Improvements to SMO Algorithm for SVM Regression[1]

S.K. Shevade shirish@csa.iisc.ernet.in
S.S. Keerthi mpessk@guppy.mpe.nus.edu.sg
C. Bhattacharyya cbchiru@csa.iisc.ernet.in
& K.R.K. Murthy murthy@csa.iisc.ernet.in

Technical Report CD-99-16

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 Smola and Schölkopf's Sequential Minimal Optimization (SMO) algorithm for SVM regression 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 for regression. These modified algorithms perform significantly faster than the original SMO on the datasets tried.

1 Introduction

Support Vector Machine (SVM) is an elegant tool for solving pattern-recognition and regression problems. Over the past few years, it has attracted a lot of researchers from the neural network and mathematical programming community; the main reason for this being their ability to provide excellent generalization performance. SVMs have also been demonstrated to be valuable for several real-world applications.

In this paper, we address the SVM regression problem. Recently, Smola and Schölkopf[5, 6] proposed an iterative algorithm called Sequential Minimal Optimization (SMO), for solving the regression problem using SVMs. This algorithm is an extension of the SMO algorithm proposed by Platt[4] for SVM classifier design. The remarkable feature of the SMO algorithms is that they are fast as well as very easy to implement. In a recent paper[3] we suggested some improvements to Platt's SMO algorithm for SVM classifier design. In this paper, we extend those ideas to Smola and Schölkopf's SMO algorithm. The improvements that we suggest in this paper enhance the value of SMO for regression even further. Our improvements overcome an important source of confusion and inefficiency caused by the way SMO maintains a single threshold value. Getting clues from optimality criteria associated with the dual problem, we suggest the use of two threshold parameters and devise two modified versions of SMO for regression that are much more efficient than the original SMO. Computational comparison on datasets show that the modifications perform significantly better than the original SMO.

The paper is organized as follows. In section 2 we briefly discuss the SVM regression problem formulation, the dual problem and the associated Karush-Kuhn-Tucker (KKT) optimality conditions. We also point out how these conditions lead to proper criteria for terminating algorithms

for designing SVM. Section 3 gives a brief overview of Smola's SMO algorithm for regression. In section 4 we point out the problem associated with the way SMO uses a single threshold value and describe the modified algorithm in section 5. Computational comparison between the original SMO algorithm and our modifications is done in section 6. The appendix gives the pseudo-codes for our SMO modifications. These pseudo-codes are very similar to those for SMO given by Smola and Schölkopf in [5,6].

2 The SVM Regression Problem and Optimality Conditions

The basic problem addressed in this paper is the regression problem. The tutorial by Smola and Schölkopf[5] 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, $z = \phi(x)$. Let the training set, $\{x_i, d_i\}$, consist of m data points, where x_i is the i-th input pattern and d_i is the corresponding target value, $d_i \in \mathbb{R}$. The goal of SVM regression is to estimate a function $f(x)$ that is as "close" as possible to the target values d_i for every x_i and at the same time, is as "flat" as possible for good generalization. The function f is represented using a linear function in the feature space:

$$f(x) = w \cdot \phi(x) + b$$

where b denotes the bias. As in all SVM designs, we define the kernel function $k(x, \hat{x}) = \phi(x) \cdot \phi(\hat{x})$, where "$\cdot$" denotes inner product in the z space. Thus, all computations will be done using only the kernel function. This inner-product kernel helps in taking the dot product of two vectors in the feature space without having to construct the feature space explicitly. Mercer's theorem[2] tells the conditions under which this kernel operator is useful for SVM designs.

For SVM regression purposes, Vapnik[7] suggested the use of ϵ-insensitive loss function where the error is not penalized as long as it is less than ϵ. It is assumed here that ϵ is known *a priori*. Using this error function together with a regularizing term, and letting $z_i = \phi(x_i)$, the optimization

problem solved by the support vector machine can be formulated as:

$$\begin{array}{ll} \min & \frac{1}{2}\|w\|^2 + C\sum_i(\xi_i + \xi_i{}') \\ \text{s.t.} & d_i - w\cdot z_i - b \le \epsilon + \xi_i \\ & w\cdot z_i + b - d_i \le \epsilon + \xi_i{}' \\ & \xi_i, \xi_i{}' \ge 0 \end{array} \tag{P}$$

The above problem is referred to as the *primal* problem. The constant $C > 0$ determines the trade-off between the smoothness of f and the amount up to which deviations larger than ϵ are tolerated. The Lagrangian for this problem is

$$L = \frac{1}{2}\|w\|^2 + C\sum_i(\xi_i+\xi_i{}') - \sum_i \alpha_i(w\cdot z_i + b - d_i + \epsilon + \xi_i) - \sum_i \alpha_i{}'(d_i - w\cdot z_i - b + \epsilon + \xi_i{}') - \sum_i(\nu_i\xi_i + \nu_i{}'\xi_i{}')$$

The KKT optimality conditions require that the partial derivatives of L with respect to the primal variables $w, \xi_i, \xi_i{}', b$ should vanish:

$$\begin{aligned} w &= \textstyle\sum_i(\alpha_i - \alpha_i{}')z_i \\ \nu_i &= C - \alpha_i \\ \nu_i{}' &= C - \alpha_i{}' \\ \textstyle\sum_i(\alpha_i - \alpha_i{}') &= 0 \end{aligned}$$

The $\nu_i^{(\prime)}$'s can be eliminated using the second and third equations. We will refer to the $\alpha^{(\prime)}$'s as Lagrange multipliers. Let us define

$$W(\alpha, \alpha') = \sum_i(\alpha_i - \alpha_i{}')z_i$$

Using Wolfe duality theory[2], it can be shown that the $\alpha^{(\prime)}$'s are obtained by solving the following *Dual* problem:

$$\begin{array}{ll} \max & \displaystyle\sum_i d_i(\alpha_i - \alpha_i{}') - \epsilon\sum_i(\alpha_i + \alpha_i{}') - \frac{1}{2}W(\alpha,\alpha')\cdot W(\alpha,\alpha') \\ \text{s.t.} & \displaystyle\sum_i(\alpha_i - \alpha_i{}') = 0 \\ & \alpha_i, \alpha_i{}' \in [0, C] \ \ \forall\, i \end{array} \tag{D}$$

Once the α_i's are obtained, the primal variables, w, b, ξ_i and $\xi_i{}'$ can be easily determined by using the KKT conditions mentioned earlier.

The feature space (and hence w) can be infinite dimensional. This makes it computationally difficult to solve the primal problem (P). The numerical approach in SVM design is to solve the dual problem since it is a finite-dimensional optimization problem. (Note that $W(\alpha,\alpha')\cdot W(\alpha,\alpha') = \sum_i \sum_j (\alpha_i - \alpha_i')(\alpha_j - \alpha_j')k(x_i,x_j)$.) 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:

$$\begin{aligned} L_D &= \frac{1}{2}W(\alpha,\alpha')\cdot W(\alpha,\alpha') - \sum_i d_i(\alpha_i - \alpha_i') + \epsilon\sum_i(\alpha_i + \alpha_i') \\ &\quad +\beta\sum_i(\alpha_i - \alpha_i') - \sum_i \pi_i\alpha_i - \sum_i \psi_i\alpha_i' - \sum_i \delta_i(C-\alpha_i) - \sum_i \eta_i(C-\alpha_i') \end{aligned}$$

Let

$$F_i = d_i - W(\alpha,\alpha')\cdot z_i$$

The KKT conditions for the dual problem are:

$$\begin{aligned} \frac{\partial L_D}{\partial \alpha_i} &= -F_i + \epsilon + \beta - \pi_i + \delta_i = 0 \\ \frac{\partial L_D}{\partial \alpha_i'} &= F_i + \epsilon - \beta - \psi_i + \eta_i = 0 \\ \pi_i\alpha_i &= 0, \quad \pi_i \geq 0, \quad \alpha_i \geq 0 \\ \psi_i\alpha_i' &= 0, \quad \psi_i \geq 0, \quad \alpha_i' \geq 0 \\ \delta_i(C-\alpha_i) &= 0, \quad \delta_i \geq 0, \quad \alpha_i \leq C \\ \eta_i(C-\alpha_i') &= 0, \quad \eta_i \geq 0, \quad \alpha_i' \leq C \end{aligned}$$

These conditions can be simplified by considering the following five cases:

Case 1: $\alpha_i = \alpha_i' = 0$

$$-\epsilon \leq (F_i - \beta) \leq \epsilon \tag{1a}$$

Case 2: $\alpha_i = C$

$$F_i - \beta \geq \epsilon \tag{1b}$$

Case 3: $\alpha_i' = C$

$$F_i - \beta \leq -\epsilon \tag{1c}$$

Case 4: $\alpha_i \in (0, C)$

$$F_i - \beta = \epsilon \tag{1d}$$

Case 5: $\alpha_i' \in (0, C)$

$$F_i - \beta = -\epsilon \tag{1e}$$

It is easy to check that at optimality, for every i, α_i and α_i' cannot be non-zero at the same time. Hence cases corresponding to $\alpha_i\alpha_i' \neq 0$ have been left out. (It is worth noting here that in the SMO regression algorithm and its modifications discussed in this paper, the condition, $\alpha_i\alpha_i' = 0$ $\forall\ i$ is maintained throughout.)

Define the following index sets at a given α: $I_{0a} = \{i : 0 < \alpha_i < C\}$; $I_{0b} = \{i : 0 < \alpha_i' < C\}$; $I_1 = \{i : \alpha_i = 0, \alpha_i' = 0\}$; $I_2 = \{i : \alpha_i = 0, \alpha_i' = C\}$; $I_3 = \{i : \alpha_i = C, \alpha_i' = 0\}$. Also, let $I_0 = I_{0a} \cup I_{0b}$. Let us also define $\tilde{F}_i$ and $\bar{F}_i$ as

$$\begin{aligned} \tilde{F}_i &= F_i + \epsilon \quad \text{if } i \in I_{0b} \cup I_2, \\ &= F_i - \epsilon \quad \text{if } i \in I_{0a} \cup I_1. \end{aligned}$$

and

$$\begin{aligned} \bar{F}_i &= F_i - \epsilon \quad \text{if } i \in I_{0a} \cup I_3, \\ &= F_i + \epsilon \quad \text{if } i \in I_{0b} \cup I_1. \end{aligned}$$

Using these definitions we can rewrite the necessary conditions mentioned in (1a)-(1e) as

$$\beta \geq \tilde{F}_i \ \forall\ i \in I_0 \cup I_1 \cup I_2; \quad \beta \leq \bar{F}_i \ \forall\ i \in I_0 \cup I_1 \cup I_3. \tag{2}$$

Let us define

$$b_{up} = \min\{\bar{F}_i : i \in I_0 \cup I_1 \cup I_3\} \ \text{ and } \ b_{low} = \max\{\tilde{F}_i : i \in I_0 \cup I_1 \cup I_2\} \tag{3}$$

Then the optimality conditions will hold at some α iff

$$b_{low} \leq b_{up} \tag{4}$$

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, α') if one of the following two sets of conditions holds:

$$i \in I_0 \cup I_1 \cup I_2, \ \ j \in I_0 \cup I_1 \cup I_3 \quad \text{and} \quad \tilde{F}_i > \bar{F}_j \tag{5a}$$

$$i \in I_0 \cup I_1 \cup I_3, \ \ j \in I_0 \cup I_1 \cup I_2 \quad \text{and} \quad \bar{F}_i < \tilde{F}_j \tag{5b}$$

Note that optimality condition 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 (4) can be replaced by

$$b_{low} \leq b_{up} + 2\tau \tag{6}$$

where τ is a positive tolerance parameter. (In the pseudo-codes given in the appendix of this paper, this parameter is referred to as `tol`). Correspondingly, the definition of violation can be altered by replacing (5a) and (5b) respectively by:

$$i \in I_0 \cup I_1 \cup I_2, \quad j \in I_0 \cup I_1 \cup I_3 \quad \text{and} \quad \tilde{F}_i > \bar{F}_j + 2\tau \tag{7a}$$

$$i \in I_0 \cup I_1 \cup I_3, \quad j \in I_0 \cup I_1 \cup I_2 \quad \text{and} \quad \bar{F}_i < \tilde{F}_j - 2\tau \tag{7b}$$

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

Let $E_i = F_i - b$. Using (1) it is easy to check that optimality holds iff there exists a b such that the following hold for every i:

$$\alpha_i > 0 \Rightarrow E_i \geq \epsilon - \tau \tag{8a}$$

$$\alpha_i < C \Rightarrow E_i \leq \epsilon + \tau \tag{8b}$$

$$\alpha_i{}' > 0 \Rightarrow E_i \leq -\epsilon + \tau \tag{8c}$$

$$\alpha_i{}' < C \Rightarrow E_i \geq -\epsilon - \tau \tag{8d}$$

These conditions are used in[5,6] together with a special choice of b to check if an example violates the KKT conditions. However, unless the choice of b turns out to be right, using the above conditions for checking optimality can be incorrect. We will say more about this in section 4 after a brief discussion of Smola and Schölkopf's SMO algorithm in the next section.

3 Smola and Schölkopf's SMO Algorithm for Regression

A number of algorithms have been suggested for solving the dual problem. Smola and Schölkopf[5, 6] give a detailed view of these algorithms and their implementations. Traditional quadratic programming algorithms such as interior point algorithms 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 Cholesky decomposition of a large sub-matrix of the kernel matrix. Third, coding of these algorithms is difficult.

Attempts have been made to develop methods that overcome some or all of these problems. One such method is chunking. The idea here is to operate on a fixed size subset of the training set at a time. This subset is called the working set and the optimization subproblem is solved with respect to the variables corresponding to the examples in the working set and a set of support vectors for the current working set is found. These current support vectors are then used to determine the new working set, the data the current estimator would make error on. The new optimization subproblem is solved and this process is repeated until the KKT conditions are satisfied for all the examples.

Platt[4] proposed an algorithm, called Sequential Minimal Optimization (SMO) for the SVM classifier design. This algorithm puts chunking to the extreme by iteratively selecting working sets of size two and optimizing the target function with respect to them. One advantage of using working sets of size 2 is that the optimization subproblem can be solved analytically. The chunking process is repeated till all the training examples satisfy KKT conditions. Smola and Schölkopf[5,6] extended these ideas for solving the regression problem using SVMs. We describe this algorithm very briefly below. The details, together with a pseudo-code can be found in [5,6]. We assume that the reader is familiar with them. To convey our ideas compactly we employ the notations used in [5,6].

The SMO algorithm employs a two loop approach. The conditions in (8) are employed for checking optimality. The Type I loop (`examineAll=1`) iterates over all the patterns violating the optimality conditions while type II loop (`examineAll=0`) runs over all patterns whose Lagrange multipliers are in $(0, C)$ and which violate the optimality conditions. Type II loops are repeated until optimality conditions are satisfied by all patterns whose Lagrange multipliers are in $(0, C)$. Then Type I loop is executed to pick up other patterns that violate optimality conditions. The algorithm maintains an error cache for storing E_i for every example whose Lagrange multiplier lies in $(0, C)$. This helps especially when the algorithm executes type II loop. After having chosen one element i which violates the optimality conditions, the other element j of the working set $\{i, j\}$ is

determined using the procedure `ChoiceTwoHeuristic`. This procedure

1. Searches for an example with non-bound Lagrange multiplier to make progress in the objective function.

2. If the above step fails, then all the other examples are looked at to find an example where progress can be made.

3. If both these steps fail, then the next i which violates the optimality conditions is considered.

After doing a joint optimization on the pair $\{i, j\}$, the algorithm chooses an example (from $\{i, j\}$) with non-bound Lagrange multiplier and uses (1d) or (1e) to determine the value of b. In the rare case that $\alpha_i^{(l)}$ and $\alpha_j^{(l)}$ are all at bound values, there exists a whole interval of admissible thresholds and b is set to the midpoint of this interval.

It is important to point out that a value of b is not needed to take a step. Suppose that the output error on the i-th pattern is given as

$$E_i = F_i - b.$$

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[5,6] shows that to take a step by varying the multipliers of i_2 and i_1, we only need to know $E_{i_1} - E_{i_2} = F_{i_1} - F_{i_2}$. Therefore, *a knowledge of the value of b is not needed to take a step.* Despite this fact, for the SMO algorithm it is necessary to maintain and update b because (8) is employed for checking optimality!

4 Problems with SMO algorithm

SMO algorithm for regression, discussed above, is very simple and easy to implement. This algorithm, after every optimization iteration on a pair of training points, computes and maintains a single threshold value (bias b) by picking an example with non-bound Lagrange multiplier and applying KKT conditions to it. It is easy to give an example where SMO can get into a confused end state and also can become inefficient, typically near a solution point, because of its way of computing and maintaining a single threshold value.

At any instant, the SMO algorithm fixes b based on the current two indices used for joint optimization. However, while checking whether the remaining examples violate optimality or not, it is quite possible that a different, shifted choice of b may do a better job. So, at some stage in the

SMO algorithm it is possible that, even though α has reached a value where optimality is satisfied (i.e., (6)), but SMO has not detected this because it has not identified the correct choice of b. It is also quite possible that, a particular index may appear to violate the optimality conditions because (8) is employed using an "incorrect" value of b although this index may not be able to pair with another to make progress in the objective function. 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 for regression, 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 always better than the original SMO algorithm for regression and in most situations, they also give quite a remarkable improvement in efficiency.

The modifications avoid the use of a single threshold value b and the use of (8) for checking optimality. Instead, two threshold parameters, b_{up} and b_{low} are maintained and (4) is used for checking optimality. The two modifications are described by the pseudo-codes given in the appendix. In this section we only give some additional pointers that will help to give an easy understanding of the pseudo-codes.

1. Suppose, at any instant, F_i is available for all i. Let i_{low} and i_{up} be indices such that

$$\tilde{F}_{i_{low}} = b_{low} = max\{\tilde{F}_i : i \in I_0 \cup I_1 \cup I_2\} \tag{9a}$$

and

$$\bar{F}_{i_{up}} = b_{up} = min\{\bar{F}_i : i \in I_0 \cup I_1 \cup I_3\} \tag{9b}$$

Then checking a particular i for optimality is easy. For example, suppose $i \in I_3$. We only have to check if $\bar{F}_i < b_{low} - 2\tau$. 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 other sets. Thus, in our approach, the checking of optimality of the first index, i_2 and the choice of 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 $\alpha_i, i \in I_0$; 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 all indices are examined for optimality.

3. The procedure `takeStep` is modified. After a successful step using a pair of indices, (i_2, i_1), let $\hat{I} = I_0 \cup \{i_1, i_2\}$. We compute, *partially*, (i_{low}, b_{low}) and (i_{up}, b_{up}) using $\hat{I}$ only (i.e., use only $i \in \hat{I}$ in (9)). 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, $\hat{I} \cap (I_0 \cup I_1 \cup I_2)$ and $\hat{I} \cap (I_0 \cup I_1 \cup I_3)$, is non-empty; hence the partially computed (i_{low}, b_{low}) and (i_{up}, b_{up}) will not be null elements.

4. When working with only $\alpha_i, i \in I_0$, i.e., a loop with `examineAll = 0`, one should note that, if (6) 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 similar to 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 $\bar{F}_{i_2} < b_{low} - 2\tau$ 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_{low}, i_{low}) and (b_{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_{low}, b_{up}). For example, if $i \in I_3$ and $\bar{F}_i < b_{low} - 2\tau$, 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_{up}) is modified using $\bar{F}_i$, i.e., if $\bar{F}_i < b_{up}$ then we do: $i_{up} := i$ and $b_{up} := \bar{F}_i$.

7. Suppose we do as described above. What happens if there is no violation for any i in a loop having `examineAll = 1`? 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 (7). Let us say, $i < j$. Then j would not have satisfied the optimality check in the above described implementation because either $\bar{F}_i$ or $\tilde{F}_i$ would have, earlier than j is seen, affected either the calculation of b_{up} and/or b_{low} 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 (6) holds it is possible for all indices to satisfy the optimality checks. Furthermore, when (6) holds and the loop over all indices has been completed, the true values of b_{up} and b_{low}, as defined in (3) would have been computed since all indices have been encountered. As a final choice of b (for later use in doing inference) it is appropriate to set: $b = 0.5(b_{up} + b_{low})$.

6 Computational Comparison

In this section we compare the performance of our modifications against Smola and Schölkopf's SMO algorithm for regression on two datasets. We implemented all these methods in C and ran them using `gcc` on a 200 MHz Pentium machine. The value, $\tau = .01$ was used for all experiments.

The first dataset is a toy dataset where the function to be approximated is a cubic polynomial, $.02x^3 + .05x^2 - x$. The domain of this function was fixed to $[-10, 10]$. A Gaussian noise of mean zero and variance 1 was added to the training set output. A hundred training samples were chosen randomly. The performance of the three algorithms for the polynomial kernel

$$k(x_i, x_j) = (1 + x_i \cdot x_j)^p$$

where p was chosen to be 3, is given in Table 1.

The Boston housing dataset is a standard benchmark for testing regression algorithms. This dataset is available at UCI Repository [1]. The dimension of the input is 13. The training set has 406 patterns. A Gaussian noise of mean zero and standard deviation 6 was added to the training data. The Gaussian kernel

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

was used. The σ value employed was 15.0. Table 2 shows the performance of the three algorithms on this dataset.

It is very clear that both modifications outperform the original SMO algorithm. In many situations the improvement in efficiency is remarkable. In particular, at large values of C the

C	SMO			SMO-Modification 1			SMO-Modification 2		
	Support Vectors		Time (s)	Support Vectors		Time (s)	Support Vectors		Time (s)
	non-bound	bound		non-bound	bound		non-bound	bound	
.01	3	69	0.41	2	67	0.15	2	67	0.17
.05	5	56	1.35	4	57	0.17	3	58	0.16
.1	4	46	1.12	8	44	0.26	5	45	0.18
.25	7	42	1.29	6	41	0.3	6	42	0.3
.5	5	41	6.22	5	41	0.52	7	40	0.57
1	6	41	2.27	6	39	0.86	7	39	2.52
2.5	7	39	7.06	7	38	1.77	8	37	3.98
5	8	39	20.82	8	37	2.74	11	35	12.5
10	14	33	37.72	10	35	5.4	9	35	18.68
50	13	35	471.45	12	31	27.24	20	27	44.04

Table 1: Comparison of 3 methods on a toy dataset.

C	SMO			SMO-Modification 1			SMO-Modification 2		
	Support Vectors		Time (s)	Support Vectors		Time (s)	Support Vectors		Time (s)
	non-bound	bound		non-bound	bound		non-bound	bound	
.01	2	209	129.96	0	198	23.44	0	192	20.89
.05	3	180	23.22	4	180	14.47	4	180	13.71
.1	6	170	22.98	5	171	10.47	6	170	7.61
.25	20	150	25.71	19	150	6.98	18	149	4.87
.5	25	144	40.01	23	145	6.26	25	144	4.87
1	42	133	50.15	40	133	6.99	41	133	5.06
2.5	58	112	54.42	58	114	11.28	60	112	7.56
5	70	99	83.67	70	99	14.11	72	98	10.45
10	90	80	229.76	92	78	20.34	94	76	14.62
50	132	49	1432.44	132	46	90.60	137	46	76.18

Table 2: Comparison of 3 methods on the Boston Housing dataset

improvement is by an order of magnitude. Between the two modifications, it is difficult to say which one is better. More detailed testing on other benchmark problems is needed to decide which one of them is better overall.

We have not reported a comparison of the generalization abilities of the three methods since all three methods apply to the same problem formulation, are terminated at the same training set accuracy, and hence give very close generalization performance.

7 Conclusion

In this paper we have pointed out an important source of inefficiency in Smola and Schölkopf'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 significantly in most situations.

References

[1] C.L. Blake and C.J. Merz, UCI Repository of machine learning databases, University of California, Department of Information and Computer Science, Irvine, CA, USA, 1998. See: *http://www.ics.uci.edu/~mlearn/MLRepository.html*.

[2] C.J.C. Burges, A tutorial on support vector machines for pattern recognition, *Data Mining and Knowledge Discovery*, 3(2), 1998.

[3] S.S. Keerthi, S.K, Shevade, C. Bhattacharyya and K.R.K. Murthy, Improvements to Platt's SMO Algorithm for SVM Classifier Design, Technical Report CD-99-14, Control Division, Dept. of Mechanical and Production Engineering, National University of Singapore, Singapore, August 1999. See: *http://guppy.mpe.nus.edu.sg/~mpessk*

[4] 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.

[5] A.J. Smola, Learning with Kernels, *PhD Thesis*, GMD, Birlinghoven, Germany, 1998

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

[7] V. Vapnik, *The Nature of Statistical Learning Theory*, Springer, NY, USA, 1995.

Appendix. Pseudo-Codes for Modified SMO Algorithm for Regression.

```
target = desired output vector
point = training point matrix
f-cache = cache vector for F_i values(contains d_i - w . z_i)

 % Note: Fi is different from the SVM output on point[i1].
 % Fi does not subtract any threshold

 procedure takeStep(i1, i2)
    if (i1 == i2) return 0
    alpha1, alpha1' = Lagrange multipliers for i1
    F1 = f-cache[i1]
    k11 = kernel(point[i1], point[i1])
    k12 = kernel(point[i1], point[i2])
    k22 = kernel(point[i2], point[i2])
    eta = -2*k12+k11+k22
    gamma = alpha1-alpha1'+alpha2-alpha2'

    % We assume that eta > 0. Otherwise one has to repeat the complete
    % reasoning similarly (i.e. compute objective functions at L and H
    % and decide which one is largest

    case1 = case2 = case3 = case4 = finished = 0
    alpha1old = alpha1, alpha1old' = alpha1'
    alpha2old = alpha2, alpha2old' = alpha2'
    deltaphi = F1 - F2
    while !finished
       % This loop is passed at most three times
       % Case variables needed to avoid attempting small changes twice
       if (case1 == 0) &&
          (alpha1 > 0 || (alpha1' == 0 && deltaphi > 0)) &&
          (alpha2 > 0 || (alpha2' == 0 && deltaphi < 0))
            compute L, H (w.r.t. alpha1, alpha2)
            if (L < H)
               a2 = alpha2 - (deltaphi / eta )
               a2 = min(a2, H)
               a2 = max(L, a2)
               a1 = alpha1 - (a2 - alpha2)
               update alpha1, alpha2 if change is larger than some eps
            else
               finished = 1
            endif
            case1 = 1
       elseif (case2 == 0) &&
```

```
        (alpha1 > 0 || (alpha1' == 0 && deltaphi > 2*epsilon)) &&
        (alpha2' > 0 || (alpha2 == 0 && deltaphi > 2*epsilon))
          compute L, H (w.r.t. alpha1, alpha2')
          if (L < H)
             a2 = alpha2' + ((deltaphi - 2*epsilon)/eta))
             a2 = min(a2, H)
             a2 = max(L, a2)
             a1 = alpha1 + (a2-alpha2')
             update alpha1, alpha2' if change is larger than some eps
          else
             finished = 1
          endif
          case2 = 1
    elseif (case3 == 0) &&
         (alpha1' > 0 || (alpha1 == 0 && deltaphi < -2*epsilon)) &&
         (alpha2 > 0 || (alpha2' == 0 && deltaphi < -2*epsilon))
          compute L, H (w.r.t. alpha1', alpha2)
          if (L < H)
              a2 = alpha2 - ((deltaphi + 2*epsilon)/eta)
              a2 = min(a2, H)
              a2 = max(L, a2)
              a1 = alpha1' + (a2 - alpha2)
              update alpha1', alpha2 if change is larger than some eps
          else
              finished = 1
          endif
          case3 = 1
    elseif (case4 == 0) &&
        (alpha1' > 0) || (alpha1 == 0 && deltaphi < 0)) &&
        (alpha2' > 0) || (alpha2 == 0 && deltaphi > 0))
          compute L, H (w.r.t. alpha1', alpha2')
          if (L < H)
              a2 = alpha2' + deltaphi/eta
              a2 = min(a2, H)
              a2 = max(L, a2)
              a1 = alpha1' - (a2 - alpha2')
              update alpha1, alpha2' if change is larger than some eps
          else
              finished = 1
          endif
          case4 = 1
    else
          finished = 1
    endif
    update deltaphi
endwhile
if changes in alpha('), alpha2(') are larger than some eps
     Update f-cache[i] for i in I_0 using new Lagrange multipliers
     Store the changes in alpha, alpha' array
     Update I_0, I_1, I_2, I_3
     Compute (i_low, b_low) and (i_up, b_up) by applying the conditions
     mentioned above, using only i1, i2 and indices in I_0
     return 1
else
     return 0
```

```
   endif
endprocedure

procedure examineExample(i2)
   alpha2, alpha2' = Lagrange multipliers for i2
   if (i2 is in I_0)
       F2 = f-cache[i2]
   else
       compute F2 = F_i2 and set f-cache[i2] = F2
       % Update (b_low, i_low) or (b_up, i_up) using (F2, i2)...
       if  (i2 is in I_1)
           if (F2+epsilon < b_up)
                b_up = F2+epsilon, i_up = i2
           elseif (F2-epsilon > b_low)
                b_low = F2-epsilon, i_low = i2
           end if
       elseif ( (i2 is in I_2) && (F2+epsilon > b_low) )
             b_low = F2+epsilon, i_low = i2
       elseif ( (i2 is in I_3) && (F2-epsilon < b_up) )
             b_up = F2-epsilon, i_up = i2
       endif
   endif
   % Check optimality using current b_low and b_up and, if
   % violated, find an index i1 to do joint optimization with i2...
   optimality = 1;
   case 1: i2 is in I_0a
           if (b_low-(F2-epsilon) > 2 * tol)
               optimality = 0;  i1 = i_low;
               % For i2 in I_0a choose the better i1...
               if ((F2-epsilon)-b_up > b_low-(F2-epsilon))
                  i1 = i_up;
               endif
           elseif ((F2-epsilon)-b_up > 2 * tol)
               optimality = 0; i1 = i_up;
               % For i2 in I_0a choose the better i1...
               if ((b_low-(F2-epsilon) > (F2-epsilon)-b_up)
                  i1 = i_low;
               endif
           endif
   case 2: i2 is in I_0b
           if (b_low-(F2+epsilon) > 2 * tol)
              optimality = 0; i1 = i_low;
              % For i2 in I_0b choose the better i1...
              if ((F2+epsilon)-b_up > b_low-(F2+epsilon))
                  i1 = i_up;
              endif
           elseif ((F2+epsilon)-b_up > 2 * tol)
              optimality = 0; i1 = i_up;
              % For i2 in I_0b choose the better i1...
              if ((b_low-(F2+epsilon) > (F2+epsilon)-b_up)
                  i1 = i_low;
              endif
           endif
   case 3: i2 is in I_1
           if (b_low-(F2+epsilon) > 2 * tol)
```

```
            optimality = 0; i1 = i_low;
            % For i2 in I1 choose the better i1...
            if ((F2+epsilon)-b_up > b_low-(F2+epsilon)
                i1 = i_up;
            endif
         elseif ((F2-epsilon)-b_up > 2 * tol)
            optimality = 0; i1 = i_up;
            % For i2 in I1 choose the better i1...
            if (b_low-(F2-epsilon) > (F2-epsilon)-b_up)
                i1 = i_low;
            endif
         endif
   case 4: i2 is in I_2
         if ((F2+epsilon)-b_up > 2*tol)
            optimality = 0, i1 = i_up
         endif
   case 5: i2 is in I_3
         if ((b_low-(F2-epsilon) > 2*tol)
            optimality = 0, i1 = i_low
         endif

   if (optimality == 1)
       return 0
   if (takeStep(i1, i2))
       return 1
   else
       return 0
   endif
endprocedure

% main routine for modification 1
procedure main

   set alpha and alpha' to zero for every example
   set I_1 to contain all the examples
   Choose any example i from the training set.
   set b_up = target[i]+epsilon
   set b_low = target[i]-espilon
   i_up = i_low = i;
   while (numChanged > 0 || examineAll)
        numChanged = 0;
        if (examineAll)
             loop I over all the 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 (b_up > b_low - 2*tol)  at any I
                       exit the loop after setting numChanged = 0
        endif
        if (examineAll == 1)
             examineAll = 0;
        elseif (numChanged == 0)
             examineAll = 1;
```

```
        endif
   endwhile
endprocedure

% main routine for modification 2
procedure main

   set alpha and alpha' to zero for every example
   set I_1 to contain all the examples
   Choose any example i from the training set.
   set b_up = target[i]+epsilon
   set b_low = target[i]-espilon
   i_up = i_low = i;
   while (numChanged > 0 || examineAll)
          numChanged = 0;
          if (examineAll)
             loop I over all the training examples
                   numChanged += examineExample(I)
          else

          %  The following loop is the only difference between the two
          %  SMO modifications. Whereas, modification 1, the type II
          %  loop selects i2 fro 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
                  i2 = i_low
                  alpha2, alpha2' = Lagrange multipliers for i2
                  F2 = f-cache[i2]
                  i1 = i_up
                  inner_loop_success = takeStep(i_up, i_low)
                  numChanged += inner_loop_success
             until ( (b_up > b_low - 2*tol) || inner_loop_success == 0)
             numChanged = 0;
          endif
          if (examineAll == 1)
             examineAll = 0
          elseif (numChanged == 0)
             examineAll = 1
          endif
   endwhile
endprocedure
```