Imrovements to Platt's SMO Algorithm for SVM Classifier Design[1]

S.S. Keerthi | S.K. Shevade | C. Bhattacharyya & K.R.K. Murthy

ssk@guppy.mpe.nus.edu.sg shirish@csa.iisc.ernet.in cbchiru@csa.iisc.ernet.in murthy@csa.iisc.ernet.in

Technical Report CD-99-14

Control Division
Dept. of Mechanical and Production Engineering
National University of Singapore
Singapore-119260
Ph: (65)-874-4684

[1]A revised version of this report is under preparation for submission to a journal. We welcome any comments and suggestions for improving this report.

Abstract

This paper points out an important source of confusion and inefficiency in Platt's Sequential Minimal Optimization (SMO) algorithm that is caused by the use of a single threshold value. Using clues from the KKT conditions for the dual problem, two threshold parameters are employed to derive modifications of SMO. These modified algorithms perform significantly faster than the original SMO on all benchmark datasets tried.

1 Introduction

In the past few years, there has been a lot of excitement and interest in Support Vector Machines[16, 2] because they have yielded excellent generalization performance on a wide range of problems. Recently, fast iterative algorithms that are also easy to implement have been suggested[9,4,7,3,6]. Platt's Sequential Minimization Algorithm (SMO)[9,11] is an important example. A remarkable feature of SMO is that it is also extremely easy to implement. Comparative testing against other algorithms, done by Platt, have shown that SMO is often much faster and has better scaling properties.

In this paper we enhance the value of SMO even further. In particular, we point out an important source of confusion and inefficiency caused by the way SMO maintains and updates a single threshold value. Getting clues from optimality criteria associated with the KKT conditions for the dual, we suggest the use of two threshold parameters and devise two modified versions of SMO that remove the confusion associated with SMO and are much more efficient than the original SMO. Computational comparison on a number of benchmark datasets shows that the modifications perform significantly faster than the original SMO in most situations. The ideas mentioned in this paper can also be applied to the SMO regression algorithm[13]. We will report the results of that extension in another paper[12].

The paper is organized as follows. In section 2 we briefly discuss the SVM problem formulation, the dual problem and the associated KKT optimality conditions. We also point out how these conditions lead to proper criteria for terminating algorithms for designing SVM classifiers. Section 3 gives a short summary of Platt's SMO algorithm. In section 4 we point out the problem associated with the way SMO uses a single threshold value, and describe the modified algorithms in section 5. Computational comparison is done in section 6. The appendix gives the pseudo-codes for our

SMO modifications. These pseudo-codes are very similar to those for the SMO given by Platt in [9]. They are short, and, it is very easy to develop a working code for SVM design using them.

2 The SVM Problem and Optimality Conditions.

The basic problem addressed in this paper is the two category classification problem. The tutorial by Burges[2] gives a good overview of the solution of this problem using SVMs. Throughout the paper we will use x to denote the input vector of the support vector machine and z to denote the feature space vector which is related to x by a transformation, $z = \phi(x)$. As in all SVM designs, we do not assume ϕ to be known; all computations will be done using only the kernel function, $k(x, \hat{x}) = \phi(x) \cdot \phi(\hat{x})$, where "$\cdot$" denotes inner product in the z space. Let $\{(x_i, y_i)\}$ denote the training set, where x_i is the i-th input pattern and y_i is the corresponding target value; $y_i = 1$ means x_i is in class 1 and $y_i = -1$ means x_i is in class 2. Let $z_i = \phi(x_i)$. The optimization problem solved by the support vector machine is:

$$\min \frac{1}{2}\|w\|^2 + C\sum_i \xi_i \tag{1a}$$

$$\text{subject to}: \quad y_i(w \cdot z_i - b) \geq 1 - \xi_i \;\; \forall\, i; \;\; \xi_i \geq 0 \,\forall\, i \tag{1b}$$

This problem is referred to as the *primal* problem. The Lagrangian for this problem is:

$$L = \frac{1}{2}\|w\|^2 + C\sum_i \xi_i + \sum_i \alpha_i[1 - \xi_i - y_i(w \cdot z_i - b)] - \sum_i \pi_i \xi_i$$

The KKT optimality conditions are given by:

$$\nabla_w L = w - \sum_i \alpha_i y_i z_i = 0; \quad \frac{\partial L}{\partial b} = \sum_i \alpha_i y_i = 0; \quad \frac{\partial L}{\partial \xi_i} = C - \alpha_i - \pi_i = 0 \quad \forall i;$$
$$\alpha_i \geq 0, \;\; \alpha_i[1 - \xi_i - y_i(w \cdot z_i - b)] = 0, \;\; \pi_i \geq 0, \;\; \pi_i \xi_i = 0 \;\; \forall i.$$

We will refer to the α_i's as Lagrange multipliers. Let us define

$$w(\alpha) = \sum_i \alpha_i y_i z_i$$

Using Wolfe duality theory[16,2] it can be shown that the α_i's are obtained by solving the following *Dual* problem:

$$\max W(\alpha) = \sum_i \alpha_i - \frac{1}{2} w(\alpha) \cdot w(\alpha) \tag{2a}$$

$$\text{subject to } 0 \le \alpha_i \le C, \ \sum_i \alpha_i y_i = 0 \tag{2b}$$

Once the α_i's are obtained, the other primal variables, w, b, ξ and π can be easily determined by using the KKT conditions mentioned earlier. It is possible that the solution is non-unique; for instance, when all α_is take the boundary values of 0 and C, it is possible that b is not unique.

The numerical approach in SVM design is to solve the dual (instead of the primal) because it is a finite-dimensional optimization problem. (Note that $w(\alpha) \cdot w(\alpha) = \sum_i \sum_j y_i y_j \alpha_i \alpha_j k(x_i, x_j)$.) To derive proper stopping conditions for algorithms which solve the dual, it is important to write down the optimality conditions for the dual. The Lagrangian for the dual is:

$$\bar{L} = \frac{1}{2} w(\alpha) \cdot w(\alpha) - \sum_i \alpha_i - \sum_i \delta_i \alpha_i + \sum_i \mu_i(\alpha_i - C) - \beta \sum_i \alpha_i y_i$$

Define

$$F_i = w(\alpha) \cdot z_i - y_i = \sum_j \alpha_j y_j k(x_i, x_j) - y_i$$

The KKT conditions for the dual problem are:

$$\frac{\partial \bar{L}}{\partial \alpha_i} = (F_i - \beta) y_i - \delta_i + \mu_i = 0, \ \ \delta_i \ge 0, \ \ \delta_i \alpha_i = 0, \ \ \mu_i \ge 0, \ \ \mu_i(\alpha_i - C) = 0 \ \ \forall\, i$$

These conditions can be simplified by considering three cases.

Case 1. $\alpha_i = 0$

$$\delta_i \ge 0, \ \mu_i = 0 \ \ \Rightarrow (F_i - \beta) y_i \ge 0 \tag{3a}$$

Case 2. $0 < \alpha_i < C$

$$\delta_i = 0, \ \mu_i = 0 \ \ \Rightarrow (F_i - \beta) y_i = 0 \tag{3b}$$

Case 3. $\alpha_i = C$

$$\delta_i = 0, \ \mu_i \ge 0 \ \ \Rightarrow (F_i - \beta) y_i \le 0 \tag{3c}$$

Define the following index sets at a given α: $I_0 = \{i : 0 < \alpha_i < C\}$; $I_1 = \{i : y_i = 1, \alpha_i = 0\}$; $I_2 = \{i : y_i = -1, \alpha_i = C\}$; $I_3 = \{i : y_i = 1, \alpha_i = C\}$; and, $I_4 = \{i : y_i = -1, \alpha_i = 0\}$. *Note that these index sets depend on* α. The necessary conditions in (3a)-(3c) can be rewritten as

$$\beta \le F_i \ \forall\, i \in I_0 \cup I_1 \cup I_2 \ ; \ \ \beta \ge F_i \ \forall\, i \in I_0 \cup I_3 \cup I_4 \tag{4}$$

Define:

$$b_{\text{up}} = \min\{F_i : i \in I_0 \cup I_1 \cup I_2\} \ \text{ and } \ b_{\text{low}} = \max\{F_i : i \in I_0 \cup I_3 \cup I_4\} \tag{5}$$

Then optimality conditions will hold at some α iff

$$b_{\text{low}} \leq b_{\text{up}} \tag{6}$$

It is easy to see the close relationship between the threshold parameter b in the primal problem and the multiplier, β. *In particular, at optimality, β and b are identical.* Therefore, in the rest of the paper β and b will denote one and the same quantity.

We will say that an index pair (i,j) defines a *violation* at α if *one* of the following sets of conditions holds:

$$i \in I_0 \cup I_3 \cup I_4\ ,\quad j \in I_0 \cup I_1 \cup I_2 \ \text{ and }\ F_i > F_j \tag{7a}$$

$$i \in I_0 \cup I_1 \cup I_2\ ,\quad j \in I_0 \cup I_3 \cup I_4 \ \text{ and }\ F_i < F_j \tag{7b}$$

Note that optimality conditions will hold at α iff there does not exist any index pair (i,j) that defines a violation.

Since, in numerical solution, it is usually not possible to achieve optimality exactly, there is a need to define approximate optimality conditions. The condition (6) can be replaced by

$$b_{\text{low}} \leq b_{\text{up}} + 2\tau \tag{8}$$

where τ is a positive tolerance parameter. (In the pseudo-codes given in [9] and the appendix of this paper, this parameter is referred to as `tol`.) Correspondingly, the definition of violation can be altered by replacing (7a) and (7b) by:

$$i \in I_0 \cup I_3 \cup I_4\ ,\quad j \in I_0 \cup I_1 \cup I_2 \ \text{ and }\ F_i > F_j + 2\tau \tag{9a}$$

$$i \in I_0 \cup I_1 \cup I_2\ ,\quad j \in I_0 \cup I_3 \cup I_4 \ \text{ and }\ F_i < F_j - 2\tau \tag{9b}$$

Hereafter in the paper, when optimality is mentioned it will mean approximate optimality.

Since β can be placed halfway between b_{low} and b_{up}, approximate optimality conditions will hold iff there exists a β such that (3a)-(3c) are satisfied with a τ-margin, i.e.,

$$(F_i - \beta)y_i \geq -\tau \ \text{ if }\ \alpha_i = 0 \tag{10a}$$

$$|(F_i - \beta)| \leq \tau \ \text{ if }\ 0 < \alpha_i < C \tag{10b}$$

$$(F_i - \beta)y_i \leq \tau \ \text{ if }\ \alpha_i = C \tag{10c}$$

(10a)-(10c) are the approximate optimality conditions employed by Platt[9], Joachims[4] and others. In [6] we have argued the soundness of using the above approximate conditions as a stopping criterion for dual algorithms.

3 Platt's SMO Algorithm.

A number of algorithms have been suggested for solving the dual problem. Traditional quadratic programming algorithms such as the active set method[5] and interior point algorithms[13] are not suitable for large size problems because of the following reasons. First, they require that the kernel matrix $k(x_i, x_j)$ be computed and stored in memory. This requires extremely large memory. Second, these methods involve expensive matrix operations such as the Cholesky decomposition of a large submatrix of the kernel matrix. Third, for practitioners who would like to develop their own implementation of an SVM classifier, coding these algorithms is very difficult.

Several attempts have been made to develop methods that overcome some or all of these problems. Vapnik[15] made the observation that if the number of support vectors is small and they are known beforehand, then one could directly solve the reduced problem involving only the support vectors and thereby deal with significantly larger datasets. Since the support vectors are not known, a beginning set of vectors is chosen and chunked into memory and the resulting problem is solved. Then the remaining vectors are tested for optimality and those that violate are included. The process is repeated until a solution is obtained. This is referred to as the chunking algorithm.

If the number of support vectors itself is large then the chunking algorithm is also unsuitable. Osuna et.al.[8] suggested the use of only a subset of the vectors as a working subset and optimize on the corresponding α_i's while freezing the others. Though the arguments given by Osuna et.al. about the convergence of the algorithm are incorrect, it is expected that the algorithm will converge asymptotically as the number of steps goes to infinity. Joachims[4] has developed an efficient algorithm for SVM by building upon the basic idea given in [8].

Recently Platt suggested an algorithm[9] – Sequential Minimal Optimization (SMO) – that puts the subset selection in Osuna et.al's algorithm to the extreme by iteratively selecting subsets only of size 2. Note that, because of the presence of the equality constraint (see (2b)), at least two variables need to be chosen for optimization so as to take a step. Platt's computational experiments[9,11] have shown SMO to be very much faster than the chunking algorithm; it also scales much better as problem size grows. The SMO algorithm also fares better than Joachim's algorithm[4].

Let us give a brief description of the SMO algorithm. Because the working set is only of size 2 and the equality constraint can be used to eliminate one of the two Lagrange multipliers, the optimization problem at each step is a quadratic minimization in just one variable. It is straightforward to write down an analytic solution for it. Complete details are derived in [9]. The procedure, `takeStep` (which is a part of the pseudocode given there) gives a clear description of the implementation. There is no need to recall all details here. We only make one important comment on the role of the threshold parameter, β. As in [9] define the output error on the i-th pattern as

$$E_i = F_i - \beta$$

Consistent with the pseudocode of [9] let us call the indices of the two multipliers chosen for optimization in one step as i_2 and i_1. A look at the details in [9] shows that to take a step by varying α_{i_1} and α_{i_2}, we only need to know $E_{i_1} - E_{i_2} = F_{i_1} - F_{i_2}$. *Therefore a knowledge of the value of β is not needed to take a step.*

The method followed to choose i_1 and i_2 at each step is crucial for efficient solution of the problem. Based on a number of experiments Platt came up with a good set of heuristics. He employs a two loop approach: the outer loop chooses i_2; and, for a chosen i_2, the inner loop chooses i_1. The outer loop iterates over all patterns violating the optimality conditions, first only over those with Lagrange multipliers neither on the upper nor lower boundary, and once all of them are satisfied, over all patterns violating the optimality conditions to ensure that the problem has indeed been solved. Clearly, the algorithm spends a large fraction of its time adjusting the multipliers which take non-boundary values and only a small amount of time with the multipliers that take boundary values. Appropriately, therefore, Platt maintains and updates a cache for E_i values for indices i corresponding to non-boundary multipliers. The remaining E_i are computed as and when needed.

Let us now see how the SMO algorithm chooses i_1. The aim is to make a large increase in the objective function. Since it is expensive to try out all possible choices of i_1 and choose the one that gives the best increase in objective function, the index i_1 is chosen to maximize $|E_{i_2} - E_{i_1}|$. (If we define $\rho(t) = W(\alpha(t))$ where t is a real parameter that denotes the change in the values of $y_{i_1}\alpha_{i_1}$ and $-y_{i_2}\alpha_{i_2}$, and $\alpha(t)$ is the corresponding Lagrangian multiplier vector, then $|\rho'(0)| = |E_{i_1} - E_{i_2}|$.)

Since E_i is available in cache for non-boundary multiplier indices, only such indices are initially used in the above choice of i_1. If such a choice of i_1 does not yield sufficient progress, then the following steps are taken. Starting from a randomly chosen index, all indices corresponding to non-bound multipliers are tried as choices for i_1, one by one. If still sufficient progress is not possible, all indices are tried as choices for i_1, one by one, again starting from a randomly chosen index. Thus the choice of random seed affects the running time of SMO; see, for example, the computational costs mentioned in section 5.

Although a value of β is not needed to take a step, it is needed if (10a)-(10c) are employed for checking optimality. In the SMO algorithm β is updated after each step. If, after a step involving (i_1, i_2), one of α_{i_1}, α_{i_2} (or both) takes a non-boundary value then (3b) is exploited to update the value of β. In the rare case that this does not happen, there exists a whole interval, say, $[\beta_{\text{low}}, \beta_{\text{up}}]$, of admissible thresholds for α_{i_1} and α_{i_2}. In this situation SMO simply chooses: $\beta = (\beta_{\text{low}} + \beta_{\text{up}})/2$. In the next section we will see the problems caused by such a choice.

4 Problems with SMO Algorithm.

SMO is a carefully organized algorithm which has excellent computational efficiency. However, because of its way of computing and using a single threshold value it can get into a confused end state and can also become inefficient. Let us illustrate the first issue using a numerical example.

Example 1. Consider the following example where there are 3 patterns:

$$y_1 = -1, \;\; y_2 = y_3 = +1, \;\; C = \frac{1}{4}, \;\; \text{Kernel Matrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 2 & 6 \end{bmatrix}$$

Suppose we start from $\alpha_1 = \alpha_2 = \alpha_3 = 0$ (the usual point where SMO starts). Calculating F_i we get $F_1 = 1$, $F_2 = F_3 = -1$. All three indices violate the optimality conditions. (Note that $b_{\text{low}} = 1$ and $b_{\text{up}} = -1$; SMO uses $\beta = 0$ to check optimality conditions.) Suppose SMO chooses indices 1 and 2 for optimization, keeping α_3 fixed at 0. It is easy to check that this leads to the point, $\alpha_1 = \alpha_2 = C$, $\alpha_3 = 0$. At this new point we have $F_1 = 3/4$, $F_2 = -3/4$, $F_3 = -1/2$. Note that $b_{\text{low}} = -3/4$ and $b_{\text{up}} = -1/2$ and hence optimality conditions are satisfied. SMO chooses $\beta = (F_1 + F_2)/2 = 0$. If this value of β is used to check optimality, the third training pattern shows a violation of the optimality criterion employed by Platt (i.e., (10)), but actually there is no violation! Note that any β chosen from the interval, $[-3/4, \ -1/2]$ would have ensured the verification of (10).

This example clearly sums up our first concern. Because SMO constrains itself unnecessarily to a particular single choice of the threshold, β, it gets into trouble, especially at termination.

The issue raised here appears to be somewhat pathological since the presence of even a single index i with $0 < \alpha_i < C$ forces β to be unique and so there is really no serious problem. (Note that unless C takes certain extreme values, there is little possibility of not having an index i with $0 < \alpha_i < C$.) But we would like to point out that there is still a practical problem of inefficiency. At any instant, the SMO algorithm fixes β based on the current two indices which are being optimized. However, while checking whether the remaining examples violate optimality or not, it is quite possible that a different, shifted choice of β may do a better job. So, in the SMO algorithm it is quite possible that, even though α has reached a value where optimality is satisfied (i.e., (8)), SMO hasn't detected this because it has not identified the correct choice of β. It is also quite possible that, a particular index may appear to violate the optimality conditions because (10) is employed using an "incorrect" value of β although this index may not be able to pair with another to define a violation. In such a situation the SMO algorithm does an *expensive* and wasteful search looking for a second index so as to take a step. We believe that this is a major source of inefficiency in the SMO algorithm.

5 Modifications of the SMO Algorithm.

In this section we suggest two modified versions of the SMO algorithm, each of which overcomes the problems mentioned in the last section. As we will see in the computational evaluation of section 6, these modifications are almost always better than the original SMO algorithm and, in most situations they give quite a remarkable improvement in efficiency when tested on several benchmark problems.

In short, the modifications avoid the use of a single threshold value β and the use of (10) for checking optimality. Instead, two threshold parameters, b_{up} and b_{low} are maintained and (8) (or (9)) is employed for checking optimality. The two modifications are adequately described by the pseudo-codes given in the appendix. We only give some additional pointers that will help to give an easy understanding of the pseudo-codes. We assume that the reader is familiar with [9] and the pseudo-codes given there.

1. Suppose, at any instant, F_i is available for all i. Let i_low and i_up be indices such that

$$F_{i\_low} = b_{\text{low}} = \max\{F_i : i \in I_0 \cup I_3 \cup I_4\} \tag{11a}$$

$$F_{i\_up} = b_{\text{up}} = \min\{F_i : i \in I_0 \cup I_1 \cup I_2\} \tag{11b}$$

Then checking a particular i for optimality is easy. For example, suppose $i \in I_1 \cup I_2$. We only have to check if $F_i < F_{i\_low} - 2\tau$. If this condition holds then there is a violation, and, in that case SMO's `takeStep` procedure can be applied to the index pair, (i, i_low). Similar steps can be given for indices in the other sets. Thus, in our approach, the checking of optimality of i_2 and the choice of the second index, i_1 go hand in hand, unlike the original SMO algorithm. As we will see below, we compute and use (i_low, b_{low}) and (i_up, b_{up}) via an efficient updating process.

2. To be efficient, we would, like in the SMO algorithm, spend much of the effort altering α_i, $i \in I_0$; cache for F_i, $i \in I_0$ are maintained and updated to do this efficiently. And, when optimality holds for all $i \in I_0$, only then examine all indices for optimality.

3. Some extra steps are added to the `takeStep` procedure. After a successful step using a pair of indices, (i_2, i_1), let $\tilde{I} = I_0 \cup \{i_1, i_2\}$. We compute, *partially*, (i_low, b_{low}) and (i_up, b_{up}) using $\tilde{I}$ only (i.e., use only $i \in \tilde{I}$ in (11)). Note that these extra steps are inexpensive because cache for $\{F_i,\ i \in I_0\}$ is available and updates of F_{i_1}, F_{i_2} are easily done. A careful look shows that, since i_2 and i_1 have been just involved in a successful step, each of the two sets, $\tilde{I} \cap (I_0 \cup I_1 \cup I_2)$ and $\tilde{I} \cap (I_0 \cup I_3 \cup I_4)$, is non-empty; hence the partially computed (i_low, b_{low}) and (i_up, b_{up}) will not be null elements. Since i_low and i_up could take values from $\{i_2, i_1\}$ and they are used as choices for i_1 in the subsequent step (see item 1 above), we keep the values of F_{i_1} and F_{i_2} also in cache.

4. When working only with α_i, $i \in I_0$, i.e., a loop with `examineAll=0`, one should note that, if (8) holds at some point then it implies that optimality holds as far as I_0 is concerned. (This is because, as mentioned in item 3 above, the choice of b_{low} and b_{up} are influenced by all indices in I_0.) This gives an easy way of exiting this loop.

5. There are two ways of implementing the loop involving indices in I_0 only (`examineAll=0`).

Method 1. This is in line with what is done in SMO. Loop through all $i_2 \in I_0$. For each i_2, check optimality and, if violated, choose i_1 appropriately. For example, if $F_{i_2} < F_{i\_low} - 2\tau$ then there is a violation, and, in that case choose $i_1 = i\_low$.

Method 2. Always work with the worst violating pair, i.e., choose $i_2 = i\_low$ and $i_1 = i\_up$.

Depending on which one of these methods is used, we call the resulting overall modification of SMO as SMO-Modification 1 and SMO-Modification 2.

6. When optimality on I_0 holds, as already said we come back to check optimality on all indices (`examineAll=1`). Here we loop through all indices, one by one. Since $(b_{\rm low}, i\_low)$ and $(b_{\rm up}, i\_up)$ have been partially computed using I_0 only, we update these quantities as each i is examined. For a given i, F_i is computed first and optimality is checked using the current $(b_{\rm low}, i\_low)$. For example, if $i \in I_1 \cup I_2$ and $F_i < b_{\rm low} - 2\tau$, then there is a violation, in which case we take a step using (i, i_low). On the other hand, if there is no violation, then $(i\_up, b_{\rm up})$ are modified using F_i, i.e, if $F_i < b_{\rm up}$ then we do: $i\_up := i$ and $b_{\rm up} := F_i$.

7. Suppose we do as described above. What happens if there is no violation for any i in a loop having `examineAll=1`1? Can we conclude that optimality holds for all i? The answer is: *YES*. This is easy to see from the following argument. Suppose, by contradiction, there does exist one (i, j) pair such that they define a violation, i.e., they satisfy (9). Let us say, $i < j$. Then j would not have satisfied the optimality check in the above described implementation because F_i would have, earlier than j is seen, affected either the calculation of $b_{\rm low}$ and/or $b_{\rm up}$ settings. In other words, even if i is mistakenly taken as having satisfied optimality earlier in the loop, j will be detected as violating optimality when it is analysed. Only when (8) holds it is possible for all indices to satisfy the optimality checks. Furthermore, when (8) holds and the loop over all indices has been completed, the true values of $b_{\rm up}$ and $b_{\rm low}$, as defined in (5) would have been computed since all indices have been encountered.

6 Computational Comparison.

In this section we compare the performance of our modifications against the original SMO algorithm. We implemented all these methods in Fortran and ran them using f77 on a 200 MHz Pentium machine. The value, $\tau = 0.001$ was used for all experiments. The following standard problems were used in our testing: Wisconsin Breast Cancer data[1, 17]; Two Spirals data[14]; Checkers

Data Set	σ^2	n	m
Wisconsin Breast Cancer	4.0	9	683
Two Spirals	0.5	2	195
Checkers	0.5	2	465
Adult-1	10.0	123	1605
Adult-4	10.0	123	4781
Adult-7	10.0	123	16100
Web-1	10.0	300	2477
Web-4	10.0	300	7366
Web-7	10.0	300	24692

Table 1: Data Set Properties.

data[5]; UCI Adult data[10]; and Web page classification data[10,4]. Except for Checkers data, for which we created a random set of points on a 4×4 checkers grid (see [6]), all other data sets were downloaded from the sites mentioned in the above references and were used in full for training. i.e., no division into training/validation/test sets was made. In the case of Adult data set, the inputs are represented in a special binary format, as used by Platt in his testing of SMO. To study scaling properties as training data grows, Platt did staged experiments on the Adult and Web data. We have used only the data from the first, fourth and seventh stages. The gaussian kernel,

$$k(x_i, x_j) = \exp(-0.5\|x_i - x_j\|^2/\sigma^2)$$

was used in all experiments. The σ^2 values employed, together with n, the dimension of the input, and m, the number of training points, are given in Table 1. The σ^2 values given in the table were chosen as follows. For the Adult and Web data the σ^2 values are the same as those used by Platt in his experiments on SMO; for other data, we chose σ^2 suitably to get good generalization.

When a particular method is used for SVM design, the value of C is usually unknown, and it has to be chosen by trying a number of values and using a validation set. Therefore, good performance of a method over a range of C values is important. Therefore for each problem we have tested the algorithms over an appropriate range of C values.

The cost of updating the cache for F_i is the dominant part of the computational cost. Hence the total number of kernel evaluations is a very good indicator of the computing cost. Since such a measure is pretty much independent of the computing environment used, it is easy for others developing new algorithms to compare their methods against the ones studied in this paper, without actually running these methods again. In Tables 2-10 we have given the total number of kernel evaluations for the various problems tried. To point out the effect of the choice of random seed on the cost associated with the original SMO algorithm, we have reported costs for two random seeds. (We haven't done this for the Web data since, for that data, change of random seed had no effect on the computational cost.) Our SMO modifications do not require any random seed.

It is very clear that the modifications outperform the original SMO algorithm. In many situations the improvement in efficiency is remarkable. Between the two modifications, the second one fares better overall.

7 Conclusion.

In this paper we have pointed out an important source of inefficiency in Platt's SMO algorithm that is caused by the operation with a single threshold value. We have suggested two modifications of the SMO algorithm that overcome the problem by efficiently maintaining and updating two threshold parameters. Our computational experiments show that these modifications speed up the SMO algorithm considerably in many situations. Platt has already established the SMO algorithm to be one of the fastest algorithms for SVM design. The modified versions of SMO presented in this paper enhance the value of the SMO algorithm even further. The ideas mentioned in this paper for SVM classification can also be extended to the SMO regression algorithm[13]. We will report the results of that extension in another paper[12].

C	SMO Random Seed 1	SMO Random Seed 2	SMO Modification 1	SMO Modification 2
0.02	47.855	36.822	1.193	3.500
0.04	2.936	2.671	2.005	1.725
0.06	2.114	2.648	2.035	1.950
0.10	1.627	1.824	1.860	1.680
0.20	1.647	2.045	1.775	1.404
0.40	1.720	1.372	1.362	1.255
0.50	1.613	1.618	1.265	1.183
0.70	1.653	1.377	1.339	1.065
1.00	1.531	1.560	1.474	1.210
2.00	1.516	1.686	1.331	1.019
3.00	1.625	1.690	1.314	0.990

Table 2: Wisconsin Breast Cancer data: Number of Kernel evaluations $\times 10^{-6}$

References

[1] R. Bennett and O.L. Mangasarian, Robust linear programming discrimination of two linearly inseparable sets, *Optimization Methods and Software*, Vol.1, 1992, pp.23-34.

[2] C.J.C. Burges, A tutorial on support vector machines for pattern recognition, *Data Mining and Knowledge Discovery*, Vol.2, Number 2, 1998.

[3] T.T. Friess, Support vector networks: The kernel adatron with bias and soft-margin, Tech. Report, The University of Sheffield, Dept. of Automatic Control and Systems Engineering, Sheffield, England, 1998.

[4] T. Joachims, Making large-scale support vector machine learning practical, in B. Schölkopf, C. Burges, A. Smola. *Advances in Kernel Methods: Support Vector Machines*, MIT Press, Cambridge, MA, December 1998.

[5] L. Kaufman, Solving the quadratic programming problem arising in support vector classification, in B. Schölkopf, C. Burges, A. Smola. *Advances in Kernel Methods: Support Vector*

C	SMO Random Seed 1	SMO Random Seed 2	SMO Modification 1	SMO Modification 2
0.02	0.093	0.100	0.096	0.096
0.04	0.110	0.134	0.097	0.097
0.06	0.135	0.469	0.097	0.097
0.10	0.116	0.117	0.097	0.097
0.20	0.099	0.148	0.097	0.097
0.40	0.255	0.179	0.172	0.171
0.50	0.198	0.241	0.284	0.323
0.70	0.457	0.445	0.240	0.210
1.00	0.559	0.548	0.571	0.443
2.00	3.343	6.055	2.900	1.520
3.00	4.128	2.911	2.905	1.710
10.0	3.343	4.413	3.043	1.690

Table 3: Two Spirals data: Number of Kernel evaluations $\times 10^{-6}$

C	SMO Random Seed 1	SMO Random Seed 2	SMO Modification 1	SMO Modification 2
1.0	1.180	1.012	0.810	0.670
5.0	1.320	1.165	1.275	1.044
10.0	1.387	1.624	1.453	1.113
50.0	3.241	2.584	2.353	1.739
10^2	6.027	5.038	4.578	2.119
5×10^2	20.187	9.970	7.556	4.607
10^3	17.518	16.943	7.321	8.569
5×10^3	62.729	96.136	49.270	38.660
10^4	60.202	68.392	52.000	17.274
5×10^4	34.093	44.377	28.380	26.450

Table 4: Checkers data: Number of Kernel evaluations $\times 10^{-6}$

C	SMO Random Seed 1	SMO Random Seed 2	SMO Modification 1	SMO Modification 2
0.10	1.129	1.077	0.324	0.325
0.20	0.917	1.042	0.569	0.570
0.40	0.752	0.751	0.546	0.545
0.50	0.846	0.734	0.543	0.539
0.70	0.834	0.944	0.545	0.541
1.00	0.723	0.728	0.547	0.647
2.00	0.891	0.868	0.630	0.610
3.00	0.888	0.863	0.727	0.696
5.00	1.053	1.082	0.845	0.749
10.0	2.041	2.089	1.428	1.198
20.0	3.921	3.904	2.463	1.946
50.0	7.915	8.446	4.740	3.402
100.0	13.315	12.358	6.543	4.502
200.0	16.656	19.692	9.382	5.588
500.0	24.019	25.676	14.715	6.942

Table 5: Adult 1 data: Number of Kernel evaluations $\times 10^{-7}$

C	SMO Random Seed 1	SMO Random Seed 2	SMO Modification 1	SMO Modification 2
0.10	9.812	9.290	6.856	5.819
0.20	10.074	8.145	4.506	4.499
0.40	7.739	7.745	4.336	4.330
0.50	9.472	7.657	5.233	4.341
0.70	6.706	6.700	4.388	4.352
1.00	6.715	7.588	4.498	4.467
2.00	7.163	7.200	5.034	4.923
3.00	7.901	6.939	5.638	5.446
5.00	8.980	9.631	7.204	5.880
10.0	16.431	15.086	11.711	9.310
20.0	33.564	33.288	20.864	15.386
50.0	77.886	71.813	42.554	29.409
100.0	128.383	126.491	66.100	48.257
200.0	207.332	217.001	112.869	78.402
500.0	384.589	393.042	216.034	122.202

Table 6: Adult 4 data: Number of Kernel evaluations $\times 10^{-7}$

C	SMO Random Seed 1	SMO Random Seed 2	SMO Modification 1	SMO Modification 2
0.10	193.680	206.289	31.590	31.590
0.40	90.758	90.648	45.742	45.720
1.00	90.091	71.677	47.993	47.490
5.00	103.471	103.198	77.732	70.700
20.0	370.405	380.250	224.689	153.932

Table 7: Adult 7 data: Number of Kernel evaluations $\times 10^{-7}$

C	SMO	SMO Modification 1	SMO Modification 2
0.10	0.679	0.633	0.632
0.20	1.197	0.755	0.727
0.40	1.215	0.893	0.971
0.50	1.013	0.912	0.918
0.70	0.963	1.070	1.032
1.00	1.206	1.063	0.988
2.00	1.365	1.260	1.142
3.00	1.449	1.308	1.270
5.00	1.252	1.178	1.242
10.0	1.421	1.397	1.348
20.0	1.570	1.364	1.221
50.0	1.621	1.373	1.363
100.0	1.666	1.301	1.250
200.0	1.336	1.366	1.257
500.0	1.378	1.442	1.420

Table 8: Web 1 data: Number of Kernel evaluations $\times 10^{-7}$

C	SMO	SMO Modification 1	SMO Modification 2
0.10	6.714	4.126	3.578
0.20	7.686	3.937	4.010
0.40	9.155	4.333	4.972
0.50	8.808	5.102	5.185
0.70	9.146	6.199	5.059
1.00	8.154	6.156	6.061
2.00	8.494	7.436	6.594
3.00	9.887	8.331	9.092
5.00	10.826	8.749	9.464
10.0	9.685	11.193	12.402
20.0	12.162	9.795	10.265
50.0	10.733	10.973	10.305
100.0	12.155	11.314	11.821
200.0	12.169	10.177	10.907
500.0	12.792	11.121	10.661

Table 9: Web 4 data: Number of Kernel evaluations $\times 10^{-7}$

C	SMO	SMO Modification 1	SMO Modification 2
0.10	134.575	33.218	38.725
0.20	143.039	40.536	46.877
0.40	132.594	48.952	40.187
0.50	115.698	50.801	45.382
0.70	106.148	42.707	53.282
1.00	120.296	49.265	48.310
2.00	146.941	56.402	64.235
3.00	112.226	58.735	69.329
5.00	115.890	82.549	69.308
10.0	113.551	85.744	86.436
20.0	103.516	95.809	93.830
50.0	129.473	93.215	89.486
100.0	136.820	91.090	110.006
200.0	148.265	93.362	94.349
500.0	125.315	94.553	105.505

Table 10: Web 7 data: Number of Kernel evaluations $\times 10^{-7}$

Machines, MIT Press, Cambridge, MA, December 1998.

[6] S.S. Keerthi, S.K. Shevade, C. Bhattacharyya and K.R.K. Murthy, A fast iterative nearest point algorithm for support vector machine classifier design, Tech. Report TR-ISL-99-03, Intelligent Systems Lab, Dept. of Computer Science and Automation, Indian Institute of Science, Bangalore, India, March 1999. See: *http://guppy.mpe.nus.edu.sg/˜mpessk*

[7] O.L. Mangasarian and D.R. Musicant, Successive overrelaxation for support vector machines, Tech. Report, Computer Sciences Dept., University of Wisconsin, Madison, WI, USA, 1998.

[8] E. Osuna, R. Freund and F. Girosi, An improved training algorithm for support vector machines, in J. Principe, L. Giles, N. Morgan and E. Wilson, editors, *Neural Networks for Signal Processing VII – Proceedings of the 1997 IEEE Workshop*, pp.276-285, New York, 1997, IEEE.

[9] J.C. Platt, Fast training of support vector machines using sequential minimal optimization, in B. Schölkopf, C. Burges, A. Smola. *Advances in Kernel Methods: Support Vector Machines*, MIT Press, Cambridge, MA, December 1998.

[10] J.C. Platt, Adult and Web Datasets. *http://www.research.microsoft.com/˜jplatt*

[11] J.C. Platt, Using sparseness and analytic QP to speed training of support vector machines, in *Advances in Neural Information Processing Systems 11*, M.S. Kearns, S.A. Solla and D.A. Cohn, eds., MIT Press, 1999.

[12] S.K. Shevade, S.S. Keerthi, C. Bhattacharyya and K.R.K. Murthy, Improved versions of the SMO algorithm for SVM regression, Tech. Rept., Dept. of Mech. and Prod. Engrg., National University of Singapore, Singapore, Aug 1999, Under Preparation.

[13] A.J. Smola and B. Schölkopf, A tutorial on support vector regression, *NeuroCOLT Technical Report TR-1998-030*, Royal Holloway College, London, UK, 1998.

[14] Two Spirals Data. *ftp://ftp.boltz.cs.cmu.edu/pub/neural-bench/bench/two-spirals-v1.0.tar.gz*

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

[16] V. Vapnik, *The Nature of Statistical Learning Theory.* Springer-Verlag, New York, 1995.

[17] Wisconsin Breast Cancer Data.
ftp://128.195.1.46/pub/machine-learning-databases/breast-cancer-wisconsin/

Appendix. Pseudo-Codes for Modified SMO Algorithms.

The pseudo-codes for the improved SMO algorithms are presented below. Here, statements starting with "%" denote comments.

```
  target = desired output vector
  point = training point matrix
  fcache = cache vector for Fi values
% Note: Our definition of Fi is different from the Ei in Platt's SMO
% algorithm. Our Fi does not subtract any threshold.

  procedure takeStep(i1,i2)
% Much of this procedure is same as that in Platt's SMO pseudo-code.
    if (i1 == i2) return 0
    alph1 = Lagrange multiplier for i1
    y1 = target[i1]
    F1 = fcache[i1]
    s = y1*y2
    Compute L, H
    if (L == H)
      return 0
    k11 = kernel(point[i1],point[i1])
    k12 = kernel(point[i1],point[i2])
    k22 = kernel(point[i2],point[i2])
    eta = 2*k12-k11-k22
    if (eta < 0)
```

```
    {
      a2 = alph2 - y2*(F1-F2)/eta
      if (a2 < L) a2 = L
      else if (a2 > H) a2 = H
    }
    else
    {
      Lobj = objective function at a2=L
      Hobj = objective function at a2=H
      if (Lobj > Hobj+eps)
        a2 = L
      else if (Lobj < Hobj-eps)
        a2 = H
      else
        a2 = alph2
    }
    if (|a2-alph2| < eps*(a2+alph2+eps))
      return 0
    a1 = alph1+s*(alph2-a2)
    Update weight vector to reflect change in a1 & a2, if linear SVM
    Update fcache[i] for i in I_0 using new Lagrange multipliers
    Store a1 and a2 in the alpha array
% The update below is simply achieved by keeping and updating information
% about alpha_i being at 0, C or in between them. Using this together with
% target[i] gives information as to which index set i belongs.
    Update I_0, I_1, I_2, I_3 and I_4
% Compute updated F values for i1 and i2...
    fcache[i1] = F1 + y1*(a1-alph1)*k11 + y2*(a2-alph2)*k12
    fcache[i2] = F2 + y1*(a1-alph1)*k12 + y2*(a2-alph2)*k22
    Compute (i_low, b_low) and (i_up, b_up) by applying equations (11a) and
```

```
    (11b), using only i1, i2 and indices in I_0; see item 3 of section 5.
    return 1
  endprocedure

  procedure examineExample(i2)
    y2 = target[i2]
    alph2 = Lagrange multiplier for i2
    if (i2 is in I_0)
    {
      F2 = fcache[i2]
    }
    else
    {
      compute F2 = F_i2 and set fcache[i2] = F2
% Update (b_low, i_low) or (b_up,i_up) using (F2,i2)...
      if ((i2 is in I_1 or I_2) && (F2 < b_up) )
        b_up = F2, i_up = i2
      else if ((i2 is in I_3 or I_4) && (F2 > b_low) )
        b_low = F2, i_low = i2
    }
% Check optimality using current b_low and b_up and, if
% violated, find an index i1 to do joint optimization with i2...
    optimality = 1
    if (i2 is in I_0, I_1 or I_2)
    {
      if (b_low-F2 > 2*tol)
        optimality = 0, i1 = i_low
    }
    if (i2 is in I_0, I_3 or I_4)
    {
```

```
      if (F2-b_up > 2*tol)
        optimality = 0, i1 = i_up
    }
    if (optimality == 1)
      return 0
% For i2 in I_0 choose the better i1...
    if (i2 is in I_0)
    {
      if (b_low-F2 > F2-b_up)
        i1 = i_low
      else
        i1 = i_up
    }
    if takeStep(i1,i2)
       return 1
    else
       return 0
  endprocedure

  main routine for Modification 1:
    initialize alpha array to all zero
    initialize b_up = -1, i_up to any one index of class 1
    initialize b_low = 1, i_low to any one index of class 2
    set fcache[i_low] = 1 and fcache[i_up] = -1
    numChanged = 0;
    examineAll = 1;
    while (numChanged > 0 | examineAll)
      {
        numChanged = 0;
        if (examineAll)
```

```
        {
          loop I over all training examples
            numChanged += examineExample(I)
        }
        else
        {
          loop I over I_0
            numChanged += examineExample(I)
% It is easy to check if optimality on I_0 is attained...
            if (bup > blow - 2*tol) at any I
              exit the loop after setting numChanged = 0
        }
        if (examineAll == 1)
          examineAll = 0
        else if (numChanged == 0)
          examineAll = 1
      }



  main routine for Modification 2:
    initialize alpha array to all zero
    initialize b_up = -1, i_up to any one index of class 1
    initialize b_low = 1, i_low to any one index of class 2
    set fcache[i_low] = 1 and fcache[i_up] = -1
    numChanged = 0;
    examineAll = 1;
    while (numChanged > 0 | examineAll)
      {
        numChanged = 0;
        if (examineAll)
```

```
        {
          loop I over all training examples
            numChanged += examineExample(I)
        }
        else
% The following loop is the only difference between the two SMO
% modifications. Whereas, in modification 1, the inner loop selects
% i2 from I_0 sequentially, here i2 is always set to the current
% i_low and i1 is set to the current i_up; clearly, this corresponds
% to choosing the worst violating pair using members of I_0 and some
% other indices.
        {
          inner_loop_success = 1;
          do until ( (bup > blow - 2*tol) | inner_loop_success = 0 )
          {
            i2 = i_low
            y2 = target(i2)
            alph2 = Lagrange multiplier for i2
            F2 = fcache[i2]
            i1 = i_up
            inner_loop_success = takeStep(i_up,i_low)
            numChanged += inner_loop_success
          }
          numChanged = 0
        }
        if (examineAll == 1)
          examineAll = 0
        else if (numChanged == 0)
          examineAll = 1
      }
```