Gary William Flake and Steve Lawrence. *Efficient SVM Regression Training with SMO*, accepted for publication, Machine Learning, 2001.

Efficient SVM Regression Training with SMO

Gary William Flake
flake@research.nj.nec.com

Steve Lawrence
lawrence@research.nj.nec.com

NEC Research Institute
4 Independence Way
Princeton, NJ 08540

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 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 problem with several modifications that enable caching to be effectively used with SMO. For regression problems, our modifications improve converge 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 [7]. 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, a theorem [2] was proved that introduced a whole new family of SVM training procedures. In a nutshell, Osuna's theorem showed that the global SVM training problem can be broken down into a sequence of smaller subproblems and that optimizing each subproblem minimizes the original QP problem. Even more recently, the sequential minimal optimization algorithm (SMO) was introduced [3, 5] as an extreme example of Osuna's theorem in practice. 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.

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 no 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, $\{(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_N, y_N)\}$, such that $\boldsymbol{x}_i \in \mathbb{R}^d$ is an input and y_i is a target output. 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 $f(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{x} \cdot \boldsymbol{w} + w_0$, and the optimization task is defined as:

$$\text{minimize } \frac{1}{2}||\boldsymbol{w}||^2, \text{ subject to } y_i(\boldsymbol{x} \cdot \boldsymbol{w} + w_0) \geq 1 \ \forall i \tag{1}$$

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 [1] and [6] 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 Equation 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 $y_i \in \{-1, 1\}$. These objective functions will always have a component that should be minimizes and linear constraints that must be obeyed. To optimize the objective function, one converts it into the primal Lagrangian

form which contains the minimization terms minus the linear constraints multiplied by Lagrangian multipliers. The primal Lagrangian is converted into a dual Lagrangian where the only free parameters are the Lagrangian multipliers. In the dual form, the objective function is quadratic in the Lagrangian multipliers; thus, 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 Lagrangian objective functions. Thus, we now define the output function of nonlinear SVMs for classification and regression, as well as the primal Lagrangian objective functions that they are optimized with respect to.

In the case of classification, with $y_i = \pm 1$, the output of an SVM is defined as:

$$f(\boldsymbol{x}, \boldsymbol{\alpha}) = \sum_{i=1}^{N} y_i \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}) + \alpha_0, \tag{2}$$

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

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

subject to the box constraint $0 \leq \alpha_i \leq C, \forall_i$ and the linear constraint $\sum_{i=1}^{N} y_i \alpha_i = 0$. 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:

$$E(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - f(\boldsymbol{x}_i, \boldsymbol{w})|_\epsilon + ||\boldsymbol{w}||^2, \tag{4}$$

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

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

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

$$f(\boldsymbol{x}, \boldsymbol{\alpha}^*, \boldsymbol{\alpha}) = \sum_{i=1}^{N} (\alpha_i^* - \alpha_i) K(\boldsymbol{x}_i, \boldsymbol{x}) + \alpha_0. \tag{5}$$

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

The primal form of Equation 4 is written as

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

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

$$\sum_{i=1}^{N}(\alpha_i^* - \alpha_i) = 0, \quad 0 \leq \alpha_i^*, \alpha_i \leq C, \forall_i. \tag{7}$$

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 primal Lagrangians in Equations 3 and 6, and the SVM output functions in Equations 2 and 5.

3 SMO AND REGRESSION

As mentioned earlier, SMO is a 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 multiplier 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. Instead, the reader should consult one of Platt's papers [3, 5] for more information.

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.

3.1 Step Size Derivation

We begin by transforming Equations 5,6, and 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 $-C \leq \lambda_i \leq C, \forall_i$. 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:

$$f(\boldsymbol{x}, \boldsymbol{\lambda}) = \sum_{i=1}^{N} \lambda_i K(\boldsymbol{x}_i, \boldsymbol{x}) + \lambda_0, \text{ and} \tag{8}$$

$$L_p(\boldsymbol{\lambda}) = \epsilon \sum_{i=1}^{N} |\lambda_i| - \sum_{i=1}^{N} \lambda_i y_i + \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j k_{ij}, \tag{9}$$

with the linear constraint $\sum_{i=1}^{N} \lambda_i = 0$. Our goal is to express analytically the minimum of Equation 9 as a function of two parameters. Let these two parameters have indices a and b so that λ_a and λ_b are the two unknowns. We can rewrite Equation 9 as

$$\begin{aligned} L_p(\lambda_a, \lambda_b) = {} & \epsilon |\lambda_a| + \epsilon |\lambda_b| - \lambda_a y_a - \lambda_b y_b + \frac{1}{2}\lambda_a^2 k_{aa} + \frac{1}{2}\lambda_b^2 k_{bb} + \\ & \lambda_a \lambda_b k_{ab} + \lambda_a v_a^* + \lambda_b v_b^* + E_c, \end{aligned} \tag{10}$$

where E_c is a term that is strictly constant with respect to λ_a and λ_b, and v_i^* is defined as

$$v_i^* = \sum_{j \neq a,b} \lambda_j^* k_{ij} = f_i^* - \lambda_a^* k_{ai} - \lambda_b^* k_{bi} - \lambda_0^* \tag{11}$$

with $f_i^* = f(\boldsymbol{x}_i, \boldsymbol{\lambda}^*)$. 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, $\sum_{i=1}^{N} \lambda_i = 0$, is true prior to any change to λ_a and λ_b, then in order for the constraint to be true after a step in parameter space, the sum of λ_a and λ_b must be held fixed. With this in mind, let $s^* = \lambda_a + \lambda_b = \lambda_a^* + \lambda_b^*$. We can now rewrite Equation 10 as a function of a single Lagrange multiplier by substituting $s^* - \lambda_b = \lambda_a$:

$$\begin{aligned} L_p(\lambda_b) &= \epsilon\,|s^* - \lambda_b| + \epsilon\,|\lambda_b| - (s^* - \lambda_b)y_a - \lambda_b y_b + \frac{1}{2}(s^* - \lambda_b)^2 k_{aa} \\ &\quad + \frac{1}{2}\lambda_b^2 k_{bb} + (s^* - \lambda_b)\lambda_b k_{ab} + (s^* - \lambda_b)v_a^* + \lambda_b v_b^* + E_c. \end{aligned} \tag{12}$$

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

$$\begin{aligned} \frac{\partial L_p}{\partial \lambda_b} &= \epsilon(\mathrm{sgn}(\lambda_b) - \mathrm{sgn}(s^* - \lambda_b)) + y_a - y_b + \\ &\quad (\lambda_b - s^*)k_{aa} + \lambda_b k_{bb} + (s^* - 2\lambda_b)k_{ab} - v_a^* + v_b^* \end{aligned} \tag{13}$$

Now, setting Equation 13 to zero yields:

$$\begin{aligned} \lambda_b(k_{bb} + k_{aa} - 2k_{ab}) &= y_b - y_a + \epsilon(\mathrm{sgn}(\lambda_a) - \mathrm{sgn}(\lambda_b)) + s^*(k_{aa} - k_{ab}) + v_a^* - v_b^* \\ &= y_b - y_a + f_a^* - f_b^* + \epsilon(\mathrm{sgn}(\lambda_a) - \mathrm{sgn}(\lambda_b)) + \\ &\quad \lambda_a^* k_{aa} - \lambda_a^* k_{ab} + \lambda_b^* k_{aa} - \lambda_b^* k_{ab} - \\ &\quad \lambda_a^* k_{aa} - \lambda_b^* k_{ab} - \lambda_0^* + \lambda_a^* k_{ab} + \lambda_b^* k_{bb} + \lambda_0^* \\ &= y_b - y_a + f_a^* - f_b^* + \epsilon(\mathrm{sgn}(\lambda_a) - \mathrm{sgn}(\lambda_b)) + \\ &\quad \lambda_b^*(k_{bb} + k_{aa} - 2k_{ab}) \end{aligned} \tag{14}$$

From Equation 14, we can write a recursive update rule for λ_b in terms of its old value:

$$\lambda_b = \lambda_b^* + \frac{1}{\eta}\left[y_b - y_a + f_a^* - f_b^* + \epsilon(\mathrm{sgn}(\lambda_a) - \mathrm{sgn}(\lambda_b)),\right] \tag{15}$$

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

3.2 Finding Solutions

Figure 1 illustrates how the partial derivative (Equation 13) of the primal Lagrangian function with respect to λ_b behaves. If the kernel function of the SVM obeys Mercer's condition (as all common ones do), then we are guaranteed that $\eta = k_{bb} + k_{aa} - 2k_{ab} \geq 0$ will always be true. If η is strictly positive, then Equation 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 Equation 13. Three possible solutions correspond to using Equation 15 with $(\mathrm{sgn}(\lambda_a) - \mathrm{sgn}(\lambda_b))$

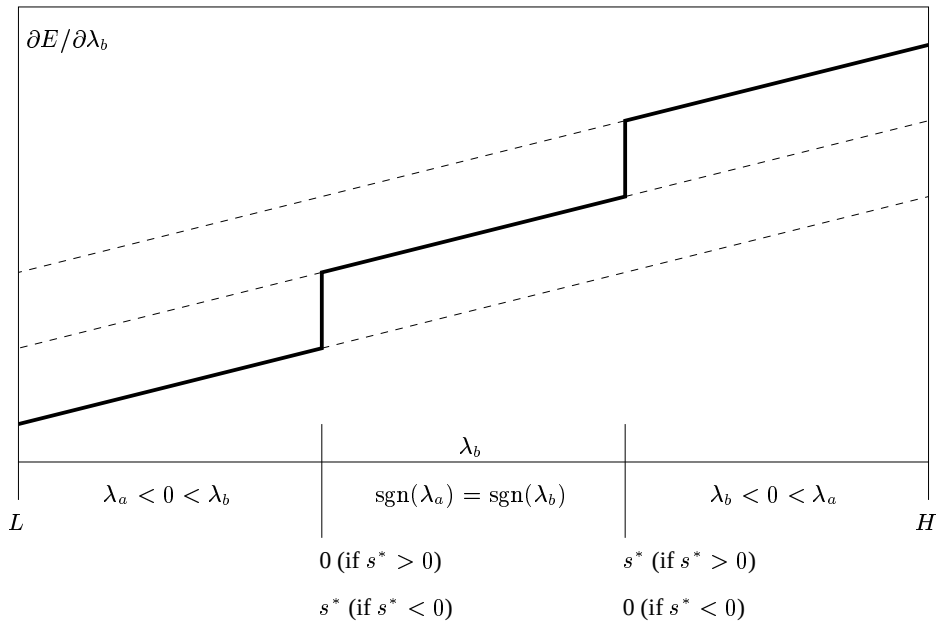

Figure 1: The derivative as a function of λ_b: If the kernel function obeys Mercer's condition, then the derivative (Equation 13) will always be strictly increasing.

set to -2, 0, and 2. The other two candidates correspond to setting λ_b to one of the transitions in Figure 1: $\lambda_b = 0$ or s^*.

We also need to consider how the linear and boxed constraints relate to one another. In particular, we need lower and upper bounds for λ_b that insure that both λ_a and λ_b are within the $\pm C$ range. Using:

$$L = \max(s^* - C, -C) \tag{16}$$
$$H = \min(C, s^* + C) \tag{17}$$

with L and H being the lower and upper bounds, respectively, guarantees that both parameters will obey the boxed constraints.

3.3 KKT Conditions

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

$$\begin{aligned} \lambda_i = 0 &\iff |y_i - f_i| < \epsilon \\ -C < \lambda_i \neq 0 < C &\iff |y_i - f_i| = \epsilon \\ |\lambda_i| = C &\iff |y_i - f_i| > \epsilon. \end{aligned} \tag{18}$$

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

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 $f_a = y_a$. The second forces $f_b = y_b$. 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.

$$\lambda_0^a = f_a^* - y_a + (\lambda_a^{\text{new}} - \lambda_a^{\text{old}})k_{aa} + (\lambda_b^{\text{new}} - \lambda_b^{\text{old}})k_{ab} + \lambda_0^{\text{old}} \quad (19)$$

$$\lambda_0^b = f_b^* - y_b + (\lambda_a^{\text{new}} - \lambda_a^{\text{old}})k_{ab} + (\lambda_b^{\text{new}} - \lambda_b^{\text{old}})k_{bb} + \lambda_0^{\text{old}} \quad (20)$$

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

3.5 Complete Update Rule

SMO can work on regression problems if the following steps are performed.

- Pick two parameters, λ_a and λ_b, such that at least one parameter violates a KKT condition as defined in Equation 18.
- Compute λ_b^{raw}
  - Try Equation 15 with $(\text{sgn}(\lambda_a) - \text{sgn}(\lambda_b))$ equal to -2, 0, and 2. If the new value is a zero to Equation 13, then accept that as the new value.
  - If the above step failed, try λ_b equal to 0 or s^*. Accept the value that has the property such that all positive (negative) perturbations yield a positive (negative) value for Equation 13.
- If $\lambda_b^{\text{raw}} > H$, set $\lambda_b^{\text{new}} = H$. If $\lambda_b^{\text{raw}} < L$, set $\lambda_b^{\text{new}} = L$. Otherwise, set $\lambda_b^{\text{new}} = \lambda_b^{\text{raw}}$.
- Set $\lambda_a^{\text{new}} = \lambda_a^{\text{old}} + \lambda_b^{\text{old}} - \lambda_b^{\text{new}}$.
- Set $\lambda_0^{\text{new}} = \frac{1}{2}(\lambda_0^a + \lambda_0^b)$ as specified in Equations 19 and 20.

The outer loop of SMO—that is, all of the non-numerical parts that make up the heuristics—can remain the same. As will be discussed in Section 5, further improvements can be made to SMO that improve the rate of convergence on regression problems by as much as an order of magnitude.

3.6 Simple Example

Figure 2 shows a plot of a scalar function which has been approximated with a nonlinear SVM. The SVM has a Gaussian kernel and was trained by the proposed algorithm with ϵ set to 0.1. The true function is shown with a solid line, the SVM approximation is shown in a dashed line, and the resulting support vectors (i.e., the input points corresponding to non-zero Lagrange multipliers) are shown as points. As can be seen, the approximation is very reasonable given the choice of ϵ.

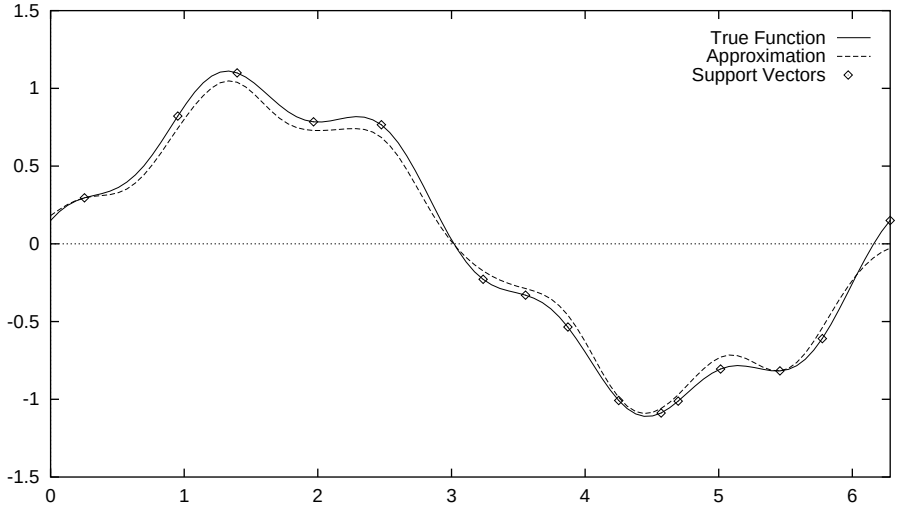

Figure 2: A simple SVM regression experiment: the true and approximated function are shown along with the calculated support vectors.

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 working set was all data points, then done.

Table 1: Basic Pseudo-code for SMO

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 1 gives basic pseudo-code for the algorithm. For more information, consult one of Platt's papers [3, 5].

Referring to Table 1, notice that the first Lagrange multiplier to work on is chosen at line 3; and its counterpart is chosen at line 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.

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 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 $O\left(p \cdot W \cdot D + (1-p) \cdot N \cdot D\right)$, where p is the probability that the second Lagrange multiplier is in the working set, W is the size of the working set, and D is the input dimensionality.

The goal of this section is to reduce the runtime complexity for a single SMO step down to $O\left(p' \cdot W + (1-p') \cdot N \cdot D\right)$, 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 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 1), 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 part of the working set, then the access is done with neither a tickle nor an insert.

As defined in this subsection, caching kernel outputs in SMO usually degrades the runtime because of the frequency of cache misses and the extra overhead incurred. This problem is even worse when the cache size is less than the number of support vectors in the solution.

4.2 Eliminating Thrashing

As shown in lines 3.1, 3.2, and 3.3 of Table 1, SMO uses a hierarchy of selection methods in order to find a second multiplier to optimize along with 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 [4] 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 end 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 1 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.

Let λ_b be the first multiplier selected in line 3 of Table 1. For all a such that k_{ab} 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_a^* and λ_a^*. 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 (Equation 3) that results from accepting the new multipliers is:

$$\begin{aligned}\Delta L_p &= \lambda_a y_a (f_a - \frac{1}{2} y_a \lambda_a k_{aa} - \lambda_0) - \lambda_a^* y_a (f_a^* - \frac{1}{2} y_a \lambda_a^* k_{aa} - \lambda_0^*) + \\ &\quad \lambda_b y_b (f_b - \frac{1}{2} y_b \lambda_b k_{bb} - \lambda_0) - \lambda_b^* y_b (f_b^* - \frac{1}{2} y_b \lambda_b^* k_{bb} - \lambda_0^*) + \\ &\quad k_{ab} (y_a y_b \lambda_a \lambda_b - y_a y_b \lambda_a^* \lambda_b^*) - \lambda_a - \lambda_b + \lambda_a^* + \lambda_b^*. \end{aligned} \tag{21}$$

Equation 21 is derived by substituting λ for α in Equation 3, and rewriting the equation so that all terms are trivially dependent or independent on λ_a and/or λ_b. 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 (Equation 9) can be similarly calculated with:

$$\begin{aligned}\Delta L_p &= \lambda_a (f_a - y_a - \frac{1}{2} \lambda_a k_{aa} - \lambda_0) - \lambda_a^* (f_a^* - y_a - \frac{1}{2} \lambda_a^* k_{aa} - \lambda_0^*) + \\ &\quad \lambda_b (f_b - y_b - \frac{1}{2} \lambda_b k_{bb} - \lambda_0) - \lambda_b^* (f_b^* - y_b - \frac{1}{2} \lambda_b^* k_{bb} - \lambda_0^*) + \\ &\quad k_{ab} (\lambda_a^* \lambda_b^* - \lambda_a \lambda_b) + \epsilon (|\lambda_a| - |\lambda_a^*| + |\lambda_b| - |\lambda_b^*|). \end{aligned} \tag{22}$$

Thus, we modify SMO by replacing line 3.1 in Table 1 with code that looks for the best second multiplier via Equation 21 or 22 for all a such that k_{ab} 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 Equation 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 Equation 2, which is extremely slow.
- Change Equation 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

[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 Equations 3 and 6.

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 λ_0 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 at least 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 [2] with SMO.

The basic idea is to iteratively build an $M \times M$ subproblem with $2 < M < N$, 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 chose 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

As a simple experiment, we model the Mackey-Glass delay-differential equation:

$$\frac{dx}{dt} = \frac{ax(t-\tau)}{1+x^{10}(t-\tau)} - bx(t)$$

with a regression SVM that uses Gaussian kernel functions. We use $\tau = 17$, $a = 0.2$, $b = 0.1$, and Δt = 1. The problem consists of 700 data points, 6 inputs with a delay value of 6, and forecasts are made 60 time steps into the future. Using ϵ equal to 0.1 and Gaussian kernels with 0.5 variance, the solution has 40 support vectors. Hence, this problem is very simple.

Heuristics	Subset	Cache	Time
none	0	0	7.893
none	100	0	5.836
none	0	100	5.754
none	100	100	3.615

Heuristics	Subset	Cache	Time
all	0	0	7.063
all	100	0	5.905
all	0	100	1.266
all	100	100	0.724

Table 2: Experimental results: all results are averaged over ten trials. 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 Subset indicate the subset sized for decomposition (with "0" meaning no decomposition). All times are in CPU seconds on a 450 MHz Pentium II machine running Linux.

Table 2 shows the average runtime for this problem with several different experimental setups. As can be seen, the runtime difference between plain SMO and SMO with all of the proposed changes is an order of magnitude in size.

6 CONCLUSIONS

This work has shown that SMO can be generalized to handle regression problems. The runtime of SMO can be greatly improved for large, dense, and high-dimensional data sets by adding caching along with other heuristics. 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 ways that are infeasible for non-cached kernel outputs.

These changes are particularly beneficial for SMO used on regression problems because such problems typically have many support vectors in the solution. While the numerical result in Section 5 is very simple, the experiment is significant because of the problem's simplicity. More specifically, had the problem had more data points, been of higher dimensionality, or had more support vectors in the solution, then the differences between plain SMO and modified SMO would have been even more pronounced.

Preliminary results also 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. Moreover, recursive decompositions may yield further improvements.

Acknowledgements

We thank Tommy Poggio, John Platt, Edgar Osuna, and Constantine Papageorgious for helpful discussions. Special thanks to Tommy Poggio and the Center for Biological and Computational Learning at MIT for hosting the first author during this research and to Eric Baum and the NEC Research Institute for funding a portion of the time to write up these results.

REFERENCES

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

[2] E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. In *Proc. of IEEE NNSP'97*, 1997.

[3] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.

[4] J. Platt. Private communication, 1999.

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

[6] A. Smola and B. Schölkopf. A tutorial on support vector regression. Technical Report NC2-TR-1998-030, NeuroCOLT2, 1998.

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