Perceptron Learning with Random Coordinate Descent

Ling Li

Learning Systems Group, California Institute of Technology

Abstract. A perceptron is a linear threshold classifier that separates examples with a hyperplane. It is perhaps the simplest learning model that is used standalone. In this paper, we propose a family of random coordinate descent algorithms for perceptron learning on binary classification problems. Unlike most perceptron learning algorithms, which require smooth cost functions, our algorithms directly minimize the training error, and usually achieve the lowest training error compared with other algorithms. The algorithms are also computationally efficient. Such advantages make them favorable for both standalone use and ensemble learning, on problems that are not linearly separable. Experiments show that our algorithms work very well with AdaBoost, and achieve the lowest test errors for half of the data sets.

1 Introduction

The perceptron was first introduced by Rosenblatt (1958) as a probabilistic model for information processing in the brain. Presented with an input vector $\mathbf{x}$, a perceptron calculates a weighted sum of $\mathbf{x}$, the inner product of $\mathbf{x}$ and its weight vector $\mathbf{w}$. If the sum is above some threshold, the perceptron outputs 1; otherwise it outputs -1.

Since a perceptron separates examples with a hyperplane in the input space, it is only capable of learning linearly separable problems.[1] For problems with more complex patterns, layers of perceptrons have to be connected to form an artificial neural network, and the back-propagation algorithm can be used for learning (Bishop, 1995).

If perfect learning is not required (i.e., nonzero training error is acceptable), the perceptron, as a standalone learning model, is actually quite useful. For instance, Shavlik et al. (1991) reported that the perceptron performed quite well under some qualifications, "hardly distinguishable from the more complicated learning algorithms" such as the feed-forward neural networks. Compared to another simple linear classifier, the decision stump (Holte, 1993), the perceptron is almost as fast to compute, but is more powerful in the sense that it can combine different input features.

Given a data set of examples labeled 1 or -1, the task of perceptron learning usually means finding a hyperplane that separates the examples of different labels with minimal error. When the data set is separable, the task is relatively easy and many algorithms can find the separating hyperplane. For example, the perceptron learning rule (Rosenblatt, 1962) is guaranteed to converge to a separating solution in a finite number of iterations. The support vector machine (SVM) can even find the optimal separating hyperplane that maximizes the minimal margin, by solving a quadratic programming problem (Vapnik, 1998).

However, these algorithms behave poorly when the data set is nonseparable, a more common situation in real-world problems. The perceptron learning rule will not converge, and is very unstable in the sense that the hyperplane might change from an optimal one to a worst-possible one in just one trial (Gallant, 1990). The quadratic programming problem of the hard-margin SVM is unsolvable; even if the soft-margin SVM is used, the solution may be heavily affected by examples that have the most negative margins, and may not be optimal for training error. It is also arguable which criterion, the margin or the training error, is more suitable for nonseparable problems.

[1] In this paper, phrases "linearly separable" and "separable" are interchangeable, and "nonseparable" means "not linearly separable."

Caltech Computer Science Technical Report CaltechCSTR:2005.006, Aug. 2005
http://resolver.caltech.edu/CaltechCSTR:2005.006

There are many other perceptron learning algorithms, some of which will be introduced briefly in the next section. Although those algorithms appear quite different, they usually optimize some cost functions that are differentiable. The training error, although a very simple cost function, has never been minimized directly by those algorithms.

In this paper, we introduce a family of new perceptron learning algorithms that directly minimizes the training error. The essential idea is random coordinate descent, i.e., iteratively optimizing the cost function along randomly picked descent directions. An efficient update procedure is used to exactly minimize the training error along the picked direction. Both the randomness in the direction picking and the exact minimization of the training error help escape from local minima, and thus our algorithms usually achieve the best training error compared with other perceptron learning algorithms.

Although many real-world data sets are simple (Holte, 1993), it is by no means true that a single perceptron is complex enough for all problems. Sometimes more sophisticated learning models are required, and they may be constructed based on perceptrons. For example, the kernel trick used in SVM (Vapnik, 1998) allows the input features to be mapped into some high-dimensional space and a perceptron to be learned there. Another approach is to aggregate many perceptrons together to form a voted ensemble. Our algorithms can work with the kernel trick, but this will be the topic of another paper. In this paper, we explore AdaBoost (Freund & Schapire, 1996) to construct ensembles of perceptrons. We will show that our algorithms, unlike many other algorithms that are not good at reducing the training error, work very well with AdaBoost.

The paper is organized as follows: Some of the existing perceptron learning algorithms are briefly discussed in Section 2. Our random coordinate descent algorithms will be introduced in Section 3. We throughly compare our algorithms with several other perceptron learning algorithms in Section 4, either as standalone learners, or working with AdaBoost. We then conclude in Section 5.

2 Related Work

We assume that the input space is a subset of $\mathbb{R}^m$. A perceptron has a weight vector $\mathbf{w}$ and a bias term b (i.e., the negative threshold). For simplicity, we use the notations $\mathbf{w} = (w_0, w_1, \ldots, w_m)$ and $w_0 = b$ to avoid treating $\mathbf{w}$ and b separately. Each input vector $\mathbf{x}$ is also a real-valued vector in $\mathbb{R}^{m+1}$, with $x_0 = 1$. The perceptron labels the input vector $\mathbf{x}$ by computing the inner product between $\mathbf{w}$ and $\mathbf{x}$,

$$g(\mathbf{x}) = \operatorname{sign}\left(\langle \mathbf{w}, \mathbf{x} \rangle\right).$$

Given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ where $y_i \in \{-1, 1\}$ is the class label, the perceptron learning rule proposed by Rosenblatt (1962) updates the perceptron weight vector when a classification error happens. That is, for an example $(\mathbf{x}, y)$, $\mathbf{w}$ is updated if $g(\mathbf{x}) \neq y$,

$$\mathbf{w}^{\text{updated}} = \mathbf{w} + y\mathbf{x}. \tag{1}$$

This learning rule is applied repeatedly to examples in the training set. If the training set is linearly separable, the perceptron convergence theorem (Rosenblatt, 1962) guarantees that a zero-error weight vector can be found in a finite number of update steps. However, if the training set is nonseparable, the algorithm will never converge and there is no guarantee that the weight vector obtained after any arbitrary number of steps can generalize well.

The pocket algorithm with ratchet (Gallant, 1990) (Algorithm 1) solves the stability problem of perceptron learning at the cost of more computational effort. It runs the learning rule (1) while keeping "in its pocket" an extra weight vector, which is the best-till-now solution. Whenever the perceptron weight vector is better than the pocket weight vector, the perceptron one replaces the

Algorithm 1: The pocket algorithm with ratchet (Gallant, 1990) (note that the training error calculation, step 8, may be skipped when the weight has not changed)

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$; Number of epochs T.

1: Initialize $\mathbf{w}$ {usually this means setting $\mathbf{w} \leftarrow \mathbf{0}$}
2: $\gamma \leftarrow 0$, $\mathbf{w}_\mathrm{p} \leftarrow \mathbf{w}$, $\gamma_\mathrm{p} \leftarrow 0$, $e_\mathrm{p} \leftarrow 1$
3: **for** $T \times N$ trials **do**
4:     Randomly pick an example $(\mathbf{x}_k, y_k)$
5:     **if** $y_k \langle \mathbf{w}, \mathbf{x}_k \rangle > 0$ **then** {$\mathbf{w}$ correctly classifies $(\mathbf{x}_k, y_k)$}
6:         $\gamma \leftarrow \gamma + 1$
7:         **if** $\gamma > \gamma_\mathrm{p}$ **then**
8:             $e \leftarrow \frac{1}{N} \sum_i [y_i \langle \mathbf{w}, \mathbf{x}_i \rangle \leq 0]$ {the training error of $\mathbf{w}$}
9:             **if** $e < e_\mathrm{p}$ **then**
10:                 $\mathbf{w}_\mathrm{p} \leftarrow \mathbf{w}$, $e_\mathrm{p} \leftarrow e$, $\gamma_\mathrm{p} \leftarrow \gamma$
11:             **end if**
12:         **end if**
13:     **else** {$\mathbf{w}$ wrongly classifies $(\mathbf{x}_k, y_k)$}
14:         $\mathbf{w} \leftarrow \mathbf{w} + y_k \mathbf{x}_k$ {the Rosenblatt's update rule (1)}
15:         $\gamma \leftarrow 0$
16:     **end if**
17: **end for**
18: **return** $\mathbf{w}_\mathrm{p}$ as the perceptron weight vector

Algorithm 2: The averaged-perceptron algorithm (Freund & Schapire, 1999)

Input: A training set $\{(\mathbf{x}_i, y_i)\}$; Number of epochs T.

1: $t \leftarrow 1$, $\gamma_t \leftarrow 0$, initialize $\mathbf{w}_t$ {usually this means setting $\mathbf{w}_t \leftarrow \mathbf{0}$}
2: **for** T epochs **do**
3:     **for** $k = 1$ to N **do**
4:         **if** $y_k \langle \mathbf{w}_t, \mathbf{x}_k \rangle > 0$ **then** {$\mathbf{w}_t$ correctly classifies $(\mathbf{x}_k, y_k)$}
5:             $\gamma_t \leftarrow \gamma_t + 1$
6:         **else** {$\mathbf{w}_t$ wrongly classifies $(\mathbf{x}_k, y_k)$}
7:             $t \leftarrow t + 1$
8:             $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} + y_k \mathbf{x}_k$ {the Rosenblatt's update rule (1)}
9:             $\gamma_t \leftarrow 1$
10:         **end if**
11:     **end for**
12: **end for**
13: **return** $\sum_{i=1}^{t} \gamma_i \mathbf{w}_i$ as the perceptron weight vector

pocket one. The ratchet check (step 9 in Algorithm 1) ensures that the training error of the pocket weight vector will only strictly decrease. Although the pocket algorithm can find an optimal weight vector that minimizes the training error with arbitrarily high probability, in practice, the number of trials required to produce an optimal weight vector is prohibitively large (Gallant, 1990).

In contrast with the pocket algorithm, which uses only the best weight vector, Freund and Schapire (1999) suggested combining all the weight vectors that occur in a normal perceptron learning by a majority vote. Each vector is weighted by its survival time, the number of trials before the vector is updated. Although this algorithm does not generate a linear classifier, one variant that uses averaging instead of voting—the averaged-perceptron algorithm (Algorithm 2)—does produce a linear classifier. Since their experiments showed that the voted- and averaged-perceptron algorithms had no significant difference in terms of performance, we will only consider the averaged-perceptron algorithm in this paper.

It is interesting to note that the perceptron learning rule (1) is actually the sequential gradient descent on a cost function known as the perceptron criterion,

$$C(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \max\left\{0, -y_i \langle \mathbf{w}, \mathbf{x}_i \rangle\right\}. \tag{2}$$

The pocket algorithm aims at minimizing the training error, but adopts the gradient of the perceptron criterion for weight update, and thus is not efficient. Using the same update rule, the averaged-perceptron algorithm also tries to minimize the perceptron criterion, but is heavily regularized via averaging.

Besides the perceptron criterion, there are algorithms that adopt other cost functions, such as the sum-of-squares error (also called the least-squares error), and minimize them by stochastic gradient descent (Zhang, 2004). Most cost functions for binary classification problems can be expressed as the sample sum of the example margin cost. That is,

$$C(\mathbf{w}) = \sum_{i=1}^{N} \varphi_i c\left(y_i \langle \mathbf{w}, \mathbf{x}_i \rangle\right),$$

where φ_i is the sample weight for example $(\mathbf{x}_i, y_i)$, $y_i \langle \mathbf{w}, \mathbf{x}_i \rangle$ is the unnormalized margin of the example, and $c\colon \mathbb{R} \to \mathbb{R}^+$ is a margin cost function. Several margin cost functions are listed in Table 1. In order to apply gradient descent, the margin cost has to be differentiable. Thus gradient descent type algorithms cannot work on the training error cost function, where $c(\rho) = [\rho \leq 0]$. Another problem with such approaches is that the optimization process usually sticks at some local minima, and cannot go close to the optimal solutions.

The minimal (normalized) margin, which is the minimal distance from the examples to the separating hyperplane, plays an important role in bounding the number of mistakes made by a normal perceptron learning (Freund & Schapire, 1999). Usually the larger the margin is, the smaller the bound is, and the better the perceptron generalizes. Thus many algorithms aim at maximizing the minimal margin. For example, SVM tries to minimize the magnitude of the weight vector $\|\mathbf{w}\|$ while keeping the unnormalized margin bounded from below (Vapnik, 1998). The averaged-perceptron may achieve a better margin distribution through averaging, similar to how AdaBoost improves the base learner (Schapire et al., 1998).

The relaxed online maximum margin algorithm (ROMMA) is another algorithm that approximately maximizes the margin (Li & Long, 2002). Each update of ROMMA tries to minimize $\|\mathbf{w}\|$ according to some relaxed constraints. When the data set is separable, a certain way of running ROMMA converges to the maximum margin solution. However, there is yet no theoretical analysis on the behavior of the algorithm when the data set is nonseparable.

It is arguable that the margin is the right criterion to optimize when the data set is nonseparable. Outliers, which usually have very negative margins, may heavily affect the solution if we insist on maximizing the minimal margin. The training error, on the contrary, suffers less from outliers since the error count is the same no matter how negative the margins are.

Table 1: Several cost functions in the form of $C(\mathbf{w}) = \sum_{i=1}^{N} \varphi_i c\left(y_i \langle \mathbf{w}, \mathbf{x}_i \rangle\right)$

cost function	$c(\rho)$
perceptron criterion	$\max\{0, -\rho\}$
SVM hinge loss	$\max\{0, 1-\rho\}$
least-squares error	$(1-\rho)^2$
modified least-squares	$(\max\{0, 1-\rho\})^2$
0/1 loss training error	$[\rho \leq 0]$

3 Random Coordinate Descent

There are two elements in the perceptron learning rule (1) that may be altered for possibly better learning. The first one is the descent direction, which is $y\mathbf{x}$ in (1), and the second is the descent step, which is always 1 in (1). If we replace them with a vector $\mathbf{d}$ and a scalar α, respectively, the learning rule becomes

$$\mathbf{w}^{\text{updated}} = \mathbf{w} + \alpha\mathbf{d}. \tag{3}$$

Different choices on $\mathbf{d}$ and α may lead to different perceptron learning rules. In this section, we propose a family of new algorithms with proper choices of $\mathbf{d}$ and α to directly minimize the training error.

3.1 Finding Optimal Descent Step

We will discuss how to choose the descent directions later in Subsection 3.2. For now, let us assume that a descent direction $\mathbf{d}$ has been picked. We will find the the descent step α to minimize the training error along the direction $\mathbf{d}$. That is, we need to solve this subproblem:

$$\min_{\alpha\in\mathbb{R}} e\left(g_{\mathbf{w}+\alpha\mathbf{d}}\right) = \sum_{i=1}^{N} \varphi_i \left[y_i \left\langle \mathbf{w}+\alpha\mathbf{d}, \mathbf{x}_i\right\rangle \le 0\right].$$

Let us first look at how the error on example $(\mathbf{x}, y)$ is decided for the weight vector $(\mathbf{w}+\alpha\mathbf{d})$. Denote $\langle \mathbf{d}, \mathbf{x}\rangle$ by δ.

- When $\delta \neq 0$,
$$\langle \mathbf{w}+\alpha\mathbf{d}, \mathbf{x}\rangle = \langle \mathbf{w}, \mathbf{x}\rangle + \alpha\delta = \delta\left(\delta^{-1}\langle \mathbf{w}, \mathbf{x}\rangle + \alpha\right). \tag{4}$$
Thus
$$g_{\mathbf{w}+\alpha\mathbf{d}}(\mathbf{x}) = \operatorname{sign}(\delta)\cdot\operatorname{sign}\left(\delta^{-1}\langle \mathbf{w}, \mathbf{x}\rangle + \alpha\right).$$
This means that the error of $g_{\mathbf{w}+\alpha\mathbf{d}}$ on example $(\mathbf{x}, y)$ is the same as the error of a 1-D linear threshold function with bias α on the example $\left(\delta^{-1}\langle \mathbf{w}, \mathbf{x}\rangle, y \operatorname{sign}(\delta)\right)$.
- When $\delta = 0$,
$$g_{\mathbf{w}+\alpha\mathbf{d}}(\mathbf{x}) = \operatorname{sign}\left(\langle \mathbf{w}+\alpha\mathbf{d}, \mathbf{x}\rangle\right) = \operatorname{sign}\left(\langle \mathbf{w}, \mathbf{x}\rangle\right).$$
Thus the descent step α will not change the output on the input $\mathbf{x}$.

The 1-D linear threshold function is actually a decision stump, which has a deterministic and efficient learning algorithm that minimizes the training error (Holte, 1993). Hence, we can transform all training examples that have $\delta_i \neq 0$ with the mapping below,

$$(\mathbf{x}_i, y_i) \mapsto \left(\delta_i^{-1}\langle \mathbf{w}, \mathbf{x}_i\rangle, y_i \operatorname{sign}(\delta_i)\right), \tag{5}$$

and then apply the decision stump learning algorithm to the transformed data set to decide the optimal descent step α^*.

Since α is not restricted to positive numbers, the direction $\mathbf{d}$ is not required to be strictly descent. As an extreme example, using $-\mathbf{d}$ as the search direction in (5) will merely negate the transformed 1-D examples, and $-\alpha^*$ will then be returned by the decision stump learning algorithm.

Note that a decision stump can have positive or negative directions. That is, it can be $\operatorname{sign}(x+\alpha)$ or $-\operatorname{sign}(x+\alpha)$. Although we expect the learning algorithm to return a decision stump with positive direction, it is still possible that a negative-direction one will be found.[2] When this happens, the weight vector should be negated; the examples with $\delta_i = 0$ will also have different errors, and thus they cannot be ignored as what we just described. The full update procedure is described in Algorithm 3. The classification error on those examples with $\delta_i = 0$ is essential in deciding the

[2] This usually happens when the initial weight vector has a training error larger than $\frac{1}{2}$.

Algorithm 3: The update procedure for random coordinate descent

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ and its sample weight $\{\varphi_i\}_{i=1}^N$; The current weight $\mathbf{w}$; A descent direction $\mathbf{d}$.

for $i = 1$ to N **do** {generate the 1-D data set}
    $\delta_i \leftarrow \langle \mathbf{d}, \mathbf{x}_i \rangle$
    if $\delta_i \neq 0$ **then**
        $x'_i \leftarrow \delta_i^{-1} \langle \mathbf{w}, \mathbf{x}_i \rangle$, $y'_i \leftarrow y_i \operatorname{sign}(\delta_i)$
    else
        $x'_i \leftarrow \infty$, $y'_i \leftarrow y_i \operatorname{sign}(\langle \mathbf{w}, \mathbf{x}_i \rangle)$ {set $\operatorname{sign}(0) = -1$ only here}
    end if
end for
Find the optimal decision stump for $\{(x'_i, y'_i)\}_{i=1}^N$ and $\{\varphi_i\}_{i=1}^N$,

$$(q^*, \alpha^*) = \arg \min_{q \in \{-1,+1\}, \alpha \in \mathbb{R}} \sum_i \varphi_i \left[y'_i \cdot q \cdot \operatorname{sign}\left(x'_i + \alpha\right) \leq 0 \right]$$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^* \mathbf{d}$
if $q^* = -1$ **then**
    $\mathbf{w} \leftarrow -\mathbf{w}$
end if

Algorithm 4: The update procedure for random coordinate descent using a positive-direction decision stump

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ and its sample weight $\{\varphi_i\}_{i=1}^N$; The current weight $\mathbf{w}$; A search direction $\mathbf{d}$.

for $i = 1$ to N **do** {generate the 1-D data set}
    $\delta_i \leftarrow \langle \mathbf{d}, \mathbf{x}_i \rangle$
    if $\delta_i \neq 0$ **then**
        $x'_i \leftarrow \delta_i^{-1} \langle \mathbf{w}, \mathbf{x}_i \rangle$, $y'_i \leftarrow y_i \operatorname{sign}(\delta_i)$
    end if
end for
Find the optimal decision stump for $\{(x'_i, y'_i)\}$ and $\{\varphi_i\}$, only considering those with $\delta_i \neq 0$,

$$\alpha^* = \arg \min_{\alpha \in \mathbb{R}} \sum_{i:\ \delta_i \neq 0} \varphi_i \left[y'_i \cdot \operatorname{sign}\left(x'_i + \alpha\right) \leq 0 \right]$$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^* \mathbf{d}$

optimal direction, acting as an error bias for the positive direction.

We also use a simplified procedure (Algorithm 4), considering only positive-direction decision stumps. Since the emergence of negative-direction decision stumps is really rare and usually happens at the beginning of the optimization, we choose the simplified one for our experiments.

The computational complexity of both the update procedures is $O[mN + N \log N]$. The mapping (5) takes N inner product operations, which has complexity $O[mN]$. The decision stump learning requires to sort the transformed 1-D data set, and the complexity is $O[N \log N]$. Looking for the optimal bias is just an operation linear in N. Compared with the standard perceptron learning whose complexity is $O[mN]$ for every epoch (to examine the inner product with N examples), our update procedure is still very efficient, especially when the number of examples is comparable to 2^m.

3.2 Choosing Descent Directions

There are many ways to choose the descent directions.

Even if the cost function we are minimizing is the 0/1 loss training error, we can still adopt the gradient of the perceptron criterion as the descent direction. Actually, we may use the gradient of any reasonable smooth cost function as our descent direction.

Algorithm 5: Random coordinate descent algorithm for perceptrons

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ and its sample weight $\{\varphi_i\}_{i=1}^N$; Number of epochs T.
1: Initialize $\mathbf{w}$
2: **for** T epochs **do**
3: Generate a random vector $\mathbf{d} \in \mathbb{R}^{m+1}$ as the descent coordinate
4: Do the weight update procedure with $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, $\{\varphi_i\}_{i=1}^N$, $\mathbf{w}$, and $\mathbf{d}$
5: **end for**
6: **return** $\mathbf{w}$ as the perceptron weight

The cyclic coordinate descent (CCD), also known as the iterative coordinate descent, can be used when the cost function is not differentiable. It picks one coordinate at a time and changes the value of that coordinate in the weight vector. In other words, if we denote the i-th basis vector by $\mathbf{e}_i$, e.g., $\mathbf{e}_0 = (1, 0, \ldots, 0)^T$, CCD uses $\mathbf{e}_i$ as the descent direction.

However, except for the possible actual meanings that the original coordinates may have, there is nothing special about the original coordinate system—we can set up another coordinate system and do CCD there. That is, we can pick a random basis, which is a set of pairwise orthogonal vectors, and iteratively use each basis vector as the descent direction. In order to avoid local minima caused by a fixed coordinate system, a different random basis shall be put in use every once in a while. Another more radical and more generalized choice is to every time pick a new random vector as the descent vector, as summarized in Algorithm 5, the *random coordinate descent* (RCD) algorithm.

We have investigated two general ways of picking random vectors. The first one, which we refer to as the *uniform random vectors*, picks each component of the vector from a uniform distribution spanned over the corresponding feature range. If the input features of the examples have been normalized to $[-1, 1]$ (see Section 4 for more details), each component is a random number uniformly in $[-1, 1]$. The other one uses Gaussian distribution instead of the uniform distribution, and is named *Gaussian random vectors*. If the features have been normalized to have zero mean and unit variance, each component is then picked from a unit Gaussian distribution. This approach has the nice property that the angle of the random vectors is uniformly distributed.

3.3 Variants of RCD

We can get different variants of RCD by using different schedules of random descent directions. For example, if $\mathbf{e}_i$ $(i = 0, \ldots, m)$ is iteratively picked as the descent direction, we get CCD.

When a random basis of $(m + 1)$ pairwise orthogonal vectors is used for every $(m + 1)$ epochs, we refer to it as RCD-conj. RCD-grad is RCD with the gradient of the perceptron criterion.

One thing we have noticed is that the range of the bias, w_0, can be quite different from those of the other components of $\mathbf{w}$. Thus it might be necessary to have a descent direction devoted to adjusting w_0 only. If the vector $\mathbf{e}_0$ is adopted every $(m + 1)$ epochs in addition to other settings, RCD becomes RCD-bias, and RCD-conj becomes RCD-conj-bias.

4 Experiments

We compare our RCD algorithms with several existing perceptron learning algorithms, as both standalone learners and base learners for AdaBoost. Experiments are carried out on nine real-world data sets[3] from the UCI machine learning repository (Hettich et al., 1998), and three artificial data

[3] They are australian (Statlog: Australian Credit Approval), breast (Wisconsin Breast Cancer), cleveland (Heart Disease), german (Statlog: German Credit), heart (Statlog: Heart Disease), ionosphere (Johns Hopkins University Ionosphere), pima (Pima Indians Diabetes), sonar (Sonar, Mines vs. Rocks), and votes84 (Congressional Voting Records), with incomplete records removed.

sets[4]. Each real-world data set is randomly shuffled and split with 80% of the data for training and the rest for testing. Each artificial data set has 5000 randomly generated examples, of which 600 are used for training. The perceptron algorithms are allowed to run $T = 2000$ epochs. This is repeated 500 times to get the mean and the standard error of the training and test errors.

Data Preprocessing. Solely based on the feature distribution in the training set, we shift and scale the features in the training set to $[-1, 1]$, and correspondingly normalize the test set.[5] Thus we use the uniform random vectors for RCD algorithms.

Initial Seeding. We initialize the perceptron weight vector with two possible vectors, the zero vector and the Fisher's linear discriminant (FLD, see for example (Bishop, 1995)). For the latter case, when the within-class covariance matrix estimate happens to be singular, we regularize it with a small eigenvalue shrinkage parameter of the value 10^{-10}, just large enough to permit numerically stable inversion (Friedman, 1999).

4.1 Comparing Variants of RCD

We first look at the in-sample performance of our RCD algorithms. Figure 1 shows, for the pima data set,[6] the training errors for several RCD algorithms. We can see that

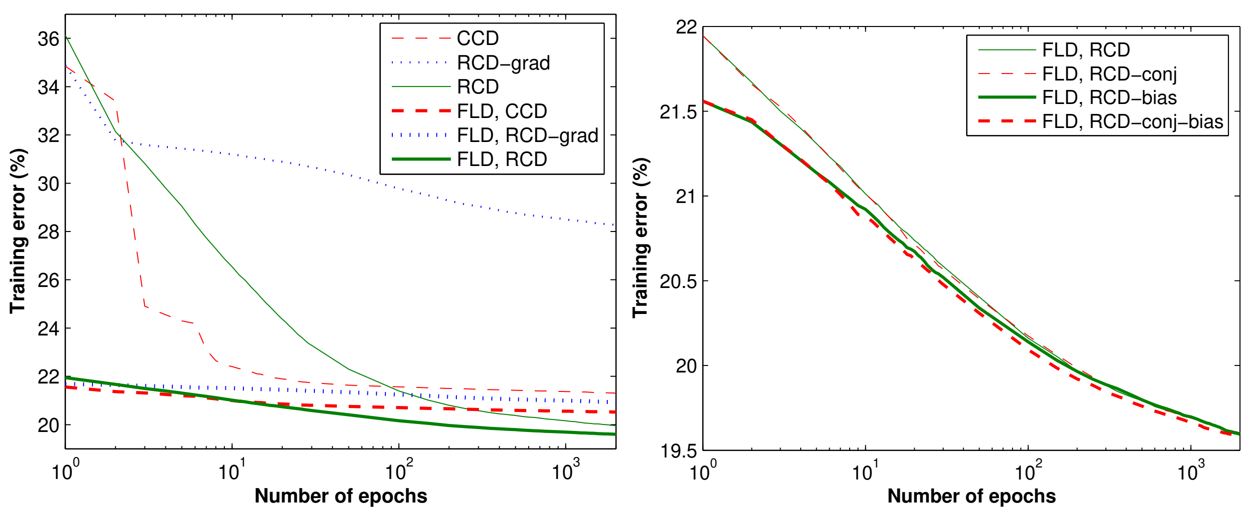

Figure 1: Training errors of several RCD algorithms on the pima data set

– With FLD as a much better initial weight vector, the RCD algorithms achieve final training errors significantly lower than those obtained from the zero starting vector.
– RCD-grad does not work as well as other RCD algorithms. Apparently this is because the descent direction it uses is the gradient of the perceptron criterion, but the optimization is for the training error.

[4] They are ringnorm and threenorm (Breiman, 1996; Breiman, 1998), and yinyang (Li et al., 2005, Yin-Yang).

[5] Note that a common practice is to normalize based on all the examples, with the benefit of doing it only once before the data splitting. However, since our RCD algorithms are affected by the range of the random descent directions, even this "tiny" peek into the test set will give our algorithms an unfair edge.

[6] Most plots in this paper are based on results on the pima data set. However, there is nothing special about pima. It is just a data set picked for illustration purposes.

– Randomness in the direction picking is important. Even without FLD, RCD surpasses CCD with FLD in the end.
– Whether to use groups of orthogonal directions seems not affecting the performance significantly.
– The bias direction $\mathbf{e}_0$ does yield a better optimization, especially at the beginning. However, the edge gets smaller with more training epochs.

Thus for clearer comparison with other perceptron learning algorithms, we shall focus on RCD and RCD-bias.

4.2 Comparing with Other Algorithms

We compare our RCD algorithms with several other perceptron algorithms, including the pocket algorithm with ratchet (pocket) (Gallant, 1990), averaged-perceptron (ave-perc) (Freund & Schapire, 1999), stochastic gradient descent with a learning rate 0.002 on the SVM hinge loss (SGD-hinge) and that on the modified least-squares (SGD-mls) (Zhang, 2004), and the soft-margin SVM with the linear kernel and parameter selection (soft-SVM) (Chang & Lin, 2001; Hsu et al., 2003).

It should be mentioned that when Freund and Schapire (1999) proposed the voted-perceptron and averaged-perceptron algorithms, they did not pay much attention to how the examples should be presented in multi-epoch runs, since their theoretical result on the error bound is only applicable to one-epoch run of the voted-perceptron. We find that cycling through examples with a fixed order[7] is not optimal for multi-epoch runs of the averaged-perceptron. Randomly permuting the training set at the beginning of each epoch or simply choosing examples at random at each trial can improve both the in-sample and the out-of-sample performance (see Figure 2 for a comparison on the pima data set). In our experiments, we use averaged-perceptron with the random sampling (see line 3 of Algorithm 6). Figure 2 also shows that using FLD only helps for early epochs.

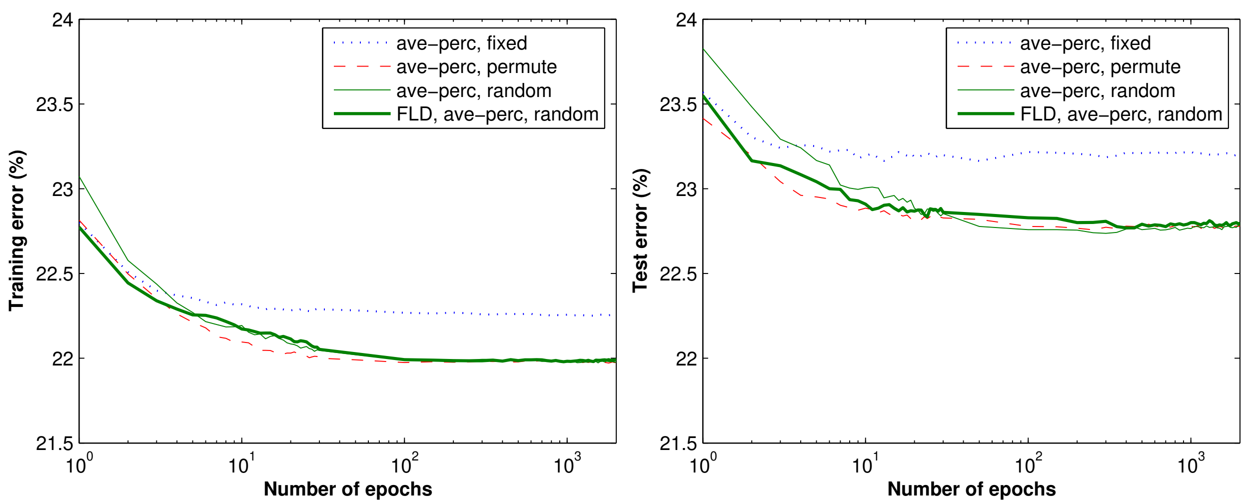

Figure 2: Training and test errors of the averaged-perceptron algorithm on the pima data set

ROMMA and aggressive ROMMA (Li & Long, 2002) perform miserably on most of the data sets we tried. The solution oscillates, especially when random sampling is used, and the training and test

[7] This is what was implied in (Freund & Schapire, 1999; Li & Long, 2002) although they did *preprocess* the training examples with a random permutation.

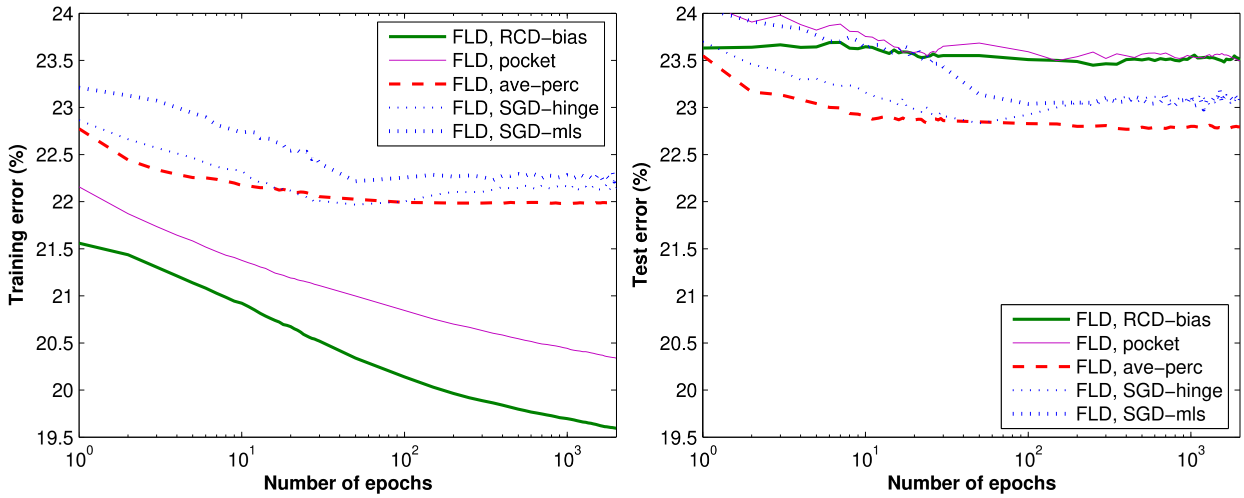

Figure 3: Training and test errors of several perceptron learning algorithms on the pima data set

errors keep high. They also have numerical problems when running for more than several hundreds of epochs, even with the normalized data. We thus exclude them from further comparisons.

Figure 3 presents the performance of the selected algorithms on the pima data set.[8] In the competition for low training errors, RCD-bias is clearly the best, and pocket follows. However, when the test error is concerned, the other three methods, especially ave-perc, are the winners. Tables 2 and 3 give the training and test errors on all the data sets at the end of the 2000 epochs. The errors of soft-SVM are also included. Again, we observe that RCD and RCD-bias achieve the lowest training errors for most data sets, but only achieve the lowest test errors for two artificial data sets, ringnorm and yinyang. The soft-SVM and ave-perc, both heavily regularized, overall achieve much better test errors. Since most real-world data sets may be noisy or contain errors, overfitting might be the reason for the inferior out-of-sample performance of the RCD algorithms.

The two artificial data sets, ringnorm and yinyang, have quite different nature. The former is 20-dimensional and inherently noisy, and the latter is 2-dimensional and has clean boundaries. However, overfitting seems to be no problems for these two data sets. Figure 4 shows that, approximately, the lower the training error, the lower the test error. We are still unclear for what problems the perceptron model will induce no or very little overfitting.

We should also note that pocket is much slower than other algorithms such as ave-perc and RCD. This is because every time a new weight vector is considered for the "pocket," mN multiplications have to be done for computing the training error. Thus pocket may actually go over all the examples many times in one epoch, especially when the initial weight has good quality. For example, for the pima data set, the average number of training error computations is 7463.7 for 2000 epochs if initialized with the zero vector, and 33170.0 if initialized with FLD.

4.3 Ensembles of Perceptrons

AdaBoost (Freund & Schapire, 1996) is probably the most popular algorithm among the boosting family that generates a linear combination of base hypotheses. It improves the accuracy of the base learner by gradually focusing on "hard" examples. At each iteration, AdaBoost gives the base

[8] We did not show the curves for RCD because they are very close to those of RCD-bias.

Table 2: Training errors (%) of several perceptron learning algorithms initialized with FLD

data set	RCD	RCD-bias	pocket	ave-perc	SGD-hinge	SGD-mls	soft-SVM
australian	10.12 ± 0.03	$\mathbf{9.98 \pm 0.03}$	10.81 ± 0.03	12.19 ± 0.03	14.11 ± 0.03	12.70 ± 0.04	14.33 ± 0.03
breast	$\mathbf{1.68 \pm 0.01}$	$\mathbf{1.68 \pm 0.01}$	1.86 ± 0.01	2.87 ± 0.02	2.66 ± 0.02	2.77 ± 0.02	2.70 ± 0.02
cleveland	$\mathbf{10.57 \pm 0.05}$	$\mathbf{10.62 \pm 0.05}$	12.07 ± 0.05	14.40 ± 0.06	14.31 ± 0.06	14.48 ± 0.06	14.74 ± 0.05
german	19.16 ± 0.04	$\mathbf{18.80 \pm 0.03}$	21.10 ± 0.03	21.31 ± 0.04	21.54 ± 0.04	22.18 ± 0.05	21.48 ± 0.04
heart	$\mathbf{9.48 \pm 0.05}$	$\mathbf{9.49 \pm 0.05}$	11.22 ± 0.05	13.64 ± 0.06	13.73 ± 0.06	13.82 ± 0.06	14.20 ± 0.06
ionosphere	3.88 ± 0.04	3.97 ± 0.04	$\mathbf{3.41 \pm 0.05}$	4.92 ± 0.06	4.55 ± 0.04	5.14 ± 0.05	6.95 ± 0.10
pima	$\mathbf{19.60 \pm 0.04}$	$\mathbf{19.60 \pm 0.03}$	20.34 ± 0.03	21.99 ± 0.04	22.15 ± 0.04	22.25 ± 0.04	22.09 ± 0.04
ringnorm	27.61 ± 0.07	$\mathbf{27.36 \pm 0.08}$	30.46 ± 0.07	35.49 ± 0.11	31.92 ± 0.09	34.52 ± 0.13	31.82 ± 0.09
sonar	2.56 ± 0.04	2.62 ± 0.04	$\mathbf{0.00 \pm 0.00}$	0.37 ± 0.02	2.23 ± 0.05	1.42 ± 0.06	11.58 ± 0.20
threenorm	$\mathbf{11.41 \pm 0.06}$	$\mathbf{11.39 \pm 0.06}$	13.53 ± 0.06	14.43 ± 0.06	14.23 ± 0.06	14.51 ± 0.06	14.47 ± 0.06
votes84	$\mathbf{1.32 \pm 0.02}$	$\mathbf{1.31 \pm 0.02}$	1.46 ± 0.02	2.42 ± 0.03	1.84 ± 0.03	2.48 ± 0.03	3.02 ± 0.04
yinyang	$\mathbf{15.33 \pm 0.05}$	$\mathbf{15.36 \pm 0.05}$	15.61 ± 0.05	19.10 ± 0.07	18.89 ± 0.08	19.03 ± 0.07	18.89 ± 0.08

(results within one standard error of the best are marked in bold)

Table 3: Test errors (%) of several perceptron learning algorithms initialized with FLD

data set	RCD	RCD-bias	pocket	ave-perc	SGD-hinge	SGD-mls	soft-SVM
australian	14.24 ± 0.12	13.92 ± 0.12	14.31 ± 0.12	$\mathbf{13.64 \pm 0.12}$	14.72 ± 0.12	13.87 ± 0.12	14.78 ± 0.12
breast	3.65 ± 0.07	3.61 ± 0.07	3.43 ± 0.06	3.36 ± 0.06	3.34 ± 0.06	$\mathbf{3.28 \pm 0.06}$	$\mathbf{3.22 \pm 0.06}$
cleveland	18.68 ± 0.22	18.57 ± 0.21	18.49 ± 0.21	$\mathbf{16.74 \pm 0.20}$	17.24 ± 0.20	$\mathbf{16.76 \pm 0.20}$	$\mathbf{16.72 \pm 0.20}$
german	24.45 ± 0.12	23.70 ± 0.13	25.24 ± 0.13	$\mathbf{23.24 \pm 0.12}$	23.66 ± 0.13	24.05 ± 0.13	23.64 ± 0.12
heart	18.13 ± 0.21	18.20 ± 0.22	17.63 ± 0.20	$\mathbf{16.51 \pm 0.20}$	16.70 ± 0.20	$\mathbf{16.49 \pm 0.20}$	$\mathbf{16.45 \pm 0.20}$
ionosphere	13.91 ± 0.17	14.72 ± 0.18	12.87 ± 0.18	12.76 ± 0.18	$\mathbf{12.45 \pm 0.17}$	12.63 ± 0.18	$\mathbf{12.57 \pm 0.17}$
pima	23.79 ± 0.14	23.50 ± 0.14	23.50 ± 0.14	$\mathbf{22.79 \pm 0.14}$	23.13 ± 0.13	23.07 ± 0.14	23.19 ± 0.14
ringnorm	35.83 ± 0.04	$\mathbf{35.65 \pm 0.04}$	36.59 ± 0.04	39.27 ± 0.08	36.01 ± 0.05	38.38 ± 0.10	35.70 ± 0.05
sonar	25.98 ± 0.29	26.20 ± 0.29	25.20 ± 0.25	25.09 ± 0.26	24.72 ± 0.28	24.90 ± 0.28	$\mathbf{23.89 \pm 0.27}$
threenorm	16.82 ± 0.03	16.86 ± 0.03	17.65 ± 0.04	16.14 ± 0.02	16.33 ± 0.02	16.18 ± 0.02	$\mathbf{16.08 \pm 0.02}$
votes84	5.21 ± 0.09	5.00 ± 0.10	5.24 ± 0.10	4.52 ± 0.10	5.17 ± 0.09	4.70 ± 0.11	$\mathbf{4.39 \pm 0.09}$
yinyang	$\mathbf{17.71 \pm 0.02}$	17.75 ± 0.02	17.74 ± 0.02	19.25 ± 0.02	19.12 ± 0.02	19.21 ± 0.02	19.21 ± 0.02

(results within one standard error of the best are marked in bold)

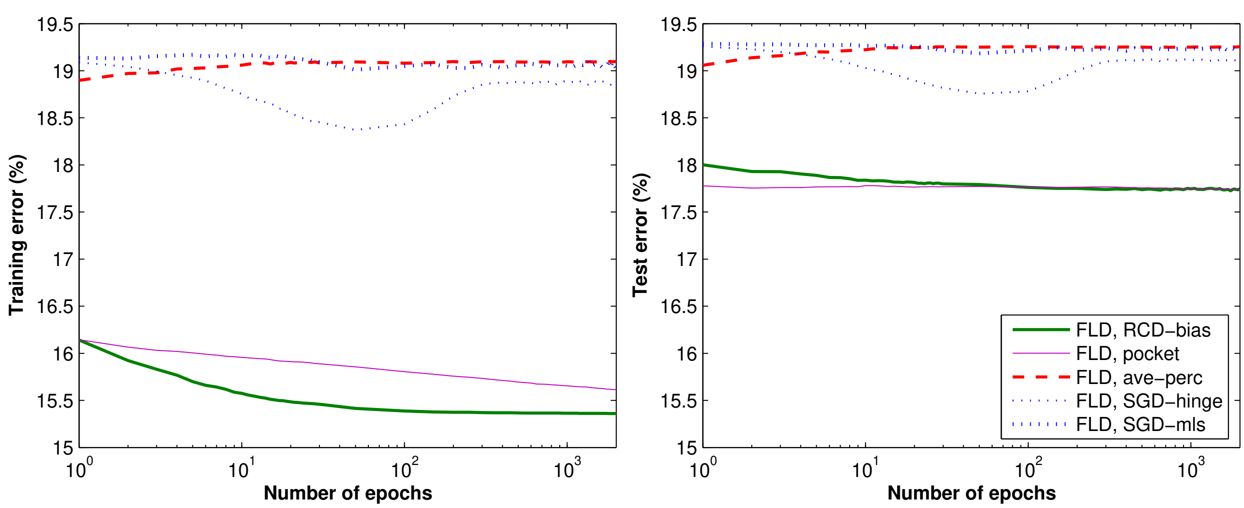

Figure 4: Training and test errors of several perceptron learning algorithms on the yinyang data set

Algorithm 6: The randomized averaged-perceptron algorithm with reweighting

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ and its sample weight $\{\varphi_i\}_{i=1}^N$; Number of epochs T.

```
t ← 1, γ_t ← 0, initialize w_t
for T × N trials do
    Randomly pick an example (x_k, y_k) with uniform probability
    if y_k ⟨w_t, x_k⟩ > 0 then {w_t correctly classifies (x_k, y_k)}
        γ_t ← γ_t + N φ_k
    else {w_t wrongly classifies (x_k, y_k)}
        t ← t + 1
        w_t ← w_{t−1} + N φ_k y_k x_k {using the sample weight}
        γ_t ← N φ_k
    end if
end for
return w = Σ_{i=1}^{t} γ_i w_i
```

learner a set of sample weights, and asks for a hypothesis that has a low weighted training error. Thus in order to work with AdaBoost, a base learner should be able to take care of weighted data.

Our RCD algorithms are ideal for working with AdaBoost, since they are designed to directly minimize the weighted training error. For the other algorithms, small modifications are needed to accommodate weighted data.

Take pocket for example. Given a set of sample weights $\{\varphi_i\}_{i=1}^N$, we may modify line 4 of Algorithm 1 to "randomly pick an example $(\mathbf{x}_k, y_k)$ according to the distribution defined by $\{\varphi_i\}$," and replace line 8 with the weighted training error $\sum_i \varphi_i \left[y_i \langle \mathbf{w}, \mathbf{x}_i \rangle \le 0 \right]$. We refer to this as *resampling*. Alternatively, we can keep picking examples with uniform probability, but modify quantities related to sample weights in a proper way. Here we change line 6 to "$\gamma \leftarrow \gamma + N\varphi_k$" and line 14 to "$\mathbf{w} \leftarrow \mathbf{w} + N\varphi_k y_k \mathbf{x}_k$." Of course we also modify line 8 as before. We refer to this as *reweighting*. The modified ave-perc with reweighting is shown in Algorithm 6. Note that the names reweighting and resampling have slightly different meanings from those by Freund and Schapire (1996).

Our experiments with AdaBoost (see Subsection 4.4 for settings) show that there is no significant difference between the resampling and reweighting methods. Since resampling usually requires $O[\log N]$ time to generate a random index according to the sample distribution, we prefer the reweighting method for its low computational overhead.

4.4 AdaBoost with Perceptrons

For the 12 data sets we use, 200 epochs seem sufficient for all perceptron learning algorithms to achieve a reasonable solution. Thus our base learners for AdaBoost are the perceptron learning algorithms with 200 epochs. We run AdaBoost up to 200 iterations. Often when the sample distribution becomes far away from the initial uniform one, the base learner fails to find a perceptron with a small training error because the cost function it tries to minimize becomes so different from the training error. When this happens, AdaBoost stops at some iteration earlier than 200. We record the training error, test error, as well as the number of iterations, at the end of the AdaBoost run. The numbers are averaged over 500 random splits of the original data set.

We tried resampling and reweighting with pocket, ave-perc, SGD-hinge, and SGD-mls. There was no significant difference in the training error, test error, or the number of AdaBoost iterations. We also tested the two initialization methods for perceptrons, zero vector and FLD, and found that there was no decisive advantage in one or the other. So we only list the results of the simplest setting, reweighting and initialization with the zero vector, in Table 4.

Table 4: Test errors (%) and number of iterations (#ite) of AdaBoost (the #ite of the first three algorithms is 200)

data set	RCD	RCD-bias	pocket	ave-perc	#ite	SGD-hinge	#ite	SGD-mls	#ite
australian	15.45 ± 0.12	15.49 ± 0.12	15.75 ± 0.12	$\mathbf{13.61 \pm 0.12}$	6.4	15.97 ± 0.13	12.3	14.00 ± 0.12	8.8
breast	$\mathbf{3.21 \pm 0.06}$	3.34 ± 0.06	3.41 ± 0.07	3.35 ± 0.06	3.2	$\mathbf{3.27 \pm 0.06}$	7.4	$\mathbf{3.24 \pm 0.06}$	4.7
cleveland	18.00 ± 0.21	18.22 ± 0.21	18.95 ± 0.20	$\mathbf{16.81 \pm 0.20}$	3.3	17.16 ± 0.20	10.0	$\mathbf{16.74 \pm 0.20}$	5.9
german	25.17 ± 0.13	25.37 ± 0.12	25.57 ± 0.13	$\mathbf{23.25 \pm 0.12}$	2.9	23.71 ± 0.13	9.2	23.96 ± 0.13	7.5
heart	17.60 ± 0.21	17.58 ± 0.22	18.94 ± 0.21	$\mathbf{16.55 \pm 0.20}$	3.0	16.95 ± 0.21	10.7	$\mathbf{16.54 \pm 0.20}$	5.3
ionosphere	$\mathbf{10.36 \pm 0.16}$	$\mathbf{10.30 \pm 0.16}$	11.65 ± 0.17	13.21 ± 0.17	6.9	11.71 ± 0.17	25.3	12.67 ± 0.17	11.0
pima	24.87 ± 0.14	24.79 ± 0.14	25.15 ± 0.14	$\mathbf{22.77 \pm 0.14}$	3.0	23.18 ± 0.14	4.9	23.01 ± 0.14	4.5
ringnorm	8.60 ± 0.05	12.22 ± 0.07	$\mathbf{7.12 \pm 0.04}$	39.29 ± 0.08	2.3	27.41 ± 0.22	29.2	38.32 ± 0.09	2.1
sonar	16.44 ± 0.25	$\mathbf{16.06 \pm 0.25}$	25.02 ± 0.27	25.77 ± 0.27	199.7	21.23 ± 0.27	195.5	25.37 ± 0.27	146.7
threenorm	$\mathbf{14.51 \pm 0.02}$	15.34 ± 0.03	14.95 ± 0.02	16.14 ± 0.02	2.7	16.27 ± 0.02	5.2	16.17 ± 0.02	3.9
votes84	$\mathbf{4.25 \pm 0.09}$	$\mathbf{4.24 \pm 0.09}$	4.54 ± 0.10	4.74 ± 0.10	7.0	4.78 ± 0.10	32.4	4.68 ± 0.10	7.8
yinyang	$\mathbf{3.95 \pm 0.03}$	$\mathbf{3.98 \pm 0.03}$	4.87 ± 0.02	19.25 ± 0.02	2.5	19.11 ± 0.02	2.6	19.23 ± 0.02	2.7

(results within one standard error of the best are marked in bold)

First we notice that algorithms not aiming at minimizing the training error, ave-perc, SGD-hinge, and SGD-mls, do not really benefit from working with AdaBoost. Their numbers of iterations are usually small, and the test errors are similar to those listed in Table 3.

AdaBoost with our RCD algorithms and pocket never early stops before the specified 200 iterations. The resulted ensembles based on RCD and RCD-bias always achieve the zero training error, and those based on pocket also almost always get the zero training error. For about half of the data sets, they also achieve the lowest test errors.

5 Conclusion

We proposed a family of new perceptron learning algorithms that directly optimizes the training error. The main ingredients are random coordinate descent (RCD) and an update procedure to efficiently minimize the training error along the descent direction. We also discussed several possible approaches to initialize the algorithms and to choose the descent directions. Our experimental results showed that RCD algorithms were efficient, and usually achieved the lowest training errors compared with several other perceptron learning algorithms. This property also makes them ideal base learners for AdaBoost.

We discussed the resampling and reweighting approaches to making several other perceptron algorithms work with AdaBoost. However, most of them optimize cost functions other than the training error, and do not benefit from aggregating. In contrast, the test error may be dramatically decreased if RCD algorithms and the pocket-ratchet algorithm are used with AdaBoost.

For noisy and/or high-dimensional data sets, regularized algorithms such as the averaged-perceptron algorithm and the soft-margin SVM may achieve better out-of-sample performance. Future work will be focused on regularizing RCD algorithms.

Acknowledgments

I wish to thank Yaser Abu-Mostafa, Hsuan-Tien Lin, and Amrit Pratap for many valuable discussions. This work was supported by the Caltech SISL Graduate Fellowship.

References

Bishop, C. M. (1995). *Neural networks for pattern recognition.* New York: Oxford University Press.

Breiman, L. (1996). *Bias, variance, and arcing classifiers* (Technical Report 460). Department of Statistics, University of California at Berkeley.

Breiman, L. (1998). Arcing classifiers. *The Annals of Statistics*, *26*, 801–824.

Chang, C.-C., & Lin, C.-J. (2001). *LIBSVM: A library for support vector machines*. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. *Machine Learning: Proceedings of the Thirteenth International Conference* (pp. 148–156).

Freund, Y., & Schapire, R. E. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, *37*, 277–296.

Friedman, J. H. (1999). Regularized discriminant analysis. *Journal of the American Statistical Association*, *84*, 165–175.

Gallant, S. I. (1990). Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, *1*, 179–191.

Hettich, S., Blake, C. L., & Merz, C. J. (1998). UCI repository of machine learning databases.

Holte, R. C. (1993). Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, *11*, 63–91.

Hsu, C.-W., Chang, C.-C., & Lin, C.-J. (2003). *A practical guide to support vector classification* (Technical Report). National Taiwan University.

Li, L., Pratap, A., Lin, H.-T., & Abu-Mostafa, Y. S. (2005). Improving generalization by data categorization. *Knowledge Discovery in Databases: PKDD 2005* (pp. 157–168). Berlin: Springer-Verlag.

Li, Y., & Long, P. M. (2002). The relaxed online maximum margin algorithm. *Machine Learning*, *46*, 361–387.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*, 386–408.

Rosenblatt, F. (1962). *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Washington, DC: Spartan.

Schapire, R. E., Freund, Y., Bartlett, P., & Lee, W. S. (1998). Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, *26*, 1651–1686.

Shavlik, J. W., Mooney, R. J., & Towell, G. G. (1991). Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning*, *6*, 111–143.

Vapnik, V. N. (1998). *Statistical learning theory*. Adaptive and Learning Systems for Signal Processing, Communications, and Control. New York: John Wiley & Sons.

Zhang, T. (2004). Solving large scale linear prediction problems using stochastic gradient descent algorithms. *Proceedings of the 21st International Conference on Machine Learning*. Omnipress.