Machine Learning, 46, 271–290, 2002

Efficient SVM Regression Training with SMO*

GARY WILLIAM FLAKE flake@research.nj.nec.com
NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, USA

STEVE LAWRENCE lawrence@research.nj.nec.com
NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, USA

Editor: Nello Cristianini

Abstract. The sequential minimal optimization algorithm (SMO) has been shown to be an effective method for training support vector machines (SVMs) on classification tasks defined on sparse data sets. SMO differs from most SVM algorithms in that it does not require a quadratic programming solver. In this work, we generalize SMO so that it can handle regression problems. However, one problem with SMO is that its rate of convergence slows down dramatically when data is non-sparse and when there are many support vectors in the solution—as is often the case in regression—because kernel function evaluations tend to dominate the runtime in this case. Moreover, caching kernel function outputs can easily degrade SMO's performance even more because SMO tends to access kernel function outputs in an unstructured manner. We address these problems with several modifications that enable caching to be effectively used with SMO. For regression problems, our modifications improve convergence time by over an order of magnitude.

Keywords: support vector machines, sequential minimal optimization, regression, caching, quadratic programming, optimization

1. Introduction

A support vector machine (SVM) is a type of model that is optimized so that prediction error and model complexity are simultaneously minimized (Vapnik, 1995). Despite having many admirable qualities, research into the area of SVMs has been hindered by the fact that quadratic programming (QP) solvers provided the only known training algorithm for years.

In 1997, Osuna, Freund, and Girosi (1997) showed that SVMs can be optimized by decomposing a large QP problem into a series of smaller QP subproblems. Optimizing each subproblem minimizes the original QP problem in such a way that once no further progress can be made with all of the smaller subproblems, the original QP problem is solved. Since each subproblem can have fixed size, optimizing via decomposition can be done with a constant size or linear memory footprint. Moreover, many experimental results indicate that decomposition can be much faster than QP. More recently, the sequential minimal optimization algorithm (SMO) was introduced (Platt, 1998, 1999b) as an extreme example

Source code availability: The source code used in this work is part of **NODElib**, the Neural Optimization and Development library. **NODElib** is freely available under a "copyleft" licensing agreement, and can be downloaded from:http://www.neci.nj.nec.com/homepages/flake/nodelib.tgz.

of decomposition. Because SMO uses a subproblem of size two, each subproblem has an analytical solution. Thus, for the first time, SVMs could be optimized without a QP solver.

In addition to SMO, other new methods (Friess, Cristianini, & Campbell, 1998; Joachims, 1999; Mangasarian & Musicant, 1999) have been proposed for optimizing SVMs online without a QP solver. While these other online methods hold great promise, SMO is the only online SVM optimizer that explicitly exploits the quadratic form of the objective function and simultaneously uses the analytical solution of the size two case.

While SMO has been shown to be effective on sparse data sets and especially fast for linear SVMs, the algorithm can be extremely slow on non-sparse data sets and on problems that have many support vectors. Regression problems are especially prone to these issues because the inputs are usually non-sparse real numbers (as opposed to binary inputs) with solutions that have many support vectors. Because of these constraints, there have been few reports of SMO being successfully used on regression problems.

In this work, we derive a generalization of SMO to handle regression problems and address the runtime issues of SMO by modifying the heuristics and underlying algorithm so that kernel outputs can be effectively cached. Conservative results indicate that for high-dimensional, non-sparse data (and especially regression problems), the convergence rate of SMO can be improved by an order of magnitude or more.

This paper is divided into six additional sections. Section 2 contains a basic overview of SVMs and provides a minimal framework on which the later sections build. In Section 3, we generalize SMO to handle regression problems. This simplest implementation of SMO for regression can optimize SVMs on regression problems but with very poor convergence rates. In Section 4, we introduce several modifications to SMO that allow kernel function outputs to be efficiently cached. Section 5 contains numerical results that show that our modifications produce an order of magnitude improvement in convergence speed. Finally, Section 6 summarizes our work and addresses future research in this area.

2. Introduction to SVMs

Consider a set of data points, $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_\ell, y_\ell)\}$, such that $\boldsymbol{x}_i \in \mathbb{R}^n$ is an input, y_i is a target output, and ℓ is the total number of exemplars. An SVM is a model that is calculated as a weighted sum of kernel function outputs. The kernel function can be an inner product, Gaussian basis function, polynomial, or any other function that obeys Mercer's condition. Thus, the output of an SVM is either a linear function of the inputs, or a linear function of the kernel outputs.

Because of the generality of SVMs, they can take forms that are identical to nonlinear regression, radial basis function networks, and multilayer perceptrons. The difference between SVMs and these other methods lies in the objective functions that they are optimized with respect to and the optimization procedures that one uses to minimize these objective functions.

In the linear, noise-free case for classification, with $y_i \in \{-1, 1\}$, the output of an SVM is written as $f(\boldsymbol{x}, \boldsymbol{w}, b) = \boldsymbol{x} \cdot \boldsymbol{w} + b$, and the optimization task is defined as:

$$\text{minimize } \frac{1}{2}\|\boldsymbol{w}\|^2, \quad \text{subject to } y_i(\boldsymbol{x} \cdot \boldsymbol{w} + b) \geq 1 \ \forall i. \tag{1}$$

Intuitively, this objective function expresses the notion that one should find the simplest model that explains the data. This basic SVM framework has been generalized to include slack variables for miss-classifications, nonlinear kernel functions, regression, as well as other extensions for other problem domains. It is beyond the scope of this paper to describe the derivation of all of these extensions to the basic SVM framework. Instead, we refer the reader to the excellent tutorials by Burges (1998) and Smola and Schölkopf (1998) for introductions to SVMs for classification and regression, respectively. We delve into the derivation of the specific objective functions only as far as necessary to set the framework from which we present our own work.

In general, one can easily construct objective functions similar to Eq. (1) that include slack variables for misclassification and nonlinear kernels. These objective functions can also be modified for the special case of performing regression, i.e., with $y_i \in \mathbb{R}$ instead of $y_i \in \{-1, 1\}$. Such objective functions will always have a component that should be minimized and linear constraints that must be obeyed. To optimize the primal objective function, one converts it into the dual form which contains the minimization terms minus the linear constraints multiplied by Lagrange multipliers. Since the dual objective function is quadratic in the Lagrange multipliers—which are the only free parameters—the obvious way to optimize the model is to express it as a quadratic programming problem with linear constraints.

Our contribution in this paper uses a variant of Platt's sequential minimal optimization method that is generalized for regression and is modified for further efficiencies. SMO solves the underlying QP problem by breaking it down into a sequence of smaller optimization subproblems with two unknowns. With only two unknowns, the parameters have an analytical solution, thus avoiding the use of a QP solver. Even though SMO does not use a QP solver, it still makes reference to the dual objective functions. Thus, we now define the output function of nonlinear SVMs for classification and regression, as well as the dual objective functions that they are optimized with respect to.

In the case of classification, with $y_i \in \{-1, 1\}$, the output of an SVM is defined as:

$$f(\boldsymbol{x}, \boldsymbol{\alpha}, b) = \sum_{i=1}^{\ell} y_i \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b, \tag{2}$$

where $K(\boldsymbol{x}_i, \boldsymbol{x})$ is the underlying kernel function. The dual objective function (which should be minimized) is:

$$W(\boldsymbol{\alpha}) = \frac{1}{2} \sum_{i=1}^{\ell} \sum_{j=1}^{\ell} y_i y_j \alpha_i \alpha_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) - \sum_{i=1}^{\ell} \alpha_i, \tag{3}$$

subject to the constraints $0 \leq \boldsymbol{\alpha}_i \leq C, \forall_i$ and $\sum_{i=1}^{\ell} y_i \boldsymbol{\alpha}_i = 0$. C is a user-defined constant that represents a balance between the model complexity and the approximation error.

Regression for SVMs minimize functionals of the form:

$$L = \frac{1}{\ell} \sum_{i=1}^{\ell} |y_i - f(\boldsymbol{x}_i, \boldsymbol{w}, b)|_{\varepsilon} + \|\boldsymbol{w}\|^2, \tag{4}$$

where $|\cdot|_\varepsilon$ is an ε-insensitive error function defined as:

$$|x|_\varepsilon = \begin{cases} 0 & \text{if } |x| < \varepsilon \\ |x| - \varepsilon & \text{otherwise} \end{cases}$$

and the output of the SVM now takes the form of:

$$f(\boldsymbol{x}, \boldsymbol{\alpha}^+, \boldsymbol{\alpha}^-, b) = \sum_{i=1}^{\ell} (\alpha_i^+ - \alpha_i^-) K(\boldsymbol{x}_i, \boldsymbol{x}) + b. \tag{5}$$

Intuitively, α_i^+ and α_i^- are "positive" and "negative" Lagrange multipliers (i.e., a single weight) that obey $0 \le \alpha_i^+, \alpha_i^-, \forall_i$ and $\alpha_i^+ \alpha_i^- = 0, \forall_i$.

The dual form of Eq. (4) is written as

$$\begin{aligned} W(\boldsymbol{\alpha}^+, \boldsymbol{\alpha}^-) = {} & \varepsilon \sum_{i=1}^{\ell} (\alpha_i^+ + \alpha_i^-) - \sum_{i=1}^{\ell} y_i (\alpha_i^+ - \alpha_i^-) \\ & + \frac{1}{2} \sum_{i=1}^{\ell} \sum_{j=1}^{\ell} (\alpha_i^+ - \alpha_i^-)(\alpha_j^+ - \alpha_j^-) K(\boldsymbol{x}_i, \boldsymbol{x}_j), \end{aligned} \tag{6}$$

where one should minimize the objective function with respect to $\boldsymbol{\alpha}^+$ and $\boldsymbol{\alpha}^-$, subject to the constraints:

$$\sum_{i=1}^{\ell} (\alpha_i^+ - \alpha_i^-) = 0, \quad 0 \le \alpha_i^+, \quad \alpha_i^- \le C, \quad \forall_i. \tag{7}$$

The parameter C is the same user-defined constant that represents a balance between the model complexity and the approximation error.

In later sections, we will make extensive use of the two dual objective functions in Eqs. (3) and (6), and the SVM output functions in Eqs. (2) and (5).

3. SMO and regression

As mentioned earlier, SMO is a relatively new algorithm for training SVMs. SMO repeatedly finds two Lagrange multipliers that can be optimized with respect to each other and analytically computes the optimal step for the two Lagrange multipliers. When no two Lagrange multipliers can be optimized, the original QP problem is solved. SMO actually consists of two parts: (1) a set of heuristics for efficiently choosing pairs of Lagrange multipliers to work on, and (2) the analytical solution to a QP problem of size two. It is beyond the scope of this paper to give a complete description of SMO's heuristics. More information can be found in Platt's papers (1998, 1999b).

Since SMO was originally designed (like SVMs) to only be applicable to classification problems, the analytical solution to the size two QP problem must be generalized in order for

SMO to work on regression problems. The bulk of this section will be devoted to deriving this solution. While others (Smola & Schölkopf, 1998; Mattera, Palmieri, & Haykin, 1999; Shevade et al., 2000) have generalized SMO to regression, we believe that ours is one of the simplest and most complete derivations. As such, this section concludes with a brief sampling of related derivations.

3.1. *Step size derivation*

We begin by transforming Eqs. (5–7) by substituting $\lambda_i = \alpha_i^+ - \alpha_i^-$, and $|\lambda_i| = \alpha_i^+ + \alpha_i^-$. Thus, the new unknowns will obey the box constraint $-C \le \lambda_i \le C, \forall_i$. We will also use the shorthand $k_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and always assume that $k_{ij} = k_{ji}$. The model output and objective function can now be written as:

$$f(\boldsymbol{x}, \boldsymbol{\lambda}, b) = \sum_{i=1}^{\ell} \lambda_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b, \text{ and} \tag{8}$$

$$W(\lambda) = \varepsilon \sum_{i=1}^{\ell} |\lambda_i| - \sum_{i=1}^{\ell} \lambda_i y_i + \frac{1}{2} \sum_{i=1}^{\ell} \sum_{j=1}^{\ell} \lambda_i \lambda_j k_{ij}, \tag{9}$$

with the linear constraint $\sum_{i=1}^{\ell} \lambda_i = 0$. Our goal is to analytically express the minimum of Eq. (9) as a function of two parameters. Let these two parameters have indices u and v so that λ_u and λ_v are the two unknowns. We can rewrite Eq. (9) as

$$\begin{aligned} W(\lambda_u, \lambda_v) = {} & \varepsilon\, |\lambda_u| + \varepsilon\, |\lambda_v| - \lambda_u y_u - \lambda_v y_v + \frac{1}{2}\lambda_u^2 k_{uu} + \frac{1}{2}\lambda_v^2 k_{vv} \\ & + \lambda_u \lambda_v k_{uv} + \lambda_u z_u^* + \lambda_v z_v^* + W_c, \end{aligned} \tag{10}$$

where W_c is a term that is strictly constant with respect to λ_u and λ_v, and z_i^* is defined as:

$$z_i^* = \sum_{j \ne u,v}^{\ell} \lambda_j^* k_{ij} = f_i^* - \lambda_u^* k_{ui} - \lambda_v^* k_{vi} - b^* \tag{11}$$

with $f_i^* = f(\boldsymbol{x}_i, \boldsymbol{\lambda}^*, b)$. Note that a superscript $*$ is used above to explicitly indicate that values are computed with the old parameter values. This means that these portions of the expression will not be a function of the new parameters (which simplifies the derivation).

If we assume that the constraint, $\sum_{i=1}^{\ell} \lambda_i = 0$, is true prior to any change to λ_u and λ_v, then in order for the constraint to be true after a step in parameter space, the sum of λ_u and λ_v must be held fixed. With this in mind, let $s^* = \lambda_u + \lambda_v = \lambda_u^* + \lambda_v^*$. We can now rewrite Eq. (10) as a function of a single Lagrange multiplier by substituting $s^* - \lambda_v = \lambda_u$:

$$\begin{aligned} W(\lambda_v) = {} & \varepsilon |s^* - \lambda_v| + \varepsilon\, |\lambda_v| - (s^* - \lambda_v) y_u - \lambda_v y_v + \frac{1}{2}(s^* - \lambda_v)^2 k_{uu} \\ & + \frac{1}{2}\lambda_v^2 k_{vv} + (s^* - \lambda_v)\lambda_v k_{uv} + (s^* - \lambda_v) z_u^* + \lambda_v z_v^* + W_c. \end{aligned} \tag{12}$$

To solve Eq. (12), we need to compute its partial derivative with respect to λ_v; however, Eq. (12) is not strictly differentiable because of the absolute value function. Nevertheless, if we take $d|x|/dx = \text{sgn}(x)$, the resulting derivative is algebraically consistent:

$$\begin{aligned}\frac{\partial W}{\partial \lambda_v} &= \varepsilon(\text{sgn}(\lambda_v) - \text{sgn}(s^* - \lambda_v)) + y_u - y_v \\ &\quad + (\lambda_v - s^*)k_{uu} + \lambda_v k_{vv} + (s^* - 2\lambda_v)k_{uv} - z_u^* + z_v^*\end{aligned} \tag{13}$$

Now, setting Eq. (13) to zero yields:

$$\begin{aligned}\lambda_v(k_{vv} + k_{uu} - 2k_{uv}) &= y_v - y_u + \varepsilon(\text{sgn}(\lambda_u) - \text{sgn}(\lambda_v)) + s^*(k_{uu} - k_{uv}) + z_u^* - z_v^* \\ &= y_v - y_u + f_u^* - f_v^* + \varepsilon(\text{sgn}(\lambda_u) - \text{sgn}(\lambda_v)) + \lambda_u^* k_{uu} \\ &\quad - \lambda_u^* k_{uv} + \lambda_v^* k_{uu} - \lambda_v^* k_{uv} - \lambda_u^* k_{uu} - \lambda_v^* k_{uv} - b^* \\ &\quad + \lambda_u^* k_{uv} + \lambda_v^* k_{vv} + b^* = y_v - y_u + f_u^* - f_v^* \\ &\quad + \varepsilon(\text{sgn}(\lambda_u) - \text{sgn}(\lambda_v)) + \lambda_v^*(k_{vv} + k_{uu} - 2k_{uv})\end{aligned} \tag{14}$$

From Eq. (14), we can write a recursive update rule for λ_v in terms of its old value:

$$\lambda_v = \lambda_v^* + \frac{1}{\eta}(y_v - y_u + f_u^* - f_v^* + \varepsilon(\text{sgn}(\lambda_u) - \text{sgn}(\lambda_v))), \tag{15}$$

where $\eta = (k_{vv} + k_{uu} - 2k_{uv})$. While Eq. (15) is recursive because of the two $\text{sgn}(\cdot)$ functions, it still has a single solution that can be found quickly, as will be shown in the next subsection.

3.2. *Finding solutions*

We note, anecdotally, that a common mistake made in other implementations of SMO with regression is to treat the $(\text{sgn}(\lambda_u) - \text{sgn}(\lambda_v))$ term in Eq. (15) as if it were $(\text{sgn}(\lambda_u^*) - \text{sgn}(\lambda_v^*))$, i.e., a function of the old parameter values. This is incorrect and will usually yield a step that ascends the objective function.

Figure 1 illustrates the behavior of the partial derivative (Eq. (13)) of the objective function with respect to λ_v. If the kernel function of the SVM obeys Mercer's condition (as all common ones do), then we are guaranteed that $\eta = k_{vv} + k_{uu} - 2k_{uv} \geq 0$ is true. If η is strictly positive, then Eq. (13) will always be increasing. Moreover, if s^* is not zero, then it will be piecewise linear with two discrete jumps, as illustrated in figure 1.

Putting these facts together means that we only have to consider five possible solutions for Eq. (13). Three possible solutions correspond to using Eq. (15) with $(\text{sgn}(\lambda_u) - \text{sgn}(\lambda_v))$ set to -2, 0, and 2. The other two candidates correspond to setting λ_v to one of the transitions in figure 1: $\lambda_v = 0$ or s^*. The update rules for λ_u and λ_v must also insure that both parameters take values within $\pm C$.

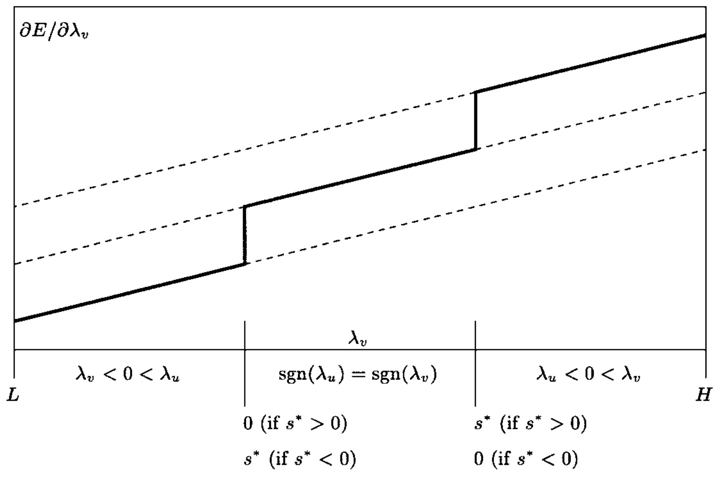
$\partial E/\partial \lambda_v$
λ_v
$\lambda_v < 0 < \lambda_u$
$\mathrm{sgn}(\lambda_u) = \mathrm{sgn}(\lambda_v)$
$\lambda_u < 0 < \lambda_v$
L
H
0 (if $s^* > 0$)
s^* (if $s^* < 0$)
s^* (if $s^* > 0$)
0 (if $s^* < 0$)

Figure 1. The derivative as a function of λ_v: If the kernel function obeys Mercer's condition, then the derivative (Eq. (13)) will always be strictly increasing.

Table 1 shows pseudo-code that implements a single step for SMO with regression. Basically, lines 4 and 5 set the Lagrange multipliers to values appropriate if the two have the same sign. If the two multipliers differ in sign, then line 8 adjusts the multipliers by $2\varepsilon/\eta$ if the adjustment can be made without affecting the sign of either multiplier. If such an adjustment can not be made, the only solution is for the two multipliers to take the values of s^* and 0. Lines 12 and 13 calculate boundaries that keep both multipliers within $\pm C$. Finally, lines 14 and 15 enforces the constraints.

3.3. KKT and convergence conditions

The step described in this section will only minimize the global objective function if one or both of the two parameters violates a Karush-Kuhn-Tucker (KKT) condition. The KKT conditions for regression are:

$$\begin{aligned} \lambda_i = 0 &\Longleftrightarrow |y_i - f_i| < \varepsilon \\ -C < \lambda_i \neq 0 < C &\Longleftrightarrow |y_i - f_i| = \varepsilon \\ |\lambda_i| = C &\Longleftrightarrow |y_i - f_i| > \varepsilon. \end{aligned} \tag{16}$$

These KKT conditions also yield a test for convergence. When no parameter violates any KKT condition, then the global minimum has been reached.

Also note, a step should only be taken if it is greater in size than some threshold. In our experience, we found that a threshold equal to the square root of the machine epsilon consistently gave stable results, but that a smaller threshold could introduce numerical instabilities. We also found that our update rule (as well as the update rules proposed by

Table 1. Pseudo-code for the analytical step for SMO generalized for regression.

```
1.   $s^* = \lambda_u^* + \lambda_v^*$;
2.   $\eta = k_{vv} + k_{uu} - 2k_{uv}$;
3.   $\Delta = 2\varepsilon/\eta$;
4.   $\lambda_v = \lambda_v^* + \frac{1}{\eta}(y_v - y_u + f_u^* - f_v^*)$;
5.   $\lambda_u = s^* - \lambda_v$;
6.   if $(\lambda_u \cdot \lambda_v < 0)$ {
7.       if $(|\lambda_v| \geq \Delta \ \wedge \ |\lambda_u| \geq \Delta)$
8.           $\lambda_v = \lambda_v - \mathrm{sgn}(\lambda_v) \cdot \Delta$;
9.       else
10.          $\lambda_v = \mathrm{step}(|\lambda_v| - |\lambda_u|) \cdot s^*$;
11.  }
12.  $L = \max(s^* - C, -C)$;
13.  $H = \min(C, s^* + C)$;
14.  $\lambda_v = \min(\max(\lambda_v, L), H)$;
15.  $\lambda_u = s^* - \lambda_v$;
```

others (Mattera, Palmieri, and Haykin, 1999)) were more numerically stable if $\lambda_u > \lambda_v$, which can be easily enforced by selectively swapping the roles of the two multipliers.

3.4. Updating the threshold

To update the SVM threshold, we calculate two candidate updates. The first update, if used along with the new parameters, forces the SVM to have $f_u = y_u$. The second forces $f_v = y_v$. If neither update for the other two parameters hits a constraint, then the two candidate updates for the threshold will be identical. Otherwise, we average the candidate updates.

$$b^u = y_u - f_u^* + (\lambda_u^* - \lambda_u)k_{uu} + (\lambda_v^* - \lambda_v)k_{uv} + b^* \tag{17}$$

$$b^v = y_v - f_v^* + (\lambda_u^* - \lambda_u)k_{uv} + (\lambda_v^* - \lambda_v)k_{vv} + b^* \tag{18}$$

These update rules are nearly identical to Platt's original derivation.

3.5. Related work

While there are at least three other earlier works that derive regression rules for SMO, all differ in some respects from this work and from each other. In this subsection we briefly sample these related works.

Smola and Schölkopf (1998) derived regression rules for SMO that use four separate Lagrange multipliers, where one pair of multipliers forms a single composite parameter. This formulation is consistent with the earliest derivations of SVM regression in that it keeps all Lagrange multipliers strictly positive. Properly handling all special cases for the four

Lagrange multipliers is somewhat difficult as demonstrated by the two pages of pseudo-code required to describe the update rule in Smola and Schölkopf (1998).

Mattera, Palmieri, and Haykin (1999) compress two Lagrange multipliers into a single parameter, as is done in our derivation. We believe that their update rule is analytically identical to ours and have numerically observed this to be the case within machine epsilon; however, the update rule described in Mattera, Palmieri, and Haykin (1999) looks very different from ours. We believe that our derivation is simpler to implement and to understand.

Finally, Shevade et al. (2000) extended the derivation contained in Smola and Schölkopf (1998) to correct for an inefficiency in how SMO updates the threshold. While the inefficiency was first identified for the case of classification (Keerthi et al., 1999), it is applicable to most implementations of SMO, including ours. Future work may combine the improvements of Shevade et al. (2000) with our derivation.

4. Building a better SMO

As described in Section 2, SMO repeatedly finds two Lagrange multipliers that can be optimized with respect to each other and analytically computes the optimal step for the two Lagrange multipliers. Section 2 was concerned with the analytical portion of the algorithm. In this section, we concentrate on the remainder of SMO which consists of several heuristics that are used to pick pairs of Lagrange multipliers to optimize. While it is beyond the scope of this paper to give a complete description of SMO, Table 2 gives basic pseudo-code for the algorithm. For more information, consult one of Platt's papers (Platt, 1998, 1999b).

Referring to Table 2, notice that the first Lagrange multiplier to work on is chosen at line 3 and that its counterpart is chosen at lines 3.1, 3.2, or 3.3. SMO attempts to concentrate its effort where it is most needed by maintaining a working set of non-bounded Lagrange multipliers. The idea is that Lagrange multipliers that are at bounds (either 0 or C for classification, or 0 or $\pm C$ for regression) are mostly irrelevant to the optimization problem and will tend to keep their bounded values.

Table 2. Basic pseudo-code for SMO.

```
While further progress can be made:
1. If this is the first iteration, or if the previous iteration made no progress, then
   let the working set be all data points.
2. Otherwise, let the working set consist only of data points with non-bounded
   Lagrange multipliers.
3. For all data points in the working set, try to optimize the corresponding
   Lagrange multiplier. To find the second Lagrange multiplier:
   3.1 Try the best one (found from looping over the non-bounded multipliers)
       according to Platt's heuristic, or
   3.2 Try all among the working set, or
   3.3 Try to find one among the entire set of Lagrange multipliers.
4. If no progress was made and the working set was all data points, then done.
```

At best, each optimization step will take time proportional to the number of Lagrange multipliers in the working set and, at worst, will take time proportional to the size of the entire data set. However, the runtime is actually much slower than this analysis implies because each candidate for the second Lagrange multiplier requires three kernel functions to be evaluated. If the input dimensionality is large, then the kernel evaluations may be a significant factor in the time complexity. All told, we can express the runtime of a single SMO step as $\Theta\left(p \cdot W \cdot n + (1-p) \cdot \ell \cdot n\right)$, where p is the probability that the second Lagrange multiplier is in the working set, W is the size of the working set, and n is the input dimensionality.

The goal of this section is to reduce the runtime complexity for a single SMO step down to $\Theta(p' \cdot W + (1-p') \cdot \ell \cdot n)$, with $p' > p$. Additionally, a method for reducing the total number of required SMO steps is also introduced, so we also reduce the cost of the outer most loop of SMO as well. Over the next five subsections, several improvements to SMO will be described. The most fundamental change is to cache the kernel function outputs. However, a naive caching policy actually slows down SMO since the original algorithm tends to randomly access kernel outputs with high frequency. Other changes are designed either to improve the probability that a cached kernel output can be used again or to exploit the fact that kernel outputs have been precomputed.

4.1. Caching kernel outputs

A cache is typically understood to be a small portion of memory that is faster than normal memory. In this work, we use *cache* to refer to a table of precomputed kernel outputs. The idea here is that frequently accessed kernel outputs should be stored and reused to avoid the cost of recomputation.

Our cache data structure contains an inverse index, I, with M entries such that I_i refers to the index (in the main data set) of the ith cached item. We maintain a two-dimensional $M \times M$ array to store the cached values. Thus, for any $1 < i,\ j < M$ with $a = I_i$ and $b = I_j$, we either have the precomputed value of k_{ab} stored in the cache or we have space allocated for that value and a flag set to indicate that the kernel output needs to be computed and saved.

The cache can have any of the following operations applied to it:

- `query(a,b)`—returns one of three values to indicate that k_{ab} is either (1) not in the cache, (2) allocated for the cache but not present, or (3) in the cache.
- `insert(a,b)`—compute k_{ab} and force it into the cache, if it is not present already. The least recently used indices in I are replaced by a and b.
- `access(a,b)`—return k_{ab} by the fastest method available.
- `tickle(a,b)`—mark indices a and b as the most recently used elements.

We use a least recently used policy for updating the cache as would be expected but with the following exceptions:

– For all i, k_{ii} is maintained in its own separate space since it is accessed so frequently.
– If SMO's working set is all Lagrange multipliers (as determined in step 1 of Table 2), then all accesses to the cache are done without tickles and without inserts.
– If the working set is a proper subset and both requested indices are not in the working set, then the access is done with neither a tickle nor an insert.

Without the above modifications, caching kernel outputs in SMO usually degrades the runtime because of the frequency of cache misses and the extra overhead incurred. Our modified caching policy makes caching beneficial; however, the next set of heuristics can improve the effectiveness of caching even more.

4.2. Eliminating thrashing

As shown in lines 3.1, 3.2, and 3.3 of Table 2, SMO uses a hierarchy of selection methods in order to find a second multiplier to optimize along with the first. It first tries to find a very good one with a heuristic. If that fails, it settles for anything in the working set. But if that fails, SMO then starts searching through the entire training set.

Line 3.3 causes problems in SMO for two reasons. First, it entails an extreme amount of work that results in only two multipliers changing. Second, if caching is used, line 3.3 could interfere with the update policies of the cache.

To avoid these problems, we use a heuristic which entails a modification to SMO such that line 3.3 is executed only if the working set is the entire data set. We must execute it, in this case, to be sure that convergence is achieved. Platt (1999a) has proposed a modification with a similar goal in mind.

In our example source code (which can be accessed via the URL given at the beginning of this paper) this heuristic corresponds to using the command line option `-lazy`, which is short for "lazy loops."

4.3. Optimal steps

The next modification to SMO takes advantage of the fact that cached kernel outputs can be accessed in constant time. Line 3.1 of Table 2 searches over the entire working set and finds the multiplier that approximately yields the largest step size. However, if kernel outputs for two multipliers are cached, then computing the change to the objective function that results from optimizing the two multipliers takes constant time to calculate. Thus, by exploiting the cached kernel outputs, we can greedily take the step that yields the most improvement, among all analytical steps possible that use only cached kernel values.

Let λ_v be the first multiplier selected in line 3 of Table 2. For all u such that k_{uv} is cached, we can calculate new values for the two multipliers analytically and in constant time. Let the old values for multipliers use $*$ superscripts, as in λ_u^* and λ_v^*. Moreover, let f_i and f_i^* be shorthand for the new and old values for the SVM output.[1]

The change to the classification objective function (Eq. (3)) that results from accepting the new multipliers is:

$$\begin{aligned}\Delta W = {} & \lambda_u y_u\left(f_u - \frac{1}{2}y_u\lambda_u k_{uu} - b\right) - \lambda_u^* y_u\left(f_u^* - \frac{1}{2}y_u\lambda_u^* k_{uu} - b^*\right)\\ & + \lambda_v y_v\left(f_v - \frac{1}{2}y_v\lambda_v k_{vv} - b\right) - \lambda_v^* y_v\left(f_v^* - \frac{1}{2}y_v\lambda_v^* k_{vv} - b^*\right)\\ & + k_{uv}(y_u y_v\lambda_u\lambda_v - y_u y_v\lambda_u^*\lambda_v^*) - \lambda_u - \lambda_v + \lambda_u^* + \lambda_v^*.\end{aligned} \tag{19}$$

Equation (19) is derived by substituting λ for α in Eq. (3), and rewriting the equation so that all terms are trivially dependent or independent of λ_u and/or λ_v. Afterwards, the difference between two choices for the two multipliers can be calculated without any summations because the independent terms cancel.

The change to the regression objective function (Eq. (9)) can be similarly calculated with:

$$\begin{aligned}\Delta W = {} & \lambda_u\left(f_u - y_u - \frac{1}{2}\lambda_u k_{uu} - b\right) - \lambda_u^*\left(f_u^* - y_u - \frac{1}{2}\lambda_u^* k_{uu} - b^*\right)\\ & + \lambda_v\left(f_v - y_v - \frac{1}{2}\lambda_v k_{vv} - b\right) - \lambda_v^*\left(f_v^* - y_v - \frac{1}{2}\lambda_v^* k_{vv} - b^*\right)\\ & + k_{uv}(\lambda_u^*\lambda_v^* - \lambda_u\lambda_v) + \varepsilon(|\lambda_u| - |\lambda_u^*| + |\lambda_v| - |\lambda_v^*|).\end{aligned} \tag{20}$$

Thus, we modify SMO by replacing line 3.1 in Table 2 with code that looks for the best second multiplier via Eq. (19) or (20) for all u such that k_{uv} is cached.

In our example source code, this heuristic corresponds to using the command line option `-best`, which is short for "best step."

4.4. On demand incremental SVM outputs

The next modification to SMO is a method to calculate SVM outputs more rapidly. Without loss of generality, assume we have an SVM that is used for classification and that the output of the SVM is determined from Eq. (2) (but with λ substituted for α). There are at least three different ways to calculate the SVM outputs after a single Lagrange multiplier, λ_j, has changed:

– Use Eq. (2), which is extremely slow.
– Change Eq. (2) so that the summation is only over the nonzero Lagrange multipliers.
– Incrementally update the new value with $f_i = f_i^* + (\lambda_j - \lambda_j^*)y_j k_{ij}$.

Clearly, the last method is the fastest. SMO, in its original form, uses the third method to update the outputs whose multipliers are non-bounded (which are needed often) and the second method when an output is needed that has not been incrementally updated.

We can improve on this method by only updating outputs when they are needed, and by computing which of the second or third method above is more efficient. To do this, we need two queues with maximum sizes equal to the number of Lagrange multipliers and a third array to store a time stamp for when a particular output was last updated. Whenever a Lagrange multiplier changes value, we store the change to the multiplier and the change to b in the queues, overwriting the oldest value.

When a particular output is required, if the number of time steps that have elapsed since the output was last updated is less than the number of nonzero Lagrange multipliers, we can calculate the output from its last known value and from the changed values in the queues. However, if there are fewer nonzero Lagrange multipliers, it is more efficient to update the output using the second method.

Since the outputs are updated on demand, if the SVM outputs are accessed in a nonuniform manner, then this update method will exploit those statistical irregularities. In our example source code, this heuristic corresponds to using the command line option `-clever`, which is short for "clever outputs."

4.5. SMO with decomposition

Using SMO with caching along with all of the proposed heuristics yields a significant runtime improvement as long as the cache size is nearly as large as the number of support vectors in the solution. When the cache size is too small to fit all of the kernel outputs for each support vector pair, accesses to the cache will fail and runtime will be increased. This particular problem can be addressed by combining Osuna's decomposition algorithm (Osuna, Freund, & Girosi, 1997) with SMO.

The basic idea is to iteratively build an $M \times M$ subproblem with $2 < M < \ell$, solve the subproblem, and then iterate with a new subproblem until the entire optimization problem is solved. However, instead of using a QP solver to solve the subproblem, we use SMO and choose M to be as large as the cache.

The benefits of this combination are two-fold. First, much evidence indicates that decomposition can often be faster than using a QP solver. Since the combination of SMO and decomposition is functionally identical to standard decomposition with SMO as the QP solver, we should expect the same benefit. Second, using a subproblem that is the same size as the cache guarantees that all of the kernel outputs required will be available at every SMO iteration except for the first for each subproblem.

However, we note that our implementation of decomposition is very naive in the way it constructs subproblems, since it essentially works on the first M randomly selected data points that violate a KKT condition. In our example source code, this heuristic corresponds to using the command line option `-ssz` M, which is short for "subset size."

5. Experimental results

To evaluate the effectiveness of our modifications to SMO we chose the Mackey-Glass system (Mackey & Glass, 1977) as a test case because it is highly chaotic (making it a challenging regression problem) and well-studied. The Mackey-Glass system is described by the delay-differential equation:

$$\frac{dx}{dt} = \frac{A \cdot x(t-\tau)}{1 + x^{10}(t-\tau)} - B \cdot x(t).$$

For all experiments, we used the parameter settings $\tau = 17$, $A = 0.2$, $B = 0.1$, and $\Delta t = 1$ (for numerical integration), which yields a very chaotic time series with an embedding dimension of 4.

To perform forecasting, we use a time-delay embedding (Takens, 1980) to approximate the map:

$$x(t+T) \leftarrow \{x(t), x(t-\Delta), \ldots, x(t-(n-1)\Delta)\},$$

with $T = 85$, $\Delta = 6$, and n equal to 4, 6, or 8. Thus, we are predicting 85 time steps into the future with an SVM with 4, 6, or 8 inputs.

The purpose of this work is *not* to evaluate the prediction accuracy of SVMs on chaotic time series as has been done before (Müller et al., 1997; Mukherjee, Osuna, & Girosi, 1997). Our focus is on the amount of time required to optimize a support vector machine. Since the objective function for optimizing SVMs is quadratic with linear constraints, all SVMs will have either a single global minimum or a collection of minima that are identical in the objective function valuation. Hence, excepting minor numerical differences between implementations, all SVM optimization routines essentially find the same solution; they only differ in how they find the solution, how long it takes to get there, and how much memory is required.

Figure 2 shows plots from two training runs that illustrate the Mackey-Glass time series and phase-space. The time series plots show predictions for the two values of ε, and the phase-space plots show the location of the support vectors in a two dimensional slice of the time-delay embedding.

The first part of our experimental results are summarized in Tables 3 and 4. In these experiments, the time series consisted of 500 data points which, depending on the values of n, Δ, and T, yield a number of exemplars less than 500. The major blocks (three in each table) summarize a specific problem instance which has a unique set of values for ε and n. Within each block, we performed all combinations of using SMO with and without caching, with and without decomposition, and with and without all three heuristics.

Each block in the tables also contains results from using the Royal Holloway/AT&T/GMD FIRST SV Machine code (RAGSVM) (Saunders et al., 1998). RAGSVM can work with three different optimization packages, but only one optimizer is freely available for research that can be used on regression problems: BOTTOU, an implementation of the conjugate gradient method. Entries in the blocks labelled as "QP" use RAGSVM with BOTTOU without the chunking option. Entries labelled as "Chunk" use "sporty chunking" which uses the decomposition method with the specified subset size and the QP solver on the subproblem.

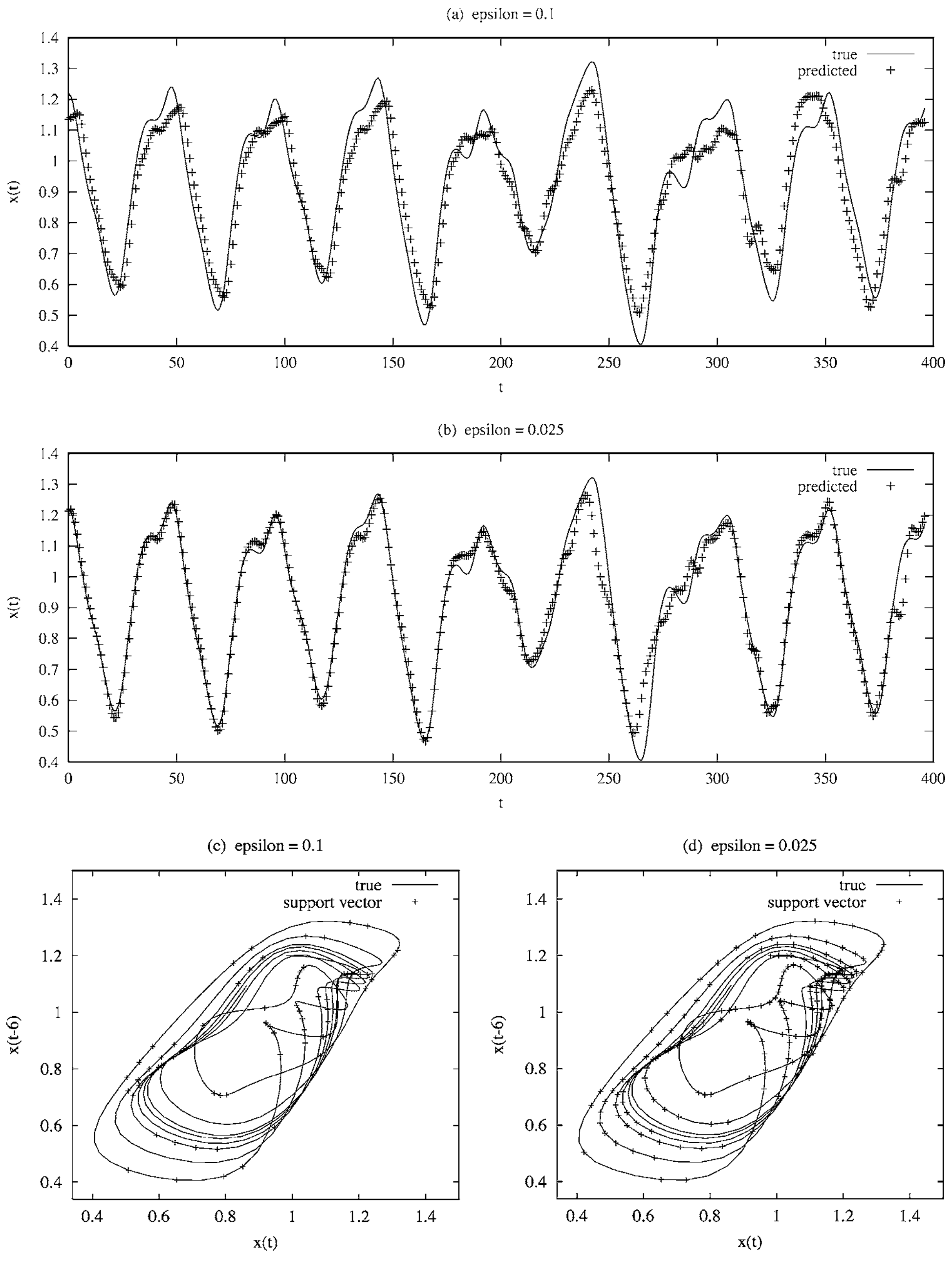

Figure 2. The Mackey-Glass system: actual and predicted time series for (a) $\varepsilon = 0.1$, and (b) $\varepsilon = 0.025$; two-dimensional phase space plots to show the location of support vectors for (c) $\varepsilon = 0.1$, and (d) $\varepsilon = 0.025$.

Table 3. Experimental results (part 1/2): all SMO results are averaged over ten trials.

Train alg.	Sub. size	Cache size	SMO options	Objective value	Num. of SVs	CPU time	Std. dev.
			Problem instance: $n = 4$, $\varepsilon = 0.1$				
SMO	0	0	none	−15.9206	70.1	104.17	14.14
SMO	0	0	all	−15.9266	68.3	9.87	1.37
SMO	0	100	none	−15.9206	70.1	32.88	4.28
SMO	0	100	all	−15.9266	68.3	6.10	0.82
SMO	100	0	none	−15.9198	70.5	136.83	24.35
SMO	100	0	all	−15.9247	67.9	12.65	1.73
SMO	100	100	none	−15.9198	70.5	42.72	7.53
SMO	100	100	all	−15.9247	67.9	7.64	1.04
QP	–	–	–	−15.9002	63	85.22	–
Chunk	100	–	–	−15.8809	59	20.24	–
			Problem instance: $n = 6$, $\varepsilon = 0.1$				
SMO	0	0	none	−5.4613	62.7	149.72	36.91
SMO	0	0	all	−5.4649	62.9	11.82	1.60
SMO	0	100	none	−5.4613	62.7	35.49	8.25
SMO	0	100	all	−5.4649	62.9	6.26	0.80
SMO	100	0	none	−5.4620	63.2	153.61	27.71
SMO	100	0	all	−5.4636	62.7	11.89	1.93
SMO	100	100	none	−5.4620	63.2	35.70	6.25
SMO	100	100	all	−5.4636	62.7	6.06	0.93
QP	–	–	–	−5.4698	59	62.86	–
Chunk	100	–	–	−5.4619	55	16.43	–
			Problem instance: $n = 8$, $\varepsilon = 0.1$				
SMO	0	0	none	−2.3008	54.5	63.75	12.56
SMO	0	0	all	−2.3027	52.8	6.33	1.27
SMO	0	100	none	−2.3008	54.5	13.90	2.38
SMO	0	100	all	−2.3027	52.8	3.43	0.59
SMO	100	0	none	−2.3005	55.4	67.73	19.18
SMO	100	0	all	−2.3031	53	7.21	1.42
SMO	100	100	none	−2.3005	55.4	13.65	3.80
SMO	100	100	all	−2.3031	53	3.45	0.59
QP	–	–	–	−2.2950	51	40.86	–
Chunk	100	–	–	−2.2899	38	6.30	–

Entries where heuristics have the value of "all" indicate that "lazy loops" (from Section 4.2), "best step" (Section 4.3), and "clever outputs" (Section 4.4) are all used. The entries for the subset size indicate the size for decomposition (with "0" meaning no decomposition). All times are in CPU seconds on a 500 MHz Pentium III machine running Linux.

Table 4. Experimental results (part 2/2): all SMO results are averaged over ten trials.

Train alg.	Sub. size	Cache size	SMO options	Objective value	Num. of SVs	CPU time	Std. dev.
			Problem instance: $n = 4$, $\varepsilon = 0.025$				
SMO	0	0	none	−84.0185	195.7	316.70	36.10
SMO	0	0	all	−84.0367	194.9	25.79	2.51
SMO	0	100	none	−84.0185	195.7	98.04	12.63
SMO	0	100	all	−84.0367	194.9	16.69	1.47
SMO	100	0	none	−84.0284	196.8	554.69	73.18
SMO	100	0	all	−83.8655	195.3	58.33	6.54
SMO	100	100	none	−84.0284	196.8	184.12	22.32
SMO	100	100	all	−83.8655	195.3	40.60	5.69
QP	–	–	–	−84.0401	194	186.63	–
Chunk	100	–	–	−84.0290	188	316.54	–
			Problem instance: $n = 6$, $\varepsilon = 0.025$				
SMO	0	0	none	−48.1057	170.1	724.51	103.24
SMO	0	0	all	−48.1256	169.8	48.17	4.17
SMO	0	100	none	−48.1057	170.1	157.97	21.10
SMO	0	100	all	−48.1256	169.8	25.43	2.15
SMO	100	0	none	−48.1120	170.5	1149.43	127.18
SMO	100	0	all	−48.1283	169	124.63	17.65
SMO	100	100	none	−48.1120	170.5	278.76	31.27
SMO	100	100	all	−48.1283	169	75.09	14.38
QP	–	–	–	−48.1430	159	245.67	–
Chunk	100	–	–	−48.1505	164	310.21	–
			Problem instance: $n = 8$, $\varepsilon = 0.025$				
SMO	0	0	none	−27.8433	159.6	977.78	146.42
SMO	0	0	all	−27.8588	155.1	67.20	10.57
SMO	0	100	none	−27.8433	159.6	261.98	87.75
SMO	0	100	all	−27.8588	155.1	34.48	4.31
SMO	100	0	none	−27.8421	164.9	1338.15	133.58
SMO	100	0	all	−27.8663	159.8	141.33	19.54
SMO	100	100	none	−27.8421	164.9	289.26	29.13
SMO	100	100	all	−27.8663	159.8	76.66	12.69
QP	–	–	–	−27.8958	149	257.40	–
Chunk	100	–	–	−27.8941	144	329.91	–

Entries where heuristics have the value of "all" indicate that "lazy loops" (from Section 4.2), "best step" (Section 4.3), and "clever outputs" (Section 4.4) are all used. The entries for the subset size indicate the size for decomposition (with "0" meaning no decomposition). All times are in CPU seconds on a 500 MHz Pentium III machine running Linux.

Table 5. Experimental results for problem instance $d = 4$, $\varepsilon = 0.1$, with 10,000 data points in the time series.

Train alg.	Sub. size	Cache size	SMO options	Objective value	Num. of SVs	CPU time	Std. dev.
SMO	0	500	all	−93.9007	396.60	1047.29	45.13
SMO	500	500	all	−93.8975	393.25	625.45	295.85
Chunk	500	–	–	−87.2486	287	9314.89	–

SMO statistics are over four trials. All times are in CPU seconds on a 500 MHz Pentium III machine running Linux.

In general, the training runs were configured as similarly as possible, each using Gaussian kernels of the form:

$$K(\boldsymbol{x}_u, \boldsymbol{x}_v) = \exp(-\|\boldsymbol{x}_u - \boldsymbol{x}_v\|^2/\sigma^2)$$

with $\sigma = \frac{1}{2}$, and $C = 10$. SMO in all configurations produces results nearly identical to RAGSVM with respect to the value of the objective function found. However, the run times are dramatically different for the two implementations. For these sets of experiments, SMO with caching and the heuristics consistently gave the fastest run times, often performing orders of magnitude faster than regular SMO, QP, and decomposition. The speed improvements for SMO ranged from a factor of 3 to as much as 25.

Interestingly, on these experiments, SMO with decomposition consistently yielded inferior run times compared to SMO without decomposition, regardless of other runtime options. Our motivation for combining SMO with decomposition was to make caching effective on problems with many data points. Since the first set of experiments only used 500 data points, we used the same Mackey-Glass parameters to generate a time series with 10,000 data points for further experimentation.

Table 5 summarizes the second set of experiments. For these experiments, we chose to only vary whether SMO was used with or without decomposition. As can be seen in the table, SMO without decomposition gives nearly an order of magnitude improvement in runtime compared to RAGSVM while SMO with decomposition yields even faster run times. However, SMO with decomposition yields a very high standard deviation with the fastest and slowest run times being 391 and 1123 seconds, respectively. We suspect that the high standard deviation is a result of our naive implementation of decomposition. Nevertheless, the worst case for SMO with decomposition is nearly as good as the best for SMO without decomposition. Moreover, on this problem set, SMO with decomposition can be nearly 25 times faster than decomposition with a QP solver. In fact, the solutions found by SMO in all experiments from Table 5 are superior to the RAGSVM solutions in that the final objective function values are significantly larger in magnitude in the SMO runs.

But why does SMO with decomposition help on large data sets? For a caching policy to be effective, the cached elements must have a relatively high probability of being reused before they are replaced. On large data sets, this goal is far more difficult to achieve. Moreover, SMO must periodically loop over all exemplars in order to check for convergence. Using SMO with decomposition makes caching much easier to implement effectively because we can make the subset size for decomposition the same size as the cache, thus guaranteeing that cached elements can be reused with high probability.

6. Conclusions

This work has shown that SMO can be generalized to handle regression and that the runtime of SMO can be greatly improved for datasets dense with support vectors. Our main improvement to SMO has been to implement caching along with heuristics that assist the caching policy. In general, the heuristics are designed to either improve the probability that cached kernel outputs will be used or exploit the fact that cached kernel outputs can be used in helpful ways that are inefficient for non-cached kernel outputs. Our numerical results show that our modifications to SMO yield dramatic runtime improvements. Moreover, our implementation of SMO can outperform state-of-the-art SVM optimization packages that use a conjugate gradient QP solver and decomposition.

Because kernel evaluations are more expensive the higher the input dimensionality, we believe, but have not shown, that our modifications to SMO will be even more valuable on larger datasets with high input dimensionality. Preliminary results indicate that these changes can greatly improve the performance of SMO on classification tasks that involve large, high-dimensional, and non-sparse data sets.

Future work will concentrate on incremental methods that gradually increase numerical accuracy. We also believe that the improvements to SMO described in Keerthi et al. (1999), Shevade et al. (2000) can be adapted to our implementation as well. Moreover, altering our decomposition scheme to better mimic the heuristics in Joachims (1999) should yield further improvements.

Acknowledgments

We thank Tommy Poggio, John Platt, Edgar Osuna, Constantine Papageorgious, and Sayan Mukherjee for helpful discussions. We also acknowledge many helpful comments from the anonymous reviewers. Special thanks to Tommy Poggio and the Center for Biological and Computational Learning at MIT for hosting the first author during this research.

Note

1. Note that in this section, we refer to all Lagrange multipliers by λ and not α. We do this to maintain consistency with earlier sections, even though this notation conflicts with Eqs. (3) and (6).

References

Burges, C. (1998). A Tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery, 2:2*, 955–974.

Friess, T., Cristianini, N., & Campbell, C. (1998). The Kernel-adatron: A fast and simple learning procedure for support vector machines. In J. Shavlik (Ed.), *Proceedings of the Fifteenth International Conference on Machine Learning* (pp. 188–196).

Joachims, T. (1999). Making large-scale support vector machine learning practical. In B. Schölkopf, C. Burges, & A. Smola (Eds.), *Advances in kernel methods—Support vector learning* (pp. 169–184). MIT Press.

Keerthi, S. S., Shevade, S., Bhattacharyya, C., & Murthy K. R. K. (1999). Improvements to Platt's SMO algorithm for SVM classifier design. Technical Report CD-99-14, Dept. of Mechanical and Production Engineering, National University of Singapore.

Mackey, M. C. & Glass, L. (1977). Oscillation and chaos in physiological control systems. *Science, 2:4300*, 287–289.

Mangasarian, O. L. & Musicant, D. R. (1999) Successive overrelaxation for support vector machines. *IEEE Transactions on Neural Networks, 10:5*, 1032–1037.

Mattera, D., Palmieri, F., & Haykin, S. (1999). An explicit algorithm for training support vector machines. *IEEE Signal Processing Letters, 6:9*, 243–245.

Mukherjee, S., Osuna, E., & Girosi, F. (1997). Nonlinear prediction of chaotic time series using support vector machines. In *Proc. of IEEE NNSP'97* (pp. 511–519).

Müller, K., Smola, A., Rätsch, G., Schölkopf, B., Kohlmorgen, J., & Vapnik, V. (1997). Predicting time series with support vector machines. In W. Gerstner, A. Germond, M. Hasler, & J.-D. Nicoud (Eds.), *Artificial neural networks—ICANN'97*, Vol. 1327 of *Springer Lecture Notes in Computer Science* (pp. 999–1004). Berlin.

Osuna, E., Freund, R., & Girosi, F. (1997). An improved training algorithm for support vector machines. In *Proc. of IEEE NNSP'97*.

Platt, J. (1998). Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. Burges, & A. Smola (Eds.), *Advances in Kernel methods—support vector learning*. Cambridge, MA: MIT Press.

Platt, J. (1999a). Private communication.

Platt, J. (1999b). Using sparseness and analytic QP to speed training of support vector machines. In M. S. Kearns, S. A. Solla, & D. A. Cohn (Eds.), *Advances in neural information processing systems 11*. Cambridge, MA: MIT Press.

Saunders, C., Stitson, M. O., Weston, J., Bottou, L., Schölkopf, B., & Smola, A. (1998). Support vector machine reference manual. Technical Report CSD-TR-98-03, Royal Holloway, University of London.

Shevade, S. K., Keerthi, S. S., Bhattacharyya, C., & Murthy, K. R. K. (2000). Improvements to the SMO algorithms for SVM regression. *IEEE Transactions on Neural Networks, 11:5*, 1188–1193.

Smola, A. & Schölkopf, B. (1998). A tutorial on support vector regression. Technical Report NC2-TR-1998-030, NeuroCOLT2.

Takens, F. (1980). Detecting strange attractors in turbulence. In D. A. Rand & L. S. Young (Eds.), *Dynamical systems and turbulence* (pp. 366–381). New York: Spinger-Verlag.

Vapnik, V. (1995). *The nature of statistical learning theory*. New York: Springer Verlag.

Received March 17, 2000
Revised Sept. 21, 2000
Accepted Oct. 30, 2000
Final manuscript Sept. 7, 2001