The Relaxed Online Maximum Margin Algorithm

Yi Li* (y.Li@bristol.ac.uk)
Department of Engineering Mathematics
Queen's Building
University of Bristol
Bristol BS8 1TR
U. K.

Philip M. Long (plong@comp.nus.edu.sg)
Department of Computer Science
National University of Singapore
Singapore 117543, Republic of Singapore

Abstract. We describe a new incremental algorithm for training linear threshold functions: the Relaxed Online Maximum Margin Algorithm, or ROMMA. ROMMA can be viewed as an approximation to the algorithm that repeatedly chooses the hyperplane that classifies previously seen examples correctly with the maximum margin. It is known that such a maximum-margin hypothesis can be computed by minimizing the length of the weight vector subject to a number of linear constraints. ROMMA works by maintaining a relatively simple relaxation of these constraints that can be efficiently updated. We prove a mistake bound for ROMMA that is the same as that proved for the perceptron algorithm. Our analysis implies that the maximum-margin algorithm also satisfies this mistake bound; this is the first worst-case performance guarantee for this algorithm. We describe some experiments using ROMMA and a variant that updates its hypothesis more aggressively as batch algorithms to recognize handwritten digits. The computational complexity and simplicity of these algorithms is similar to that of perceptron algorithm, but their generalization is much better. We show that a batch algorithm based on aggressive ROMMA converges to the fixed threshold SVM hypothesis.

Keywords: Online Learning, Large Margin Classifiers, Perceptrons, Support Vector Machines.

1. Introduction

The perceptron algorithm [38, 39] and the maximum-margin classifier [4] have similar theoretical bases, but different strengths. In the case of linearly separable data, Block [3], Novikoff [33] and Minsky and Papert [31] showed that the number of mistakes made by the perceptron algorithm is upper bounded by a function of the margin, i.e. the minimal distance from any instance to the separating hyperplane. Freund and Schapire [11] generalized this result to the inseparable

* Part of this work was done while this author was pursuing PhD degree at the School of Computing of the National University of Singapore.

case. The maximum-margin algorithm uses quadratic programming to find the weight vector that classifies all the training data correctly and maximizes the margin. Generalization guarantees in terms of the margin have also been given for this algorithm [44, 41, 2, 9].

When comparing these and other algorithms, it is worthwhile to keep in mind the different types of setting where they may be applied. In a batch setting, an algorithm has a fixed collection of examples in hand, and uses them to construct a hypothesis, which is used thereafter for classification without further modification. In an online setting, the algorithm continually modifies its hypothesis as it is being used; it repeatedly receives a pattern, predicts its classification, finds out the correct classification, and possibly updates its hypothesis accordingly. The maximum-margin algorithm is most naturally thought of as a batch algorithm, while the perceptron algorithm is an online algorithm.

An algorithm designed for either of the above settings can be converted to the other. One could use a batch algorithm in an online setting by repeatedly applying the algorithm to all the pattern-classification pairs encountered up to some point in time. The most common way to convert an online algorithm into a batch algorithm is to repeatedly cycle through a dataset, processing the examples one at a time until the algorithm converges in some sense, although a variety of other conversions have been proposed [14, 21, 30, 17].

In batch settings, the maximum-margin algorithm is typically slower than the perceptron algorithm, but generalizes better [11]. On the other hand, the perceptron algorithm is more suitable for online settings.

Both the perceptron algorithm and the maximum-margin algorithm can be applied in conjunction with kernel functions [1, 4] to enable the efficient use of large collections of features that are functions of a problem's raw features. After the patterns are embedded into the expanded feature space, the data is often linearly separable.

In this paper, we design and analyze a new simple online algorithm called ROMMA (the **R**elaxed **O**nline **M**aximum **M**argin **A**lgorithm) for classification using a linear threshold function. ROMMA has similar time complexity to the perceptron algorithm, but its generalization performance in our experiments is much better on average. Moreover, ROMMA can be applied with kernel functions to run efficiently when patterns are embedded in high-dimensional feature spaces in certain ways.

As mentioned above, there are a variety of ways to decide on the best prediction rule given the sequence of different classifiers that an online algorithm such as ROMMA generates [14, 29, 17, 11]. The majority voting method proposed by Freund and Schapire [11], applying the leave-one-out method of Helmbold and Warmuth [17], has the effect

of improving the distribution of margins of the training examples. For a detailed analysis, see [40]. Experiments show that the voted perceptron algorithm has better performance than the standard perceptron algorithm [11]. In this paper, the final prediction vector of ROMMA is used to predict labels of the test set, and how to apply the leave-one-out method to vote different prediction vectors produced by ROMMA is analyzed and discussed in [27].

We conducted experiments similar to those performed by Cortes and Vapnik [8] and Freund and Schapire [11] on the problem of handwritten digit recognition. We tested the standard perceptron algorithm, the voted perceptron algorithm (for details, see [11]) and our new algorithm, using the polynomial kernel function. We found that ROMMA performed better than the standard perceptron algorithm, and an aggressive variant of ROMMA had slightly better performance than the voted perceptron.

In many treatments of algorithms using linear threshold hypotheses, the threshold is fixed at 0 (see [18]). This is often seen to be without loss of generality; an extra feature can be added that always takes the value of -1, and then the weight corresponding to that feature plays the role of a threshold. When the margin of the hypothesis is considered, however, the situation is more complicated, since the margin of the 0-threshold hypothesis might be less than the margin of the corresponding hypothesis with a variable threshold (see subsection 5.2 for the discussion). However, a related reduction was described and discussed in the context of the perceptron algorithm by Cristianini and Shawe-Taylor [9]. A fact implicit in their analysis implies that, for many analyses concerning the margin of linear threshold hypothesis, one can assume a fixed threshold of 0 while losing only a small constant factor; we write this down in Section 4.

The paper is organized as follows. In Section 2, we describe ROMMA in enough detail to determine its predictions, and prove a mistake bound for it. In Section 3, we describe ROMMA in more detail. The observation about reduction to the 0-threshold case is covered in Section 4. In Section 5, we compare the experimental results of ROMMA and an aggressive variant of ROMMA with the perceptron and the voted perceptron algorithms. We also discuss scaling of the features in this section. Some related work [37, 13, 22, 25] and comparisons are discussed in Section 6. We conclude with Section 7.

2. A mistake-bound analysis

2.1. The online algorithms

For concreteness, our analysis will concern the case in which instances (also called patterns) and weight vectors are in $\mathbf{R}^n$ for fixed $n \in \mathbf{N}$, and the ordinary dot product is used, but it is easy to see that our analysis generalizes to arbitrary inner product spaces, and therefore that our results also apply when kernel functions are used.

In the standard online learning model [28], learning proceeds in *trials*. In the tth trial, the algorithm is first presented with an instance $\vec{x}_t \in \mathbf{R}^n$. Next, the algorithm outputs a prediction $\hat{y}_t$ of the classification of $\vec{x}_t$. Finally, the algorithm finds out the correct classification $y_t \in \{-1, 1\}$. If $\hat{y}_t \neq y_t$, then we say that the algorithm makes a mistake. It is worth emphasizing that in this model, when making its prediction for the tth trial, the algorithm only has access to instance-classification pairs for previous trials.

All of the online algorithms that we will consider work by maintaining a weight vector $\vec{w}_t$ which is updated between trials, and predicting $\hat{y}_t = \text{sign}(\vec{w}_t \cdot \vec{x}_t)$, where $\text{sign}(z)$ is 1 if z is positive, -1 if z is negative, and 0 otherwise.[1]

The perceptron algorithm. The perceptron algorithm, due to Rosenblatt [38, 39], starts off with $\vec{w}_1 = \mathbf{0}$. When its prediction differs from the label y_t, it updates its weight vector by $\vec{w}_{t+1} = \vec{w}_t + y_t\vec{x}_t$. If the prediction is correct then the weight vector is not changed.

The next three algorithms that we will consider assume that all of the data seen by the online algorithm is collectively linearly separable, i.e. that there is a weight vector $\vec{u}$ such that for all each trial t, $y_t = \text{sign}(\vec{u} \cdot \vec{x}_t)$. When kernel functions are used, this is often the case in practice.

The ideal online maximum margin algorithm. On each trial t, this algorithm chooses a weight vector $\vec{w}_t$ for which for all previous trials $s < t$, $\text{sign}(\vec{w}_t \cdot \vec{x}_s) = y_s$, and which maximizes the minimum distance of any $\vec{x}_s$ for $s < t$ to the separating hyperplane. It is known [4, 45] that this can be implemented by choosing $\vec{w}_t$ to minimize $\|\vec{w}_t\|$ subject to the constraints that $y_s(\vec{w}_t \cdot \vec{x}_s) \geq 1$ for all $s < t$. These constraints define a convex polyhedron in weight space which we will refer to as P_t.

The relaxed online maximum margin algorithm. This is our new algorithm. The first difference is that trials in which mistakes are

[1] The prediction of 0, which ensures a mistake, is to make the proofs simpler. The usual mistake bound proof for the perceptron algorithm goes through with this change.

not made are ignored. The second difference is in how the algorithm responds to mistakes. The relaxed algorithm starts off like the ideal algorithm. Before the second trial, it sets $\vec{w}_2$ to be the shortest weight vector such that $y_1(\vec{w}_2 \cdot \vec{x}_1) \geq 1$. If there is a mistake on the second trial, it chooses $\vec{w}_3$ as would the ideal algorithm, to be the smallest element of

$$\{\vec{w} : y_1(\vec{w} \cdot \vec{x}_1) \geq 1\} \cap \{\vec{w} : y_2(\vec{w} \cdot \vec{x}_2) \geq 1\}. \tag{1}$$

However, if the third trial is a mistake, then it behaves differently. Instead of choosing $\vec{w}_4$ to be the smallest element of

$$\{\vec{w} : y_1(\vec{w} \cdot \vec{x}_1) \geq 1\} \cap \{\vec{w} : y_2(\vec{w} \cdot \vec{x}_2) \geq 1\} \cap \{\vec{w} : y_3(\vec{w} \cdot \vec{x}_3) \geq 1\},$$

it lets $\vec{w}_4$ be the smallest element of

$$\{\vec{w} : (\vec{w}_3 \cdot \vec{w}) \geq \|\vec{w}_3\|^2\} \cap \{\vec{w} : y_3(\vec{w} \cdot \vec{x}_3) \geq 1\}.$$

This can be thought of as, before the third trial, replacing the polyhedron defined by (1) with the halfspace $\{\vec{w} : (\vec{w}_3 \cdot \vec{w}) \geq \|\vec{w}_3\|^2\}$ (see Figure 1). Note that this halfspace contains the polyhedron of (1); in

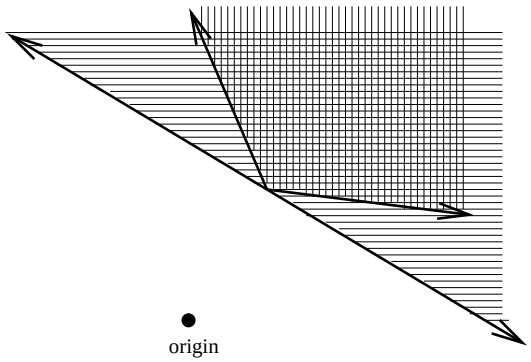

Figure 1. In ROMMA, a convex polyhedron in weight space consisting of all weight vectors satisfying two linear constraints is replaced with the halfspace with the same smallest element.

fact, it contains any convex set whose smallest element is $\vec{w}_3$. Thus, it can be thought of as the least restrictive convex constraint for which the smallest satisfying weight vector is $\vec{w}_3$. Let us call this halfspace H_3. The algorithm continues in this manner. If the tth trial is a mistake, then $\vec{w}_{t+1}$ is chosen to be the smallest element of $H_t \cap \{\vec{w} : y_t(\vec{w} \cdot \vec{x}_t) \geq 1\}$, and H_{t+1} is set to be $\{\vec{w} : (\vec{w}_{t+1} \cdot \vec{w}) \geq \|\vec{w}_{t+1}\|^2\}$. If the tth trial is not a mistake, then $\vec{w}_{t+1} = \vec{w}_t$ and $H_{t+1} = H_t$. We will call H_t the *old constraint*, and $\{\vec{w} : y_t(\vec{w} \cdot \vec{x}_t) \geq 1\}$ the *new constraint*.

Note that after each mistake, this algorithm needs only to solve a quadratic programming problem with two linear constraints. In fact,

there is a simple closed-form expression for $\vec{w}_{t+1}$ as a function of $\vec{w}_t$, $\vec{x}_t$ and y_t that enables it to be computed incrementally using time similar to that of the perceptron algorithm. This is described in Section 3.

The relaxed online maximum margin algorithm with aggressive updating. The algorithm is the same as the previous algorithm, except that an update is made after any trial in which $y_t(\vec{w}_t \cdot \vec{x}_t) < 1$, not just after mistakes.

2.2. Upper bound on the number of mistakes made

Now we prove a bound on the number of mistakes made by ROMMA. As in previous mistake bound proofs (e.g. [30]), we will show that mistakes result in an increase in a "measure of progress", and then appeal to a bound on the total possible progress. Our proof will use the squared length of $\vec{w}_t$ as its measure of progress.

We begin with a property (Lemma 1) which is applicable to both ROMMA and aggressive ROMMA. Although Lemma 2 is applicable only to ROMMA, and our mistake bound analysis can proceed without it, it is useful in deriving our efficient implementation in Section 3.

LEMMA 1. *On any run of ROMMA on linearly separable data, if there was an update after trial t, then the new constraint is binding at the new weight vector, i.e.* $y_t(\vec{w}_{t+1} \cdot \vec{x}_t) = 1$.

Proof: For the purpose of contradiction, suppose the new constraint is not binding at the new weight vector $\vec{w}_{t+1}$. Since $\vec{w}_t$ fails to satisfy this constraint, the line connecting $\vec{w}_{t+1}$ and $\vec{w}_t$ intersects with the border hyperplane of the new constraint, and we denote the intersecting point as $\vec{w}_q$. Then $\vec{w}_q$ can be represented as $\vec{w}_q = \alpha\vec{w}_t + (1-\alpha)\vec{w}_{t+1}$, $0 < \alpha < 1$.

Since the squared Euclidean length $\|\cdot\|^2$ is a convex function, the following holds:

$$\|\vec{w}_q\|^2 \leq \alpha\|\vec{w}_t\|^2 + (1-\alpha)\|\vec{w}_{t+1}\|^2.$$

Note that $\vec{w}_t$ is the unique smallest member of $H_{t-1} \cap \{\vec{w} : y_{t-1}(\vec{w} \cdot \vec{x}_{t-1}) \geq 1\}$ due to the strict convexity of the objective function $\|\cdot\|^2$ [10, 5] and $\vec{w}_{t+1} \neq \vec{w}_t$, we have $\|\vec{w}_t\|^2 < \|\vec{w}_{t+1}\|^2$, which implies

$$\|\vec{w}_q\|^2 < \|\vec{w}_{t+1}\|^2. \tag{2}$$

Since $\vec{w}_t$ and $\vec{w}_{t+1}$ are both in H_t, $\vec{w}_q$ is too, and hence (2) contradicts the definition of $\vec{w}_{t+1}$. □

LEMMA 2. *On any run of ROMMA on linearly separable data, if trial t was a mistake, and not the first one, then the old constraint is binding at the new weight vector, i.e.* $(\vec{w}_{t+1} \cdot \vec{w}_t) = \|\vec{w}_t\|^2$.

Proof: Let A_t be the plane of weight vectors that make the new constraint tight, i.e.

$$A_t = \{\vec{w} : y_t(\vec{w} \cdot \vec{x}_t) = 1\}.$$

By Lemma 1, $\vec{w}_{t+1} \in A_t$. Let $\vec{a}_t = y_t\vec{x}_t/\|\vec{x}_t\|^2$ be the element of A_t that is perpendicular to it. Then each $\vec{w} \in A_t$ satisfies:

$$\|\vec{w}\|^2 = \|\vec{a}_t\|^2 + \|\vec{w} - \vec{a}_t\|^2.$$

Therefore the length of a vector $\vec{w}$ in A_t is minimized when $\vec{w} = \vec{a}_t$ and is monotone in the distance from $\vec{w}$ to $\vec{a}_t$. Thus, if the old constraint is not binding, then $\vec{w}_{t+1} = \vec{a}_t$, since otherwise the solution could be improved by moving $\vec{w}_{t+1}$ a little bit toward $\vec{a}_t$. But the old constraint requires that

$$(\vec{w}_t \cdot \vec{w}_{t+1}) \geq \|\vec{w}_t\|^2,$$

and if $\vec{w}_{t+1} = \vec{a}_t = y_t\vec{x}_t/\|\vec{x}_t\|^2$, this means that

$$(\vec{w}_t \cdot (y_t\vec{x}_t/\|\vec{x}_t\|^2)) \geq \|\vec{w}_t\|^2.$$

Rearranging, we get

$$y_t(\vec{w}_t \cdot \vec{x}_t) \geq \|\vec{x}_t\|^2\|\vec{w}_t\|^2 > 0,$$

($\|x_t\| > 0$ follows from the fact that the data is linearly separable, and $\|w_t\| > 0$ follows from the fact that there was at least one previous mistake). But since trial t was a mistake, $y_t(\vec{w}_t \cdot \vec{x}_t) \leq 0$, a contradiction. □

We proved Lemma 1 and Lemma 2 using a direct, geometrically intuitive argument. In fact they are also consequences of the KKT conditions (see [9]), which was pointed out by anonymous referees.

Now we're ready to prove the mistake bound.

THEOREM 3. *Choose $\ell \in \mathbf{N}$, and a sequence $(\vec{x}_1, y_1), \cdots, (\vec{x}_\ell, y_\ell)$ of pattern-classification pairs in $\mathbf{R}^n \times \{-1, +1\}$. Let $R = \max_t \|\vec{x}_t\|$. If there is a weight vector $\vec{u}$ such that $y_t(\vec{u} \cdot \vec{x}_t) \geq 1$ for all $1 \leq t \leq \ell$, then the number of mistakes made by ROMMA on $(\vec{x}_1, y_1), \cdots, (\vec{x}_\ell, y_\ell)$ is at most $R^2\|\vec{u}\|^2$.*

Proof: First, we claim that for all t, $\vec{u} \in H_t$. This is easily seen since $\vec{u}$ satisfies all the constraints that are ever imposed on a weight vector, and therefore all relaxations of such constraints. Since $\vec{w}_t$ is the smallest element of H_t, we have $\|\vec{w}_t\| \leq \|\vec{u}\|$.

We have $\vec{w}_2 = y_1\vec{x}_1/\|\vec{x}_1\|^2$, and therefore $\|\vec{w}_2\| = 1/\|\vec{x}_1\| \geq 1/R$ which implies $\|\vec{w}_2\|^2 \geq 1/R^2$. We claim that if any trial $t > 1$ is a

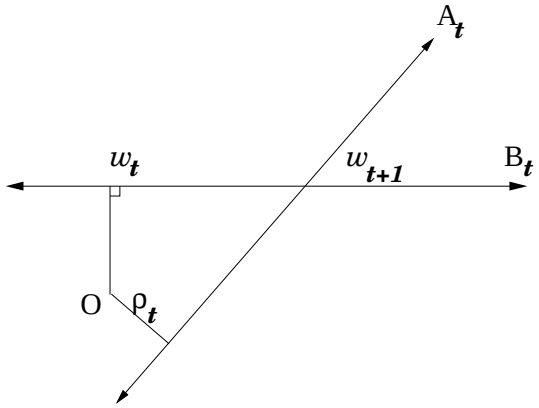

Figure 2. Progress made by ROMMA on trial t.

mistake, then $\|\vec{w}_{t+1}\|^2 \geq \|\vec{w}_t\|^2 + 1/R^2$. This will imply by induction that after M mistakes, the squared length of the algorithm's weight vector is at least M/R^2, which, since all of the algorithm's weight vectors are no longer than $\|\vec{u}\|$, will complete the proof.

Choose an index $t > 1$ of a trial in which a mistake is made. Let

$$A_t = \{\vec{w} : y_t(\vec{w} \cdot \vec{x}_t) = 1\}$$

and

$$B_t = \{\vec{w} : (\vec{w} \cdot \vec{w}_t) = \|\vec{w}_t\|^2\}.$$

By Lemmas 1 and 2, $\vec{w}_{t+1} \in A_t \cap B_t$.

The distance from $\vec{w}_t$ to A_t (call it ρ_t) satisfies

$$\rho_t = \frac{|y_t(\vec{w}_t \cdot \vec{x}_t) - 1|}{\|\vec{x}_t\|} \geq \frac{1}{\|\vec{x}_t\|} \geq \frac{1}{R}, \tag{3}$$

since the fact that there was a mistake in trial t implies $y_t(\vec{x}_t \cdot \vec{w}_t) \leq 0$. As shown in Figure 2, since $\vec{w}_{t+1} \in A_t$,

$$\|w_{t+1} - w_t\| \geq \rho_t. \tag{4}$$

Because $\vec{w}_t$ is the normal vector of B_t and $\vec{w}_{t+1} \in B_t$, we have

$$\|\vec{w}_{t+1}\|^2 = \|\vec{w}_t\|^2 + \|\vec{w}_{t+1} - \vec{w}_t\|^2.$$

Thus, applying (3) and (4), we have

$$\|\vec{w}_{t+1}\|^2 - \|\vec{w}_t\|^2 = \|\vec{w}_{t+1} - \vec{w}_t\|^2 \geq \rho_t^2 \geq 1/R^2,$$

which, as discussed above, completes the proof (where $R = \max_{t=1}^{\ell} \|\vec{x}_t\|$).

□

Since, as is easily proved by induction, for all t, $P_t \subseteq H_t$, we have the following, which complements analyses of the maximum margin algorithm using independence assumptions [4, 45, 41].

COROLLARY 4. *Choose $\ell \in \mathbf{N}$, and a sequence $(\vec{x}_1, y_1), \cdots, (\vec{x}_\ell, y_\ell)$ of pattern-classification pairs in $\mathbf{R}^n \times \{-1, +1\}$. Let $R = \max_t \|\vec{x}_t\|$. If there is a weight vector $\vec{u}$ such that $y_t(\vec{u} \cdot \vec{x}_t) \geq 1$ for all $1 \leq t \leq \ell$, then the number of mistakes made by the ideal online maximum margin algorithm on $(\vec{x}_1, y_1), \cdots, (\vec{x}_\ell, y_\ell)$ is at most $R^2\|\vec{u}\|^2$.*

Next, we turn to an analysis of aggressive ROMMA.

THEOREM 5. *Choose $\delta > 0$, $\ell \in \mathbf{N}$, and a sequence of pattern-classification pairs $(\vec{x}_1, y_1), \cdots, (\vec{x}_\ell, y_\ell)$ from $\mathbf{R}^n \times \{-1, +1\}$. Let $R = \max_t \|\vec{x}_t\|$. If there is a weight vector $\vec{u}$ such that $y_t(\vec{u} \cdot \vec{x}_t) \geq 1$ for all $1 \leq t \leq \ell$, then if $(\vec{x}_1, y_1), \cdots, (\vec{x}_\ell, y_\ell)$ are presented online, the number of trials in which aggressive ROMMA has $y_t(\vec{w}_t \cdot \vec{x}_t) < 1 - \delta$ is at most $R^2\|\vec{u}\|^2/\delta^2$.*

Proof: For positive δ, suppose after trial t of aggressive ROMMA, an update is made and $y_t(\vec{w}_t \cdot \vec{x}_t) < 1 - \delta$. We claim that the progress made is always at least δ^2/R^2, which will complete the proof. Define ρ_t, A_t and B_t as in the proof of Theorem 3.

Lemma 1 still holds for aggressive ROMMA, while Lemma 2 may not hold, i.e. the old constraint may not be binding at the new weight vector $\vec{w}_{t+1}$. (See Figure 3.)

Since $\vec{w}_{t+1} \in A_t$, $\|\vec{w}_t - \vec{w}_{t+1}\| \geq \rho_t$, which implies

$$\|\vec{w}_t - \vec{w}_{t+1}\|^2 \geq \rho_t^2. \tag{5}$$

Since $\vec{w}_{t+1}$ satisfies the old constraint,

$$(\vec{w}_{t+1} \cdot \vec{w}_t) \geq \|\vec{w}_t\|^2 = (\vec{w}_t \cdot \vec{w}_t). \tag{6}$$

Thus

$$\begin{aligned} \|\vec{w}_{t+1}\|^2 &= \|(\vec{w}_{t+1} - \vec{w}_t) + \vec{w}_t\|^2 \\ &= ((\vec{w}_{t+1} - \vec{w}_t) \cdot (\vec{w}_{t+1} - \vec{w}_t)) + 2((\vec{w}_{t+1} - \vec{w}_t) \cdot \vec{w}_t) + (\vec{w}_t \cdot \vec{w}_t) \\ &\geq \rho_t^2 + \|\vec{w}_t\|^2, \end{aligned}$$

by (5) and (6). Since $\rho_t \geq \delta/R$, this completes the proof.

□

For a certain way of converting aggressive ROMMA to a batch algorithm, we can prove that it converges to the (fixed threshold) maximum margin hypothesis. The conversion is as follows: given $(\vec{x}_1, y_1)$, ..., $(\vec{x}_\ell, y_\ell)$,

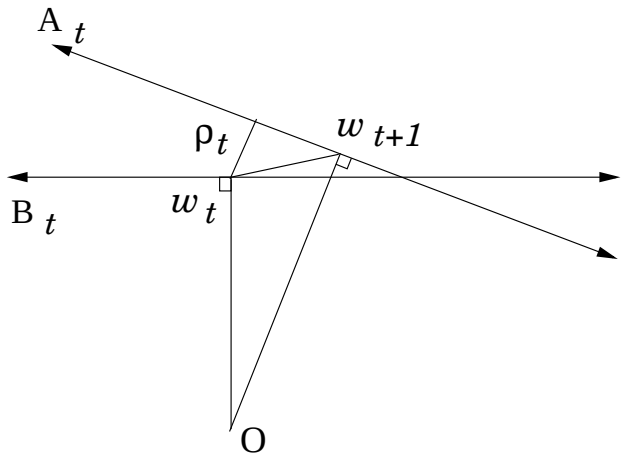

Figure 3. Progress made by aggressive ROMMA on trial t when $\vec{w}_{t+1}$ is not on the border hyperplane of the old constraint.

- initialize ROMMA's hypothesis $\vec{w}$ to $\mathbf{0}$, and
- repeatedly
  - choose $i \in \{1, ..., \ell\}$ to minimize $y_i(\vec{w} \cdot \vec{x}_i)$
  - if $y_i(\vec{w} \cdot \vec{x}_i) < 1$, perform the aggressive ROMMA update on $\vec{x}_i$ and y_i.

Let us call the converted algorithm *Funnel ROMMA*.

THEOREM 6. *Choose $\ell \in \mathbf{N}$, and a sequence of pattern-classification pairs $(\vec{x}_1, y_1), \cdots, (\vec{x}_\ell, y_\ell)$ from $\mathbf{R}^n \times \{-1, +1\}$. Let the (fixed threshold) maximum margin hypothesis for $(\vec{x}_1, y_1), \cdots, (\vec{x}_\ell, y_\ell)$ be the unique weight vector $\vec{u} \in \mathbf{R}^n$ that minimizes $\|\vec{u}\|^2$ subject to the constraints that $y_i(\vec{u} \cdot \vec{x}_i) \geq 1$ for all $1 \leq i \leq \ell$.*

Funnel ROMMA's weight vector approaches that of the fixed threshold maximum margin hypothesis in the limit.

Proof: Choose $\epsilon > 0$. Theorem 5 implies that after some point in time, for all hypothesis weight vectors $\vec{w}$ that Funnel ROMMA produces and all $1 \leq i \leq \ell$, $y_i(\vec{w} \cdot \vec{x}_i) \geq 1 - \frac{\epsilon^2}{2\|\vec{u}\|^2}$. Choose any of Funnel ROMMA's hypotheses after that point in time, and call it $\vec{w}$. We claim that $\|\vec{w} - \vec{u}\| \leq \epsilon$, which will complete the proof.

Let $\delta = \frac{\epsilon^2}{2\|\vec{u}\|^2}$. Note that, for all $1 \leq i \leq \ell$,

$$y_i \left(\frac{\vec{w}}{1 - \delta} \cdot \vec{x}_i \right) \geq 1.$$

Let P be the polytope in weight space consisting of all weight vectors $\vec{v}$ such that for all t, $y_t(\vec{v} \cdot \vec{x}_t) \geq 1$. Since $\vec{u}$ is the shortest weight vector in P, all elements of P are in the halfspace $\{\vec{v} : (\vec{v} \cdot \vec{u}) \geq (\vec{u} \cdot \vec{u})\}$. Thus, in particular, $\left(\frac{\vec{w}}{1-\delta} \cdot \vec{u}\right) \geq (\vec{u} \cdot \vec{u})$, which implies

$$(\vec{w} \cdot \vec{u}) \geq (1 - \delta)(\vec{u} \cdot \vec{u}). \tag{7}$$

On the other hand, the design of (aggressive) ROMMA guarantees that at any time t, $\vec{w}_t$ is the smallest element of H_t, and remember that P is contained in H_t. Thus

$$\|\vec{w}\| \leq \|\vec{u}\|. \tag{8}$$

Now,

$$\begin{aligned} \|\vec{w} - \vec{u}\|^2 &= (\vec{w} \cdot \vec{w}) - 2(\vec{w} \cdot \vec{u}) + (\vec{u} \cdot \vec{u}) \\ &\leq (\vec{w} \cdot \vec{w}) - (1 - 2\delta)(\vec{u} \cdot \vec{u}) \qquad \text{(by (7))} \\ &\leq 2\delta(\vec{u} \cdot \vec{u}) \qquad \text{(by (8))} \\ &\leq \epsilon^2, \end{aligned}$$

completing the proof. □

That $\vec{u}$ in Theorem 6 is unique follows directly from the fact that it minimizes a strictly convex function subject to a convex constraint.

Funnel ROMMA was designed for the convenience of proving convergence. We have not worked out a proof of convergence for more ordinary conversions yet. In subsection 6.1 we use a different conversion, which seems more efficient.

3. An efficient implementation

3.1. Implementation of ROMMA

When the prediction of ROMMA differs from the expected label, according to Lemma 1 and Lemma 2, the algorithm chooses $\vec{w}_{t+1}$ to minimize $\|\vec{w}_{t+1}\|$ subject to $A\vec{w}_{t+1} = b$, where $A = \begin{pmatrix} \vec{w}_t^T \\ \vec{x}_t^T \end{pmatrix}$ and $b = \begin{pmatrix} \|\vec{w}_t\|^2 \\ y_t \end{pmatrix}$. Routine calculation shows that

$$\begin{aligned} \vec{w}_{t+1} &= A^T(AA^T)^{-1}b \\ &= \left(\frac{\|\vec{x}_t\|^2\|\vec{w}_t\|^2 - y_t(\vec{w}_t \cdot \vec{x}_t)}{\|\vec{x}_t\|^2\|\vec{w}_t\|^2 - (\vec{w}_t \cdot \vec{x}_t)^2}\right)\vec{w}_t + \left(\frac{\|\vec{w}_t\|^2(y_t - (\vec{w}_t \cdot \vec{x}_t))}{\|\vec{x}_t\|^2\|\vec{w}_t\|^2 - (\vec{w}_t \cdot \vec{x}_t)^2}\right)\vec{x}_t. \end{aligned} \tag{9}$$

If on trials t in which a mistake is made,

$$c_t = \frac{\|\vec{x}_t\|^2\|\vec{w}_t\|^2 - y_t(\vec{w}_t \cdot \vec{x}_t)}{\|\vec{x}_t\|^2\|\vec{w}_t\|^2 - (\vec{w}_t \cdot \vec{x}_t)^2} \tag{10}$$

and

$$d_t = \frac{\|\vec{w}_t\|^2(y_t - (\vec{w}_t \cdot \vec{x}_t))}{\|\vec{x}_t\|^2\|\vec{w}_t\|^2 - (\vec{w}_t \cdot \vec{x}_t)^2}, \tag{11}$$

and on other trials $c_t = 1$ and $d_t = 0$, then always

$$\vec{w}_{t+1} = c_t\vec{w}_t + d_t\vec{x}_t, \tag{12}$$

and

$$\|\vec{w}_{t+1}\|^2 = c_t^2\|\vec{w}_t\|^2 + 2c_td_t(\vec{w}_t \cdot \vec{x}_t) + d_t^2\|\vec{x}_t\|^2.$$

Note that, due to Lemmas 1 and 2, the denominators in (9) will never be zero.

Since the computations required by ROMMA involve inner products together with a few operations on scalars, we can apply the kernel method to our algorithm, efficiently solving the original problem in a very high dimensional space. Computationally, we only need to modify the algorithm by replacing each inner product computation $(\vec{x}_i \cdot \vec{x}_j)$ with a kernel function computation $\mathcal{K}(\vec{x}_i, \vec{x}_j)$.

To make a prediction for the tth trial, the algorithm must compute the inner product between $\vec{x}_t$ and prediction vector $\vec{w}_t$. In order to apply the kernel function, as in [4, 11], we store each prediction vector $\vec{w}_t$ in an implicit manner, as the weighted sum of examples on which mistakes occur during the training. In particular, each $\vec{w}_t$ is represented as

$$\vec{w}_t = \left(\prod_{j=1}^{t-1} c_j\right)\vec{w}_1 + \sum_{j=1}^{t-1}\left(\prod_{n=j+1}^{t-1} c_n\right) d_j\vec{x}_j, \tag{13}$$

where $\vec{w}_1$ is the initial weight vector (see subsection 5.2 on how to set the initial weight vector in our experiments). If we let

$$\alpha_0 = \prod_{j=1}^{t-1} c_j \tag{14}$$

and

$$\alpha_j = \left(\prod_{n=j+1}^{t-1} c_n\right) d_j,\ 1 \le j \le t-1, \tag{15}$$

then (13) can be written as

$$\vec{w}_t = \alpha_0\vec{w}_1 + \sum_{j=1}^{t-1}\alpha_j\vec{x}_j.$$

Formula (13) may seem daunting; however, making use of the recurrence $(\vec{w}_{t+1} \cdot \vec{x}) = c_t(\vec{w}_t \cdot \vec{x}) + d_t(\vec{x}_t \cdot \vec{x})$, it is obvious that the complexity of ROMMA is similar to that of the perceptron algorithm when kernel functions are applied (in the update of the perceptron algorithm, $c_t = 1$, $d_t = y_t$; however the kernel computation is the main computational cost, and making a prediction for tth trial involves p kernel function evaluations if p updates have been made so far). This was born out by our experiments in Section 5.

3.2. Implementation of aggressive ROMMA

Suppose an update is needed on trial t in aggressive ROMMA, i.e. $y_t(\vec{w}_t \cdot \vec{x}_t) < 1$, then the new constraint is binding at the new weight vector $\vec{w}_{t+1}$ by Lemma 1, but the old constraint may not be binding at $\vec{w}_{t+1}$ (see the proof of Lemma 2 for the reason). If the old constraint is not binding at the new weight vector $\vec{w}_{t+1}$, $\vec{w}_{t+1}$ is the smallest vector satisfying only the new constraint, i.e. $\vec{w}_{t+1} = \frac{y_t \vec{x}_t}{\|\vec{x}_t\|^2}$; otherwise $\vec{w}_{t+1}$ is calculated as in ROMMA. The old constraint is not binding at the new weight vector $\vec{w}_{t+1}$ and satisfied by it if and only if

$$\begin{aligned} & (\vec{w}_t \cdot \vec{w}_{t+1}) \geq \|\vec{w}_t\|^2 \\ \text{i.e.} \quad & y_t(\vec{w}_t \cdot \vec{x}_t) \geq \|\vec{x}_t\|^2 \|\vec{w}_t\|^2. \end{aligned}$$

So we get the following implementation of aggressive ROMMA:

$$\begin{aligned} \text{If } 1 > y_t(\vec{w}_t \cdot \vec{x}_t) \geq \|\vec{x}_t\|^2 \|\vec{w}_t\|^2 \text{ then} \quad & \vec{w}_{t+1} = \tfrac{y_t \vec{x}_t}{\|\vec{x}_t\|^2}, \\ \text{otherwise} \quad & \vec{w}_{t+1} = c_t \vec{w}_t + d_t \vec{x}_t, \end{aligned}$$

where c_t and d_t are expressed in (10) and (11) respectively.

3.3. Two other points for improving efficiency

From (9) and (12) we can find that the dominant computation in our algorithm is inner products between pairs of instances. Even though inner product computation may be replaced with a kernel function, the inner product is still the main factor of speed since kernel functions usually involve inner product computation.

Our inner product was implemented as the following pseudocodes:

```
InnerProduct(x1, x2, size)
  sum:= 0;
  for (i=0; i<size; i++)
      if ((x1[i] ≠ 0)  &&  (x2[i] ≠ 0))
          sum:= sum + x1[i] × x2[i];
```

This implementation makes use of sparseness of input since the operation of comparisons is much faster than the operation of any kind of multiplication for digital computers. We believe that our sparse inner product code has a similar effect in exploiting sparseness to storing input vectors as sparse vectors [37].

The other efficiency comes from the recurrence of update rule (12) in our new algorithm. In this subsection we assume that the subscript of the weight vector does not change when no update is made and that we have enough memory to store the estimated outputs for every training example. Suppose the weight vector just before going through $\vec{x}_i$ is $\vec{w}_T$, then the estimated output for $\vec{x}_i$ is $(\vec{w}_T \cdot \vec{x}_i)$. Provided that the weight vector is $\vec{w}_{T+k}$ when $\vec{x}_i$ is to be gone through again, then the present estimated output for $\vec{x}_i$ is

$$
\begin{aligned}
&(\vec{w}_{T+k} \cdot \vec{x}_i) \\
&= \left(\prod_{j=T}^{T+k-1} c_j \right) (\vec{w}_T \cdot \vec{x}_i) + \sum_{j=T}^{T+k-1} \left(\prod_{n=j+1}^{T+k-1} c_n \right) d_j \, (\vec{x}_{I[j]} \cdot \vec{x}_i),
\end{aligned}
$$

where $I[\ ]$ is an array, whose jth element represents the index of the training example on which jth update is made. In other words, the estimated output $(\vec{w} \cdot \vec{x})$ for training example $\vec{x}$ can make use of intermediate results computed before as long as it exists.

4. Reduction to the 0 threshold case

Throughout this paper we restrict our attention to algorithms using hypotheses with a threshold of 0. The following motivates this choice. One can apply it for example to get analogues of Theorem 3, Corollary 4, and Theorem 5 for the variable threshold case that are a constant factor worse than the original bounds; similar facts hold concerning PAC-style generalization guarantees for the maximum-margin algorithm [41, 2, 9].

PROPOSITION 7. ([9]). *Let $(\vec{x}_1, y_1), ..., (\vec{x}_\ell, y_\ell)$ be any sequence of pattern-classification pairs in $\mathbf{R}^n \times \{-1, +1\}$, and let $R = \max_i \|\vec{x}_i\|$. Suppose $\vec{u} \in \mathbf{R}^n$ and $\theta \in \mathbf{R}$ have the property that for all $t \in \{1, ..., \ell\}$, $y_t(\vec{u} \cdot \vec{x}_t - \theta) \geq 1$. For each i, form $\vec{x}'_i \in \mathbf{R}^{n+1}$ by concatenating an additional component with value $-R$ to $\vec{x}_i$. Then there is a weight vector $\vec{u}' \in \mathbf{R}^{n+1}$ such that for all $t \in \{1, ..., \ell\}$, $y_t(\vec{u}' \cdot \vec{x}'_t) \geq 1$, and $\|\vec{u}'\|^2 \leq 2\|\vec{u}\|^2$. (For all t, $\|\vec{x}'_t\|^2 \leq 2R^2$.)*

5. Experiments

We did some experiments using the perceptron algorithm, ROMMA and aggressive ROMMA as batch algorithms on the MNIST OCR database. [2] LeCun et al. [26] have published a detailed comparison of the performance of some of the best algorithms on this dataset. The best test error rate they achieved is 0.7%, through boosting on top of the neural net LeNet4, which was crafted through a series of experiments in architecture, combined with an analysis of the characteristics of recognition errors. A version of the optimal margin classifier (Soft Margin SVMs) [8] achieves a test error rate of 1.1%.

5.1. Experimental settings

Every example in this database has two parts, the first is a 28×28 matrix which represents the image of the corresponding digit. Each entry in the matrix takes on values from $\{0, \cdots, 255\}$. The second part is a label taking on values from $\{0, \cdots, 9\}$. The dataset consists of $60,000$ training examples and $10,000$ test examples.

To cope with multiclass data, we trained the perceptron algorithm, ROMMA or aggressive ROMMA once for each of the 10 labels. When training on class $l \in \{0, \cdots, 9\}$, we replaced each labeled instance $(\vec{x}_i, y_i)$ by the binary-labeled instance $(\vec{x}_i, b_i)$, where $b_i = +1$ if $y_i = l$, otherwise $b_i = -1$. Classification of a test pattern is done according to the maximum output of these ten classifiers. There are some other ways to combine many two-class classifiers into a multiclass classifier [36, 12, 24].

To produce output given a test instance $\vec{x}$, besides using the final hypothesis, we also tried the "voting" method to convert the standard perceptron algorithm to a batch learning. The "voting" method is adopted in [11] and is an application of the general leave-one-out method of [17]. It records the number of trials each prediction vector survives during the training, which is denoted $\mathrm{sur}_{l,i}$ in the following, where l represents that the classifier is for label l, i is the index of the prediction vector. If k_l prediction vectors are produced during the training, the output generated by the voting method is:

$$\sum_{i=1}^{k_l} \mathrm{sur}_{l,i} * \mathrm{sign}(\vec{w}_{l,i} \cdot \vec{x}).$$

We obtained a batch algorithm from our new online algorithm in the usual way, making a number of passes over the dataset and using

[2] National Institute for Standards and Technology, special database 3. See http://www.research.att.com/~yann/ocr for information on obtaining this dataset.

the final weight vector to classify the test data because the above voting method sometimes hurts our new online algorithm. How to apply the general leave-one-out method and vote different prediction vectors produced by (aggressive) ROMMA is analyzed and discussed in [27].

5.2. Kernel functions and bias

In the experiments we adopt the following polynomial kernel function

$$\mathcal{K}(\vec{x}_i, \vec{x}_j) = (1 + (\vec{x}_i \cdot \vec{x}_j))^d. \tag{16}$$

This kernel function corresponds to using an expanded collection of features including all products of at most d components of the original feature vector (see [45]). Let us refer to the mapping from the original feature vector (say in $\mathbf{R}^n$) to the expanded feature vector (say in $\mathbf{R}^p$) as Φ. Note that one component of $\Phi(\vec{x})$ is always 1, and without loss of generality, we take it as the first component of $\Phi(\vec{x})$. Therefore the first component of the weight vector can be viewed as a bias. In all our experiments, we set the initial weight vector $\vec{w}_1 = \Phi(\vec{0})$ rather than $\vec{0}$ to speed up the learning of the coefficient corresponding to the bias. Hence

$$bias = \alpha_0 + \sum_{i=1}^{t-1} \alpha_i = \prod_{j=1}^{t-1} c_j + \sum_{i=1}^{t-1} \left(\prod_{n=i+1}^{t-1} c_n \right) d_i \tag{17}$$

according to (14) and (15), where c_j is always positive, and d_j may be positive or negative. Note that α_0 is the coefficient of $\vec{w}_1$ and that in the standard perceptron algorithm, $c_j = 1$, $d_j = y_j$ for those training examples on which mistakes were made, hence α_0 is always 1, so whether $\vec{w}_1 = \Phi(\vec{0})$ or $\vec{w}_1 = \vec{0}$ makes little difference for the perceptron algorithm.

With the help of the polynomial kernel function in (16), ROMMA and aggressive ROMMA make predictions by the linear functions with implicit bias, however, the solution which aggressive ROMMA converges to when repeatedly cycling through the examples may be different from the solution which support vector machines find, because in the expanded high dimensional feature space, the task of support vector machines is to

$$\begin{array}{ll} \min & \frac{1}{2}\|\vec{w}\|^2 = \frac{1}{2}\sum_{i=2}^{p} w_i^2 \\ \text{subject to} & y_j[(\vec{w} \cdot \Phi(\vec{x}_j)) + b] \geq 1 \qquad j = 1 \ldots \ell, \end{array}$$

where ℓ is the number of training examples. The above formula for the squared length of $\vec{w}$ is due to that for support vector machines adopting this kernel function, $w_1 = \sum_{j=1}^{\ell} y_j \alpha_j = 0$, where $\alpha_j \geq 0$ are Lagrange multipliers (see Section 6 for the Wolfe dual problem). The task of aggressive ROMMA is to

$$\begin{array}{ll} \min & \frac{1}{2}\|\vec{w}\|^2 = \frac{1}{2}\left(w_1^2 + \sum_{i=2}^{p} w_i^2\right) \\ \text{subject to} & y_j(\vec{w}\cdot\Phi(\vec{x}_j)) \geq 1 \qquad j = 1 \ldots \ell. \end{array}$$

There exists some simple problems where $w_1 = \sum_i \alpha_i$ and $\sum_{i=2}^{p} w_i^2$ which aggressive ROMMA converges to coincide with b and $\sum_{i=2}^{p} w_i^2$ which support vector machines obtain. However, we haven't identified yet the conditions on which the above coincidence occurs or does not occur. Intuitively there should be some problems where the coincidence does not occur, for example, SVM solution favors large bias, but aggressive ROMMA solution favors small or moderate bias due to the objective function $\frac{1}{2}\left(bias^2 + \sum_{i=2}^{p} w_i^2\right)$.

Note that the reduction to the 0 threshold case in Proposition 7 involved adding a component with value $-R$ (R is the length of the longest feature vector), while using the polynomial kernel gives us a component with value 1. By rescaling the components of $\vec{x}$, we can make the size of the other components small relative to the constant component, with a similar effect. Scaling of features is discussed further in the following subsection.

5.3. Scaling and its possible explanations

As every entry in the image matrix takes value from $\{0, \cdots, 255\}$, the order of magnitude of $\mathcal{K}(\vec{x}, \vec{x})$ is about 10^{26}. For ROMMA and aggressive ROMMA, c_j has the order of magnitude of 1, d_j has the order of magnitude of 10^{-26}, which might cause round-off error in the computation of $(\vec{w}_t \cdot \Phi(\vec{x})) = c_{t-1}(\vec{w}_{t-1} \cdot \Phi(\vec{x})) + d_{t-1}\mathcal{K}(\vec{x}, \vec{x}_{t-1})$. We scale the data by dividing each entry with 1100 when training with ROMMA or aggressive ROMMA. It is obvious that there is little excess in computation if scaling is implemented in the process of dot products.

The advantage brought by the scaling factor of 1100 to ROMMA and aggressive ROMMA is that both α_0 and $\sum_{i=1}^{t-1} \alpha_i$ in (17) now have the order of magnitude of 1 instead of α_0 having the order of magnitude of 1 and $\sum_{i=1}^{t-1} \alpha_i$ having the order of magnitude of 10^{-22} when there is no scaling. Taking into account that c_j is always positive, and d_j may be positive or negative, it's important to force α_0 and $\sum_{i=1}^{t-1} \alpha_i$ to have the same order of magnitude. Note that the standard perceptron algorithm does not have this problem (see subsection 5.2 for the reason). The scaling factor of 255 has a similar but weaker such effect on the bias.

Scaling may play other roles. In the point of view of kernel methods, scaling is an operator from the expanded feature space to itself, and the entropy number of this operator which serves as capacity control may be minimized over the different choices of scaling factors on the corresponding components of the high dimensional feature space [16, 48, 7].

It remains open to obtain the optimal scaling operator for polynomial kernel functions that can be implemented efficiently due to the facts that it is very hard to calculate the eigenvalues of polynomial kernel functions (recently Smola et al. gave a method to calculate the eigenvalues of analytic kernel functions on the domain of unit sphere, but for the domain of unit ball, it is very technical [42]) and that kernel functions only transform the original input space to high dimensional feature space implicitly. In the experiment we only tried no scaling, a scaling factor of 255, and a scaling factor of 1100 for the perceptron algorithm, ROMMA and aggressive ROMMA, and presented their best results, i.e. the results of the perceptron algorithm with a scaling vector of 255, the results of ROMMA and aggressive ROMMA with a scaling vector of 1100.

5.4. Input noise

To deal with data which are still linearly inseparable in the high dimensional feature space, and/or to improve generalization by trade-off between empirical estimates and confidence interval, Friess et al. [13] suggested the use of quadratic slack penalty in the cost function (i.e. minimize $\frac{1}{2}\|w\|^2 + \frac{1}{2\lambda}\sum_i \xi_i^2$), which can be implemented using a slightly different kernel function [13, 22, 23]:

$$\tilde{\mathcal{K}}(x_k, x_j) = \mathcal{K}(x_k, x_j) + \delta_{kj}\lambda, \tag{18}$$

where δ_{kj} is the Kronecker delta function, λ is a predefined parameter. We use this method to deal with noise in our experiments.

The derivation of (18) can be obtained by investigating the close relationship between Gaussian Processes (GP) and SVMs [34, 46, 47]. We give a brief description of it in the following for completeness. Given arbitrary points $x_1, \cdots, x_\ell$ of the input space, suppose the set of function values

$$\vec{f} = (f(x_1), \ldots, f(x_\ell))^T$$

have a joint Gaussian prior distribution:

$$p(\vec{f}) = \frac{1}{\sqrt{(2\pi)^\ell \det K}} e^{-\frac{1}{2}(\vec{f}-\vec{m})^T K^{-1}(\vec{f}-\vec{m})},$$

where $\vec{m} = (m(x_1), \cdots, m(x_\ell))^T$ is the mean and

$$K = E[\vec{f}\vec{f}^T] - \vec{m}\vec{m}^T$$

is the covariance matrix having elements

$$K(x_i, x_j), \qquad i,\ j \in \{1, \cdots, \ell\}.$$

For SVMs, the covariance function $K(x_i, x_j)$ is completely equivalent to the kernel function.

Input noise $\vec{\xi}$ is defined as a random vector added to the function value vector $\vec{f}$. The simplest case for $\vec{\xi}$ is the independent Gaussian noise with zero mean and variance matrix λI, where I is the identity matrix. We use the fact that the process $\vec{f}+\vec{\xi}$ – due to the Gaussianity of the noise – is also a Gaussian process with the following covariance matrix:

$$\tilde{K} = E[(\vec{f}+\vec{\xi})(\vec{f}+\vec{\xi})^T] - E[(\vec{f}+\vec{\xi})]E[(\vec{f}+\vec{\xi})^T] = K + \lambda I.$$

Based on the equivalence between the covariance matrix and the matrix composed of kernel functions in SVMs, (18) was obtained.

If the input noise follows Laplace noise $p(\xi) = \frac{C}{2}exp(-C|\xi|)$, we obtain the linear slack penalty in the cost function (i.e. minimizing $\frac{1}{2}\|w\|^2 + C\sum_i \xi_i$), which the soft margin support vector machine in [8, 45] aims to optimize.

5.5. Experimental results and their discussions

Since the performance of online learning algorithm is affected by the order of sample sequence, all the results shown in the tables in the paper average over 10 random permutations of sample sequence. We present results for a batch setting until four epochs (the algorithm goes through all the instances once in one epoch and T in the tables represents the number of epochs).

We conducted three groups of experiments, one for the perceptron algorithm (denoted "percep") and voted perceptron (denoted "vote-percep") whose detailed description is in [11], the second for ROMMA, the third for aggressive ROMMA (denoted "agg-R").

Mistake numbers are the total number of mistakes made during the training for the 10 labels. Note that the mistake number of the voted perceptron algorithm is the same as that of the standard perceptron algorithm since the voting procedure occurs at the predictions of test set, not in the training process. In aggressive ROMMA, an update is made during training whenever $y(\vec{w}\cdot\vec{x}) < 1$, hence the update/correction number is larger than the mistake number, while for the perceptron algorithm and ROMMA, they are the same. Comparison of mistake numbers gives some idea of the relative practical utility of the algorithms in online settings. The number of updates during training is the dominant factor of speed for all three groups according to (9) and (13), which was verified by our observations.

Data in the first group are scaled with 255, data in the last two groups are scaled with 1100 (see subsection 5.3 for the reasons of

scaling of the features). Note that these choices are the best for the corresponding groups among three alternatives (i.e. no scaling, scaling factor of 255, and of 1100). All three groups set the initial weight vector $\vec{w}_1$ to $\Phi(\vec{0})$ (see subsection 5.2 for the reason).

We first set the degree of the polynomial kernel function in (19) to 4 since in experiments on the same problem conducted in [11], the best results occur with this value. Test error rates (in percentage), mistake numbers and update numbers for $d = 4$ are shown in Table I.

Table I. Experimental results on MNIST data for $d = 4$

	$T =$	1	2	3	4
$d = 4$	percep(scale=255)	2.71	2.14	2.03	1.85
	voted-percep	2.23	1.86	1.76	1.71
	Mistake No.	7901	10421	11787	12637
	ROMMA (scale=1100)	2.48	1.96	1.79	1.77
	Mistake No.	7963	9995	10971	11547
	agg-R (scale=1100)	2.14	1.82	1.71	1.67
	Mistake No.	6077	7391	7901	8139
	Update No.	26802	39642	51853	64005

The test error rates in Table I demonstrate that ROMMA has better performance, and aggressive ROMMA has much better performance than the standard perceptron algorithm. Aggressive ROMMA has slightly better performance than voted perceptron. The update numbers in Table I indicate that the perceptron algorithm and ROMMA has similar training time, while aggressive ROMMA has much longer training time, which coincide with our observations (on average, training for the perceptron algorithm takes 55 minutes, for ROMMA 51 minutes, and for aggressive ROMMA 4.5 hours, for four epochs on a PVM cluster of four Pentium II 400MHz PCs). Aggressive ROMMA has unpleasant ratio of performance/training time in the current conversion from online learning to batch learning, we described and implemented a simple efficient conversion to speed up aggressive ROMMA in subsection 6.1.

Since the standard perceptron algorithm and voted perceptron in [11] have similar performance with $d = 4$ and $d = 5$, and ROMMA and aggressive ROMMA regularize more than the perceptron algorithm, we tried higher degrees than 4 for all three groups, and found that only $d = 5$ had some interesting results (degrees higher than 5 have

statistically significantly worse test error rates), which was shown in Table II.

Table II. Experimental results on MNIST data for $d = 5$

	$T =$	1	2	3	4
$d = 5$	percep(scale=255)	2.85	2.20	1.97	1.90
	voted-percep	2.28	1.95	1.82	1.78
	Mistake No.	7912	10290	11507	12176
	ROMMA (scale=1100)	2.25	1.83	1.74	1.66
	Mistake No.	8076	9734	10437	10777
	agg-R (scale=1100)	2.10	1.78	1.73	1.71
	Mistake No.	6465	7511	7820	7939
	Update No.	24134	33774	42388	50437

The error rates and update numbers in Table II show that $d = 4$ provides better test error rate and similar training time for the standard perceptron algorithm; $d = 5$ produces better performance and similar training time for ROMMA, while $d = 5$ has similar performance, but shorter training time for aggressive ROMMA. Both ROMMA and aggressive ROMMA regularize more than the standard perceptron algorithm, hence $d = 5$ is expected to have some advantage over $d = 4$ for (aggressive) ROMMA, however it remains open to explain why $d = 5$ provides better performance for ROMMA, but better training time for aggressive ROMMA.

We control noise via modified kernel function

$$\tilde{\mathcal{K}}(x_k, x_j) = \mathcal{K}(x_k, x_j) + \delta_{kj}\lambda \tag{19}$$

introduced in subsection 5.4 to deal with data which are still linearly inseparable in the high dimensional feature space. The parameter λ was decided by a small (1000 training instances and 500 test instances) validation set extracted from MNIST data in our experiments. We found that for a scaling factor of 255, λ should take the value of 5×10^5, and that for a scaling factor of 1100, λ should take the value of 30. The test error rates (in percentage), mistake numbers and update numbers for the perceptron algorithm with scale= 255 and $d = 4$, those for ROMMA and aggressive ROMMA with scale= 1100 and $d = 5$ are shown in Table III ($d = 4$ is a better choice for the perceptron algorithm, and $d = 5$ a better choice for ROMMA and aggressive ROMMA based on Tables I and II).

Table III. Experimental results with noise control on MNIST data

	$T=$	1	2	3	4
$d=4$	percep	2.71	2.13	2.00	1.93
scale=255	voted-percep	2.23	1.86	1.76	1.73
$\lambda = 5 \times 10^5$	Mistake No.	7901	10417	11748	12548
$d=5$	ROMMA	2.20	1.83	1.72	1.70
scale=1100	Mistake No.	8036	9598	10252	10580
$\lambda = 30$	agg-R	2.05	1.75	1.68	1.67
	Mistake No.	6336	7295	7563	7660
	Update No.	25316	35328	44268	52565

Comparing the results in Table III with those in Table I and Table II, we find that the method of noise control by introducing a quadratic slack penalty in the objective function is effective only to ROMMA and aggressive ROMMA, make little difference to the perceptron algorithm. The possible reason for this may be that (19) was derived by investigating the close relationship between Gaussian Processes (GP) and SVMs (see subsection 5.4), and (aggressive) ROMMA is close to SVMs with fixed threshold, while the perceptron algorithm is rather far away on this data set. Another explanation might be that the parameter λ we set for the perceptron algorithm is not good enough.

6. Related work

Support Vector Machines (SVMs) [8] manifest an impressive resistance to overfitting, which can be explained by the small effective VC dimension [41]. Their training is performed by minimizing the length of the weight vector subject to a number of linear constraints when the data can be perfectly separated in the feature space. Namely, the primal problem is to minimize the objective function

$$\frac{1}{2}\|\vec{w}\|^2 \tag{20}$$

subject to

$$y_i[(\vec{w}\cdot\vec{x}_i)+b] \geq 1 \qquad\qquad i=1,\cdots,\ell, \tag{21}$$

where ℓ is the number of training examples.

The Wolfe dual of the above QP (Quadratic Programming) problem (20) and (21) is to maximize the Lagrangian

$$\sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j (\vec{x}_i \cdot \vec{x}_j) \tag{22}$$

subject to

$$\alpha_i \geq 0 \qquad\qquad i = 1, \cdots, \ell$$
$$\sum_i \alpha_i y_i = 0.$$

The $\{\alpha_i\}$ are called Lagrange multipliers, and $\vec{w} = \sum_i \alpha_i y_i \vec{x}_i$. Those $\vec{x}_i$'s corresponding to nonzero Lagrange multipliers are called support vectors. $b = -\frac{1}{2}[(\vec{w} \cdot \vec{x}_+) + (\vec{w} \cdot \vec{x}_-)]$, where we denote by $\vec{x}_+$ any positive support vector and by $\vec{x}_-$ any negative support vector.

The above SVM solving linearly separable data in the feature space is called the hard margin SVM. To construct the optimum margin hyperplane when the data are linearly nonseparable, nonnegative slack variables $\xi_i \geq 0$ are introduced and the primal problem is to minimize

$$\frac{1}{2}\|\vec{w}\|^2 + C\left(\sum_{i=1}^{\ell} \xi_i^\sigma\right) \tag{23}$$

subject to

$$y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 - \xi_i \qquad\qquad i = 1, \cdots, \ell, \tag{24}$$

where $C > 0$ and σ are given values. The above SVM (23) and (24) dealing with linearly nonseparable data in the feature space is called the soft-margin SVM.

To simplify computations, let $\sigma = 1$, then the corresponding dual problem is to maximize the Lagrangian

$$\sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j (\vec{x}_i \cdot \vec{x}_j) \tag{25}$$

under slightly different constraints:

$$0 \leq \alpha_i \leq C \qquad\qquad i = 1, \cdots, l \tag{26}$$

$$\sum_i \alpha_i y_i = 0. \tag{27}$$

In practice, σ can also be set to 2, as is implemented by a slightly different kernel function (see subsection 5.4). Keerthi et al. [22] noted that the linear slack penalty in the objective function resulted in a smaller

number of support vectors compared to the quadratic slack penalty in their experiments. The theoretical explanation of this phenomenon remains open.

When the number of training examples is large, QP is very difficult to solve, and standard QP routines have substantial memory requirements. One direction of the development of simple solutions to SVMs is centered on splitting the problem into a series of smaller size subtasks [35, 43, 20, 19]. Another direction is focused on an iterative algorithm for training SVMs, which will be discussed below.

SMO (Sequential Minimal Optimization) proposed by Platt [37] works on the Wolfe dual problem and chooses to solve the smallest possible optimization problem at every step. For the standard SVM QP problem, the smallest possible optimization problem involves two Lagrange multipliers, hence at every step, SMO applies some heuristics to choose two Lagrange multipliers to jointly optimize, and there is an analytical solution for the minimal optimization problem as in ROMMA. SMO is especially well-suited for sparse data sets, or training linear SVMs, or soft-margin SVMs with many support vectors at the upper bound.

KA (the Kernel Adatron) proposed by Friess, Cristianini and Campbell [13] also works on the Wolfe dual problem and maximizes the Lagrangian (22) using stochastic gradient ascent based on the derivative of the Lagrangian with respect to individual $\alpha_i's$. KA introduces an additional parameter of learning rate and is proved to converge.

The above two algorithms implement the corresponding soft-margin SVMs by imposing an extra constraint on the Lagrange multipliers as in [8], that is, they solve the dual problem (25), (26) and (27). There is no theoretical analysis of the convergence rate for these two algorithms yet. The training times of SMO were shown to be subquadratic in the number of training examples in the experiments conducted in [6].

Recently a method of training SVMs based on computing the nearest point between two convex polytopes was independently proposed by Kowalczyk [25] and Keerthi et al. [22]. Kowalczyk designed a perceptron-like learning rule to compute the nearest point and proved a convergence rate of $\frac{2R^2\|u\|^2}{\delta^2} ln \frac{R\|u\|}{2}$, where R, u and δ represent the same as in Theorem 5. Keerthi et al. combined and modified two known algorithms [15, 32] of solving nearest point problem and only proved its convergence. Both of their algorithms implement the corresponding soft-margin SVMs by introducing a quadratic slack penalty in the objective function (i.e. $\sigma = 2$), which can be easily converted back to hard margin SVMs with the help of kernel functions. Noise control adopted in our experiments took this idea.

6.1. Comparisons between SMO and aggressive ROMMA

Platt ran his SMO also on MNIST dataset [37]. One classifier (for digit 8) was trained with d set to 5 in the same polynomial kernel function. The inputs are non-binary and are stored as sparse vectors so that his sparse dot product codes can take effect. A KKT tolerance of 0.02 was used to match the AT&T accuracy results in his experiment, that is, the examples on the positive margins have outputs between 0.98 and 1.02. (He argued that recognition systems typically did not need to have the KKT conditions fulfilled to high accuracy and that SVM algorithms would not converge as quickly if required to produce very high accuracy output.) In one of his experiments on this dataset, he set C to 100, and none of the 3450 support vectors is at the upper bound, hence we believe that the solution he got in that experimental setting is for hard margin SVMs. In another experiment, he set C to 10, and some of the support vectors are at the upper bound, therefore we take that the solution got in the latter setting is for soft margin SVMs.

SMO in his experiment was written in C++, using Microsoft's Visual C++ 5.0 compiler. The algorithm was run on an unloaded 266 MHz Pentium II processor with 128M memory and Windows NT 4 system. The CPU time covers the execution of the entire algorithm but excludes file I/O.

To compare with SMO on MNIST dataset, unlike in section 5 where update condition is $y(\vec{w}\cdot\vec{x}) < 1$ and aggressive ROMMA was run until four epochs, we set $\delta = 0.02$ (the desired accuracy level in SMO for this dataset), make an update whenever $y(\vec{w}\cdot\vec{x}) < 1-\delta$ (note that the update condition here is a little different), and ran aggressive ROMMA until $y(\vec{w}\cdot\vec{x}) \geq 1-\delta = 0.98$ for all training instances $(\vec{x}, y)$.

Remember that in aggressive ROMMA, if an update is done on trial t and if $y_t(\vec{w}_t\cdot\vec{x}_t) < 1-\epsilon$, the progress made is at least ϵ^2/R^2. To speed up convergence, we set the initial value of δ to 1, dynamically decrease δ until desired accuracy level and make an update whenever $y(\vec{w}\cdot\vec{x}) < 1-\delta$. In this experiment, δ takes on values from $[1, 0.7, 0.4, 0.1, 0.06, 0.02]$ ($[1, 0.7, 0.4, 0.1, 0.06, 0.02]$ is an arbitrary decreasing list of numbers), an update is needed whenever $y(\vec{w}\cdot\vec{x}) < 1-\delta$, and δ does not take the next value until for all $(\vec{x}, y)$ and the current value of δ, $y(\vec{w}\cdot\vec{x}) \geq 1-\delta$. That δ takes on values from a sequence of numbers of decreasing order and change update condition to $y(\vec{w}\cdot\vec{x}) < 1-\delta$ in aggressive ROMMA is called *control of progress*. Note that control of progress affects mainly speed when the desired accuracy level is small. When there is no control of progress, δ is the desired accuracy level, and the update condition is $y(\vec{w}\cdot\vec{x}) < 1-\delta$.

Our ROMMA series were written in C++, using g++ compiler. The algorithm was run on one (since only the classifier for digit 8 is trained) unloaded 400MHz Pentium II processor with Linux2.2 and 256M memory. Note that the maximum memory requirement of our algorithm for MNIST dataset is about 63M, so 256M memory is a luxury. The CPU time reported in table IV covers the execution of the entire algorithm, excluding file I/O.

The definition of support vectors in (aggressive) ROMMA is slightly different from the standard one, where a training example is viewed as a support vector if its corresponding Lagrange multiplier in the dual problem is different from zero. Since ROMMA works on the primal problem, a training instance is a support vector if it ever causes an update. Thus the number of support vectors in (aggressive) ROMMA is the size of the union of all instances on which an update was made during training.

If the desired accuracy level is different from 0, the solution which aggressive ROMMA obtained is not unique, since for some $(\vec{x}, y)$, $y(\vec{w} \cdot \vec{x})$ may be at least 1 although for all $(\vec{x}, y)$, $y(\vec{w} \cdot \vec{x}) \geq 1 - \delta$. Which of the instances have larger outputs depends on the order of the sample sequence. However, our experiment showed that the total number of support vectors was quite stable when the desired accuracy level is small.

Each component of the input was divided by 1100 in aggressive ROMMA for this dataset, which was implemented efficiently in the dot product. One reason to scale features is for bias (see Subsection 5.2 for details). SMO did not do any preprocessing on this dataset. The CPU time and number of support vectors for aggressive ROMMA in Table IV and Table V average over 10 random permutations of sample sequence, accompanied by 95% confidence interval.

Since SMO and aggressive ROMMA implement different kinds of soft-margin SVMs, we first compare their training times for hard margin SVMs. Among the 10 random runs of aggressive ROMMA, CPU time ranges from 6504 seconds to 7074 seconds, and the number of support vectors ranges from 2600 to 2647. If there is no control of progress, the average training time is 10300 seconds and the average number of support vectors is 3645. Considering that the processor we used is only about as 1.5 times fast as the processor Platt used, it seems that aggressive ROMMA with/without control of progress converged faster than SMO for training hard margin SVMs on this dataset (but their solutions may be different, see the discussion in subsection 5.2).

Next we compare the training time of SMO with that of aggressive ROMMA for soft margin SVMs. Remember that SMO implements the soft margin SVMs by introducing linear slack penalty, while (ag-

Table IV. Comparisons of training times between SMO for hard margin SVMs and aggressive ROMMA (with control of progress) for hard margin fixed threshold SVMs on MNIST dataset (for digit 8 and d=5)

	CPU seconds	# of SVs
SMO (run on 266MHz PC)	29471	3450
agg-ROMMA (run on 400MHz PC)	6708±139	2624±13

gressive) ROMMA by introducing quadratic slack penalty. In Platt's experiment on this dataset for soft margin SVMs, he set C to 10, and 149 out of 3412 support vectors are at the upper bound. In our experiment using aggressive ROMMA with control of progress, we set λ to 30 as in section 5.5. The CPU times and number of support vectors are shown in Table V, where all results for aggressive ROMMA average over 10 random permutations of sample sequence, accompanied by 95% confidence interval.

Table V. Comparisons of training times between SMO for soft margin SVMs and aggressive ROMMA (with control of progress) for soft margin fixed threshold SVMs on MNIST dataset (for digit 8 and d=5)

	CPU seconds	# of SVs
SMO (run on 266MHz PC)	25096	3412
agg-ROMMA (run on 400MHz PC)	6899±121	2763±19

Among the 10 random runs of aggressive ROMMA, CPU time ranges from 6580 seconds to 7125 seconds, and the number of support vectors ranges from 2714 to 2812. Again aggressive ROMMA with control of progress might converge faster than SMO for training soft margin SVMs on this dataset, but to different solutions. We observed that quadratic slack penalty caused the number of support vectors and training times to increase a little; while the linear slack penalty caused the number of support vectors and training times to decrease, which coincides with what Keerthi et al. noted [22].

Note that our algorithm, just like SMO, can also make use of the sparseness of the input, and that the update rule in our algorithm has good property such that many intermediate results can be reused (see subsection 3.3), we expect that aggressive ROMMA converges faster than SMO in general settings if bias can be implemented by the kernel function or other methods (but to different solutions). More experiments are needed to verify this.

7. Conclusion

We designed and analyzed a new incremental algorithm called ROMMA for training linear threshold functions, which can be applied with kernel methods. ROMMA can be viewed as an approximation to the maximum margin classifiers, and its computational complexity and simplicity is similar to that of the perceptron algorithm. Experiments on the MNIST handwritten digits showed that ROMMA performed better than the perceptron algorithm, and aggressive ROMMA had slightly better performance than the voted perceptron algorithm.

Aggressive ROMMA converges to the fixed threshold maximum margin classifier, is simple to implement, does not require a lot of memory, and comes with theoretical bounds on its convergence rate. The future theoretical work might be exploring whether the convergence rate of aggressive ROMMA can be improved, or rather, if it is the best rate achievable by perceptron-like algorithms. It seems that aggressive ROMMA was faster than SMO according to the comparisons in subsection 6.1, however we are not sure of this, since we only got access to the results of SMO run on MNIST with non-fixed threshold, while our aggressive ROMMA implemented the variable threshold with the help of kernel functions (see subsection 5.2 for the difference between them).

We also briefly discussed the role of scaling of features in our experiments. However, it is an open problem to obtain efficiently the optimal scaling for polynomial kernel functions. Bias in (aggressive) ROMMA was implemented by kernel functions at the time being; the future work might be to work out a way to compute bias explicitly for them or to identify the conditions on which the solution that aggressive ROMMA with implicit bias converges to is close to SVM solution (with variable threshold).

Acknowledgments

We thank Adam Kowalczyk for useful discussions, Wee Sun Lee for helpful suggestions and anonymous referees for valuable comments.

References

1. Aizerman, M. A., E. M. Braverman, and L. I. Rozonoer: 1964, 'Theoretical foundations of the potential function method in pattern recognition learning'. *Automation and remote control* **25**, 821–837.
2. Anthony, M. and P. L. Bartlett: 1999, *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.
3. Block, H. D.: 1962, 'The perceptron: A model for brain functioning'. *Reviews of Modern Physics* **34**, 123–135.
4. Boser, B. E., I. M. Guyon, and V. N. Vapnik: 1992, 'A training algorithm for optimal margin classifiers'. *Proceedings of the 1992 Workshop on Computational Learning Theory* pp. 144–152.
5. Burges, C. and D. J. Crisp: 1999, 'Uniqueness of the SVM solution'. In: *Advances in Neural Information Processing Systems 13*.
6. Campbell, C. and N. Cristianini: 1998, 'Simple learning algorithms for training support vector machines'. Technical report, University of Bristol.
7. Chapelle, O. and V. Vapnik: 1999, 'Model selection for support vector machines'. In: *Advances in Neural Information Processing Systems 13*.
8. Cortes, C. and V. Vapnik: 1995, 'Support-vector networks'. *Machine Learning* **20**(3), 273–297.
9. Cristianini, N. and J. Shawe-Taylor: 2000, *An introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press.
10. Fletcher, R.: 1987, *Practical Methods of Optimization*. John Wiley and Sons, Inc., 2nd edition.
11. Freund, Y. and R. E. Schapire: 1998, 'Large margin classification using the perceptron algorithm'. *Proceedings of the 1998 Conference on Computational Learning Theory*.
12. Friedman, J. H.: 1996, 'Another approach to polychotomous classification'. Technical report, Stanford Univ. Department of Statistics. http://www-stat.stanford.edu/reports/friedman/poly.ps.Z.
13. Friess, T. T., N. Cristianini, and C. Campbell: 1998, 'The kernel adatron algorithm: a fast and simple learning procedure for support vector machines'. In: *Proc. 15th Int. Conf. on Machine Learning*.
14. Gallant, S. I.: 1986, 'Optimal Linear Discriminants'. In: *Eighth International Conference on Pattern Recognition*. Paris, France, pp. 849–852.
15. Gilbert, E. G.: 1996, 'Minimizing the quadratic form on a convex set'. *SIAM J. Control* **4**, 61–79.
16. Guo, Y., P. L. Bartlett, J. Shawe-Taylor, and R. Williamson: 1999, 'Covering numbers for support vector machines'. In: *Proceedings of the 1999 Conference on Computational Learning Theory*. pp. 267–277.
17. Helmbold, D. and M. K. Warmuth: 1995, 'On weak learning'. *Journal of Computer and System Sciences* **50**, 551–573.

18. Hertz, J. A., A. Krogh, and R. Palmer: 1991, *Introduction to the theory of neural computation*. Addison-Wesley.
19. Joachims, T.: 1998, 'Making large-scale support vector machines learning practical'. In: B. S. olkopf, C. Burges, and A. Smola (eds.): *Advances in Kernel Methods: Support vector machines*. pp. 169–184.
20. kaufman, L.: 1998, 'Solving the quadratic programming problem arising in support vector classification'. In: B. S. olkopf, C. Burges, and A. Smola (eds.): *Advances in Kernel Methods: Support vector machines*.
21. Kearns, M., M. Li, L. Pitt, and L. G. Valiant: 1987, 'On the Learnability of Boolean Formulae'. *Proceedings of the 19th Annual Symposium on the Theory of Computation* pp. 285–295.
22. Keerthi, S. S., S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy: 99, 'A fast iterative nearest point algorithm for support vector machine classiifer design'. Technical report, Indian Institute of Science. TR-ISL-99-03.
23. Klasner, N. and H. U. Simon: 1995, 'From noise-free to noise-tolerant and from on-line to batch learning'. *Proceedings of the 1995 Conference on Computational Learning Theory* pp. 250–257.
24. Knerr, S., L. Personnaz, and G. Dreyfus: 1990, 'Single-layer learning revisited: A stepwise procedure for building and training a neural network'. In: Fogelman-Soulie and Herault (eds.): *Neurocomputing: Algorithms, Architectures and Applications*. NATO ASI. Springer.
25. Kowalczyk, A.: 1999, 'Maximal Margin Perceptron'. In Smola, Bartlett, Schölkopf, and Schuurmans, editors, *Advances in Large Margin Classifiers*. MIT Press.
26. LeCun, Y., L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, and V. Vapnik: 1995, 'Comparison of learning algorithms for handwritten digit recognition'. In: *Int. conf. on Artificial Neural Networks*. pp. 53–60.
27. Li, Y.: 2000, 'Selective Voting for Perceptron-like Online Learning'. In: *International Conference on Machine Learning 2000*.
28. Littlestone, N.: 1988, 'Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm'. *Machine Learning* **2**, 285–318.
29. Littlestone, N.: 1989a, 'From On-line to Batch Learning'. In: *Proceedings of the 2nd Workshop on Computational Learning Theory*. pp. 269–284.
30. Littlestone, N.: 1989b, 'Mistake Bounds and Logarithmic Linear-threshold Learning Algorithms'. Ph.D. thesis, UC Santa Cruz.
31. Minsky, M. and S. Papert: 1969, expanded edition 1988, *Perceptrons*. Cambridge, MA: MIT Press.
32. Mitchell, B. F., V. F. Dem'yanov, and V. N. Malozemov: 1974, 'Finding the point of a polyhedron closet to the origin'. *SIAM J. Control* **12**, 19–26.
33. Novikoff, A. B. J.: 1962, 'On convergence proofs on perceptrons'. In: *Proceedings of the Symposium on the Mathematical Theory of Automata*. pp. 615–622.
34. Opper, M. and O. Winther: 1999, 'Gaussian Processes and SVM: Mean field results and leave-one-out'. In: Smola, Bartlett, Schölkopf, and Schuurmans (eds.): *Advances in Large margin Classifiers*. MIT Press: Cambridge, MA.
35. Osuna, E., R. Freund, and F. Girosi: 1997, 'An improved training algorithm for support vector machines'. In: J. Principle, L. Gile, N. Margan, and E. Wilson (eds.): *Neural Networks for signal processing VII – Proceedings if the 1997 IEEE workshop*. pp. 276–285.

36. Platt, J., N. Cristianini, and J. Shawe-Taylor: 1999, 'Large margin DAGs for multiclass classification'. In: *Advances in Neural Information Processing Systems 13*. pp. 547–553.
37. Platt, J. C.: 1998, 'Fast training of support vector machines using sequential minimal optimization'. In B. Schõlkopf, C. Burges, A. Smola, editors, *Advances in Kernel Methods: Support Vector Machines*. MIT Press.
38. Rosenblatt, F.: 1958, 'The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain'. *Psychological Review* **65**, 386–407.
39. Rosenblatt, F.: 1962, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Washington, D. C.: Spartan Books.
40. Schapire, R. E., Y. Freund, P. Bartlett, and W. S. Lee: 1998, 'Boosting the Margin: A new Explanation for the Effectiveness of Voting Methods'. *The Annals of Statistics* **26**(5), 1651–1686.
41. Shawe-Taylor, J., P. Bartlett, R. Williamson, and M. Anthony: 1998, 'Structural risk minimization over data-dependent hierarchies'. *IEEE Transactions on Information Theory* **44**(5), 1926–1940.
42. Smola, A., Ó. Z., and W. R.: 2000, 'Regularization with dot-product kernels'. In: *Advances in Neural Information Processing Systems 14*.
43. Vapnik, V.: 1998, *Statistical Learning Theory*. New York. forthcoming.
44. Vapnik, V. N.: 1995a, *The Nature of Statistical Learning Theory*. Springer.
45. Vapnik, V. N.: 1995b, *The Nature of Statistical Learning Theory*. Springer.
46. Wahba, G.: 1999, 'Support vector machines, reproducing kernel hilbert spaces and the randomized GACV'. In: B. Schölkopf, C. J. C. Burges, and A. J. Smola (eds.): *Advances in Kernel Methods — Support Vector Learning*. MIT Press: Cambridge, MA, pp. 69–88.
47. Williams, C. K. I.: 1998, 'Prediction with Gaussian Processes: From linear regression to linear prediction and beyond'. In: M. I. Jordan (ed.): *Learning and Inference in Graphical Models*. Kluwer.
48. Williamson, R. C., A. Smola, and B. Scholkpof: to appear, 'Generalization performance of regularization networks and support vector machines via entropy numbers of compact operators'. *IEEE Transactions on Information Theory*.