sum_j \sum_{i:x_i \in X_j} w(x_i) \exp\{-y_i h^j\}$$
$$= W_0 + \sum_j \left[\overline{W}_+^j \, e^{-|h^j|} + \overline{W}_-^j \, e^{|h^j|} \right].$$

This is minimized for

$$|h^j| = \left| \frac{1}{2} \ln\left(\frac{\overline{W}_+^j}{\overline{W}_-^j} \right) \right|. \qquad (1.9\text{-}2)$$

This expression of h^j is more interesting than the one (1.8-3) proposed by Schapire and Singer for two reasons. If we get back to the definition of $\overline{W}_-^j$ (1.9-1), we see that it is equivalent to a reduction of ε as defined in (1.2-4) to X_j. ε represents the sum of the weights of all wrong classified instances, and $\overline{W}_-^j$ is the sum of the weights of the wrong classified instances belonging to X_j. In the same way, $\overline{W}_+^j$ is related to $1 - \varepsilon$, so that h^j (1.9-2) is very similar to the voting weight α as introduced in (1.2-2). It is exactly what we want, because we are trying to infer self-voted weak hypotheses. Like in the approach of Schapire and Singer, instead of a single average voting weight α we have now a set $\{h^j\}$ of voting weights specific to each block X_j, but here they are defined consistently with α.

The search for the minimal $\overline{Z}$ is however not as simple as before, because the $\overline{W}_\pm^j$ depend on h_t. Another problem arises with the sign of h^j which is not determined directly by (1.9-2). In the case of binary splits (stumps, $j = 1, 2$) we can proceed in our search anyway. The algorithm examines each possible splitting point for each attribute. For a given splitting point, we require that h_t has different signs on each side. This leaves us with two combinations for which we have to evaluate $\overline{Z}$. We call them A and B.

$$A : h^1 > 0; h^2 < 0$$
$$B : h^1 < 0; h^2 > 0.$$

To evaluate $\overline{Z}_A$ and $\overline{Z}_B$, we have to carefully map the $\overline{W}_\pm^j$ on the $W_\pm^j$, according to assumption A or B. The $W_\pm^j$ can be computed from the data directly. We get

$$\begin{aligned}
\overline{Z}_A = {} & W_+^1 \, e^{-\left|\frac{1}{2}\ln\left(\frac{W_+^1}{W_-^1}\right)\right|} + W_-^1 \, e^{+\left|\frac{1}{2}\ln\left(\frac{W_+^1}{W_-^1}\right)\right|} + \\
& W_-^2 \, e^{-\left|\frac{1}{2}\ln\left(\frac{W_-^2}{W_+^2}\right)\right|} + W_+^2 \, e^{+\left|\frac{1}{2}\ln\left(\frac{W_-^2}{W_+^2}\right)\right|}, \\
\overline{Z}_B = {} & W_-^1 \, e^{-\left|\frac{1}{2}\ln\left(\frac{W_-^1}{W_+^1}\right)\right|} + W_+^1 \, e^{+\left|\frac{1}{2}\ln\left(\frac{W_-^1}{W_+^1}\right)\right|} + \\
& W_+^2 \, e^{-\left|\frac{1}{2}\ln\left(\frac{W_+^2}{W_-^2}\right)\right|} + W_-^2 \, e^{+\left|\frac{1}{2}\ln\left(\frac{W_+^2}{W_-^2}\right)\right|}
\end{aligned} \qquad (1.9\text{-}3)$$

Finally, the sign of h_j is determined by which of $\overline{Z}_A$ or $\overline{Z}_B$ is lowest, and the magnitude of h_j is set by expression (1.9-2). This value of h_j reflects better a measure of the confidence for the prediction, because it is based on a ratio of good- and bad-classified instances instead of a ratio of positive to negative label, like in Schapire and Singer's definition. A further advantage of this version is that is allows to require that the hypotheses on the two leaves of a stump are of opposite sign. In practice, it happens that the weak learner determined by Z returns two hypotheses of the same sign, which doesn't make sense.

2 Boosting Simple Weak Learners

2.1 Boosting stumps

A stump is a terminology introduced by Breiman [2] for a one-level decision tree. Such a weak learner finds a prediction based on the result of a single test comparing one of the attributes to one of its possible values. For discrete attributes, equality is tested, and for continuous attributes, a threshold value is compared. The best weak hypothesis of this form which optimizes the learning criterion Z defined above can be efficiently found by a direct search.
The use of stumps as weak learners is interesting form different points of view, because a stump is the most elementary form of decision tree, and maybe the most elementary form of classifier. As such, stumps require relatively little computational power to be learnt. They also allow a good interpretability of the boosted classifier, unlike more sophisticated weak learners.

The boosted stumps have shown remarkable performance as compared to more complex weak learners, as pointed out by Iba and Langley [17], Holte [16], Mitchell [24]. Stumps have become a common workbench weak learner for boosting in many papers, among which Freund and Schapire [11], Schapire and Singer [32] and Freund and Mason [10].

For all these reasons, we focus on boosted stumps in this study. The algorithm we use is equivalent to Schapire and Singer's *real AdaBoost.MH* [32]. It is based on a special case of the results of section 1.8. A stump partitions the attribute domain in only two subsets X^1 and X^2, so all results of section 1.8 apply with $j = 1, 2$. The weak learner considers all possible splits and selects the one that minimizes Z (see section 3.1 for implementation issues). The determinant attribute is stored, as well as the best threshold value if it is a continuous attribute, or the best partition of its values if it is a nominal attribute. Finally, the weak hypothesis (h^1, h^2) is calculated and passed to the booster.

A slightly different version of stumps is built in the same manner with the criterion $\overline{Z}$ devised in section 1.9. The criterion $\overline{Z}$ is specially designed for finding binary stumps.

2.2 Boosting Consistent Stumps

As we mentioned, a special property of stumps is that the test at the single decision node is done on a single attribute x_k. Thus, the contribution h_t to the final hypothesis is only function of x_k. For example, if the attribute is continuous, h_t is a step function. After some iterations, this same attribute is chosen a number of times, and the contribution of x_k to the final hypothesis can be isolated as

$$H_T^k(x_k) = \sum_{T_k} h_t(x)$$

where $T_k = \{t \le T \mid h_t = h_t(x_k)\}$ represents all iterations where x_k was chosen as the most determinant pattern. According to this, the final hypothesis can be expressed as

$$H_T(x) = \sum_k H_T^k(x_k)$$

We now focus on one component $H_k(x_k)$. $H_k(x_k)$ is a superposition of step functions, and is piece wise constant. In an interval where it is constant, a given value of $H_k(x_k)$ can be determined by different configurations of weak hypotheses. It can be a sum of large magnitude but contradicting $h_t(x)$'s. Alternatively it can be the sum of lower magnitude weak hypotheses having all the same sign. The latter possibility would give rise to a more reliable H_k, because the stumps are consistent in this region.

In other words, because stumps depend only on one attribute, one can measure a posteriori the consistency of the all past weak hypotheses. If the magnitude of H_k represents the confidence in that

component of the final hypothesis, it should be decreased when the weak hypotheses disagree. The disagreement between weak hypotheses can be measured by the coefficient

$$Q_T^k(x_k) = \frac{\left| \sum_{T_k^+} |h_t(x)| - \sum_{T_k^-} |h_t(x)| \right|}{\sum_{T_k} |h_t(x)|}.$$

$T_k^+ = \{t \le T \mid h_t = h_t(x_k), h_t > 0\}$ represents all iterations where x_k was chosen and the predicted label is +1, and vice-versa. The coefficient Q^k is equal to 1 if all $h_t(x)$ have the same sign (consistent prediction). It goes down to zero if the weak hypotheses disagree and compensate exactly. Note that in this specific case H_k is zero anyway.

The standard AdaBoost sets the magnitude of the h_t, thus the voting weight, once for all at t. The idea here is to adapt the voting of all weak hypotheses at each iteration using Q^k as an a posteriori referee.

$$\tilde{H}_T(x) = \sum_k H_T^k(x_k) Q_T^k(x_k).$$

The rest of the AdaBoost algorithm remains the same except the weight-updating rule which becomes

$$w_{T+1}(z_i) = \frac{1}{\xi_T} \exp\left\{-y_i \tilde{H}_T(x_i)\right\},$$

according to equation (1.6-2). We call the variant of AdaBoost *consistent boosting*.

2.3 The Alternating Decision Tree

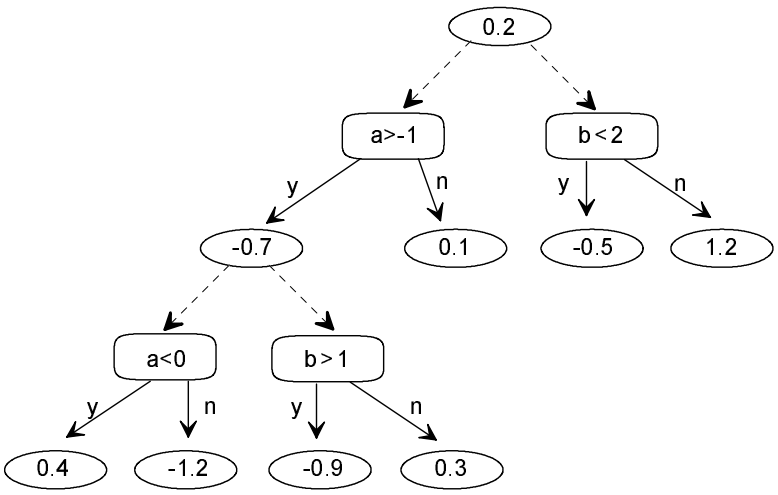

Figure 2-1 : An example of an alternating decision tree.

Freund & Mason (10) have introduced the Alternating Decision Tree (ADT) model as a generalization of voted stumps. It allows to recursively build decision trees of more than one level. Boosting is used to add new decision rules at any level. As we see below, Alternating decision trees also generalize standard decision trees, such as those generated by CART or C5.0. Freund & Mason show that the performance of their ADT is similar to C5.0 on some benchmark binary problems.

ADT's have the advantage of being smaller in size than standard decision trees, thus improving interpretability. As another advantage, they give, in addition to classification, a measure of confidence.

The example shown in figure 2-1 defines a rule which maps instances of the form $(a, b) \in \mathbb{R}^2$ into one of the two classes denoted by -1 and $+1$. Each decision step is represented by two nodes : a *splitter node* (rectangle) and a *prediction node* (ellipse). The decision node is identical to what we had in the classical decision tree model on figure 1-1, while the prediction node is associated with a real valued number. The ADT's were called "alternating trees", because they consist of alternating layers of prediction nodes and splitter nodes.

As in standard decision trees, an instance is mapped into a path along the tree from the root to one of the leaves. However, unlike standard decision trees, the classification associated with the path is not the label of the leaf, but the sum of the predictions along the path.

Furthermore, in standard trees, only leaf nodes can be split. In ADT's each node is allowed to be split multiple times. In this case, an instance defines a set of paths in the decision tree. When reaching a prediction node, a path continues with all of the children of the node. The classification is then the sum of all prediction nodes that are included in the set of paths corresponding to an instance.

In the example of figure 2-1, the classification of the instance $a = b = 0.5$ is $\text{sign}(0,2 + [-0.7 - 1.2 + 0.3] - 0.5) = \text{sign}(-1.4) = -1$. The score in square brackets results from the paths going through the $[a > -1]$ splitter node. The classification of $a = -2,\ b = 2$ would be $\text{sign}(0.1 - 0.5) = \text{sign}(-0.4) = -1$. In both cases the classification is -1, however, we can think of the first as more reliable as the second.

The classification rule described by an ADT can be rewritten as a weighted majority vote. We associate with each of the decision nodes a rule of the following form:

```
h(a,b) =
if (precondition) then
if (condition) then output h1
else output h2
else output 0
```

By combining such rules with the constant prediction of the root node, we can rewrite the classification rule associated with the ADT above as $\text{sign}\left(0.2 + \sum_{i=1}^{4} h_i(a, b)\right)$. From this description, it is clear that ADT's are a generalization of voted stumps. These correspond to alternating trees with exactly three layers: a root, a set of decision nodes, and the predictor nodes associated with these decision nodes.

We have shown that ADT's can be defined as a sum of simple base rules. As a result, boosting can be applied to the problem of learning an ADT from examples. The only special thing here is that the set of base rules (weak hypotheses) that are considered at each stage is not constant but increases as the tree is grown. The rules themselves remain as simple as the stumps. It is just the number of possible preconditions, and so the number of partitions of the input space to consider at each step, that increases.

To build an algorithm to learn an ADT, we can directly extend the frame work of AdaBoost combined with the approach of Schapire and Singer for finding a real valued weak learner. Let C be the set of base conditions. These are inequalities comparing a real valued attribute and a constant, or equalities between a discrete attribute and a partition of its possible values. The algorithm maintains a set of preconditions P_t updated at every iteration t. The initial precondition set is $P_1 = \{True\}$. The notation $W_{\pm}(c)$ represents the total weight of the training examples which satisfy the predicate c and that are labeled $+1$ or -1.

2.3.1 The ADT Boost Algorithm

Initialize :

1. Set weights $w(z_i) = 1/N$ for all $i = 1...N$.
2. Set $h_0(x)$ to be the base rule whose precondition and condition are both *True*, and whose first prediction value is

$$h = \frac{1}{2} \ln \frac{W_+(True)}{W_-(True)}$$

Do for $t = 1...T$ **:**

1. For each precondition $p \in P_t$ and each condition $c \in C$ calculate

$$\begin{aligned} Z(p,c) = & 2\sqrt{W_+(p \wedge c) W_-(p \wedge c)} \\ & + 2\sqrt{W_+(p \wedge \neg c) W_-(p \wedge \neg c)} \\ & + W(\neg p) + W_0. \end{aligned}$$

2. Select p and c which minimize $Z(p,c)$.
   Set $h_t(x)$to be the rule whose precondition is p, condition is c,
   The prediction values are:

$$h^c = \frac{1}{2} \ln \frac{W_+(p \wedge c)}{W_-(p \wedge c)}; \qquad h^{\neg c} = \frac{1}{2} \ln \frac{W_+(p \wedge \neg c)}{W_-(p \wedge \neg c)}$$

$$h^{\neg p} = h^{\text{missing}(c)} = 0$$

3. Set P_{t+1} to be P_t with the addition of $p \wedge c$ and $p \wedge \neg c$.
4. Update the weights:

$$\omega_{t+1}(z_i) = \frac{\omega_t(z_i) \exp\{-y_i \, h_t(x_i)\}}{Z_t},$$

   where Z_t is a normalization constant such that $\sum_{i=1}^{N} \omega_{t+1}(z_i) = 1$.

Output :

$$H(x) = \sum_{t=0}^{T} h_t(x).$$

The predicted label is $\text{sign}(H(x))$.

The algorithm starts by finding the best constant prediction for the entire data set. This prediction is placed at the root of the tree. It then grows the tree iteratively, adding one base rule at a time. The added base rule corresponds to a subtree with a decision node as its root and two prediction nodes as the leaves. This subtree is added as a child of a predictor node which might or might not be a leaf node.

Note that the algorithm ADT Boost can handle missing attributes. As the stump weak learner, it associates them with the weight W_0, and outputs a zero hypothesis (denoted by $h^{\text{missing}(c)}$above). The Z number is computed in a way similar to previously, except that the term $W(\neg p)$ is added for the

examples excluded by the precondition. This term increases the value of Z for all examples not treated by the node because they belong to other branches of the tree. It prevents the tree to grow too much in depth even if very accurate rules can be found for a small number of instances.

We have found that in the case of a data set with a low ratio of positive labels, the first step of the algorithm is not appropriate. The initial precondition is too high, and the subsequent contributions h_t are not large enough bring the final hypothesis back to negative for negative labels. To prevent this, we suggest to start with a zero precondition, and to adapt the initial weights accordingly, i.e. $W_+ = W_-$. We replace the initializing step of the algorithm by

Initialize :

1. Set weights:

$$w_o(y_i = 1) = \frac{0.5}{\sum I(y_i=1)}; \qquad w_o(y_i = -1) = \frac{0.5}{\sum I(y_i=-1)}.$$

2. Set first precondition value $h_0 = 0$.

2.4 Handling Noisy and Unbalanced Data

In this section, we review the problems that occur when AdaBoost is applied to noisy data. We describe the soft-margin regularization answer to this problem. Then we introduce an adaptation of AdaBoost to handle cost-sensitive classification. This is useful for unbalanced data, where the ratio of classes is strongly uneven. Finally we propose an unifying framework to have regularization and cost-sensitivity working together.

2.4.1 Overfitting

Schapire et al. [32] prove a bound on the training error similar in some aspects to the bound we used in section 1.8. This result deals with the whole margin distribution over the training set S. If the weak learning algorithm generates classifiers with weighted training errors $\varepsilon_1, ..., \varepsilon_T$, Then for any θ we have that

$$P_S\left(y\tilde{H}(x) \le \theta\right) \le 2^T \prod_{t=1}^{T} \sqrt{\varepsilon_t^{1-\theta}(1-\varepsilon_t)^{1-\theta}}$$

$\tilde{H}$ is the weighted majority vote of the classifiers, i.e. $H/\Sigma|h|$. This bound guarantees that the training error converges to zero for a large number of iterations. This bound holds if ε_t is bounded away from 0.5. This is guaranteed even on noisy data by the results of Jian [18]. We will see that this convergence of the training error does not however imply the convergence of the generalization error.

Lets consider examples outside a training set S of size N, drawn according to the same distribution D. The following bound is provided by Schapire et al. [32]. With probability $1-\delta$ over the random choice of S, for all $\theta > 0$ the final hypothesis satisfies

$$P_D(yH(x) \le 0) \le P_S(yH(x) \le \theta) + O\left(\sqrt{\frac{1}{N}\left[\frac{d\log\frac{2N}{d}}{\theta^2} + \log\frac{1}{\delta}\right]}\right) \quad (2.4\text{-}1)$$

Here d is the Vapnik-Chevronenkis (VC) dimension of the set of all hypotheses the weak learner can possibly deliver. It measures the complexity of the weak hypothesis space available. For more

details, see [32]. More elaborated bounds can be found in the literature, for example in Mansour and McAllester [23], Kegl et al. [21], Shawe-Taylor and Cristianini [36].

This central result on the convergence of AdaBoost's generalization error is based on the margin distribution on the training set. According to the bound (2.4-1), for a given VC dimension, the reason for the success of AdaBoost is the maximization of the margin. It can be experimentally observed that AdaBoost maximizes the margin of instances which are most difficult, i.e. have the smallest margin.

However, Breiman [2] showed early after the bound above was published that different margin distributions for the same VC-dimension has little effect on the generalization error. The problem is that AdaBoost is not noise robust, and exhibits sub optimal generalization ability in the presence of noise. We call "noisy" data which have at least one of the following properties: (a) overlapping class probability distributions, (b) outliers, (c) mislabeled patterns. On such data, by increasing the minimum margin of a few examples, AdaBoost also globally reduces the margin of the rest of the examples.

This phenomenon is called overfitting. Overfitting is encountered more generally with other learning algorithms when the hypothesis class H is too "large" or "complex" relative to the size of the training set. This means that the hypothesis accounts more for the individual instances of the specific training set used than it represents the distribution underlying it. In AdaBoost, the complexity of the final hypothesis is increased at each step by adding a different weak hypothesis. This implies that typically, the generalization error stops to decrease after some iterations, and increases again as the hypothesis becomes too complex.

This is related to a much more general feature of all nonparametric statistical inference methods. It is called the bias and variance dilemma, exposed by Geman and Bienenstock [15]. The basic idea is the following. For a given learning task, with a given finite amount of training data, the best generalization performance will be achieved if the right balance is struck between the accuracy attained on a particular training set, and the "capacity" of the machine, that is, the ability of the machine to learn any training set with minimal error. A machine with too much capacity (high variance) is like a botanist with a photographic memory who, when presented with a new tree, concludes that it is not a tree because it has a different number of leaves from anything she has seen before. A machine with too little capacity (high bias) is like the botanist's lazy brother, who declares that if it is green, it is a tree. Neither can generalize well.

The asymptotic (large sample) convergence of an estimator to the optimal estimation is called consistency. The general recipe for obtaining consistency is to slowly remove bias. This procedure is somewhat delicate, since the variance must also be decreased simultaneously, which dictates a gradual reduction of bias. In the framework of boosting, the bias measures the error that would remain even if we had an infinite number of independently trained classifiers. The variance measures the error that is due to fluctuations that are part of generating a single classifier. The idea is that by averaging over many classifiers one can reduce the variance. In the case of stumps, boosting acts also a bias reduction process, as adding a new stump increases the complexity of the final hypothesis.

To illustrate the bad performance of a margin classifier in case of noisy data, we analyze the toy example on figure (2-2). The left sketch shows the case without noise, where the optimal separating hyperplane is estimated correctly. In the middle we have an outlier, on which AdaBoost will concentrate its weights. A single point spoils the whole estimation. On the left side of figure (2-2) a more complex decision line is considered. The overfitting problem becomes even more critical if more and more complexity is created through combining more and more hypotheses. Then all training patterns, even mislabeled can be classified correctly. But the decision line becomes too complex, and this high variance gives bad generalization performance.

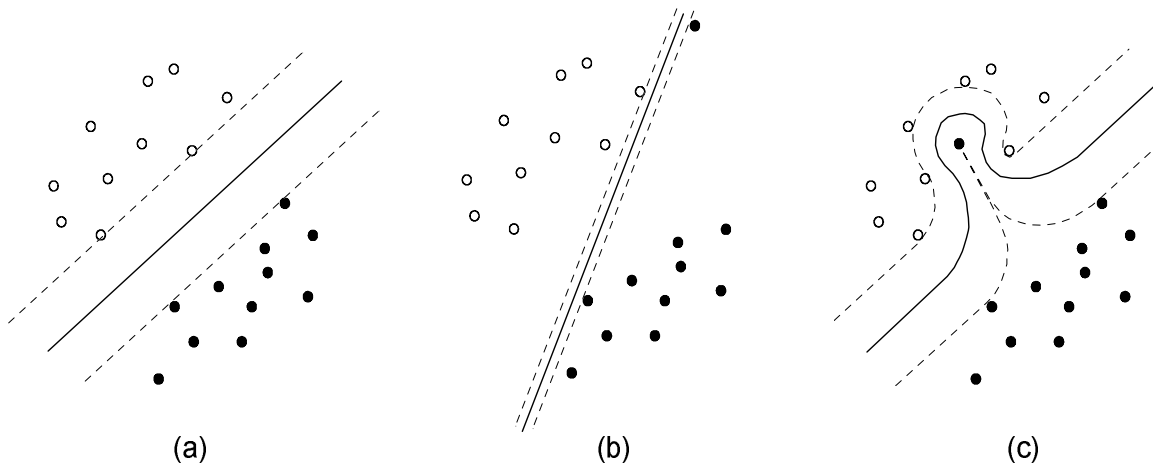

Figure 2-2 : The problem of finding a maximum margin classifier on reliable and not overlapping data (a), data with an outlier (b), and with a mislabeled pattern (right). The solid line shows the decision border and the dashed lines mark the margin area.

From these cartoons, it becomes apparent that AdaBoost is noise sensitive, and maximizing the smallest margin in the case of noisy data can lead to bad results. Therefore, we need to allow for a possibility of mistrusting the data.

It is indeed not obvious from bound (2.4-1) that the smallest margin should be minimized, because it takes the whole margin distribution into account. If we allow a non-zero training error, then the first term on the right hand side of (2.4-1) becomes non-zero. But then θ is larger such that the second term is much smaller. The tightest bound can be achieved at a θ greater than the smallest margin. This shows that a classifier aimed at minimizing strongly the margin is not always optimal.

Recent papers of Jiang [18] provide interesting results on AdaBoost applied to noisy data. They prove that because the training error is bound to converge, the variance of the resulting classifier becomes too after a number of iterations, which implies overfitting. On another hand, they show that the best prediction generated during the boosting process (before overfitting occurs) is consistent.

The problem is that identifying the optimal stopping point is difficult. Some modifications of AdaBoost have been proposed to find algorithms converging to the optimal predictions. Freund Mansour and Schapire [9] show that allowing AdaBoost to abstain for the examples with margins smaller than a given threshold give reliable results. Jiang [19] shows that the quantization of continuous attributes allows asymptotic consistency. Raetsch [31] obtains good generalization by aiming AdaBoost at finding an hypothesis with margin greater than a given ρ, not only positive margins.

Raetsch [31] also introduced the concept of *soft margin*, which allows the possibility of mistrusting part of the data. In this approach, the G error functional (1.6-1) is changed by introducing a new term. This term modifies the margin to prevent AdaBoost from concentrating its weights on a few examples. We will focus on this approach, because it integrates well in the framework we set in section 1.6.

2.4.2 Soft margin AdaBoost

In this paragraph, we restrict the developments to the case of self-weighted weak hypotheses. First we define the *influence* of a pattern to the final hypothesis according to Raetsch [31]

$$\mu_t(z_i, h) = \sum_{r=1}^{t} \frac{h_r(x_i)}{|h|_t} w(z_i),$$

where $|h|_t$ represents the 1-norm of h, i.e. $|h|_t = \sum_{r=1}^{t} |h_r|$, the sum of the amplitudes of the weak hypotheses contributing up to iteration t. The influence is the (weighted) average voting weight of a pattern computed during the boosting process. A pattern which is very often misclassified (i.e. difficult to classify) will have a high average weight, and thus a large influence.

We define the *soft margin* as a tradeoff between the margin and the influence of a pattern to the final hypothesis as follows

$$\widehat{mg}(z_i, h) = mg(z_i, h) + C|h|_t \mu_t(z_i, h)^p,$$

where $C \geq 0$ and p are fixed parameters of the learning algorithm. Note that it is important to keep the regularizing factor in the expression above of the same order of magnitude than the margin. For any example, the margin is always bound between $-|h|$ and $|h|$. It is equal to $-|h|$ if the instance is misclassified by all weak hypotheses. Thus, we want the regularizing term to grow as $|h|$ for all choices of p. This explains the construction of μ_t and $\widehat{mg}(.,.)$.

We can now run AdaBoost with the new aim of maximizing the soft margin. This is achieved by minimizing the regularized version of the error functional defined in (1.6-1)

$$\hat{G}_t(h) = \sum_{i=1}^{N} \exp\left\{-\widehat{mg}_t(z_i, h)\right\}$$

The choice of a weak hypothesis is not as easy as in the standard case, and we will address this point in a more general framework in a following section.

2.4.3 Cost-sensitive boosting

In some samples, there are strongly uneven proportions of the different labels. This is also the case when a multilabel problem is split in several binary problems through the one-versus-all approach. The more classes in the multilabel problem, the weaker the proportion of positive labels in each binary problem. It is then more important to identify a minority instance, at the cost of missing some of the most frequent. Here comes the idea of attributing different misclassification costs depending on the different types of mistakes possible.

Fan et al. [4] address the problem in the context of fraud detection. In this application, it is obviously much more important not to miss any fraud, at the cost of having some regular examples considered suspicious. These authors present an adapted version of AdaBoost that performs cost-sensitive boosting.

The most straightforward way to reduce the misclassification cost is to give the costly examples higher initial weights. The weak learner's goal is to minimize the weighted error for this initial distribution. however, after a few steps of boosting, the advantage given to costly example vanishes, because the weight updating rule does not take the costs into account. Furthermore, only one cost can be attributed to one label indifferently whether it has been correctly classified or not.

To overcome these shortcomings, the notion of misclassification cost has to be included in the error functional G underlying AdaBoost. The resulting weight updating rule brings higher weights for the expensive examples and comparatively lower weights for inexpensive examples. Furthermore, this weight updating rule increases the weights of costly wrong classifications more aggressively, but decrease the weights of costly correct classifications more conservatively. Each weak hypothesis in the boosting process correctly predicts more expensive examples for such a distribution. The final voted ensemble will also correctly predict more costly instances.

In Fan et al.'s [4] framework, each instance z_i is associated with a cost factor $c_i \in \mathbb{R}^+$. They define a cost adjustment function $\beta(\text{sign}(y_i h_t(x_i)) \ , \ c_i)$ 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. The margin of an example is then

$$\widehat{\widetilde{mg}}(z_i, h, c_i) = mg(z_i, h)\beta(\text{sign}(y_i h_t(x_i)) \ , \ c_i).$$

The goal is now to uniformly minimize the cost-sensitive margin. This is achieved by a version of AdaBoost built on the following error functional:

$$\hat{\widehat{G}}_t(h) = \sum_{i=1}^{N} \exp\left\{-\widehat{\widetilde{mg}}_t(z_i, h, c_i)\right\}.$$

The weight updating rule is again the derivative of $\hat{\widehat{G}}_t(h)$ with respect to the margin. This gives explicitly

$$w_{t+1}(z_i) = \frac{1}{\widehat{\widehat{Z}}_t} w_t(z_i) \exp\{-y_i h_t(x_i)\beta(i)\},$$

where $\beta(i) = \beta(\text{sign}(y_i h_t(x_i)) \ , \ c_i)$, and $\widehat{\widehat{Z}}_t$ is the normalization factor chosen such that W_{t+1} is a distribution.

Fan et al. [4] derive a bound of the training error for cost-sensitive boosting similar to theorem 1.

$$\sum_{i=1}^{N} c_i I[H(x_i) \neq y_i] \ \leq \ \left(\sum_i c_i\right) \prod_{t=1}^{T} \widehat{\widehat{Z}}_t \qquad (2.4\text{-}2)$$

This tells us that the weak hypothesis can be found by minimizing $\widehat{\widehat{Z}}_t$ at each iteration as we did in section (1.7.2) with the standard Z.

2.4.4 Choosing a weak hypothesis in the general case

In this section we show the path to follow to find an optimal weak hypothesis in the most general case treated in this work. We consider a version of AdaBoost which operates with a soft margin, and which is at the same time is cost-sensitive. The error functional is

$$\tilde{\widetilde{G}}_t(h) = \sum_{i=1}^{N} \exp\left\{-y_i \sum_{r=1}^{t} h_r(x_i)\beta(i) - C|h|_t \mu_t(z_i, h)^p\right\}$$

This error function is based on a margin which is in some sense the most general because it has the form $\widetilde{mg}(z_i, h) = mg(z_i, h) \cdot A \ + \ B$, where A and B are specific to each example, and B is a nonlinear function of the whole history of the boosting process.

Now, choosing a weak hypothesis requires a little more attention than in the non-regularized case. First, the approach of Schapire and Singer doesn't apply because theorem 1 does not hold. We have seen its equivalent (2.4-2) if $C = 0$, i.e. in the pure cost-sensitive framework. But no such bound exists in the regularized case, because the soft margin allows for some error at each iteration. From this point of view, it seems that we cannot choose the weak hypothesis by minimizing some $\widetilde{Z}_t$ at each iteration. But we can get around this difficulty and reach the same goal by showing that $\tilde{\widetilde{G}}_t(h)$ can be expressed as the product of the $\widetilde{Z}_t$, following the scheme introduced in section 1.8.1. From that point, we can again target the weak learner to minimize $\widetilde{Z}_t$ at each iteration.

Proposition 4 : Given the notations above,

$$\widetilde{G}_t(h) = \prod_{r=1}^{t} \widetilde{Z}_r$$

where $\widetilde{Z}_t$ is the normalizing factor in the weight updating rule.

Proof: Similarly to what we did for the standard AdaBoost, we can derive an expression for the weights at iteration $t+1$ as the gradient of $\widetilde{G}_t(h)$ with respect to the soft margin $\widetilde{mg}_t(z_i, h)$.

$$w_{t+1}(z_i) = \frac{1}{\widetilde{\xi}_t} \exp\{-mg(z_i, h)\beta(i) - C|h|_t \mu_t(z_i, h)^p\}, \tag{2.4-3}$$

Again, $\widetilde{\xi}_t$ is a normalization factor assuring that the weights at iteration $t+1$ sum up to one. For any p we can derive an update rule based on the weights at the previous iteration.

$$w_{t+1}(z_i) = \frac{1}{\widetilde{Z}_t} w_t(z_i) \exp\left\{-y_i h_t(x_i)\beta(i) - C|h|_t \mu_t^p(z_i, h) + C|h|_{t-1} \mu_{t-1}^p(z_i, h)\right\}.$$

The terms concerning the last iteration in $\widetilde{G}_t$ can be taken out of the exponential

$$\widetilde{G}_t(h) = \sum_{i=1}^{N} \exp\left\{-y_i \sum_{r=1}^{t-1} h_r(x_i)\beta(i) - C|h|_{t-1} \mu_t(z_i, h)^p\right\} \cdot$$
$$\exp\left\{-y_i h_t(x_i)\beta(i) - C|h|_t \mu_t^p(z_i, h) + C|h|_{t-1} \mu_{t-1}^p(z_i, h)\right\}.$$

According to (2.4-3), the first exponential is the expression of $w_t(z_i)$ without the factor $\widetilde{\xi}_{t-1}$. Thus we get

$$\widetilde{G}_t(h) = \widetilde{\xi}_{t-1} \sum_{i=1}^{N} w_t(z_i) \exp\left\{-y_i h_t(x_i)\beta(i) - C|h|_t \mu_t^p(z_i, h) + C|h|_{t-1} \mu_{t-1}^p(z_i, h)\right\}.$$

By construction, $\widetilde{\xi}_{t-1} = \prod_{r=1}^{t-1} \widetilde{Z}_r$. We recognize in the remaining sum the exact expression of $\widetilde{Z}_t$, which yields the result.

■

Proposition 4 above allows the minimization of $\widetilde{G}_t$ to be done by minimizing at each step

$$\widetilde{Z}_t = \sum_{i=1}^{N} w_t(z_i) \exp\left\{-y_i h_t(x_i)\beta(i) - C|h|_t \mu_t^p(z_i, h) + C|h|_{t-1} \mu_{t-1}^p(z_i, h)\right\}.$$

At this point, we can return to the approach of Schapire and Singer described in section 1.7 to find an optimal weak learner for this booster. This implies dividing the input space into disjoint blocks $X_1, ..., X_M$ for which $h(x) = h(x')$ for all $x, x' \in X_j$. We can again denote $h^j = h(x)$ for $x \in X_j$. The weak learner outputs $h = 0$ for the examples that have missing values for the attributes needed to determine h, and the corresponding weight are grouped in W_0. Then $\widetilde{Z}_t$ can be rewritten as

$$\widetilde{Z}_t = W_0 + \sum_j \sum_{i: x_i \in X_j} w_t(z_i) \exp\{u(h^j, |h|_{t-1}, w_{1...t}(z_i))\}$$

The argument u of the exponential depends not only on h^j as before, but also on the history of the hypotheses and weights, so it is specific to each instance. Therefore, it is not possible to find an analytical form for h^j like (1.8-3). However, in the case of stumps, we want to find the optimal binary split, which requires opposite hypotheses $\text{sign}(h^1) = -\text{sign}(h^2)$. For a given splitting point, we can

compute numerically the h^j that minimizes $\widetilde{Z}_t$ on X_j supposing one time that it is positive and one time that it is negative. The polarity yielding the lowest $\widetilde{Z}_t$ wins.

The problem is that this has to be tested on all possible splitting points in the attribute space. This represents twice as many numerical minimizations of $\widetilde{Z}_t$. So, although it is theoretically possible to build the optimal weak hypothesis, it might represent a prohibitive cost in computational power.

To overcome this difficulty, one can use the weak learner of the not-regularized, not-cost-sensitive AdaBoost as introduced in section 1.8. Using a non-optimal weak hypothesis forces us to reintroduce a voting weight α in the algorithm to rescale h_t. Everything remains the same, except that a factor α is added preceding each occurrence of h_t in all expressions. Once the weak learner returned h_t, α is set by minimizing $\widetilde{Z}_t(\alpha)$ as in section (1.7.2). Thus, only one numerical minimization is necessary at each iteration, and the computational effort is of the same order as for the basic AdaBoost.

2.4.5 Favored boosting

If computing time is critical, one can devise an even more straightforward adaptation of AdaBoost for the cost-sensitive approach. To avoid the numerical minimization of Z, one can use the scaled hypotheses of the standard weak learner (not-cost-sensitive), and let them vote just as in the original AdaBoost. The only modification would be to replace the standard weight updating rule by the expression we found in section 2.5.3.

$$w_{t+1}(z_i) = \frac{1}{\widehat{\widehat{Z}}_t} w_t(z_i) \exp\{-y_i h_t(x_i)\beta(i)\}$$

Under this modification, the algorithm is not anymore consistent with the minimization of the loss functional $\widehat{\widehat{G}}$. Indeed h_t is not correctly scaled. But h_t is evaluated from the weights computed at the previous iteration. So, it takes the $\beta(i)$ into account, so to say one iteration later. This hybrid solution is not guaranteed to yield satisfactory results, and is certainly not optimal. It has to be tested on real data to assess its validity. Due to the lack of time we had to restrict ourselves to this poor variant in the experimental part. In the following, we call this variant of AdaBoost *favoring boosting* or *favoring stumps*, because it just favors the minority label by increasing its weight a bit at each iteration.

When no specific information about the cost of misclassification is available (which is always the case in this work), we simply choose the cost adjustment function to be a function of the positive to negative label ratio

$$\beta_+ = \frac{N_-}{N_+}; \qquad \beta_- = \frac{N_+}{N_-},$$

where β_+ is associated with a positive label, and vice-versa, as suggested in [4].

3 Experimental results on boosting

3.1 Implementation setup

In this section we shortly show how the algorithms described in the sections above are implemented. This part is not mandatory for the understanding of the rest of this report, but might be

useful to someone who wishes to reproduce some of the experiments. It also helps to understand the limitations imposed by the speed of the machine and of the programming language used.

3.1.1 The object structure

For the ease of graphical representation, the programming language chosen is `Matlab 6`. This software supports object oriented programming, and the problem at hand is well suited for an object approach. Figure 3-1 is an organization chart of the different classes involved in the construction of a booster.

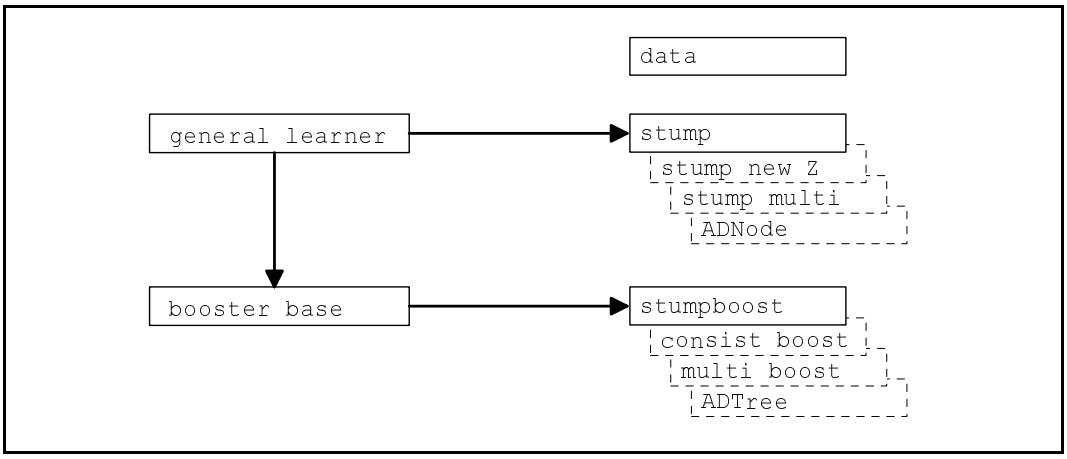

Figure 3-1 : The `Matlab` classes implemented for the boosting of stumps and all possible variants. The arrows mean "derived into" or "is a parent class of". Only the classes on the right side are actually instanciated.

The `data` class is not affiliated with any of the others, but it plays the role of a complex input argument to the weak learner and the booster classes. The fields of the `data` class include of course the attributes and the labels for each examples, possibly spread into distinct training and test sets. Additional fields contain temporary data from the preprocessing of the data. The methods of the `data` class include retrieving the basic data, splitting the data in a training and test set, and doing the preprocessing. Preprocessing and cross-validation will be discussed in a the next sections.

The `general learner` class contains everything common to all learning classes. Its fields include a set of weights for the data, and an hypothesis vector. The methods of the `general learner` allow to calculate the regular or weighted training error. From this class derive directly all the specific weak learners implemented here, such as `stump`. The fields of the class `stump` contain all the results of the learning, i.e. the most determinant attribute, the best splitting point of a continuous attribute or the best partition of a nominal attribute, and the values of the hypothesis for both leaves of the tree. The two main methods allow to train the learner on labeled examples, and to calculate the prediction on an unknown set of examples.

A booster is also a form of a learner. Therefore the general class `booster base` representing all boosters also derives from the `general learner` class. It is never instanciated as itself, but is a parent for all boosters, and gathers all fields and methods common to all boosters. Among these fields are: a library of `general learner` objects containing all the trained weak learners, and an array with the training errors at each step. The methods include various plot routines, for example to visualize the evolution of the training and test error.

From the `booster base` class derive all the specific boosters, such as `stumpboost` which are actually instanciated in the end. They contain the specific method for learning from the training set, and to calculate the prediction on an unknown dataset. These methods of course recursively call the corresponding methods of the weak learner objects.

The booster and the weak learner can take different forms, as shown on figure 3-1 by the dashed rectangles. To collaborate correctly, they must have corresponding types. For example the booster `ADTree` works with the weak learner `ADNode`, to build an alternating decision tree. `multi boost` works with `stump multi` for the multilabel case (not described in this paper). There are multiple variants depending on the Z criterion used and whether consistent, cost-sensitive, or regularized boosting is required. This structure allows a very simple handling of the learning procedure, and storage of the results.

3.1.2 Speed issues and preprocessing

At each iteration of boosting, the Z criterion as given in equation (1.8-4) or (1.9-3) has to be evaluated many times. It has to be calculated for each possible split of the data (ordinal attribute compared to a threshold value, or nominal attribute tested for belonging to a partition of values). Given a split, we have to evaluate sums of weights, conditioned on the split and label. This implies performing at least two tests for every example, which is very time consuming.

For a continuous attribute taking values $\{x_1; x_2; ...; x_m\}$, Z has to be evaluated at all possible pivots $(x_i + x_{i+1})/2$ for $i = 1...m-1$. If m is large, and if the training set is big, it implies too many tests. It is then necessary to discretize the range of continuous attributes to reduce the number of tests required. This can be done uniformly, by dividing the range of the attribute $[x_1, x_m]$ in a number of constant intervals. It is preferred however to use a non-uniform discretization that handles better outliers and provides more accuracy in the regions where a lot of examples are available. Setting the discretization bins as percentiles of the attribute distribution does the job. So, we can require that there is at most k tests done for a continuous attribute. It equivalent to set bins such that there are in average m/k examples in each bin.

For a nominal attribute, all possible binary partitions of its values have to be tested. If the attribute takes values $\{x_1; x_2; ...; x_m\}$, 2^{m-1}partitions have to be tested. This even more critical when we consider that for a given partition, testing if an example belongs to one subset requires testing equality with all members of this subset. Even the first step, determining all possible binary partitions, take significant computing time for m above 7 or 8.

This issue is overcome by preprocessing the data to minimize the amount of operations needed at each iteration, at the expense of memory. We decided to work with masks. A mask is an array of booleans having the same size as the training set. For each example, a 1 implies that the a given test is passed. In all iterations, the weak learner uses always the same tests, only the weights change. Having the masks for each test available in memory saves a lot of time. For example, to calculate the term $W_{y=+1}^{x_i>a}$ needed for the computation of Z (equation 1.8-4), the weak learner just calculates

$$W_{y=+1}^{x^j>a} = \sum_{i=1}^{n} w(x_i) . * M^{x^j>a}(i) . * M^{y=+1}(i)$$

where, $M^{x^j>a}$ is the mask for the test $(x^j > a)$, and $.*$ represents the element by element multiplication.

This approach allowed a gain of 70% in the speed of the algorithms. However, a shortcoming of `Matlab` prevents to take the full advantage of this method. The `Matlab` environment has no special data type for booleans and considers them as double precision numbers. This increases greatly the amount of memory necessary to store the masks, and also increases the time necessary to compute Z.

3.2 Binary datasets

To assess the performance of the different variants of the boosting algorithm we have proposed in the previous sections, we test them on a number of binary datasets. Multilabel datasets are more common in applications, but the multilabel classifiers are a superposition of binary problems. So binary experiments give useful information about the boosting process in general.

The performance is measured by cross validation as described above. For each test, we compute the average training and generalization errors at each iteration. We call the resulting curves the *error curves*. Note that for each dataset, the error curves of various learning algorithms are reported in adjacent graphs with the same scale, to allow comparison.

3.2.1 Synthetic datasets

These artificial data sets are made out of random numbers. The attributes are drawn from two different multivariate normal distributions, one for the positive label, and another for the negative label. The mean and the variance of these distributions can be set independently for each attribute.

This is of course not the most general case, because the examples belonging to a given class are clustered in a single bell curve , whereas they could be scattered between different groupings along the attribute range. It is also important to note that the value of an attribute is independent from the others, which is not always true. These artificial data sets also don't have any nominal attribute.

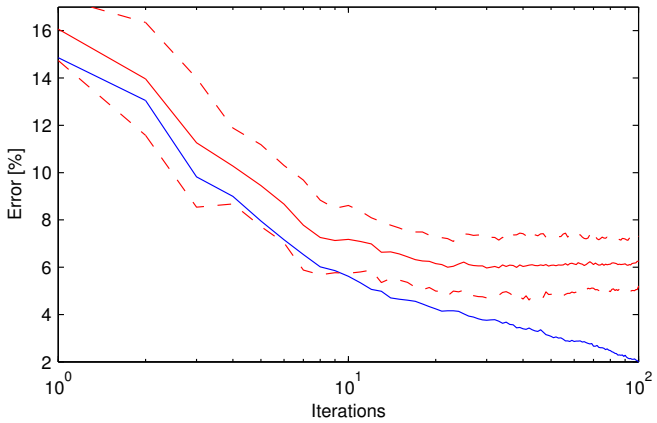

Figure 3-1 : Boosting 200 stumps on an synthetic dataset (see text for description). The blue and red solid lines represent the training and test errors respectively, averaged over 20 cross-validation runs. The corresponding dashed lines show intervals of two standard deviations above and below the average.

The dataset studied here is composed of an equal number of positive and negative labels and 8 attributes drawn from normal distributions with $\sigma = 1$, and means according to the following table

μ Label +1	0	0	0	0	0	0	0	0
μ Label -1	1	1	1	1	1	0.5	0.5	0.2

We show on figure 3-1 the evolution of the training and test errors during the boosting process. The solid curves show the errors $\overline{\varepsilon}_{\text{train}}$ and $\overline{\varepsilon}_{\text{test}}$ at each iteration averaged over 20 *trials* of

cross-validation. The training error decreases steadily as predicted by the theory. The test error first drops down similarly to the training error, but then stops to decrease. It is interesting to note that no or very little overfitting is observed on this dataset. This is probably due to the fact that the attributes are independent.

At each step, the standard deviation s_ε is calculated. A good representation of the spread of the trajectories is a confidence interval of two standard deviations around the average $\overline{\varepsilon} \pm 2s_\varepsilon$. Such an interval contains 95% of the errors if they are normally distributed around $\overline{\varepsilon}$. This interval is represented by dashed lines on figure (3-1). It shows how much the test error can vary from one assay to another. For clarity, we don't show the $\overline{\varepsilon} \pm 2s_\varepsilon$ curves on the following error plots in this work. Instead, we indicate as $\mathrm{CI} = 2s_\varepsilon$ the representative confidence interval averaged over the last iterations, to give just a sense of how the error trajectories spread.

To have an insight of how the boosting process operates, it is interesting to examine the distribution of the weights. Figure 3-2 tracks the evolution of this distribution by plotting its 25%, 50%, 75% and 95% quantiles as well as the maximum. To make the observations independent from the sample size, the weights are multiplied by it. Thus in the beginning, when all examples have the same weight, they all have value one. We see that the 25% and 50% quantiles are decreasing, corresponding to the lower and lower attention given by the booster to the easy-to-learn examples. The maximum of the weights distribution gives an idea of how the learner focuses on hard examples. If it goes too high, it means that the booster focuses on too few examples, and the rules found are not representative of the whole sample. This would lead to overfitting.

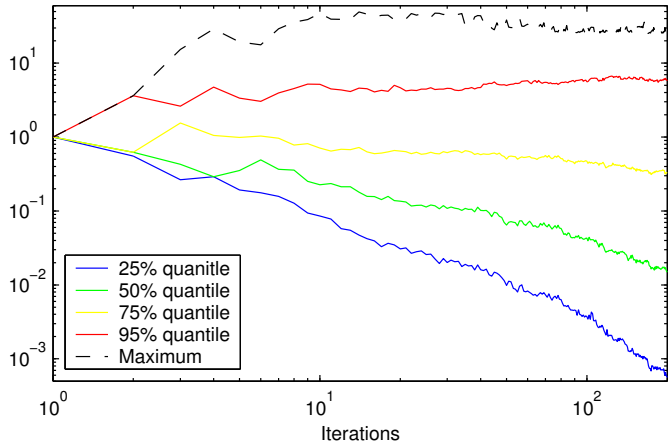

Figure 3-2 : Evolution of the weight distribution during the steps of boosting on the a synthetic dataset.

3.2.2 Project Management data

We consider here a project management database. In this section, we use the data as a workbench to assess the performance of our classifiers.

We focus on predicting the variable `Status`, which is a global appreciation of the project based on several appreciations of business managers and the reviewers. Here, the `Status` variable takes values in $\{1;2\}$.

Classifiers of four types are trained on the project management database.

- plain stumps as described in section 1.8

- stumps based on the New Z criterion (section 1.9)
- consistent stumps (section 2.2)
- alternating decision trees (section 2.3)

All graphs on figure 3-3 have the same scale to allow easy comparison. The ADT learner was stopped at 50 iterations, because it is very power consuming, and it already started to overfit. The training error has a similar behavior for all classifiers except the ADT, for which it drops much faster (under 10% at iteration 50). The test errors of the three first classifiers also have a similar behavior. With respect to the plain stumps, the New Z version achieves a slightly better performance, and seems to reach its minimum earlier. The consistent stumps achieve the best generalization error among the four boosters tested. Some precautions must apply to these observations, given the quite large confidence intervals.

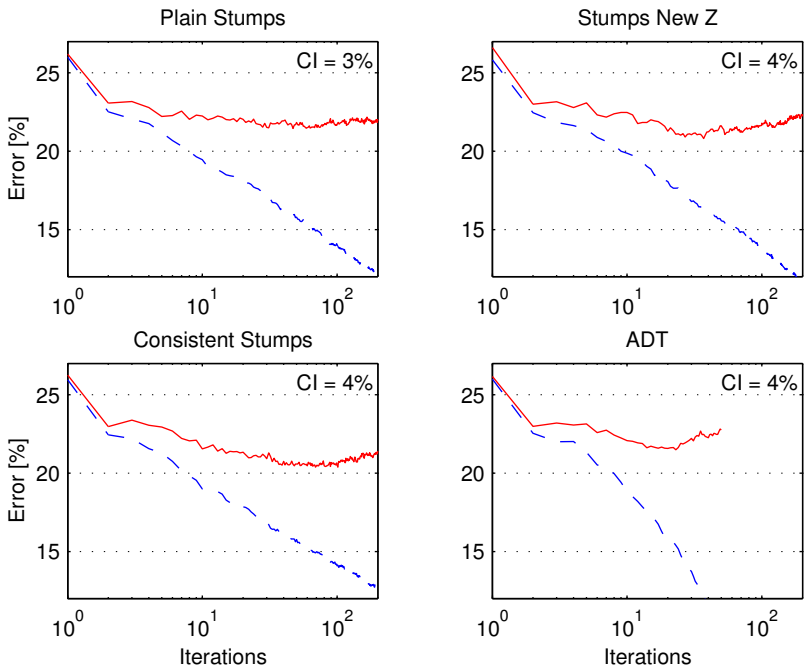

Figure 3-3 : Four different boosted stump methods applied on the a binary problem based on `Status`. The dashed line is the training error, and the plain line is the test error, both averaged on 20-fold cross validation. The spread (Confidence Intervalle) of the test error curve is given by CI in the upper right corner of each graph.

3.2.3 kr-vs-kp data

The `kr-vs-kp` binary dataset is taken from the UCI data repository [38]. It is composed of 3198 instances, each represented by 36 boolean attributes without any missing value. There are 52% of positive labels. The data is supposed to be very deterministic, because it represents the possible outcomes of a game of chess.

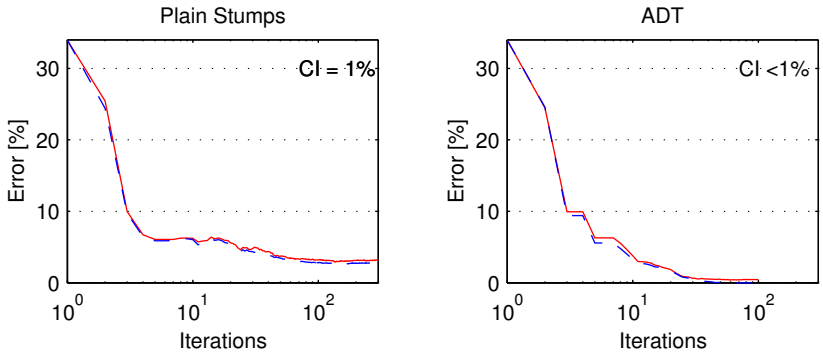

Figure 3-4 : Training and test errors of plain stumps and ADT on the dataset `kr-vs-kp`.

Boosting stumps on the `kr-vs-kp` binary dataset gives excellent convergence results. The training error reaches below 3% after 200 iterations. More interestingly, the test error has in average exactly the same behavior (with higher variations). This is a characteristic feature of an easy to learn dataset, without any noise or missing values. The performance of the ADT on this dataset is even better, as seen on the left side of figure 3-4. The test error reaches almost zero after 30 iterations already. We obtain exactly the same results as Freund and Mason [10].

3.2.4 cleve data

The `cleve` dataset is taken from the UCI data repository [38]. It is composed of only 303 examples with 13 attributes and a lot of missing values. There is about 54% of positive labels.

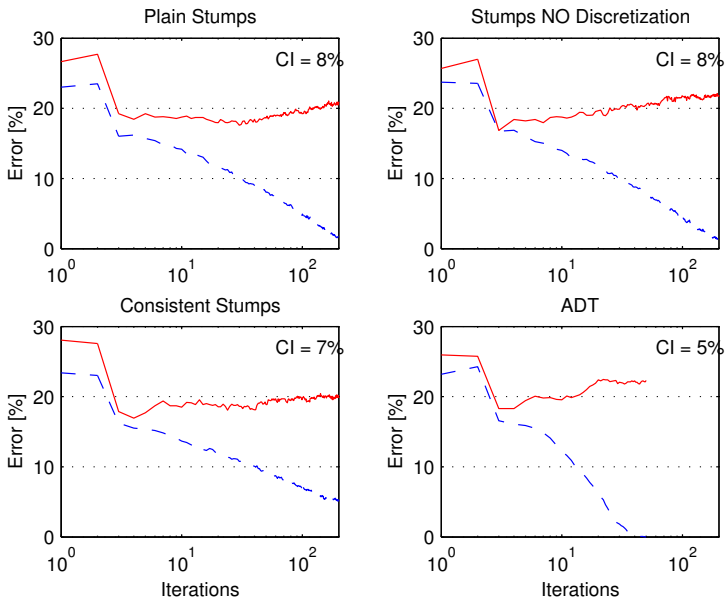

Figure 3-5 : Three different boosting methods applied to the `cleve` dataset. The dashed line is the training error, and the plain line is the test error, both averaged on 20 *trials*. The upper right graph represents plain stumps as the upper left one, but no discretization was used for the continuous variables.

As we see in figure 3-5, the `cleve` dataset is difficult to learn. the training error decreases steadily with all four boosting variants used. However, overfitting appears as early as the fourth iteration. We first consider the plain stumps, consistent stumps and ADT, all three trained with the usual discretization of the continuous attributes. The consistent stumps reach the lowest test error. The accuracy of the ADT is slightly worse, but overfitting is much more severe. The comparison between the three methods cannot be taken further, given the very high confidence intervals reported in figure 3-5.

The case of the plain stumps is interesting, because our experiment shows a higher generalization error than reported in Schapire and Singer [32] as well as in Freund and Mason [10]. We also have a much more gentle overfitting than these authors. The only difference between their work and ours could be in the discretization methods used (not well described in the above papers). For this reason we conducted an assay without any discretization, which could be done reasonably fast, given the small size of the dataset. The result is shown on the upper right graph of figure 3-5. This time, the error curves are similar to those found in the literature. The generalization error reaches a lower value, but the overfitting effect is stronger, and occurs very early. This is an illustration of the theoretical results of Jiang [18] implying that discretization prevents from overfitting.

3.2.5 hypothyroid dataset

This is a set of clinical data composed of 3163 examples having each 25 attributes. There are missing values among the attributes and there is only about 5% of positive labels, This dataset is also taken from the UCI data repository [38].

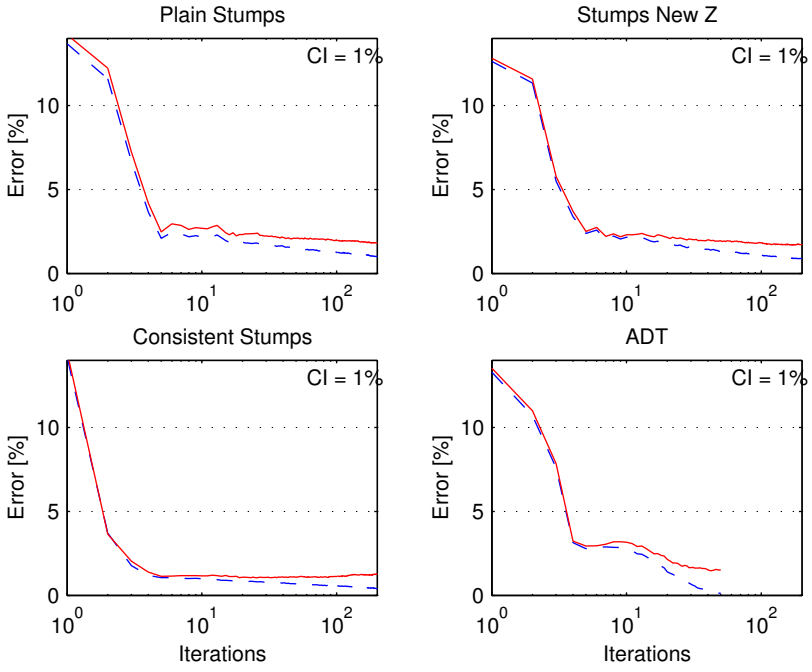

Figure 3-6 : Four different boosting methods applied to the `hypothyroid` dataset. The dashed line is the training error, and the plain line is the test error, both averaged on 20 trials.

The first statement we can make on the learning error curves in figure 3-6 is that the plain stumps and the stumps with new Z show an almost identical performance. The consistent stumps converge much faster and to the lowest generalization error obtained for this dataset. The ADT experiment had to be limited to 50 iterations, due to the size of the dataset. This is enough to bring the training error to zero, but the test error at this iteration is not much lower than the test error achieved by plain stumps. Whether the convergence would continue in further iterations would be worth investigating, as well as for the stumps. Anyway, the results shown here have noting in common with the results reported by Schapire and Singer [32] as well as in Freund and Mason [10]. The latter have obtained generalization errors of 1.1% after only 60 iterations for stumps, and 0.8% after 7 iterations for ADT's ! As in the case of the `cleve` dataset, the assumed cause for this is the discretization of the continuous attributes. It would be interesting to let the algorithms run for 1000 iterations or more, to see is they converge to errors as good as reported in the papers cited above. This would show that for datasets of this type, one can avoid overfitting, and still get accurate predictions, at the expense of larger trees.

3.3 Conclusions on the general boosting experiments

We summarize here the different findings of the experiments conducted with our boosting algorithms.

- The error curves obtained are similar to those reported by Schapire and Singer [32]. Some deviations are due to the discretization of the continuous attributes. Discretization has the effect of delaying the apparition of overfitting, at the expense of some accuracy at the optimal point.
- The stumps perform very well as compared to the much more complex ADT. The ADT allows the training error to decrease much faster, but overfitting occurs also much earlier. For ill determined or noisy datasets (such as the project management data), stumps perform as well as ADT's in terms of lowest generalization error achieved. ADT's show a better performance than stumps on datasets with no noise, where no overfitting occurs.
- The two Z criteria yield very similar results. This proves that both approaches are valid. The standard Z could be preferred however because it requires only half of computing time.
- Consistent boosting slightly outperforms all other methods. The generalization error reaches a lower level and seemingly at earlier iterations.
- The favoring stumps induce instabilities in the earlier iterations or in the entire learning process. This is due to the fact that the algorithm is not totally consistent as explained in section 2.5. The weak learner does not scale the hypotheses according to the misclassification cost, except through the weights of the previous iteration. This delay could be responsible for oscillations. It would be very informative to implement the complete cost-sensitive algorithm given in section 2.5.

4 References

[1] Allwein E. L., Schapire R. E., Singer Y., *Reducing Multiclass to Binary: A Unifying Approach for Margin Classifiers*, Journal of Machine Learning Research, 1:113-141, 2000.

[2] Breiman L., *Prediction Games and Arcing Algorithms*, Technical Report 504, Statistics Department, University of California, 1997.

[3] Breiman L., *Some Infinity Theory for Predictor Ensembles*, Technical Report 577, Statistics Department, University of California, 2000.

[4] Fan W., Stolfo S., Zhang J., Chan P.K., *AdaCost: Misclassification Cost-sensitive Boosting*, Proceedings of the Sixteenth International Conference on Machine Learning (ICML'99), pages 97-105, 1999.

[5] Freund Y., Schapire R. E., *A Short Introduction to Boosting*, Journal of Japanese Society for Artificial Intelligence, 14(5):771-780, 1999.

[6] Freund Y., Schapire R. E., *A Brief Introduction to Boosting*, Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999.

[7] Freund Y., Schapire R. E., *A Decision-theoretic Generalization of On-line Learning and an Application to Boosting*, Journal of Computer and System Sciences, 55(1):119-139, 1997.

[8] Freund Y., Schapire R. E., *Experiments with a New Boosting Algorithm*, Machine Learning: Proceedings of the Thirteenth International Conference, pages 148-156, 1996.

[9] Freund Y., Mansour Y., Schapire R. E., Iyer R., *Why Averaging Classifiers Can Protect against Overfitting*, Proceedings of the Eighth International Workshop on Artificial Intelligence and Statistics, 2001.

[10] Freund Y., Mason L., *The Alternating Decision Tree Learning Algorithm*, ICML-99, 1999.

[11] Freund Y., Schapire R. E., *Discussion of the paper Arcing Classifiers by Leo Breiman*, The Annals of Statistics, 26(3):824-832, 1998.

[12] Freund Y., Schapire R. E., *Game Theory, On-line Prediction and Boosting*, Proceedings of the Ninth Annual Conference on Computational Learning Theory, pages 325-332, 1996.

[13] Freund Y., Schapire R. E., *Adaptive Game Playing using MultiplicativeWeights,* Games and Economic Behavior, 29:79-103, 1999.

[14] Friedman, J. H., Hastie T., Tibshirani R., *Additive Logistic Regression: a Statistical View of Boosting*, Technical Report, Department of Statistics, Stanford University, 1998.

[15] Geman S., Bienenstock E, Doursat R., *Neural Networks and the Bias/Variance Dilemma*, Neural Computation, MIT Press, 4:1-58, 1992.

[16] Holte R. C., *Very Simple Classification Rules Perform Well on Most Commonly Used Datasets*, Machine Learning, 3: 63-91, 1993.

[17] Iba W., Langley P., *Induction of One-level Decision Trees*, Proceedings of the Ninth International Machine Learning Conference. Aberdeen, Scotland: Morgan Kaufmann, 1992.

[18] Jiang Wenxin, *Some Theoretical Aspects of Boosting in the Presence of Noisy Data*, Proceedings: The Eighteenth International Conference on Machine Learning, Morgan Kaufmann, ICML-2001, 2001.

[19] Jiang Wenxin, *Is Regularization Unnecessary for Boosting?*, Technical Report 00-04, Department of Statistics, Northwestern University, 2000.

[20] Jiang Wenxin, *Process Consistency for AdaBoost*, Technical Report 00-05, Department of Statistics, Northwestern University, 2000.

[21] Kégl B., Linder T., Lugosi G., *Data-dependent Margin-based Generalization Bounds for Classification*, COLT 2001: The Fourteenth Annual Conference on Computational Learning Theory, 2001.

[22] Loh W.-Y., Lim T.-S., Shih Y.-S., *A Comparison of Prediction Accuracy, Complexity, and Training Time of Thirty-three Old and New Classification Algorithms*, Machine Learning Journal, 40:203-228, 2000.

[23] Mansour Y., McAllester D., *Generalization Bounds for Decision Trees*, COLT 2001: The Fourteenth Annual Conference on Computational Learning Theory, 2001.

[24] Mitchell R. A., *Boosting Stumps from Positive Only Data*, Technical Report UNSW-CSE-TR-9907, 1999.

[25] Quinlan J., *Boosting, Bagging, and C4.5*, In Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, pages 725-730, 1996.

[26] Quinlan R., *MiniBoosting Decision Trees*, Journal of Artificial Intelligence Research, 1998.

[27] Rätsch G., Schökopf B., Smola A., Müller K.-R., Onoda T., and Mika S., *nu -Arc: Ensemble Learning in the Presence of Outliers*, Advances in Neural Information Processing Systems 12: Proc. of NIPS'99. MIT Press, pages 561-567, 2000.

[28] Rätsch G., Onoda T., Müller K.-R., *Regularizing AdaBoost*, Advances in Neural Information Processing Systems 11: Proc. of NIPS'98, MIT Press, pages 564-570, 1999.

[29] Rätsch G., Warmuth M.K., *Marginal Boosting*, NeuroCOLT2 Technical Report 97, Royal Holloway College, London, 2001.

[30] Rätsch G., Onoda T., Müller K.-R., *Soft Margins for AdaBoost*, Machine Learning, 42(3):287-320, 2001.

[31] Schapire R. E., Singer Y., *Improved Boosting Algorithms using Confidence-rated Predictions*, Machine Learning, 37(3):297-336, 1999.

[32] Schapire R. E., Freud Y., Bartlett P., Lee W. S., *Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods*, The Annals of Statistics, 26(5):1651-1686, 1998.

[33] Schapire R.E., Freund Y., Mansour Y., *Why Averaging Classifiers Can Protect against Overfitting*, In Proceedings of the Eighth International Workshop on Artificial Intelligence, 2001.

[34] Shafer J., Agrawal R., Mehta M., *SPRINT: A Scalable Parallel Classifier for Data Mining*, Proceedings of the 22nd VLDB Conference, Bombay, India, 1996.

[35] Shawe-Taylor J., Cristianini N., *On the Generalisation of Soft Margin Algorithms*, NeuroCOLT Technical Reports, 2000-082, 2000.

[36] Tumer K., Bollacker K., Ghosh J., *A Mutual Information Based Ensemble Method to Estimate Bayes Error*, Intelligent Engineering Systems Through Artificial Neural Networks, Vol. 8, ASME Press, (Proc ANNIE '98), 17-22, 1998.

[37] Merz, C. J., Murphy, P. M., *UCI repository of machine learning databases*. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[38] Schapire R. E., *Theoretical Views of Boosting and Applications*, Tenth International Conference on Algorithmic Learning Theory, 1999.