Practical Methods for Exploiting Bounds on Change in the Margin

Andrew Guillory
Department of Computer Science and Engineering
University of Washington
guillory@cs.washington.edu

Jeff Bilmes
Department of Electrical Engineering
University of Washington
bilmes@ee.washington.edu

Abstract

We present methods for speeding up the training and evaluation of linear and kernel classifiers by exploiting bounds on change in the margin for data points. Assuming the classifier's decision boundary changes slowly, we show we can avoid recalculating the margin for many data points. We discuss several extensions and applications of this simple technique and show results applying it to gradient descent and stochastic subgradient descent.

1 Introduction

In this paper primarily we study binary classifiers of the form

$$y = sign(w^T x)$$

where y is a binary class, x is a data vector, and w a weight vector. We discuss extensions to nonlinear classifiers via the kernel trick in a later section. Given a set of training data x_i and y_i for $i = 1...N$, a common way to train a linear classifier is by minimizing the objective function

$$\sum_{i=1}^{N} l(y_i w^T x_i) + \frac{\lambda}{2} w^T w \tag{1}$$

where l is a loss function and λ is a regularization parameter. l is a function of a point's margin $y_i w^T x_i$ which is positive for correctly classified points and negative for incorrectly classified points. The magnitude of the margin is the distance from x_i to the decision boundary. Typically, l is chosen to be a convex upper-bound on the zero-one loss ($l(z) = 1$ if $z > 0$, 0 otherwise). A popular choice for l is the hinge loss function ($l(z) = \max(0, 1 - z)$). In this case, the solution to (1) is a soft margin Support Vector Machine (SVM) with a linear kernel function.

Many methods exist to optimize the objective function, perhaps best summarized by visiting `www.kernel-machines.org`, but also see the recent summary in (Shalev-Shwartz, Singer, & Srebro 2007). These include quadratic programing methods in either the primal or the dual space, and also subgradient based methods, which degenerate to gradient descent when the loss function upper-bound is everywhere differentiable. In this paper, we exploit the fact that whenever we are in a context where w changes slowly, it is possible to achieve significant speedups on evaluating the loss function over a data set. We consider and evaluate this in a training context under (sub)gradient descent based optimization, but see Section 5 for a list of other potential applications.

2 Bounds on change in the margin

In training and evaluating linear classifiers we often need to calculate a loss function or its gradient (or a subgradient) for a series of different w values. If w changes slowly, it's possible to speed up these computations by using $||w - w'||$ and $y_i w'^T x_i$ where w' is some previous weight vector to get upper and lower bounds on the margin for x_i. A lower bound is derived as

$$\begin{aligned} y_i w^T x_i &= y_i w^T x_i - y_i w'^T x_i + y_i w'^T x_i \\ &= y_i w'^T x_i + y_i((w - w')^T x_i) \\ &>= y_i w'^T x_i - ||w - w'||_2 \max_i ||x_i||_2 \end{aligned}$$

using the Cauchy-Schwarz inequality. An upper bound is similarly derived with the inequality reversed and a sign change giving

$$\begin{aligned} y_i w'^T x_i - ||w - w'||_2 \max_i ||x_i||_2 <= y_i w^T x_i \\ <= y_i w'^T x_i + ||w - w'||_2 \max_i ||x_i||_2 \end{aligned} \tag{2}$$

Assuming we have precomputed $y_i w'^T x_i$ for each x_i, and that we only need to know if the margin for x_i is above or below a threshold, we can use these bounds to potentially avoid computing $y_i w^T x_i$.

A geometric interpretation of (2) is that it is the maximum and minimum of the dot product between $w - w'$ and data vectors contained within an origin centered sphere with radius $\max_i ||x_i||_2$. Figure 1 shows this interpretation.

If we also sort the data by $y_i w'^T x_i$, we get monotonically increasing lower bounds and monotonically decreasing upper bounds with which we can quickly identify sub portions of the list with margins above or below a threshold. For certain loss functions we can then exactly calculate the loss and its gradient for the data set using stored summations.

Algorithms 1 and 2 illustrate this method for calculating hinge loss. Algorithm 2 computes the loss and subgradient for the data set and periodically calls Algorithm 1 to resort

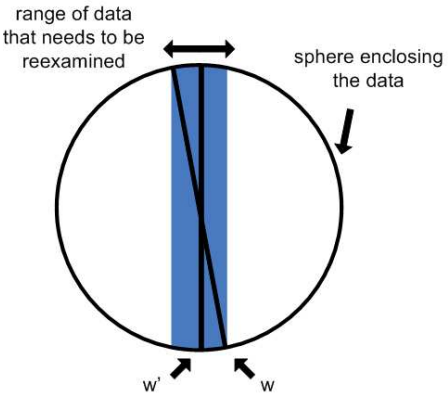

Figure 1: Geometric interpretation of the bound on change in the margin (2). The blue shaded region shows the range of data which needs to be reexamined for calculating the zero-one loss for w after sorting by the margin of w'.

Algorithm 1 SortByMargin(w)

1: $w' \Leftarrow w$
2: Calculate and store $y_i w'^T x_i$ for all i
3: Sort the data so that $y_i w'^T x_i <= y_{i+1} w'^T x_{i+1}$ for $i = 1...N-1$
4: $\texttt{Boundary} \Leftarrow$ smallest i such that $y_i w'^T x_i >= 1$
5: $\texttt{Sum}_i \Leftarrow \sum_{j<i} y_j x_j$

the data by the margin. The algorithm traverses the data set forwards and backwards from the point at which the data set crossed the 1 margin boundary for w'. In the forward traversal, if the lower bound on the margin becomes greater than or equal to 1 (Line 7), the remainder of the list has zero loss and the remaining points do not need to be examined. In the backward traversal, if the upper bound on the margin becomes less than 1 (Line 15), then the remainder of the list has non-zero loss and this loss can be calculated exactly using the summation stored by Algorithm 1. Lines 7 and 15 of Algorithm 1 use the values for $y_i w'^T x_i$ stored by Algorithm 2 and Line 5 uses a stored value for $\max_i ||x_i||_2$.

The performance benefit of the algorithm depends on a number of factors but primarily the expected magnitude of $||w - w'||_2$. Assuming the number of points for which $y_i w'^T x_i = 1$ (i.e. the number of points exactly on the 1 boundary of the margin) is constant, the number of points which need to be examined at each step will be constant for sufficiently small $||w - w'||_2$. Here we consider a point examined if we need to explicitly calculate the margin for that point (Lines 9 and 19). For larger $||w - w'||_2$, the performance benefit depends on how the data is distributed with respect to the margin. Intuitively, for a fixed $||w - w'||_2$, performance is better if more points were further from the 1 margin boundary when we last sorted.

The parameter `SortPeriod` controls how frequently the algorithm resorts the data. By setting this parameter proportional to $\log(N)$ we can ensure the algorithm is still in

Algorithm 2 EvaluateHingeLoss(w)

1: **if** `Iteration` mod `SortPeriod` = 0 **then**
2: **SortByMargin**(w)
3: **end if**
4: `Iteration++`
5: $m_\delta \Leftarrow ||w - w'||_2 \max_i ||x_i||_2$
6: **for** $i =$ `Boundary` to N **do**
7: **if** $y_i w'^T x_i - m_\delta >= 1$ **then**
8: Break
9: **else if** $y_i w^T x_i < 1$ **then**
10: $\nabla w += -y_i x_i$
11: $\texttt{HingeLoss} += 1 - y_i w^T x_i$
12: **end if**
13: **end for**
14: **for** $i =$ `Boundary` to 1 **do**
15: **if** $y_i w'^T x_i + m_\delta < 1$ **then**
16: $\nabla w += -\texttt{Sum}_i$
17: $\texttt{HingeLoss} += i - (w^T \texttt{Sum}_i)$
18: Break
19: **else if** $y_i w^T x_i < 1$ **then**
20: $\nabla w += -y_i x_i$
21: $\texttt{HingeLoss} += 1 - y_i w^T x_i$
22: **end if**
23: **end for**

the worst case $O(N)$. More specifically, we should set

$$\texttt{SortPeriod} = c\frac{\texttt{SortTime}}{\texttt{EvalTime}} \tag{3}$$

where `SortTime` and `EvalTime` are respectively estimates of the time needed to sort the data by the margin and evaluate the objective function respectively. c is a parameter which controls the worst case overhead. With $c = 10$ we expect that in the worst case we will spend $\frac{1}{11}$th of the time sorting. In our experiments we use a rough estimate of (3) that ignores constant factors

$$\texttt{SortPeriod} = c\frac{sdN + N\log(N) + d}{sdN + d} \tag{4}$$

where d is the dimensionality of the data set and s is the percentage of nonzero values in the data set. A more advanced implementation could estimate constant factors or compute empirical timing estimates for each data set.

3 Variations and Extensions

We found that a slightly more complicated rule for determining when to resort the data is useful. We resort the data if `SortPeriod` iterations have past since we last sorted the data and one of two conditions hold: either a) on the previous iteration we pruned less than half of the data, or b) (examples pruned since last sort) $-$ (iterations since last sort)(examples pruned last iteration) $>$ `SortCost`. Here we call an example pruned if we did not have to compute the margin for it directly. The first condition is straightforward and is meant to bootstrap the method. The second condition is intended to throttle the method when w is changing slowly. The quantity on the left

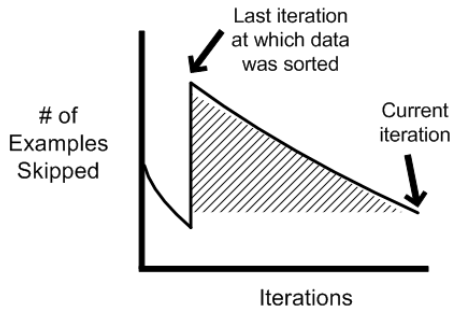

Figure 2: To decide when to resort the data by the margin, we compare the area of the shaded region to an estimate of the cost of resorting the data.

of the inequality of b) is the cumulative performance benefit of the last sort above the currently observed performance benefit. Figure 2 visualizes this quantity. `SortCost` is the relative cost of resorting the data, which can be estimated as

$$\texttt{SortCost} = \frac{\texttt{SortTime}}{\texttt{LossTime}}$$

where `LossTime` is an estimate of the time for computing loss for a point with sd non zero entries. We can again ignore constant factors for a simple, rough estimate as in (4), and we use this approach in our experiments. This rule has the effect of resorting the data less frequently when the number of pruned examples per iteration is high. In other words, when the distance between successive w vectors decreases, the sorting frequency also decreases.

It is not difficult to incorporate an offset term b so that $y = w^T x + b$. One easy way to add an offset term is to add an additional feature to every data point x_i which is always set to 1. The entry in w corresponding to this feature is then effectively an offset term, and our method can be used without modification. A drawback to this approach is that it increases both $||w - w'||_2$ (when b changes) and $\max_i ||x_i||_2$ (from the addition of a new feature) and therefore increases m_δ. Alternatively, we can separately store and sort the positive and negative examples by $y_i(w'^T x_i + b')$ and traverse each of these lists. In these traversals we replace the check in line 7 of Algorithm 2 with $y_i(w'^T x_i + b') - m_\delta + y_i(b - b') > 1$ and the check in line 15 with $y_i(w'^T x_i + b') + m_\delta + y_i(b - b') < 1$ where b is the current offset and b' was the offset when we last sorted. By seperately storing the positive and negative examples we can directly compute the change in margin due to the change in b.

The algorithm as written applies directly to high dimensional sparse data. However, for high dimensional sparse data storing the (dense) $\texttt{Sum}_i$ for every i can be expensive. In this case we can instead only store $\texttt{Sum}_{\texttt{Boundary}}$ and recursively compute $\texttt{Sum}_i$ for other i as we move backwards through the sorted data. This change could affect numerical accuracy, but we have not found it to be a practical problem with our double precision implementation.

A technique for speeding up loss and loss gradient calculations is to estimate these values from a random sub set of the data set (Shalev-Shwartz, Singer, & Srebro 2007). For random samples it may not make sense to sort the data, but we can still use the upper and lower bounds (2) to avoid calculating the margin for some points. Algorithm 3 shows this method.

Algorithm 3 EstimateHingeLossFromSample(w)

```
1: if Iteration mod SortPeriod = 0 then
2:    SortByMargin(w)
3: end if
4: Iteration++
5: w_δ ⇐ ||w − w'||_2
6: for i = 1 to NumSamples do
7:    sample x and y from the data set
8:    if y w'^T x − w_δ ||x||_2 > 1 then
9:       Continue
10:   else if y w'^T x + w_δ ||x||_2 < 1 then
11:      ∇w += −y_i x_i
12:      HingeLoss += 1 − y_i w^T x_i
13:   else if y_i w^T x_i < 1 then
14:      ∇w += −y_i x_i
15:      HingeLoss += 1 − y_i w^T x_i
16:   end if
17: end for
```

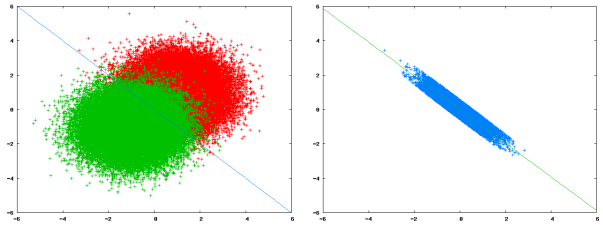

Figure 3: Left: test synthetic data with a hyperplane w. Right: points which need to be examined to calculate zero-one loss on the data set in Figure 3 after adding Gaussian noise ($\Sigma = .01I$) to w.

When called from this algorithm, the sorting step in Algorithm 1 is skipped. Lines 8 and 10 again can use stored values for $yw'^T x$ stored by Algorithm 1 and also stored values for $||x||_2$ for each x. Here we use separate norms for each data point in place of the $\max_i ||x_i||_2$ used in (2). Since we are not sorting the data it is no longer important for points to share bounds. Algorithm 3 is only useful for sample sizes great enough that the benefit outweighs the overhead of calculating $||w - w'||_2$. The potential performance benefit is also in general less than that of Algorithm 2: the method can potentially avoid computing many dot products, but in the best case the algorithm is still $O(\texttt{NumSamples})$.

4 Experiments

4.1 Synthetic Data

To illustrate our method we generated 100000 points from a mixture of two dimensional Gaussians, one Gaussian for the negative class centered at $(-1, -1)$ and one for the positive class centered at $(1, 1)$. Both Gaussians have covariance

Data Set	Training Size	Test Size	Dimensionality	Percent Non-Zero	λ
Covtype	522911	58101	54	22	1E-7
MNIST	60000	10000	780	19	1E-3
RCV1	20242	677399	47236	.16	1E-7
USPS	266079	75383	675	15	1E-5

Table 1: Data sets used in our experiments

matrices $\Sigma = I$. We set the initial hyperplane w to the plane best separating the Gaussians, $(1, 1)$, calculated $y_i w^T x_i$ for each point, and sorted the data by the margin. The left of Figure 3 shows the data set and w. We then added random Gaussian noise to w with zero mean and covariance $\Sigma = \sigma^2 I$ and calculated zero-one loss with a variation of Algorithm 2, recording the number of points which need to be examined. Here we consider a point examined if we need to explicitly calculate the margin for that point. The right of Figure 3 shows the points which need to be examined after adding noise with covariance $\Sigma = .01I$ ($\sigma = .1$) to w.

4.2 Gradient Descent

We tested our method on real world data by training linear classifiers to minimize a smooth approximation of hinge loss, Huber loss (Chapelle 2007).

$$l(z) = \begin{cases} 1 - z, & \text{for } z <= 1 - h; \\ \frac{(1+h-z)^2}{4h} & \text{for } 1 - h < z < 1 + h; \\ 0 & \text{for } z >= 1 + h; \end{cases} \quad (5)$$

Huber loss is the same as hinge loss for margin values greater than $1 + h$ and less than $1 - h$ but adds a quadratic segment within this region, making the loss function everywhere differentiable. As the parameter h goes to zero, the loss equals hinge loss.

We trained classifiers on 4 data sets with gradient descent on (1), using a variation of Algorithm 2 modified to use Huber loss and an offset term (b). Table 1 describes the data sets. We use the version of the Covtype data set used in (Shalev-Shwartz, Singer, & Srebro 2007) and (Joachims 2006). The USPS digit recognition data set is from (Tsang, Kwok, & Cheung 2005). Finally, we use the binary version of the RCV1 text classification and the MNIST digit recognition data set available from `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`. For the multi class MNIST we train on class 8. For each data set we use the λ value that gave the lowest error on a hold-out validation set trying all powers of ten between 10^2 and 10^{-10}. Where there is a tie, we use the largest λ value with the lowest error (on some data sets we found all λ below a threshold gave identical, optimal validation error).

We adapt Algorithm 2 for Huber loss by simply increasing m_δ by h and directly computing loss for points in the quadratic region of the loss function. In our experiments we use $h = .01$. We also experimented with $h = .1$ and $h = .001$ and found these values gave mostly comparable results, although for h larger than .01 the larger quadratic segment slows down our method.

We use a simple rule for adapting the gradient descent step size. We initialize the step size to a large value (we used 100) and monitor the objective function (1) value. If at any iteration the objective function increases we halve the step size. We found that with this method the objective function quickly decreases to a value (usually around .1) at which the optimization makes slow steady convergence. We stop the algorithm when the relative change in the objective function is less than ϵ.

We use the adaptive rule for determining when to resort described in Section 3 with $c = 10$. We also switch to standard gradient calculations if at any time our method fails to eliminate any points. We then switch back to our method the next time the data is resorted. This helps avoid wasted overhead when w is changing quickly. We also use the modification for sparse data described in Section 3.

Table 2 shows our results with CPU times (in seconds) for training with and without our sorting method. Reported times are minimums over 10 runs (other values are the same for every run as we use the same initialization, $w = 0$). Timing experiments were run on a machine with 2 GB of RAM and dual 2.4 GHz Intel Xeon processors (our code is single threaded). In all cases the sorting method helps, except for on the Covtype data set for $\epsilon = 10^{-4}$ and $\epsilon = 10^{-5}$. For $\epsilon = 10^{-4}$ on the Covtype data set, by chance the method stopped early. The sorting method helps more for the longer training runs (with smaller ϵ). We think this is an interesting benefit of our method. The choice of stopping criteria is less critical as our method makes the time per iteration proportional in some way to the progress in the optimization.

A potential criticism of these results is that the algorithm took many iterations to converge and that for faster converging algorithms our sorting method may not be as beneficial. However, we tried several different methods for adjusting the step size (using zero-one loss to determine when to reduce the step size, using a validation set to determine when to reduce the step size, and using a constant step size) and weren't able to achieve consistently faster convergence with these methods. We found Newton's method could give significantly faster convergence on some of our data sets, but this is not directly comparable as it requires inverting the Hessian at each iteration. Other second order methods may also give faster convergence, but we note that it is possible to combine our method with a second order method. The benefit will of course depend on how quickly w changes and how many iterations the method takes to converge. We also note that in the next section we give results with stochastic gradient descent which has been shown to be competitive with second order methods. We also found fixing $b = 0$ could sometimes speed things up, but this is not comparable

Data Set	ϵ	Iterations	Time w/o	Time w/	Speed Up	Train Error	Test Error
Covtype	1E-4	5	1.09	4.29	0.25	36.43	36.74
Covtype	1E-5	1390	111.99	115.98	0.97	23.32	23.41
Covtype	1E-6	7657	609.39	289.30	2.11	23.01	23.12
MNIST	1E-4	831	66.27	20.75	3.19	4.04	3.93
MNIST	1E-5	2119	166.12	34.58	4.80	3.94	3.73
MNIST	1E-6	4099	318.77	48.74	6.54	3.88	3.76
RCV1	1E-4	6031	243.01	100.80	2.41	1.64	3.73
RCV1	1E-5	7868	307.37	124.58	2.47	1.39	3.66
RCV1	1E-6	7868	313.71	131.39	2.39	1.39	3.66
USPS	1E-4	1164	273.69	103.15	2.65	2.95	3.50
USPS	1E-5	4200	960.44	181.38	5.30	2.81	3.39
USPS	1E-6	12311	2776	300	9.25	2.76	3.33

Table 2: Time with and without our method (in seconds) for training a Huber loss classifier with gradient descent for different convergence thresholds.

as it changes the objective function. Finally, the error rates reported here are not optimal and continued to decrease after we stopped training.

4.3 Stochastic Subgradient Descent

We also performed experiments using Algorithm 3 with Pegasos (Shalev-Shwartz, Singer, & Srebro 2007). We use a modified version of the code provided by the authors of (Shalev-Shwartz, Singer, & Srebro 2007). Pegasos is stochastic subgradient descent with two distinctions: 1) the learning rate for iteration t is set to $1/(\lambda t)$, and 2) w is projected into a ball of radius $1/\sqrt{\lambda}$ after each step. These changes give an improved theoretical convergence rate over standard stochastic gradient descent. The method has also been shown to be competitive with other state of the art methods for training linear SVMs (Shalev-Shwartz, Singer, & Srebro 2007).

We use the same data sets we used for gradient descent and the same λ values, except for the Covtype data set where we use $\lambda = 10^{-6}$ as in (Shalev-Shwartz, Singer, & Srebro 2007) and the RCV1 data set where we use $\lambda = 10^{-4}$ which was used in (Shalev-Shwartz, Singer, & Srebro 2007) for a different version of this data set. Pegasos's parameters are k, the number of examples to examine per iteration, and T, the number of iterations. We ran experiments for $k = 100$ and $k = 1000$ with T set to powers of ten up to 10^6 for $k = 100$ and up to 10^5 for $k = 1000$ ($kT <= 10^8$). We again use our adaptive rule from Section 3 with $c = 10$ for determining when to recompute the stored margins for the data set and switch to the standard gradient calculations if our method skips no examples.

We repeated each experiment 10 times with our method and 10 times without. Timing measurements reported are the minimum over the 10 trials. Table 3 shows our results. Except for on USPS, we report results for each k for T set to the smallest power of ten such that the absolute objective function value (averaged over all 20 trials) decreased by less than .01 compared to the previous power of ten. On USPS, the average objective function value did not converge according to this criteria before the maximum T we tried, so we report results for the maximum T.

Our method helps everywhere except for on the $RCV1$ data set where overhead made our method slower. As with the gradient descent results, our method was more beneficial on the longer runs. Our method also has more of an effect for the larger k, as here the overhead of computing $|w - w'|_2$ at each step is less significant. Unfortunately runs with larger k sometimes seem to converge more slowly relative to kT, although this was not uniformly the case. We also note that the data sets used here are all to some extent sparse and that we expect this method to be more effective with dense data sets.

We emphasize that the contribution of the paper is not a new method for training SVMs but rather a general method for speeding up loss function calculations that is potentially useful in many applications. The results here are meant to show the method is useful in reasonable applications.

5 Further Extensions and Applications

The general technique presented here has many potential extensions and applications. We close by outlining several of them here. We note that we don't claim our method will be useful in every case and certainly some of these applications may prove to be more useful than others.

Tighter Bounds The geometric interpretation of (2) makes it clear how one can achieve potentially tighter bounds on $|(w - w')^T x_i|$. Perhaps the simplest extension would be to use a sphere with a different center (for example the average of the data). In this case the bound would be $(w - w')^T c + ||w - w'||_2 r$ where c is the center of the sphere and $r = \max_i ||x_i - c||_2$. Other approaches could bound the data by a hypercube (or some other shape) instead of a sphere, use separate bounds for different clusters of the data, or even incorporate hierarchical representations of the data such as a kd-tree or metric tree.

Kernel Classifiers As usual for linear classifiers, the method can be extended to nonlinear classifiers by projecting the the training data x_i into a high dimensional feature space $\phi(x_i)$. If we can efficiently compute dot products in this feature space through a kernel function $k(x_i, x_j) = \phi(x_i)^T \phi(x_j)$, we can avoid explicitly representing w in this feature space and instead represent w as a weighted combination of the training vectors $w = \sum_i y_i \alpha_i \phi(x_i)$. We can calculate the distance between two hyperplanes w and

Data Set	k	T	Time w/o	Time w/	Speed Up
Covtype	100	10^5	12.45	11.62	1.07
Covtype	1000	10^5	141.49	116.52	1.21
MNIST	100	10^4	2.78	1.45	1.13
MNIST	1000	10^4	29.82	16.35	1.82
RCV1	100	10^4	3.44	14.80	0.23
RCV1	1000	10^3	3.70	4.32	0.86
USPS	100	10^6	228.69	183.55	1.25
USPS	1000	10^5	238.06	187.21	1.27

Table 3: Time with and without our method (in seconds) for training a Hinge loss classifier with Pegasos.

$w' = \sum_i y_i \alpha'_i \phi(x_i)$ using

$$||w - w'||_2 = \sqrt{\sum_{i,j} y_i y_j k(x_i, x_j)(\alpha_i \alpha_j + \alpha'_i \alpha'_j - 2\alpha_i \alpha'_j)} \tag{6}$$

In the nonlinear case, we can no longer stop early during the backwards traversal of the list in Algorithm 2, because $\sum_i l(y_i w^T \phi(x_i)) \neq l(y_i w^T \phi(\sum_i x_i))$ (stated differently we cannot efficiently store $\sum_i \phi(x_i)$). We can, however, still stop early in the forward direction (where loss is zero), and we can also stop early in either direction when calculating zero-one loss (in general we can avoid examining examples where the loss function is constant).

For kernel classifiers the performance characteristics of the algorithm are different. Simply calculating $||w - w'||_2$ takes $O(N_{sv}^2)$ where N_{sv} is the number of data points with nonzero α values, so the method only makes sense when $N_{sv} < N$. One situation in which our method could be particularly useful is when the entries in the kernel matrix needed for calculating (6) are cached (precomputed), but other kernel entries are not. It may also be possible to use a cheap upper bound on $||w - w'||_2$.

Other Loss Functions It's relatively straightforward to apply this technique to loss functions that are piecewise constant or piecewise linear with respect to the margin for certain ranges of the margin. It should also be possible to apply it to quadratic portions of loss functions by storing $\sum_{j<i} x_j x_j^T$ for each i, but this is costly for high dimensional data. It may also be possible to apply it to calculating fast upper bounds for general convex loss functions, by using a (data dependent) linear upper bound (the existence of which is guaranteed for convex losses).

Dual optimization Many standard SVM learning algorithms (Platt 1999) (Fan, Chen, & Lin 2005) (Joachims 1999) take a decomposition based approach to solving the dual of (1). These methods select small subsets of the training data over which to optimize at each training step–in the case of Sequential Minimal Optimization (Platt 1999) (Fan, Chen, & Lin 2005), minimal subsets of size two. A common heuristic for selecting these subsets is to take points whose Lagrange multipliers have slack in the direction of their partial derivatives. These partial derivatives are a simple function of the margin and our method could be used to avoid recalculating these partials, assuming we only want a few points with large partials. The standard approach to this problem is to explicitly maintain the partials for each x_i or each x_i in some subset of the data. For linear classifiers, however, maintaining the partials takes about as long as simply recalculating them, while recomputing is comparatively fast. Here our method may be more beneficial.

Many dual-based SVM solvers also use a heuristic called shrinking: if for several iterations a training example does not violate the optimality conditions it is removed from the data set for the remainder of the optimization (shrinking the data set). After optimizing over the shrunken data set, the removed points are usually then rechecked for optimality to ensure they still do not violate the optimality conditions. Our method could be accurately described as an exact, dynamic shrinking method and could be used to more aggressively reduce the data set size.

Sparse greedy approximation and reduced complexity classifiers A increasingly popular method for training SVMs is to, instead of optimizing (1) over the entire data set, optimize over a smaller subset of the data that is greedily selected according to some rule (Tsang, Kwok, & Cheung 2005). Similarly, the method in (Joachims 2006) for linear classifiers greedily adds constraints to a different but equivalent objective function. The rules used for selecting points to add to the reduced set are sometimes simple functions of the margin. In (Tsang, Kwok, & Cheung 2005) the point with the smallest margin is used, while the constraints in (Joachims 2006) correspond to sums over all of the misclassified points. Our method could be used to speed up finding these points exactly. Often to avoid computing the margin for every point these methods instead restrict the search for the next point to a random sample from data (e.g. in (Tsang, Kwok, & Cheung 2005) they use the point with the smallest margin from a sample of size 59). Here it may be possible to use Algorithm 3, although for small sample sizes it may be difficult to achieve speed ups.

Adaptation to changing data When w has not changed but instead the x_i have changed, we can also use a similar trick, assuming we have a bound on $||x_i - x'_i||_2$ where x'_i are the previous x_i

$$\begin{aligned} y_i w^T x'_i - ||w||_2 \max_i ||x_i - x'_i||_2 &<= y_i w^T x_i \\ &<= y_i w^T x'_i + ||w||_2 \max_i ||x_i - x'_i||_2 \end{aligned}$$

If both w and x_i have changed, we can combine the two bounds. This bound could be used to maintain a set of labels for a set of points as these points change, under the assumption that the points do not move quickly. For example,

the x_i could be changing sensor readings from a sensor network. Assuming the sensor readings vary smoothly and that the time resolution is high enough, we can avoid recalculating margins for points far away from the current decision boundary.

References

Chapelle, O. 2007. Training a support vector machine in the primal. *Neural Computation* 19(5):1155–1178.

Fan, R.-E.; Chen, P.-H.; and Lin, C.-J. 2005. Working set selection using second order information for training support vector machines. *JMLR* 6:1889–1918.

Joachims, T. 1999. Making large-scale support vector machine learning practical. *Advances in kernel methods: support vector learning* 169–184.

Joachims, T. 2006. Training linear SVMs in linear time. In *KDD06*, 217–226. New York, NY, USA: ACM Press.

Platt, J. C. 1999. Using analytic QP and sparseness to speed training of support vector machines. In *NIPS98*, 557–563. Cambridge, MA, USA: MIT Press.

Shalev-Shwartz, S.; Singer, Y.; and Srebro, N. 2007. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. In *ICML07*.

Tsang, I. W.; Kwok, J. T.; and Cheung, P.-M. 2005. Core vector machines: Fast SVM training on very large data sets. *JMLR* 6:363–392.