The Relaxed Online Maximum Margin Algorithm

Yi Li (liyi@comp.nus.edu.sg)
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 more computationally intensive 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 describe a sense in which the performance of ROMMA converges to that of SVM if the threshold is fixed at 0.

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

1. Introduction

The perceptron algorithm [33, 34] and the maximum-margin classifier [3] have similar theoretical bases, but different strengths. In the case of linearly separable data, Block [2], Novikoff [29] and Minsky and Papert [27] 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 [9] generalized this result to the inseparable case. The maximum-margin algorithm uses quadratic programming to find the weight vector that classifies all the training data correctly and maximizes the margin.

The maximum-margin algorithm is slower than the perceptron algorithm, but generalizes better. On the other hand, the perceptron algorithm is naturally suited for online learning, where patterns are


july6-00.tex; 21/07/2000; 12:57; p.1
encountered one at a time, and for each pattern, a prediction of its classification must be made, then the correct classification is received and the algorithm must update its hypothesis before making the next prediction.

Both the perceptron algorithm and the maximum-margin algorithm can be applied in conjunction with kernel functions [1, 3] 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 Relaxed Online Maximum Margin Algorithm) 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.

The traditional way the perceptron algorithm is used for learning from a batch of training examples is to make a number of passes over the training set, updating the hypothesis when the algorithm's prediction is incorrect. The final prediction vector is used to predict labels outside the training set. There are also several other ways to decide on the best prediction rule given the sequence of different classifiers that the algorithm generates [12, 25, 15, 9]. The majority voting method proposed by Freund and Schapire [9], applying the leave-one-out method of Helmbold and Warmuth [15], has the effect of improving the distribution of margins of the training examples. For detailed analysis, see [35]. Experiments show that the voted perceptron algorithm has better performance than the standard perceptron algorithm [9]. 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 the companion paper [23].

We conducted experiments similar to those performed by Cortes and Vapnik [7] and Freund and Schapire [9] on the problem of handwritten digit recognition. We tested the standard perceptron algorithm, the voted perceptron algorithm (for details, see [9]) and our new algorithm, using the polynomial kernel function with \( d = 4 \) (the choice that was best in [9]). 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.

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. In Section 4, 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 [32, 11, 18, 21] is discussed in Section 5. We conclude with Section 6.

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 [24], learning proceeds in trials. In the \( t \)th trial, the algorithm is first presented with an instance \( \mathbf{x}_t \in \mathbf{R}^n \). Next, the algorithm outputs a prediction \( \hat{y}_t \) of the classification of \( \mathbf{x}_t \). Finally, the algorithm finds out the correct classification \( y_t \in \{-1, 1\} \). If \( y_t \neq \hat{y}_t \), then we say that the algorithm makes a mistake. It is worth emphasizing that in this model, when making its prediction for the \( t \)th 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 \( \mathbf{w}_t \) which is updated between trials, and predicting \( \hat{y}_t = \text{sign}(\mathbf{w}_t \cdot \mathbf{x}_t) \), 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 [33, 34], starts off with \( \mathbf{w}_1 = \mathbf{0} \). When its prediction differs from the label \( y_t \), it updates its weight vector by \( \mathbf{w}_{t+1} = \mathbf{w}_t + y_t \mathbf{x}_t \). 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 \( \mathbf{u} \) such that for all each trial \( t \), \( y_t = \text{sign}(\mathbf{u} \cdot \mathbf{x}_t) \). 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 \( \mathbf{w}_t \) for which for all previous

\(^1\) 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.
trials $s < t$, $\text{sign}(\vec{w}_s \cdot \vec{x}_s) = y_s$, and which maximizes the minimum distance of any $\vec{x}_s$ for $s < t$ to the separating hyperplane. It is known [3, 38] that this can be implemented by choosing $\vec{w}_s$ to minimize $\|\vec{w}_s\|$ subject to the constraints that $y_s(\vec{w}_s \cdot \vec{x}_s) \geq 1$ 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 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

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

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](image_url)

*Figure 1.* In ROMMA, a convex polyhedron in weight space 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 $\tilde{w}_3$. Let us call this halfspace $H_3$. The algorithm continues in this manner. If the $t$th trial is a mistake, then $\tilde{w}_{t+1}$ is chosen to be the smallest element of $H_t \cap \{ \tilde{w} : y_t(\tilde{w} \cdot \tilde{x}_t) \geq 1 \}$, and $H_{t+1}$ is set to be $\{ \tilde{w} : \tilde{w}_{t+1} \cdot \tilde{w} \geq ||\tilde{w}_{t+1}||^2 \}$. If the $t$th trial is not a mistake, then $\tilde{w}_{t+1} = \tilde{w}_t$ and $H_{t+1} = H_t$. We will call $H_t$ the old constraint, and $\{ \tilde{w} : y_t(\tilde{w} \cdot \tilde{x}_t) \geq 1 \}$ 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 $\tilde{w}_{t+1}$ as a function of $\tilde{w}_t$, $\tilde{x}_t$ 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 $y_t(\tilde{w}_t \cdot \tilde{x}_t) < 1$, 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. [26]), 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 $\tilde{w}_t$ as its measure of progress.

We begin with a couple of properties of ROMMA. Our analysis can proceed without these, but they also are useful in deriving our efficient implementation.

**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. $y_t(\tilde{w}_{t+1} \cdot \tilde{x}_t) = 1$.

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

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

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

Note that $\tilde{w}_t$ is the unique smallest member of $H_{t-1} \cap \{ \tilde{w} : y_{t-1}(\tilde{w} \cdot \tilde{x}_{t-1}) \geq 1 \}$ due to the strict convexity of the objective function $\| \cdot \|^2$ [8, 4] and $\tilde{w}_{t+1} \neq \tilde{w}_t$, we have $|\tilde{w}_t|^2 < \|\tilde{w}_{t+1}\|^2$, which implies

$$\|\tilde{w}_q\|^2 < \|\tilde{w}_{t+1}\|^2.$$ (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 $\vec{w}_t+1$. \hfill\Box

**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, $\vec{w}_t \in A_t$. 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

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

($||\vec{x}_t|| > 0$ follows from the fact that the data is linearly separable, and $||\vec{w}_t|| > 0$ 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. \hfill\Box

Now we're ready to prove the mistake bound.

**Theorem 3.** Choose $m \in \mathbb{N}$, and a sequence $(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)$ of pattern-classification pairs in $\mathbb{R}^n \times \{-1, +1\}$. Let $R = \max_{i} ||\vec{x}_i||$. If there is a weight vector $\vec{u}$ such that $y_t(\vec{u} \cdot \vec{x}_i) \geq 1$ for all $1 \leq i \leq m$, then the number of mistakes made by ROMMA on $(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)$ 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 \( \vec{v}_t \) is the smallest element of \( H_t \), we have \( |\vec{v}_t| \leq ||\vec{v}|| \).

We have \( \vec{v}_2 = y_t \vec{x}_1 / ||\vec{x}_1||^2 \), and therefore \( ||\vec{v}_2|| = 1/||\vec{x}_1|| \geq 1/R \) which implies \( ||\vec{v}_2||^2 \geq 1/R^2 \). We claim that if any trial \( t > 1 \) is a mistake, then \( ||\vec{v}_{t+1}||^2 \geq ||\vec{v}_t||^2 + 1/R^2 \). 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 \( ||\vec{v}|| \), will complete the proof.

\[
\begin{align*}
A_t &= \{ \vec{w} : y_t(\vec{w} \cdot \vec{x}_t) = 1 \} \\
B_t &= \{ \vec{w} : (\vec{w} \cdot \vec{w}_t) = ||\vec{w}_t||^2 \}.
\end{align*}
\]

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 \( \rho_t \)) satisfies
\[
\rho_t = \frac{|y_t(\vec{w}_t \cdot \vec{x}_t) - 1|}{||\vec{x}_t||} \geq \frac{1}{||\vec{x}_t||} \geq \frac{1}{R},
\]

since the fact that there was a mistake in trial \( t \) implies \( y_t(\vec{x}_t \cdot \vec{x}_t) \leq 0 \).

As shown in Figure 2, since \( \vec{w}_{t+1} \in A_t \),
\[
||\vec{w}_{t+1} - \vec{w}_t|| \geq \rho_t.
\]

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
\[ |\bar{w}_{t+1}^2 - \bar{w}_t^2| = |\bar{w}_{t+1} - \bar{w}_t|^2 \geq \rho_t^2 \geq 1/R^2, \]
which, as discussed above, completes the proof. \( \square \)

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 [3, 38, 36].

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

Next, we turn to an analysis of aggressive ROMMA.

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

**Proof:** For positive \( \delta \), suppose after trial \( t \) of aggressive ROMMA, an update is made and \( y_t(\bar{w}_t \cdot \bar{x}_t) < 1 - \delta \). We claim that the progress made is always at least \( \delta^2/R^2 \), which will complete the proof. Define \( \rho_t, A_t \) 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 \( \bar{w}_{t+1} \). (See Figure 3.)

Since \( \bar{w}_{t+1} \in A_t \), \( \|\bar{w}_t - \bar{w}_{t+1}\| \geq \rho_t \), which implies
\[ \|\bar{w}_t - \bar{w}_{t+1}\|^2 \geq \rho_t^2. \] (5)

Since \( \bar{w}_{t+1} \) satisfies the old constraint,
\[ \bar{w}_{t+1} \cdot \bar{w}_t = \|\bar{w}_t\|^2 = \bar{w}_t \cdot \bar{w}_t. \] (6)

Thus
\[ \|\bar{w}_{t+1}\|^2 = \|(\bar{w}_{t+1} - \bar{w}_t) + \bar{w}_t\|^2 \]
\[ = (\bar{w}_{t+1} - \bar{w}_t) \cdot (\bar{w}_{t+1} - \bar{w}_t) + 2(\bar{w}_{t+1} - \bar{w}_t) \cdot \bar{w}_t + \bar{w}_t \cdot \bar{w}_t \]
\[ \geq \rho_t^2 + \|\bar{w}_t\|^2, \]
by (5) and (6). Since \( \rho_t \geq \delta/R \), this completes the proof. \( \square \)
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.

Theorem 5 implies that, in a sense, repeatedly cycling through a dataset using aggressive ROMMA will eventually converge to SVM; note however that bias is not considered.

3. An efficient implementation

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 \( A = \begin{pmatrix} \vec{w}_t^T \\ x_t \end{pmatrix} \) and \( b = \begin{pmatrix} |\vec{w}_t|^2 \\ y_t \end{pmatrix} \). Routine calculation shows that

\[
\vec{w}_{t+1} = A^T(AA^T)^{-1}b = \left( \frac{\|\vec{w}_t\|^2 - y_t(\vec{w}_t \cdot \vec{x}_t)}{\|\vec{w}_t\|^2} \right) \vec{w}_t + \left( \frac{\|\vec{w}_t\|^2(y_t - (\vec{w}_t \cdot \vec{x}_t))}{\|\vec{w}_t\|^2 - (\vec{w}_t \cdot \vec{x}_t)^2} \right) \vec{x}_t. \tag{7}
\]

If on trials \( t \) in which a mistake is made, \( c_t = \frac{1}{\|\vec{x}_t\|^2} - \frac{y_t(\vec{w}_t \cdot \vec{x}_t)}{\|\vec{w}_t\|^2 - (\vec{w}_t \cdot \vec{x}_t)^2} \) and \( d_t = \frac{1}{\|\vec{x}_t\|^2 - (\vec{w}_t \cdot \vec{x}_t)^2} \), and on other trials \( c_t = 1 \) and \( d_t = 0 \), then always

\[ \vec{w}_{t+1} = c_t \vec{w}_t + d_t \vec{x}_t. \]
Note that, due to Lemmas 1 and 2, the denominators in (7) 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 \( \langle \vec{x}_i, \vec{x}_j \rangle \) with a kernel function computation \( \mathcal{K}(\vec{x}_i, \vec{x}_j) \).

To make a prediction for the \( t \)th 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 [3, 9], 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

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

where \( \vec{w}_1 \) is the initial weight vector. If we let

\[
\alpha_0 = \prod_{j=1}^{t-1} c_j
\]

and

\[
\alpha_j = \left( \prod_{n=j+1}^{t-1} c_n \right) d_j, \quad 1 \leq j \leq t - 1,
\]

then (8) can be written as

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

Formula (8) may seem daunting; however, making use of the recurrence \( (\vec{w}_{t+1} \cdot \vec{x}) = c_t \langle \vec{w}_t \cdot \vec{x} \rangle + d_t \langle \vec{x}_t \cdot \vec{x} \rangle \), it is obvious that the complexity of our new algorithm is similar to that of perceptron algorithm (in the updating of the perceptron algorithm, \( c_t = 1 \), \( d_t = y_t \)). This was born out by our experiments.

The implementation for aggressive ROMMA is similar to the above: Suppose on trial \( t \), an update is needed, i.e. \( y_t (\vec{w}_t \cdot \vec{x}_t) < 1 \).

If \( y_t (\vec{w}_t \cdot \vec{x}_t) \geq \| \vec{x}_t \|^2 \| \vec{w}_t \|^2 \) then \( \vec{w}_{t+1} = \frac{y_t \vec{x}_t}{\| \vec{x}_t \|^2} \),

\[
\text{otherwise } \vec{w}_{t+1} = c_t \vec{w}_t + d_t \vec{x}_t.
\]
4. 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. [22] have published a detailed comparison of the performance of some of the best algorithms on this dataset. The best test error rate they achieve is 0.7\%, through boosting on top of the neural net LeNet4. A version of the optimal margin classifier (Soft Margin SVMs) [7] achieves a test error rate of 1.1\%.

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 value from \{0, \ldots, 255\}. The second part is a label taking value from \{0, \ldots, 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 \(l \in \{0, \ldots, 9\}\), we replaced each labeled instance \((\bar{x}_i, y_i)\) by the binary-labeled instance \((\bar{x}_i, b_i)\), 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 [31, 10, 20].

To produce output given a test instance \(\bar{x}\), 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 [9] and is an application of the general leave-one-out method of [15]. It records the number of trials each prediction vector survives during the training, which is denoted \(\text{sur}_{i,l}\) 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:

\[
\sum_{i=1}^{k_l} \text{sur}_{i,l} \cdot \text{sign}(w_{l,i} \cdot \bar{x}).
\]

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

produced by ROMMA is analyzed and discussed in the companion paper [23].

We adopt the following polynomial kernel function

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

with $d = 4$ since in experiments on the same problem conducted in [9, 7], the best results occur with this value. 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 [38]). Let us refer to the mapping from the original feature vector to the expanded feature vector as $\Phi$. Note that one component of $\Phi(\vec{x})$ is always 1, and therefore the component of the weight vector corresponding to that component 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

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

(11)

according to (9) and (10), where $c_j$ is always positive, and $d_j$ may be positive or negative. Note that in the standard perceptron algorithm, $c_j = 1$, $d_j = y_j$ for those training examples on which mistakes were made, hence $\alpha_0$ is always 1, so whether $\vec{w}_1 = \Phi(\vec{0})$ or $\vec{w}_1 = \vec{0}$ makes little difference for the perceptron algorithm.

As every entry in the image matrix takes value from $\{0, \ldots, 255\}$, 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 $\vec{w}_i \cdot \Phi(\vec{x}) = c_{t-1}(\vec{w}_{t-1} \cdot \Phi(\vec{x})) + d_{t-1} \mathcal{K}(\vec{x}, \vec{x}_{t-1})$. 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 $\alpha_0$ and $\sum_{i=1}^{t-1} \alpha_i$ in (11) now have the order of magnitude of 1 instead of $\alpha_0$ 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 $\alpha_0$ 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.

Scaling may play other roles. In the point of view of kernel spaces, 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 feature space [14, 6, 39]. There is no efficient method to obtain the optimal scaling for polynomial kernel functions. 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.

Table I. Test error rates on MNIST data

<table>
<thead>
<tr>
<th></th>
<th>$T = 1$</th>
<th>$T = 2$</th>
<th>$T = 3$</th>
<th>$T = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>percep (scale=255)</td>
<td>2.71</td>
<td>2.14</td>
<td>2.03</td>
<td>1.85</td>
</tr>
<tr>
<td>voted-percep (scale=255)</td>
<td>2.23</td>
<td>1.86</td>
<td>1.76</td>
<td>1.71</td>
</tr>
<tr>
<td>ROMMA (scale=1100)</td>
<td>2.48</td>
<td>1.96</td>
<td>1.79</td>
<td>1.77</td>
</tr>
<tr>
<td>agg-ROMMA (scale=1100)</td>
<td>2.14</td>
<td>1.82</td>
<td>1.71</td>
<td>1.67</td>
</tr>
<tr>
<td>agg-ROMMA-NC (scale=1100)</td>
<td>2.05</td>
<td>1.76</td>
<td>1.67</td>
<td>1.63</td>
</tr>
</tbody>
</table>

Table II. Mistake numbers on MNIST data

<table>
<thead>
<tr>
<th></th>
<th>$T = 1$</th>
<th>$T = 2$</th>
<th>$T = 3$</th>
<th>$T = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>percep (scale=255)</td>
<td>7901</td>
<td>10421</td>
<td>11787</td>
<td>12637</td>
</tr>
<tr>
<td>voted-percep (scale=255)</td>
<td>7901</td>
<td>10421</td>
<td>11787</td>
<td>12637</td>
</tr>
<tr>
<td>ROMMA (scale=1100)</td>
<td>7963</td>
<td>9995</td>
<td>10971</td>
<td>11547</td>
</tr>
<tr>
<td>agg-ROMMA (scale=1100)</td>
<td>6077</td>
<td>7391</td>
<td>7901</td>
<td>8139</td>
</tr>
<tr>
<td>agg-ROMMA-NC (scale=1100)</td>
<td>5909</td>
<td>6979</td>
<td>7339</td>
<td>7484</td>
</tr>
</tbody>
</table>

Since the performance of online learning is affected by the order of sample sequence, all the results shown in Table I and Table II average over 10 random permutations. Although online learning would involve only one epoch, we present results for a batch setting until four epochs ($T$ in Table I and Table II represents the number of epochs).

To deal with data which are linearly inseparable in the feature space, and also to improve generalization, Friess et al. [11] suggested the use of quadratic penalty in the cost function, which can be implemented...
using a slightly different kernel function [11, 18] (see also [19]):

\[ \tilde{K}(x_k, x_j) = K(x_k, x_j) + \delta_{kj}\lambda, \]

where \( \delta_{kj} \) is the Kronecker delta function, \( \lambda \) is a predefined parameter. The last rows in Table I and Table II are the results of aggressive ROMMA using this method to control noise (\( \lambda = 30 \) for 10 classifiers).

We conducted three groups of experiments, one for the perceptron algorithm (denoted “percep”), the second for the voted perceptron (denoted “voted-percep”) whose detailed description is in [9], the third for ROMMA, aggressive ROMMA (denoted “agg-ROMMA”), and aggressive ROMMA with noise control (denoted “agg-ROMMA-NC”). Data in the first two groups are scaled with 255, data in the third group are scaled with 1100. All three groups set the initial weight vector \( \tilde{w}_1 = \Phi(\tilde{0}) \).

The test error rates in Table I demonstrate that ROMMA has better performance than the standard perceptron, and aggressive ROMMA has slightly better performance than the voted perceptron. Aggressive ROMMA with noise control should not be compared with perceptrons without noise control. Its presentation is used to show what performance our new online algorithm could achieve (of course it’s not the best, since all 10 classifiers use the same \( \lambda \), which may not be the optimal value). A remarkable phenomenon is that our new algorithms behave very well at the first two epochs. Aggressive ROMMA with noise control running for four epochs takes the longest time, around 5 hours on a cluster of four 400MHz Pentium II PCs with PVM parallel programming.

Mistake numbers shown in Table II 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. Comparison of mistake numbers gives some idea of the relative practical utility of the algorithms in online settings.

5. Related work

Support Vector Machines (SVMs) [7] manifest an impressive resistance to overfitting, which can be explained by the small effective \( VC \) dimension [36]. 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
\[ \frac{1}{2} \| \vec{w} \|^2 \] (12)

subject to
\[ y_i (\vec{w} \cdot \vec{x}_i + b) \geq 1 \quad i = 1, \cdots, l \] (13)

where \( l \) is the number of training examples.

The Wolfe dual of the above QP (Quadratic Programming) problem (12) and (13) is to maximize the Lagrangian
\[ \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (\vec{x}_i \cdot \vec{x}_j) \] (14)

subject to
\[ \alpha_i \geq 0 \quad i = 1, \cdots, l \]
\[ \sum_i \alpha_i y_i = 0. \]

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
\[ \frac{1}{2} \| \vec{w} \|^2 + C \left( \sum_{i=1}^l \xi_i^\sigma \right) \] (15)

subject to
\[ y_i (\vec{w} \cdot \vec{x}_i + b) \geq 1 - \xi_i \quad i = 1, \cdots, l \] (16)

where \( C > 0 \) and \( \sigma \) are given values. The above SVM (15) and (16) 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
\[ \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (\vec{x}_i \cdot \vec{x}_j) \] (17)

under slightly different constraints:
\[ 0 \leq \alpha_i \leq C \quad i = 1, \cdots, l \] (18)
\[ \sum_i^{\alpha_i y_i = 0}. \] (19)

In practice, \( \sigma \) can also be set to 2, as is implemented by a slightly different kernel function (see section 4). Keerthi et. al. [18] argued 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 [30, 37, 17, 16]. Another direction is focused on an iterative algorithm for training SVMs, which will be discussed below.

SMO (Sequential Minimal Optimization) proposed by Platt [32] 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 optimal optimization problem as in ROMMA. SMO is well-suited for sparse data sets, or training linear SVMs, or soft-margin SVMs with many support vectors at the upper bound.

KA (Kernel Adatron) proposed by Friess, Cristianini and Campbell [11] also works on the Wolfe dual problem and maximizes the Lagrangian (14) 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 [7], that is, they solve the dual problem (17), (18) and (19). The training times of SMO and KA are shown to be subquadratic in the number of training examples in the experiments conducted in [5], but there is no theoretical analysis to the convergence rate yet.

Recently a method of training SVMs based on computing the nearest point between two convex polytopes was independently proposed by Kowalczyk [21] and Keerthi et al. [18]. Kowalczyk designed a perceptron-like learning rule to compute the nearest point and proved a convergence rate of \( \frac{2R^2}{\delta^2} \ln \frac{R}{\delta} \frac{d}{u} \), where \( R, u \) and \( \delta \) represent the same as in Theorem 5. Keerthi et al. combined and modified two known algorithms [13, 28] 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.

5.1. Comparisons between SMO and aggressive ROMMA

Platt ran his SMO also on MNIST dataset [32]. 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 a sparse vector 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.) Although he set $C$ to 100, 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 SVM.

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 covering the execution of the entire algorithm but excluding file I/O was reported to be 29471 seconds [32].

Since SMO and aggressive ROMMA implement different kinds of soft-margin SVMs, we compare their training times for hard margin SVMs.

We proved that aggressive ROMMA converges to SVM at the rate of $R^2 \|u\|^2 / \delta^2$ in section 2, where $R$ is the maximal length among all training examples, $\bar{u}$ is a weight vector satisfying the constraints all the training examples impose, and $\delta$ is an accuracy parameter such that for all training examples $(x, y)$ and the obtained aggressive ROMMA solution $\bar{w}$, $y(\bar{w} \cdot x) \geq 1 - \delta$. To compare with SMO on this dataset, unlike in section 4 where $\delta = 0$ and aggressive ROMMA was run until four epochs, we set $\delta = 0.02$, and run aggressive ROMMA until it converges, i.e. $y(\bar{w} \cdot x) \geq 1 - \delta = 0.98$ for all $(x, y)$.

Remember that in aggressive ROMMA, if an update is done on trial $t$ and $y_t(\bar{w}_t \cdot x_t) < 1 - \epsilon$, the progress made is at least $\epsilon^2 / R^2$. To speed up convergence, we set the initial value of $\delta$ to 1, dynamically decrease the value of $\delta$ until desired accuracy level and decide whether to make an update by checking $y(\bar{w} \cdot x) < 1 - \delta$. In other words, $\delta$ takes on values from $[1, 0.7, 0.4, 0.1, 0.06, 0.02]$ in this experiment, an update is needed
whenever \( y(\mathbf{w} \cdot \mathbf{x}) < 1 - \delta \), and \( \delta \) does not take the next value until for all \((\mathbf{x}, y)\) and the current value of \( \delta \), \( y(\mathbf{w} \cdot \mathbf{x}) \geq 1 - \delta \). That \( \delta \) takes on values from a sequence of numbers of decreasing order in aggressive ROMMA is called control of progress. Note that control of progress affects mainly speed, not performance. When there is no control of progress, \( \delta \) is the desired accuracy level, and the processing of every example tries to make the most of the information provided by that single example. Hence, control of progress is suited to batch settings, while no control of progress is suited to online settings.

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 III 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 \( \delta \neq 0 \), the solution which aggressive ROMMA obtained is not unique, since for some \((\mathbf{x}, y)\), \( y(\mathbf{w} \cdot \mathbf{x}) \) may be at least 1 although for all \((\mathbf{x}, y)\), \( y(\mathbf{w} \cdot \mathbf{x}) \geq 1 - \delta \). 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.

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. SMO did not do any preprocessing on this dataset. The CPU time and number of support vectors for aggressive ROMMA in table III average over 10 random permutations of sample sequence, accompanied by 95% confidence interval.

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 1.5 times faster than the processor Platt used, it is clear that aggressive ROMMA converged faster than SMO for training SVMs in this exper-
Table III. Comparisons between SMO and aggressive ROMMA with control of progress on MNIST dataset (for digit 8 and d=5)

<table>
<thead>
<tr>
<th></th>
<th>CPU seconds</th>
<th># of SVs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMO (run on 266MHz PC)</td>
<td>2947</td>
<td>3450</td>
</tr>
<tr>
<td>agg-ROMMA (run on 400MHz PC)</td>
<td>6708±139</td>
<td>2624±13</td>
</tr>
</tbody>
</table>

iment. Note that our algorithm can also make use of the sparseness of the input by implementing dot product efficiently, and that the update rule in our algorithm has good property such that many intermediate results can be reused; we believe that aggressive ROMMA is faster than SMO in general settings if bias can be implemented by the kernel function.

6. Conclusions

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. The performance of an aggressive variant of ROMMA is proved to converge to that of SVMs if bias is not considered. 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 approximates the maximum margin classifier, is simple to implement, does not require a lot of memory, and comes with theoretical guarantees of fast convergence. We also briefly discussed the role of scaling in our experiments. However, it is an open problem to obtain efficiently the optimal scaling for polynomial kernel functions. Bias in ROMMA was implemented by kernel functions at the time being; it may be possible to work out a way to compute bias explicitly for ROMMA.

Acknowledgements

We thank Adam Kowalczyk for useful discussions and Wee Sun Lee for helpful suggestions.
References