ELSEVIER

Artificial Intelligence 97 (1997) 325-343

Artificial Intelligence

Technical Note

The Perceptron algorithm versus Winnow: linear versus logarithmic mistake bounds when few input variables are relevant [1]

J. Kivinen [a,2], M.K. Warmuth [b,3], P. Auer [c,4,*]

[a] *Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland*

[b] *Computer and Information Sciences, University of California, Santa Cruz, Santa Cruz, CA 95064, USA*

[c] *Institute for Theoretical Computer Science, Graz University of Technology, Klosterwiesgasse 32/2, A-8010 Graz, Austria*

Received October 1995; revised May 1996

Abstract

We give an adversary strategy that forces the Perceptron algorithm to make $\Omega(kN)$ mistakes in learning monotone disjunctions over N variables with at most k literals. In contrast, Littlestone's algorithm Winnow makes at most $O(k \log N)$ mistakes for the same problem. Both algorithms use thresholded linear functions as their hypotheses. However, Winnow does multiplicative updates to its weight vector instead of the additive updates of the Perceptron algorithm. In general, we call an algorithm additive if its weight vector is always a sum of a fixed initial weight vector and some linear combination of already seen instances. Thus, the Perceptron algorithm is an example of an additive algorithm. We show that an adversary can force any additive algorithm to make $(N + k - 1)/2$ mistakes in learning a monotone disjunction of at most k literals. Simple experiments show that for $k \ll N$, Winnow clearly outperforms the Perceptron algorithm also on nonadversarial random data. © 1997 Elsevier Science B.V.

Keywords: Linear threshold functions; Perceptron algorithm; Relevant variables; Multiplicative updates; Mistake bounds

* Corresponding author. Email: pauer@igi.tu-graz.ac.at.

[1] A preliminary version appeared in: *Proceedings 8th Annual Conference on Computational Learning Theory* (ACM, New York, 1995) 289-296.

[2] This work was done while the author was visiting the University of California, Santa Cruz. Supported by the Academy of Finland and by the ESPRIT Project NeuroCOLT. Email: jkivinen@cs.helsinki.fi.

[3] Supported by NSF grant IRI-9123692. Email: manfred@cse.ucsc.edu.

[4] Supported by the ESPRIT Project NeuroCOLT.

PII S0004-3702(97)00039-8

1. Introduction

This paper addresses the familiar problem of predicting with a *linear classifier*. The *instances*, for which one tries to predict a binary classification, are N-dimensional real vectors. A linear classifier is represented by a pair $(\boldsymbol{w}, \theta)$, where $\boldsymbol{w} \in \mathbb{R}^N$ is an N-dimensional *weight vector* and $\theta \in \mathbb{R}$ is a *threshold*. The linear classifier represented by the pair $(\boldsymbol{w}, \theta)$ has the value 1 on an instance $\boldsymbol{x}$ if $\boldsymbol{w} \cdot \boldsymbol{x} \geqslant \theta$, and the value 0 otherwise. Each instance $\boldsymbol{x} \in \mathbb{R}^N$ can be thought of as a value assignment for N input variables: x_i is the value for the ith input variable X_i.

In this paper we study the performance of certain families of learning algorithms for linear classifiers. We use as a test case monotone disjunctions, which are special linear classifiers. The monotone k-literal disjunction $X_{i_1} \vee \cdots \vee X_{i_k}$ corresponds to the linear classifier represented by the pair $(\boldsymbol{w}, 1/2)$ where $w_{i_1} = \cdots = w_{i_k} = 1$ and $w_j = 0$ for $j \notin \{i_1, \ldots, i_k\}$. For a given disjunction, the variables in the disjunction are called *relevant* and the remaining variables *irrelevant*. In this paper we are particularly interested in the case in which the number k of relevant variables is much smaller than the total number N of variables.

We analyze the algorithms in the following simple on-line prediction model of learning. The learning proceeds in trials. In trial t, the algorithm's current hypothesis is given by a weight vector $\boldsymbol{w}_t$ and a threshold θ_t. Upon receiving the next instance $\boldsymbol{x}_t$ the algorithm produces its prediction $\hat{y}_t$ using its current hypothesis. The algorithm then receives a binary outcome y_t and may update its weight vector and threshold to $\boldsymbol{w}_{t+1}$ and θ_{t+1}. If the outcome differs from the prediction, we say that the algorithm made a mistake. Following Littlestone [8,9], our goal is to minimize the total number of mistakes that the learning algorithm makes for certain sequences of trials.

The standard on-line algorithm for learning with linear classifiers is the simple Perceptron algorithm of Rosenblatt [15]. An alternate algorithm called Winnow was introduced by Littlestone [8,9]. To see how the algorithms work, consider a binary vector $\boldsymbol{x}_t \in \{0,1\}^N$ as an instance, and assume that the algorithm predicted 0 while the outcome was 1. Then both algorithms increment those weights $w_{t,i}$ for which the corresponding input $x_{t,i}$ was 1. These weights are called the *active* weights. Neither algorithm changes the *inactive* weights, i.e., the weights $w_{t,i}$ with $x_{t,i} = 0$. This causes the dot product to increase as it should, i.e., $\boldsymbol{w}_{t+1} \cdot \boldsymbol{x}_t > \boldsymbol{w}_t \cdot \boldsymbol{x}_t$. The difference between the algorithms is in how they increment the active weights. The Perceptron algorithm *adds* a positive constant to each of them, whereas Winnow *multiplies* each of them by a constant that is larger than one. Similarly, if the prediction was 1 and the outcome 0, then the active weights are decremented either by subtracting a positive constant or by dividing by a constant larger than one. The choice of the constants for the updates, as well as the initial weights and thresholds, can significantly affect the performance of the algorithms. We call choosing these parameters *tuning*.

In addition to the two basic algorithms described above, we wish to study a whole class of algorithms that includes the Perceptron algorithm. To be concrete, let η denote the positive constant that is added to or subtracted from the active weights of the Perceptron algorithm after each mistake, as described above. This constant is called the learning rate. Recall that $\hat{y}_t$ and y_t are the prediction and the correct outcome at trial t.

We can now write the weight vector $\boldsymbol{w}_t$ of the Perceptron algorithm as

$$\boldsymbol{w}_t = \boldsymbol{w}_1 + \sum_{j=1}^{t-1} \alpha_{t,j} \boldsymbol{x}_j, \tag{1}$$

where $\boldsymbol{w}_1$ is the initial weight vector and $\alpha_{t,j} = (y_j - \hat{y}_j)\eta$. In general, we say a learning algorithm is *additive* if its weight vector can be written in form (1) for *some* scalar coefficients $\alpha_{t,j}$. (The coefficients $\alpha_{t,j}$ for $t \leqslant j$ do not appear in (1), and we consider them undefined.) Thus, for an additive algorithm, the difference $\boldsymbol{w}_t - \boldsymbol{w}_1$ is in the span of the instances $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{t-1}$. The Perceptron algorithm has the special property that in the representation (1), $\alpha_{t,j} = \alpha_{t+1,j}$ for all $t > j$. This property allows for a more efficient implementation, as the algorithm can compute the next weight vector from the current instance and the last weight vector. However, this is by no means a necessary property of additive algorithms in general, and an additive algorithm might well store all the examples and allow the coefficient $\alpha_{t,j}$ of the jth instance to change as more information is obtained. In particular, the learning algorithm based on the ellipsoid method for linear programming [13] is an example of such a more complicated additive algorithm.

In contrast, the weight vector $\boldsymbol{w}_t$ of Winnow can be written in the form

$$w_{t,i} = w_{1,i} \prod_{j=1}^{t-1} \beta_{t,j}^{x_{j,i}} \tag{2}$$

where now $\beta_{t,j} = \exp((y_t - \hat{y}_t)\eta)$ for some positive learning rate η. Thus, we could call Winnow an example of *multiplicative* algorithms. Analogously with the Perceptron algorithm, Winnow has the property $\beta_{t,j} = \beta_{t+1,j}$ for all $t > j$, which simplifies implementation.

If there is a linear classifier $(\boldsymbol{u}, \psi)$ such that for all t we have $y_t = 1$ if and only if $\boldsymbol{u} \cdot \boldsymbol{x}_t \geqslant \psi$, we say that the trial sequence is *consistent* with the classifier $(\boldsymbol{u}, \psi)$ and say that the classifier $(\boldsymbol{u}, \psi)$ is a *target* of the trial sequence. It is easy to tune Winnow so that it makes at most $O(k \log N)$ mistakes [8, 10] on any sequence with a disjunction of at most k literals as a target. If the tuning is allowed to depend on k, the tighter bound $O(k + k \log(N/k))$ is obtainable. This upper bound is optimal to within a constant factor, since the Vapnik–Chervonenkis (VC) dimension [2, 20] of the class of k-literal disjunctions is $\Omega(k + k \log(N/k))$ [8] and this dimension is always a lower bound for the optimal mistake bound.

Thus, for example, if the number k of relevant variables in the target disjunction is kept constant, the number of mistakes made by Winnow grows only logarithmically in the total number N of variables. In this paper, we wish to contrast this to the behavior of additive algorithms, such as the Perceptron algorithm. For any value $k \leqslant N$, we show that any additive algorithm can be forced to make at least $(N + k - 1)/2$ mistakes in a trial sequence that has a monotone disjunction of at most k literals as a target. Thus, even for a constant k, the mistake bound of any additive algorithm grows at least linearly in N.

One might also ask whether there are significant differences within the class of additive algorithms, when they are applied to learning disjunctions with at most k literals. The best upper bound we know for learning k-literal monotone disjunctions with the Perceptron algorithm is $O(kN)$ mistakes, which comes from the classical Perceptron Convergence Theorem [4]. We also show that the Perceptron algorithm in its basic form can make $2k(N-k+1)+1$ mistakes, so the bound is essentially tight. On the other hand, it is possible to construct an additive algorithm that never makes more than $N+O(k\log N)$ mistakes. Thus, one can save at least a factor k by choosing the coefficients $\alpha_{t,j}$ in (1) in a more sophisticated manner than done by the Perceptron algorithm. However, this is only a minor improvement. Our lower bounds show that when k is small, then the mistake bound of *any* additive algorithm is *exponential* in the optimal mistake bound (in this case essentially the VC dimension).

The lower bounds for additive algorithms and for the Perceptron algorithm are based on an adversary argument. To show that the advantage of Winnow is not just an artifact of the adversarial learning model we performed some simple experiments. We found that with random data, too, the number of mistakes made by the Perceptron algorithm increases as a function of N much faster than the number of mistakes made by Winnow, when k is kept as a small constant.

The difference in the performances of the algorithms points out that the multiplicative algorithms have a different bias in their search for a good hypothesis. Intuitively, Winnow favors weight vectors that are in some sense sparse, and wins if the target weight vector is sparse ($k \ll N$ in the disjunction case). If the target weight vector is dense ($k=\Omega(N)$ in the disjunction case) and the instances are sparse (few non-zero components), the advantage of Winnow becomes much smaller. Note that if it is known that k is close to N, Winnow can also be tuned so that it simulates the classical elimination algorithm for learning disjunctions [19]. In this case it makes at most $N-k$ mistakes for k literal monotone disjunctions but is not robust against noise.

We introduce the details of the on-line prediction model and the algorithms we consider in Section 2. Section 3 gives our adversarial lower bound constructions for the class of additive algorithms. In Section 4 we show that the Perceptron algorithm is not the best additive algorithm for our problem. Our experimental results are presented in Section 5. In Section 6 we discuss some open problems and point out possible extensions to deal with noisy data and more general concept classes.

2. The prediction model and algorithms

2.1. The basic setting

We use a pair $(\boldsymbol{u},\psi)$ to represent a *linear classifier* with the *weight vector* $\boldsymbol{u}\in\mathbb{R}^N$ and the *threshold* ψ. The classifier represented by $(\boldsymbol{u},\psi)$ is denoted by $\Phi_{\boldsymbol{u},\psi}$ and defined for $\boldsymbol{x}\in\mathbb{R}^N$ by $\Phi_{\boldsymbol{u},\psi}(\boldsymbol{x})=1$ if $\boldsymbol{u}\cdot\boldsymbol{x}\geqslant\psi$ and $\Phi_{\boldsymbol{u},\psi}(\boldsymbol{x})=0$ otherwise. We are mostly concerned with the special case $\boldsymbol{x}\in\{0,1\}^N$.

An *N-dimensional trial sequence* is a game played between two players, the learner and the teacher. For the purposes of the present paper, we restrict ourselves to learners

that predict using linear classifiers, in a manner we shall soon describe in more detail. The game has l rounds, or *trials*, for some positive integer l. In a trial sequence, trial t for $t = 1, \ldots, l$ proceeds as follows:

(i) The learner chooses its *hypothesis* $(\boldsymbol{w}_t, \theta_t)$, with $\boldsymbol{w}_t \in \mathbb{R}^N$ and $\theta_t \in \mathbb{R}$.

(ii) The teacher presents the *instance* $\boldsymbol{x}_t \in \{0,1\}^N$.

(iii) The learner's *prediction* is now defined to be $\hat{y}_t = \Phi_{\boldsymbol{w}_t,\theta_t}(\boldsymbol{x}_t)$.

(iv) The teacher presents the *outcome* $y_t \in \{0,1\}$.

After the last trial, the teacher must present a *target* $(\boldsymbol{u}, \psi)$, with $\boldsymbol{u} \in \mathbb{R}^N$ and $\psi \in \mathbb{R}$, such that $\Phi_{\boldsymbol{u},\psi}(\boldsymbol{x}_t) = y_t$ for all t. The goal of the learner is to minimize the number of *mistakes*, i.e., trials with $y_t \neq \hat{y}_t$. The teacher, on the other hand, tries to force the learner to make many mistakes.

This worst-case model of prediction, with an adversarial teacher, can be justified by the fact that there are algorithms that can be guaranteed to make a reasonable number of mistakes as learners in this model. We soon introduce two such algorithms, the Perceptron algorithm and Winnow, and their mistake bounds. The model could be made even more adversarial by allowing the teacher a given number of *classification errors*, i.e., trials with $\Phi_{\boldsymbol{u},\psi}(\boldsymbol{x}_t) \neq y_t$. On the other hand, we often restrict the teacher by restricting the target. In this paper we consider the case where the target is required to be a monotone k-literal disjunction, i.e., to have $\psi = 1/2$ and $\boldsymbol{u} \in \{0,1\}^N$ with exactly k components u_i with value 1.

An *on-line linear prediction algorithm* is a deterministic algorithm that can act as the learner in the game described above. A general on-line prediction algorithm would be allowed to choose as its hypothesis any mapping from $\{0,1\}^N$ to $\{0,1\}$ instead of a linear classifier. For the class of on-line *linear* prediction algorithms to be less powerful than the full class of on-line prediction algorithms it is essential that the learner is required to fix its tth hypothesis $(\boldsymbol{w}_t, \theta_t)$ before the tth instance $\boldsymbol{x}_t$ is given. Otherwise, the learner could run a simulation of any on-line prediction algorithm and at each trial choose its hypothesis to be either the constant threshold function $(\boldsymbol{0}, -1)$ or $(\boldsymbol{0}, 1)$ depending on what the prediction of the simulated algorithm would be on the instance $\boldsymbol{x}_t$. This would achieve the power of an arbitrary on-line prediction algorithm while nominally using linear classifiers as hypotheses.

We use the term *trial sequence* for the sequence $S = ((\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l))$ that gives the teacher's part of the game. Given a fixed deterministic learning algorithm, the learner's part is completely determined by the trial sequence.

2.2. The Perceptron algorithm and Winnow

Both for the Perceptron algorithm and Winnow, the new hypothesis $(\boldsymbol{w}_{t+1}, \theta_{t+1})$ depends only on the old hypothesis $(\boldsymbol{w}_t, \theta_t)$ and the observed instance $\boldsymbol{x}_t$ and outcome y_t. We call this dependence the *update rule* of the algorithm. In addition to the update rule, we must also give the *initial hypothesis* $(\boldsymbol{w}_1, \theta_1)$ to characterize an algorithm. The most usual initial weight vectors $\boldsymbol{w}_1$ are of the form $\boldsymbol{w}_1 = (a, \ldots, a)$ for some scalar $a \in \mathbb{R}$. Note that the definition of a linear on-line prediction algorithm allows the new hypothesis $(\boldsymbol{w}_{t+1}, \theta_{t+1})$ to depend on earlier instances $\boldsymbol{x}_i$ and outcomes y_i, $i < t$, and there are indeed some more sophisticated algorithms with such dependencies.

The Perceptron algorithm and Winnow are actually families of algorithms, both parameterized by the initial hypothesis and a *learning rate* $\eta > 0$. To give the update rules of the algorithms, let us first denote by σ_t the sign of the prediction error at trial t, that is, $\sigma_t = \hat{y}_t - y_t$. In their basic forms, both the Perceptron algorithm and Winnow maintain a fixed threshold, i.e., $\theta_t = \theta_1$ for all t. Given an instance $\boldsymbol{x}_t \in \{0,1\}^N$, the sign σ_t, and a learning rate η, the update of the Perceptron algorithm can be written componentwise as

$$w_{t+1,i} = w_{t,i} - \eta\sigma_t x_{t,i} \tag{3}$$

and the update of Winnow as

$$w_{t+1,i} = w_{t,i}\mathrm{e}^{-\eta\sigma_t x_{t,i}}. \tag{4}$$

Note that this basic version of Winnow (the algorithm Winnow2 of [8]) only uses positive weights (assuming that the initial weights are positive). The algorithm can be generalized for negative weights by a simple reduction [8]. See Littlestone [9] and Auer and Warmuth [1] for a discussion on the learning rates and other parameters of Winnow. Here we just point out the standard method of allowing the threshold to be fixed to 0 at the cost of increasing the dimensionality of the problem by one. To do this, each instance $\boldsymbol{x} = (x_1,\ldots,x_N)$ is replaced by $\boldsymbol{x}' = (1,x_1,\ldots,x_N)$. Then a linear classifier $(\boldsymbol{w},\theta)$ with a nonzero threshold can be replaced by $(\boldsymbol{w}',0)$ where $\boldsymbol{w}' = (-\theta,w_1,\ldots,w_N)$. This useful technique gives a method for effectively updating the threshold together with the components of the weight vector.

It is known that if the target is a monotone k-literal disjunction, Winnow makes $\mathrm{O}(k\log N)$ mistakes [8]. There are several other algorithms that make multiplicative weight updates and achieve similar mistake bounds [9]. The best upper bound we know for the Perceptron algorithm comes from the Perceptron Convergence Theorem given, e.g., by Duda and Hart [4, pp. 142–145]. Assuming that the target is a monotone k-literal disjunction and the instances $\boldsymbol{x}_t \in \{0,1\}^N$ satisfy $\sum_i x_{t,i} \leqslant X$ for some value X, the bound is $\mathrm{O}(kX)$ mistakes. (Note that always $X \leqslant N$.) In Section 4 we show that this bound can be tight. We give an adversary strategy that forces a version of the Perceptron algorithm to make $\Omega(kN)$ mistakes when learning k-literal disjunctions.

As Maass and Turán [13] have pointed out, several linear programming methods can be transformed into efficient linear on-line prediction algorithms. Most notably, this applies to Khachiyan's ellipsoid algorithm [6] and to a newer algorithm due to Vaidya [18]. Vaidya's algorithm achieves an upper bound of $\mathrm{O}(N^2\log N)$ mistakes for an arbitrary linear classifier as the target when the instances are from $\{0,1\}^N$. The Perceptron algorithm and Winnow are not suitable for learning arbitrary linear classifiers over the domain $\{0,1\}^N$. Maass and Turán show that in the worst case the number of mistakes of both algorithms is exponential in N. The proof of the $\mathrm{O}(N^2\log N)$ mistake bound for general linear classifiers is based on first observing that arbitrary real weights in a linear classifier can be replaced with integer weights no larger than $\mathrm{O}(N^{\mathrm{O}(N)})$ without changing the classification of any point in $\{0,1\}^N$. For monotone disjunctions, all the weights u_i and the threshold ψ can directly be chosen from $\{0,1,2\}$, which leads to the better bound of $\mathrm{O}(N\log N)$ mistakes.

In what follows we assume that the arithmetic operations of the various algorithms can be performed exactly, without rounding errors.

2.3. Additive algorithms

The main results of this paper are lower bounds for the class of *additive* algorithms.

Definition 1. A linear on-line prediction algorithm is *additive* if for all t, the algorithm's tth weight vector $\boldsymbol{w}_t$ can be written as

$$\boldsymbol{w}_t = \boldsymbol{w}_1 + \sum_{j=1}^{t-1} \alpha_{t,j} \boldsymbol{x}_j \tag{5}$$

for some fixed initial weight vector $\boldsymbol{w}_1$ and for some coefficients $\alpha_{t,j} \in \mathbb{R}$.

As we are considering on-line prediction algorithms, the coefficients $\alpha_{t,j}$ in (5) of course depend only on the instances $\boldsymbol{x}_i$ and outcomes y_i for $i < t$.

The Perceptron algorithm is additive. By comparing (3) and (5) we see that we can take $\alpha_{t,j} = -\eta\sigma_j$ for the Perceptron algorithm.

Consider now Winnow with initial weights $\boldsymbol{w}_1 = \mathbf{1}$, learning rate $\eta = \ln 2$, and threshold $\theta_1 = N = 3$. Let $\boldsymbol{x}_1 = (1,1,0)$, $\boldsymbol{x}_2 = (1,0,1)$, and $y_1 = y_2 = 1$. This is consistent with the target $((1,0,0),1/2)$, and gives $\boldsymbol{w}_3 = (4,2,2)$. As the vector $\boldsymbol{w}_3 - \boldsymbol{w}_1 = (3,1,1)$ is not in the span of $\{\boldsymbol{x}_1, \boldsymbol{x}_2\}$, we see that Winnow is not additive.

Recall that a square matrix $A \in \mathbb{R}^{m \times m}$ is *orthogonal* if its columns are orthogonal to each other, and *orthonormal* if it is orthogonal and its columns have Euclidean norm 1. Thus, for an orthogonal matrix A the product $A^{\mathrm{T}}A$ is a diagonal matrix, and for an orthonormal matrix $A^{\mathrm{T}}A = I$ where I is the $m \times m$ identity matrix.

Consider an orthonormal matrix $A \in \mathbb{R}^{m \times m}$. If we think of a vector $\boldsymbol{x} \in \mathbb{R}^m$ as a list of coordinates of some point in m-dimensional space, then $A\boldsymbol{x}$ can be considered the list of coordinates of the same point in a new coordinate system. The basis vectors of the new coordinate system are represented in the original coordinate system by the column vectors of A. Thus, orthonormal matrices represent rotations (and reflections) of the coordinate system. Let us write $\tilde{\boldsymbol{x}} = A\boldsymbol{x}$. Rotations preserve angles: $\tilde{\boldsymbol{w}} \cdot \tilde{\boldsymbol{x}} = (A\boldsymbol{w})^{\mathrm{T}}A\boldsymbol{x} = \boldsymbol{w}^{\mathrm{T}}(A^{\mathrm{T}}A)\boldsymbol{x} = \boldsymbol{w} \cdot \boldsymbol{x}$. In a situation in which this geometric interpretation is meaningful, it would be natural to assume that the choice of coordinate system is irrelevant, i.e., nothing changes if one systematically replaces $\boldsymbol{x}$ by $\tilde{\boldsymbol{x}}$ everywhere.

Definition 2. A linear on-line prediction algorithm is *rotation invariant* if for all orthonormal matrices $A \in \mathbb{R}^{N \times N}$ and all trial sequences $S = ((\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l))$, the predictions made by the algorithm given the trial sequence S are the same as its predictions given the trial sequence $\tilde{S} = ((A\boldsymbol{x}_1, y_1), \ldots, (A\boldsymbol{x}_l, y_l))$.

In general, being rotation invariant is not necessarily a natural or desirable property of an algorithm. For instance, the components $x_{t,i}$ of the instances often represent some

physical quantities that for different i may have entirely different units. It is also common to scale the instances to make, for example, $-1 \leqslant x_{t,i} \leqslant 1$ hold for all t and i. In such cases, the original coordinate system clearly has a special meaning. However, there are several common algorithms that are rotation invariant.

To discuss the rotation invariance of the Perceptron algorithm and Winnow, we extend them to arbitrary real inputs simply by allowing arbitrary real $x_{t,i}$ in the update rules (3) and (4). Alternatively, we could have restricted ourselves to rotations that map $\{0,1\}^N$ to itself, but that would have left us with just variable renamings, which are not very interesting.

The Perceptron algorithm with zero start vector is rotation invariant. The linear on-line prediction algorithm one obtains by applying the reduction given by Maass and Turán to the ellipsoid method for linear programming is also rotation invariant. This is because the initial ellipsoid used by the algorithm is a ball centered at the origin, and the updates of the ellipsoid are done in a rotation invariant manner. If one uses Vaidya's algorithm for the linear programming in the reduction, one gets an algorithm that is not rotation invariant. Vaidya's algorithm uses a polytope that is updated in a rotation invariant manner, but the initialization of the polytope cannot be rotation invariant.

Winnow is not rotation invariant, either. To see this, consider a two-dimensional trial sequence with $\boldsymbol{x}_1 = (1,0)$, $\boldsymbol{x}_2 = (0,1)$, and $y_1 = y_2 = 1$. Assume that Winnow uses the initial weight vector $\boldsymbol{w}_1 = \mathbf{1}$ and a threshold such that $\hat{y}_1 = \hat{y}_2 = 0$. Then after the two trials, Winnow has the weight vector $\boldsymbol{w}_3 = (\mathrm{e}^{\eta}, \mathrm{e}^{\eta})$. Consider now the orthonormal matrix

$$A = 2^{-1/2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

After seeing the counterexamples $(A\boldsymbol{x}_1, 1)$ and $(A\boldsymbol{x}_2, 1)$, Winnow has the hypothesis $\widetilde{\boldsymbol{w}}_3 = (\mathrm{e}^{\eta\sqrt{2}}, 1)$. As $\boldsymbol{w}_3$ is linear in e^{η} and $\widetilde{\boldsymbol{w}}_3$ is not, it is clear that Winnow cannot be rotation invariant. To be specific, consider the instance $\boldsymbol{x}_3 = (r, -r)$ for some $r \in \mathbb{R}$. Then $\boldsymbol{w}_3 \cdot \boldsymbol{x}_3 = 0$, while $\widetilde{\boldsymbol{w}}_3 \cdot A\boldsymbol{x}_3 = r\sqrt{2}$. Therefore, for some values of r the predictions of Winnow are not the same for the rotated and the original instances.

We have the following general result.

Theorem 3. *If a linear on-line prediction algorithm is rotation invariant, then it is an additive algorithm with zero initial weight vector.*

Proof. Let $(\boldsymbol{w}_{t+1}, \theta_{t+1})$ be the hypothesis of a rotation invariant algorithm after it has seen the instances $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$ and outcomes $y_1, \ldots, y_t$. We claim that $\boldsymbol{w}_{t+1}$ is in the subspace spanned by the set $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t\}$. It is easy to construct an orthonormal matrix $A \in \mathbb{R}^{N \times N}$ such that $A\boldsymbol{x}_i = \boldsymbol{x}_i$ for $i = 1, \ldots, t$, and $A\boldsymbol{x} = -\boldsymbol{x}$ for any vector $\boldsymbol{x}$ that is orthogonal to X. Since $A\boldsymbol{x}_t = \boldsymbol{x}_t$, the definition of a rotation invariant algorithm implies for all $\boldsymbol{x} \in \mathbb{R}^N$ that $\boldsymbol{w}_{t+1} \cdot \boldsymbol{x} \geqslant \theta_{t+1}$ if and only if $\boldsymbol{w}_{t+1} \cdot A\boldsymbol{x} \geqslant \theta_{t+1}$. Therefore, $\boldsymbol{w}_{t+1} \cdot \boldsymbol{x} = \boldsymbol{w}_{t+1} \cdot A\boldsymbol{x}$ for all $\boldsymbol{x}$. If we choose a vector $\boldsymbol{x}$ that is orthogonal to X, we have $\boldsymbol{w}_{t+1} \cdot \boldsymbol{x} = \boldsymbol{w}_{t+1} \cdot A\boldsymbol{x} = -\boldsymbol{w}_{t+1} \cdot \boldsymbol{x}$, so $\boldsymbol{w}_{t+1} \cdot \boldsymbol{x} = 0$. Hence, $\boldsymbol{w}_{t+1}$ is in the subspace spanned by X. □

Conversely, consider an algorithm that is additive and has zero initial weight vector. If further the algorithm's thresholds θ_t and the coefficients $\alpha_{t,j}$ in (5) depend only on the outcomes and the dot products $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$, then the algorithm is easily seen to be rotation invariant.

3. Lower bounds for additive algorithms

Given two vectors $\boldsymbol{p} \in \{-1,1\}^N$ and $\boldsymbol{q} \in \{-1,1\}^N$, we denote by $D(\boldsymbol{p},\boldsymbol{q})$ their Hamming distance, i.e., the number of indices i such that $p_i \neq q_i$.

In the proofs we use some basic properties of *Hadamard matrices*. A Hadamard matrix is an orthogonal matrix with its elements in $\{-1,1\}$. Multiplying a row or a column of a Hadamard matrix by -1 leaves it a Hadamard matrix. Note that if $\boldsymbol{p}$ and $\boldsymbol{q}$ are two different rows in an $N \times N$ Hadamard matrix, we have $D(\boldsymbol{p},\boldsymbol{q}) = N/2$. The following definition gives the most straightforward way of obtaining high-dimensional Hadamard matrices.

Definition 4. When $n = 2^d$ for some d, let H_n be the $n \times n$ Hadamard matrix obtained by the recursive construction $H_1 = (1)$,

$$H_{2n} = \begin{pmatrix} H_n & H_n \\ H_n & -H_n \end{pmatrix}.$$

Note that every element in the first column of H_n is 1, as is every element in the first row. We also have the following property.

Proposition 5. *For $n = 2^d$ where d is a positive integer, let H_n be the $n \times n$ Hadamard matrix defined in Definition 4. Then for any vector $\boldsymbol{p} \in \{-1,1\}^n$ there is an index j such that $D(\boldsymbol{p},\boldsymbol{q}) \geqslant n/2$ holds if $\boldsymbol{q}$ is the jth column of H_n.*

Consider now an additive algorithm and its weight vector given in (5). Its prediction on the instance $\boldsymbol{x}_t$ can depend only on the dot products $\boldsymbol{w}_1 \cdot \boldsymbol{x}_t$ and $\boldsymbol{x}_i \cdot \boldsymbol{x}_t$ where $i < t$. Thus, for an adversary it would be helpful to have for $\boldsymbol{x}_t$ two different candidates z' and z'' for which these dot products do not differ. This motivates the following definition.

Definition 6. Let $B = ((z'_1, z''_1), \ldots, (z'_l, z''_l))$, where z'_t and z''_t are in $\{0,1\}^N$ for all t. We say that B is a sequence with *pairwise constant dot products* if for $1 \leqslant i < t \leqslant l$ we have $z'_i \cdot z'_t = z'_i \cdot z''_t$ and $z''_i \cdot z'_t = z''_i \cdot z''_t$.

Our basic idea is to form a sequence with pairwise constant dot products by choosing z'_t to be the tth row of an $l \times l$ Hadamard matrix, and $z''_t = -z'_t$, but a simple transformation is necessary to make the instances binary. We also add some padding to the instances to handle the case $k > 1$ efficiently.

Merely having pairwise constant dot products is not sufficient for generating mistakes. The adversary needs a target $(\boldsymbol{u}, \psi)$ that is suitably different from the algorithm's

initial hypothesis. To get an idea about this, consider two instance candidates z'_t and z''_t with, say, $\boldsymbol{w}_t \cdot z'_t \leqslant \boldsymbol{w}_t \cdot z''_t$. Depending on the algorithm's threshold θ_t, the algorithm may either predict $\hat{y}_t = 0$ for both $\boldsymbol{x}_t = z'_t$ and $\boldsymbol{x}_t = z''_t$, predict $\hat{y}_t = 0$ for $\boldsymbol{x}_t = z'_t$ and $\hat{y}_t = 1$ for $\boldsymbol{x}_t = z''_t$, or predict $\hat{y}_t = 1$ for both $\boldsymbol{x}_t = z'_t$ and $\boldsymbol{x}_t = z''_t$. If the target $(\boldsymbol{u}, \psi)$ now is such that $\boldsymbol{u} \cdot z''_t < \psi < \boldsymbol{u} \cdot z'_t$, then by choosing either z'_t or z''_t for the tth instance $\boldsymbol{x}_t$ the adversary can force the algorithm to make a mistake regardless of its choice of θ_t. Note that if the adversary is choosing its instances from a sequence with pairwise constant dot products and the algorithm is additive, the condition $\boldsymbol{w}_t \cdot z'_t \leqslant \boldsymbol{w}_t \cdot z''_t$ is equivalent with $\boldsymbol{w}_1 \cdot z'_t \leqslant \boldsymbol{w}_1 \cdot z''_t$ and hence independent of the updates made by the algorithm. This leads to the following definition.

Definition 7. Let $B = ((z'_1, z''_1), \ldots, (z'_l, z''_l))$, where z'_t and z''_t are in $\{0,1\}^N$ for all t. Let $\boldsymbol{w} \in \mathbb{R}^N$ be a weight vector and $(\boldsymbol{u}, \psi) \in \mathbb{R}^N \times \mathbb{R}$ a linear classifier. We say that the weight vector $\boldsymbol{w}$ and the classifier $(\boldsymbol{u}, \psi)$ *differ* at trial t on the sequence B if either $\boldsymbol{w} \cdot z'_t \leqslant \boldsymbol{w} \cdot z''_t$ and $\boldsymbol{u} \cdot z'_t > \psi > \boldsymbol{u} \cdot z''_t$, or $\boldsymbol{w} \cdot z'_t \geqslant \boldsymbol{w} \cdot z''_t$ and $\boldsymbol{u} \cdot z'_t < \psi < \boldsymbol{u} \cdot z''_t$.

Using the basic idea given above, one can now prove the following result.

Lemma 8. *Let $B = ((z'_1, z''_1), \ldots, (z'_l, z''_l))$ be a sequence with pairwise constant dot products. Consider an additive linear on-line prediction algorithm with the initial weight vector $\boldsymbol{w}_1$. For any linear classifier $(\boldsymbol{u}, \psi)$, the adversary can choose a trial sequence with $(\boldsymbol{u}, \psi)$ as target and $\boldsymbol{x}_t \in \{z'_t, z''_t\}$ for all t in such a way that the algorithm makes a mistake at all trials at which $\boldsymbol{w}_1$ and $(\boldsymbol{u}, \psi)$ differ on B.*

Proof. Consider a trial sequence $S = ((\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l))$, in which $y_t = \Phi_{\boldsymbol{u},\psi}(\boldsymbol{x}_t)$ for $t = 1, \ldots, l$. Assume that for $i = 1, \ldots, t-1$ we have $\boldsymbol{x}_i \in \{z'_i, z''_i\}$. Let $(\boldsymbol{w}_t, \theta_t)$ be the hypothesis of an additive linear on-line prediction algorithm at trial t. Write $\boldsymbol{w}_t = \boldsymbol{w}_1 + \sum_{j=1}^{t-1} \alpha_{t,j} \boldsymbol{x}_j$, and assume that the initial weight vector $\boldsymbol{w}_1$ and the target $(\boldsymbol{u}, \psi)$ differ at trial t on the sequence B.

Consider first the case with $\boldsymbol{w}_1 \cdot z'_t \leqslant \boldsymbol{w}_1 \cdot z''_t$ and $\boldsymbol{u} \cdot z'_t > \psi > \boldsymbol{u} \cdot z''_t$. Since B has pairwise constant dot products, we also have $\boldsymbol{w}_t \cdot z'_t \leqslant \boldsymbol{w}_t \cdot z''_t$. If $\theta_t \leqslant \boldsymbol{w}_t \cdot z'_t$, the adversary chooses $\boldsymbol{x}_t = z''_t$. In this case $\hat{y}_t = 1$ and $y_t = 0$, so the algorithm makes a mistake. Otherwise, the adversary chooses $\boldsymbol{x}_t = z'_t$, so $\hat{y}_t = 0$ and $y_t = 1$ and again the algorithm makes a mistake. The case $\boldsymbol{w}_1 \cdot z'_t \geqslant \boldsymbol{w}_1 \cdot z''_t$ and $\boldsymbol{u} \cdot z'_t < \psi < \boldsymbol{u} \cdot z''_t$ is similar. □

Thus, proving lower bounds is reduced to finding for a given initial weight vector a sequence with pairwise constant dot products and a target such that the initial weight vector and the target differ sufficiently often. The sequence we use is given in the following definition.

Definition 9. Let $N = 2^d + k - 1$ for some positive integers d and k. Let H_{2^d} be the $2^d \times 2^d$ Hadamard matrix given in Definition 4, and for $t = 1, \ldots, 2^d$, let $\boldsymbol{h}_t$ be the tth row of H_{2^d}. We define B_{H} to be the sequence $((z'_1, z''_1), \ldots, (z'_{2^d}, z''_{2^d}))$ where

$$z'_t = ((h_{t,1}+1)/2, \ldots, (h_{t,2^d}+1)/2, 0, \ldots, 0),$$

$$z''_t = ((-h_{t,1}+1)/2, \ldots, (-h_{t,2^d}+1)/2, 0, \ldots, 0).$$

Lemma 10. *The sequence B_H defined in Definition 9 has pairwise constant dot products.*

Proof. Follows from the facts that $\boldsymbol{h}_t \cdot \boldsymbol{h}_{t'} = 0$ for $t \neq t'$ and $\sum_{i=1}^{N} h_{t,i} = -\sum_{i=1}^{N} h_{t,i} = 0$ for $t \geqslant 2$. □

The basic idea of the following lower bound proofs is to first find a monotone 1-literal disjunction that differs with a given initial weight vector at as many trials of B_H as possible. The adversary can then use the sequence B_H to force mistakes, as in Lemma 8. This part of the sequence effectively uses only the first 2^d variables and chooses exactly one of them to be relevant. The adversary is then still free to choose any subset of the remaining $k-1$ variables as relevant, which makes it easy to produce $k-1$ additional mistakes in $k-1$ additional trials.

Theorem 11. *Let $N = 2^d + k - 1$ for some positive integers d and k. For any additive linear on-line prediction algorithm with a zero initial weight vector $\boldsymbol{w}_1 = \boldsymbol{0}$ there is an N-dimensional trial sequence with a monotone disjunction of at most k-literals as a target such that the algorithm makes N mistakes on the trial sequence.*

Proof. Let $B_H = ((z'_1, z''_1), \ldots, (z'_{2^d}, z''_{2^d}))$ be as in Definition 9. We then have $z'_{t,1} = 1$ and $z''_{t,1} = 0$ for $t = 1, \ldots, 2^d$. Consider now a vector $\boldsymbol{u} \in \{0,1\}^N$ with $u_1 = 1$ and $u_i = 0$ for $2 \leqslant i \leqslant 2^d$. The components u_i for $2^d + 1 \leqslant i \leqslant N$ are left unspecified for now. The constraints we have set for $\boldsymbol{u}$ imply $\boldsymbol{u} \cdot \boldsymbol{z}''_t = 0 < 1/2 < 1 = \boldsymbol{u} \cdot \boldsymbol{z}'_t$ for all t. We always have $\boldsymbol{0} \cdot \boldsymbol{z}'_t = \boldsymbol{0} \cdot \boldsymbol{z}''_t = 0$. Hence, the zero weight vector and the classifier $(\boldsymbol{u}, 1/2)$ differ on B_H at trials $1, \ldots, 2^d$, regardless of how the remaining components $u_{2^d+1}, \ldots, u_N$ are chosen. By Lemma 8, the adversary can therefore choose the instances from the sequence B_H in such a way that an additive algorithm makes a mistake on every one of the trials $1, \ldots, 2^d$ when $(\boldsymbol{u}, 1/2)$ is the target.

After the first 2^d instances chosen from the sequence B_H, the adversary continues the trial sequence with an additional $k-1$ trials, in which the instances are unit vectors. Thus, for $t = 2^d + 1, \ldots, N$, we set $x_{t,t} = 1$ and $x_{t,i} = 0$ for $i \neq t$. After seeing the algorithm's tth hypothesis $(\boldsymbol{w}_t, \theta_t)$, the adversary chooses $u_t = 0$ and $y_t = 0$ if $\boldsymbol{w}_t \cdot \boldsymbol{x}_t \geqslant \theta_t$, and $u_t = 1$ and $y_t = 1$ otherwise. Then clearly the algorithm makes a mistake at each of the trials $2^d + 1, \ldots, N$, and $(\boldsymbol{u}, 1/2)$ is a monotone disjunction which at most k literals and is consistent with the trial sequence. The total number of mistakes made by the algorithm is $2^d + k - 1 = N$. □

Theorem 12. *Let $N = 2^d + k - 1$ for some positive integers d and k. For any additive linear on-line prediction algorithm there is an N-dimensional trial sequence with a monotone disjunction with at most k literals as a target such that the algorithm makes at least $(N + k - 1)/2$ mistakes on the trial sequence.*

Proof. Let $\boldsymbol{w}_1$ be the initial weight vector of the algorithm. Define a vector $\boldsymbol{p} \in \{-1,1\}^{2^d}$ by $p_t = -1$ if $\boldsymbol{w}_1 \cdot \boldsymbol{z}'_t \leqslant \boldsymbol{w}_1 \cdot \boldsymbol{z}''_t$, and $p_t = 1$ otherwise. According to Proposition 5, we can choose an index i such that $D(\boldsymbol{p},\boldsymbol{q}) \geqslant 2^{d-1}$ when $\boldsymbol{q}$ is the ith column of the Hadamard matrix H_{2^d}. We now partially define the target weight vector $\boldsymbol{u}$ by setting $u_i = 1$ and $u_j = 0$ when $j \leqslant 2^d$ and $j \neq i$. By the construction of B_{H}, for $t \leqslant 2^d$ we have $\boldsymbol{u} \cdot \boldsymbol{z}'_t = 0$ and $\boldsymbol{u} \cdot \boldsymbol{z}''_t = 1$ when $q_t = -1$, and $\boldsymbol{u} \cdot \boldsymbol{z}'_t = 1$ and $\boldsymbol{u} \cdot \boldsymbol{z}''_t = 0$ when $q_t = 1$. Therefore, the vector $\boldsymbol{w}_1$ and the disjunction $(\boldsymbol{u}, 1/2)$ differ at trial t on B_{H} whenever $p_t \neq q_t$. By Lemma 8, the adversary can therefore choose a trial sequence with $\boldsymbol{x}_t \in \{\boldsymbol{z}'_t, \boldsymbol{z}''_t\}$ for which the algorithm makes at least 2^{d-1} mistakes at trials $1,\ldots,2^d$.

Thus, the adversary can force 2^{d-1} mistakes in the first 2^d trials by choosing the instances from the sequence B_{H}. This requires fixing in the target $(\boldsymbol{u}, 1/2)$ all the components u_i with $i \leqslant 2^d$, one component to value 1 and the rest to 0. However, the adversary can still choose for each of the remaining $k-1$ components either 0 or 1 completely freely. As in the proof of Theorem 11, this freedom enables the adversary to easily force $k-1$ additional mistakes in the remaining $k-1$ trials, in which the instances are unit vectors. The total number of mistakes is therefore $2^{d-1} + k - 1 = (N + k - 1)/2$. □

By the comments made in Section 2, Theorem 11 gives a lower bound of N mistakes for the ellipsoid algorithm and for the Perceptron algorithm with zero as its initial weight vector. Theorem 12 gives a lower bound of $(N + k - 1)/2$ mistakes for the Perceptron algorithm with arbitrary initial weight vectors. Both of the above lower bounds for the Perceptron algorithm allow the algorithm to use arbitrary thresholds in each trial. In the next section we see that if we assume that the Perceptron algorithm adjusts its threshold in a natural additive manner we can get a sharper lower bound.

4. Perceptron versus other additive algorithms

In this section we first give an adversary strategy which forces the Perceptron algorithm to make $\Omega(kN)$ mistakes. For simplicity we assume that the Perceptron algorithm starts with weight vector zero and uses a constant learning rate. The basic argument of this proof also works for more general versions of the Perceptron algorithm, but the formal proof becomes much more complicated. After presenting the lower bound for the Perceptron algorithm we show how to construct different additive algorithms that make $O(N)$ mistakes when the sample is consistent with a k-literal disjunction.

In the following we assume that $\boldsymbol{w}_t = (w_{t,0}, w_{t,1}, \ldots, w_{t,N})$ is the weight vector of the Perceptron algorithm before trial t. The algorithm receives an instance $\boldsymbol{x}_t = (1, x_{t,1}, \ldots, x_{t,N})$ and predicts $\hat{y}_t = 1$ if $\boldsymbol{w}_t \cdot \boldsymbol{x}_t \geqslant 0$ and $\hat{y}_t = 0$ otherwise. After receiving the correct output y_t the weights are updated as $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta(y_t - \hat{y}_t)\boldsymbol{x}_t$. If $\boldsymbol{w}_1 = \boldsymbol{0}$ then η can be set to 1 without changing the predictions of the algorithm. Note that in this version of the Perceptron algorithm, $w_{t,0}$ can be seen as the threshold used in trial t, and this threshold is also updated additively.

Theorem 13. *Let N be the number of variables and $m = 2k(N-k+1)$. Then there is a trial sequence $((\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m))$ with a monotone k-literal disjunction as target such that the Perceptron algorithm with the zero initial weight vector $\boldsymbol{w}_1 = \boldsymbol{0}$ makes a mistake in every trial.*

Proof. Let $X_1 \vee \cdots \vee X_k$ be the target disjunction. Furthermore let $n = N - k$, so $m = 2k(n+1)$. We partition the m trials into groups $G_1, \ldots, G_k$ of length $2(n+1)$. In each group G_i the weight w_i is learned, so that after the trials in this group, that is at trial $t = 2i(n+1) + 1$, we have

$$w_{t,1} = \cdots = w_{t,i} = n + 1 \quad \text{and} \quad w_{t,0} = w_{t,i+1} = \cdots = w_{t,N} = 0. \tag{6}$$

Observe that condition (6) is satisfied for $i = 0$ before the first trial since $\boldsymbol{w}_1 = \boldsymbol{0}$.

Within a group G_i we choose instances such that $x_j = 0$ holds for all $j = 1, \ldots, k$, $j \neq i$. Thus, X_i is the only relevant variable active during the trials of group G_i. We can therefore disregard the other relevant variables, and it suffices to find a trial sequence of length $2(n+1)$ over $n+1$ variables with X_1 as target such that

(i) the Perceptron algorithm makes a mistake in each trial and

(ii) after all the $2(n+1)$ trials the weights of the Perceptron algorithm satisfy $w_{2(n+1)+1,1} = n + 1$ and $w_{2(n+1)+1,0} = w_{2(n+1)+1,2} = \cdots = w_{2(n+1)+1,n+1} = 0$.

The second condition guarantees that (6) holds after the trials in group G_i.

Such a trial sequence can be constructed as follows. For all trials $t = 1, \ldots, 2(n+1)$, we set $x_{t,1} = 0$ if t is odd and $x_{t,1} = 1$ if t is even. For all t we set $x_{t,2} = \cdots = x_{t,n+1} = 1$. As can be seen by induction, we get $w_{t,1} = (t-1)/2$ and $w_{t,0} = w_{t,2} = \cdots = w_{t,n+1} = 0$ for $t = 1, 3, \ldots, 2(n+1) + 1$, and $w_{t,1} = t/2 - 1$ and $w_{t,0} = w_{t,2} = \cdots = w_{t,n+1} = -1$ for $t = 2, 4, \ldots, 2(n+1)$. Therefore, $\boldsymbol{w}_t \cdot \boldsymbol{x}_t \geqslant 0$ holds for odd and $\boldsymbol{w}_t \cdot \boldsymbol{x}_t < 0$ for even trials t. Hence, the Perceptron algorithm makes a mistake in each trial, which was our first condition. Finally, the second condition is also satisfied since $w_{2(n+1)+1,1} = n + 1$ and $w_{2(n+1)+1,0} = w_{2(n+1)+1,2} = \cdots = w_{2(n+1)+1,n+1} = 0$. This concludes the proof of the theorem. □

We now show how any on-line linear prediction algorithm A can be converted into an additive algorithm A' such that on any trial sequence the number of mistakes made by A' does not exceed the number of mistakes made by A by more than N. Before trial t, the algorithm A' first determines the hypothesis $(\boldsymbol{w}_t, \theta_t)$ the algorithm A would use at trial t. The hypothesis of A' is then chosen to be $(\boldsymbol{q}_t, \theta_t)$ where $\boldsymbol{q}_t$ is the projection of $\boldsymbol{w}_t$ into the span of $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{t-1}\}$. The algorithm A' is by definition additive and uses $\boldsymbol{w}_1 = \boldsymbol{0}$ as a start vector. If at a trial t the predictions of A and A' differ, we have $\boldsymbol{x}_t \cdot \boldsymbol{q}_t \neq \boldsymbol{x}_t \cdot \boldsymbol{w}_t$, and therefore $\boldsymbol{x}_t$ is not in the span of $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{t-1}\}$. Thus, whenever A' makes a mistake but A does not, the dimension of the set $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t\}$ increases by one, and this can happen at most N times.

If we choose the algorithm A in the conversion to be the classical elimination algorithm with mistake bound $N - k$ [19], we obtain an additive algorithm with mistake bound $2N - k$. For small k, a better additive algorithm is obtained by taking A to be Winnow, which yields A' with a mistake bound $N + O(k \log N)$. It remains an open question whether any additive algorithm can exactly match the lower bounds proven in Section 3.

5. Experiments

This section describes some experiments performed on instances drawn from a simple random distribution. The purpose of these experiments is to illustrate that behavior qualitatively similar to that predicted by the worst-case bounds can occur on quite natural, non-adversarial data. We see that even with our random data, Winnow clearly outperforms the Perceptron algorithm when a large majority of the variables are irrelevant. When the proportion of relevant variables is increased, the advantage of Winnow disappears, and when most of the variables are relevant, the Perceptron algorithm performs at least as well as Winnow.

Our input data distributions are parameterized by the number N of variables and the number k of relevant variables. The data is noiseless, i.e., a fixed monotone k-literal disjunction can always predict the outcomes correctly. At each trial we give each input variable the value 0 with probability $2^{-1/k}$ and the value 1 with probability $1 - 2^{-1/k}$. The value given to an input variable is independent of the values of other input variables and the values of the input variable at previous trials. Hence, the probability of a positive instance, i.e., an instance with at least one of the relevant variables set to 1, is $1/2$.

Note that for large values of k, our setting leads to instances in which a very large majority of the variables have value 0. This emphasizes the point that our random data are not meant to simulate any real-world problems. Rather, we use this setting as a simple way of producing for arbitrary ratios k/N instance sequences with roughly equal numbers of positive and negative instances. There is also a theoretical reason that makes this setting interesting. As we have remarked, if at any given trial at most X input variables have value 1, then the Perceptron algorithm has a worst-case mistake bound of $O(kX)$ mistakes. The bounds we know for Winnow do not have such a dependency on X. Instead, they increase as N increases even if X were kept constant. Hence, we expect that combining sparse targets with dense instances and dense targets with sparse instances would most clearly show that Winnow and the Perceptron are incomparable in the sense that depending on the problem, either algorithm could be better.

For both the Perceptron algorithm and Winnow, there are certain parameters the user can specify. Here we provide the algorithms with a fixed threshold θ instead of using the reduction that allows the algorithms to learn the threshold, too. In addition to the threshold, each algorithm needs an initial weight vector $\boldsymbol{w}_1 \in \mathbb{R}^N$ and a learning rate η. By restricting all the initial weights to be equal, i.e., setting $\boldsymbol{w}_1 = (w_1, \ldots, w_1)$ for some $w_1 \in \mathbb{R}$, we end up with having to provide the three real-valued parameters θ, η, and w_1. The algorithms can be quite sensitive to the values of these parameters, and tuning them well based only on the data available to the learner is often quite difficult. To guarantee a reasonable tuning for both algorithms in our comparisons, we have based the tuning on certain additional information that would not be available in an actual learning situation.

For Winnow, we used parameter tunings that lead to known worst-case upper bounds. We chose the tuning used by Auer and Warmuth [1], which at least in some cases leads to better results than Littlestone's original tuning [8]. The parameter values to be used depend on whether $k < N/\mathrm{e}$ holds. (Here e is the base of the natural logarithm.) For $k < N/\mathrm{e}$, we take $\eta = 0.875$, $\theta = 0.441$, and $w_1 = 2N/5$. This guarantees at most

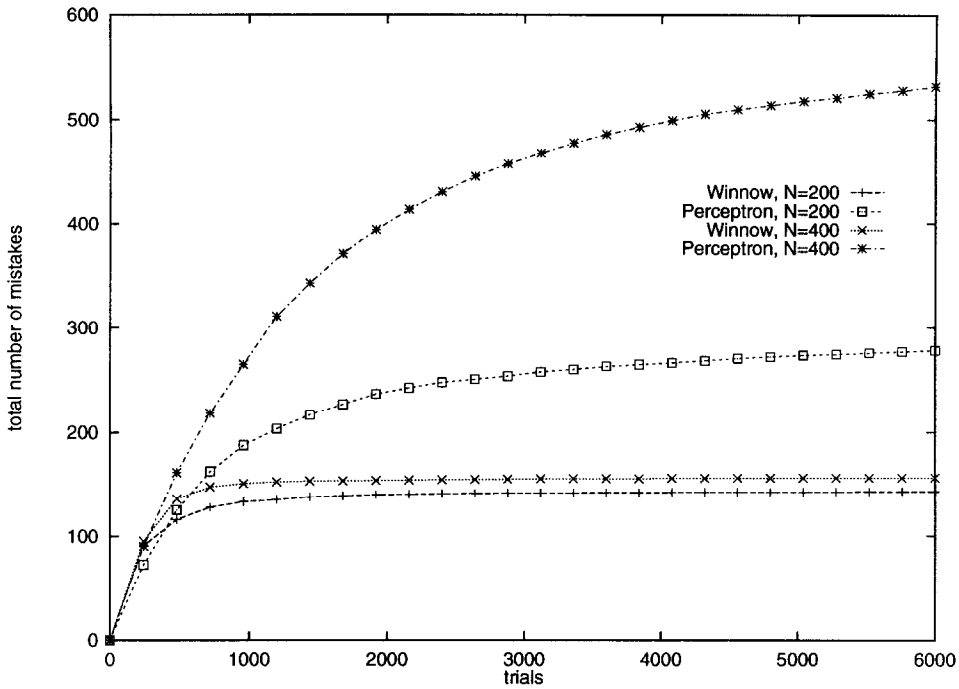

Fig. 1. Cumulative mistake counts for Winnow and the Perceptron algorithm with a monotone 20-literal disjunction as target.

$3.9k \ln N + 3.4A + 1.6$ mistakes [1], where A is the number of attribute errors in the instances (hence $A = 0$ for our experiments). For $k \geqslant N/\mathrm{e}$, we take $\eta = 1$, $\theta = 0.425$, and $w_1 = 0.368$, which guarantees at most $1.37N + 3.72A$ mistakes. Even if one would not know the exact value of k beforehand, choosing between these two sets of parameter values should be significantly easier than searching through the whole three-dimensional parameter space for good values.

For the Perceptron algorithm, we fixed $\eta = 1$, which can be done without loss of generality since multiplying all the parameters by a constant leaves the predictions of the algorithm unchanged. The remaining parameters θ and w_1 were chosen empirically for each individual pair (k, N).

In our first pair of experiments, we considered the value $k = 20$ both with $N = 200$ and with $N = 400$. Fig. 1 shows for both values of N, and both algorithms, how the number of mistakes made in trials $1, \ldots, t$ ("total number of mistakes") increases as t ("trials") increases from 0 to 6000. The curves shown in the figure result from generating for both values of N ten different trial sequences and then for each algorithm averaging the mistake counts from these ten sequences. For the Perceptron algorithm, we used the parameter values $\theta = 5.5$ and $w_1 = 1.1$ for $N = 200$, and $\theta = 8.7$ and $w_1 = 1.2$ for $N = 400$. These values were chosen because they gave the least total number of mistakes over another set of ten trial sequences generated with the same parameters. From Fig. 1

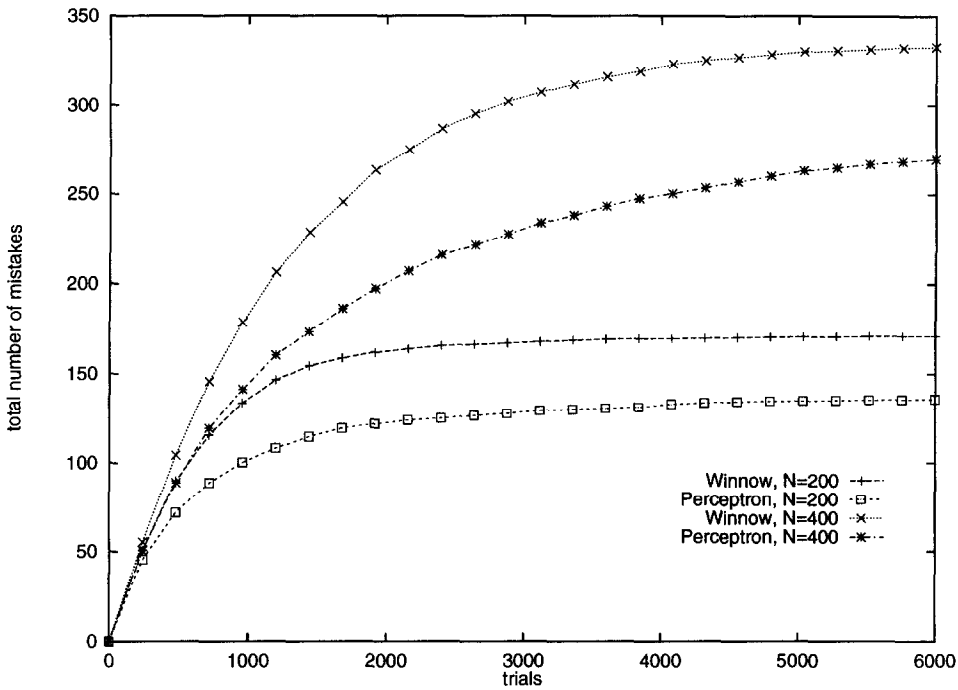

Fig. 2. Cumulative mistake counts for Winnow and the Perceptron algorithm with a monotone $(N/2)$-literal disjunction as target.

we see how doubling the total number N of variables has very little effect on Winnow, when the number k of relevant variables is kept as a small constant. The Perceptron algorithm suffers much more from such a doubling.

In the second pair of experiments, we considered the combinations $k = 100$ and $N = 200$, and $k = 200$ and $N = 400$. Thus, exactly half of the variables were relevant. The results are shown in Fig. 2; the curves have the same meaning as in Fig. 1. For both values of N, the values $\theta = 0.1$ and $w_1 = 1.0$ turned out to give optimal performance for the Perceptron algorithm. Here we see that the Perceptron algorithm actually makes fewer mistakes than Winnow, and both algorithms suffer from doubling the number of variables (and, hence, the number of relevant variables, too).

It should be noted that if the trial sequence happens to be such that in every instance the number of irrelevant variables with value 1 is at most one, then the Perceptron algorithm with parameter values $\eta = 1.0$, $\theta = 0.1$, and $w_1 = 1.0$ actually simulates the classical elimination algorithm for learning monotone disjunctions. The elimination algorithm makes at most $N - k$ mistakes, which for k close to N is a very good bound. With the setting we have, for large values of k it is rare to have two or more irrelevant variables with value 1 at a trial, so in the experiments summarized in Fig. 2 the Perceptron algorithm has performed much like the elimination algorithm would.

Winnow, on the other hand, can easily be tuned to simulate the elimination algorithm exactly (e.g. $\theta = 0.5$, $w_1 = 1.0$, and η very large). This would lead to improved performance in the experiments depicted in Fig. 2, with mistakes counts somewhat lower than those of the Perceptron algorithm. However, the algorithm would then be extremely sensitive to noise. We therefore felt it reasonable to use the parameter values suggested by Auer and Warmuth [1] which would guarantee good performance even if noise were present.

6. Discussion and open problems

We have compared two algorithms, the Perceptron algorithm and Winnow, on the very restricted problem of learning short monotone disjunctions. A worst-case analysis shows that the number of mistakes the Perceptron algorithm makes can be forced to be linear in the total number of variables, even if the number of relevant variables is kept as a small constant. Thus, the bias of the Perceptron algorithm does not allow it to take advantage of the number of relevant variables being small. The lower bounds actually apply to a very general class of *additive algorithms*. In contrast, it is known that the number of mistakes Winnow makes is linear in the number of relevant variables, but only logarithmic in the number of irrelevant variables [8]. Simple experiments show that this effect also occurs outside of our worst-case analysis: On seemingly natural random data, the Perceptron algorithm suffers from additional irrelevant variables much more than Winnow. Another feature of the worst-case bounds, that is to an extent reflected in the experiments, is that the Perceptron algorithm can take advantage of sparse instances: If only few variables are active in the input at any given time but most of the variables are relevant, the Perceptron algorithm outperforms Winnow at least with the reasonable parameter tuning we have used.

The linear lower bounds for additive algorithms obviously extend to any class that contains monotone 1-literal disjunctions. On the other hand, Winnow can learn more general classes than disjunctions, for example r-of-k threshold functions over N variables with $O(kr \log N)$ mistakes [8]. However, when learning general linear classifiers over N binary variables, both Winnow and the Perceptron algorithm can make exponentially many mistakes, while certain methods based on linear programming are guaranteed to make only a polynomial number of mistakes [13].

Even general linear classifiers are really a very restricted concept class. Algorithms that use linear classifiers as their hypotheses can be extended for more general concept classes by introducing as new input variables the values of some nonlinear basis functions. This is especially attractive for algorithms such as Winnow with a mistake bound that is only logarithmic in the number of irrelevant variables. In this case introducing an exponential number of basis function leads only to a linear growth in the mistake bounds, assuming that only very few of the basis functions are relevant. For example, by giving to Winnow as inputs the values of all possible 3^N conjunctions we would get an algorithm that would learn k-term DNF over N variables with $O(kN)$ mistakes. In this particular case, the computational problems caused by the expansion of the input dimensionality seem too

difficult to solve efficiently. However, Winnow, and other multiplicative algorithms [3, 21] with logarithmic mistakes bounds, have been successfully applied when the structure of the basis functions allows simulating an exponential number of input variables and their weights in polynomial time [5, 14, 16]. Extending these results is a promising direction for new theoretical and empirical work.

The fundamental question about any learning algorithm is of course its applicability to real-world problems. This paper is more aimed at understanding what kind of conditions favor either Winnow or the Perceptron algorithm. Finding out how different real-word domains are situated on this scale is an entirely different question that remains open for empirical study. Related to this is the question of how the algorithms tolerate noise. For some empirical studies using artificial noisy data, see Littlestone [11]. Recently Auer and Warmuth [1] have shown how Winnow can be modified to cope with a situation where there is not only noise, but the target is changing over time, as well.

Our lower bound proofs for additive algorithms are not based on a particular ordering of the instances shown to the algorithms. Consider now using the set of instances we used in the on-line lower bound proof for training a batch-style additive algorithm. If one gives any subset of the instances as the training set, then any additive hypothesis would still be wrong on roughly half of the remaining instances. Thus, we see that under a distribution that is uniform over the instances used in the bounds, the sample size required for obtaining a small expected error also grows linearly in N. A similar reasoning leads to linear lower bounds for the PAC model [19]. On the other hand, Winnow with its worst-case upper bounds can be transformed into a batch algorithm that achieves a small expected error and PAC style bounds with sample size that is linear in $k \log N$.

The proofs of the worst-case mistake upper bounds for the Perceptron algorithm [4] and for Winnow [1, 10] can be understood as amortized analysis in terms of a potential function. In the case of the Perceptron algorithm, the potential is based on the Euclidean distance, in the case of Winnow, on an entropic distance measure. The situation is similar also in on-line linear regression [7]. There the Euclidean distance can be used as a potential function for the gradient descent algorithm, which is additive, and the relative entropy can be used for an algorithm called the exponentiated gradient algorithm, which is multiplicative. In the case of regression, it is particularly clear how sparse targets benefit the multiplicative algorithms and sparse instances the additive algorithms, both in worst-case mistakes bounds and in actual behavior on random data. Further, in the regression case the potential functions can be used not only to analyze the algorithms but to actually derive the updates. It is an open problem to devise a framework for deriving updates from the potential functions in the linear classification case.

So far, the evaluation of our algorithms on random data is only experimental. However, it seems possible to obtain closed formulas for the expected total number of mistakes of the Perceptron algorithm in some thermodynamic limit (see, e.g., [17, 22]). We wish to study how these closed formulas relate to the worst-case upper bounds and the adversary lower bounds. Studying this behavior will lead to a deeper understanding of how high dimensionality hurts the Perceptron algorithm and other additive algorithms.

References

[1] P. Auer and M.K. Warmuth, Tracking the best disjunction, in: *Proceedings 36th Symposium on the Foundations of Computer Science*, Milwaukee, WI (IEEE Computer Society Press, Los Alamitos, CA, 1995) 312–321.

[2] A. Blumer, A. Ehrenfeucht, D. Haussler and M.K. Warmuth, Learnability and the Vapnik–Chervonenkis dimension, *J. ACM* 36 (1989) 929–965.

[3] N. Cesa-Bianchi, Y. Freund, D.P. Helmbold, D. Haussler, R.E. Schapire and M.K. Warmuth, How to use expert advice, Report UCSC-CRL-94-33, University of California, Santa Cruz, CA (1994); extended abstract in: *Proceedings 25th Annual ACM Symposium on Theory of Computing* (1993) 382–391.

[4] R.O. Duda and P.E. Hart, *Pattern Classification and Scene Analysis* (Wiley, New York, 1973).

[5] D.P. Helmbold and R.E. Schapire, Predicting nearly as well as the best pruning of a decision tree, in: *Proceedings 8th Annual Conference on Computational Learning Theory* (ACM Press, New York, 1995) 61–68.

[6] L.G. Khachiyan, A polynomial algorithm in linear programming, *Dokl. Akad. Nauk SSSR* 244 (1979) 1093–1096 (in Russian); English translation: *Soviet Math. Dokl.* 20 (1979) 191–194.

[7] J. Kivinen and M.K. Warmuth, Exponentiated gradient versus gradient descent for linear predictors, *Inform. and Comput.* 132 (1997) 1–63.

[8] N. Littlestone, Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm, *Machine Learning* 2 (1988) 285–318.

[9] N. Littlestone, Mistake bounds and logarithmic linear-threshold learning algorithms, Ph.D. Thesis, Report UCSC-CRL-89-11, University of California, Santa Cruz, CA (1989).

[10] N. Littlestone, Redundant noisy attributes, attribute errors and linear threshold learning using Winnow, in: *Proceedings 4th Annual Workshop on Computational Learning Theory* (Morgan Kaufmann, San Mateo, CA, 1991) 147–156.

[11] N. Littlestone, Comparing several linear-threshold learning algorithms on tasks involving superfluous attributes, in: *Proceedings 12th International Conference on Machine Learning*, Lake Tahoe, CA (Morgan Kaufmann, San Francisco, CA, 1995) 353–361.

[12] N. Littlestone, P.M. Long and M.K. Warmuth, On-line learning of linear functions, *J. Comput. Complexity* 5 (1995) 1–23.

[13] W. Maass and G. Turán, How fast can a threshold gate learn, in: *Computational Learning Theory and Natural Learning Systems*, Vol. I (MIT Press, Cambridge, MA, 1994) 381–414.

[14] W. Maass and M.K. Warmuth, Efficient learning with virtual threshold gates, in: *Proceedings 12th International Conference on Machine Learning*, Lake Tahoe, CA (Morgan Kaufmann, San Francisco, 1995) 378–386.

[15] F. Rosenblatt, The Perceptron: a probabilistic model for information storage and organization in the brain, *Psych. Rev.* 65 (1958) 386–407; reprinted in: *Neurocomputing* (MIT Press, Cambridge, MA, 1988).

[16] Y. Singer, Adaptive mixture of probabilistic transducers, in: *Advances in Neural Information Processing Systems*, Vol. 8 (MIT Press, Cambridge, MA, 1996) 381–387.

[17] H. Sompolinsky, H.S. Seung and N. Tishby, Learning curves in large neural networks, in: *Proceedings 4th Annual Workshop on Computational Learning Theory* (Morgan Kaufmann, San Mateo, CA, 1991) 112–127.

[18] P.M. Vaidya, A new algorithm for minimizing convex functions over convex sets, in: *Proceedings 30th Symposium on Foundations of Computer Science* (IEEE Computer Society Press, Los Alamitos, CA, 1989) 338–343.

[19] L.G. Valiant, A theory of the learnable, *Comm. ACM* 27 (1984) 1134–1142.

[20] V.N. Vapnik and A.Y. Chervonenkis, On the uniform convergence of relative frequencies of events to their probabilities, *Theory Probab. Appl.* 16 (1971) 264–280.

[21] V. Vovk, Aggregating strategies, in: *Proceedings 3rd Annual Workshop on Computational Learning Theory* (Morgan Kaufmann, San Mateo, CA, 1990) 371–383.

[22] T.L.H. Watkin, A. Rau and M. Biehl, The statistical mechanics of learning a rule, *Rev. Mod. Phys.* 65 (1993) 499–556.