Journal of Computer and System Sciences
doi:10.1006/jcss.2001.1796, available online at http://www.idealibrary.com on IDEAL®

Boosting using Branching Programs

Yishay Mansour

Department of Computer Science, Tel-Aviv University, Ramat Aviv, Tel Aviv 69978, Israel
E-mail: mansour@cs.tau.ac.il

and

David McAllester

AT&T Research
E-mail: dmac@research.att.com

Received October 28, 2000; revised March 22, 2001

It is known that decision tree learning can be viewed as a form of boosting. Given a weak learning hypothesis one can show that the training error of a decision tree declines as $|T|^{-\beta}$ where $|T|$ is the size of the decision tree and β is a constant determined by the weak learning hypothesis. Here we consider the case of decision DAGs—decision trees in which a given node can be shared by different branches of the tree, also called *branching programs* (BP). Node sharing allows a branching program to be exponentially more compact than the corresponding decision tree. We show that under the same weak learning assumption used for decision tree learning there exists a greedy BP-growth algorithm whose training error is guaranteed to decline as $2^{-\beta\sqrt{|T|}}$, where $|T|$ is the size of the branching program and β is a constant determined by the weak learning hypothesis. Therefore, from the perspective of boosting theory, branching programs are exponentially more efficient than decision trees.

1. INTRODUCTION

Boosting algorithms have proven to be very powerful in computational learning theory. They are based on the assumption that natural sets of "base predicates" have the property that for any sample there exists a base predicate performing better than random guessing on that sample. This apparently modest assumption is quite powerful. It implies the ability to construct highly accurate decision rules built from the base predicates, e.g., decision trees or weighted threshold functions. In practice, the boosting algorithms, such as AdaBoost [FS95], have proven to be very successful, and are widely used by practitioners in the Machine Learning

0022-0000/02 $35.00

community. One can also show that the popular decision tree algorithms, such as CART and C4.5, can be viewed as boosting algorithms [KM96].

For the case of decision tree learning, tree learning requires a tree size that grows exponentially in $1/\delta$, where δ is the bias of the weak learning hypothesis. This exponential growth can be shown to be inherent to using a decision tree representation (for example, in the case of majority functions). In this work we attempt to overcome this obstacle by using a different representation, branching programs rather than decision trees.

A branching program is a directed acyclic graph, where each non-terminal node has a predicate and each terminal node has a label. Similar to decision trees, given an input we traverse a path in the graph; in each non-terminal node we select an outgoing edge using its predicate, and when we reach a terminal node its label classifies the input.

Branching programs are very powerful. Even if one restricts them to constant width, still one can represent any polynomial size formula (over the basis of AND, OR, and NOT) by a width 5 polynomial depth branching program [Bar86]. Very few and limited positive results are known for learning branching programs [RW93, EKR95, BTW96, BBTV97]. This should be of no surprise since branching programs are a generalization of decision trees and with the same number of nodes can represent significantly more powerful functions.

Here we develop boosting algorithms based on branching programs. The basic technique of building the branching program is very similar to the one used in decision trees—a greedy algorithm based on an index function. Similar to decision trees, we use the index function to select how to split a node. Unlike in decision trees, we need to also merge nodes together. We perform the merging based on the fraction of examples labeled one in each node. Namely, nodes with similar fraction of ones are likely to be merged. (At first this may look unnatural, but one can view this as an attempt to purify the nodes and drive the index function down to zero.)

We show that our simple greedy algorithm has very interesting theoretical properties. The training error of our branching program T is bounded by $exp(-\Omega(\gamma\sqrt{|T|}))$, where $|T|$ is the number of nodes in the branching program and γ is a parameter that depends quadratically on the bias δ of the weak learning hypothesis. This is a great improvement over the decision tree results, and the bound is only quadratic in the lower bounds.

One should take the theoretical results with a grain of salt when applying them to real problems. Although the theoretical results for decision trees are exponentially weaker than for AdaBoost, in practice they exhibit very similar performance [DKM96]. One explanation, studied in [DKM96], is that while in the decision tree the bias δ remains relatively stable as we grow the tree, the bias in AdaBoost is driven down very rapidly. Although we did not do any experiments, we believe it would be interesting to compare the branching program techniques presented in this work to existing decision tree and boosting algorithms.

The paper is organized as follows. In Section 2 we define the learning model, branching programs, the weak learning hypothesis, and the weak index reduction hypothesis. We describe the algorithm and analyze it in Section 3. Section 4 concludes with a summary and open problems.

2. PRELIMINARIES

2.1. Learning Model

We assume a set $\mathscr{X}$ of instances. A "training set" is a finite set S of pairs $\langle x, y \rangle$ with $x \in \mathscr{X}$ and $y \in \{0, 1\}$. Given a training set S and a function f from $\mathscr{X}$ to $\{0, 1\}$ we define the training error of f, denoted $\hat{\varepsilon}(f)$, to be the fraction of pairs $\langle x, y \rangle \in S$ such that $f(x) \neq y$.

In this paper we will not attempt to analyze the generalization error of the rules learned by our algorithm. Rather, we address the question of how rapidly the training error can be driven down as a function of the size of the branching program. Overfitting can be avoided in various ways, e.g., by bounding the allowed size of the branching program or by using holdout data to measure the generalization error of branching programs of various sizes. For a given size limit, smaller training error seems preferable. Hence we are interested in minimizing training error as a function of program size.

2.2. Branching Programs

We let $\mathscr{H}$ be a set of predicates on $\mathscr{X}$. An $\mathscr{H}$-BP is a directed acyclic graph whose nodes are divided into leaf nodes and internal nodes. Leaf nodes have no outgoing edges and each internal node is labeled with a predicate in $\mathscr{H}$ and has exactly two outgoing edges corresponding to the two possible truth values of the predicate. A given instance $x \in \mathscr{X}$ determines a unique directed path through an $\mathscr{H}$-BP T starting at the root of T and following the outgoing arc from internal nodes indicated by the value of the predicate at that node on x. (Note that a decision tree is a special case of a branching program where all the nodes, except the root, have only one incoming edge.) For any $\mathscr{H}$-BP T we let $N(T)$ denote the set of all nodes of T (both internal and leaf) and we let $L(T)$ denote the set of leaf nodes of T.

We say x reaches $n \in N(T)$ if n is on the path through T defined by x. For a given $\mathscr{H}$-BP T, node $n \in N(T)$, and sample S, we write S_n to denote the set of pairs $\langle x, f(x) \rangle$ in S such that x reaches n. For $n \in N(T)$ we define $\hat{p}_n$ to be the fraction of the sample reaching node n, i.e., $|S_n|/|S|$. For any sample W, where typically W is a subset of S, we define $\hat{q}(W)$ to be the fraction of the pairs $\langle x, f(x) \rangle$ in W for which $f(x) = 1$. For $n \in N(T)$ we define $\hat{q}(n)$ to be $\hat{q}(S_n)$. The training error of T, denoted $\hat{\varepsilon}(T)$, is defined as

$$\hat{\varepsilon}(T) \equiv \sum_{\ell \in L(T)} \hat{p}_\ell \min(\hat{q}_\ell, 1 - \hat{q}_\ell).$$

We define $|T|$ to be the number of nodes of T, i.e., $|T| = |N(T)|$.

2.3. The Weak Learning Hypothesis and Boosting

Here, as in [KM96], we view top-down decision tree learning as a form of Boosting [Sch90, Fre95]. Boosting describes a general class of iterative algorithms based on a weak learning hypothesis [Kea88, Sch90, Fre95]. The weak learning hypothesis applies to classes of Boolean functions. For $\delta > 0$ the δ-weak learning hypothesis for a hypothesis class $\mathscr{H}$ states that for any distribution on $\mathscr{X}$ there exists an $h \in \mathscr{H}$ with $Pr_D(h(x) \neq f(x)) \leqslant 1/2 - \delta$, where f is the target function. Algorithms designed to exploit this particular hypothesis for classes of Boolean functions have proved to be quite useful in practice [FS95].

Kearns and Mansour [KM96] show that the key to using the weak learning hypothesis for decision tree learning is the use of a continuous index function $I: [0, 1] \rightarrow [0, 1]$ such that $I(0) = I(1) = 0$, $I(q) \geqslant \min(q, (1-q))$, $I(q)$ is monotonically increasing on the interval $[0, \frac{1}{2}]$ and monotonically decreasing on the interval $[\frac{1}{2}, 1]$. For any branching program T, we define $I(T)$ to be

$$I(T) = \sum_{\ell \in L(T)} \hat{p}_\ell I(\hat{q}(\ell)).$$

Note that these conditions imply that $\hat{\varepsilon}(T) \leqslant I(T)$. For any $h \in \mathscr{H}$ let T_h be the decision tree consisting of a single internal node labeled with h and two leaves corresponding to the possible values of h. Let $I_W(T_h)$ denote the value of $I(T_h)$ as measured with respect to the sample W. Let $\Delta(W, h)$ denote $I(\hat{q}(W)) - I_W(T_h)$ (recall that $\hat{q}(W)$ is the fraction of pairs $\langle x, f(x) \rangle \in W$ such that $f(x) = 1$). The quantity $\Delta(W, h)$ is the reduction in the index for sample W achieved by introducing a single branch into a decision tree. Also note that $\hat{p}_\ell \Delta(S_\ell, h)$ is the reduction in $I(T)$ when the leaf ℓ of a decision tree is replaced by the branch h. Kearns and Mansour [KM96] prove the following lemma.

Lemma 2.1 (Kearns and Mansour). *Assuming the δ-weak learning hypothesis for $\mathscr{H}$, and taking $I(q) = 2\sqrt{q(1-q)}$, we have that for any sample W there exists an $h \in \mathscr{H}$ such that $\Delta(W, h) \geqslant (\delta^2/16)\, I(\hat{q}(W))$.*

This lemma motivates the following definition.

Definition 1. We say that $\mathscr{H}$ and I satisfy the γ-weak index reduction hypothesis if for any sample W from $\mathscr{X}$ there exists an $h \in \mathscr{H}$ such that $\Delta(W, h) \geqslant \gamma I(\hat{q}(W))$.

Note that in the above definition the parameter γ is proportional to δ^2, of the δ-weak learning hypothesis. The γ-weak index reduction hypothesis was used in [MM99] to study the effects of different split sizes for decision trees.

3. A BOOSTING ALGORITHM USING BRANCHING PROGRAMS

The nodes of the constructed branching program form a two dimensional grid of depth d and of varying width—for each depth j we have a width w_j. Each node has the form $n_{i,j}$ where i and j are integers such that $0 \leqslant j \leqslant d$ and $1 \leqslant i \leqslant w_j$. The graph is leveled; i.e., all arcs are from a node of the form $n_{i,j}$ to a node of the form $n_{i',j+1}$. The first level has only a single node $n_{1,0}$ which we take to be the root node of the branching program. Nodes at depth less than d are internal nodes and all nodes at depth d are leaf nodes. Given a branching program of depth d, and $1 \leqslant j \leqslant d$, we define T_j to be the branching program that results from deleting all nodes $n_{i,k}$ with $k > j$ so that the nodes $n_{i,j}$ become leaves. Our algorithm constructs T_{j+1} from T_j. Some nodes may be unreachable from the root and unreachable nodes can be discarded. We write $S_{i,j}$, $\hat{p}_{i,j}$ and $\hat{q}(i,j)$ for $S_{n_{i,j}}$, $\hat{p}_{n_{i,j}}$, and $\hat{q}(n_{i,j})$, respectively.

For each $j > 1$ we assume a sequence of values $u_{0,j}, \ldots, u_{w_j,j}$ with $u_{0,j} = 0$, $u_{w,j} = 1$ and $u_{i,j} < u_{i+1,j}$. For any $\hat{q} \in [0,1]$ we define $i(\hat{q}, j)$ to be 1 if $\hat{q} = 0$ and otherwise the least i such that $u_{i,j} \geqslant \hat{q}$. This gives $1 \leqslant i(\hat{q}, j) \leqslant w_j$ and $\hat{q} \in [u_{i(\hat{q},j)-1,j}, u_{i(\hat{q},j),j}]$. The algorithm maintains the following "bucket invariant" for all values of j:

Bucket Invariant: $\hat{q}(i,j) \in [u_{i-1,j}, u_{i,j}]$.

The algorithm is defined as follows.

ALGORITHM.

1. Define T_0 to consist of the single node $n_{1,0}$.
2. For j from 0 to $d-1$ define T_{j+1} as follows.
   - (a) Set $w_{j+1} = (9/\gamma)(\ln 1/I(T_j)) + c$, where the constant c will be specified latter.
   - (b) Select values $u_{0,j+1}, \ldots, u_{w_{j+1},j+1}$ as a function of w_{j+1} in a manner specified below.
   - (c) For each node $n_{i,j}$ reachable from the root
     - i. select a predicate h such that $\Delta(S_{i,j}, h) \geqslant \gamma I(\hat{q}(i,j))$;
     - ii. let $S^1_{i,j}$ and $S^0_{i,j}$ be the subsets of $S_{i,j}$ on which h is true and false respectively;
     - iii. and install edges from $n_{i,j}$ to $n_{i(\hat{q}(S^1_{i,j}),j+1),j+1}$ and $n_{i(\hat{q}(S^0_{i,j}),j+1)),j+1}$ for the true and false cases of h respectively.

We first note some basic invariants. Since we select a predicate with a strict reduction in the index function (for non-pure nodes), the sets $S^1_{i,j}$ and $S^0_{i,j}$ must both be non-empty. This implies that if node $n_{i,j}$ is reachable from the root then $S_{i,j} \neq \emptyset$, and vice versa.

We will write $n_{k,j} \xrightarrow{b} n_{i,j+1}$ to indicate that there is a branch from node $n_{k,j}$ to node $n_{i,j+1}$ corresponding to the truth value b. We now have

$$S_{i,j+1} = \bigcup_{n_{k,j} \xrightarrow{b} n_{i,j+1}} S^b_{k,j}.$$

By construction, for $n_{k,j} \xrightarrow{b} n_{i,j+1}$ we have $\hat{q}(S^b_{k,j}) \in [u_{i-1,j+1}, u_{i,j+1}]$. So we get $\hat{q}(i, j+1) \in [u_{i-1,j+1}, u_{i,j+1}]$ and the bucket invariant holds.

To ensure that $I(T_j)$ decreases on each iteration of the algorithm, it is important to select sufficiently finely spaced values $u_{0,j+1}, \ldots, u_{w_{j+1},j+1}$. For a given value w and set of values $u_0, \ldots, u_w$, and $\hat{q} \in [0,1]$, we define $i(\hat{q})$ as before and define $I^+(\hat{q})$ as

$$I^+(\hat{q}) \equiv \max_{x \in [u_{i(\hat{q})-1}, u_{i(\hat{q})}]} I(x).$$

The basic idea is that $I^+(\hat{q})$ "forgets" where $\hat{q}$ is in the interval $[u_{i(\hat{q})-1}, u_{i(\hat{q})}]$ and then assumes the worst case. Note that $I^+(\hat{q}) \geqslant I(\hat{q})$. To ensure a sufficiently fine spacing of the values $u_0, \ldots, u_w$ we use the following definition.

Definition 2. An (ε, λ)-net is a sequence of values $u_0, \ldots, u_w$ with $u_0 = 0$, $u_w = 1$, and such that for all $\hat{q} \in [0, 1]$ we have $I^+(\hat{q}) \leqslant \max(\varepsilon, (1+\lambda)\, I(\hat{q}))$.

Lemma 3.1. *For any index function I, and any $\varepsilon > 0$ and $\lambda \in (0, \frac{1}{2}]$, there exists an (ε, λ)-net with $w \leqslant 4 + 3\ln(1/\varepsilon)/\lambda$.*

Proof. We use the fact that an index function is a continuous function on the interval $[0, 1]$ such that $I(0) = I(1) = 0$, $I(1/2) = 1$, $I(q) \leqslant 1$, and $I(q)$ is monotonically increasing on $[0, \frac{1}{2}]$ and monotonically decreasing on $[\frac{1}{2}, 1]$. Define w to be the integer $2 + 2\lceil \frac{\ln(1/\varepsilon)}{\ln(1+\lambda)} \rceil$. Note that w is even so that $w/2$ bisects the possible values of i with $0 \leqslant i \leqslant w$. We define $u_{w/2}$ to be $\frac{1}{2}$. Now for $I \leqslant I(\frac{1}{2})$ we define $I_L^{-1}(I)$ to be the unique value of $\hat{q} \in [0, 1/2]$ such that $I(\hat{q}) = I$. For $1 \leqslant k \leqslant \lceil \ln(1/\varepsilon)/\ln(1+\lambda) \rceil$ we define $u_{w/2-k}$ as

$$u_{\frac{w}{2}-k} \equiv I_L^{-1}\left(\frac{I(1/2)}{(1+\lambda)^k}\right).$$

This gives $I(u_{w/2-k})/I(u_{w/2-k-1}) = 1 + \lambda$, for $1 \leqslant k < \lceil \ln(1/\varepsilon)/\ln(1+\lambda) \rceil$, which implies that for $\hat{q} \leqslant 1/2$ we have $I^+(\hat{q}) \leqslant (1+\lambda)\, I(\hat{q})$. Furthermore, we have

$$I(u_1) \leqslant (1+\lambda)^{-\ell} = e^{-\ell \ln(1+\lambda)}; \qquad \ell \geqslant \frac{\ln \frac{1}{\varepsilon}}{\ln(1+\lambda)}.$$

This gives $I(u_1) \leqslant \varepsilon$. Values of $u_{w/2+k}$ are defined analogously and analogous properties hold for $\hat{q} \geqslant 1/2$ and u_{w-1}. So we have an (ε, γ)-net with with $w = 2 + 2\lceil \frac{\ln(1/\varepsilon)}{\ln(1+\lambda)} \rceil$.

Finally we show that $w \leqslant 4 + \frac{3\ln(1/\varepsilon)}{\lambda}$. By the concavity of the ln function we have $\ln(1+\lambda) \geqslant \frac{\lambda}{z} \ln(1+z)$ provided $\lambda \leqslant z$. For $\lambda \leqslant \frac{1}{2}$ this gives

$$\begin{aligned} w = 2 + 2\left\lceil \frac{\ln \frac{1}{\varepsilon}}{\ln(1+\lambda)} \right\rceil &\leqslant 4 + 2\frac{\ln \frac{1}{\varepsilon}}{\ln(1+\lambda)} \\ &\leqslant 4 + \frac{\ln \frac{1}{\varepsilon}}{\lambda \ln(\frac{3}{2})} \\ &\leqslant 4 + \frac{3 \ln \frac{1}{\varepsilon}}{\lambda}. \end{aligned}$$

∎

Lemma 3.2. *If $u_{0, j+1}, \ldots, u_{w_{j+1}, j+1}$ forms an (ε, λ)-net then $I(T_{j+1}) \leqslant (1+\lambda)(1-\gamma)\, I(T_j) + \varepsilon$.*

Proof. Let $i(\hat{q})$ abbreviate $i(\hat{q}, j+1)$ and let u_i abbreviate $u_{i, j+1}$. Let $I^+(\hat{q})$ be defined in terms of the values $u_0, \ldots, u_{w_{j+1}}$. The bucket invariant implies that

$I^+(\hat{q}_{i,j}) = I^+(u_i)$. This implies $I(\hat{q}_{i,j}) \leqslant I^+(u_i)$ and we get the following where $\hat{p}^b_{k,j} = |S^b_{k,j}|/|S_{k,j}|$,

$$\begin{aligned}
I(T_{j+1}) &\leqslant \sum_{i=1}^{w_{j+1}} \hat{p}_{i,j+1} I^+(u_i) \\
&= \sum_{k=1}^{w_j} \hat{p}_{k,j} \sum_{b \in \{0,1\}} \hat{p}^b_{k,j} I^+(u_{i(\hat{q}(S^b_{k,j}))}) \\
&\leqslant \sum_{k=1}^{w_j} \hat{p}_{k,j} \sum_{b \in \{0,1\}} \hat{p}^b_{k,j} \max(\varepsilon, (1+\lambda)\, I(\hat{q}(S^b_{k,j})) \\
&\leqslant \sum_{k=1}^{w_j} \hat{p}_{k,j} \sum_{b \in \{0,1\}} \hat{p}^b_{k,j} (\varepsilon + (1+\lambda)\, I(\hat{q}(S^b_{k,j})) \\
&\leqslant \varepsilon + (1+\lambda) \sum_{k=1}^{w_j} \hat{p}_{k,j} \sum_{b \in \{0,1\}} \hat{p}^b_{k,j} I(\hat{q}(S^b_{k,j})) \\
&\leqslant \varepsilon + (1+\lambda) \sum_{k=1}^{w_j} \hat{p}_{k,j} (1-\gamma)\, I(\hat{q}(k,j)) \\
&\leqslant \varepsilon + (1+\lambda)(1-\gamma)\, I(T_j).
\end{aligned}$$

∎

Finally, Lemmas 3.1 and 3.2 imply the following.

LEMMA 3.3. *Assuming the γ-weak index reduction hypothesis, the width w_{j+1} and values $u_{0,j+1}, \ldots, u_{w_{j+1},j+1}$ can be selected in such a way that we have*

$$w_{j+1} \leqslant \frac{9}{\gamma} \ln\left(\frac{1}{I(T_j)}\right) + c$$

$$I(T_{j+1}) \leqslant \left(1 - \frac{\gamma}{2}\right) I(T_j),$$

where c is independent of j and is $O((1/\gamma)\ln(1/\gamma))$.

Proof. Select ε to be $\frac{1}{6}\gamma I(T_j)$ and select λ to be $\frac{\gamma}{3}$. Then let $u_{0,j+1}, \ldots, u_{w_{j+1},j+1}$ be an (ε, λ)-net, as guaranteed to exist by Lemma 3.1, with $w_{j+1} \leqslant 4 + \frac{3\ln(1/\varepsilon)}{\lambda}$. This gives the specified value for w_{j+1} and by Lemma 3.2 we have

$$I(T_{j+1}) \leqslant \left(1 + \frac{\gamma}{3}\right)(1-\gamma)\, I(T_j) + \frac{\gamma}{6} I(T_j) \leqslant \left(1 - \frac{\gamma}{2}\right) I(T_j).$$

∎

This implies the following.

LEMMA 3.4. *Assuming the γ-weak index reduction hypothesis, the algorithm can be run in such a way that*

$$|T_j| \leqslant \frac{9}{\gamma^2} L_j^2 + bL_j + 1,$$

where

$$L_j \equiv \ln(1/I(T_j))$$

and b is independent of j and is $O((1/\gamma^2)\ln(1/\gamma))$.

Proof. We first note the since $I(T_{j+1}) \leqslant (1-\gamma/2)\, I(T_j)$ we have

$$\begin{aligned} L_{j+1} &= \ln \frac{1}{I(T_{j+1})} \\ &\geqslant \ln \frac{1}{(1-\frac{\gamma}{2})\, I(T_j)} \\ &= L_j - \ln\left(1-\frac{\gamma}{2}\right) \\ &\geqslant L_j + \frac{1}{2}\gamma. \end{aligned}$$

We also have $w_{j+1} \leqslant \frac{9}{\gamma} L_j + c$ which gives

$$|T_{j+1}| \leqslant |T_j| + aL_j + c,$$

where a is the constant $9/\gamma$. We now consider any sequences $x_0, x_1, x_2, x_3, \ldots$ and $y_0, y_1, y_2, y_3, \ldots$ satisfying the following for constants δ, a, and c:

$$\begin{aligned} x_0 &\geqslant 0 \\ x_{i+1} &\geqslant x_i + \delta \end{aligned}$$

$$\begin{aligned} y_0 &\leqslant 1 \\ y_{i+1} &\leqslant y_i + ax_i + c. \end{aligned}$$

We will show that for any such pair of sequences we have

$$y_i \leqslant \frac{a}{2\delta} x_i^2 + \frac{c}{\delta} x_i + 1.$$

The proof is by induction on i. The base case follows from $x_0 \geqslant 0$ and $y_0 \leqslant 1$. For the induction case we have

$$\begin{aligned} y_{i+1} &\leqslant y_i + ax_i + c \\ &\leqslant \frac{a}{2\delta} x_i^2 + \frac{c}{\delta} x_i + 1 + ax_i + c \\ &\leqslant \frac{a}{2\delta}(x_{i+1}-\delta)^2 + \frac{c}{\delta}(x_{i+1}-\delta) + 1 + a(x_{i+1}-\delta) + c \end{aligned}$$

$$
\begin{aligned}
&= \frac{a}{2\delta} x_{i+1}^2 - a x_{i+1} + \frac{a\delta}{2} + \frac{c}{\delta} x_{i+1} - c + 1 + a x_{i+1} - a\delta + c \\
&\leqslant \frac{a}{2\delta} x_{i+1}^2 + \frac{c}{\delta} x_{i+1} + 1.
\end{aligned}
$$

The result follows by interpreting x_i as L_i and y_i as $|T_i|$. ▮

We now have that $|T_i|$ is $O(L_i^2/\gamma^2)$ which implies that L_i is $\Omega(\gamma\sqrt{|T_i|})$. So we have the following.

THEOREM 3.5. *Assuming the γ-weak index reduction hypothesis, the algorithm can be run in a way that satisfies*

$$I(T_j) \leqslant e^{-\Omega(\gamma\sqrt{|T_j|})}.$$

4. SUMMARY AND OPEN PROBLEMS

This work developed boosting algorithms using branching programs. The algorithms themselves are fairly natural greedy algorithms based on an index function. Using a weak index reduction hypothesis we are able to bound the error of the branching program as a function of its size. The training error drops exponentially in the square root of the size of the branching program. This is in contrast to decision trees, where the error drops only polynomially in the size of the decision tree. Hence there is an exponential gap in boosting between decision trees and branching program.

The algorithm that we proposed is not adaptive in the sense that it assumes a fixed γ. However, one can easily modify the algorithm to be adaptive by choosing for each $n_{i,j}$ the hypothesis h with the largest drop in the index function. We can define I_{j+1} to be the index function applied to the sets $S_{i,j}^b$, i.e., $I_{j+1} = \sum_k \hat{p}_{k,j} \sum_b \hat{p}_{k,j}^b \hat{q}(S_{k,j}^b)$. Now we can define the average reduction as $\gamma_j = (I(T_j) - I_{j+1})/I(T_j)$ and set the width as $w_{j+1} = O(1/\gamma_j \ln 1/I(T_j))$. Using an identical proof to Lemma 3.2, we have that $I(T_{j+1}) \leqslant (1+\lambda)(1-\gamma_j)\, I(T_j) + \varepsilon$. This implies that the index function at level k is bounded by

$$I(T_k) \leqslant \prod_{j=1}^{k} \left(1 - \frac{\gamma_j}{2}\right) I(T_0),$$

and the size of the branching program T_k is bounded by $|T_k| \leqslant \sum_{j=1}^{k} w_j$.

There are many challenging open problems for future research. First, in this work we concentrate on binary branching, namely, an internal node has at most two children. The understanding of how to perform splits of various size is left as a challenge for the future. For decision trees the understanding of how to compare splits of different sizes was done using a weak index reduction hypothesis [MM99]. One can hope that the techniques developed in [MM99] can be extended to branching programs and can address the issue of different split size in a branching program.

Second, it is not clear if the upper bound we derive for boosting using branching programs is the best possible. It would be interesting either to derive an improved upper bound or to exhibit a lower bound for boosting using branching programs.

Finally, there is the experimental aspect. It would be interesting to derive experimental results for our algorithms and compare them with existing decision tree and boosting algorithms.

REFERENCES

[Bar86] D. A. Mix Barrington, Bounded-width polynomial-size branching programs recognize exactly those languages in nc^1, *in* "Proceedings of the 18th Annual ACM Symposium on Theory of Computing, Berkeley, California, 1986," pp. 1–5.

[BBTV97] F. Bergadano, N. H. Bshouty, C. Tamon, and S. Varricchio, On learning branching programs and small depth circuits, *in* "Computational Learning Theory: Eurocolt '97," pp. 150–161, Springer-Verlag, New York/Berlin, 1997.

[BTW96] N. H. Bshouty, C. Tamon, and D. K. Wilson, On learning width two branching programs, *in* "Proc. 9th Annu. Conf. on Comput. Learning Theory," pp. 224–227, Assoc. Comput. Mach., New York, 1996.

[DKM96] T. Dietterich, M. Kearns, and Y. Mansour, Applying the weak learning framework to understand and improve c4.5, *in* ICML, 1996.

[EKR95] F. Ergün, R. S. Kumar, and R. Rubinfeld, On learning bounded-width branching programs, *in* "Proc. 8th Annu. Conf. on Comput. Learning Theory," pp. 361–368, Assoc. Comput. Mach., New York, 1995.

[Fre95] Y. Freund, Boosting a weak learning algorithm by majority, *Inform. Comput.* **121** (1995), 256–285.

[FS95] Y. Freund and R. E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, *in* "Proceedings of the Second European Conference on Computational Learning Theory, 1995," pp. 23–37.

[Kea88] M. Kearns, Thoughts on hypothesis boosting, unpublished, December 1988.

[KM96] M. Kearns and Y. Mansour, On the boosting ability of top-down decision tree learning algorithms, *in* STOC, 1996; *J. Comput. System Sci.* **58** (1999), 109–128.

[MM99] Y. Mansour and D. McAllester, Boosting with multi-way branching in decision trees, *in* NIPS, 1999.

[RW93] V. Raghavan and D. Wilkins, Learning μ-branching programs with queries, *in* "Proc. 6th Annu. Workshop on Comput. Learning Theory," pp. 27–36, Assoc. Comput. Mach., New York, 1993.

[Sch90] R. E. Schapire, The strength of weak learnability, *Mach. Learning* **5** (1990), 197–227.