Large Margin Classification Using the Perceptron Algorithm

Yoav Freund **Robert E. Schapire**
AT&T Labs
180 Park Avenue
Florham Park, NJ 07932-0971 USA
{yoav, schapire}@research.att.com

Abstract

We introduce and analyze a new algorithm for linear classification which combines Rosenblatt's perceptron algorithm with Helmbold and Warmuth's leave-one-out method. Like Vapnik's maximal-margin classifier, our algorithm takes advantage of data that are linearly separable with large margins. Compared to Vapnik's algorithm, however, ours is much simpler to implement, and much more efficient in terms of computation time. We also show that our algorithm can be efficiently used in very high dimensional spaces using kernel functions. We performed some experiments using our algorithm, and some variants of it, for classifying images of handwritten digits. The performance of our algorithm is close to, but not as good as, the performance of maximal-margin classifiers on the same problem.

1 INTRODUCTION

One of the most influential developments in the theory of machine learning in the last few years is Vapnik's work on support vector machines (SVM) [16]. Vapnik's analysis suggests the following simple method for learning complex binary classifiers. First, use some fixed mapping Φ to map the instances into some very high dimensional space in which the two classes are linearly separable. Then use quadratic programming to find the vector that classifies all the data correctly and maximizes the *margin*, i.e., the minimal distance between the separating hyperplane and the instances.

There are two main contributions of his work. The first is a proof of a new bound on the difference between the training error and the test error of a linear classifier that maximizes the margin. The significance of this bound is that it depends only on the size of the margin (or the number of support vectors) and not on the dimension. It is superior to the bounds that can be given for arbitrary consistent linear classifiers.

The second contribution is a method for computing the maximal-margin classifier efficiently for some specific high dimensional mappings. This method is based on the idea of kernel functions, which are described in detail in Section 4.

The main part of algorithms for finding the maximal-margin classifier is a computation of a solution for a large quadratic program. The constraints in the program correspond to the training examples so their number can be very large. Much of the recent practical work on support vector machines is centered on finding efficient ways of solving these quadratic programming problems.

In this paper, we introduce a new and simpler algorithm for linear classification which takes advantage of data that are linearly separable with large margins. We named the new algorithm the *voted-perceptron* algorithm. The algorithm is based on the well known perceptron algorithm of Rosenblatt [14, 15] and a transformation of online learning algorithms to batch learning algorithms developed by Helmbold and Warmuth [7]. Moreover, we show that kernel functions can be used with our algorithm so that we can run our algorithm efficiently in very high dimensional spaces. Our algorithm and its analysis involve little more than combining these three known methods. On the other hand, the resulting algorithm is very simple and easy to implement, and the theoretical bounds on the expected generalization error of the new algorithm are almost identical to the bounds for SVM given by Vapnik and Chervonenkis [17] in the linearly separable case.

We repeated some of the experiments performed by Cortes and Vapnik [5] on the use of SVM on the problem of classifying handwritten digits. We tested both the voted-perceptron algorithm and a variant based on averaging rather than voting. These experiments indicate that the use of kernel functions with the perceptron algorithm yields a dramatic improvement in performance, both in test accuracy and in computation time. In addition, we found that, when training time is limited, the voted-perceptron algorithm performs better than the traditional way of using the perceptron algorithm (although all methods converge eventually to roughly the same level of performance).

The paper is organized as follows. In Section 2 we describe the voted perceptron algorithm. In Section 3 we derive upper bounds on the expected generalization error for both the linearly separable and inseparable cases. In Section 4 we review the method of kernels and describe how it is used in our algorithm. In Section 5 we summarize the results of

COLT 98 Madison WI USA
Copyright ACM 1998 1-58113-057--0/98/ 7...$5.00

our experiments on the handwritten digit recognition problem. We conclude with Section 6 in which we summarize our observations on the relations between the theory and the experiments and suggest some new open problems.

2 THE ALGORITHM

We assume that all instances are points $\mathbf{x} \in \mathbb{R}^n$. We use $||\mathbf{x}||$ to denote the Euclidean length of $\mathbf{x}$. For most of the paper, we assume that labels y are in $\{-1, +1\}$.

The basis of our study is the classical perceptron algorithm invented by Rosenblatt [14, 15]. This is a very simple algorithm most naturally studied in the on-line learning model. The on-line perceptron algorithm starts with an initial zero prediction vector $\mathbf{v} = \mathbf{0}$. It predicts the label of a new instance $\mathbf{x}$ to be $\hat{y} = \text{sign}(\mathbf{v} \cdot \mathbf{x})$. If this prediction differs from the label y, it updates the prediction vector to $\mathbf{v} = \mathbf{v} + y\mathbf{x}$. If the prediction is correct then $\mathbf{v}$ is not changed. The process then repeats with the next example.

The most common way the perceptron algorithm is used for learning from a batch of training examples is to run the algorithm repeatedly through the training set until it finds a prediction vector which is correct on all of the training set. This prediction rule is then used for predicting the labels on the test set.

Block [2], Novikoff [13] and Minsky and Papert [12] have shown that if the data are linearly separable, then the perceptron algorithm will make a finite number of mistakes, and therefore, if repeatedly cycled through the training set, will converge to a vector which correctly classifies all of the examples. Moreover, the number of mistakes is upper bounded by a function of the gap between the positive and negative examples, a fact that will be central to our analysis.

In this paper, we propose to use a more sophisticated method of applying the on-line perceptron algorithm to batch learning, namely, a variation of the leave-one-out method of Helmbold and Warmuth [7]. In the *voted-perceptron* algorithm, we store more information during training and then use this elaborate information to generate better predictions on the test data. The algorithm is detailed in Figure 1. The information we maintain during training is the list of *all* prediction vectors that were generated after each and every mistake. For each such vector, we count the number of iterations it "survives" until the next mistake is made; we refer to this count as the "weight" of the prediction vector.[1] To calculate a prediction we compute the binary prediction of each one of the prediction vectors and combine all these predictions by a weighted majority vote. The weights used are the survival times described above. This makes intuitive sense as "good" prediction vectors tend to survive for a long time and thus have larger weight in the majority vote.

3 ANALYSIS

In this section, we give an analysis of the voted-perceptron algorithm for the case $T = 1$ in which the algorithm runs exactly once through the training data. We also quote a theorem of Vapnik and Chervonenkis [17] for the linearly separable case. This theorem bounds the generalization error of the consistent perceptron found after the perceptron algorithm is run to convergence. Interestingly, for the linearly separable case, the theorems yield very similar bounds.

```
Training
Input:  a labeled training set $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\rangle$
        number of epochs $T$
Output: a list of weighted perceptrons
        $\langle(\mathbf{v}_1, c_1), \ldots, (\mathbf{v}_k, c_k)\rangle$

  • Initialize: $k := 0$, $\mathbf{v}_1 := \mathbf{0}$, $c_1 := 0$.
  • Repeat $T$ times:
      – For $i = 1, \ldots, m$:
          * Compute prediction: $\hat{y} := \text{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$
          * If $\hat{y} = y$ then $c_k := c_k + 1$.
                      else $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$;
                           $c_{k+1} = 1$;
                           $k := k + 1$.

Prediction
Given:  the list of weighted perceptrons:
        $\langle(\mathbf{v}_1, c_1), \ldots, (\mathbf{v}_k, c_k)\rangle$
        an unlabeled instance: $\mathbf{x}$
compute a predicted label $\hat{y}$ as follows:

        $s = \sum_{i=1}^{k} c_i \, \text{sign}(\mathbf{v}_i \cdot \mathbf{x})$;    $\hat{y} = \text{sign}(s)$ .
```

Figure 1: The voted-perceptron algorithm.

As we shall see in the experiments, the algorithm actually continues to improve performance after $T = 1$. We have no theoretical explanation for this improvement.

If the data are linearly separable, then the perceptron algorithm will eventually converge on some consistent hypothesis (i.e., a prediction vector that is correct on all of the training examples). As this prediction vector makes no further mistakes, it will eventually dominate the weighted vote in the voted-perceptron algorithm. Thus, for linearly separable data, when $T \to \infty$, the voted-perceptron algorithm converges to the regular use of the perceptron algorithm, which is to predict using the final prediction vector.

As we have recently learned, the performance of the final prediction vector has been analyzed by Vapnik and Chervonenkis [17]. We discuss their bound at the end of this section.

We now give our analysis for the case $T = 1$. The analysis is in two parts and mostly combines known material. First, we review the classical analysis of the on-line perceptron algorithm in the linearly separable case, as well as an extension to the inseparable case. Second, we review an analysis of the leave-one-out conversion of an on-line learning algorithm to a batch learning algorithm.

[1] Storing all of these vectors might seem an excessive waste of memory. However, as we shall see, when perceptrons are used together with kernels, the excess in memory and computition is really quite minimal.

3.1 THE ON-LINE PERCEPTRON ALGORITHM IN THE SEPARABLE CASE

Our analysis is based on the following well known result first proved by Block [2] and Novikoff [13]. The significance of this result is that the number of mistakes does not depend on the dimension of the instances. This gives reason to believe that the perceptron algorithm might perform well in high dimensional spaces.

Theorem 1 (Block, Novikoff) *Let $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\rangle$ be a sequence of labeled examples with $||\mathbf{x}_i|| \leq R$. Suppose that there exists a vector* **u** *such that $||\mathbf{u}|| = 1$ and $y_i(\mathbf{u}\cdot\mathbf{x}_i) \geq \gamma$ for all examples in the sequence. Then the number of mistakes made by the on-line perceptron algorithm on this sequence is at most $(R/\gamma)^2$.*

Proof: Although the proof is well known, we repeat it for completeness.

Let $\mathbf{v}_k$ denote the prediction vector used prior to the kth mistake. Thus, $\mathbf{v}_1 = \mathbf{0}$ and, if the kth mistake occurs on $(\mathbf{x}_i, y_i)$ then $y_i(\mathbf{v}_k \cdot \mathbf{x}_i) \leq 0$ and $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i\mathbf{x}_i$.

We have

$$\mathbf{v}_{k+1} \cdot \mathbf{u} = \mathbf{v}_k \cdot \mathbf{u} + y_i(\mathbf{u} \cdot \mathbf{x}_i) \geq \mathbf{v}_k \cdot \mathbf{u} + \gamma.$$

Therefore, $\mathbf{v}_{k+1} \cdot \mathbf{u} \geq k\gamma$.

Similarly,

$$||\mathbf{v}_{k+1}||^2 = ||\mathbf{v}_k||^2 + 2y_i(\mathbf{v}_k \cdot \mathbf{x}_i) + ||\mathbf{x}_i||^2 \leq ||\mathbf{v}_k||^2 + R^2.$$

Therefore, $||\mathbf{v}_{k+1}||^2 \leq kR^2$.

Combining, gives

$$\sqrt{k}R \geq ||\mathbf{v}_{k+1}|| \geq \mathbf{v}_{k+1} \cdot \mathbf{u} \geq k\gamma$$

which implies $k \leq (R/\gamma)^2$ proving the theorem. ■

3.2 ANALYSIS FOR THE INSEPARABLE CASE

If the data are not linearly separable then Theorem 1 cannot be used directly. However, we now give a generalized version of the theorem which allows for some mistakes in the training set. As far as we know, this theorem is new, although the proof technique is very similar to that of Klasner and Simon [9, Theorem 2.2].

Theorem 2 *Let $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\rangle$ be a sequence of labeled examples with $||\mathbf{x}_i|| \leq R$. Let* **u** *be any vector with $||\mathbf{u}|| = 1$ and let $\gamma > 0$. Define the deviation of each example as*

$$d_i = \max\{0, \gamma - y_i(\mathbf{u} \cdot \mathbf{x}_i)\},$$

and define $D = \sqrt{\sum_{i=1}^m d_i^2}$. Then the number of mistakes of the on-line perceptron algorithm on this sequence is bounded by

$$\left(\frac{R+D}{\gamma}\right)^2.$$

Proof: The case $D = 0$ follows from Theorem 1, so we can assume that $D > 0$.

The proof is based on a reduction of the inseparable case to a separable case in a higher dimensional space. As we will see, the reduction does not change the algorithm.

We extend the instance space $\mathbb{R}^n$ to $\mathbb{R}^{n+m}$ by adding m new dimensions, one for each example. Let $\mathbf{x}'_i \in \mathbb{R}^{n+m}$ denote the extension of the instance $\mathbf{x}_i$. We set the first n coordinates of $\mathbf{x}'_i$ equal to $\mathbf{x}_i$. We set the $(n+i)$'th coordinate to Δ where Δ is a positive real constant whose value will be specified later. The rest of the coordinates of $\mathbf{x}'_i$ are set to zero.

Next we extend the comparison vector $\mathbf{u} \in \mathbb{R}^n$ to $\mathbf{u}' \in \mathbb{R}^{n+m}$. We use the constant Z, which we calculate shortly, to ensure that the length of $\mathbf{u}'$ is one. We set the first n coordinates of $\mathbf{u}'$ equal to $\mathbf{u}/Z$. We set the $(n+i)$'th coordinate to $(y_i d_i)/(Z\Delta)$. It is easy to check that the appropriate normalization is $Z = \sqrt{1 + D^2/\Delta^2}$.

Consider the value of $y_i(\mathbf{u}' \cdot \mathbf{x}'_i)$:

$$\begin{aligned} y_i(\mathbf{u}' \cdot \mathbf{x}'_i) &= y_i\left(\frac{\mathbf{u} \cdot \mathbf{x}_i}{Z} + \Delta\frac{y_i d_i}{Z\Delta}\right) \\ &= \frac{y_i(\mathbf{u} \cdot \mathbf{x}_i)}{Z} + \frac{d_i}{Z} \\ &\geq \frac{y_i(\mathbf{u} \cdot \mathbf{x}_i)}{Z} + \frac{\gamma - y_i(\mathbf{u} \cdot \mathbf{x}_i)}{Z} \\ &= \frac{\gamma}{Z}. \end{aligned}$$

Thus the extended prediction vector $\mathbf{u}'$ achieves a margin of $\gamma/\sqrt{1 + D^2/\Delta^2}$ on the extended examples.

In order to apply Theorem 1, we need a bound on the length of the instances. As $R \geq ||\mathbf{x}_i||$ for all i, and the only additional non-zero coordinate has value Δ, we get that $||\mathbf{x}'_i||^2 \leq R^2 + \Delta^2$. Using these values in Theorem 1 we get that the number of mistakes of the on-line perceptron algorithm if run in the extended space is at most

$$\frac{(R^2 + \Delta^2)(1 + D^2/\Delta^2)}{\gamma^2}.$$

Setting $\Delta = \sqrt{RD}$ minimizes the bound and yields the bound given in the statement of the theorem.

To finish the proof we show that the predictions of the perceptron algorithm in the extended space are equal to the prediction of the perceptron in the original space. We use $\mathbf{v}_i$ to denote the prediction vector used for predicting the instance $\mathbf{x}_i$ in the original space and $\mathbf{v}'_i$ to denote the prediction vector used for predicting the corresponding instance $\mathbf{x}'_i$ in the extended space. The claim follows by induction over $1 \leq i \leq m$ of the following three claims:

1. The first n coordinates of $\mathbf{v}'_i$ are equal to those of $\mathbf{v}_i$.
2. The $(n + i)$'th coordinate of $\mathbf{v}'_i$ is equal to zero.
3. $\text{sign}(\mathbf{v}'_i \cdot \mathbf{x}'_i) = \text{sign}(\mathbf{v}_i \cdot \mathbf{x}_i)$.

■

3.3 CONVERTING ON-LINE TO BATCH

We now have an algorithm that will make few mistakes when presented with the examples one by one. However, the setup we are interested in here is the batch setup in which we are given a training set, according to which we generate a hypothesis, which is then tested on a seperate test set. If the data is linearly separable then the perceptron algorithm eventually converges and we can use this final prediction rule as our hypothesis. However, the data might not be separable or we might not want to wait till convergence is achieved. In this case we have to decide on the best prediction rule given the sequence of different classifiers that the online algorithm genarates. One solution to this problem is to use the prediction rule that has survived for the longest time before it was changed. A prediction rule that has survived for a long time is likely to be better than one that has only survived for a few iterations. This method was suggested by Gallant [6] who called it the *pocket method*. Littlestone [11], suggested a two-phase method in which the performance of all of the rules is tested on a seperate test set and the rule with the least error is then used. Here we use a different method for converting the online perceptron algorithm into a batch learning algorithm, the method combines all of the rules generated by the online algorithm after it was run for just a single time through the training data.

We now describe Helmbold and Warmuth's [7] very simple "leave-one-out" method of converting an online learning algorithm into a batch learning algorithm. Our voted-perceptron algorithm is a simple application of this general method. We start with the randomized version. Given a training set $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\rangle$ and an unlabeled instance $\mathbf{x}$, we do the following. We select a number r in $\{0, \ldots, m\}$ uniformly at random. We then take the first r examples in the training sequence and append the unlabeled instance to the end of this subsequence. We run the online algorithm on this sequence of length $r + 1$, and use the prediction of the online algorithm on the last unlabeled instance.

In the deterministic leave-one-out conversion, we modify the randomized leave-one-out conversion to make it deterministic in the obvious way by choosing the most likely prediction. That is, we compute the prediction that would result for all possible choices of r in $\{0, \ldots, m\}$, and we take majority vote of these predictions. It is straightforward to show that taking a majority vote runs the risk of doubling the probability of mistake while it has the potential of significantly decreasing it. In this work we decided to use deterministic voting rather than randomization.

The following theorem follows directly from Helmbold and Warmuth [7]. (See also Kivinen and Warmuth [8] and Cesa-Bianchi et al. [4].)

Theorem 3 *Assume all examples $(\mathbf{x}, y)$ are generated i.i.d. Let E be the* expected *number of mistakes that the online algorithm A makes on a randomly generated sequence of $m + 1$ examples. Then given m random training examples, the expected probability that the randomized leave-one-out conversion of A makes a mistake on a randomly generated test instance is at most $E/(m + 1)$. For the deterministic leave-one-out conversion, this expected probability is at most $2E/(m + 1)$.*

3.4 PUTTING IT ALL TOGETHER

It can be verified that the deterministic leave-one-out conversion of the on-line perceptron algorithm is exactly equivalent to the voted-perceptron algorithm of Figure 1 with $T = 1$. Thus, combining Theorems 2 and 3, we have:

Corollary 4 *Assume all examples are generated i.i.d. at random. Let $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\rangle$ be a sequence of training examples and let $(\mathbf{x}_{m+1}, y_{m+1})$ be a test example. Let $R = \max_{1 \leq i \leq m+1} ||\mathbf{x}_i||$. For $||\mathbf{u}|| = 1$ and $\gamma > 0$, let*

$$D_{\mathbf{u},\gamma} = \sqrt{\sum_{i=1}^{m+1} (\max\{0, \gamma - y_i(\mathbf{u} \cdot \mathbf{x}_i)\})^2}.$$

Then the probability (over the choice of all $m + 1$ examples) that the voted-perceptron algorithm with $T = 1$ does not predict y_{m+1} on test instance $\mathbf{x}_{m+1}$ is at most

$$\frac{2}{m+1} \mathrm{E}\left[\inf_{||\mathbf{u}||=1;\gamma>0} \left(\frac{R + D_{\mathbf{u},\gamma}}{\gamma}\right)^2\right]$$

(where the expectation is also over the choice of all $m + 1$ examples).

In fact, the same proof yields a slightly stronger statement which depends only on examples on which mistakes occur. Formally, this can be stated as follows:

Corollary 5 *Assume all examples are generated i.i.d. at random. Suppose that we run the online perceptron algorithm once on the sequence $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{m+1}, y_{m+1})\rangle$, and that k mistakes occur on examples with indices $i_1, \ldots, i_k$. Redefine $R = \max_{1 \leq j \leq k} ||\mathbf{x}_{i_j}||$, and redefine*

$$D_{\mathbf{u},\gamma} = \sqrt{\sum_{j=1}^{k} (\max\{0, \gamma - y_{i_j}(\mathbf{u} \cdot \mathbf{x}_{i_j})\})^2}.$$

Now suppose that we run the voted-perceptron algorithm on training examples $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\rangle$ for a single epoch. Then the probability (over the choice of all $m + 1$ examples) that the voted-perceptron algorithm does not predict y_{m+1} on test instance $\mathbf{x}_{m+1}$ is at most

$$\frac{2}{m+1} \mathrm{E}[k] \leq \frac{2}{m+1} \mathrm{E}\left[\inf_{||\mathbf{u}||=1;\gamma>0} \left(\frac{R + D_{\mathbf{u},\gamma}}{\gamma}\right)^2\right]$$

(where the expectation is also over the choice of all $m + 1$ examples).

A rather similar theorem was proved by Vapnik and Chervonenkis [17, Theorem 6.1] for training the perceptron algorithm to convergence and predicting with the final perceptron vector.

Theorem 6 (Vapnik and Chervonenkis) *Assume all examples are generated i.i.d. at random. Suppose that we run the online perceptron algorithm on the sequence $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{m+1}, y_{m+1})\rangle$ repeatedly until convergence, and that mistakes occur on a total of k examples with indices $i_1, \ldots, i_k$. Let $R = \max_{1 \leq j \leq k} ||\mathbf{x}_{i_j}||$, and let*

$$\gamma = \max_{||\mathbf{u}||=1} \min_{1 \leq j \leq k} y_{i_j}(\mathbf{u} \cdot \mathbf{x}_{i_j}).$$

Assume $\gamma > 0$ with probability one.

Now suppose that we run the perceptron algorithm to convergence on training examples $\langle(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\rangle$.

Then the probability (over the choice of all $m+1$ examples) that the final perceptron does not predict y_{m+1} on test instance $\mathbf{x}_{m+1}$ is at most

$$\frac{1}{m+1}\mathrm{E}\left[\min\left\{k,\left(\frac{R}{\gamma}\right)^2\right\}\right]$$

(where the expectation is also over the choice of all $m+1$ examples).

For the separable case (in which $D_{\mathbf{u},\gamma}$ can be set to zero), Corollary 5 is almost identical to Theorem 6. One difference is that in Corolary 5, we lose a factor of 2. This is because we use the deterministic algorithm, rather than the randomized one. The other, more important difference is that k, the number of mistakes that the perceptron makes, will almost certainly be larger when the perceptron is run to convergence than when it is run just for a single epoch. This gives us some indication that running the voted-perceptron algorithm with $T=1$ might be better than running it to convergence; however, our experiments do not support this prediction.

Vapnik [18] also gives a very similar bound for the expected error of support-vector machines. There are two differences between the bounds. First, the set of vectors on which the perceptron makes a mistake is replaced by the set of "essential support vectors." Second, the radius R is the maximal distance of any support vector from some optimally chosen vector, rather than from the origin. (The support vectors are the training examples which fall closest to the decision boundary.)

4 KERNEL-BASED CLASSIFICATION

We have seen that the voted-perceptron algorithm has guaranteed performance bounds when the data are (almost) linearly separable. However, linear separability is a rather strict condition. One way to make the method more powerful is by adding dimensions or features to the input space. These new coordinates are nonlinear functions of the original coordinates. Usually if we add enough coordinates we can make the data linearly separable. If the separation is sufficiently good (in the senses of Theorems 1 and 2) then the expected generalization error will be small (provided we do not increase the complexity of instances too much by moving to the higher dimensional space).

However, from a computational point of view, computing the values of the additional coordinates can become prohibitively hard. This problem can sometimes be solved by the elegant method of kernel functions. The use of kernel functions for classification problems was suggested by Boser, Guyon and Vapnik [3], continuing the work of Aizerman, Braverman and Rozonoer [1].

Kernel functions are functions of two variables $K(\mathbf{x},\mathbf{y})$ which can be represented as an inner product $\Phi(\mathbf{x})\cdot\Phi(\mathbf{y})$ for some function $\Phi:\mathbb{R}^n\rightarrow\mathbb{R}^N$ and some $N>0$. In other words, we can calculate $K(\mathbf{x},\mathbf{y})$ by mapping $\mathbf{x}$ and $\mathbf{y}$ to vectors $\Phi(\mathbf{x})$ and $\Phi(\mathbf{y})$ and then taking their inner product.

For instance, an important kernel function that we use in this paper is the polynomial expansion

$$K(\mathbf{x},\mathbf{y})=(1+\mathbf{x}\cdot\mathbf{y})^d\ . \qquad (1)$$

There exist general conditions for checking if a function is a kernel function. In this case, however, it is straightforward to construct Φ witnessing that K is a kernel function. For instance, for $n=3$ and $d=2$, we can choose

$$\Phi(\mathbf{x})=(1,x_1^2,x_2^2,x_3^2,ax_1,ax_2,ax_3,ax_1x_2,ax_1x_3,ax_2x_3)$$

where $a=\sqrt{2}$. In general, we can define $\Phi(\mathbf{x})$ to have one coordinate $cM(\mathbf{x})$ for each monomial $M(\mathbf{x})$ of degree at most d over the variables $x_1,\ldots,x_n$, and where c is an appropriately chosen constant.

Boser, Guyon and Vapnik observed that Vapnik's maximal-margin classifier algorithm can be formulated in such a way that all computations involving instances are in fact in terms of inner products $\mathbf{x}\cdot\mathbf{y}$ between pairs of instances. Thus, if we want to map each instance $\mathbf{x}$ to a vector $\Phi(\mathbf{x})$ in a high dimensional space, we only need to be able to compute inner products $\Phi(\mathbf{x})\cdot\Phi(\mathbf{y})$, which is exactly what is computed by a kernel function. Conceptually, then, with the kernel method, we can work with vectors in a very high dimensional space and the algorithm's performance only depends on linear separability in this expanded space. Computationally, however, we only need to modify the algorithm by replacing each inner product computation $\mathbf{x}\cdot\mathbf{y}$ with a kernel function computation $K(\mathbf{x},\mathbf{y})$.

In this paper, we observe that all the computations in the voted-perceptron learning algorithm involving instances can also be written in terms of inner products, which means that we can apply the kernel method to the perceptron algorithm as well. Referring to Figure 1, we see that both training and prediction involve inner products between instances $\mathbf{x}$ and prediction vectors $\mathbf{v}_k$. In order to perform this operation efficiently, we store each prediction vector $\mathbf{v}_k$ in an implicit form, as the sum of instances that were added or subtracted in order to create it. That is, each $\mathbf{v}_k$ can be written and stored as a sum

$$\mathbf{v}_k=\sum_{j=1}^{k-1}y_{i_j}\mathbf{x}_{i_j}$$

for appropriate indices i_j. We can thus calculate the inner product with $\mathbf{x}$ as

$$\mathbf{v}_k\cdot\mathbf{x}=\sum_{j=1}^{k-1}y_{i_j}(\mathbf{x}_{i_j}\cdot\mathbf{x}).$$

To use a kernel function K, we would merely replace each $\mathbf{x}_{i_j}\cdot\mathbf{x}$ by $K(\mathbf{x}_{i_j},\mathbf{x})$.

Computing the prediction of the final vector $\mathbf{v}_k$ on a test instance $\mathbf{x}$ requires k kernel calculations where k is the number of mistakes made by the algorithm during training. Naively, the prediction of the voted-perceptron would seem to require $O(k^2)$ kernel calculations since we need to compute $\mathbf{v}_j\cdot\mathbf{x}$ for each $j\leq k$, and since v_j itself involves a sum of $j-1$ instances. However, taking advantage of the recurrence $\mathbf{v}_{j+1}\cdot\mathbf{x}=\mathbf{v}_j\cdot\mathbf{x}+y_{i_j}(\mathbf{x}_{i_j}\cdot\mathbf{x})$, it is clear that we can compute the prediction of the voted-perceptron also using only k kernel calculations.

Thus, calculating the prediction of the voted-perceptron when using kernels is only marginally more expensive than calculating the prediction of the final prediction vector, assuming that both methods are trained for the same number of epochs.

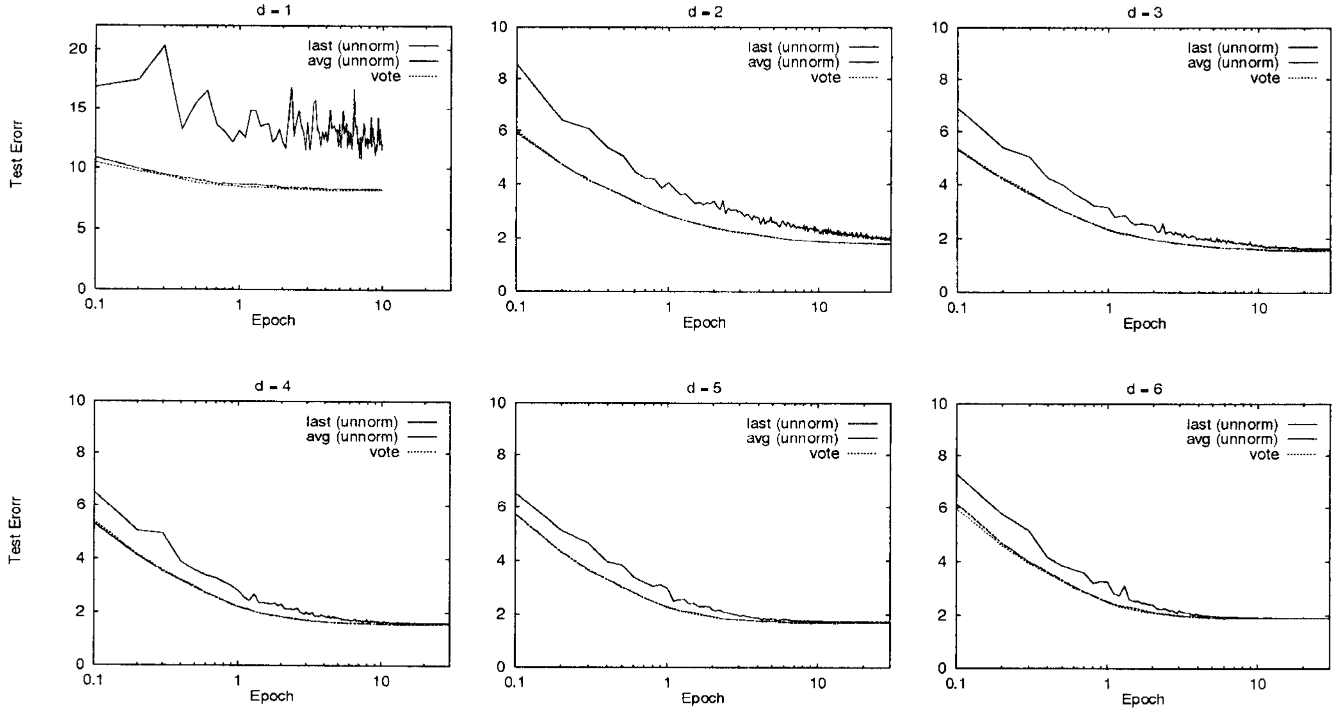

Figure 2: Learning curves for algorithms tested on NIST data.

5 EXPERIMENTS

In our experiments, we followed closely the experimental setup used by Cortes and Vapnik [5] in their experiments on the NIST OCR database.[2] We chose to use this setup because the dataset is widely available and because LeCun et al. [10] have published a detailed comparison of the performance of some of the best digit classification systems in this setup.

Examples in this NIST database consist of labeled digital images of individual handwritten digits. Each instance is a 28×28 matrix in which each entry is an 8 bit representation of a grey value, and labels are from the set $\{0, \ldots, 9\}$. The dataset consists of 60,000 training examples and 10,000 test examples. We treat each image as a vector in $\mathbb{R}^{784}$, and, like Cortes and Vapnik, we use the polynomial kernels of Eq. (1) to expand this vector into very high dimensions.

To handle multiclass data, we essentially reduced to 10 binary problems. That is, we trained the voted-perceptron algorithm once for each of the 10 classes. When training on class ℓ, we replaced each labeled example $(\mathbf{x}_i, y_i)$ (where $y_i \in \{0, \ldots, 9\}$) by the binary-labeled example $(\mathbf{x}_i, +1)$ if $y_i = \ell$ and by $(\mathbf{x}_i, -1)$ if $y_i \neq \ell$. Let

$$\langle (\mathbf{v}^\ell_1, c^\ell_1), \ldots, (\mathbf{v}^\ell_{k_\ell}, c^\ell_{k_\ell}) \rangle$$

be the sequence of weighted prediction vectors which result from training on class ℓ.

To make predictions on a new instance $\mathbf{x}$, we tried three different methods. In each method, we first compute a score s_ℓ for each $\ell \in \{0, \ldots, 9\}$ and then predict with the label receiving the highest score:

$$\hat{y} = \arg\max_{\ell} s_\ell.$$

The first method is to compute each score using the respective final prediction vector:

$$s_\ell = \mathbf{v}^\ell_{k_\ell} \cdot \mathbf{x}.$$

This method is denoted "last (unnormalized)" in the results. A variant of this method is to compute scores after first normalizing the final prediction vectors:

$$s_\ell = \frac{\mathbf{v}^\ell_{k_\ell} \cdot \mathbf{x}}{||\mathbf{v}^\ell_{k_\ell}||}.$$

This method is denoted "last (normalized)" in the results. Note that normalizing vectors has no effect for binary problems, but can plausibly be important in the multiclass case.

The next method (denoted "vote") uses the analog of the deterministic leave-one-out conversion. Here we set

$$s_\ell = \sum_{i=1}^{k_\ell} c^\ell_i \ \mathrm{sign}(\mathbf{v}^\ell_i \cdot \mathbf{x}).$$

The last method (denoted "average (unnormalized)") uses an *average* of the predictions of the prediction vectors

$$s_\ell = \sum_{i=1}^{k_\ell} c^\ell_i \ (\mathbf{v}^\ell_i \cdot \mathbf{x}).$$

As in the "last" method, we also tried a variant (denoted "average (normalized)") using normalized prediction vectors:

$$s_\ell = \sum_{i=1}^{k_\ell} c^\ell_i \ \left(\frac{\mathbf{v}^\ell_i \cdot \mathbf{x}}{||\mathbf{v}^\ell_i||} \right).$$

[2] National Institute for Standards and Technology, Special Database 3. See http://www.research.att.com/~yann/ocr/ for information on obtaining this dataset and for a list of relevant publications.

		$T =$	0.1	1	2	3	4	10	30
$d = 1$	Vote		10.5	8.4	8.3	8.2	8.1	8.2	
	Avg.	(unnorm)	10.9	8.6	8.4	8.3	8.2	8.2	
		(norm)	10.7	8.5	8.4	8.3	8.3	8.2	
	Last	(unnorm)	16.8	13.1	12.2	13.8	13.0	12.0	
		(norm)	16.3	12.9	11.6	13.7	12.7	11.7	
	SupVec		2,484	19,722	24,204	26,625	28,234	32,966	
	Mistake		3,326	25,444	48,446	70,903	93,086	223,798	
$d = 2$	Vote		6.0	2.8	2.4	2.2	2.1	1.8	1.8
	Avg.	(unnorm)	6.0	2.8	2.4	2.2	2.1	1.9	1.8
		(norm)	6.2	3.0	2.5	2.3	2.2	1.9	1.8
	Last	(unnorm)	8.6	4.0	3.4	3.0	2.7	2.3	2.0
		(norm)	8.4	3.9	3.3	3.0	2.7	2.3	1.9
	SupVec		1,639	8,190	9,888	10,818	11,424	12,963	13,861
	Mistake		2,150	10,201	15,290	19,093	22,100	32,451	41,614
$d = 3$	Vote		5.4	2.3	1.9	1.8	1.7	1.6	1.6
	Avg.	(unnorm)	5.3	2.3	1.9	1.8	1.7	1.6	1.5
		(norm)	5.5	2.5	2.0	1.8	1.8	1.6	1.5
	Last	(unnorm)	6.9	3.1	2.5	2.2	2.0	1.7	1.6
		(norm)	6.8	3.1	2.5	2.2	2.0	1.7	1.6
	SupVec		1,460	6,774	8,073	8,715	9,102	9,883	10,094
	Mistake		1,937	8,475	11,739	13,757	15,129	18,422	19,473
$d = 4$	Vote		5.4	2.2	1.8	1.7	1.6	1.6	1.6
	Avg.	(unnorm)	5.3	2.2	1.8	1.7	1.7	1.6	1.6
		(norm)	5.5	2.3	1.9	1.7	1.6	1.6	1.6
	Last	(unnorm)	6.5	2.8	2.3	2.0	1.9	1.6	1.6
		(norm)	6.5	2.8	2.3	2.0	1.9	1.6	1.6
	SupVec		1,406	6,338	7,453	7,944	8,214	8,673	8,717
	Mistake		1,882	7,977	10,543	11,933	12,780	14,375	14,538
$d = 5$	Vote		5.7	2.2	1.9	1.8	1.8	1.7	1.7
	Avg.	(unnorm)	5.7	2.3	1.9	1.8	1.7	1.7	1.7
		(norm)	5.7	2.3	1.9	1.8	1.7	1.7	1.6
	Last	(unnorm)	6.6	3.0	2.2	1.9	1.9	1.8	1.7
		(norm)	6.3	2.9	2.1	1.9	1.9	1.7	1.7
	SupVec		1,439	6,327	7,367	7,788	7,990	8,295	8,313
	Mistake		1,953	8,044	10,379	11,563	12,215	13,234	13,289
$d = 6$	Vote		6.0	2.5	2.1	2.0	1.9	1.9	1.9
	Avg.	(unnorm)	6.2	2.5	2.1	2.0	1.9	1.9	1.9
		(norm)	6.0	2.5	2.1	2.0	1.9	1.8	1.8
	Last	(unnorm)	7.3	3.2	2.4	2.2	2.0	1.9	1.9
		(norm)	6.9	3.0	2.3	2.1	2.0	1.9	1.9
	SupVec		1,488	6,521	7,572	7,947	8,117	8,284	8,285
	Mistake		2,034	8,351	10,764	11,892	12,472	13,108	13,118

Table 1: Results of experiments on NIST data.

Our analysis is applicable only for the case of voted predictions and where $T = 1$. However, in the experiments, we ran the algorithm with T up to 30. When using polynomial kernels of degree 5 or more, the data becomes linearly separable. Thus, after several iterations, the perceptron algorithm converges to a consistent prediction vector and makes no more mistakes. After this happens, the final perceptron gains more and more weight in both "vote" and "average." This tends to have the effect of causing all of the variants to converge eventually to the same solution. By reaching this limit we compare the voted-perceptron algorithm to the standard way in which the perceptron algorithm is used, which is to find a consistent prediction rule.

We performed experiments with polynomial kernels for dimensions $d = 1$ (which corresponds to no expansion) up to $d = 6$. We preprocessed the data on each experiment by randomly permuting the training sequence. For $d > 1$, each experiment was repeated 10 times, each time with a different random permutation of the training examples. For $d = 1$, however, we were only able to run the experiment twice, and only for ten epochs, for reasons which are described below.

Figure 2 shows plots of the test error as a function of the number of epochs for three of the prediction methods — "vote" and the unnormalized versions of "last" and "average" (we omitted the normalized versions for the sake of readability). Test errors are averaged over the multiple runs of the algorithm, and are plotted one point for every tenth of an epoch.

Some of the results are also summarized numerically in Table 1 which shows (average) test error for several values of T for the five different methods in the rows marked "Vote," "Avg. (unnorm)," etc. The rows marked "SupVec" show the number of "support vectors," that is, the total number of instances that actually are used in computing scores as above. In other words, this is the size of the union of all instances on which a mistake occured during training. The rows marked "Mistake" show the total number of mistakes made during training for the 10 different labels. In every case, we have averaged over the multiple runs of the algorithm.

The column corresponding to $T = 0.1$ is helpful for getting an idea of how the algorithms perform on smaller datasets since in this case, each algorithm has only used a tenth of the available data (about 6000 training examples).

Ironically, the algorithm runs slowest with small values of d. For larger values of d, we move to a much higher dimensional space in which the data becomes linearly separable. For small values of d — especially for $d = 1$ — the data is not linearly separable which means that the perceptron algorithm tends to make many mistakes which slows down the algorithm significantly. This is why, for $d = 1$, we were only able to run the algorithm twice, and we could not even complete a run out to 30 epochs but had to stop at $T = 10$ (after about six days of computation). In comparison, for $d = 2$, we can run 30 epochs in about 25 hours, and for $d = 5$ or 6, a complete run takes about 8 hours. (All running times are on a single SGI MIPS R10000 processor running at 194 MHZ.)

The most significant improvement in performance is clearly between $d = 1$ and $d = 2$. The migration to a higher dimensional space makes a tremendous difference compared to running the algorithm in the given space. The improvements for $d > 2$ are not nearly as dramatic.

Our results indicate that voting and averaging perform better than using the last vector. This is especially true prior to convergence of the perceptron updates. For $d = 1$, the data is highly inseparable, so in this case the improvement persists for as long as we were able to run the algorithm. For higher dimensions ($d > 1$), the data becomes more separable and the perceptron update rule converges (or almost converges), in which case the performance of all the prediction methods is very similar. Still, even in this case, there is an advantage to using voting or averaging for a relatively small number of epochs.

There does not seem to be any significant difference between voting and averaging in terms of performance. Using normalized vectors seems to sometimes help a bit for the "last" method, but can help or hurt performance slightly for the "average" method; in any case, the differences in performance between using normalized and unnormalized vectors are always minor.

LeCun et al. [10] give a detailed comparison of algorithms on this dataset. The best of the algorithms that they tested is (a rather old version of) boosting on top of the neural net LeNet 4 which achieves an error rate of 0.7%. A version of the optimal margin classifier algorithm [5], using the same kernel function, performs significantly better than ours, achieving a test error rate of 1.1% for $d = 4$.

6 CONCLUSIONS AND SUMMARY

The most significant result of our experiments is that running the perceptron algorithm in a higher dimensional space using kernel functions produces very significant improvements in performance, yielding accuracy levels that are comparable, though still inferior, to those obtainable with support-vector machines. On the other hand, our algorithm is much faster and easier to implement than the latter method. In addition, the theoretical analysis of the expected error of the perceptron algorithm yields very similar bounds to those of support-vector machines. It is an open problem to develop a better theoretical understanding of the empirical superiority of support-vector machines.

We also find it significant that voting and averaging work better than just using the final hypothesis. This indicates that the theoretical analysis, which suggests using voting, is capturing some of the truth. On the other hand, we do not have a theoretical explanation for the improvement in performance following the first epoch.

Acknowledgments

We thank Vladimir Vapnik for some helpful discussions and for pointing us to Theorem 6.

References

[1] M. A. Aizerman, E. M. Braverman, and L. I. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[2] H. D. Block. The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34:123–135, 1962. Reprinted in "Neurocomputing" by Anderson and Rosenfeld.

[3] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pages 144–152, 1992.

[4] Nicolò Cesa-Bianchi, Yoav Freund, David Haussler, David P. Helmbold, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. *Journal of the Association for Computing Machinery*, 44(3):427–485, May 1997.

[5] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[6] S. I. Gallant. Optimal linear discriminants. In *Eighth International Conference on Pattern Recognition*, pages 849–852. IEEE, 1986.

[7] David P. Helmbold and Manfred K. Warmuth. On weak learning. *Journal of Computer and System Sciences*, 50:551–573, 1995.

[8] Jyrki Kivinen and Manfred K. Warmuth. Additive versus exponentiated gradient updates for linear prediction. *Information and Computation*, 132(1):1–64, January 1997.

[9] Norbert Klasner and Hans Ulrich Simon. From noise-free to noise-tolerant and from on-line to batch learning. In *Proceedings of the Eighth Annual Conference on ComputationalLearning Theory*, pages 250–264, 1995.

[10] Y. LeCun, L. D. Jackel, L. Bottou, A. Brunot, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. A. Muller, E. Sackinger, P. Simard, and V Vapnik. Comparison of learning algorithms for handwritten digit recognition. In *International Conference on Artificial Neural Networks*, pages 53–60, 1995.

[11] Nick Littlestone. From on-line to batch learning. In *Proceedings of the Second Annual Workshop on ComputationalLearning Theory*, pages 269–284, July 1989.

[12] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* The MIT Press, 1969.

[13] A. B. J. Novikoff. On convergence proofs on perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume XII, pages 615–622, 1962.

[14] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–407, 1958. (Reprinted in *Neurocomputing* (MIT Press, 1988).).

[15] F. Rosenblatt. *Principles of Neurodynamics.* Spartan, New York, 1962.

[16] V. N. Vapnik. *Estimation of Dependences Based on Empirical Data.* Springer-Verlag, 1982.

[17] V. N. Vapnik and A. Ya. Chervonenkis. *Theory of pattern recognition.* Nauka, Moscow, 1974. (In Russian).

[18] Vladimir N. Vapnik. *Statistical Learning Theory.* Wiley, 1998 (to appear).