Journal of Machine Learning Research 3 (2002) 271-301 Submitted 11/01; Revised 7/02; Published 10/02

On Online Learning of Decision Lists

Ziv Nevo ZIVN@CS.TECHNION.AC.IL
Ran El-Yaniv RANI@CS.TECHNION.AC.IL
Department of Computer Science
Technion
Haifa, 32000, Israel

Editor: Dana Ron

Abstract

A fundamental open problem in computational learning theory is whether there is an attribute efficient learning algorithm for the concept class of decision lists (Rivest, 1987; Blum, 1996). We consider a weaker problem, where the concept class is restricted to decision lists with D alternations. For this class, we present a novel online algorithm that achieves a mistake bound of $O(r^D \log n)$, where r is the number of relevant variables, and n is the total number of variables. The algorithm can be viewed as a strict generalization of the famous Winnow algorithm by Littlestone (1988), and improves the $O(r^{2D} \log n)$ mistake bound of Balanced Winnow. Our bound is stronger than a similar PAC-learning result of Dhagat and Hellerstein (1994). A combination of our algorithm with the algorithm suggested by Rivest (1987) might achieve even better bounds.

1. Introduction

Decision lists play an important role in computational learning theory, as well as in practical systems. Since their introduction by Rivest (1987), it was shown that k-decision lists generalize classes such as monomials, k-DNF, k-CNF and depth k decision trees (Rivest, 1987), as well as rank-k decision trees (Blum, 1992). Even the simple class of 1-decision lists coincides with fragments of important classes such as disguised Horn functions, read-once functions, threshold functions and nested differences of concepts (Eiter et al., 1998). Efficient algorithms for learning decision lists, are therefore of great importance. This article focuses on *online algorithms* and analyzes their *mistake bound*.

An important class of efficient learning algorithms are the *attribute efficient* algorithms. An attribute efficient algorithm is able to learn a target class, making a small number of errors, which depends only logarithmically on the total number of attributes in the examples vectors, and polynomially on the number of relevant attributes. Such algorithms were shown to be particularly useful on the important task of *feature selection*, where relevant attributes should be distinguished from many irrelevant ones.

Much effort was put in exploring the properties of attribute efficient algorithms. New algorithms were found for some wide concept classes, and hardness results were given to others (see Blum et al., 1995; Littlestone, 1989; Bshouty and Hellerstein, 1998, and more). However, it is still an open problem whether there is an attribute efficient learning algorithm for the concept class of decision lists (Rivest, 1987; Blum, 1996). Recent work

©2002 Ziv Nevo and Ran El-Yaniv.

(Servadio, 2000) indicates that the task of attribute-efficient learning of decision lists may be computationally hard.

We therefore look at the restricted problem of learning decision lists with D alternations. For this class we present a novel online algorithm that makes at most $O(r^D \log n)$ mistakes, where r is the number of relevant attributes (or the number of unique nodes in the decision list), and n is the total number of attributes in each example. This improves the best mistake bound, known so far, of $O(r^{2D} \log n)$, achieved by Balanced Winnow (Blum and Singh, 1990; Valiant, 1999).

Our algorithm, called Multi-Layered Winnow, makes use of multiple independent copies of the famous Winnow algorithm (Littlestone, 1988). Different copies are expected to learn concepts which correspond to different sections in the decision list (which we call *layers*). For that to happen, the predictions of the Winnows are evaluated in an appropriate way, and a special scheme is used to update Winnows that predicted incorrectly.

The paper is organized as follows. Section 2 contains the basic definitions and tools that are used in this article. In Section 3 we briefly discuss previous works, which relate to the online learning of decision lists. Then, in Section 4 we present Multi-Layered Winnow, analyze its mistake bound, and propose a way to combine it with Rivest's algorithm. Finally, in Section 5 we present a numerical example, illustrating some strengths and weaknesses of the proposed algorithms.

2. Preliminaries

In this section we introduce the learning model, the concept class of decision lists, and the Winnow algorithm, which is a basic component in our Multi Layered Winnow.

2.1 The Mistake Bound Model

Learning algorithms for boolean concept classes have a major role in computational learning theory. The task of such algorithms is finding a boolean function h, called the *hypothesis*, which is as "similar" as possible to another unknown boolean function, c, called the *target function*. To accomplish this task, a learning algorithm first receives the domain of c, which is usually $X_n = \{0,1\}^n$ for some n. Sometimes it also gets a set of boolean functions, $C \subseteq 2^{X_n}$ (called the *concept class*), to which c belongs. In addition, it is allowed to examine a number of *classified instances* of c. A classified instance of c is a pair $\langle \vec{x}, c(\vec{x}) \rangle$, where $\vec{x} \in X_n$ is called an *instance* and $c(\vec{x})$ is said to be its *classification*.

We will sometime refer to the boolean *attributes* $x_1, x_2, \ldots, x_n$, which define the instance. We say that a function c *depends* on an atrribute x_i, if there exist two instances $\vec{x}$ and $\vec{y}$ such that $x_j = y_j$ for all $j \neq i$ and $c(\vec{x}) \neq c(\vec{y})$. An attribute x_i is called a *relevant* attribute of c, if c depends on x_i.

To analyze the performances of learning algorithms, different models were suggested. We consider the *mistake bound* model. Within this model, a worst-case analysis is used to study the performance of *on-line learning algorithms*. This model is due to Littlestone (1988, 1989), and can be defined as follows. Let $X_n = \{0,1\}^n$ be the instance space (or the domain) and let $C \subseteq 2^{X_n}$ be a concept class. We consider two players, a learning algorithm $\mathcal{A}$ and an adversary. Both players know X_n and C in advance, and play the following game.

At first, the adversary chooses a target function $c \in C$, unknown to $\mathcal{A}$. The learning session now proceeds in trials, where at the t-th trial:

1. The adversary picks an instance $\vec{x}^{\,(t)} \in X_n$.
2. $\mathcal{A}$ outputs $h(\vec{x}^{\,(t)})$, where $h : X_n \to \{0, 1\}$ is its current hypothesis.
3. The adversary reveals the correct classification of the instance, $c(\vec{x}^{\,(t)})$.
4. If $h(\vec{x}^{\,(t)}) \neq c(\vec{x}^{\,(t)})$ then a single mistake is incurred by $\mathcal{A}$ (i.e., 0/1 loss).
5. $\mathcal{A}$ is allowed to update its hypothesis, in light of the new information.

The adversary can stop the game at any point.

We say that $M(c)$ is a *mistake bound* for $\mathcal{A}$ on some $c \in C$, if for any sequence of instances (possibly infinite), $\mathcal{A}$ never makes more than $M(c)$ mistakes while learning c. We say that $\mathcal{A}$ *learns the class* C *within the mistake bound model* if for any $c \in C$ its mistake bound on c is $M(c) = poly(n, r, size(c))$, and if its running time during each trial is also $poly(n, r, size(c))$, where r is the number of relevant attributes in c and $size(c)$ is the size in bits of c, given some standard representation. We gloss over some technicalities involved with defining $size(c)$. Precise definitions can be found in a book by Anthony and Biggs (1992). If furthermore, we have that $M(c) = poly(r, size(c)) \cdot polylog(n)$, then $\mathcal{A}$ is said to be *attribute efficient* w.r.t. C.

The mistake bound model is closely related to Angluin's exact learning model (Angluin, 1988) with equivalence queries only. Also, learnability in the mistake bound model implies learnability in the PAC model, defined by Valiant (1984).

2.2 Decision Lists

Decision lists are maybe the simplest model for hierarchial decision making, but despite their simplicity, they can be used for representing a wide range of classifiers. A decision list might be viewed as a hierarchy of experts. When a classification is needed, the first expert in the hierarchy is addressed. If this expert suggests a classification, then his opinion is taken to be the classification of the decision list. Otherwise, the second expert in the hierarchy is asked for his classification. If he chooses to abstain as well, the third expert is addressed, and so on.

Alternatively, programmers prefer presenting decision lists as sequences of if-then-else statements, intended for classifying an instance $\vec{x}$. For example:

```
if condition1(x) is true then output = output1(x)
else if condition2(x) is true then output = output2(x)
else if condition3(x) is true then output = output3(x)
.
.
.
else output = default_output (x)
```

Restricting ourselves to boolean functions, the most general form of decision list is an ordered list of pairs of boolean function, $\langle\langle t_1, o_1\rangle, \langle t_2, o_2\rangle, \ldots, \langle t_r, o_r\rangle\langle 1, o_{def}\rangle\rangle$. All functions have the same range $X_n = \{0,1\}^n$. Each pair $\langle t_i, o_i\rangle$ is called a *node* of the decision list, the function t_i being the *test* function, and the function o_i, the *output* function. The last node in a decision list is called the *default* node, and has a constant test function, that evaluates to 1. The evaluation of a decision list on an input $\vec{x} \in X_n$, is obtained by first finding the minimal i, such that $t_i(\vec{x}) = 1$, and then outputting $o_i(\vec{x})$. A schematic drawing of a simple decision list is shown in Figure 1(a).

A *k-decision list* (Rivest, 1987) is a decision list in which all the test functions are monomials with at most k literals, and the output functions are the constants 0 and 1. That is, $t_1, \ldots, t_r \in \{l_{i_1} \wedge l_{i_2} \wedge \cdots \wedge l_{i_j} : 1 \le j \le k \wedge l_{i_s} \in \{x_{i_s}, \bar{x}_{i_s}\}\}$ and $o_1, \ldots, o_r, o_{def} \in \{0,1\}$. Rivest (1987) showed that k-decision lists generalize the basic classes of k-DNF, k-CNF and depth k decision trees. Blum (1992) extended the last result for rank-k decision trees. Blum and Singh (1990) mentioned that k-decision lists, in which the output function changes k times from 1 to 0 and back, can express any function of k terms, that is, any function of the form $f(T_1, T_2, \ldots, T_k)$ where $T_1, \ldots, T_k$ are monomials. It was later shown that even the simple class of 1-decision lists coincides with fragments of important classes such as disguised Horn functions, read-once functions, threshold functions and nested differences of concepts (Eiter et al., 1998). We therefore observe that the class of k-decision lists is a highly expressive subset of the more general class of decision lists. This paper focuses on finding an efficient learning algorithm for the class of k-decision lists.

A *monotone decision list* is a decision list in which all test functions are monotone monomials (i.e., without negations). General properties of monotone decision lists, as well as their learnability in the query models, were studied by Guijarro et al. (2001). As the following reductions show, an algorithm for learning the class of monotone 1-decision lists (all test functions are single variables) with a $\langle 1, 0\rangle$ default node, can be transformed into an algorithm for learning k-decision lists. The proofs presented here are inspired by Littlestone (1988) and are considered "folklore".

Lemma 1 *For any constant k, if $\mathcal{A}$ learns the class of monotone 1-decision lists in the mistake bound model, then $\mathcal{A}$ can be used for learning the class of k-decision lists. Furthermore, if $\mathcal{A}$ is attribute efficient w.r.t. monotone 1-decision lists, then $\mathcal{A}$ can be used for attribute-efficient learning of k-decision lists, for any finite k.*

Proof Let $n_2 = \sum_{i=0}^{k} 2^i \binom{n_1}{i}$. Any instance $\vec{x} = \langle x_1, \ldots, x_{n_1}\rangle$, given in the learning process, is first transformed into the instance $T(\vec{x}) = \langle c_1(\vec{x}), c_2(\vec{x}), \ldots, c_{n_2}(\vec{x})\rangle$, where the $c_i(\vec{x})$ range over all monomials of at most k literals. The transformed instance is then given as input to $\mathcal{A}$, and the prediction of $\mathcal{A}$ is taken.

Notice that the transformation $T(\vec{x})$, maps the original instance space $\{0,1\}^{n_1}$ into $\{0,1\}^{n_2}$. It also induces a mapping from the class of k-decision lists over $\{0,1\}^{n_1}$ into the class of monotone 1-decision lists over $\{0,1\}^{n_2}$. Let $L_1 = \langle\langle t_1, o_1\rangle, \ldots, \langle t_m, o_m\rangle, \langle 1, o_{m+1}\rangle\rangle$ be a k-decision list over $\{0,1\}^{n_1}$ with r_1 relevant variables. Then there exist $y_{i_1}, ..., y_{i_m} \in \{0,1\}^{n_2}$ such that the decision list $L_2 = \langle\langle y_{i_1}, o_1\rangle, \ldots, \langle y_{i_m}, o_m\rangle, \langle 1, o_{m+1}\rangle\rangle$ outputs 1 on $T(\vec{x})$ iff L_1 outputs 1 on $\vec{x}$. It is therefore possible for $\mathcal{A}$ to learn L_1 over $\{0,1\}^{n_1}$, by learning L_2 over $\{0,1\}^{n_2}$, but its mistake bound should be analyzed w.r.t. n_1 and r_1.

One can show that $n_2 \le (2n_1)^k + 1$, and thus if $\mathcal{A}$'s mistake bound is polynomial in the total number of variables it sees n_2, it is also polynomial in n. Similarly, if $\mathcal{A}$'s mistake bound is poly-logarithmic in n_2, it also remains so over n_1 (recall that k is a constant).

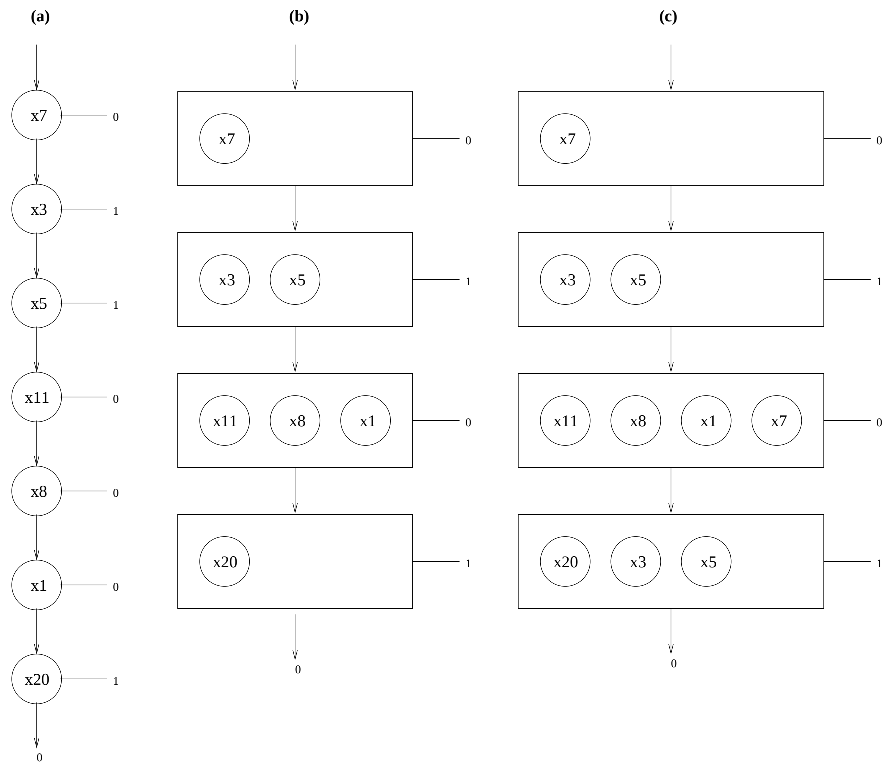

Figure 1: A decision list (a), its layered structure (b), and an isomorphic decision list (c).

Also, $\mathcal{A}$ learns a concept with m relevant variables. Using a similar bound we can show that $m \leq (2r_1)^k + 1$. Therefore, both learnability and attribute efficiency are preserved. ■

Lemma 2 *If $\mathcal{A}$ learns the class of monotone 1-decision lists with a $\langle 1, 0\rangle$ default node, then $\mathcal{A}$ can be used for learning monotone 1-decision lists. Attribute efficiency is also preserved.*

Proof Transform each instance $\vec{x}$, into $\langle x_1, \ldots, x_n, 1\rangle$ (an additional variable, x_{n+1} is always assigned 1). Applying this transformation, any decision list $\langle\langle x_{i_1}, o_1\rangle, \ldots, \langle x_{i_r}, o_r\rangle, \langle 1, 1\rangle\rangle$, is now isomorphic to the decision list $\langle\langle x_{i_1}, o_1\rangle, \ldots, \langle x_{i_r}, o_r\rangle, \langle x_{n+1}, 1\rangle, \langle 1, 0\rangle\rangle$, which has a $\langle 1, 0\rangle$ default node, and can be learned by $\mathcal{A}$. Adding a single variable and a single node keeps both learnability and attribute efficiency properties. ■

Following the above lemmas, we will further focus on monotone 1-decision lists with a $\langle 1, 0\rangle$ default node, and the term "decision list" will be used, from now on, to refer to this restricted class. We now introduce a few properties of decision lists.

First, we would like to define what is meant by $size(c)$ when the boolean function c is a decision list. We therefore have to define an adequate bit representation for decision lists. We shall use the rather natural representation, in which we write the list of nodes sequentially. For each node we have to write both test function and output function. Recall that the test function is simply a single variable, so $\lceil \log n \rceil$ bits would suffice to write the index of the variable. The output function is either 1 or 0, contributing additional bit for each node. Summing everything up, we can write any decision list c, using $\lceil \log n \rceil + 1$ bits for each of the r nodes, having $size(c) = r(\lceil \log n \rceil + 1)$. Note that there is no need to write the default node as both test and output functions are fixed.

Next we consider an alternative way to look at decision lists.

Definition 3 *A **layer** in a decision list (sometimes called a **level**) is a sub-list of contiguous nodes all having the same output function. Any adjacent nodes to the nodes of the layer, must not have the same output function. The default node does not belong to any layer.*

The last definition implies that any decision list can be viewed as an ordered list of layers $\langle\langle l_1, o_1\rangle, \langle l_2, o_2\rangle, \ldots, \langle l_D, o_D\rangle, \langle 1, 0\rangle\rangle$, where each layer consists of an ordered list of variables, and the o_is alternate between 1 and 0. We can ignore the order of the variables in each layer, as changing the order within a single layer, results in an isomorphic decision list. This in turn implies that we can treat each layer as a single node, whose test function is a disjunction of all the variables in that layer. It turns out, as we later observe, that the number of layers in a decision list (sometimes referred to as the number of alternations), is another indication of its complexity, along with the number of nodes. Finally, we notice the following observation:

Observation 4 *Let L be a decision list. Let $\tilde{L}$ be a decision list that satisfies:*

- *$\tilde{L}$ has the same number of layers as L.*
- *Every variable x_i that first appears in L at level j (i.e., j is the minimal index s.t. x_i appears in layer j), also first appears in $\tilde{L}$ at level j, and vice versa.*

Then $\tilde{L}$ is isomorphic to L.

The above observation implies, that adding extra variables already in the list to bottom[1] layers, does not change the boolean function, computed by the decision list. This is due to the hierarchial way in which the output is determined. The upper most layer, which contains a variable with an assignment 1, determines the output. This variable can appear again in lower layers (even with different output), but only its upper most appearance has an effect. Figure 1 shows an example of a decision list, along with its layered structure and an isomorphic decision list with extra variables.

2.3 The Winnow Algorithm

The Winnow algorithm, first introduced by Littlestone (1988), is a powerful algorithm for learning a variety of concept classes, and considered to be the canonical example for attribute efficient learning. One application of Winnow is for learning monotone disjunctions in the mistake bound model. It achieves a mistake bound of $O(r \cdot \log n)$ for a disjunction of r out of n possible attributes.

Figure 2 presents a simple version of Winnow that maintains a vector $\vec{w} \in \mathbb{R}^n$ of weights, one for each of the n variables. A large weight for a specific variable, represents the algorithm's "belief" that the variable belongs to the disjunction. A fixed parameter $\alpha > 1$, controls the rate at which the weights change.

1. Throughout this paper layers with high indices are referred to as *low* or *bottom* layers. This notation is rather common in the literature, though some might find it a bit confusing.

Winnow (α)

1. Initialize each of the weights $w_1, \ldots, w_n$ to 1.
2. Given an instance, $\vec{x}$, calculate $y := \vec{w} \cdot \vec{x}$.
3. Output $h(\vec{x}) = 1$ if $y \geq n$, and output $h(\vec{x}) = 0$ otherwise.
4. Receive the correct classification, $c(\vec{x})$.
5. If the algorithm made a mistake:
   (a) If the mistake is a false negative ($h(\vec{x}) = 0$, while $c(\vec{x}) = 1$):
   For each $1 \leq j \leq n$, set $w_j := \alpha^{x_j} w_j$. We call this step a *promotion*.
   (b) If the mistake is a false positive ($h(\vec{x}) = 1$, while $c(\vec{x}) = 0$):
   For each $1 \leq j \leq n$, set $w_j := \alpha^{-x_j} w_j$. We call this step a *demotion*.
6. Goto 2.

Figure 2: The Winnow algorithm.

Theorem 5 (Littlestone 1988) *Winnow (α) learns the class of monotone disjunctions in the mistake bound model, making at most $(\alpha+1)r \log_\alpha \alpha n + \frac{\alpha}{\alpha-1}$ mistakes, given that the target concept is a disjunction of r variables.*

Having this result, we will later use multiple instances of Winnow for learning the monotone disjunctions in each of the layers of a decision list. However, for this scheme to work, we must rely on another behaviour Winnow exhibits: its ability to tolerate a small number of misclassified instances.

Definition 6 *The number of **attribute errors** in the data, w.r.t. a monotone disjunction c (denoted by A_c), is defined as follows. For each example, labeled positive, but satisfies no variable in c, we add 1 to A_c. For each example, labeled negative, but satisfies k variables in c we add k to A_c.*

Theorem 7 (Littlestone 1989) *For any sequence of examples, and any monotone disjunction c, the mistake bound for Winnow is $O(r \log n + A_c)$.*

Theorem 7 can be used directly to analyze the mistake bound of our Multi-Layered Winnow in those simple cases where the number of layers in the target decision list is small. Our proof does not rely on this result, but rather provides a straightforward analysis for any number of layers.

3. Related Results

Since the problem of learning decision lists was introduced by Rivest (1987), a few learning algorithms were suggested, and were analyzed within the various learning models. However, none of the proposed algorithms achieve attribute efficiency (not even attribute efficiency

Rivest's Algorithm (D)

1. Initialize all weights $w_{i,j}$ to 1 ($i = 1 \dots D, j = 1 \dots n$).
2. Given an instance $\vec{x} = \langle x_1, \dots, x_n \rangle$, calculate $\vec{y} := \mathbf{W} \cdot \vec{x}$.
3. Let k^* be the minimal k s.t. $y_k > 0$. If there is no such k, set $k^* := D + 1$.
4. Output o_{k^*}, the output function of layer k^*.
5. Receive $c(\vec{x})$.
6. If the algorithm made a mistake:

   For each $x_j = 1$, set $w_{k^*,j} := 0$.
7. Goto 2.

Figure 3: Rivest's Algorithm.

in the PAC model, which should be easier), and it remains an open question whether such an algorithm exists. A recent hardness result hints that the task might be computationally hard (see below).

Nevertheless, some algorithms achieve a weaker notion of attribute efficiency by attaining a mistake bound that depends logarithmically on n, the total number of attributes, but exponentially on r, the number of relevant attributes, where the exponent of r is linear in D, the number of layers. Since the number of layers can be as large as r, such algorithms cannot be considered attribute efficient as defined. However, on those cases when $D << r << n$, they might become rather useful.

We present in this section a few important algorithms for learning decision lists. In Subsection 3.1 we detail the first published algorithm for learning decision lists, suggested by Rivest. This algorithm achieves a mistake bound of $O(D \cdot n)$. In Subsection 3.2 we discuss how Littlestone's Balanced Winnow algorithm, can be used for learning decision lists, achieving a mistake bound of $O(r^{2D} \log n)$. In comparison, the mistake bound of our new algorithm, Multi-Layered Winnow, is $O(r^D \log n)$ (Section 4). In Subsection 3.3, we discuss what might be done with a computationally unbounded learner and then briefly quote the main results of a paper by Servedio, where he proved a hardness result, concerning the learning of decision lists. Subsection 3.4 discusses a PAC learning algorithm by Dhagat and Hellerstein (1994), with a sample complexity of $O(\frac{1}{\epsilon}(\log \frac{1}{\delta} + r^D \log n \log^D(\frac{r}{\epsilon})))$. The algorithm is basically an Occam algorithm, fitting more into the world of offline algorithms, but its basic layered approach resembles ideas on which Multi-Layered Winnow is based.

3.1 Rivest's Algorithm

The first algorithm for learning decision lists was proposed by Rivest (1987). The algorithm was analyzed within the PAC model, and was later adapted to the Mistake Bound model (Helmbold et al., 1990). Figure 3 gives a modified version of the original algorithm that better suits the notion of layers.

This version assumes that the number of layers, D, is known in advance, and that a $\langle 1, 0\rangle$ default node ends the list. Small modifications to the algorithm allow for any kind of default node, and can eliminate the prior knowledge of D.

The algorithm uses a matrix $\mathbf{W} \in \{0,1\}^{D\times n}$, where $w_{i,j} = 1$, if x_j is believed to be in layer i. At first, all variables are assumed to be in all layers. Given an instance, the algorithm's prediction is the output of the first layer, that contains a variable with an assignment 1. After each mistake, the variables that caused the mistake are eliminated from the predicting layer. The output function of the i-th layer, o_i, is set to be $o_i = (D - i + 1) \mod 2$, so that the output functions alternate between 1 and 0, and the last layer has $o_D = 1$ ($o_{def} = o_{D+1} = 0$).

For the mistake bound analysis, notice that an adversary can force the algorithm to eliminate only one variable from one layer at a time. Since at first, all variables are assumed to appear in all layers, the algorithm can make $O(D \cdot n)$ mistakes, before eliminating all redundant variables.

Theorem 8 *Rivest's algorithm learns the class of monotone 1-decision lists with a $\langle 1, 0\rangle$ default node and D layers, making $O(D \cdot n)$ mistakes.*[2]

3.2 Balanced Winnow

Balanced Winnow was proposed by Littlestone (1989) as yet another algorithm for learning *linear threshold functions*. A linear threshold function is a function $f : [0,1]^n \to \{0,1\}$, for which there exist a vector of real coefficients $\vec{\mu} = \mu_1, \ldots, \mu_n$ and a threshold θ, such that

$$f(\vec{x}) = \begin{cases} 1, & \vec{x} \cdot \vec{\mu} \geq \theta; \\ 0, & \text{otherwise.} \end{cases}$$

The algorithm resembles the previously described Winnow algorithm, but it maintains **two** vectors of weights $\vec{w}^1, \vec{w}^0 \in \mathbb{R}^n$. Intuitively, a large value for w_j^1, indicates that whenever x_j will have an assignment 1, the output should tend to be 1. Similarly, a large value for w_j^0, indicates that a 1 assignment for x_j should divert the output towards 0. The algorithm takes two parameters, $\alpha > 1$ and $0 < \beta < 1$, that influence the rate at which the weights change. It is given in Figure 4.

Littlestone showed that Balanced Winnow cannot efficiently learn every linear threshold function. Instead it learns only functions, which exhibit a stronger type of linear separability. These functions must have a "δ-wide margin" which separates the positive sub-domain (the points for which $f(\vec{x}) = 1$) from the negative one. Formally, the coefficients vector of such functions should be defined using two vectors, $\vec{\mu}^+, \vec{\mu}^- \in {\mathbb{R}^+}^n$, and there must be a separation parameter $0 < \delta \leq \frac{1}{2}$ such that

$$\begin{aligned} &\sum_{i=1}^{n} \mu_i^+ + \sum_{i=1}^{n} \mu_i^- = 1; && \\ &\vec{\mu}^+ \cdot \vec{x} - \vec{\mu}^- \cdot \vec{x} \geq \delta && \text{whenever } f(\vec{x}) = 1; \\ \text{and } &\vec{\mu}^+ \cdot \vec{x} - \vec{\mu}^- \cdot \vec{x} \leq -\delta && \text{whenever } f(\vec{x}) = 0. \end{aligned}$$

Littlestone proved the following theorem (adopted version):

2. It is common to find in the literature an $O(r \cdot n)$ mistake bound, which is obviously less tight.

Balanced Winnow (α, β)

1. Initialize all weights in $\vec{w}^1$ and in $\vec{w}^0$ to 1.
2. Given an instance $\vec{x}$, calculate $y := \vec{w}^1 \cdot \vec{x} - \vec{w}^0 \cdot \vec{x}$
3. Output 1 if $y \geq 0$, and 0 otherwise.
4. Receive $c(\vec{x})$
5. If the algorithm made a mistake:
   (a) If $c(\vec{x}) = 1$:
   For each $1 \leq j \leq n$, set $w_j^1 := \alpha^{x_j} w_j^1$.
   For each $1 \leq j \leq n$, set $w_j^0 := \beta^{x_j} w_j^0$.
   (b) If $c(\vec{x}) = 0$:
   For each $1 \leq j \leq n$, set $w_j^1 := \beta^{x_j} w_j^1$.
   For each $1 \leq j \leq n$, set $w_j^0 := \alpha^{x_j} w_j^0$.
6. Goto 2.

Figure 4: Balanced Winnow

Theorem 9 (Littlestone) *Balanced Winnow* $(1+\frac{\delta}{2}, 1-\frac{\delta}{2})$ *learns the class of linear threshold functions, that satisfy the above* δ*-margin separation condition, making at most* $\frac{6 \log 2n}{\delta^2}$ *mistakes.*[3]

It was later mentioned (Blum and Singh, 1990; Dhagat and Hellerstein, 1994; Valiant, 1999), that any decision list can be written as a linear threshold function, but in order to satisfy the above separation rule, it must be "augmented" in the following sense. One extra variable, x_{n+1}, must be added, and it is given a constant assignment 1 in all instances. We can now write, for example, the decision list in Figure 1, as:

$$f(\vec{x}) = \begin{cases} 1, & -\frac{3}{59}x_1 + \frac{11}{59}x_3 + \frac{11}{59}x_5 - \frac{25}{59}x_7 - \frac{3}{59}x_8 - \frac{3}{59}x_{11} + \frac{2}{59}x_{20} - \frac{1}{59}x_{n+1} \geq 0; \\ 0, & \text{otherwise.} \end{cases}$$

It can be easily verified that $f(\vec{x})$ satisfies the δ-margin separation with $\delta = \frac{1}{59}$. Thus, Balanced Winnow can be used for learning decision lists.

In general, for the δ-margin property, the coefficient of a variable in layer i, must be $O(r)$ times larger than the coefficient of a variable in layer $i+1$, so there must be an $O(r^D)$ gap between the coefficients of the variables in the first layer and those of the last. It follows that the separation parameter for decision lists is $\delta = O(1/r^D)$. The resulting mistake bound is given by the following theorem.

Theorem 10 *Balanced Winnow learns the class of monotone 1-decision lists with D layers, making* $O(r^{2D} \log n)$ *mistakes.*

3. Learning still occurs with different parameters given to the algorithm. Littlestone (1989) describes extended results.

Valiant (1999) used a careful analysis to show that the above bound can be reduced to $O\left((\frac{2r}{D})^{2D}\log n\right)$. However, the $\left(\frac{2}{D}\right)^{2D}$ multiplicative constant does not change the asymptotic behaviour of Balanced Winnow (recall that we consider the case where D is a constant).

3.3 Computationally Unbounded Learning and Hardness Results

It might well be that achieving a $poly(r, size(c)) \cdot polylog(n)$ mistake bound in learning decision lists, requires computational resources which are more than polynomial (in r, in $size(c)$ or in n). In fact a *computationally unbounded* learner, can learn decision lists making $O(r \log n)$ mistakes, using the "Halving Algorithm". The Halving Algorithm (Littlestone, 1988) is a general algorithm for learning any finite target class C, within the mistake bound model. At any stage of the game, the algorithm keeps the set $CONS$, which is the set of all target functions in C that are consistent with all instances seen so far (initially this set equals C). Whenever a prediction is needed, the algorithm takes the majority vote of all functions in $CONS$. Thus, whenever a mistake is made, at least half of the functions in $CONS$ are found wrong and are taken out from $CONS$. The resulting mistake bound is therefore $\lceil \log |C| \rceil$.

In the case where C consists of decision lists over n variables and with r nodes, the cardinality of C is $O((2n)^r)$ (choose r variables out of possible n into the r nodes, then choose either 1 or 0 output function for each node). The mistake bound of the Halving algorithm for that C is therefore $O(r \log n)$. However, the time complexity of calculating the output of $O((2n)^r)$ decision lists is exponential in r.

A recent study by Servadio (2000) showed a few hardness results, relating to the polynomial time learnability of decision lists. In particular, he proved the following.

Definition 11 *A function $f : \{0,1\}^* \to \{0,1\}^*$ is said to be* length preserving $q(n)$-one-way permutation *if the following conditions hold:*

- *$|f(x)| = |x|$ for all $x \in \{0,1\}^*$.*
- *There is a deterministic poly-time algorithm which computes $f(x)$.*
- *For any probabilistic poly$(q(n))$-time algorithm $\mathcal{A}$, for any polynomial Q, and for any sufficiently large n, we have that $\Pr_{x \in \mathcal{U}_n}[\mathcal{A}(f(x)) = x] < \frac{1}{Q(q(n))}$, where $x \in \mathcal{U}_n$ means that x is chosen from the set $\{0,1\}^n$ according to the uniform distribution.*

Theorem 12 (Servadio) *For any integer $c \geq 2$, let $\log(c, n)$ denote $\overbrace{\log\log\cdots\log}^{c} n$. Let $q(c, n) = n^{\log(c,n)}$. If there is some integer $c \geq 2$, such that length-preserving $q(c, n)$-one-way permutation exists, then there exists a concept class C of $O(\log(c, n))$-decision lists which has the following properties in the mistake bound model:*

- *A computationally unbounded learner can learn C with at most 1 mistake;*
- *C can be learned in polynomial time;*
- *C cannot be learned in polynomial time by an attribute efficient algorithm.*

This result may indicate, that attribute efficient learning of decision lists, is computationally hard, however it certainly does not prove it. Notice that the existence of length-preserving $q(c, n)$-one-way permutation is not a standard cryptographic assumption (see discussion by Servadio, 2000). Also, the concept class C, is of $O(\log(c, n))$-decision lists, so the number of literals in each node, increases with n (though it does so very slowly). It is therefore still possible that the class of 1-decision lists (where 1 does not depend on n) has an attribute efficient learning algorithm.

3.4 PAC-Based Related Results

The PAC model (Valiant, 1984) is widely used for analyzing the performance of offline learning algorithms. In the PAC model, the goal of the learner is to produce an approximately correct hypothesis (with high probability), after seeing a random sample of classified instances, called the *training set*. More formally, it is hypothesized that a fixed distribution over the instance space is used to draw a number of classified instances (classified using a target function $c \in C$). A learning algorithm $\mathcal{A}$ can then observe the training set and should generate a hypothesis, whose *error* is defined to be the probability of drawing an instance (under the same distribution) on which the hypothesis and the target function disagree.

We say that algorithm $\mathcal{A}$ *PAC learns C by H*, if for any distribution, any target function $c \in C$ and any error parameters ϵ and δ, there exist polynomials $s(n, \frac{1}{\epsilon}, \frac{1}{\delta}, size(c))$ and $t(n, \frac{1}{\epsilon}, \frac{1}{\delta}, size(c))$ such that given a training set of size $s(n, \frac{1}{\epsilon}, \frac{1}{\delta}, size(c))$, $\mathcal{A}$ runs at time at most $t(n, \frac{1}{\epsilon}, \frac{1}{\delta}, size(c))$, and outputs a hypothesis $h \in H$ that with probability $1 - \delta$ has error at most ϵ. Algorithms that need a relatively small training set for PAC learning of certain classes are said to have a low *sample complexity*.

We say that algorithm $\mathcal{A}$ is *consistent* with c if after seeing any training set, classified by c, it produces a hypothesis h, such that $h(x) = c(x)$ for any instance x in the training set. $\mathcal{A}$ is an *Occam algorithm* for C if for any $c \in C$ and for any training set of size m, classified by c, $\mathcal{A}$ is consistent and produces a hypothesis of size at most $(size(c))^{\beta} m^{\alpha}$, for constants $\alpha < 1$ and $\beta \geq 1$. Blumer et al. (1987) showed that any Occam algorithm for C, producing hypotheses from H, PAC learns C by H.

Dhagat and Hellerstein (1994) proposed an Occam algorithm for PAC learning (non-monotone) 1-decision lists with D layers, for which the following theorem holds.

Theorem 13 (Dhagat and Hellerstein) *Dhagat and Hellerstein's algorithm PAC learns the class of 1-decision lists with D layers and r relevant variables by the class of 1-decision lists with D layers and $O(r^D \log^D(\frac{r}{\epsilon}))$ relevant variables. The sample complexity of the algorithm is bounded by a polynomial $s = O\left(\frac{1}{\epsilon}(\log \frac{1}{\delta} + r^D \log n \log^D(\frac{r}{\epsilon}))\right)$, and it runs in time bounded by a polynomial $t = O(sn^2)$.*

Their algorithm uses as a subroutine a well known greedy approximation algorithm for the *set cover problem*. The set cover problem takes as input a collection S of subsets of a finite set U, and asks for a subcollection $S' \subseteq S$, such that the union of subsets in S' is U (S' is a *cover* of U), and S' contains the smallest possible number of subsets. The set cover problem is NP-complete. The greedy approximation algorithm constructs a cover of size $z(\ln |U| + 1)$, where z is the size of the smallest cover. Generally speaking, it is used to build a small explanation for a phenomenon, from components that describe parts of the phenomenon.

Multi-Layered Winnow (D, α)

1. Initialize all weights $w_{i,j}$ to 1.
2. Given an instance $\vec{x} = \langle x_1, \ldots, x_n \rangle$, calculate $\vec{y} := \mathbf{W} \cdot \vec{x}$.
3. Let i^* be the minimal i s.t. $y_i \geq n$. If there is no such i, set $i^* := D + 1$.
4. Output o_{i^*}, the output function of layer i^*. We say that layer i^* *fired*.
5. Receive $c(\vec{x})$.
6. If the algorithm made a mistake:
   (a) If $i^* \leq D$:
   For each $1 \leq j \leq n$, set $w_{i^*,j} := \alpha^{-x_j} w_{i^*,j}$ (*demote* layer i^*).
   (b) If $i^* > 1$:
   For each $1 \leq j \leq n$, set $w_{i^*-1,j} := \alpha^{x_j} w_{i^*-1,j}$ (*promote* layer $i^* - 1$).
7. Goto 2.

Figure 5: Multi Layered Winnow.

The basic approach in Dhagat and Hellerstein's algorithm is finding a set of *candidates* for each layer, starting with layer 1 and moving downwards. A specific literal l is considered a candidate for layer i if all classified instances in the training set, in which l is satisfied but the classification differs from the output of layer i, can be explained by a "small" set of candidates from higher layers with the right output. Greedy set cover is used for finding such a "small" set, where small is defined to be $r(\ln |U| + 1)$, with U being the set of contradicting instances. The candidates for layer 1 are those for which no contradicting instances exist. After setting the sets of candidates, the hypothesis is built by carefully putting literals from these sets into the layers of the decision list. Greedy set cover is used in this stage too.

A major advantage of this algorithm is that it produces a hypothesis which is a decision list, depending on a relatively small number of variables. However, being an Occam algorithm, it carries an offline flavor as it must examine all data before giving a conclusion. It is unclear whether it can be transformed into an online algorithm, and whether it can be analyzed within the mistake bound model (since its hypothesis is approximately correct, there is no guarantee it will ever stop making mistakes).

4. Multi Layered Winnow

We now present a novel algorithm for learning a decision list with a $\langle 1, 0 \rangle$ default node and D layers. The algorithm attempts to learn an appropriate disjunction in each layer separately, using the basic Winnow algorithm (Section 2.3). A matrix $\mathbf{W} \in \mathbb{R}^{D \times n}$ of weights is used, in which the i-th row represents the weight vector of the Winnow instance in the i-th layer.

As in Rivest's algorithm $o_i = (D - i + 1) \mod 2$. A fixed parameter $\alpha > 1$ controls the learning rate. The algorithm is shown in Figure 5.

Notice that Multi-Layered Winnow $(1, \alpha)$ is identical to the Winnow algorithm, shown in Subsection 2.3 (a decision list with one layer is simply a monotone disjunction). Therefore, one can view Multi-Layered Winnow as a strict generalization of Winnow.

After each mistake, Multi-Layered Winnow updates the weight vectors of at most **two** layers: layer i^* and layer $i^* - 1$. The output of layer i^* is the wrong output for $\vec{x}$, and the output of layer $i^* - 1$ is the right one. The reason for the two updates is that any mistake can occur due to two reasons:

- Layer i^* was too strong, forcing its output, while a lower layer could classify the instance correctly.
- Layer $i^* - 1$ was too weak. It should have "captured" the instance before layer i^*, but failed to do so.

Since there is no obvious way to distinguish between the two cases, we try to correct both errors, demoting layer i^*, and promoting layer $i^* - 1$. However, one of the updates might be wrong, providing a specific Winnow instance with a misclassification. We therefore have to prove that after each mistake, at least one of the updates is correct, and that a correct update can compensate for a false update.

4.1 A Mistake Bound Analysis

Definitions 14–17 refer to the target decision list, which the algorithm attempts to learn. Definition 18 defines for each update step whether it moved the algorithm's hypothesis closer to the target function or not.

Definition 14 *The **minimal layer** of a variable x_j (denoted $l(x_j)$) is the index of the highest layer in which the variable appears. If no such layer exists, $l(x_j) = D + 1$.*

Example 1 *In the decision list, shown in Figure 1, the minimal layer of x_5 is 2 ($l(x_5) = 2$). The minimal layer of x_{16} is 5.*

Definition 15 *For any layer i, the **primary variable set** (denoted R_i) is defined to be $R_i \triangleq \{x_j : l(x_j) = i\}$.*

Example 2 *In the decision list, shown in Figure 1, the primary variable set of layer 3 is $R_3 = \{x_{11}, x_8, x_1\}$.*

Definition 16 *For any layer i, the **relevance variable set** (denoted $\tilde{R}_i$) is defined to be*

$$\tilde{R}_i \triangleq \bigcup_{\substack{o_j = o_i \\ j \le i}} R_j.$$

Example 3 *In the decision list, shown in Figure 1, the relevance variable set of layer 3 is $\tilde{R}_3 = \{x_{11}, x_8, x_1, x_7\}$. Notice that for layer 1, $\tilde{R}_1 = R_1 = \{x_7\}$ (there are no layers with a smaller index).*

Note: From now on we will ignore zero instances, i.e., instances in which $\vec{x} = \vec{0}$. Such instances do not change the mistake bound, as the algorithm always predicts 0 for them, which is the correct classification in any decision list with a $\langle 1, 0\rangle$ default node.

Definition 17 *Given an instance $\vec{x}$, we define the* ***dominant layer*** *(denoted $l_{dom}(\vec{x})$), as*

$$l_{dom}(\vec{x}) \triangleq \min\{l(x_j) : x_j = 1\}.$$

Following the definition of a decision list, this is the layer, whose output determines $c(\vec{x})$. We also define the ***dominant variable*** *(denoted $x_{dom}(\vec{x})$) to be any variable of the (non empty) set $\{x_j : l(x_j) = l_{dom}(\vec{x}) \wedge x_j = 1\}$.*

Example 4 *In the decision list, shown in Figure 1, the dominant layer for the instance $\vec{x} = 10000001000\ldots0$ is layer 3 ($l_{dom}(\vec{x}) = 3$). The dominant variable can be either x_1 or x_8 (but not x_{11} as it gets a 0 assignment).*

Definition 18 *When layer i^* is demoted, we say that it is* ***falsely demoted*** *if there is a variable $x_j \in \tilde{R}_{i^*}$, for which $w_{i^*,j}$ is decreased. Otherwise we say that it is* ***correctly demoted****. Similarly, when layer $i^* - 1$ is promoted, we say that it is* ***falsely promoted*** *if there is no variable $x_j \in \tilde{R}_{i^*-1}$, for which $w_{i^*-1,j}$ is increased. Otherwise we say that it is* ***correctly promoted****.*

Lemma 19 *On each update, if no layer was demoted, then layer D was correctly promoted.*

Proof By inspection of the algorithm, if no layer was demoted it must be that $i^* = D + 1$, and layer D was promoted. Since the correct output for $\vec{x}$ was 1, layer $l_{dom}(\vec{x})$ must have output 1. It follows that $x_{dom}(\vec{x}) \in \tilde{R}_D$. Also, $x_{dom}(\vec{x})$ must have an assignment 1 in $\vec{x}$, so its weight in layer D was increased. Layer D was therefore correctly promoted. ■

Lemma 20 *On each update, if no layer was promoted, then layer 1 was correctly demoted.*

Proof By inspection of the algorithm, if no layer was promoted, we clearly had $i^* = 1$ (layer 1 fired), and thus layer 1 was demoted. If layer 1 was falsely demoted, we must had a variable $x_j \in \tilde{R}_1 = R_1$, for which the weight in layer 1 was decreased. This means x_j had an assignment 1 in $\vec{x}$. However giving an assignment 1 to $x_j \in R_1$ forces the **target** decision list to output the output function of layer 1. Thus, no mistake should have happened. ■

Lemma 21 *If the algorithm falsely demoted layer i^* after seeing $\vec{x}$, then $l_{dom}(\vec{x}) < i^*$.*

Proof Clearly, $l_{dom}(\vec{x}) \neq i^*$, as the two layers must have different outputs. By definition, any variable x_j, for which $l(x_j) < l_{dom}(\vec{x})$ must had a 0 assignment in $\vec{x}$. Therefore, if $l_{dom}(\vec{x}) > i^*$, then any variable $x_j \in \tilde{R}_{i^*}$ (for which $l(x_j) \leq i^*$) must had a 0 assignment. It follows that there was no variable $x_j \in \tilde{R}_{i^*}$, for which the weight $w_{i^*,j}$ was decreased, and that layer i^* was correctly demoted. ■

Lemma 22 *After each mistake, the algorithm either correctly promotes a layer, or correctly demotes a layer (or both).*

Proof We show that the algorithm cannot falsely demote layer i^* and falsely promote layer $i^* - 1$. Suppose the algorithm falsely demoted layer i^*. From Lemma 21 we know that $l_{dom}(\vec{x}) \leq i^* - 1$. Also, layer $l_{dom}(\vec{x})$ and layer $i^* - 1$ have the same output, so by definition $x_{dom}(\vec{x}) \in \tilde{R}_{i^*-1}$. Since $x_{dom}(\vec{x})$ had an assignment 1, its weight in layer $i^* - 1$ was increased. Therefore, layer $i^* - 1$ was correctly promoted. ■

The above lemmas imply that we can consider only 5 possible updates:

Case 1: Layer 1 is correctly demoted, while no other layer is promoted.

Case 2: Layer i^* is correctly demoted, and layer $i^* - 1$ is correctly promoted.

Case 3: Layer i^* is falsely demoted, and layer $i^* - 1$ is correctly promoted.

Case 4: Layer i^* is correctly demoted, and layer $i^* - 1$ is falsely promoted.

Case 5: Layer D is correctly promoted, while no other layer is demoted.

The above 5 cases form a partition of all possible updates. Since every mistake is followed by an update, a bound on the number of updates, entails a mistake bound for the algorithm. Denoting by $M(i)$ a bound on the number of updates, that fall into Case i, $M(1)+M(2)+M(3)+M(4)+M(5)$ is a mistake bound for the algorithm. Before bounding each of the $M(i)$s, we need the following lemma (inspired by Littlestone, 1988).

Lemma 23 *Let $w_{i,j}^{(t)}$ denote the weight $w_{i,j}$ after the t-th trial. Then $\forall i, j, t, w_{i,j}^{(t)} < \alpha n$.*

Proof Initially, all weights $w_{i,j}^{(0)}$ are set to 1. Since $\alpha > 1$, we have that $w_{i,j}^{(0)} < \alpha n$. The value of $w_{i,j}^{(t)}$ was increased in the t-th trial only if $\sum_{j=1}^{n} w_{i,j}^{(t-1)} x_j < n$ and $x_j = 1$. These conditions can occur only if $w_{i,j}^{(t-1)} < n$. Thus, $w_{i,j}^{(t)} < \alpha n$. ■

Theorem 24 $M(2) + M(3) + M(5) < 2r^D(\log_\alpha n + 1)$.

Proof Consider the following potential function:

$$\Phi(t) = \sum_{i=1}^{D} r^{D-i} \sum_{x_j \in \tilde{R}_i} \left(\log_\alpha \alpha n - \log_\alpha w_{i,j}^{(t)} \right).$$

Informally, $\Phi(t)$ measures in each layer how far are the weights of the relevant variables from αn (weights of relevant variables should get values as high as possible). Through the coefficient r^{D-i}, the function gives a higher importance to higher layers than to lower layers.

First, we notice that $\forall t : \Phi(t) > 0$. This is immediate from Lemma 23. Also, since all weights are initialized to 1 we get:

$$\Phi(0) = \log_\alpha \alpha n \sum_{i=1}^{D} r^{D-i} \sum_{x_j \in \tilde{R}_i} 1 = \log_\alpha \alpha n \sum_{i=1}^{D} r^{D-i} |\tilde{R}_i|.$$

An upper bound for $\Phi(0)$ is achieved by pessimistically assuming that all r relevant variables appear in all $\tilde{R}_i$s. Thus we have:

$$\Phi(0) \leq \log_\alpha \alpha n \sum_{i=1}^{D} r^{D-i} r = \log_\alpha \alpha n \sum_{i=1}^{D} r^i < 2r^D(\log_\alpha n + 1).$$

We now notice that updates of Case 1 and of Case 4 do not change the potential function (i.e., after an update of type 1 or 4 at the t-th trial, $\Delta\Phi(t) = \Phi(t) - \Phi(t-1) = 0$). This is because when layer i^* is correctly demoted, the weight $w_{i^*,j}$ is unchanged for each variable $x_j \in \tilde{R}_{i^*}$. Similarly, when layer i^*-1 is falsely promoted, the weight $w_{i^*-1,j}$ is unchanged for any $x_j \in \tilde{R}_{i^*-1}$.

Next, we show that every update of Cases 2, 3 and 5 decreases $\Phi(t)$ by at least 1. Since the potential function is always positive and cannot increase, this proves the theorem.

Case 2: Since layer i^* is correctly demoted, the weight $w_{i^*,j}$ is unchanged for any variable $x_j \in \tilde{R}_{i^*}$. Therefore, the demotion step does not change the potential function. Since layer i^*-1 is correctly promoted, there is at least one variable $x_j \in \tilde{R}_{i^*-1}$, that has an assignment 1. Thus,

$$\begin{aligned}
\Delta\Phi(t) &= \Phi(t) - \Phi(t-1) = \\
&= \sum_{i=1}^{D} r^{D-i} \sum_{x_j \in \tilde{R}_i} \left(\log_\alpha \alpha n - \log_\alpha w_{i,j}^{(t)}\right) - \sum_{i=1}^{D} r^{D-i} \sum_{x_j \in \tilde{R}_i} \left(\log_\alpha \alpha n - \log_\alpha w_{i,j}^{(t-1)}\right) = \\
&= r^{D-(i^*-1)} \sum_{x_j \in \tilde{R}_{i^*-1}} \left(-\log_\alpha w_{i^*-1,j}^{(t)}\right) - r^{D-(i^*-1)} \sum_{x_j \in \tilde{R}_{i^*-1}} \left(-\log_\alpha w_{i^*-1,j}^{(t-1)}\right) = \\
&= r^{D-(i^*-1)} \sum_{\substack{x_j \in \tilde{R}_{i^*-1} \\ x_j=1}} \left(\log_\alpha w_{i^*-1,j}^{(t-1)} - \log_\alpha w_{i^*-1,j}^{(t)}\right) = \\
&= r^{D-(i^*-1)} \sum_{\substack{x_j \in \tilde{R}_{i^*-1} \\ x_j=1}} \left(\log_\alpha w_{i^*-1,j}^{(t-1)} - \log_\alpha \alpha w_{i^*-1,j}^{(t-1)}\right) = \\
&= r^{D-(i^*-1)} \sum_{\substack{x_j \in \tilde{R}_{i^*-1} \\ x_j=1}} \log_\alpha \tfrac{1}{\alpha} = -r^{D-(i^*-1)} \sum_{\substack{x_j \in \tilde{R}_{i^*-1} \\ x_j=1}} 1 \leq -r^{D-(i^*-1)} \leq -1.
\end{aligned}$$

Case 3: We follow the equalities of Case 2, to state that the correct promotion of layer i^*-1 has the following contribution to $\Delta\Phi(t)$:

$$\Delta\Phi_{\text{prom}}(t) = -r^{D-(i^*-1)} \sum_{\substack{x_j \in \tilde{R}_{i^*-1} \\ x_j=1}} 1$$

and, similarly, the false demotion of layer i^* (in which weights are **divided** by α) contributes the following:

$$\Delta\Phi_{\text{dem}}(t) = r^{D-i^*} \sum_{\substack{x_j \in \tilde{R}_{i^*} \\ x_j=1}} 1.$$

Clearly $x_{dom}(\vec{x}^{(t)}) \notin \tilde{R}_{i^*}$, so we can give an upper bound for $\Delta\Phi_{\text{dem}}(t)$:

$$\Delta\Phi_{\text{dem}}(t) = r^{D-i^*} \sum_{\substack{x_j \in \tilde{R}_{i^*} \\ x_j=1}} 1 \leq (r-1)r^{D-i^*}.$$

On the other hand, since layer i^*-1 is correctly promoted, there is at least one variable $x_m \in \tilde{R}_{i^*-1}$, that has an assignment 1, and for which the weight $w_{i^*-1,m}$ is doubled. Therefore,

$$\Delta\Phi_{\text{prom}}(t) = -r^{D-(i^*-1)} \sum_{\substack{x_j \in \tilde{R}_{i^*-1} \\ x_j=1}} 1 \leq -r^{D-(i^*-1)}$$

and we can conclude that

$$\Delta\Phi(t) = \Delta\Phi_{\text{dem}}(t) + \Delta\Phi_{\text{prom}}(t) \leq (r-1)r^{D-i^*} - r^{D-(i^*-1)} = -r^{D-i^*} \leq -1.$$

Case 5: Same as Case 2, without any layer being demoted and with $i^* = D+1$. ■

Notice that only Case 3 updates make use of the exponential coefficient r^{D-i}, which contributes the r^D factor to the mistake bound. Thus, Case 3 updates are the real "bottleneck" of our algorithm.

Theorem 25 $M(1) + M(2) + M(3) + M(4) < \frac{\alpha^{D+1}-\alpha}{\alpha-1}\left[\frac{\alpha}{(\alpha-1)^2} + M(5)\right]$.

Proof Consider the following potential function:

$$\Psi(t) = \sum_{i=1}^{D} \frac{\alpha^{i+1}-\alpha}{(\alpha-1)^2} \sum_{j=1}^{n} \frac{w_{i,j}^{(t)}}{n}.$$

Informally, $\Psi(t)$ measures how much "noise" exists in our system (i.e., how high are the weights, including weights of irrelevant attributes). The function is more sensitive to noise in lower layers, than to noise in upper layers.

First, we notice that $\forall t : \Psi(t) > 0$, as the weights are always positive. Next, we give an upper bound for $\Psi(0)$:

$$\Psi(0) = \sum_{i=1}^{D} \frac{\alpha^{i+1}-\alpha}{(\alpha-1)^2} \sum_{j=1}^{n} \frac{1}{n} = \sum_{i=1}^{D} \frac{\alpha^{i+1}-\alpha}{(\alpha-1)^2} < \frac{\alpha}{(\alpha-1)^2} \sum_{i=1}^{D} \alpha^i = \frac{\alpha}{(\alpha-1)^2} \cdot \frac{\alpha^{D+1}-\alpha}{\alpha-1}.$$

We now show that every update, except of Case 5 updates, decreases the potential function by at least 1.

Case 1: Layer 1 is demoted following trial t. Therefore, layer 1 was the layer that fired, so before the demotion we had that $\sum_{x_j=1} w_{1,j}^{(t-1)} \geq n$. It follows that

$$\Delta\Psi(t) = \Psi(t) - \Psi(t-1) = \sum_{i=1}^{D} \frac{\alpha^{i+1}-\alpha}{(\alpha-1)^2} \sum_{j=1}^{n} \frac{w_{i,j}^{(t)}}{n} - \sum_{i=1}^{D} \frac{\alpha^{i+1}-\alpha}{(\alpha-1)^2} \sum_{j=1}^{n} \frac{w_{i,j}^{(t-1)}}{n} =$$

$$= \frac{\alpha^2-\alpha}{n(\alpha-1)^2} \sum_{j=1}^{n} w_{1,j}^{(t)} - \frac{\alpha^2-\alpha}{n(\alpha-1)^2} \sum_{j=1}^{n} w_{1,j}^{(t-1)} = \frac{\alpha}{n(\alpha-1)} \sum_{x_j=1} \left(\tfrac{1}{\alpha} w_{1,j}^{(t-1)} - w_{1,j}^{(t-1)} \right) =$$

$$= \frac{\alpha}{n(\alpha-1)} \frac{1-\alpha}{\alpha} \sum_{x_j=1} w_{1,j}^{(t-1)} = -\frac{1}{n} \sum_{x_j=1} w_{1,j}^{(t-1)} \leq -\frac{1}{n} n = -1.$$

Cases 2, 3 and 4: As in case 1, since layer i^* fired, we have that $\sum_{x_j=1} w_{i^*,j}^{(t-1)} \geq n$. Similarly, since layer i^*-1 did not fire, it must be that $\sum_{x_j=1} w_{i^*-1,j}^{(t-1)} < n$. Therefore,

$$\Delta\Psi(t) = \Psi(t) - \Psi(t-1) = \sum_{i=1}^{D} \frac{\alpha^{i+1}-\alpha}{(\alpha-1)^2} \sum_{j=1}^{n} \frac{w_{i,j}^{(t)}}{n} - \sum_{i=1}^{D} \frac{\alpha^{i+1}-\alpha}{(\alpha-1)^2} \sum_{j=1}^{n} \frac{w_{i,j}^{(t-1)}}{n} =$$

$$= \frac{\alpha^{i^*}-\alpha}{n(\alpha-1)^2} \sum_{x_j=1} \left(\alpha w_{i^*-1,j}^{(t-1)} - w_{i^*-1,j}^{(t-1)} \right) + \frac{\alpha^{i^*+1}-\alpha}{n(\alpha-1)^2} \sum_{x_j=1} \left(\tfrac{1}{\alpha} w_{i^*,j}^{(t-1)} - w_{i^*,j}^{(t-1)} \right) =$$

$$= \frac{\alpha^{i^*}-\alpha}{n(\alpha-1)^2} \cdot (\alpha-1) \sum_{x_j=1} w_{i^*-1,j}^{(t-1)} + \frac{\alpha^{i^*+1}-\alpha}{n(\alpha-1)^2} \cdot \frac{1-\alpha}{\alpha} \sum_{x_j=1} w_{i^*,j}^{(t-1)} <$$

$$< \frac{\alpha^{i^*}-\alpha}{n(\alpha-1)} \cdot n - \frac{\alpha^{i^*}-1}{n(\alpha-1)} \cdot n = \frac{1-\alpha}{\alpha-1} = -1.$$

Case 5: Layer D was promoted, so it surely did not fire. We must have then that $\sum_{x_j=1} w_{D,j}^{(t-1)} < n$. If follows that:

$$\Delta\Psi(t) = \frac{\alpha^{D+1}-\alpha}{(\alpha-1)^2} \sum_{x_j=1} \frac{\alpha w_{D,j}^{(t-1)} - w_{D,j}^{(t-1)}}{n} =$$

$$= \frac{\alpha^{D+1}-\alpha}{(\alpha-1)^2} \cdot \frac{\alpha-1}{n} \sum_{x_j=1} w_{D,j}^{(t-1)} < \frac{\alpha^{D+1}-\alpha}{(\alpha-1)^2} \cdot \frac{\alpha-1}{n} n = \frac{\alpha^{D+1}-\alpha}{\alpha-1}.$$

It is now immediate that there cannot be more than $\frac{\alpha^{D+1}-\alpha}{\alpha-1}\left[\frac{\alpha}{(\alpha-1)^2} + M(5)\right]$ updates of Cases 1, 2, 3 and 4, as the potential function must remain positive. This completes the proof. ∎

Theorem 26 *Multi-Layered Winnow (D, α) learns the class of monotone 1-decision lists with a $\langle 1, 0 \rangle$ default node and D layers, making at most $(D+1)\alpha^D \left[2r^D \log_\alpha(\alpha n) + \frac{\alpha}{(\alpha-1)^2}\right]$ mistakes.*

Proof First notice that the running time of the algorithm is $O(Dn)$ per instance, and therefore polynomial in n, in r and in $size(c)$ (recall that $size(c) = O(r \log n)$). By adding the bounds from Theorem 24 and Theorem 25 we conclude that the mistake bound, M, satisfies

$$\begin{aligned}
M &< M(1) + 2M(2) + 2M(3) + M(4) + M(5) < \\
&< \frac{\alpha^{D+1} - \alpha}{\alpha - 1}\left[\frac{\alpha}{(\alpha-1)^2} + M(5)\right] + 2r^D \log_\alpha(\alpha n) < \\
&< \frac{\alpha^{D+1} - \alpha}{\alpha - 1}\left[\frac{\alpha}{(\alpha-1)^2} + 2r^D \log_\alpha(\alpha n)\right] + 2r^D \log_\alpha(\alpha n) = \\
&= 2\frac{\alpha^{D+1} - 1}{\alpha - 1} \cdot r^D \log_\alpha(\alpha n) + \frac{\alpha^{D+2} - \alpha^2}{(\alpha-1)^3} < \\
&< \sum_{i=0}^{D} \alpha^i \cdot \left[2r^D \log_\alpha(\alpha n) + \frac{\alpha}{(\alpha-1)^2}\right] < (D+1)\alpha^D \left[2r^D \log_\alpha(\alpha n) + \frac{\alpha}{(\alpha-1)^2}\right].
\end{aligned}$$

■

Substituting $\alpha = 2^{1/D}$ simplifies the bound to become $O(D^2 r^D \log n)$.

Using the reductions shown in Lemma 1 and in Lemma 2 we can extend the target class that Multi-Layered Winnow can learn, as given by the following corollary.

Corollary 27 *For any constant k, Multi-Layered Winnow (D, α) learns the class of k-decision lists with D layers, making $O(r^{kD} \log n^k) = O(kr^{kD} \log n)$ mistakes.*

The following theorem and corollary state that the mistake bound's exponential dependence on D is inherent to the algorithm.

Theorem 28 *For any decision list with D layers that satisfies $n \geq r\frac{\alpha}{\alpha-1}$, there exists a sequence of instances, on which Multi-Layered Winnow (D, α) makes at least $\prod_{i=1}^{D} |R_i|$ mistakes.*

Proof See Appendix A. ■

Corollary 29 *Consider a decision list with D layers of equal size, that is, $|R_i| = \frac{r}{D}$ for any $1 \leq i \leq D$. If such a decision list is defined over $n \geq r\frac{\alpha}{\alpha-1}$ attributes, then there exists a sequence of instances on which Multi-Layered Winnow (D, α) makes at least $(\frac{r}{D})^D$ mistakes.*

4.2 Learning with an Arbitrary (Unknown) Number of Layers

We now want to eliminate the requirement of knowing in advance the number D of layers. We therefore present a meta-algorithm, which uses Multi-Layered Winnow $(D,\ \alpha)$ as a black box. The algorithm simply increases the guess for D, as long as Multi-Layered Winnow $(D,\ \alpha)$ makes more mistakes than the worst case bound. Since the bound depends

Multi-Layered Winnow

for $r = 1$ to n

  for $d = 1$ to r

    for $D = 1$ to d

      Run Multi-Layered Winnow $(D, 2)$ until the number of mistakes it makes, exceeds $8(2r)^d \log(2n)$.

Figure 6: Multi Layered Winnow - works for an arbitrary number of layers with $\alpha = 2$

also on r, the number of relevant variables, our algorithm also guesses a value for r, and increases it as well.

To make things a bit simpler, we now analyze the case $\alpha = 2$ (the same idea works for any $\alpha > 1$). For this case, the mistake bound given by Theorem 26 can be upper-bounded by $8(2r)^D \log(2n)$. Consider the algorithm in Figure 6. To see why this algorithm works, and why each of the three loops is necessary, let c be the target decision list, and suppose c has $\hat{D}$ layers and $\hat{r}$ relevant variables.

Lemma 30 *In the outer most loop, r never exceeds $\hat{r}$.*

Proof Consider the run of Multi-Layered Winnow $(D, 2)$, in which $r = \hat{r}$ and $d = D = \hat{D}$. From Theorem 26 we know that a mistake bound for this run is $M = (2^{D+2} - 2)r^D \log 2n + 2^{D+2} - 4$. Therefore, the number of mistakes will never exceed $8(2r)^d \log 2n$, and this run will never end. ■

Lemma 31 *If the inner most loop is started with values for r and d that satisfy $8(2r)^d \log 2n > 8(2\hat{r})^{\hat{D}} \log 2n$, then $d > \hat{D}$, but the inner most loop will never continue beyond $D = \hat{D}$.*

Proof From Lemma 30 we know that $r \leq \hat{r}$. It follows that when the inner loop is started with values for r and d which satisfy the inequality, we must have that $d > \hat{D}$. As a result, the inner most loop might run Multi-Layered Winnow(D) with $D = \hat{D}$, but as in Lemma 30, the number of mistakes in such run will never exceed $8(2r)^d \log 2n$, and the loop will never continue beyond $D = \hat{D}$. ■

Lemma 32 *Any run of Multi-Layered Winnow $(D, 2)$ cannot make more than $8(2\hat{r})^{\hat{D}+1} \log 2n$ mistakes.*

Proof Consider a specific time, in which the inner most loop is started with specific values of r and d. If r and d satisfy $8(2r)^d \log 2n \leq 8(2\hat{r})^{\hat{D}} \log 2n$, then the result trivially follows. Otherwise, consider the first time in which

$$8(2r)^d \log 2n > 8(2\hat{r})^{\hat{D}} \log 2n.$$

Since this is the first run, we know that

$$8(2r)^{d-1}\log 2n \leq 8(2\hat{r})^{\hat{D}}\log 2n$$

(if $d = 1$ this is surely true). Multiplying by $2\hat{r}$ we get:

$$8(2\hat{r})(2r)^{d-1}\log 2n \leq 8(2\hat{r})^{\hat{D}+1}\log 2n,$$

and since $r \leq \hat{r}$ we conclude that

$$8(2r)^{d}\log 2n \leq 8(2\hat{r})^{\hat{D}+1}\log 2n.$$

Also, from Lemma 31 we know that $d > \hat{D}$, and that the inner most loop will never continue beyond $D = \hat{D}$. Thus, no run of Multi-Layered Winnow $(D, 2)$ with $r \leq \hat{r}$ makes more than $8(2\hat{r})^{\hat{D}+1}\log 2n$ mistakes. ■

Theorem 33 *Multi-Layered Winnow learns any decision list over n variables with $\hat{r}$ relevant variables and $\hat{D}$ layers, making at most $8(2\hat{r})^{\hat{D}+4}\log 2n$ mistakes.*

Proof Since the algorithm will never execute an iteration with $r > \hat{r}$ (Lemma 30), the number of Multi-Layered Winnow $(D, 2)$ runs is bounded by $\hat{r}^3$. Since no run of Multi-Layered Winnow $(D, 2)$ will make more than $8(2\hat{r})^{\hat{D}+1}\log 2n$ mistakes (Lemma 32), the total number of mistakes in all runs is bounded by $8(2\hat{r})^{\hat{D}+4}\log 2n$. ■

4.3 A Hybrid Algorithm — Multi-Layered Winnow + Rivest's

Looking closely at the mistake bound analysis of Multi-Layered Winnow (D, α), it appears that false demotions have a major contribution to the mistake bound of the algorithm. In fact, it is only due to false demotions (Case 3 updates), that the mistake bound has the expression r^D in it. Furthermore, the number of mistakes depends heavily on the number of relevant variables, for which the weights were decreased. If in all instances, only a single relevant variable is set (has an assignment 1), there would be no false demotions, and the mistake bound would become $O(rD\log n)$, instead of $O(r^D\log n)$.

Therefore, we can deduce that Multi-Layered Winnow makes many mistakes, if it encounters instances, in which many (relevant) variables are set. This behaviour is contrary to the behaviour exhibited by Rivest's algorithm. Since Rivest's algorithm eliminates all irrelevant variables, which are set, it makes very few mistakes if it encounters instances with many 1-assigned variables. On the other hand, its weakness is when all instances have a small number of variables which are set. This opposite behaviour of the two algorithms, suggests to combine them together. This way, each algorithm can compensate for the weaknesses of the other.

Figure 7 shows hybrid Multi-Layered Winnow, into which Rivest's algorithm was incorporated. This hybrid runs much like Multi-Layered Winnow, but when a mistake occurs it adds to the demotion and promotion steps, an elimination step as in Rivest's algorithm. The

Multi-Layered Winnow + Rivest's (D, α)

1. Initialize all weights $w_{i,j}$ to 1.
2. Given an instance $\vec{x} = \langle x_1, \ldots, x_n \rangle$, calculate $\vec{y} := \mathbf{W} \cdot \vec{x}$.
3. Let i^* be the minimal i s.t. $y_i \geq n$. If there is no such i, set $i^* := D + 1$.
4. Let k^* be the minimal k s.t. $y_k > 0$. If there is no such k, set $k^* := D + 1$.
5. Output o_{i^*}, the output function of layer i^*.
6. Receive $c(\vec{x})$.
7. If the algorithm made a mistake:
   (a) If $i^* \leq D$:
   For each $1 \leq j \leq n$, set $w_{i^*,j} := \alpha^{-x_j} w_{i^*,j}$ (demotion).
   (b) If $i^* > 1$:
   For each $1 \leq j \leq n$, set $w_{i^*-1,j} := \alpha^{x_j} w_{i^*-1,j}$ (promotion).
   (c) If $k^* \leq D$ and $o_{k^*} = o_{i^*}$:
   For each $1 \leq j \leq n$, set $w_{k^*,j} := (1 - x_j) w_{k^*,j}$ (elimination).
8. Goto 2.

Figure 7: A hybrid Multi-Layered Winnow — including Rivest's elimination step

following theorem shows that the elimination step never eliminates weights of relevant variables. It follows that the hybrid can only help Multi-Layered Winnow in removing "noise" (large weights for irrelevant variables), reducing faster the potential function $\Psi(t)$. Thus, the hybrid's mistake bound cannot exceed the mistake bound of Multi-Layered Winnow.

Theorem 34 *An elimination step on layer k^*, never sets a weight $w_{k^*,j}$ to 0 if $x_j \in \tilde{R}_{k^*}$.*

Proof Suppose it did. Consider the first trial t, in which it happened. Before the contradicting elimination step, for any layer i, there was no $x_j \in \tilde{R}_i$ for which $w_{i,j} = 0$ (the demotion step never sets a weight to 0). That includes of course the weight for $x_{dom}(\vec{x}^{(t)})$ in layer $l_{dom}(\vec{x}^{(t)})$. Therefore, in trial t we must had that $k^* \leq l_{dom}(\vec{x}^{(t)})$, because k^* is the first layer in which there exists a non-zero weight for a 1-assigned variable. But if $k^* = l_{dom}(\vec{x}^{(t)})$, then since $o_{k^*} = o_{i^*}$, no mistake should have happened. If $k^* < l_{dom}(\vec{x}^{(t)})$ then by definition there was no $x_j \in \tilde{R}_{k^*}$ with assignment 1. Therefore, there was no $x_j \in \tilde{R}_{k^*}$, for which the weight $w_{k^*,j}$ was set to 0. ∎

By inspection of the algorithm, it is clear that the elimination step is not executed after each mistake, because the output of layer i^* might not be the same as the output of layer k^*. An adversary can use this to make the elimination step rather negligible: it might provide instances with only a few set variables whenever an elimination step cannot be avoided.

However, the elimination step might sharply reduce the mistake bound of Multi-Layered Winnow, if most instances contain many variables with an assignment 1. In Section 5 we show this empirically.

5. A Numerical Example

To gain some insight and for illustration purposes, we compared the empirical mistake bounds of four online algorithms: Rivest's algorithm, Balanced Winnow, our Multi-Layered Winnow and our hybrid algorithm. The PAC algorithm of Dhagat and Hellerstein was not tested, as we considered an online learning scenario. The Halving algorithm was not tested because of the overwhelming amounts of memory and time it requires.

All four algorithms were given the task of online learning a few decision lists with a different number of nodes and layers. The instances for the learning session, were created by giving each variable a random assignment, determined by flipping a biased coin, with probability p for giving 1. Two parameters were changed: the probability p, and the total number n of variables in each instance. For each choice of p and n and for each algorithm, we made 10 runs, and took their maximal mistake bound (having in mind the worst-case nature of mistake-bound analysis) along with the average mistake bound and the standard deviation. The mistake bound for each run was determined by taking the number of mistakes the algorithm had made, until it came up with a hypothesis that was consistent with the next 10,000,000 instances. Table 1 shows typical results for these tests. The results refer to the following decision list (contains 10 nodes in 4 layers): $\langle\langle x_1, 0\rangle, \langle x_2, 1\rangle, \langle x_3, 1\rangle, \langle x_4, 0\rangle, \langle x_5, 0\rangle, \langle x_6, 0\rangle, \langle x_7, 1\rangle, \langle x_8, 1\rangle, \langle x_9, 1\rangle, \langle x_{10}, 1\rangle, \langle 1, 0\rangle\rangle$.

We also calculated the theoretical mistake bound for these algorithms on the given decision list, as presented in previous sections. These bounds are given in Table 2. Notice that for Balanced Winnow the theoretical bound uses the parameters $\alpha = \frac{313}{312}$ and $\beta = \frac{311}{312}$, which by Theorem 9 fit the separation parameter $\delta = \frac{1}{156}$. This separation parameter is minimal for the given decision list. However, it turns out that using these values for α and β gives very poor empirical results, which do not reflect the real power of Balanced Winnow. We therefore used in the tests the values $\alpha = 2$ and $\beta = \frac{1}{3}$, which were found best. Also notice that for the hybrid algorithm we do not know of a better theoretical mistake bound than the mistake bound of the basic Multi-Layered Winnow.

The results indicate that Rivest's algorithm is highly sensitive to the proportion of 1's in the instances. If there are only a few variables with an assignment 1, the mistake bound rises dramatically, getting close to the theoretical mistake bound. This means that almost-worst-case sequences for Rivest's algorithm are easily produced using random data. On the other hand, a high proportion of 1's gives low mistake bounds, even for large n. This gap is because only 1 valued variables are eliminated from the layers.

A similar behaviour is observed for Balanced Winnow, though the gaps between low and high values of p are less dramatic. Also, the number of mistakes the algorithm makes is far from the theoretical bound by a few orders of magnitude. This indicates that worst-case sequences are relatively hard to produce, and are not likely to occur in a random data.

Multi-Layered Winnow, behaves differently: its empirical mistake bound usually decreases with p. This is easily understood by noticing that a high proportion of 1's can make

4. Did not stop making mistakes after 50,000,000 instances

Algorithm	p	$n = 20$			$n = 200$			$n = 2000$		
		max	avg	std	max	avg	std	max	avg	std
Rivest's (4)	0.01	**59**	**57.4**	1.0	639	621	8.3	1709	1675	20.5
	0.05	**56**	**52.1**	2.0	258	247	5.8	426	415	7.7
	0.5	**20**	**16.6**	2.4	**34**	**31.7**	1.8	**47**	**43.9**	2.3
Bal-Winnow $(2,\frac{1}{3})$	0.01	73	63.6	5.6	469	424	30.9	> 700 [4]		
	0.05	116	89.3	14.1	456	406	21.0	525	490	27.2
	0.5	67	58.1	3.9	148	132	11.0	208	167	25.3
ML-Winnow (4,2)	0.01	173	160	8.5	258	245	8.8	334	327	5.9
	0.05	187	164	14.0	294	265	16.8	419	390	20.9
	0.5	196	172	13.2	404	338	33.7	472	412	38.3
MLW+Rivest (4,2)	0.01	133	131	1.7	**220**	**213**	4.7	**327**	**318**	5.3
	0.05	140	132	5.7	**246**	**230**	12.5	**362**	**348**	8.5
	0.5	114	103	7.7	142	127	10.3	269	224	27.7

Table 1: Mistakes made, while learning a 10 nodes, 4 layers decision list (10 runs). Best results appear in boldface.

Algorithm	$n = 20$	$n = 200$	$n = 2000$
Rivest's (4)	80	800	8000
Balanced Winnow $\left(\frac{313}{312}, \frac{311}{312}\right)$	777087	1262142	1747196
Multi-Layered Winnow (4,2)	3299655	5359251	7418846
Multi-Layered Winnow + Rivest (4,2)	3299655	5359251	7418846

Table 2: Theoretical mistake bounds for a 10 nodes 4 layers decision list.

false demotions and promotions very expensive. As with Balanced Winnow, the theoretical mistake bound remains far away from the empirical results.

Finally, we observe that the combination of Multi-Layered Winnow and Rivest's algorithm, yields a balanced behaviour. The algorithm seems to be resistant to the value of p, and achieves low mistake bounds even for large n. Though the algorithm has the worst theoretical bounds (for this specific choice of target decision list and values for n), in most cases it achieves empirically the best results.

Remark: The above numerical example was generated based on one target decision list of length 10. We obtained qualitatively similar results with a few other target decision lists (with lengths in the range $[10, 100]$). Nevertheless, it should be emphasized that these numerical examples are certainly not exhaustive and therefore can at best provide only a weak evidence for the above observations.

6. Concluding Remarks and Open Problems

In this work, we considered the problem of online learning the target class of decision lists. We introduced a new online algorithm for learning decision lists, called Multi-Layered Winnow. This algorithm achieves a mistake bound of $O(r^D \log n)$, improving previous bounds. A combination of our algorithm with Rivest's is empirically shown to perform well. We now describe some open problems, regarding online learning of decision lists.

- *Is there an attribute efficient learning algorithm for learning decision lists?* This remains the most important open problem.
- *Is it possible to give better bounds for the proposed hybrid algorithm?* A careful analysis might possibly show that the elimination step significantly reduces the mistake bound.
- *Can Multi-Layered Winnow learn real-valued concepts?* We considered the boolean case where the domain was $\{0,1\}^n$. However, Multi-Layered Winnow can be easily generalized to classify instances over $[0,1]^n$, just like Winnow was generalized by Littlestone (1988). It might be interesting to characterize the real-valued functions that Multi-Layered Winnow can learn.
- *Can randomization help?* There are a few places where random decisions might have an effect. For example, it might be randomly chosen whether to demote, promote or eliminate a layer, and whether to increase or decrease a specific weight.
- *Which results may be achieved by using different kinds of loss functions?* The results obtained here, and in all previous results on online decision lists learning, were obtained in terms of 0/1 loss. One can hypothesize decision lists learning algorithms which generate "soft" predictions (e.g., values in $[0,1]$) or other types of non-binary predictions. Performance guarantees for such algorithms can be considered in terms of other loss functions (e.g., absolute or square).
- *Are there "natural" learning problems (and data sets), for which algorithms, specializing in learning decision lists, perform better than general purpose classifiers?*

Acknowledgments

We wish to thank Avrim Blum and the anonymous referees for their useful comments.

Appendix A.

In this appendix we prove the following theorem from Section 4.1:

Theorem 28 *For any decision list with D layers that satisfies $n \geq r\frac{\alpha}{\alpha-1}$, there exists a sequence of instances, on which Multi-Layered Winnow (D,α) makes at least $\prod_{i=1}^{D}|R_i|$ mistakes.*

For simplicity we use the following notations:

- The acronym MLW refers to the Multi-Layered Winnow from Figure 5.
- R_{D+1} denotes the set of irrelevant attributes.
- R_i^j $(i \leq j)$ denotes the set $\bigcup_{k=i}^{j} R_k$. If $i > j$ then R_i^j denotes the empty set.

Adversary's strategy

While there is a layer i with $x_j \in R_i$, s.t. $w_{i,j} + \sum_{x_k \in R_{i+1}^{D+1}} w_{i,k} < n$ do:

1. Let $i^\diamond$ be the lowest layer that satisfies the above with $x_{j^\diamond} \in R_{i^\diamond}$.
2. Give MLW the instance in which $x_l = \begin{cases} 1 & x_l \in R_{i^\diamond+1}^{D+1} \text{ or } l = j^\diamond \\ 0 & \text{otherwise} \end{cases}$.
3. While there is a layer $i^\star$, for which there is $x_j \in R_{i^\star+1}^{D+1}$ s.t. $w_{i^\star,j} > 1$ do:
   (a) Give MLW the instance in which $x_l = \begin{cases} 1 & x_l \in R_{i^\star+1}^{D+1} \\ 0 & \text{otherwise} \end{cases}$.

Figure 8: The adversary's strategy for generating a worst-case sequence.

Also, notice that the assumption in the theorem implies $\alpha|R_{D+1}| \geq n$ (as $n = r + |R_{D+1}|$).

The adversary's strategy for generating a worst-case sequence is given in Figure 8. The main idea is maximizing the number of false demotions (Case 3 updates), making each false demotion as "harmful" as possible and making each correct promotion as "useless" as possible. More specifically, the adversary keeps producing instances which cause a correct promotion of only a single variable $x_j \in R_{i^*-1}$, but imply a false demotion of all variables in R_{i^*}. This way, any promotion of a single variable from R_1 is accompanied by the demotion of all variables in R_2. Each time a single variable in R_2 is promoted again, all variables in R_3 are demoted, and so on. It follows that each promotion of a single variable in R_1, causes a cascade of updates (and therefore a sequence of mistakes), which is $\prod_{i=2}^{D} |R_i|$ long. Since there are $|R_1|$ such variables, the theorem holds.

Steps 1–2 of the algorithm take care of these worst-case false demotions, while Step 3 performs some sort of "clean-up", making Multi-Layered Winnow ready for the next false demotion. The following lemmas prove this formally.

Lemma 35 *At any time, in any layer i, $w_{i,j} = w_{i,k}$ for any $x_j, x_k \in R_{D+1}$.*

Proof Notice that the adversary gives MLW only instances in which every attribute from R_{D+1} is set. Therefore, MLW promotes and demotes their weights together in each layer. ■

Lemma 36 *Suppose that at some point the following conditions hold:*

1. *In a single layer $i^\star$, $w_{i^\star,j} \leq w_{i^\star,k} = \alpha$ for any $x_k \in R_{D+1}$ and for any $x_j \in R_{i^\star+1}^D$.*
2. *In any layer $i \neq i^\star$, $w_{i,j} \leq w_{i,k} \leq 1$ for any $x_k \in R_{D+1}$ and for any $x_j \in R_{i+1}^D$.*

Then by applying Step 3(a) the following statements become true:

1. *$w_{i^\star-1,j} \leq w_{i^\star-1,k} \leq \alpha$ for any $x_k \in R_{D+1}$, and for any $x_j \in R_{i^\star}^D$ (unless $i^\star = 1$).*
2. *In any layer $i \neq i^\star - 1$, $w_{i,j} \leq w_{i,k} \leq 1$ for any $x_k \in R_{D+1}$ and for any $x_j \in R_{i+1}^D$.*

Proof Since layer $i^\star$ is the only layer for which there is $x_j \in R_{i^\star+1}^{D+1}$ s.t. $w_{i^\star,j} > 1$, MLW is given in Step 3(a) the instance, in which only attributes in $R_{i^\star+1}^{D+1}$ are set. Since in any layer $i < i^\star$, $w_{i,j} \leq 1$ for any $x_j \in R_{i^\star+1}^{D+1}$, the value of y_i for such layers is smaller than n. Since $w_{i^\star,j} = \alpha$ for any $x_j \in R_{D+1}$, the value of $y_{i^\star}$ is at least $\alpha|R_{D+1}| \geq n$. It follows that layer $i^\star$ fires, and that MLW outputs $o_{i^\star}$.

MLW's prediction is clearly a mistake, since the highest layer which contains a set variable is $i^\star + 1$. Therefore, MLW demotes layer $i^\star$ and promotes layer $i^\star - 1$ (unless $i^\star = 1$). Following the demotion in layer $i^\star$, the weights $w_{i^\star,j}$ of variables in $R_{i^\star+1}^{D+1}$ cannot exceed 1. Following the promotion in layer $i^\star - 1$ (if such occured) we will have that $w_{i^\star-1,j} \leq w_{i^\star-1,k} \leq \alpha$ for any $x_k \in R_{D+1}$, and for any $x_j \in R_{i^\star}^D$. Weights in other layers are unchanged, therefore the lemma holds. ∎

Lemma 37 *Suppose the conditions of Lemma 36 hold before Step 3 starts. Then when Step 3 is done, in any layer i, $w_{i,j} \leq w_{i,k} \leq 1$ for any $x_k \in R_{D+1}$ and for any $x_j \in R_{i+1}^D$.*

Proof Notice that if the conditions for Lemma 36 hold, then after executing Step 3(a) there are three possibilities.

1. $i^\star = 1$, which proves the lemma.
2. $w_{i^\star-1,j} \leq w_{i^\star-1,k} \leq 1$ for any $x_k \in R_{D+1}$, and for any $x_j \in R_{i^\star}^D$, which proves lemma.
3. $w_{i^\star-1,j} \leq w_{i^\star-1,k} = \alpha$ for any $x_k \in R_{D+1}$, and for any $x_j \in R_{i^\star}^D$. In this case Lemma 36 can be repeatedly applied, this time with layer $i^\star - 1$ as the single layer.

By Lemma 35 there cannot be a fourth possibilty in which only some of the $x_k \in R_{D+1}$ have $w_{i^\star-1,k} = \alpha$, while others do not. Since in Step 3, Step 3(a) is executed as long as there is a layer $i^\star$, for which there is $x_j \in R_{i^\star+1}^{D+1}$ s.t. $w_{i^\star,j} > 1$, Lemma 36 can be applied again and again, until either Case 1 or Case 2 occur. ∎

Lemma 38 *In each execution of steps 1–3 the following statements are true:*

1. *Before the execution of Step 2, in any layer i, $w_{i,j} \leq w_{i,k} \leq 1$ for any $x_k \in R_{D+1}$ and for any $x_j \in R_{i+1}^D$.*
2. *Layer $i^\diamond + 1$ fires when MLW processes the instance it is given in Step 2.*
3. *MLW makes a mistake at Step 2.*

Proof By induction on t, the number of times Steps 1–3 were already executed.

Base case: $t = 0$. Recall that MLW starts with $\mathbf{W} = \mathbf{1}$ (proved 1). Therefore, $i^\diamond = D$, as for each $x_j \in R_D$ we have that $w_{D,j} + \sum_{x_k \in R_{D+1}} w_{D,k} = |R_{D+1}| + 1 < n$. Also, for any $1 \leq i \leq D$ we have that $y_i = |R_{D+1}| + 1$ (the number of set attributes in the instance). It follows that all y_i's are smaller than n ($1 \leq i \leq D$), and that layer $D + 1$ fires (proved 2). Thus, MLW predicts 0, which is definitely a mistake since the only relevant variable which is set is from layer D, so the output should be 1 (proved 3).

Assume the lemma holds for the first $t-1$ iterations. On Step 2 of iteration $t-1$, MLW made a mistake, which was followed by a demotion in layer $i^\diamond+1$ and a promotion in layer $i^\diamond$. It follows that when this step is done, layer $i^\diamond$ is the only layer in which there might be $x_j \in R_{i^\diamond+1}^{D+1}$ s.t. $w_{i^\diamond,j} > 1$ (assuming Statement 1 was true for iteration $t-1$).

If there is no such x_j then Statement 1 is true for iteration t. Otherwise, by Lemma 35 the conditions for Lemma 37 hold. Applying Lemma 37 proves Statement 1 for iteration t.

Statement 2 follows from Statement 1, as MLW is given the instance in which set attributes are either from $R_{i^\diamond+1}^{D+1}$ or $x_{j^\diamond}$. In any layer $i < i^\diamond$, $w_{i,j} \leq 1$ for any x_j with assignment 1, thus $y_i < n$ for these layers. In layer $i^\diamond$, $y_{i^\diamond} < n$ by definition, but since this is the lowest layer which satisfies the condition, we must have that $y_{i^\diamond+1} \geq n$ or that $i^\diamond = D$. On both cases, layer $i^\diamond+1$ fires, proving Statement 2. MLW's prediction is wrong, as the highest layer which contains a set variable is $i^\diamond$ (containing $x_{j^\diamond}$), proving Statement 3. ■

Lemma 39 *At any time, in any layer $i < D-1$, $w_{i,j} = w_{i,k}$ for any $x_j \in R_{i+2}^D$ and for any $x_k \in R_{D+1}$.*

Proof By Lemma 38 we have that on Step 2 layer $i^\diamond$ is promoted, while possibly layer $i^\diamond+1$ is demoted. The instance which causes these updates has all the attributes in $R_{i^\diamond+1}^{D+1}$ set, so their weights in the two layers are promoted and demoted together. By Lemma 37 we have that on Step 3(a) layer $i^\star$ is demoted, and that possibly layer $i^\star-1$ is promoted. Again, the instance has all the attributes in $R_{i^\star+1}^{D+1}$ set. ■

Lemma 40 *In any layer $i < D$, Step 3 never promotes weights $w_{i,j}$ for any $x_j \in R_{i+1}$.*

Proof Again, this is because Step 3(a) demotes layer $i^\star$ and possibly promotes layer $i^\star-1$, having only attributes in $R_{i^\star+1}^{D+1}$ set. ■

Lemma 41 *For any layer i and for any $x_j \in R_i$, $w_{i,j}$ is promoted only in Step 2 and only when $i^\diamond = i$ and $j^\diamond = j$.*

Proof Using the same arguments as in the proof of Lemma 40 we show that Step 3(a) never promotes $w_{i,j}$. By Lemma 38 we know that in Step 2 layer $i^\diamond+1$ fires. It follows that only layer $i^\diamond$ is promoted. Moreover, by the adversary's strategy $x_{j^\diamond}$ is the only set variable not in $R_{i^\diamond+1}^{D+1}$. Therefore, no $w_{i,j}$ can be promoted when $x_j \in R_i$, unless $i^\diamond = i$ and $j^\diamond = j$. ■

Lemma 42 *Let $i^\diamond(t)$ denote the layer chosen in Step 1 of iteration t. If $i^\diamond(t) < D$ then at the end of iteration t the followings hold:*

1. *For any $x_k \in R_{D+1}$ it holds that $w_{i^\diamond(t)+1,k} = \frac{1}{\alpha}$ and $w_{i,k} = 1$ for any $i \neq i^\diamond(t)+1$*
2. *On iteration $t+1$, $i^\diamond(t+1) = i^\diamond(t)+1$, and $x_{j^\diamond}$ can be any $x_j \in R_{i^\diamond(t+1)}$.*

If $i^\diamond(t) = D$ then at the end of iteration t, $w_{i,k} = 1$ for any layer i and for any $x_k \in R_{D+1}$.

Proof By induction on t, the iteration number.

Base case: From the proof of Lemma 38 we know that $i^\diamond(1) = D$. Therefore, no layer is demoted in Step 2, while layer D is promoted. Observe that for any $x_k \in R_{D+1}$, Step 3 will only change $w_{D,k}$ from α to 1, leaving any other $w_{i,k}$ unchanged.

Assume the lemma holds for the first $t-1$ iterations. If $i^\diamond(t) = D$ then by Lemma 38 and by the above assumption, we have that for any $x_k \in R_{D+1}$, $w_{D,k} \in \{1, \frac{1}{\alpha}\}$ before iteration t starts. Either way, after layer D is promoted in Step 2, and after Step 3 is applied, $w_{i,k} = 1$ for any $x_k \in R_{D+1}$ and for $i = D$, as well as for any other layer i.

Otherwise, we are assured that in layer $i^\diamond(t)+1$, $w_{i^\diamond(t)+1,k} = 1$ for any $x_k \in R_{D+1}$ before iteration t starts. It follows that when layer $i^\diamond(t) + 1$ is demoted in Step 2 of iteration t, these weights become $\frac{1}{\alpha}$. Step 2 also promotes layer $i^\diamond(t)$, but the weights of attributes in R_{D+1} are set back to 1 by step 3. Step 3 also makes sure that such weights in layers above layer $i^\diamond(t)$ will also remain 1. Layers below layer $i^\diamond(t) + 1$ are not affected by iteration t.

It now must be that for any $x_j \in R_{i^\diamond(t)+1}$, $w_{i^\diamond(t)+1,j} + \sum_{x_k \in R^{D+1}_{i^\diamond(t)+2}} w_{i^\diamond(t)+1,k} < n$, as before the last time $w_{i^\diamond(t)+1,j}$ was promoted (by Lemma 41 this could only happen in Step 2), this inequality was clearly true. Otherwise, by the MLW algorithm the promotion could not take place. $w_{i^\diamond(t)+1,j}$ now has the same value as before its last promotion, while for any $x_k \in R_{D+1}$, $w_{i^\diamond(t)+1,k}$ is now $\frac{1}{\alpha}$, and by our assumption could not be less at any time. By Lemma 39 we have that weights in $R^D_{i^\diamond(t)+3}$ behave just the same. By Lemma 40 we have that weights in $R_{i^\diamond(t)+2}$ cannot exceed their value on the last promotion. It follows that Statement 2 is true. ■

Proof (of Theorem 28). We now show that Steps 1–3 can be applied at least $\prod_{i=1}^{D} |R_i|$ times. Since each execution of Step 2 is followed by a mistake, this proves the theorem. If $|R_1| > 1$ then at some point layer 1 will become the lowest layer that satisifies the loop's inequality and will be chosen in Step 1. Using Lemma 42 we get that this iteration will start a cascade of iterations, which is $\prod_{i=2}^{D} |R_i|$ long. First, all relevant variables of layer 2 will be demoted. Each time one of the relevant variables of layer 2 will be promoted again all relevant variables of layer 3 will be demoted. Each time one of the relevant variables of layer 3 will be promoted again all relevant variables of layer 4 will be demoted, and so on. By Lemma 41 we are assured that a relevant variable is never promoted unless chosen. It follows that in any demoted layer, all relevant variables will in fact be chosen again. Layer 1 will be selected in Step 1 at least $|R_1|$ times, which proves this case.

If $|R_1| = 1$ it might be that layer 1 will never be selected in Step 1, when all the weights in layer 1 are 1. In this case, we can only be sure that layer 2 will be selected, once for each relevant variable it has. However, this is sufficient, as $R_1 = 1$ does not contribute to the product. ■

References

Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

Martin Anthony and Norman Biggs. *Computational Learning Theory: An Introduction*. Cambridge University Press, Cambridge, 1992.

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's Razor. *Information Processing Letters*, 24:377–380, 1987.

Avrim Blum. Rank-r decision trees are a subclass of r-decision lists. *Information Processing Letters*, 42(4):183–185, 1992.

Avrim Blum. On-line algorithms in machine learning. In *Proceedings of the Workshop on On-Line Algorithms*, Dagstuhl, June 1996. Available electronically at `http://www-2.cs.cmu.edu/~avrim`

Avrim Blum, Lisa Hellerstein, and Nicholas Littlstone. Learning in the presence of finitely or infintely many irrelevant attributes. *Journal of Computer and System Sciences*, 50(1):32–40, 1995.

Avrim Blum and Mona Singh. Learning functions of k terms. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, pages 144–153, Morgan Kaufmann, 1990.

Nader H. Bshouty and Lisa Hellerstein. Attribute-efficient learning in query and mistake-bound models. *Journal of Computer and System Sciences*, 56:310–319, 1998.

Aditi Dhagat and Lisa Hellerstein. PAC learning with irrelevant attributes. In *Proceedings of the IEEE symposium on Foundation of Computer Science*, pages 67–74, Santa Fe, New Mexico, 1994.

Thomas Eiter, Toshihide Ibaraki, and Kazuhisa Makino. Decision lists and related boolean functions. Research Report 9804, Institute of Informatics, University of Giessen, 1998.

David Guijarro, Victor Lavin and Vijay Raghavan. Monotone term decision lists. *Theoretical Computer Science*, 259(1-2):549–575, 2001.

David Helmbold, Robert Sloan, and Manfred K. Warmuth. Learning nested differences of intersection closed concept classes. *Machine Learning*, 5(2):165–196, 1990.

Nicholas Littlestone. Learning when irrelevant attributes abound: a new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

Nicholas Littlestone. *Mistake Bounds and Logarithmic Linear-Threshold Learning Algorithms*. PhD thesis, University of California, Santa Cruz, March 1989.

Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

Rocco A. Servadio. Computational sample complexity and attribute efficient learning. *Journal of Computer and System Sciences*, 60(1):161–178, 2000.

Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

Leslie G. Valiant. Projection learning. *Machine Learning*, 37(2):115–130, 1999.