Sequential Minimal Optimization:

A Fast Algorithm for Training Support Vector Machines

John C. Platt
Microsoft Research
jplatt@microsoft.com
Technical Report MSR-TR-98-14
April 21, 1998
© 1998 John Platt

ABSTRACT

This paper proposes a new algorithm for training support vector machines: *Sequential Minimal Optimization*, or *SMO*. Training a support vector machine requires the solution of a very large quadratic programming (QP) optimization problem. SMO breaks this large QP problem into a series of smallest possible QP problems. These small QP problems are solved analytically, which avoids using a time-consuming numerical QP optimization as an inner loop. The amount of memory required for SMO is linear in the training set size, which allows SMO to handle very large training sets. Because matrix computation is avoided, SMO scales somewhere between linear and quadratic in the training set size for various test problems, while the standard chunking SVM algorithm scales somewhere between linear and cubic in the training set size. SMO's computation time is dominated by SVM evaluation, hence SMO is fastest for linear SVMs and sparse data sets. On real-world sparse data sets, SMO can be more than 1000 times faster than the chunking algorithm.

1. INTRODUCTION

In the last few years, there has been a surge of interest in Support Vector Machines (SVMs) [19] [20] [4]. SVMs have empirically been shown to give good generalization performance on a wide variety of problems such as handwritten character recognition [12], face detection [15], pedestrian detection [14], and text categorization [9].

However, the use of SVMs is still limited to a small group of researchers. One possible reason is that training algorithms for SVMs are slow, especially for large problems. Another explanation is that SVM training algorithms are complex, subtle, and difficult for an average engineer to implement.

This paper describes a new SVM learning algorithm that is conceptually simple, easy to implement, is generally faster, and has better scaling properties for difficult SVM problems than the standard SVM training algorithm. The new SVM learning algorithm is called *Sequential Minimal Optimization* (or *SMO*). Instead of previous SVM learning algorithms that use numerical quadratic programming (QP) as an inner loop, SMO uses an analytic QP step.

This paper first provides an overview of SVMs and a review of current SVM training algorithms. The SMO algorithm is then presented in detail, including the solution to the analytic QP step,

heuristics for choosing which variables to optimize in the inner loop, a description of how to set the threshold of the SVM, some optimizations for special cases, the pseudo-code of the algorithm, and the relationship of SMO to other algorithms.

SMO has been tested on two real-world data sets and two artificial data sets. This paper presents the results for timing SMO versus the standard "chunking" algorithm for these data sets and presents conclusions based on these timings. Finally, there is an appendix that describes the derivation of the analytic optimization.

1.1 Overview of Support Vector Machines

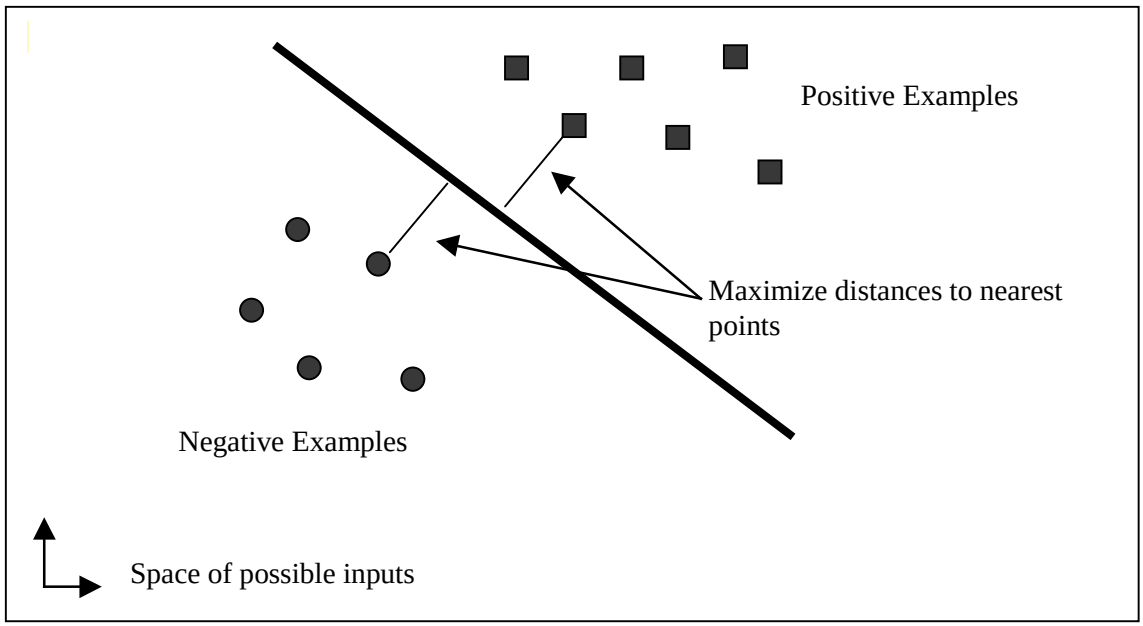

Figure 1 A linear Support Vector Machine

Vladimir Vapnik invented Support Vector Machines in 1979 [19]. In its simplest, linear form, an SVM is a hyperplane that separates a set of positive examples from a set of negative examples with maximum margin (see figure 1). In the linear case, the margin is defined by the distance of the hyperplane to the nearest of the positive and negative examples. The formula for the output of a linear SVM is

$$u = \vec{w} \cdot \vec{x} - b, \tag{1}$$

where w is the normal vector to the hyperplane and x is the input vector. The separating hyperplane is the plane $u=0$. The nearest points lie on the planes $u = \pm 1$. The margin m is thus

$$m = \frac{1}{||w||_2}. \tag{2}$$

Maximizing margin can be expressed via the following optimization problem [4]:

$$\min_{\vec{w},b} \tfrac{1}{2}||\vec{w}||^2 \text{ subject to } y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1, \forall i, \tag{3}$$

where x_i is the ith training example, and y_i is the correct output of the SVM for the ith training example. The value y_i is +1 for the positive examples in a class and –1 for the negative examples.

Using a Lagrangian, this optimization problem can be converted into a dual form which is a QP problem where the objective function Ψ is solely dependent on a set of Lagrange multipliers α_i,

$$\min_{\vec{\alpha}} \Psi(\vec{\alpha}) = \min_{\vec{\alpha}} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j (\vec{x}_i \cdot \vec{x}_j) \alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i, \tag{4}$$

(where N is the number of training examples), subject to the inequality constraints,

$$\alpha_i \geq 0, \forall i, \tag{5}$$

and one linear equality constraint,

$$\sum_{i=1}^{N} y_i \alpha_i = 0. \tag{6}$$

There is a one-to-one relationship between each Lagrange multiplier and each training example. Once the Lagrange multipliers are determined, the normal vector $\vec{w}$ and the threshold b can be derived from the Lagrange multipliers:

$$\vec{w} = \sum_{i=1}^{N} y_i \alpha_i \vec{x}_i, \quad b = \vec{w} \cdot \vec{x}_k - y_k \text{ for some } \alpha_k > 0. \tag{7}$$

Because $\vec{w}$ can be computed via equation (7) from the training data before use, the amount of computation required to evaluate a linear SVM is constant in the number of non-zero support vectors.

Of course, not all data sets are linearly separable. There may be no hyperplane that splits the positive examples from the negative examples. In the formulation above, the non-separable case would correspond to an infinite solution. However, in 1995, Cortes & Vapnik [7] suggested a modification to the original optimization statement (3) which allows, but penalizes, the failure of an example to reach the correct margin. That modification is:

$$\min_{\vec{w},b,\vec{\xi}} \frac{1}{2} ||\vec{w}||^2 + C \sum_{i=1}^{N} \xi_i \text{ subject to } y_i (\vec{w} \cdot \vec{x}_i - b) \geq 1 - \xi_i, \forall i, \tag{8}$$

where ξ_i are slack variables that permit margin failure and C is a parameter which trades off wide margin with a small number of margin failures. When this new optimization problem is transformed into the dual form, it simply changes the constraint (5) into a box constraint:

$$0 \leq \alpha_i \leq C, \forall i. \tag{9}$$

The variables ξ_i do not appear in the dual formulation at all.

SVMs can be even further generalized to non-linear classifiers [2]. The output of a non-linear SVM is explicitly computed from the Lagrange multipliers:

$$u = \sum_{j=1}^{N} y_j \alpha_j K(\vec{x}_j, \vec{x}) - b, \tag{10}$$

where K is a kernel function that measures the similarity or distance between the input vector $\vec{x}$ and the stored training vector $\vec{x}_j$. Examples of K include Gaussians, polynomials, and neural network non-linearities [4]. If K is linear, then the equation for the linear SVM (1) is recovered.

The Lagrange multipliers α_i are still computed via a quadratic program. The non-linearities alter the quadratic form, but the dual objective function Ψ is still quadratic in α:

$$\begin{aligned} \min_{\vec{\alpha}} \Psi(\vec{\alpha}) = \min_{\vec{\alpha}} \tfrac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j K(\vec{x}_i, \vec{x}_j) \alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i, \\ 0 \le \alpha_i \le C, \forall i, \\ \sum_{i=1}^{N} y_i \alpha_i = 0. \end{aligned} \tag{11}$$

The QP problem in equation (11), above, is the QP problem that the SMO algorithm will solve. In order to make the QP problem above be positive definite, the kernel function K must obey Mercer's conditions [4].

The Karush-Kuhn-Tucker (KKT) conditions are necessary and sufficient conditions for an optimal point of a positive definite QP problem. The KKT conditions for the QP problem (11) are particularly simple. The QP problem is solved when, for all i:

$$\begin{aligned} \alpha_i = 0 \Leftrightarrow y_i u_i \ge 1, \\ 0 < \alpha_i < C \Leftrightarrow y_i u_i = 1, \\ \alpha_i = C \Leftrightarrow y_i u_i \le 1. \end{aligned} \tag{12}$$

where u_i is the output of the SVM for the ith training example. Notice that the KKT conditions can be evaluated on one example at a time, which will be useful in the construction of the SMO algorithm.

1.2 Previous Methods for Training Support Vector Machines

Due to its immense size, the QP problem (11) that arises from SVMs cannot be easily solved via standard QP techniques. The quadratic form in (11) involves a matrix that has a number of elements equal to the square of the number of training examples. This matrix cannot be fit into 128 Megabytes if there are more than 4000 training examples.

Vapnik [19] describes a method to solve the SVM QP, which has since been known as "chunking." The chunking algorithm uses the fact that the value of the quadratic form is the same if you remove the rows and columns of the matrix that corresponds to zero Lagrange multipliers. Therefore, the large QP problem can be broken down into a series of smaller QP problems, whose ultimate goal is to identify all of the non-zero Lagrange multipliers and discard all of the zero Lagrange multipliers. At every step, chunking solves a QP problem that consists of the following examples: every non-zero Lagrange multiplier from the last step, and the M worst examples that violate the KKT conditions (12) [4], for some value of M (see figure 2). If there are fewer than M examples that violate the KKT conditions at a step, all of the violating examples are added in. Each QP sub-problem is initialized with the results of the previous sub-problem. At the last step,

the entire set of non-zero Lagrange multipliers has been identified, hence the last step solves the large QP problem.

Chunking seriously reduces the size of the matrix from the number of training examples squared to approximately the number of non-zero Lagrange multipliers squared. However, chunking still cannot handle large-scale training problems, since even this reduced matrix cannot fit into memory.

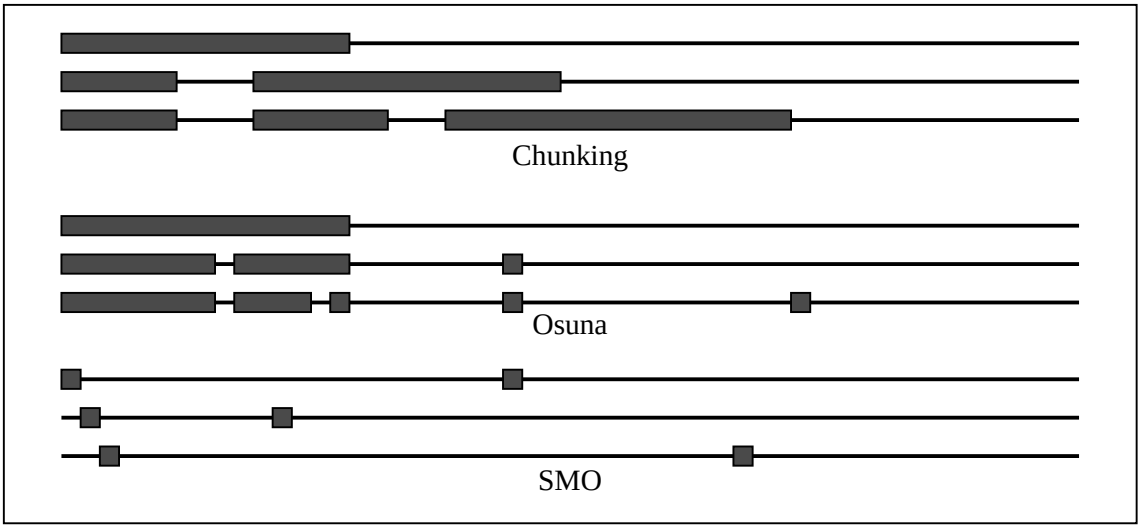

Figure 2. Three alternative methods for training SVMs: Chunking, Osuna's algorithm, and SMO. For each method, three steps are illustrated. The horizontal thin line at every step represents the training set, while the thick boxes represent the Lagrange multipliers being optimized at that step. For chunking, a fixed number of examples are added every step, while the zero Lagrange multipliers are discarded at every step. Thus, the number of examples trained per step tends to grow. For Osuna's algorithm, a fixed number of examples are optimized every step: the same number of examples is added to and discarded from the problem at every step. For SMO, only two examples are analytically optimized at every step, so that each step is very fast.

In 1997, Osuna, et al. [16] proved a theorem which suggests a whole new set of QP algorithms for SVMs. The theorem proves that the large QP problem can be broken down into a series of smaller QP sub-problems. As long as at least one example that violates the KKT conditions is added to the examples for the previous sub-problem, each step will reduce the overall objective function and maintain a feasible point that obeys all of the constraints. Therefore, a sequence of QP sub-problems that always add at least one violator will be guaranteed to converge. Notice that the chunking algorithm obeys the conditions of the theorem, and hence will converge.

Osuna, et al. suggests keeping a constant size matrix for every QP sub-problem, which implies adding and deleting the same number of examples at every step [16] (see figure 2). Using a constant-size matrix will allow the training on arbitrarily sized data sets. The algorithm given in Osuna's paper [16] suggests adding one example and subtracting one example every step. Clearly this would be inefficient, because it would use an entire numerical QP optimization step to cause one training example to obey the KKT conditions. In practice, researchers add and subtract multiple examples according to unpublished heuristics [17]. In any event, a numerical QP solver is required for all of these methods. Numerical QP is notoriously tricky to get right; there are many numerical precision issues that need to be addressed.

2. SEQUENTIAL MINIMAL OPTIMIZATION

Sequential Minimal Optimization (SMO) is a simple algorithm that can quickly solve the SVM QP problem without any extra matrix storage and without using numerical QP optimization steps at all. SMO decomposes the overall QP problem into QP sub-problems, using Osuna's theorem to ensure convergence.

Unlike the previous methods, SMO chooses to solve the smallest possible optimization problem at every step. For the standard SVM QP problem, the smallest possible optimization problem involves two Lagrange multipliers, because the Lagrange multipliers must obey a linear equality constraint. At every step, SMO chooses two Lagrange multipliers to jointly optimize, finds the optimal values for these multipliers, and updates the SVM to reflect the new optimal values (see figure 2).

The advantage of SMO lies in the fact that solving for two Lagrange multipliers can be done analytically. Thus, numerical QP optimization is avoided entirely. The inner loop of the algorithm can be expressed in a short amount of C code, rather than invoking an entire QP library routine. Even though more optimization sub-problems are solved in the course of the algorithm, each sub-problem is so fast that the overall QP problem is solved quickly.

In addition, SMO requires no extra matrix storage at all. Thus, very large SVM training problems can fit inside of the memory of an ordinary personal computer or workstation. Because no matrix algorithms are used in SMO, it is less susceptible to numerical precision problems.

There are two components to SMO: an analytic method for solving for the two Lagrange multipliers, and a heuristic for choosing which multipliers to optimize.

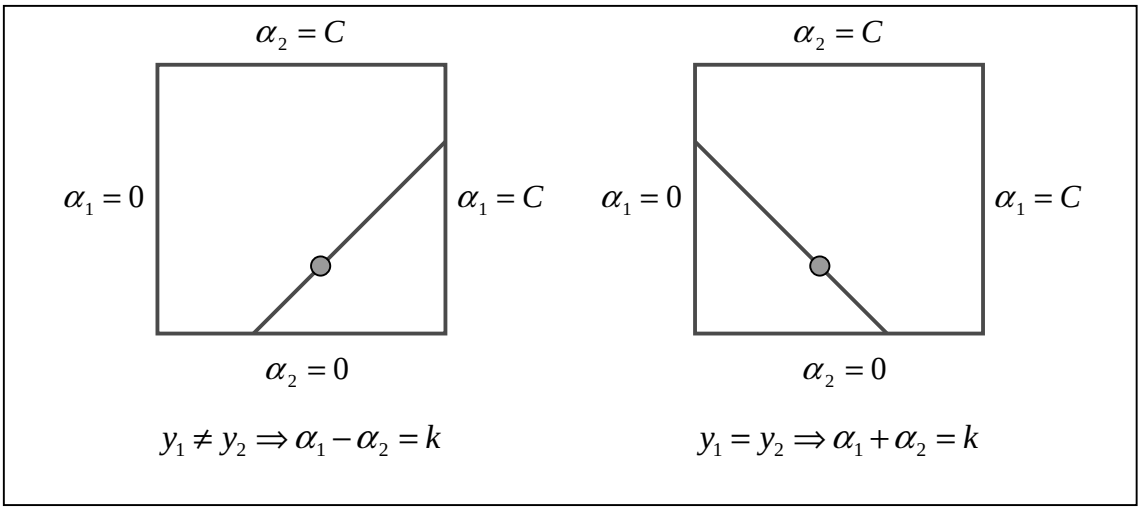

Figure 1. The two Lagrange multipliers must fulfill all of the constraints of the full problem. The inequality constraints cause the Lagrange multipliers to lie in the box. The linear equality constraint causes them to lie on a diagonal line. Therefore, one step of SMO must find an optimum of the objective function on a diagonal line segment.

2.1 Solving for Two Lagrange Multipliers

In order to solve for the two Lagrange multipliers, SMO first computes the constraints on these multipliers and then solves for the constrained minimum. For convenience, all quantities that refer to the first multiplier will have a subscript 1, while all quantities that refer to the second multiplier will have a subscript 2. Because there are only two multipliers, the constraints can be easily be displayed in two dimensions (see figure 3). The bound constraints (9) cause the Lagrange multipliers to lie within a box, while the linear equality constraint (6) causes the Lagrange multipliers to lie on a diagonal line. Thus, the constrained minimum of the objective function must lie on a diagonal line segment (as shown in figure 3). This constraint explains why two is the minimum number of Lagrange multipliers that can be optimized: if SMO optimized only one multiplier, it could not fulfill the linear equality constraint at every step.

The ends of the diagonal line segment can be expressed quite simply. Without loss of generality, the algorithm first computes the second Lagrange multiplier α_2 and computes the ends of the diagonal line segment in terms of α_2. If the target y_1 does not equal the target y_2, then the following bounds apply to α_2:

$$L = \max(0, \alpha_2 - \alpha_1), \qquad H = \min(C, C + \alpha_2 - \alpha_1). \tag{13}$$

If the target y_1 equals the target y_2, then the following bounds apply to α_2:

$$L = \max(0, \alpha_2 + \alpha_1 - C), \qquad H = \min(C, \alpha_2 + \alpha_1). \tag{14}$$

The second derivative of the objective function along the diagonal line can be expressed as:

$$\eta = K(\vec{x}_1, \vec{x}_1) + K(\vec{x}_2, \vec{x}_2) - 2K(\vec{x}_1, \vec{x}_2). \tag{15}$$

Under normal circumstances, the objective function will be positive definite, there will be a minimum along the direction of the linear equality constraint, and η will be greater than zero. In this case, SMO computes the minimum along the direction of the constraint :

$$\alpha_2^{\text{new}} = \alpha_2 + \frac{y_2(E_1 - E_2)}{\eta}, \tag{16}$$

where $E_i = u_i - y_i$ is the error on the ith training example. As a next step, the constrained minimum is found by clipping the unconstrained minimum to the ends of the line segment:

$$\alpha_2^{\text{new,clipped}} = \begin{cases} H & \text{if} \quad \alpha_2^{\text{new}} \geq H; \\ \alpha_2^{\text{new}} & \text{if} \quad L < \alpha_2^{\text{new}} < H; \\ \text{L} & \text{if} \quad \alpha_2^{\text{new}} \leq L. \end{cases} \tag{17}$$

Now, let $s = y_1 y_2$. The value of α_1 is computed from the new, clipped, α_2:

$$\alpha_1^{\text{new}} = \alpha_1 + s(\alpha_2 - \alpha_2^{\text{new,clipped}}). \tag{18}$$

Under unusual circumstances, η will not be positive. A negative η will occur if the kernel K does not obey Mercer's condition, which can cause the objective function to become indefinite. A zero η can occur even with a correct kernel, if more than one training example has the same input

vector *x*. In any event, SMO will work even when η is not positive, in which case the objective function Ψ should be evaluated at each end of the line segment:

$$\begin{aligned}
f_1 &= y_1(E_1 + b) - \alpha_1 K(\vec{x}_1, \vec{x}_1) - s\alpha_2 K(\vec{x}_1, \vec{x}_2), \\
f_2 &= y_2(E_2 + b) - s\alpha_1 K(\vec{x}_1, \vec{x}_2) - \alpha_2 K(\vec{x}_2, \vec{x}_2), \\
L_1 &= \alpha_1 + s(\alpha_2 - L), \\
H_1 &= \alpha_1 + s(\alpha_2 - H), \\
\Psi_L &= L_1 f_1 + L f_2 + \tfrac{1}{2} L_1^2 K(\vec{x}_1, \vec{x}_1) + \tfrac{1}{2} L^2 K(\vec{x}_2, \vec{x}_2) + sLL_1 K(\vec{x}_1, \vec{x}_2), \\
\Psi_H &= H_1 f_1 + H f_2 + \tfrac{1}{2} H_1^2 K(\vec{x}_1, \vec{x}_1) + \tfrac{1}{2} H^2 K(\vec{x}_2, \vec{x}_2) + sHH_1 K(\vec{x}_1, \vec{x}_2).
\end{aligned} \tag{19}$$

SMO will move the Lagrange multipliers to the end point that has the lowest value of the objective function. If the objective function is the same at both ends (within a small ε for round-off error) and the kernel obeys Mercer's conditions, then the joint minimization cannot make progress. That scenario is described below.

2.2 Heuristics for Choosing Which Multipliers To Optimize

As long as SMO always optimizes and alters two Lagrange multipliers at every step and at least one of the Lagrange multipliers violated the KKT conditions before the step, then each step will decrease the objective function according to Osuna's theorem [16]. Convergence is thus guaranteed. In order to speed convergence, SMO uses heuristics to choose which two Lagrange multipliers to jointly optimize.

There are two separate choice heuristics: one for the first Lagrange multiplier and one for the second. The choice of the first heuristic provides the outer loop of the SMO algorithm. The outer loop first iterates over the entire training set, determining whether each example violates the KKT conditions (12). If an example violates the KKT conditions, it is then eligible for optimization. After one pass through the entire training set, the outer loop iterates over all examples whose Lagrange multipliers are neither 0 nor C (the non-bound examples). Again, each example is checked against the KKT conditions and violating examples are eligible for optimization. The outer loop makes repeated passes over the non-bound examples until all of the non-bound examples obey the KKT conditions within ε. The outer loop then goes back and iterates over the entire training set. The outer loop keeps alternating between single passes over the entire training set and multiple passes over the non-bound subset until the entire training set obeys the KKT conditions within ε, whereupon the algorithm terminates.

The first choice heuristic concentrates the CPU time on the examples that are most likely to violate the KKT conditions: the non-bound subset. As the SMO algorithm progresses, examples that are at the bounds are likely to stay at the bounds, while examples that are not at the bounds will move as other examples are optimized. The SMO algorithm will thus iterate over the non-bound subset until that subset is self-consistent, then SMO will scan the entire data set to search for any bound examples that have become KKT violated due to optimizing the non-bound subset.

Notice that the KKT conditions are checked to be within ε of fulfillment. Typically, ε is set to be 10^{-3}. Recognition systems typically do not need to have the KKT conditions fulfilled to high accuracy: it is acceptable for examples on the positive margin to have outputs between 0.999 and 1.001. The SMO algorithm (and other SVM algorithms) will not converge as quickly if it is required to produce very high accuracy output.

Once a first Lagrange multiplier is chosen, SMO chooses the second Lagrange multiplier to maximize the size of the step taken during joint optimization. Now, evaluating the kernel function K is time consuming, so SMO approximates the step size by the absolute value of the numerator in equation (16): $|E_1 - E_2|$. SMO keeps a cached error value E for every non-bound example in the training set and then chooses an error to approximately maximize the step size. If E_1 is positive, SMO chooses an example with minimum error E_2. If E_1 is negative, SMO chooses an example with maximum error E_2.

Under unusual circumstances, SMO cannot make positive progress using the second choice heuristic described above. For example, positive progress cannot be made if the first and second training examples share identical input vectors x, which causes the objective function to become semi-definite. In this case, SMO uses a hierarchy of second choice heuristics until it finds a pair of Lagrange multipliers that can be make positive progress. Positive progress can be determined by making a non-zero step size upon joint optimization of the two Lagrange multipliers . The hierarchy of second choice heuristics consists of the following. If the above heuristic does not make positive progress, then SMO starts iterating through the non-bound examples, searching for an second example that can make positive progress. If none of the non-bound examples make positive progress, then SMO starts iterating through the entire training set until an example is found that makes positive progress. Both the iteration through the non-bound examples and the iteration through the entire training set are started at random locations, in order not to bias SMO towards the examples at the beginning of the training set. In extremely degenerate circumstances, none of the examples will make an adequate second example. When this happens, the first example is skipped and SMO continues with another chosen first example.

2.3 Computing the Threshold

The threshold b is re-computed after each step, so that the KKT conditions are fulfilled for both optimized examples. The following threshold b_1 is valid when the new α_1 is not at the bounds, because it forces the output of the SVM to be y_1 when the input is x_1:

$$b_1 = E_1 + y_1(\alpha_1^{\text{new}} - \alpha_1)K(\vec{x}_1, \vec{x}_1) + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2)K(\vec{x}_1, \vec{x}_2) + b. \tag{20}$$

The following threshold b_2 is valid when the new α_2 is not at bounds, because it forces the output of the SVM to be y_2 when the input is x_2:

$$b_2 = E_2 + y_1(\alpha_1^{\text{new}} - \alpha_1)K(\vec{x}_1, \vec{x}_2) + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2)K(\vec{x}_2, \vec{x}_2) + b. \tag{21}$$

When both b_1 and b_2 are valid, they are equal. When both new Lagrange multipliers are at bound and if L is not equal to H, then the interval between b_1 and b_2 are all thresholds that are consistent with the KKT conditions. SMO chooses the threshold to be halfway in between b_1 and b_2.

2.4 An Optimization for Linear SVMs

To compute a linear SVM, only a single weight vector $\vec{w}$ needs to be stored, rather than all of the training examples that correspond to non-zero Lagrange multipliers. If the joint optimization succeeds, the stored weight vector needs to be updated to reflect the new Lagrange multiplier values. The weight vector update is easy, due to the linearity of the SVM:

$$\vec{w}^{\text{new}} = \vec{w} + y_1(\alpha_1^{\text{new}} - \alpha_1)\vec{x}_1 + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2)\vec{x}_2. \tag{22}$$

2.5 Code Details

The pseudo-code below describes the entire SMO algorithm:

```
 target = desired output vector
 point = training point matrix

 procedure takeStep(i1,i2)
   if (i1 == i2) return 0
   alph1 = Lagrange multiplier for i1
   y1 = target[i1]
   E1 = SVM output on point[i1] – y1 (check in error cache)
   s = y1*y2
   Compute L, H via equations (13) and (14)
   if (L == H)
     return 0
   k11 = kernel(point[i1],point[i1])
   k12 = kernel(point[i1],point[i2])
   k22 = kernel(point[i2],point[i2])
   eta = k11+k22-2*k12
   if (eta > 0)
   {
      a2 = alph2 + y2*(E1-E2)/eta
      if (a2 < L) a2 = L
      else if (a2 > H) a2 = H
   }
   else
   {
      Lobj = objective function at a2=L
      Hobj = objective function at a2=H
      if (Lobj < Hobj-eps)
         a2 = L
      else if (Lobj > Hobj+eps)
         a2 = H
      else
         a2 = alph2
   }
   if (|a2-alph2| < eps*(a2+alph2+eps))
      return 0
   a1 = alph1+s*(alph2-a2)
   Update threshold to reflect change in Lagrange multipliers
   Update weight vector to reflect change in a1 & a2, if SVM is linear
   Update error cache using new Lagrange multipliers
   Store a1 in the alpha array
   Store a2 in the alpha array
   return 1
endprocedure

procedure examineExample(i2)
   y2 = target[i2]
   alph2 = Lagrange multiplier for i2
   E2 = SVM output on point[i2] – y2 (check in error cache)
   r2 = E2*y2
   if ((r2 < -tol && alph2 < C) || (r2 > tol && alph2 > 0))
   {
     if (number of non-zero & non-C alpha > 1)
     {
       i1 = result of second choice heuristic (section 2.2)
       if takeStep(i1,i2)
         return 1
     }
```

```
     loop over all non-zero and non-C alpha, starting at a random point
     {
        i1 = identity of current alpha
        if takeStep(i1,i2)
          return 1
     }
     loop over all possible i1, starting at a random point
     {
        i1 = loop variable
        if (takeStep(i1,i2)
          return 1
     }
   }
   return 0
endprocedure

main routine:
    numChanged = 0;
    examineAll = 1;
    while (numChanged > 0 | examineAll)
    {
       numChanged = 0;
       if (examineAll)
          loop I over all training examples
             numChanged += examineExample(I)
       else
          loop I over examples where alpha is not 0 & not C
             numChanged += examineExample(I)
       if (examineAll == 1)
          examineAll = 0
       else if (numChanged == 0)
          examineAll = 1
   }
```

2.6 Relationship to Previous Algorithms

The SMO algorithm is related both to previous SVM and optimization algorithms. The SMO algorithm can be considered a special case of the Osuna algorithm, where the size of the optimization is two and both Lagrange multipliers are replaced at every step with new multipliers that are chosen via good heuristics.

The SMO algorithm is closely related to a family of optimization algorithms called Bregman methods [3] or row-action methods [5]. These methods solve convex programming problems with linear constraints. They are iterative methods where each step projects the current primal point onto each constraint. An unmodified Bregman method cannot solve the QP problem (11) directly, because the threshold in the SVM creates a linear equality constraint in the dual problem. If only one constraint were projected per step, the linear equality constraint would be violated. In more technical terms, the primal problem of minimizing the norm of the weight vector $\vec{w}$ over the combined space of all possible weight vectors $\vec{w}$ with thresholds b produces a Bregman D-projection that does not have a unique minimum [3][6].

It is interesting to consider an SVM where the threshold b is held fixed at zero, rather than being solved for. A fixed-threshold SVM would not have a linear equality constraint (6). Therefore, only one Lagrange multiplier would need to be updated at a time and a row-action method can be used. Unfortunately, a traditional Bregman method is still not applicable to such SVMs, due to

the slack variables ξ_i in equation (8). The presence of the slack variables causes the Bregman *D*-projection to become non-unique in the combined space of weight vectors $\vec{w}$ and slack variables ξ_i

Fortunately, SMO can be modified to solve fixed-threshold SVMs. SMO will update individual Lagrange multipliers to be the minimum of Ψ along the corresponding dimension. The update rule is

$$\alpha_1^{\text{new}} = \alpha_1 + \frac{y_1 E_1}{K(\vec{x}_1, \vec{x}_1)}. \tag{23}$$

This update equation forces the output of the SVM to be y_1 (similar to Bregman methods or Hildreth's QP method [10]). After the new α is computed, it is clipped to the [0,*C*] interval (unlike previous methods). The choice of which Lagrange multiplier to optimize is the same as the first choice heuristic described in section 2.2.

Fixed-threshold SMO for a linear SVM is similar in concept to the perceptron relaxation rule [8], where the output of a perceptron is adjusted whenever there is an error, so that the output exactly lies on the margin. However, the fixed-threshold SMO algorithm will sometimes reduce the proportion of a training input in the weight vector in order to maximize margin. The relaxation rule constantly increases the amount of a training input in the weight vector and, hence, is not maximum margin. Fixed-threshold SMO for Gaussian kernels is also related to the resource allocating network (RAN) algorithm [18]. When RAN detects certain kinds of errors, it will allocate a kernel to exactly fix the error. SMO will perform similarly. However SMO/SVM will adjust the heights of the kernels to maximize the margin in a feature space, while RAN will simply use LMS to adjust the heights and weights of the kernels.

3 BENCHMARKING SMO

The SMO algorithm was tested against the standard chunking SVM learning algorithm on a series of benchmarks. Both algorithms were written in C++, using Microsoft's Visual C++ 5.0 compiler. Both algorithms were run on an unloaded 266 MHz Pentium II processor running Windows NT 4.

Both algorithms were written to exploit the sparseness of the input vector. More specifically, the kernel functions rely on dot products in the inner loop. If the input is a sparse vector, then an input can be stored as a sparse array, and the dot product will merely iterate over the non-zero inputs, accumulating the non-zero inputs multiplied by the corresponding weights. If the input is a sparse binary vector, then the position of the "1"s in the input can be stored, and the dot product will sum the weights corresponding to the position of the "1"s in the input.

The chunking algorithm uses the projected conjugate gradient algorithm [11] as its QP solver, as suggested by Burges [4]. In order to ensure that the chunking algorithm is a fair benchmark, Burges compared the speed of his chunking code on a 200 MHz Pentium II running Solaris with the speed of the benchmark chunking code (with the sparse dot product code turned off). The speeds were found to be comparable, which indicates that the benchmark chunking code is reasonable benchmark.

Ensuring that the chunking code and the SMO code attain the same accuracy takes some care. The SMO code and the chunking code will both identify an example as violating the KKT condition if the output is more than 10^{-3} away from its correct value or half-space. The threshold of 10^{-3} was chosen to be an insignificant error in classification tasks. The projected conjugate gradient code has a stopping threshold, which describes the minimum relative improvement in the objective function at every step [4]. If the projected conjugate gradient takes a step where the relative improvement is smaller than this minimum, the conjugate gradient code terminates and another chunking step is taken. Burges [4] recommends using a constant 10^{-10} for this minimum.

In the experiments below, stopping the projected conjugate gradient at an accuracy of 10^{-10} often left KKT violations larger than 10^{-3}, especially for the very large scale problems. Hence, the benchmark chunking algorithm used the following heuristic to set the conjugate gradient stopping threshold. The threshold starts at $3x10^{-10}$. After every chunking step, the output is computed for all examples whose Lagrange multipliers are not at bound. These outputs are computed in order to compute the value for the threshold (see [4]). Every example suggests a proposed threshold. If the largest proposed threshold is more than $2x10^{-3}$ above the smallest proposed threshold, then the KKT conditions cannot possibly be fulfilled within 10^{-3}. Therefore, starting at the next chunk, the conjugate gradient threshold is decreased by a factor of 3. This heuristic will optimize the speed of the conjugate gradient: it will only use high precision on the most difficult problems. For most of the tests described below, the threshold stayed at $3x10^{-10}$. The smallest threshold used was $3.7x10^{-12}$, which occurred at the end of the chunking for the largest web page classification problem.

The SMO algorithm was tested on an income prediction task, a web page classification task, and two different artificial data sets. All times listed in all of the tables are in CPU seconds.

3.1 Income Prediction

The first data set used to test SMO's speed was the UCI "adult" data set, which is available at ftp://ftp.ics.uci.edu/pub/machine-learning-databases/adult. The SVM was given 14 attributes of a census form of a household. The task of the SVM was to predict whether that household has an income greater than $50,000. Out of the 14 attributes, eight are categorical and six are continuous. For ease of experimentation, the six continuous attributes were discretized into quintiles, which yielded a total of 123 binary attributes, of which 14 are true. There were 32562 examples in the "adult" training set. Two different SVMs were trained on the problem: a linear SVM, and a radial basis function SVM that used Gaussian kernels with variance of 10. This variance was chosen to minimize the error rate on a validation set. The limiting value of C was chosen to be 0.05 for the linear SVM and 1 for the RBF/SVM. Again, this limiting value was chosen to minimize error on a validation set.

The timing performance of the SMO algorithm versus the chunking algorithm for the linear SVM on the adult data set is shown in the table below:

Training Set Size	SMO time	Chunking time	Number of Non-Bound Support Vectors	Number of Bound Support Vectors
1605	0.4	37.1	42	633
2265	0.9	228.3	47	930
3185	1.8	596.2	57	1210
4781	3.6	1954.2	63	1791
6414	5.5	3684.6	61	2370
11221	17.0	20711.3	79	4079
16101	35.3	N/A	67	5854
22697	85.7	N/A	88	8209
32562	163.6	N/A	149	11558

The training set size was varied by taking random subsets of the full training set. These subsets are nested. The "N/A" entries in the chunking time column had matrices that were too large to fit into 128 Megabytes, hence could not be timed due to memory thrashing. The number of non-bound and the number of bound support vectors were determined from SMO: the chunking results vary by a small amount, due to the tolerance of inaccuracies around the KKT conditions.

By fitting a line to the log-log plot of training time versus training set size, an empirical scaling for SMO and chunking can be derived. The SMO training time scales as $\sim N^{1.9}$, while chunking scales as $\sim N^{3.1}$. Thus, SMO improves empirical scaling for this problem by more than one order.

The timing performance of SMO and chunking using a Gaussian SVM is shown below:

Training Set Size	SMO time	Chunking time	Number of Non-Bound Support Vectors	Number of Bound Support Vectors
1605	15.8	34.8	106	585
2265	32.1	144.7	165	845
3185	66.2	380.5	181	1115
4781	146.6	1137.2	238	1650
6414	258.8	2530.6	298	2181
11221	781.4	11910.6	460	3746
16101	1784.4	N/A	567	5371
22697	4126.4	N/A	813	7526
32562	7749.6	N/A	1011	10663

The SMO algorithm is slower for non-linear SVMs than linear SVMs, because the time is dominated by the evaluation of the SVM. Here, the SMO training time scales as $\sim N^{2.1}$, while chunking scales as $\sim N^{2.9}$. Again, SMO's scaling is roughly one order faster than chunking. The income prediction test indicates that for real-world sparse problems with many support vectors at bound, that SMO is much faster than chunking.

3.2 Classifying Web Pages

The second test of SMO was on text categorization: classifying whether a web page belongs to a category or not. The training set consisted of 49749 web pages, with 300 sparse binary keyword attributes extracted from each web page. Two different SVMs were tried on this problem: a linear SVM and a non-linear Gaussian SVM, which used a variance of 10. The C value for the linear SVM was chosen to be 1, while the C value for the non-linear SVM was chosen to be 5. Again, these parameters were chosen to maximize performance on a validation set.

The timings for SMO versus chunking for a linear SVM are shown in the table, below:

Training Set Size	SMO time	Chunking time	Number of Non-Bound Support Vectors	Number of Bound Support Vectors
2477	2.2	13.1	123	47
3470	4.9	16.1	147	72
4912	8.1	40.6	169	107
7366	12.7	140.7	194	166
9888	24.7	239.3	214	245
17188	65.4	1633.3	252	480
24692	104.9	3369.7	273	698
49749	268.3	17164.7	315	1408

For the linear SVM on this data set, the SMO training time scales as $\sim N^{1.6}$, while chunking scales as $\sim N^{2.5}$. This experiment is another situation where SMO is superior to chunking in computation time.

The timings for SMO versus chunking for a non-linear SVM are shown in the table, below:

Training Set Size	SMO time	Chunking time	Number of Non-Bound Support Vectors	Number of Bound Support Vectors
2477	26.3	64.9	439	43
3470	44.1	110.4	544	66
4912	83.6	372.5	616	90
7366	156.7	545.4	914	125
9888	248.1	907.6	1118	172
17188	581.0	3317.9	1780	316
24692	1214.0	6659.7	2300	419
49749	3863.5	23877.6	3720	764

For the non-linear SVM on this data set, the SMO training time scales as $\sim N^{1.7}$, while chunking scales as $\sim N^{2.0}$. In this case, the scaling for SMO is somewhat better than chunking: SMO is a factor of between two and six times faster than chunking. The non-linear test shows that SMO is still faster than chunking when the number of non-bound support vectors is large and the input data set is sparse.

3.3 Artificial Data Sets

SMO was also tested on artificially generated data sets to explore the performance of SMO in extreme scenarios. The first artificial data set was a perfectly linearly separable data set. The input data was random binary 300-dimensional vectors, with a 10% fraction of "1" inputs. A 300-dimensional weight vector was generated randomly in [-1,1]. If the dot product of the weight with an input point is greater than 1, then a positive label is assigned to the input point. If the dot product is less than –1, then a negative label is assigned. If the dot product lies between –1 and 1, the point is discarded. A linear SVM was fit to this data set.

The linearly separable data set is the simplest possible problem for a linear SVM. Not surprisingly, the scaling with training set size is excellent for both SMO and chunking. The running times are shown in the table below:

Training Set Size	SMO time	Chunking time	Number of Non-Bound Support Vectors	Number of Bound Support Vectors
1000	15.3	10.4	275	0
2000	33.4	33.0	286	0
5000	103.0	108.3	299	0
10000	186.8	226.0	309	0
20000	280.0	374.1	329	0

Here, the SMO running time scales as $\sim N$, which is slightly better than the scaling for chunking, which is $\sim N^{1.2}$. For this easy sparse problem, therefore, chunking and SMO are generally comparable. Both algorithms were trained with C set to 100. The chunk size for chunking is set to be 500.

The acceleration of both the SMO algorithm and the chunking algorithm due to the sparse dot product code can be measured on this easy data set. The same data set was tested with and without the sparse dot product code. In the case of the non-sparse experiment, each input point was stored as a 300-dimensional vector of floats. The result of the sparse/non-sparse experiment is shown in the table below:

Training Set Size	SMO time (sparse)	SMO time (non-sparse)	Chunking time (sparse)	Chunking time (non-sparse)
1000	15.3	145.1	10.4	11.7
2000	33.4	345.4	33.0	36.8
5000	103.0	1118.1	108.3	117.9
10000	186.8	2163.7	226.0	241.6
20000	280.0	3293.9	374.1	397.0

For SMO, use of the sparse data structure speeds up the code by more than a factor of 10, which shows that the evaluation time of the SVM totally dominates the SMO computation time. The sparse dot product code only speeds up chunking by a factor of approximately 1.1, which shows that the evaluation of the numerical QP steps dominates the chunking computation. For the linearly separable case, there are absolutely no Lagrange multipliers at bound, which is the worst case for SMO. Thus, the poor performance of non-sparse SMO versus non-sparse chunking in this experiment should be considered a worst case.

The sparse versus non-sparse experiment shows that part of the superiority of SMO over chunking comes from the exploitation of sparse dot product code. Fortunately, many real-world problems have sparse input. In addition to the real-word data sets described in section 3.1 and section 3.2, any quantized or fuzzy-membership-encoded problems will be sparse. Also, optical character recognition [12], handwritten character recognition [1], and wavelet transform coefficients of natural images [13] [14] tend to be naturally expressed as sparse data.

The second artificial data set stands in stark contrast to the first easy data set. The second set is generated with random 300-dimensional binary input points (10% "1") and random output labels. The SVMs are thus fitting pure noise. The *C* value was set to 0.1, since the problem is fundamentally unsolvable. The results for SMO and chunking applied to a linear SVM are shown below:

Training Set Size	SMO time	Chunking time	Number of Non-Bound Support Vectors	Number of Bound Support Vectors
500	1.0	6.4	162	263
1000	3.5	57.9	220	632
2000	15.7	593.8	264	1476
5000	67.6	10353.3	283	4201
10000	187.1	N/A	293	9034

Scaling for SMO and chunking are much higher on the second data set. This reflects the difficulty of the problem. The SMO computation time scales as $\sim N^{1.8}$, while the chunking computation time scales as $\sim N^{3.2}$. The second data set shows that SMO excels when most of the support vectors are at bound. Thus, to determine the increase in speed caused by the sparse dot product code, both SMO and chunking were tested without the sparse dot product code:

Training Set Size	SMO time (sparse)	SMO time (non-sparse)	Chunking time (sparse)	Chunking time (non-sparse)
500	1.0	6.0	6.4	6.8
1000	3.5	21.7	57.9	62.1
2000	15.7	99.3	593.8	614.0
5000	67.6	400.0	10353.3	10597.7
10000	187.1	1007.6	N/A	N/A

In the linear SVM case, sparse dot product code sped up SMO by about a factor of 6, while chunking sped up only minimally. In this experiment, SMO is faster than chunking even for non-sparse data.

The second data set was also tested using Gaussian SVMs that have a variance of 10. The *C* value is still set to 0.1. The results for the Gaussian SVMs are presented in the two tables below:

Training Set Size	SMO time	Chunking time	Number of Non-Bound Support Vectors	Number of Bound Support Vectors
500	5.6	5.8	22	476
1000	21.1	41.9	82	888
2000	131.4	635.7	75	1905
5000	986.5	13532.2	30	4942
10000	4226.7	N/A	48	9897

Training Set Size	SMO time (sparse)	SMO time (non-sparse)	Chunking time (sparse)	Chunking time (non-sparse)
500	5.6	19.8	5.8	6.8
1000	21.1	87.8	41.9	53.0
2000	131.4	554.6	635.7	729.3
5000	986.5	3957.2	13532.2	14418.2
10000	4226.7	15743.8	N/A	N/A

For the Gaussian SVM's fit to pure noise, the SMO computation time scales as $\sim N^{2.2}$, while the chunking computation time scales as $\sim N^{3.4}$. The pure noise case yields the worst scaling so far, but SMO is superior to chunking by more than one order in scaling. The total run time of SMO is

still superior to chunking, even when applied to the non-sparse data. The sparsity of the input data yields a speed up of approximately a factor of 4 for SMO for the non-linear case, which indicates that the dot product speed is still dominating the SMO computation time for non-linear SVMs

4 CONCLUSIONS

SMO is an improved training algorithm for SVMs. Like other SVM training algorithms, SMO breaks down a large QP problem into a series of smaller QP problems. Unlike other algorithms, SMO utilizes the smallest possible QP problems, which are solved quickly and analytically, generally improving its scaling and computation time significantly.

SMO was tested on both real-world problems and artificial problems. From these tests, the following can be deduced:

- SMO can be used when a user does not have easy access to a quadratic programming package and/or does not wish to tune up that QP package.
- SMO does very well on SVMs where many of the Lagrange multipliers are at bound.
- SMO performs well for linear SVMs because SMO's computation time is dominated by SVM evaluation, and the evaluation of a linear SVM can be expressed as a single dot product, rather than a sum of linear kernels.
- SMO performs well for SVMs with sparse inputs, even for non-linear SVMs, because the kernel computation time can be reduced, directly speeding up SMO. Because chunking spends a majority of its time in the QP code, it cannot exploit either the linearity of the SVM or the sparseness of the input data.
- SMO will perform well for large problems, because its scaling with training set size is better than chunking for all of the test problems tried so far.

For the various test sets, the training time of SMO empirically scales between $\sim N$ and $\sim N^{2.2}$. The training time of chunking scales between $\sim N^{1.2}$ and $\sim N^{3.4}$. The scaling of SMO can be more than one order better than chunking. For the real-world test sets, SMO can be a factor of 1200 times faster for linear SVMs and a factor of 15 times faster for non-linear SVMs.

Because of its ease of use and better scaling with training set size, SMO is a strong candidate for becoming the standard SVM training algorithm. More benchmarking experiments against other QP techniques and the best Osuna heuristics are needed before final conclusions can be drawn.

ACKNOWLEDGEMENTS

Thanks to Lisa Heilbron for assistance with the preparation of the text. Thanks to Chris Burges for running a data set through his projected conjugate gradient code. Thanks to Leonid Gurvits for pointing out the similarity of SMO with Bregman methods.

REFERENCES

1. Bengio, Y., LeCun, Y., Henderson, D., "Globally Trained Handwritten Word Recognizer using Spatial Representation, Convolutional Neural Networks and Hidden Markov Models," *Advances in Neural Information Processing Systems*, 5, J. Cowan, G. Tesauro, J. Alspector, eds., 937-944, (1994).

2. Boser, B. E., Guyon, I. M., Vapnik, V., "A Training Algorithm for Optimal Margin Classifiers", *Fifth Annual Workshop on Computational Learning Theory*, ACM, (1992).

3. Bregman, L. M., "The Relaxation Method of Finding the Common Point of Convex Sets and Its Application to the Solution of Problems in Convex Programming," *USSR Computational Mathematics and Mathematical Physics*, 7:200-217, (1967).

4. Burges, C. J. C., "A Tutorial on Support Vector Machines for Pattern Recognition," submitted to Data Mining and Knowledge Discovery, http://svm.research.bell-labs.com/SVMdoc.html, (1998).

5. Censor, Y., "Row-Action Methods for Huge and Sparse Systems and Their Applications", *SIAM Review*, 23(4):444-467, (1981).

6. Censor, Y., Lent, A., "An Iterative Row-Action Method for Interval Convex Programming," *J. Optimization Theory and Applications*, 34(3):321-353, (1981).

7. Cortes, C., Vapnik, V., "Support Vector Networks," *Machine Learning*, 20:273-297, (1995).

8. Duda, R. O., Hart, P. E., *Pattern Classification and Scene Analysis*, John Wiley & Sons, (1973).

9. Joachims, T., "Text Categorization with Support Vector Machines", LS VIII Technical Report, No. 23, University of Dortmund, ftp://ftp-ai.informatik.uni-dortmund.de/pub/Reports/report23.ps.Z, (1997).

10. Hildreth, C., "A Quadratic Programming Procedure," *Naval Research Logistics Quarterly*, 4:79-85, (1957).

11. Gill, P. E., Murray, W., Wright, M. H., *Practical Optimization*, Academic Press, (1981).

12. LeCun, Y., Jackel, L. D., Bottou, L., Cortes, C., Denker, J. S., Drucker, H., Guyon, I., Muller, U. A., Sackinger, E., Simard, P. and Vapnik, V., "Learning Algorithms for Classification: A Comparison on Handwritten Digit Recognition," *Neural Networks: The Statistical Mechanics Perspective*, Oh, J. H., Kwon, C. and Cho, S. (Ed.), World Scientific, 261-276, (1995).

13. Mallat, S., *A Wavelet Tour of Signal Processing*, Academic Press, (1998).

14. Oren, M., Papageorgious, C., Sinha, P., Osuna, E., Poggio, T., "Pedestrian Detection Using Wavelet Templates," *Proc. Computer Vision and Pattern Recognition '97*, 193-199, (1997).

15. Osuna, E., Freund, R., Girosi, F., "Training Support Vector Machines: An Application to Face Detection," *Proc. Computer Vision and Pattern Recognition '97*, 130-136, (1997).

16. Osuna, E., Freund, R., Girosi, F., "Improved Training Algorithm for Support Vector Machines," *Proc. IEEE NNSP '97*, (1997).

17. Osuna, E., Personal Communication.

18. Platt, J. C., "A Resource-Allocating Network for Function Interpolation," *Neural Computation*, 3(2):213-225, (1991).

19. Vapnik, V., *Estimation of Dependences Based on Empirical Data*, Springer-Verlag, (1982).

20. Vapnik, V., *The Nature of Statistical Learning Theory*, Springer-Verlag, (1995).

APPENDIX: DERIVATION OF TWO-EXAMPLE MINIMIZATION

Each step of SMO will optimize two Lagrange multipliers. Without loss of generality, let these two multipliers be α_1 and α_2. The objective function Ψ from equation (11) can thus be written as

$$\Psi = \tfrac{1}{2}K_{11}\alpha_1^2 + \tfrac{1}{2}K_{22}\alpha_2^2 + sK_{12}\alpha_1\alpha_2 + y_1\alpha_1 v_1 + y_2\alpha_2 v_2 - \alpha_1 - \alpha_2 + \Psi_{\text{constant}}, \tag{24}$$

where

$$\begin{aligned} K_{ij} &= K(\vec{x}_i, \vec{x}_j), \\ v_i &= \sum_{j=3}^{N} y_j \alpha_j^* K_{ij} = u_i + b^* - y_1\alpha_1^* K_{1i} - y_2\alpha_2^* K_{2i}, \end{aligned} \tag{25}$$

and the starred variables indicate values at the end of the previous iteration. Ψ_{constant} are terms that do not depend on either α_1 or α_2.

Each step will find the minimum along the line defined by the linear equality constraint (6). That linear equality constraint can be expressed as

$$\alpha_1 + s\alpha_2 = \alpha_1^* + s\alpha_2^* = w. \tag{26}$$

The objective function along the linear equality constraint can be expressed in terms on α_2 alone:

$$\begin{aligned} \Psi = {} & \tfrac{1}{2}K_{11}(w - s\alpha_2)^2 + \tfrac{1}{2}K_{22}\alpha_2^2 + sK_{12}(w - s\alpha_2)\alpha_2 \\ & + y_1(w - s\alpha_2)v_1 - w + s\alpha_2 + y_2\alpha_2 v_2 - \alpha_2 + \Psi_{\text{constant}}. \end{aligned} \tag{27}$$

The extremum of the objective function is at

$$\frac{d\Psi}{d\alpha_2} = -sK_{11}(w - s\alpha_2) + K_{22}\alpha_2 - K_{12}\alpha_2 + sK_{12}(w - s\alpha_2) - y_2 v_1 + s + y_2 v_2 - 1 = 0. \tag{28}$$

If the second derivative is positive, which is the usual case, then the minimum of α_2 can be expressed as

$$\alpha_2(K_{11} + K_{22} - 2K_{12}) = s(K_{11} - K_{12})w + y_2(v_1 - v_2) + 1 - s. \tag{29}$$

Expanding the equations for *w* and *v* yields

$$\alpha_2(K_{11}+K_{22}-2K_{12}) = \alpha_2^*(K_{11}+K_{22}-2K_{12}) + y_2(u_1-u_2+y_2-y_1). \qquad (30)$$

More algebra yields equation (16).