12 Fast Training of Support Vector Machines using Sequential Minimal Optimization

John C. Platt
Microsoft Research
1 Microsoft Way, Redmond, WA 98052, USA
jplatt@microsoft.com
http://www.research.microsoft.com/~jplatt

This chapter describes a new algorithm for training Support Vector Machines: Sequential Minimal Optimization, or SMO. Training a Support Vector Machine (SVM) 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 large matrix computation is avoided, SMO scales somewhere between linear and quadratic in the training set size for various test problems, while a standard projected conjugate gradient (PCG) chunking 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. For the MNIST database, SMO is as fast as PCG chunking; while for the UCI Adult database and linear SVMs, SMO can be more than 1000 times faster than the PCG chunking algorithm.

12.1 Introduction

SVMs are starting to enjoy increasing adoption in the machine learning (27; 24) and computer vision research communities (33; 35). However, SVMs have not yet enjoyed widespread adoption in the engineering community. There are two possible reasons for the limited use by engineers. First, the training of SVMs is slow, especially for large problems. Second, SVM training algorithms are complex, subtle, and sometimes difficult to implement.

This chapter describes a new SVM learning algorithm that is conceptually simple, easy to implement, is often faster, and has better scaling properties than a standard "chunking" algorithm that uses projected conjugate gradient (PCG) (9). The new SVM learning algorithm is called *Sequential Minimal Optimization* (or SMO). Unlike previous SVM learning algorithms, which use numerical quadratic programming (QP) as an inner loop, SMO uses an analytic QP step. Because SMO spends most of its time evaluating the decision function, rather than performing QP, it can exploit data sets which contain a substantial number of zero elements. In this chapter, these data sets are called *sparse*. SMO does particularly well for sparse data sets, with either binary or non-binary input data.

This chapter first reviews current SVM training algorithms in section 12.1.1. The SMO algorithm is then described in detail in section 12.2, which includes 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, and some optimizations for special cases. Section 12.3 contains the pseudo-code of the algorithm, while section 12.4 discusses the relationship of SMO to other algorithms. Section 12.5 presents results for timing SMO versus a standard PCG chunking algorithm for various real-world and artificial data sets. Conclusions are drawn based on these timings in section 12.6. Two appendices (sections 12.7 and 12.8) contain the derivation of the analytic optimization and detailed tables of SMO versus PCG chunking timings.

Quadratic Program

For an overview of SVMs, please consult chapter 1. For completeness, the QP problem to train an SVM is shown below:

$$\begin{aligned} \max_{\boldsymbol{\alpha}} W(\boldsymbol{\alpha}) &= \sum_{i=1}^{\ell} \alpha_i - \frac{1}{2}\sum_{i=1}^{\ell}\sum_{j=1}^{\ell} y_i y_j k(\vec{x}_i, \vec{x}_j)\alpha_i\alpha_j, \\ 0 \le \alpha_i &\le C, \quad \forall i, \\ \sum_{i=1}^{\ell} y_i\alpha_i &= 0. \end{aligned} \tag{12.1}$$

The QP problem in equation (12.1) is solved by the SMO algorithm. A point is an optimal point of (12.1) if and only if the Karush-Kuhn-Tucker (KKT) conditions are fulfilled and $Q_{ij} = y_i y_j k(\vec{x}_i, \vec{x}_j)$ is positive semi-definite. Such a point may be a non-unique and non-isolated optimum. The KKT conditions are particularly simple; the QP problem is solved when, for all i:

KKT Conditions

$$\begin{aligned} \alpha_i = 0 &\Rightarrow y_i f(\vec{x}_i) \ge 1, \\ 0 < \alpha_i < C &\Rightarrow y_i f(\vec{x}_i) = 1, \\ \alpha_i = C &\Rightarrow y_i f(\vec{x}_i) \le 1. \end{aligned} \tag{12.2}$$

The KKT conditions can be evaluated one example at a time, which is useful in the construction of the SMO algorithm.

12.1.1 Previous Methods for Training Support Vector Machines

Due to its immense size, the QP problem (12.1) that arises from SVMs cannot easily be solved via standard QP techniques. The quadratic form in (12.1) involves a matrix that has a number of elements equal to the square of the number of training examples. This matrix cannot fit into 128 Megabytes if there are more than 4000 training examples (assuming each element is stored as an 8-byte double precision number).

Chunking

(52) 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 correspond 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 (12.2) (9), for some value of M (see figure 12.1). 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. The size of the QP sub-problem tends to grow with time, but can also shrink. 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 may not handle large-scale training problems, since even this reduced matrix may not fit into memory. One way to solve this problem is to use sophisticated data structures in the QP method (see, e.g., chapter 10). These data structures avoid the need to store the entire Hessian. The inner loop of such QP methods perform dot products between vectors and rows (or columns) of the Hessian, instead of a full matrix-vector multiply. In this chapter, the chunking benchmarks were implemented using the PCG algorithm, as suggested in the tutorial by (9).

Decomposition Algorithm

(34) suggested a new strategy for solving the SVM QP problem. Osuna showed 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 reduces the overall objective function and maintains a feasible point that obeys all of the constraints. Therefore, a sequence of QP sub-problems that always add at least one violator will asymptotically converge.

Osuna et al. suggest keeping a constant size matrix for every QP sub-problem, which implies adding and deleting the same number of examples at every step (34) (see figure 12.1). Using a constant-size matrix allows the training of arbitrarily sized data sets. The algorithm given in Osuna's paper (34) suggests adding one example and subtracting one example at every step. In practice, researchers add

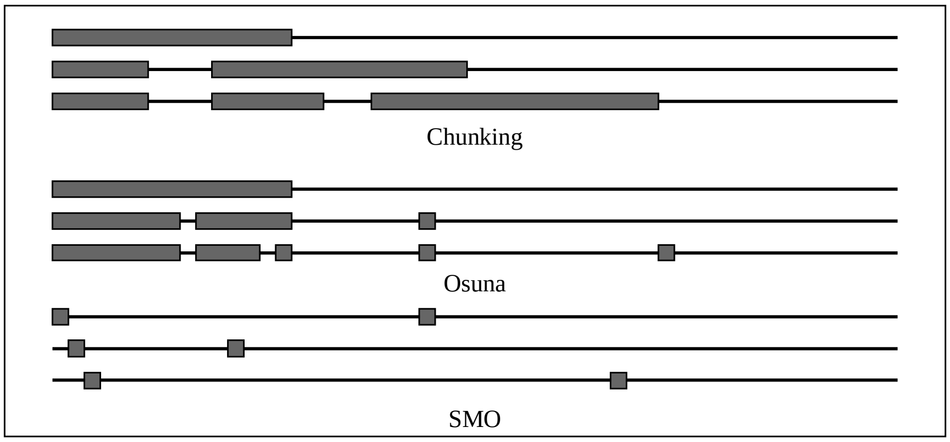

Figure 12.1 Three alternative methods for training SVMs: Chunking, Osuna's algorithm, and SMO. For each method, three steps are illustrated. The horizontal thin line at every step represents the training set, while the thick boxes represent the Lagrange multipliers being optimized at that step. A given group of three lines corresponds to three training iterations, with the first iteration at the top.

and subtract multiple examples using various techniques (see, e.g., chapter 11). In any event, a numerical QP solver is required for all of these methods. Numerical QP is tricky to get right; there are many numerical precision issues that need to be addressed.

12.2 Sequential Minimal Optimization

Sequential Minimal Optimization (SMO) is a simple algorithm that quickly solves the SVM QP problem without any extra matrix storage and without invoking an iterative numerical routine for each sub-problem. SMO decomposes the overall QP problem into QP sub-problems similar to Osuna's method.

SMO

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 12.1) [1]

1. It is possible to analytically optimize a small number of Lagrange multipliers that is greater than 2 (say, 3 or 4). No experiments have been done to test the effectiveness of such

The advantage of SMO lies in the fact that solving for two Lagrange multipliers can be done analytically. Thus, an entire inner iteration due to numerical QP optimization is avoided. The inner loop of the algorithm can be expressed in a small amount of C code, rather than invoking an entire iterative 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 can be solved quickly.

In addition, SMO does not require extra matrix storage (ignoring the minor amounts of memory required to store any 2x2 matrices required by SMO). Thus, very large SVM training problems can fit inside of the memory of an ordinary personal computer or workstation. Because manipulation of large matrices is avoided, SMO may be less susceptible to numerical precision problems.

There are three components to SMO: an analytic method to solve for the two Lagrange multipliers (described in section 12.2.1), a heuristic for choosing which multipliers to optimize (described in section 12.2.2), and a method for computing b (described in section 12.2.3). In addition, SMO can be accelerated using techniques described in section 12.2.4.

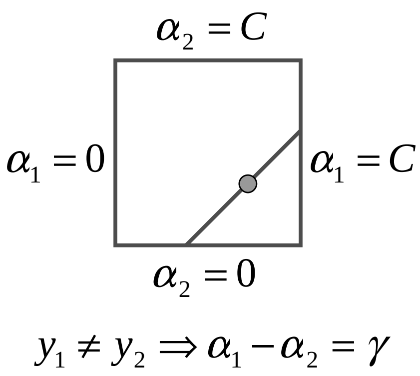

Figure 12.2 The two Lagrange multipliers must fulfill all of the constraints of the full problem. The inequality constraints cause the Lagrange multipliers to lie in the box. The linear equality constraint causes them to lie on a diagonal line. Therefore, one step of SMO must find an optimum of the objective function on a diagonal line segment. In this figure, $\gamma = \alpha_1^{\mathrm{old}} + s\alpha_2^{\mathrm{old}}$, is a constant that depends on the previous values of α_1 and α_2, and $s = y_1 y_2$.

a strategy. See chapter 11 for an algorithm that numerically optimizes a small number of multipliers.

12.2.1 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 maximum. 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 easily be displayed in two dimensions (see figure 12.2). The bound constraints in (12.1) cause the Lagrange multipliers to lie within a box, while the linear equality constraint in (12.1) causes the Lagrange multipliers to lie on a diagonal line. Thus, the constrained maximum of the objective function must lie on a diagonal line segment (as shown in figure 12.2). 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.

Constraints on α_2

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:

$$L = \max(0, \alpha_2^{\rm old} - \alpha_1^{\rm old}), \qquad H = \min(C, C + \alpha_2^{\rm old} - \alpha_1^{\rm old}). \tag{12.3}$$

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

$$L = \max(0, \alpha_1^{\rm old} + \alpha_2^{\rm old} - C), \qquad H = \min(C, \alpha_1^{\rm old} + \alpha_2^{\rm old}). \tag{12.4}$$

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

$$\eta = 2k(\vec{x}_1, \vec{x}_2) - k(\vec{x}_1, \vec{x}_1) - k(\vec{x}_2, \vec{x}_2). \tag{12.5}$$

The next step of SMO is to compute the location of the constrained maximum of the objective function in equation (12.1) while allowing only two Lagrange multipliers to change. The derivation of the maximum location is shown in section 12.7.

Under normal circumstances, there will be a maximum along the direction of the linear equality constraint, and η will be less than zero. In this case, SMO computes the maximum along the direction of the constraint:

Unconstrained Maximum

$$\alpha_2^{\rm new} = \alpha_2^{\rm old} - \frac{y_2(E_1 - E_2)}{\eta}, \tag{12.6}$$

where $E_i = f^{\rm old}(\vec{x}_i) - y_i$ is the error on the ith training example. Next, the constrained maximum is found by clipping the unconstrained maximum to the ends of the line segment:

Constrained Maximum

$$\alpha_2^{\rm new,clipped} = \begin{cases} H, & \text{if } \alpha_2^{\rm new} \geq H; \\ \alpha_2^{\rm new}, & \text{if } L < \alpha_2^{\rm new} < H; \\ L, & \text{if } \alpha_2^{\rm new} \leq L. \end{cases} \tag{12.7}$$

α_1 Computation

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

$$\alpha_1^{\text{new}} = \alpha_1^{\text{old}} + s(\alpha_2^{\text{old}} - \alpha_2^{\text{new,clipped}}). \tag{12.8}$$

Under unusual circumstances, η will not be negative. A zero η can occur if more than one training example has the same input vector $\vec{x}$. In any event, SMO will work even when η is not negative, in which case the objective function W should be evaluated at each end of the line segment. Only those terms in the objective function that depend on α_2 need be evaluated (see equation (12.23)). SMO moves the Lagrange multipliers to the end point with the highest 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 maximization cannot make progress. That scenario is described below.

12.2.2 Heuristics for Choosing Which Multipliers to Optimize

SMO will always optimize two Lagrange multipliers at every step, with one of the Lagrange multipliers having previously violated the KKT conditions before the step. That is, SMO will always alter two Lagrange multipliers to move uphill in the objective function projected into the one-dimensional feasible subspace. SMO will also always maintain a feasible Lagrange multiplier vector. Therefore, the overall objective function will increase at every step and the algorithm will converge asymptotically (34). In order to speed convergence, SMO uses heuristics to choose which two Lagrange multipliers to jointly optimize.

First Choice Heuristic

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 (12.2). If an example violates the KKT conditions, it is then eligible for immediate optimization. Once a violated example is found, a second multiplier is chosen using the second choice heuristic, and the two multipliers are jointly optimized. The feasibility of the dual QP (12.1) is always maintained. The SVM is then updated using these two new multiplier values, and the outer loop resumes looking for KKT violators.

Outer Loop

To speed training, the outer loop does not always iterate through the entire training set. After one pass through the training set, the outer loop iterates over only those 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 immediate optimization and update. 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 iterates over the entire training set again. 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. At that point, 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, Lagrange multipliers that are at the bounds are likely to stay at the bounds, while Lagrange multipliers that are not at the bounds will change 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.

Loose KKT Conditions

SMO verifies that the KKT conditions are fulfilled within ϵ. Typically, ϵ can typically be set in the range 10^{-2} to 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 required to produce very high accuracy output.

Second Choice Heuristic

Once a first Lagrange multiplier is chosen, SMO chooses the second Lagrange multiplier to maximize the size of the step taken during joint optimization. Evaluating the kernel function k is time consuming, so SMO approximates the step size by the absolute value of the numerator in equation (12.6): $|E_1 - E_2|$. 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.

Second Choice Hierarchy

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 $\vec{x}$, which causes the objective function to become flat along the direction of optimization. To avoid this problem, SMO uses a hierarchy of second choice heuristics until it finds a pair of Lagrange multipliers that can make positive progress. Positive progress can be determined by making a non-zero step upon joint optimization of the two Lagrange multipliers. The hierarchy of second choice heuristics consists of the following: (A) if the above heuristic does not make positive progress, then SMO starts iterating through the non-bound examples, searching for a second example that can make positive progress; (B) 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 (A) and the iteration through the entire training set (B) 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.

12.2.3 The Threshold and the Error Cache

Solving (12.1) for the Lagrange multipliers $\boldsymbol{\alpha}$ does not determine the threshold b of the SVM, so b must be computed separately. After each step, b is re-computed, 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 $\vec{x}_1$:

$$b_1 = E_1 + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})k(\vec{x}_1, \vec{x}_1) + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}})k(\vec{x}_1, \vec{x}_2) + b^{\text{old}}. \quad (12.9)$$

The following threshold b_2 is valid when the new α_2 is not at the bounds, because it forces the output of the SVM to be y_2 when the input is $\vec{x}_2$:

$$b_2 = E_2 + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})k(\vec{x}_1, \vec{x}_2) + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}})k(\vec{x}_2, \vec{x}_2) + b^{\text{old}}. \quad (12.10)$$

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. In this case, SMO chooses the threshold to be halfway in between b_1 and b_2.

Error Cache

As discussed in section 12.2.2, a cached error value E is kept for every example whose Lagrange multiplier is neither zero nor C. When a Lagrange multiplier is non-bound and is involved in a joint optimization, its cached error is set to zero. Whenever a joint optimization occurs, the stored errors for all non-bound multipliers α_k that are not involved in the optimization are updated according to

$$\begin{aligned} E_k^{\text{new}} = E_k^{\text{old}} &+ y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})k(\vec{x}_1, \vec{x}_k) \\ &+ y_2(\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}})k(\vec{x}_2, \vec{x}_k) + b^{\text{old}} - b^{\text{new}}. \end{aligned} \quad (12.11)$$

When an error E is required by SMO, it will look up the error in the error cache if the corresponding Lagrange multiplier is not at bound. Otherwise, it will evaluate the current SVM decision function based on the current $\boldsymbol{\alpha}$ vector.

12.2.4 Speeding Up SMO

A linear SVM can be sped up by only storing a single weight vector, rather than all of the training examples that correspond to non-zero Lagrange multipliers. If the joint optimization succeeds, this stored weight vector must be updated to reflect the new Lagrange multiplier values. The weight vector update is easy, due to the linearity of the SVM:

$$\vec{w}^{\text{new}} = \vec{w}^{\text{old}} + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})\vec{x}_1 + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}})\vec{x}_2. \quad (12.12)$$

Because much of the computation time of SMO is spent evaluating the decision function, anything that can speed up the decision function will speed up SMO. If the input data is sparse, then SMO can be sped up substantially.

Normally, an input vector is stored as a vector of floating-point numbers. A sparse input vector is stored as two arrays: `id` and `val`. The `id` array is an integer array that stores the location of the non-zero inputs, while the `val` array is a floating-point array that stores the corresponding non-zero values. The length of both arrays is the number of non-zero inputs.

The dot product for two sparse vectors (`id1, val1`, length = `num1`) and (`id2, val2`, length = `num2`) can be computed quite quickly by scanning through both

Sparse Dot Product

vectors, as shown in the pseudo-code below:

```
p1 = 0,  p2 = 0,  dot = 0
while (p1 < num1 && p2 < num2)
{
  a1 = id1[p1],   a2 = id2[p2]
  if (a1 == a2)
  {
    dot += val1[p1]*val2[p2]
    p1++, p2++
  }
  else if (a1 > a2)
    p2++
  else
    p1++;
}
```

The sparse dot product code can be used to compute linear kernels and polynomial kernels directly. Gaussian kernels can also use the sparse dot product code through the use of the following identity:

$$||\vec{x} - \vec{y}||^2 = \vec{x} \cdot \vec{x} - 2\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{y}. \tag{12.13}$$

For every input, the dot product of each input with itself is pre-computed and stored to speed up Gaussians even further.

Sparse Linear SVM

For a linear SVM, the weight vector is not stored as a sparse array. The dot product of the weight vector `w` with a sparse input vector (`id,val`) can be expressed as

$$\sum_{i=0}^{\mathtt{num}} \mathtt{w}[\mathtt{id}[\mathtt{i}]] * \mathtt{val}[\mathtt{i}]. \tag{12.14}$$

For binary inputs, storing the array `val` is not even necessary, since it is always 1. In the sparse dot product code, the floating-point multiplication becomes an increment. For a linear SVM, the dot product of the weight vector with a sparse input vector becomes

$$\sum_{i=0}^{\mathtt{num}} \mathtt{w}[\mathtt{id}[\mathtt{i}]]. \tag{12.15}$$

Notice that other code optimizations can be used, such as using look-up tables for the non-linearities or placing the dot products in a cache. Using a dot product cache can substantially speed up many of the SVM QP algorithms, at the expense of added code complexity and memory usage. In chapter 11, using a dot-product cache sped up $SVM^{light}$by a factor of 2.8 in one experiment. Combining SMO with a dot-product cache has not yet been tried.

12.3 Pseudo-Code

The pseudo-code for the overall SMO algorithm is presented below:

```
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
  if (L == H)
    return 0
  k11 = kernel(point[i1],point[i1])
  k12 = kernel(point[i1],point[i2])
  k22 = kernel(point[i2],point[i2])
  eta = 2*k12-k11-k22
  if (eta < 0)
  {
    a2 = alph2 - y2*(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 linear SVM
  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
      if takeStep(i1,i2)
        return 1
    }
    loop over all non-zero and non-C alpha, starting at 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:
  initialize alpha array to all zero
  initialize threshold to zero
  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
      }
```

12.4 Relationship to Previous Algorithms

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

Bregman Methods

SMO is closely related to a family of optimization algorithms called Bregman methods (6) or row-action methods (10). The classic Bregman method will minimize a function $F(\vec{x})$ subject to multiple constraints $\sum_i \vec{x} \cdot \vec{a}_i \leq b_i$. The Bregman method is iterative and updates an estimate of the optimum, $\vec{p}$. The method defines a function $D(\vec{x}, \vec{y})$:

$$D(\vec{x}, \vec{y}) = F(\vec{x}) - F(\vec{y}) - \nabla F(\vec{y}) \cdot (\vec{x} - \vec{y}). \tag{12.16}$$

The Bregman method iterates through all constraints. For each constraint, it finds the point, $\vec{z}$, that lies on the constraint and minimizes $D(\vec{p}, \vec{z})$. The estimate $\vec{p}$ is then set to $\vec{z}$. Each step is called a D-projection. Given certain conditions on F, including the requirement that the D-projection is unique, the Bregman method will converge (6; 11).

Unfortunately, the classic Bregman method does not work on an SVM with a threshold b. The input space of the function F must be the joint space $(\vec{w}, b)$. The function F would be the primal objective function τ which minimizes the norm of the weight vector in equation (1.9). In this case,

$$D([\vec{w}_1, b_1], [\vec{w}_2, b_2]) = \frac{1}{2}||\vec{w}_1 - \vec{w}_2||^2; \tag{12.17}$$

and the D-projection is not unique, because it cannot determine b. Hence, the classic Bregman method would not converge. Another way of explaining this outcome is that there is a linear equality constraint in the dual problem caused by b. Row-action methods can only vary one Lagrange multiplier at a time, hence they cannot fulfill the linear equality constraint.

Fixed-b SVMs

It is interesting to consider an SVM where b is held fixed at zero, rather than being a solved variable. A fixed-b SVM would not have a linear equality constraint in (12.1). Therefore, only one Lagrange multiplier would need to be updated at a time and a row-action method can be used. A traditional Bregman method is still not applicable to such SVMs, due to the slack variables ξ_i in equation (1.37). The presence of the slack variables causes the Bregman D-projection to become non-unique in the combined space of weight vectors and slack variables ξ_i.

Fortunately, SMO can be modified to solve fixed-b SVMs. SMO will update individual Lagrange multipliers to be the maximum of $W(\boldsymbol{\alpha})$ along the corresponding dimension. The update rule is

$$\alpha_1^{\text{new}} = \alpha_1^{\text{old}} + \frac{y_1 E_1}{k(\vec{x}_1, \vec{x}_1)}. \tag{12.18}$$

This update equation forces the output of the SVM to be y_1 (similar to Bregman methods or Hildreth's QP method (23)). After the new α_1 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 12.2.2.

Fixed-b SMO for a linear SVM is similar in concept to the perceptron relaxation rule (16), 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-b 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-b SMO for Gaussian kernels is also related to the Resource Allocating Network (RAN) algorithm (36). When RAN detects certain kinds of errors, it will allocate a basis function to exactly fix the error. SMO will perform similarly. However SMO/SVM will adjust the height of the basis functions to maximize the margin in a feature space, while RAN will simply use LMS to adjust the heights of the basis functions.

12.5 Benchmarking SMO

The SMO algorithm was tested against a standard PCG chunking SVM learning algorithm (9) 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. The CPU time for both algorithms are measured. The CPU time covers the execution of the entire algorithm, including kernel evaluation time, but excluding file I/O time.

The code for both algorithms is written to exploit the sparseness of the input vector and the linearity of the SVM, as described in section 12.2.4.

PCG

The chunking algorithm uses the PCG (18) algorithm as its QP solver (9). The chunk size was chosen to be 500. When the PCG code is initialized for a chunk, it assumes that all multipliers that are at bound have active equality constraints. It then releases those multipliers one at a time. This initialization causes the solver to avoid spuriously releasing and re-binding a large number of at-bound multipliers. Furthermore, the chunking algorithm re-uses the Hessian matrix elements from one chunk to the next, in order to minimize the number of extraneous dot products evaluated. In order to limit the amount of memory used by the algorithms, neither the chunking nor the SMO code use kernel caching to evaluate the decision function over the entire training set. Kernel caching for the decision function would favor SMO, because most of the computation time in SMO is spent in computing the decision function.

To further speed up PCG, the computation of the gradient is done sparsely: only those rows or columns of the Hessian that correspond to non-zero Lagrange multipliers are multiplied by the estimated Lagrange multiplier vector (see chapter 10). The computation of the quadratic form in the PCG algorithm is also performed sparsely: the computation is only performed over the active variables.

In order to ensure that the chunking algorithm is a fair benchmark, Burges compared the speed of his PCG 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 a reasonable benchmark.

Stopping Criteria

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. A larger threshold may be equally insignificant and cause both QP algorithms to become faster.

The PCG code has a stopping threshold which describes the minimum relative improvement in the objective function at every step (9). If the PCG takes a step where the relative improvement is smaller than this minimum, the conjugate gradient code terminates and another chunking step is taken. (9) recommends using a constant 10^{-10} for this minimum, which works well with a KKT tolerance of 2×10^{-2}.

In the experiments below, stopping the PCG at an accuracy of 10^{-10} sometimes 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 3×10^{-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 determine the value for b (see 9). Every example suggests a proposed threshold. If the largest proposed threshold is more than 2×10^{-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 stopping 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 3×10^{-10}. The smallest threshold used was 3.7×10^{-12}, which occurred at the end of the chunking for the largest web page classification problem.

Experiment	Kernel	Sparse Code Used	Training Set Size	C	% Sparse
Adult Linear Small	Linear	Y	11221	0.05	89%
Adult Linear Large	Linear	Y	32562	0.05	89%
Web Linear	Linear	Y	49749	1	96%
Lin. Sep. Sparse	Linear	Y	20000	100	90%
Lin. Sep. Dense	Linear	N	20000	100	0%
Random Linear Sparse	Linear	Y	10000	0.1	90%
Random Linear Dense	Linear	N	10000	0.1	0%
Adult Gaussian Small	Gaussian	Y	11221	1	89%
Adult Gaussian Large	Gaussian	Y	32562	1	89%
Web Gaussian	Gaussian	Y	49749	5	96%
Random Gaussian Sparse	Gaussian	Y	5000	0.1	90%
Random Gaussian Dense	Gaussian	N	5000	0.1	90%
MNIST	Polynomial	Y	60000	100	81%

Table 12.1 Parameters for various experiments

12.5.1 Experimental Results

The SMO algorithm was tested on the UCI Adult benchmark set, a web page classification task, the MNIST database, and two different artificial data sets. A summary of the experimental results are shown in tables 12.1 and 12.2.

In table 12.2, the scaling of each algorithm is measured as a function of the training set size, which was varied by taking random nested subsets of the full training set. A line was fitted to the log of the training time versus the log of the training set size. The slope of the line is an empirical scaling exponent.

The "N/A" entries in the chunking time column of table 12.2 had matrices that were too large to fit into 128 Megabytes, hence could not be timed due to memory thrashing.

All of the data sets (except for MNIST and the linearly separable data sets) were trained both with linear SVMs and Gaussian SVMs with a variance of 10. For the Adult and Web data sets, the C parameter and the Gaussian variance were chosen to optimize accuracy on a validation set.

UCI Adult Data Set

The first data set used to test SMO's speed was the UCI Adult data set (30). The SVM was given 14 attributes of a census form of a household and asked to predict whether that household has an income greater than \$50,000. Out of the 14 attributes, eight are categorical and six are continuous. The six continuous attributes were discretized into quintiles, which yielded a total of 123 binary attributes. The full timings for the Adult data set are shown in tables 12.3 and 12.4 in section 12.8. For this data set, the scaling for SMO is approximately one order in the exponent faster than PCG chunking. For the entire Adult training set,

Experiment	SMO Time (sec)	Chunking Time (sec)	SMO Scaling Exponent	PCG Scaling Exponent
Adult Linear Small	17.0	20711.3	1.9	3.1
Adult Linear Large	163.6	N/A	1.9	3.1
Web Linear	268.3	17164.7	1.6	2.5
Lin. Sep. Sparse	280.0	374.1	1.0	1.2
Lin. Sep. Dense	3293.9	397.0	1.1	1.2
Random Linear Sparse	67.6	10353.3	1.8	3.2
Random Linear Dense	400.0	10597.7	1.7	3.2
Adult Gaussian Small	781.4	11910.6	2.1	2.9
Adult Gaussian Large	7749.6	N/A	2.1	2.9
Web Gaussian	3863.5	23877.6	1.7	2.0
Random Gaussian Sparse	986.5	13532.2	2.2	3.4
Random Gaussian Dense	3957.2	14418.2	2.3	3.1
MNIST	29471.0	33109.0	N/A	N/A

Table 12.2 Summary of Timings of SMO versus PCG Chunking on various data sets.

SMO is more than 1000 times faster than PCG chunking for a linear SVM and approximately 15 times faster than PCG chunking for the Gaussian SVM. The adult data set shows that, for real-world sparse problems with many support vectors at bound, SMO is much faster than PCG chunking.

Web Page Data Set

Another test of SMO was on text categorization: classifying whether a web page belongs to a category or not. Each input was 300 sparse binary keyword attributes extracted from each web page. The full timings are shown in tables 12.5 and 12.6. For the linear SVM, the scaling for SMO is one order better than PCG chunking. For the non-linear SVM, SMO is between two and six times faster than PCG chunking. The non-linear test shows that SMO is still faster than PCG chunking when the number of non-bound support vectors is large and the input data set is sparse.

MNIST Data Set

Yet another test of SMO was the MNIST database of 60,000 handwritten digits, from AT&T Research Labs (27). One classifier of MNIST was trained: class 8. The inputs are non-binary and are stored as a sparse vector. A fifth-order polynomial kernel, a C of 100, and a KKT tolerance of 0.02 was used to match the AT&T accuracy results. There were 3450 support vectors, with no support vectors at upper bound. Scaling experiments were not done on the MNIST database. However, the MNIST data was trained with both $C = 100$ and $C = 10$. The results for both of these training runs is shown in table 12.7. The MNIST experiment shows that SMO is competitive with PCG chunking for non-linear SVMs trained on moderately sparse data sets with none or very few support vectors at the upper bound.

Linearly Separable Data Set

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 consisted of random binary 300-dimensional vectors, with a 10% fraction of "1" inputs. If the dot product of a stored vector (uniform random in $[-1, 1]$) with an input point was greater than 1, then a positive label was assigned to the input point. If the dot product was less than -1, then a negative label was assigned. If the dot product lay between -1 and 1, the point was discarded. A linear SVM was fit to this data set. The full timing table is shown in table 12.8.

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 PCG chunking. For this easy sparse problem, therefore, PCG chunking and SMO are generally comparable.

Sparse vs. Non-Sparse

The acceleration of both the SMO algorithm and the PCG 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 full timing table for this experiment is shown in table 12.9.

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 decision function totally dominates the SMO computation time. The sparse dot product code only speeds up PCG chunking by about 6%, which shows that the evaluation of the numerical QP steps dominates the PCG 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 PCG 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 PCG chunking comes from the exploitation of sparse dot product code. Fortunately, real-world problems with sparse input are not rare. Any quantized or fuzzy-membership-encoded problems will be sparse. Also, optical character recognition (27), handwritten character recognition (3), and wavelet transform coefficients of natural images (33; 28) can be naturally expressed as sparse data.

Random Data Set

The second artificial data set was generated with random 300-dimensional binary input points (10% "1") and random output labels. Timing experiments were performed for both linear and Gaussian SVMs and for both sparse and non-sparse code. The results of the timings are shown in tables 12.10 through 12.13. Scaling for SMO and PCG chunking is much higher on the second data set both for the linear and Gaussian SVMs. The second data set shows that SMO excels when most of the support vectors are at bound.

For the second data set, non-sparse SMO is still faster than PCG chunking. For the linear SVM, sparse dot product code sped up SMO by about a factor of 6. For the Gaussian SVM, the sparse dot product code sped up SMO by about a factor of 4. In neither case did the PCG chunking code have a noticable speed up. These experiments illustrate that the dot product speed is still dominating the SMO computation time for both linear and non-linear SVMs.

12.6 Conclusions

As can be seen in table 12.2, SMO has better scaling with training set size than PCG chunking for all data sets and kernels tried. Also, the memory footprint of SMO grows only linearly with the training set size. SMO should thus perform well on the largest problems, because it scales very well.

Table 12.2 also shows the effect of sparseness on the speed of SMO. Linear SVMs with 90% sparseness are a factor of 6 to 12 times faster using sparse binary SMO code over standard floating-point array SMO code. Even non-linear SVMs with 90% sparseness are a factor of 4 times faster. These results show that SMO is dominated by decision function evaluation time, and hence benefits from sparseness and binary inputs. In contrast, PCG chunking is dominated by numerical QP time: PCG chunking only speeds up by 6% by exploiting sparse decision function code. These experiments indicate that SMO is well-suited for sparse data sets.

SMO is up to a factor of 1200 times faster for linear SVMs, while up to a factor of 15 times faster for non-linear SVMs. Linear SVMs benefit from the acceleration of the decision function as described in section 12.2.4. Therefore, SMO is well-suited for learning linear SVMs.

Finally, SMO can be implemented without requiring a QP library function, which leads to simplification of the code and may lead to more widespread use of SVMs in the engineering community. While SMO is not faster than PCG chunking for all possible problems, its potential for speed-up should make it a key element in an SVM toolbox.

Acknowledgements

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

12.7 Appendix: Derivation of Two-Example Maximization

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 from equation (12.1) can thus be written as

$$\begin{aligned} W(\alpha_1, \alpha_2) = \alpha_1 + \alpha_2 - \frac{1}{2}K_{11}\alpha_1^2 - \frac{1}{2}K_{22}\alpha_2^2 - sK_{12}\alpha_1\alpha_2 \\ -y_1\alpha_1 v_1 - y_2\alpha_2 v_2 + W_{\text{constant}}, \end{aligned} \tag{12.19}$$

where

$$K_{ij} = k(\vec{x}_i, \vec{x}_j), \tag{12.20}$$

$$v_i = \sum_{j=3}^{\ell} y_j \alpha_j^{\text{old}} K_{ij} = f^{\text{old}}(\vec{x}_i) + b^{\text{old}} - y_1 \alpha_1^{\text{old}} K_{1i} - y_2 \alpha_2^{\text{old}} K_{2i}, \qquad (12.21)$$

and the variables with "old" superscripts indicate values at the end of the previous iteration. W_{constant} are terms that do not depend on either α_1 or α_2.

Each step will find the maximum along the line defined by the linear equality constraint in (12.1). That linear equality constraint can be expressed as

$$\alpha_1 + s\alpha_2 = \alpha_1^{\text{old}} + s\alpha_2^{\text{old}} = \gamma. \qquad (12.22)$$

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

$$\begin{aligned} W = \gamma - s\alpha_2 + \alpha_2 - \frac{1}{2}K_{11}(\gamma - s\alpha_2)^2 - \frac{1}{2}K_{22}\alpha_2^2 - sK_{12}(\gamma - s\alpha_2)\alpha_2 \\ -y_1(\gamma - s\alpha_2)v_1 - y_2\alpha_2 v_2 + W_{\text{constant}}. \end{aligned} \qquad (12.23)$$

The stationary point of the objective function is at

$$\begin{aligned} \frac{dW}{d\alpha_2} = sK_{11}(\gamma - s\alpha_2) - K_{22}\alpha_2 + K_{12}\alpha_2 - sK_{12}(\gamma - s\alpha_2) \\ +y_2 v_1 - s - y_2 v_2 + 1 = 0. \end{aligned} \qquad (12.24)$$

If the second derivative along the linear equality constraint is positive, then the maximum of the objective function can be expressed as

$$\alpha_2^{\text{new}}(K_{11} + K_{22} - 2K_{12}) = s(K_{11} - K_{12})\gamma + y_2(v_1 - v_2) + 1 - s. \qquad (12.25)$$

Expanding the equations for γ and v yields

$$\begin{aligned} \alpha_2^{\text{new}}(K_{11} + K_{22} - 2K_{12}) = \alpha_2^{\text{old}}(K_{11} + K_{22} - 2K_{12}) \\ +y_2(f(\vec{x}_1) - f(\vec{x}_2) + y_2 - y_1). \end{aligned} \qquad (12.26)$$

More algebra yields equation (12.6).

12.8 Appendix: SMO vs. PCG Chunking Tables

This section contains the timing tables for the experiments described in this chapter.

A column labeled "Non-Bound SVs" contains the number of examples whose Lagrange multipliers lie in the open interval $(0, C)$. A column labeled "Bound SVs" contains the number of examples whose Lagrange multipliers exactly equal C. These numbers are produced by SMO: the number of support vector produced by PCG chunking is slightly different, due to the loose KKT stopping conditions.

A column labeled "SMO Iterations" contains the number of successful joint optimizations taken (joint optimizations that do not make progress are excluded). A column labeled "PCG Iterations" contains the number of projected conjugate gradient steps taken, summed over all chunks.

Training Set Size	SMO Time (CPU sec)	PCG Time (CPU sec)	Non-Bound SVs	Bound SVs	SMO Iterations	PCG Iterations
1605	0.4	37.1	42	633	3230	1328
2265	0.9	228.3	47	930	4635	3964
3185	1.8	596.2	57	1210	6950	6742
4781	3.6	1954.2	63	1791	9847	10550
6414	5.5	3684.6	61	2370	10669	12263
11221	17.0	20711.3	79	4079	17128	25400
16101	35.3	N/A	67	5854	22770	N/A
22697	85.7	N/A	88	8209	35822	N/A
32562	163.6	N/A	149	11558	44774	N/A

Table 12.3 SMO and PCG Chunking for a linear SVM on the Adult data set.

Training Set Size	SMO Time (CPU sec)	PCG Time (CPU sec)	Non-Bound SVs	Bound SVs	SMO Iterations	PCG Iterations
1605	15.8	34.8	106	585	3349	1064
2265	32.1	144.7	165	845	5149	2159
3185	66.2	380.5	181	1115	6773	3353
4781	146.6	1137.2	238	1650	10820	5164
6414	258.8	2530.6	298	2181	14832	8085
11221	781.4	11910.6	460	3746	25082	14479
16101	1784.4	N/A	567	5371	34002	N/A
22697	4126.4	N/A	813	7526	51316	N/A
32562	7749.6	N/A	1011	10663	77103	N/A

Table 12.4 SMO and PCG Chunking for a Gaussian SVM on the Adult data set.

Training Set Size	SMO Time (CPU sec)	PCG Time (CPU sec)	Non-Bound SVs	Bound SVs	SMO Iterations	PCG Iterations
2477	2.2	13.1	123	47	25296	1929
3470	4.9	16.1	147	72	46830	2379
4912	8.1	40.6	169	107	66890	4110
7366	12.7	140.7	194	166	88948	7416
9888	24.7	239.3	214	245	141538	8700
17188	65.4	1633.3	252	480	268907	27074
24692	104.9	3369.7	273	698	345736	32014
49749	268.3	17164.7	315	1408	489302	63817

Table 12.5 SMO and PCG Chunking for a linear SVM on the Web data set.

Training Set Size	SMO Time (CPU sec)	PCG Time (CPU sec)	Non-Bound SVs	Bound SVs	SMO Iterations	PCG Iterations
2477	26.3	64.9	439	43	10838	1888
3470	44.1	110.4	544	66	13975	2270
4912	83.6	372.5	616	90	18978	5460
7366	156.7	545.4	914	125	27492	5274
9888	248.1	907.6	1118	172	29751	5972
17188	581.0	3317.9	1780	316	42026	9413
24692	1214.0	6659.7	2300	419	55499	14412
49749	3863.5	23877.6	3720	764	93358	24235

Table 12.6 SMO and PCG Chunking for a Gaussian SVM on the Web data set.

C	SMO (CPU sec)	Chunking (CPU sec)	Non-Bound SVs	Bound SVs
10	25096	29350	3263	149
100	29471	33109	3450	0

Table 12.7 CPU time for MNIST while varying C

Training Set Size	SMO Time (CPU sec)	PCG Time (CPU sec)	Non-Bound SVs	Bound SVs	SMO Iterations	PCG Iterations
1000	15.3	10.4	275	0	66920	1305
2000	33.4	33.0	286	0	134636	2755
5000	103.0	108.3	299	0	380395	7110
10000	186.8	226.0	309	0	658514	14386
20000	280.0	374.1	329	0	896303	20794

Table 12.8 SMO and PCG Chunking for a linear SVM on a linearly separable data set.

Training Set Size	Sparse SMO (CPU sec)	Non-Sparse SMO (CPU sec)	Sparse Chunking (CPU sec)	Non-Sparse Chunking (CPU sec)
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

Table 12.9 Comparison of sparse and non-sparse training time for a linearly separable data set.

Training Set Size	SMO Time (CPU sec)	PCG Time (CPU sec)	Non-Bound SVs	Bound SVs	SMO Iterations	PCG Iterations
500	1.0	6.4	162	263	5697	548
1000	3.5	57.9	220	632	12976	1529
2000	15.7	593.8	264	1476	38107	3720
5000	67.6	10353.3	283	4201	87109	7815
10000	187.1	N/A	293	9034	130774	N/A

Table 12.10 SMO and PCG Chunking for a linear SVM on a random data set.

Training Set Size	Sparse SMO (CPU sec)	Non-Sparse SMO (CPU sec)	Sparse Chunking (CPU sec)	Non-Sparse Chunking (CPU sec)
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

Table 12.11 Comparison of sparse and non-sparse training time for linear SVM applied to a random data set.

Training Set Size	SMO Time (CPU sec)	PCG Time (CPU sec)	Non-Bound SVs	Bound SVs	SMO Iterations	PCG Iterations
500	5.6	5.8	22	476	901	511
1000	21.1	41.9	82	888	1840	1078
2000	131.4	635.7	75	1905	3564	3738
5000	986.5	13532.2	30	4942	7815	14178
10000	4226.7	N/A	48	9897	15213	N/A

Table 12.12 SMO and PCG Chunking for a Gaussian SVM on a random problem.

Training Set Size	Sparse SMO (CPU sec)	Non-Sparse SMO (CPU sec)	Sparse Chunking (CPU sec)	Non-Sparse Chunking (CPU sec)
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

Table 12.13 Comparison of sparse and non-sparse training time for a Gaussian SVM applied to a random data set.

13 Support Vector Machines for Dynamic Reconstruction of a Chaotic System