

# An Efficient Implementation of an Active Set Method for SVMs

**Katya Scheinberg**

KATYAS@US.IBM.COM

*Mathematical Science Department  
IBM T. J. Watson Research Center  
1101 Kitchawan Road  
Yorktown Heights, NY*

**Editors:** Kristin P. Bennett and Emilio Parrado-Hernández

## Abstract

We propose an active set algorithm to solve the convex quadratic programming (QP) problem which is the core of the support vector machine (SVM) training. The underlying method is not new and is based on the extensive practice of the Simplex method and its variants for convex quadratic problems. However, its application to large-scale SVM problems is new. Until recently the traditional active set methods were considered impractical for large SVM problems. By adapting the methods to the special structure of SVM problems we were able to produce an efficient implementation. We conduct an extensive study of the behavior of our method and its variations on SVM problems. We present computational results comparing our method with Joachims' SVM<sup>light</sup> (see Joachims, 1999). The results show that our method has overall better performance on many SVM problems. It seems to have a particularly strong advantage on more difficult problems. In addition this algorithm has better theoretical properties and it naturally extends to the incremental mode. Since the proposed method solves the standard SVM formulation, as does SVM<sup>light</sup>, the generalization properties of these two approaches are identical and we do not discuss them in the paper.

**Keywords:** active set methods, support vector machines, quadratic programming

## 1. Introduction

In this paper we introduce an active set method to solve the following convex quadratic programming (QP) optimization problem which is defined by the classic soft margin SVM formulation (see, for example, Cristianini and Shawe-Taylor, 2000).

$$\begin{aligned} \max \quad & -\frac{1}{2}\alpha^T Q\alpha - c^T \xi \\ & -Q\alpha + by + s - \xi = -e, \\ & 0 \leq \alpha \leq c, s \geq 0, \xi \geq 0, \end{aligned} \tag{1}$$

where  $\alpha \in \mathbf{R}^n$  is the vector of the dual variables,  $b$  is the bias (scalar) and  $s$  and  $\xi$  are the  $n$ -dimensional vectors of the slack and the surplus variables, respectively.  $y$  is a vector of the labels,  $\pm 1$ .  $Q$  is the label encoded kernel matrix,  $Q_{ij} = y_i y_j K(x_i, x_j)$ ,  $e$  is the vector of all 1's of length  $n$  and  $c$  is the penalty vector associated with the errors (in standard soft margin SVMs the vector  $c$  is a product of vector  $e$  and a scalar penalty  $C$ , but here we will allow for any nonnegative vector  $c$ ). The dual of this problem is

$$\min \quad \frac{1}{2}\alpha^T Q\alpha - e^T \alpha$$

$$\begin{aligned} \text{s.t.} \quad & y^T \alpha = 0, \\ & 0 \leq \alpha \leq c. \end{aligned} \tag{2}$$

To confirm that problem (1) is equivalent to the traditional soft margin SVM formulation

$$\begin{aligned} \min \quad & \frac{1}{2} w^T w + c^T \xi \\ \text{s.t.} \quad & y_i (w^T x_i - b) - s_i + \xi_i \geq 1, \quad i = 1, \dots, n \\ & s \geq 0, \xi \geq 0, \end{aligned} \tag{3}$$

observe that (2) is the same as the dual of (3) and from optimality conditions of (3) and (2) we have  $w = \sum_{i=1}^n y_i \alpha_i x_i$ . Substituting this expression for  $w$  to (3) and denoting  $Q_{ij} = y_i y_j \alpha_i \alpha_j x_i^T x_j$  (or  $Q_{ij} = y_i y_j \alpha_i \alpha_j K(x_i, x_j)$  in the kernel case) we obtain the convex QP formulation (1), which we will consider in this paper. Hence, (1) and soft margin SVM enjoy the *same generalization properties*.

General convex QPs are typically solved by one of the two approaches: *interior point method* approach or *active set method* approach. If the Hessian of an objective function (matrix  $Q$  in the case of SVM) and/or the constraint matrix of the QP problem is large and sparse then an interior point method is usually the method of choice. If the problem is of moderate size but the matrices are dense, then active set method is preferable. In SVM problems the  $Q$  matrix is typically dense. Thus, large SVM problems present a challenge for both approaches. It was shown by Fine and Scheinberg (2001) and Ferris and Munson (2000) that for some classes of SVMs, for which  $Q$  is dense but low-rank, one can adapt an interior point method to work very efficiently. However, if the rank of  $Q$  is high, an active set approach seems to remain the only main alternative.

One of the most “traditional” active set methods in the optimization literature is the *Simplex* method for linear programming (LP) problems. The Simplex method is known to have very good practical performance. The QP analogues, though not as extensively tested in practice, are also considered to be very efficient. There are a few methods based on the Simplex method idea for solving QP problems (see Fletcher, 1971; Goldfarb, 1972; Goldfarb and Idnani, 1983). Many of them are theoretically equivalent, meaning that they produce the same sequence of iterations, but they have different numerical properties (such as per-iteration complexity and numerical stability). In this paper we derive an implementation targeted to SVM problems based on the framework described in Fletcher (1971), Goldfarb (1972) and Nocedal and Wright (1999).

The main idea of this method in the context of SVM is to fix, at each iteration, all variables in the current dual active set<sup>1</sup> at their current values (0 or  $c$ ), and then to solve the reduced dual problem. After obtaining a solution - decide whether it is optimal for the overall dual problem (same as being feasible for the overall primal problem), or if any of the dual variables should be released from the active set.

When applied to SVM, this approach poses the following problem: if the complement of the dual active set (the set of “free” variables) has large cardinality, then solving the restricted subproblems may be too expensive, since  $Q$  is completely dense. Also determining the next variable to leave the active set may be expensive for the same reason. Therefore, updating all “free” variables at once was considered impractical.

The most common approach to large SVM problems is to use a restricted active set method, such as chunking (Boser et al., 1992) or decomposition (Osuna et al., 1997; Joachims, 1999) where

---

1. The dual active set is the set of dual variables  $\alpha$  whose values are at their bound.

at each iteration only a small number of variables are allowed to be varied. The size of such “chunk” is determined heuristically or is chosen by the user. There are a few skillfully implemented SVM solvers based on this type of restricted active set methods (Joachims, 1999; Platt, 1999). The main disadvantage of these methods is that they tend to have slow convergence when getting closer to the optimal solution. Moreover, their performance is sensitive to the changes in the chunk size and there is no good way of predicting a good choice for the size of the chunks for a particular problem.<sup>2</sup>

A full active set method, such as the one presented in this paper, avoids these disadvantages. The method itself is not new (see Nocedal and Wright, 1999). Our contribution is to adapt it to the SVM framework and provide an efficient implementation.

First we notice that a support vector that violates the margin constraint (that is the  $\xi$  surplus variable is positive) corresponds to a variable  $\alpha$  which is at its upper bound and therefore is in the dual active set. The complement of the dual active set contains variables  $\alpha$  that are strictly between the upper and lower bounds. Such variables correspond only to the support vectors that are exactly on the margin (that is both the corresponding slack and the surplus variables are zero). The current number of such support vectors,  $n_s$ , is the size of the reduced QP (RQP). Solving such RQP directly (say, by an IPM method, as it is done in SVM<sup>light</sup>) requires at least  $O(n_s^3)$  operations, which might be prohibitively expensive if repeated over and over again and if  $n_s$  is relatively large. We do not solve RQP directly, but only make one step toward its solution at each iteration. Moreover, the active set is incremented only by one variable at a time (either one variable leaving, or one entering the active set), hence we can store and update a factorization of the reduced matrix  $Q$ . Each update takes  $O(n_s^2)$  operations and so does solving a system of equations with the reduced matrix  $Q$ .

At each such step toward the optimal solution of the RQP, we either find that solution or encounter a bound on one of the “free” variables. In the latter case this variable is included into the active set and the process repeats.

This process does not always produce an optimal solution to the subproblem, but usually produces a good approximation of it. Typically this does not affect the overall number of iterations significantly, whereas the reduction of the per-iteration cost is significant.

The RQP may sometimes have an infinite solution, if reduced matrix  $Q$  is singular. In this case an infinite descent direction is computed and a step is taken along this direction until one of the variable bounds is encountered. We provide the full treatment of the various cases for solving the RQP subproblem. We use the approach described in Frangioni (1996).

If the search for the optimum of the RQP subproblem is terminated then our method determines whether the primal feasibility was achieved and if not, which dual variable should leave the active set. To do that we need to compute a product of a submatrix of  $Q$  that corresponds to the variables at their upper bounds and the unit vector of an appropriate length. This can be very expensive to compute at each iteration, instead one should rather store and update the result of this multiplication. Another advantage of using “one-variable-at-a-time” increments is in potentially reducing the cost of such updates.

The multiple updates to the active set, which are used in “chunking” and “decomposition” methods could still have an advantage if the overall number of iterations were significantly smaller than in the case of single updates. But as our computational results indicate this is not the case. We offer some intuition to support this claim. Assume that your data contains 10 identical data points which at the current iteration are the most violated examples and we would like to introduce them into the

---

2. See Section 12.1.1 in Platt (1999) for a similar discussion which motivated Platt’s SMO. Essentially SMO is an active set method in which the chunk size is fixed to be the smallest possible, namely 2.

next “chunk”. Introducing all 10 at once implies 10 times more work than introducing just one. Yet since they are identical, introducing just one produces the same result as introducing all ten. Since the training data is often somewhat repetitive (there may not be *identical* points, but rather very similar points, for instance, in clustered data sets) this example is not too far fetched.

As the computational results show, our method has particular advantage over  $SVM^{light}$  on problems where the number of the support vectors or the number of outliers is large (but not necessarily excessive, such as  $\sim 1000$  out of 20000 vectors). Our algorithm currently requires the storage of the Cholesky factors of the reduced matrix  $Q$ , which might require excessive amount of memory for problems where the number of unbounded support vectors is very large. However, this often means that the chosen kernel suffers from overfitting the data, so the problem is badly posed in some sense, unless the entire test set is very large, in which case one should consider a different implementation, and, possibly, a more powerful computer.

The most expensive step of our algorithm (and of  $SVM^{light}$ , in fact) is pricing the primal constraints and choosing the next constraint to enter the active set. We will compare two approaches. One of these approaches is *shrinking*, which is used by  $SVM^{light}$ , and the other is *sprint* which is an industry standard in advanced implementations of LP solvers (Bixby et al., 1992). We observe that *sprint* appears to work better than *shrinking* on difficult SVM problems.

The proposed method enjoys several theoretical advantages compared to the methods based on *chunking*. First of all it converges in a finite number of iterations (Fletcher, 1971; Frangioni, 1996). In the worst case this number might be exponential, but it is hardly the case in practice. The method is also well suited for analysis of various situations. For instance, in Balcazar et al. (2001) a randomized active set algorithm for SVM is introduced and shown to have a quasi-linear average complexity. Our algorithm can be easily adapted to fit the randomized framework of Balcazar et al. (2001), hence similar average case results apply.

Recently, active set methods for SVM similar to ours were used in Cauwenberghs and Poggio (2001) for incremental learning and in Hastie et al. (2004) for generating the entire regularization path. Their methods, unlike ours, require primal and dual feasibility to be satisfied at every iteration and progress by changing the optimization problem itself (in a manner dictated by the respective uses of their methods). However, many of the efficiency issues of the algorithms are similar, such as the possible singularity of the reduced matrix  $Q$  and efficient updates of its Cholesky factorization. Though we choose to focus on one specific active set method in this paper, we believe the the experience we present here will be useful for other active set methods for SVM problems. Some of the ideas presented in the paper to improve the efficiency of the active set methods for SVMs were also suggested in Kaufman (1998).

The paper is organized as follows. In the next section we introduce the dual active set method for the soft-margin SVM problem and describe the details of solving the reduced QP problem. In Section 3 we will present the results of comparing our method to  $SVM^{light}$  on a selection of classification problems from the UCI repository (Blake and Merz, 1998). In Section 4 we will focus on various implementational issues that arise in the attempt to improve the performance of the method. In Subsection 4.4 we apply our method to the incremental case. Section 5 contains some conclusions.

## 2. Dual Active Set Method for SVMs

Any optimal solution to problems (1) or (2) must satisfy the Karush-Kuhn-Tucker (KKT) necessary and sufficient optimality conditions:

- 1  $\alpha_i s_i = 0, \quad i = 1, \dots, n$
- 2  $(c_i - \alpha_i) \xi_i = 0, \quad i = 1, \dots, n$
- 3  $y^T \alpha = 0,$
- 4  $-Q\alpha + by + s - \xi = -e,$
- 5  $0 \leq \alpha \leq c,$
- 6  $s \geq 0, \xi \geq 0.$

Let us introduce some notation. A primal-dual solution  $(\alpha, b, s, \xi)$  is called *dual basic feasible* if it satisfies condition **1-5** of the KKT system, but may violate condition **6**. For a given dual basic feasible solution,  $(\alpha, b, s, \xi)$ , we partition the index set  $I = \{1, \dots, n\}$  into three sets  $I_0, I_c$  and  $I_s$  in the following way:  $\forall i \in I_0 s_i \geq 0$  and  $\alpha_i = 0, \forall i \in I_c \xi_i \geq 0$  and  $\alpha_i = c_i$  and  $\forall i \in I_s s_i = \xi_i = 0$  and  $0 < \alpha_i < c_i$ . It is easy to see that  $I_0 \cup I_c \cup I_s = I$  and  $I_0 \cap I_c = I_c \cap I_s = I_0 \cap I_s = \emptyset$ . We will refer to  $I_s$  as the *primal active set* and to  $I_0 \cup I_c$  as the *dual active set*. Let  $n_s = |I_s|, n_0 = |I_0|$  and  $n_c = |I_c|$ ,

Based on the partition  $(I_0, I_c, I_s)$  we define  $Q_{ss}$  ( $Q_{cs}$   $Q_{sc}$   $Q_{cc}$ ,  $Q_{0s}$ ,  $Q_{00}$ ) as the submatrix of  $Q$  whose columns are the columns of  $Q$  indexed by the set  $I_s$  ( $I_c, I_s, I_c, I_0, I_0$ ) and whose rows are the rows of  $Q$  indexed by  $I_s$  ( $I_s, I_c, I_c, I_s, I_0$ ). We also define  $y_s$  ( $y_c, y_0$ ) and  $\alpha_s$  ( $\alpha_c, \alpha_0$ ) and the subvectors of  $y$  and  $\alpha$  whose entries are indexed by  $I_s$  ( $I_c, I_0$ ).  $c_c$  is the part of vector  $c$  indexed by  $I_c$  and by  $e$  we denote a vector of all ones whose size is determined by the context.

To initiate the algorithm we assume that we have a dual basic feasible solution  $\alpha^0, b, s^0, \xi^0$  and the corresponding partition  $(I_0^0, I_c^0, I_s^0)$ . For example setting  $\alpha^0 = 0$  and  $I_0 = \{1, \dots, n\}$  produces a starting point for the algorithm.

We know that  $\forall i \in I_0 \alpha_i = 0$  and  $\forall i \in I_c \alpha_i = c_i$ . Then if we fix the variables in the dual active set then our dual problem reduces to

$$\begin{aligned} \min_{\alpha_s} \quad & \frac{1}{2} \alpha_s^T Q_{ss} \alpha_s + c_c^T Q_{cs} \alpha_s - e^T \alpha_s \\ \text{s.t.} \quad & y_s^T \alpha_s = -y_c^T c_c, \\ & 0 \leq \alpha_s \leq c. \end{aligned}$$

The outline of the algorithm is the following:

**Step 0** Given  $\alpha^0, \beta^0, s^0, \xi^0$  find initial  $I_s, I_0$  and  $I_c$ .

**Step 1** If  $I_s = \emptyset$ , go to Step 2, otherwise:

(i) Solve

$$\begin{aligned} \min_{\alpha_s} \quad & \frac{1}{2} \alpha_s^T Q_{ss} \alpha_s + c^T Q_{cs} \alpha_s - e^T \alpha_s \\ \text{s.t.} \quad & y_s^T \alpha_s = -y_c^T c_c. \end{aligned} \tag{4}$$

If a finite solution,  $\alpha_s^*$ , exists, then set  $d = \alpha_s^* - \alpha_s$ , otherwise find  $d$  - an infinite descent direction.

- (ii) From the current iterate make a step along direction  $d$  until for some  $i \in I_s$   $\alpha_i = 0$  or  $\alpha_i = c_i$  or until solution is reached.  $\alpha_s^{k+1}$  is the new point.
- (iii) If for some  $i \in I_s$ ,  $\alpha_i^{k+1} = 0$ ,  
then update  $I_s := I_s \setminus \{i\}$ ,  $I_0 := I_0 \cup \{i\}$ ,  $k := k + 1$  and go to the beginning of **Step 1**.
- (iv) If for some  $i \in I_s$ ,  $\alpha_i^{k+1} = c_i$ ,  
then update  $I_s := I_s \setminus \{i\}$ ,  $I_c := I_c \cup \{i\}$ ,  $k := k + 1$  and go to the beginning of **Step 1**.
- (v) If the optimum is reached in step (ii); that is  $\alpha_s^{k+1} = \alpha_s^*$ , proceed to **Step 2**.

**Step 2** Partition  $I_0$  into  $I'_0$  and  $I''_0$  and partition  $I_c$  into  $I'_c$  and  $I''_c$

- (i) Compute  $s'_0$ , the subvector of  $s$  indexed by  $I'_0$ :

$$s'_0 = -Q'_{0s}\alpha_s^{k+1} - y'_0\beta + 1 - Q'_{0c}c_c$$

and  $\xi'_c$ , the subvector of  $\xi$  indexed by  $I'_c$ :

$$\xi'_c = Q'_{cs}\alpha_s^{k+1} + y'_c\beta - 1 + Q'_{cc}c_c,$$

where  $Q'_{0s}$  and  $Q'_{0c}$  ( $Q'_{cs}$  and  $Q'_{cc}$ ) are the submatrices of  $Q_{0s}$  and  $Q_{0c}$ , respectively, ( $Q_{cs}$  and  $Q_{cc}$ , respectively) with rows index by  $I'_0$  ( $I'_c$ ).

- (ii) Find  $i_0 = \operatorname{argmin}_i \{s_i : i \in I'_0\}$ .  
Find  $i_c = \operatorname{argmin}_i \{\xi_i : i \in I'_c\}$ .
- (iii) If  $s_{i_0} \geq 0$  and  $\xi_{i_c} \geq 0$  then if  $I'_0 \neq I_0$  or  $I'_c \neq I_c$  then let  $I'_0 := I_0$  and  $I'_c := I_c$  and go to Step 2(i). Else, the current solution is optimal, **Exit**.  
If  $s_{i_0} \leq \xi_{i_c}$ , then  $I_s := I_s \cup \{i_0\}$  and  $I_0 := I_0 \setminus \{i_0\}$ .  
Else,  $I_s := I_s \cup \{i_c\}$  and  $I_c := I_c \setminus \{i_c\}$ .  
 $k := k + 1$ , go to **Step 1**.

We will now discuss in details the implementation of the steps of the algorithm.

## 2.1 Solving the Quadratic Subproblem

When matrix  $Q_{ss}$  is strictly positive definite then problem (4) has a unique finite solution. This solution satisfies the KKT conditions:

$$\begin{aligned} -Q_{ss}\alpha_s + y_s^T\beta &= -e^T + c_c^T Q_{cs}\alpha_s \\ y_s^T\alpha_s &= -c_c^T y_c, \end{aligned}$$

or, in matrix form,

$$\begin{bmatrix} -Q_{ss} & y_s \\ y_s^T & 0 \end{bmatrix} \begin{pmatrix} \alpha_s \\ \beta \end{pmatrix} = \begin{pmatrix} -e + Q_{sc}c_c \\ -c_c^T y_c \end{pmatrix}. \quad (5)$$

Since we are considering the case when  $Q_{ss}$  is nonsingular, we can find  $\beta$  by taking the Schur complement of the above system

$$(y_s^T Q_{ss}^{-1} y_s)\beta = y_s^T Q_{ss}^{-1} (-e + Q_{sc}c_c) - c_c^T y_c.$$

Consider the Cholesky factorization  $Q_{ss} = L_s L_s^T$  and denote  $L_s^{-1} y_s$  by  $r_1$  and  $L_s^{-1}(-e + c_c^T Q_{cs})$  by  $r_2$ . Then the solution to (5) is

$$\beta = \frac{r_1^T r_2 - c_c^T y_c}{r_1^T r_1}, \quad \alpha_s = L_s^{-T} (r_1 \beta - r_2).$$

It is often the case, however, that  $Q_{ss}$  is not strictly positive definite. This can even occur when an RBF kernel (which is strictly positive definite for distinct data points) is used, if the set  $I_s$  contains indices of two identical data points with different labels.

If, due to singularity of  $Q_{ss}$ , system (5) is underdetermined, this means that problem (4) has an unbounded solution. In this case Step 1(i) should produce an infinite descent direction for (4). A direction  $d$  is an infinite direction if it satisfies  $Q_{ss}d = 0$  and  $y_s^T d = 0$ . Depending on the sign of  $(-e + c_c^T Q_{cs})^T d$  either  $d$  or  $-d$  is chosen as the infinite *descent* direction. Variable  $\beta$  remains unchanged in this case. We use the approach for positive semidefinite QP problems described in Frangioni (1996) and Kiwiel (1989).

We consider several cases.

### Case 1

Let  $Q_{ss}$  have only one zero eigenvalue. Then, subject to permutation and without loss of generality, its Cholesky factorization can be written as

$$Q_{ss} = \begin{bmatrix} L_s & 0 \\ l_s^T & 0 \end{bmatrix} \begin{bmatrix} L_s^T & l_s \\ 0 & 0 \end{bmatrix},$$

where  $L_s \in \mathbf{R}^{(n_s-1) \times (n_s-1)}$  and  $l_s \in \mathbf{R}^{n_s-1}$ . Then system (5) can be written as

$$\begin{bmatrix} -L_s L_s^T & -L_s l_s & y_{1:n_s-1} \\ -l_s^T L_s & -l_s^T l_s & y_{n_s} \\ y_{1:n_s-1} & y_{n_s} & 0 \end{bmatrix} \begin{pmatrix} \alpha_{1:n_s-1} \\ \alpha_{n_s} \\ \beta \end{pmatrix} = \begin{pmatrix} (-e + Q_{sc} c_c)_{1:n_s-1} \\ (-e + Q_{sc} c_c)_{n_s} \\ -c_c^T y_c \end{pmatrix},$$

where, following Matlab notation,  $y_{1:n_s-1}$  ( $\alpha_{1:n_s-1}$ ,  $(-e + Q_{sc} c_c)_{1:n_s-1}$ ) denote the first  $n_s - 1$  elements of vector  $y$  ( $\alpha$ ,  $(-e + c_c^T Q_{cs} e)$ ) and  $y_{n_s}$  ( $\alpha_{n_s}$ ,  $(-e + Q_{sc} c_c)_{n_s}$ ) denotes the last component of this vector.

Let  $r_1 = L_s^{-1} y_{1:n_s-1}$  and  $r_2 = L_s^{-1} (-e + Q_{sc} c_c)_{1:n_s-1}$ . By expressing  $\alpha_{1:n_s-1}$  in the above system through  $\alpha_{n_s}$  and  $\beta$ , and by consecutively eliminating  $\alpha_{n_s}$  we obtain

$$(-l_s^T r_1 + y_{n_s}) \beta = (-e + Q_{sc} c_c)_{n_s} - l_s^T r_2.$$

We now have two cases.

(a) If  $l_s^T r_1 \neq y_{n_s}$  then system (5) still has a unique solution

$$\begin{aligned} \beta &= \frac{(-e + Q_{sc} c_c)_{n_s} - l_s^T r_2}{-l_s^T r_1 + y_{n_s}}, \\ \alpha_{n_s} &= \frac{c_c^T y_c + r_1^T r_2 - \beta r_1^T r_1}{-r_1^T l_s + y_{n_s}} \\ \alpha_s &= L_s^{-T} (-l_s \alpha_{n_s} + r_1 \beta - r_2). \end{aligned}$$

- (b) If  $l_s^T r_1 + y_{n_s} = 0$  then system (5) is singular, hence we are looking for an infinite direction  $d$ .  $d_s = ((L_s^{-1} l_s)^T, -1)^T$  is such a direction. It can be easily shown that  $Q_{ss} d_s = 0$  from the form of the factorization of  $Q_{ss}$ , and it can be easily shown that  $y_s^T d_s = 0$  from the fact that  $l_s^T r_1 + y_{n_s} = 0$ .

**Case 2**

Let us now consider the case when  $Q_{ss}$  has exactly two zero eigenvalues. Then, again w.l.o.g., we can write its Cholesky factorization as

$$Q_{ss} = \begin{bmatrix} L_s & 0 & 0 \\ l_{s_1}^T & 0 & 0 \\ l_{s_2}^T & 0 & 0 \end{bmatrix} \begin{bmatrix} L_s^T & l_{s_1} & l_{s_2} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

where  $L_s \in \mathbf{R}^{n_s-2 \times n_s-2}$ ,  $l_{s_1}, l_{s_2} \in \mathbf{R}^{n_s-2}$ . The system (5) is always underdetermined in this case, hence an infinite direction always exists. There are, again, two possible cases.

- (a) If  $l_{s_2}^T r_1 \neq y_{n_s}$  then the following direction

$$d = (L_s^{-T} (l_{s_1} - \rho l_{s_2}), -1, \rho), \text{ where } \rho = \frac{y_{n_s-1} - l_{s_1}^T r_1}{y_{n_s} - l_{s_2}^T r_1}$$

is an infinite direction.  $Q_{ss} d = 0$  follows from the form of the Cholesky factorization and  $y^T d = 0$  is also easily shown by substitution.

- (b) If  $l_{s_2}^T r_1 = y_{n_s}$  then

$$d = (L_s^{-T} l_{s_2}, 0, -1)$$

is an infinite direction. This case can be shown similarly to Case 1(b).

**Case 3**

Finally, let us consider the case when  $Q_{ss}$  has more than two zero eigenvalues. First, we observe that this case can only happen in the early stage of the algorithm. Whenever  $Q_{ss}$  has more than one zero eigenvalue, then system (5) is underdetermined and an infinite direction is found during Step 1(i). Hence, during Step 1(ii) a boundary is always encountered. This means that the set  $I_s$  gets reduced by one element and the number of zero eigenvalues of  $Q_{ss}$  may only decrease or remain the same. Step 1 repeats until  $Q_{ss}$  has at most one zero eigenvalue. Hence, the only way that  $Q_{ss}$  may have more than two zero eigenvalues is if a starting solution with such  $Q_{ss}$  matrix is given to the algorithm. Such case arises when a warm start is used to initiate the algorithm, as described in Subsection 4.3, therefore, we consider this case here. Let  $k > 2$  be the number of zero eigenvalues of  $Q_{ss}$ ; as before we write, w.l.o.g., the factorization of  $Q_{ss}$ :

$$Q_{ss} = \begin{bmatrix} L_s & 0 & 0 & 0 \\ l_{s_1}^T & 0 & 0 & 0 \\ l_{s_2}^T & 0 & 0 & 0 \\ H_s & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} L_s^T & l_{s_1} & l_{s_2} & H_s^T \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$



where  $L_s \in \mathbf{R}^{n_s-k \times n_s-k}$ ,  $l_{s1}, l_{s2} \in \mathbf{R}^{n_s-k}$  and  $H_s \in \mathbf{R}^{n_s-k \times k-2}$ . We generate the infinite direction for the first  $n_s - k + 2$  variables exactly as it is done in Case 2 and we do not change the last  $k - 2$  variables. During each application of Case 3 of Step 1 we reduce  $I_s$  by one elements until  $Q_{ss}$  has at most 1 nonzero eigenvalue, and Case 3 does not arise again for that problem.

## 2.2 Rank-one Updates to $Q_{ss}$

On each iteration of the algorithm the set  $I_s$  can decrease by one element only and/or increase by one element only. Hence, from each iteration to the next,  $Q_{ss}$  changes by an addition and/or a deletion of one row and column. Instead of recomputing the Cholesky factorization each time, which would require  $O(n_s^3)$  operations, it is more efficient to keep the Cholesky factorization of  $Q_{ss}$  and update it accordingly when a row and a column are added to or deleted from  $Q_{ss}$ . Each such update requires only  $O(n_s^2)$  operations. These updates can be found in Golub and Van Loan (1996), but we present them here for completeness.

**Increasing  $I_s$ .** Assume first that  $Q_{ss}$  is nonsingular and  $Q_{ss} = L_s L_s^T$  is its Cholesky factorization. Let  $q_s \in \mathbf{R}^{n_s+1}$  be the new row (column) that is added to  $Q_{ss}$ . Aside from possible numerical issues, which we discuss later,  $q_s$  can be added as the last row and column of  $Q_{ss}$ . Then the Cholesky factorization of the new matrix is

$$\begin{bmatrix} L_s & L_s^{-1}(q_s)_{1:n_s} \\ 0 & (q_s)_{n_s+1}^2 - (q_s)_{1:n_s}^T L_s^{-T} L_s^{-1} (q_s)_{1:n_s} \end{bmatrix},$$

where  $(q_s)_{1:n_s}$  are the first  $n_s$  components of the vector  $q_s$  and  $(q_s)_{n_s+1}$  is its last component. It is easy to see that obtaining the new factorization requires  $O(n_s^2)$  operations.

If  $Q_{ss}$  is singular, then from the discussion in Case 3 of the previous subsection, it can only have one nonzero eigenvalue, since  $I_s$  is increased and hence Step 2 was performed. In this case we permute the rows and columns of  $Q_{ss}$  so that the dependent column and row are at the end of  $Q_{ss}$  and inserted column and row are placed in the one before last positions. The the last two rows of Cholesky factorization may need to be updated in a similar manner to above, however the total work is still  $O(n_s^2)$ . In case when  $Q_{ss}$  is nonsingular, but nearly so, it is sometimes important for numerical stability to use pivoting during its Cholesky factorization procedure (Fine and Scheinberg, 2001). In such a case refactorization of several rows of  $L_s$  might be required even if only one row and column are added to  $Q_{ss}$ . However, we did not encounter such situations in our computational experiments.

**Decreasing  $I_s$ .** When  $I_s$  is decreased by one element, then a row and a column are removed from  $Q_{ss}$  which corresponds to removing a row from the Cholesky factor  $L_s$ . If, say,  $k$ -th row was removed from  $L_s$  then it is no longer lower triangular. In fact it is nearly lower triangular, except for the elements in positions  $(j, j + 1)$  for  $j = k + 1, \dots, n_s - 1$ . To zero out these elements we apply Givens rotations (Golub and Van Loan, 1996) to the new matrix  $L_s$ ; in other words we multiply  $L_s$  on the right by orthogonal matrices of the form

$$\begin{bmatrix} 1 & \cdots & 0 & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & c & -s & 0 \\ 0 & 0 & s & c & 0 \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix}.$$

Each such matrix multiplication takes  $O(n_s)$  operations and zeros out one off-diagonal elements, hence we need  $O(n_s - k)$  such multiplications, which results in the total work of  $O(n_s^2)$  to update the Cholesky factorization of  $Q_{ss}$  when an elements is removed from  $I_s$ .

**Remark 1** *In Frangioni (1996) there are efficient updates for the vectors  $r_1$  and  $r_2$ , that we introduced in Subsection 2.1. These vectors are results of backsolves with the Cholesky factors of  $Q_{ss}$  and given right hand side vectors. In the case of Frangioni (1996) the right hand side vectors remain the same throughout the algorithm and only the Cholesky factors change. In our case this is true only for  $r_1$  but not for  $r_2$ , which changes each time the set  $I_c$  changes. These updates can also improve the efficiency of the algorithm when these backsolves have a noticeable contribution to the overall workload of the algorithms. Since this does not occur very frequently we do not get into further details in this paper.*

**Remark 2** *If  $n_s$  is very large and is comparable to  $n$  then even storing and updating the Cholesky factors of  $Q_{ss}$  become too expensive compared to solving the entire problem. Our method is not practical on such problems. However, it is questionable whether such problems should ever be solved, since the resulting classifiers is most likely overfitting the data and its generalization properties are expected to be very poor.*<sup>3</sup>

**Updating  $I_c$**  Finally we discuss a trivial but useful updates to  $Q_{sc}c_c$ ,  $Q_{0c}c_c$  and  $Q_{cc}c_c$  when the set  $I_c$  is either increased or decreased by one element. We maintain vector  $Q_{cc}c_c$  throughout the algorithm, when index  $i$  is added to  $I_c$  then a  $c_i$  multiple of the  $i$ -th column of  $Q$  is added to  $Q_{cc}c_c$ . If index  $i$  is removed from  $I_c$ , then such a vector is subtracted from  $Q_{cc}c_c$ .

### 3. Comparison to SVM<sup>light</sup>

In this section we compare our implementation of the proposed algorithm, which we call SVM-QP, to SVM<sup>light</sup>. SVM-QP currently is implemented in Fortran 77, although a C++ version is under development. SVM-QP is an open source software and is available from the [www.coin-or.org](http://www.coin-or.org) website. We used a high-end IBM RS/6000 workstation in our experiments. We made the same amount of memory available to both methods. Just as in SVM<sup>light</sup> the sparsity of the examples is exploited by SVM-QP during the kernel evaluations. Unlike SMO (Platt, 1999) there is no special handling for the case of linear kernel.

We used the following data sets in our experiments:

- Letter-G: The Letter Image Recognition data set from the UCI Repository (Blake and Merz, 1998) - A large number of black-and-white character images were randomly distorted to produce a file of 20,000 unique stimuli. Each stimulus was converted into 16 primitive numerical attributes (statistical moments and edge counts) which were then scaled to fit into a range of integer values from 0 through 15. We examined performances on an arbitrary binary classification problem which was set to separate the letter "G" from all the other letters.

---

3. See for example Cristianini and Shawe-Taylor (2000), Theorem 4.25, for the generalization power of compression schemes, and the discussion right after and in Chapter 6.

- OCR: USPS (United States Postal Service) data set of hand written digits. This data set comprises 7291 training and 2007 test patterns, represented as 257 dimensional vectors with entries between 0 and 255. T0(T9) stand for the binary classification problem in which the target is the digit 0(9) vis. the all the other digits.
- Web and Adult<sup>4</sup> : We used the tasks that was compiled by Platt and available from the SMO home page<sup>5</sup>
  - Adult - The goal is to predict whether a household has an income greater than \$50000. After discretization of the continuous attributes, there are 123 binary features, with  $\approx 14$  non-zeros per example.
  - Web - A text classification problem with binary representation based on 300 keyword features. This representation is extremely sparse. On the average there are only  $\approx 12$  non-zero features per example.

For both problems we chose the test cases with half of the overall available example. We did so to enable to complete many computational tests in a reasonable amount of time. We also present a table with the results of comparing only the runtime of SVM-QP and SVM<sup>light</sup> on the full test sets for these two problems.

- Abalone: The Abalone data set from the UCI Repository (Blake and Merz, 1998). Since, we were not interested in evaluating generalization performances, we fed the training algorithm with increasing subsets up to the whole set (of size 4177). The gender encoding (male/female/infant) was mapped into  $\{(1,0,0),(0,1,0),(0,0,1)\}$ . Then data was scaled to lie in the  $[-1,1]$  interval.
- Spam: This is another data set from the UCI Repository. It was created by M. Hopkins, E. Reeber, G. Forman and J. Suermondt of Hewlett-Packard Labs. It contains 4601 examples of emails roughly 39% of which are classified as spam. There are 57 attributes for each example, most of which represent how frequently certain words or characters appear in the email.

For each data set we used a selection of kernels and parameters to demonstrate how the performance of the methods is affected by  $n_s$  - the number of support vectors at the margin, and  $n_c$  - the number of support vectors at the upper bound. For the same reason we use various values of  $C$ . We use RBF kernel with parameter  $\sigma$ . We also use the linear kernel for a Letter-G and Spam problems and polynomial kernel of degree 5 for the Abalone data set. In the tables of results we indicate the kernel and the value of  $C$  in the name of the test case. For instance **web\_100\_10** stands for the web data set with parameter  $\sigma = 100$  and  $C = 10$ . Name **letter\_lin\_100** stands for the Letter-G set with linear kernel and  $C = 100$ , finally **abalone\_p5\_100** stands for the Abalone set with polynomial kernel of degree 5 and  $C = 100$ .

We provide two columns of CPU times for SVM<sup>light</sup>. The first one, SVM<sup>light</sup>, contains the time of the runs with default accuracy  $10^{-3}$ . The second column, SVM <sub>$\epsilon$</sub> <sup>light</sup> contains the CPU time of the runs with the accuracy set to  $10^{-6}$  which is the accuracy of SVM-QP. Both algorithms apply the accuracy tolerance to the constraints  $-Q\alpha + by + s - \xi = -e$ . Specifically, SVM-QP applies a given tolerance  $\epsilon$  on **Step 2(iii)** of the algorithm (see Section 2) to determine if  $s_{i_0} \geq -\epsilon$  and  $\xi_{i_c} \geq \epsilon$ .

4. Original data set is from the UCI Repository (Blake and Merz, 1998).

5. <http://www.research.microsoft.com/jplatt/smo.html>

Name	$n$	$k$	$n_s$	$n_c$	SVM <sup>light</sup>	SVM <sub>e</sub> <sup>light</sup>	SVM-QP
web_100_100	24692	300	980	453	380	918	65
web_40_10	24692	300	1037	568	241	377	68
web_40_100	24692	300	1214	313	368	685	84
web_100_10	24692	300	679	835	203	358	40
letter_100_100	20000	16	241	39	19	26	3
letter_40_1	20000	16	250	266	6	7	5
letter_40_100	20000	16	346	8	11	16	4
letter_100_10	20000	16	193	146	10	15	4
letter_40_10	20000	16	320	57	8	10	4
letter_lin_100	20000	16	17	1056	1052	1190	35
ocr9_256_100	7291	256	378	0	13	13	5
ocr0_256_100	7291	256	309	0	8	9	4
abalone_4_100	4177	10	64	1863	135	198	5
abalone_p5_100	4177	10	304	1520	-	-	22
spam_300_100	4601	57	1417	181	90	-	64
spam_lin_100	4601	57	58	822	-	-	11
adult_100_1	16100	123	97	5996	153	154	81
adult_100_100	16100	123	871	4823	515	856	175
adult_200_1	16100	123	168	5785	159	152	85
adult_200_100	16100	123	483	5219	332	447	140
adult_50_10	16100	123	615	5143	207	243	120

Table 1: Performance comparison of SVM-QP and SVM<sup>light</sup>.

We chose CPU time (in seconds) as the most reasonable performance measure in our setting. The “-” in the table indicates the failure of SVM<sup>light</sup> on that problem.

Table 1 contains the results for the test problems that we examine in this paper. As we can see, SVM-QP is faster than even the lower accuracy SVM<sup>light</sup>, on all of the problems. It is faster by at least a factor of 2 on almost all of the problems and by a factor of 5 or more on a few problems.

In Table 2 we present the comparison of SVM-QP and SVM<sup>light</sup> on the full test sets for **adult** and **web**. These results were obtained by Hans Mittleman, at Arizona State University using a high-end Unix workstation. The “\*” in the last column of the last row indicates that SVM-QP ran out of memory, since it was trying to store an  $n_s \times n_s$  matrix, with  $n_s \approx 10000$ . The stopping tolerance was set to be  $10^{-6}$  for both codes, but it is interesting to note that the resulting support vector sets differed significantly. For example, the number of active support vectors for **web\_10\_100** reported by SVM-QP was 3446, while the same number reported by SVM<sup>light</sup> was 4025. This discrepancy is due to the fact that SVM<sup>light</sup> converges to the optimal active set asymptotically, while SVM-QP steps from one feasible active set to another in an “exact” manner, until the optimal is found.

#### 4. Implementation Issues

Now we will discuss some implementation choices.

Name	$n$	$k$	$n_s$	$n_c$	SVM $_c^{light}$	SVM-QP
web_1000_100	49749	300	297	1702	694	92
web_100_100	49749	300	1404	905	3581	174
web_10_100	49749	300	3446	527	1354	715
adult_1000_100	32561	123	143	11361	937	278
adult_100_100	32561	123	1317	9879	5685	460
adult_10_100	32561	123	9959	3200	14466	*

Table 2: Performance comparison of SVM-QP and SVM $^{light}$  on large data sets.

#### 4.1 Selecting the Incoming Element of $I_s$

In this subsection we discuss the implementation of Step 2(i). First of all we note that the computational cost of Step 2(i) depends on whether the kernel values are available in the memory or have to be computed. We need  $O(|I_s|(|I'_0| + |I'_c|))$  kernel values at each iteration when Step 2 is invoked. Specifically we need the elements of matrix  $Q$  whose column indices are in  $I_s$  and whose row indices are in  $I'_c$  and  $I'_0$ .

We note that we *always* store the  $n_s \times n_s$  matrix  $Q_{ss}$ . This can be a problems when  $n_s$  is large. Our algorithm requires storage of the Cholesky factor of  $Q_{ss}$ , hence even if we do not store  $Q_{ss}$  itself, the storage requirement can be reduced at most by half. In our experiments the size of  $Q_{ss}$  and its Cholesky factor was reasonable. For extremely large problems a different implementation may be necessary which solves the linear system in Step 1 by an iterative solver.

To reduce the computational cost it is best to be able to store the entire  $Q_s$  matrix (that is the submatrix of  $Q$  whose column indices are in  $I_s$ ). In some cases this might be prohibitively expensive in terms of memory. In our experiments we were able to store  $Q_s$  in the space not exceeding 400MB. At the end of this subsection we will discuss the memory saving version of our code.

Let us assume for now that matrix  $Q_s$  is available. We will consider various ways of reducing the number of elements in  $I'_0$  and  $I'_c$  at each repetition of Step 2(i). One simple way to achieve this is to compute the elements of  $s'_0$  and  $\xi'_c$  until a negative element is encountered, hence, not looking for the maximum violation, but for any violation. This may reduce the per-iteration time, but greatly increases the number of iterations, as has been shown by the extensive practice of the Simplex method in linear programming (Vanderbei, 2001). We will demonstrate this in the section on the incremental mode, since the incremental mode lacks the ability to “look ahead” and select the maximum violated constraint. We conclude that it is important to select the most negative or nearly the most negative element of  $s'_0$  and  $\xi'_c$  during Step 2.

We use the following concepts, common in LP literature. The primal slack and surplus variables  $s_i$  and  $\xi_i$  are the reduced costs of the associated dual variable  $\alpha_i$ , whose value is currently at a bound. Computing the values of the reduced costs (recall that for each  $i$  only one of the reduced costs is not equal to zero) is called *pricing* of the appropriate dual variable. Hence it is important to price all variables with indices in  $I'_0$  and  $I'_c$  and maintain these sets in such a way that they contain indices of substantially negative reduced costs.

The efficiency of the large-scale SVM training relies heavily on the fact that at the optimal solution the cardinality of  $I_s$  is often much smaller than the total number of data points  $n$ . Hence, the cardinalities of  $I_0$  and, possibly,  $I_c$  are expected to be large in comparison to  $I_s$ . If in Step 2(i)  $I'_0$  and

$I_c'$  are large, while  $I_s$  is not very small, then the complexity of this step, which is  $O(|I_s|(|I_0'| + |I_c'|))$ , might become too high.

Let us assume for a moment that we know some of the indices that at optimality belong to  $I_c$  and  $I_0$ . Then we can place these indices in  $I_0''$  and  $I_c''$  at the beginning of each Step 2. This can result in substantial savings in the run time, since Step 2(i) requires  $O(|I_s|(|I_0'| + |I_c'|))$  operations and  $|I_0'| = |I_0| - |I_0''|$  and  $|I_c'| = |I_c| - |I_c''|$ . When all the reduced costs of variables whose indices are in  $I_0'$  and  $I_c'$  are nonnegative, then so are the reduced costs of variables whose indices are in  $I_0''$  and  $I_c''$ , due to our assumption about these two subsets.

Naturally, we usually do not know which indices will be in  $I_0$  and  $I_c$  at optimality, however, to reduce the workload at each iteration we try to *guess* which indices are the most likely ones to end up in  $I_0$  and  $I_c$  at optimality. We place such indices in  $I_0''$  and  $I_c''$  sets, respectively. If we guess well, then after all the reduced costs for  $I_0'$  and  $I_c'$  become nonnegative, hopefully, only a few reduced costs for  $I_0''$  and  $I_c''$  are negative. Here we see a trade-off: if we select  $I_0''$  and  $I_c''$  too small, then the computational saving is insignificant, and if we select  $I_0''$  and  $I_c''$  too large, then some of large negative reduced costs might be missed and the overall number of iterations might increase. Moreover, once all the dual variables with indices in  $I_0'$  and  $I_c'$  are priced, then we have to price all variables with indices in  $I_0''$  and  $I_c''$ , which are large. So it is important to choose  $I_0''$  and  $I_c''$  in such a way that pricing the variables in  $I_0''$  and  $I_c''$  does not occur too many times.

We will describe two possible strategies for maintaining sets  $I_0'$ ,  $I_c'$ ,  $I_0''$  and  $I_c''$ . One strategy is very simple and is called *shrinking* in SVM literature (Joachims, 1999). At each iteration an index is placed in  $I_0''$  or  $I_c''$  if its appropriate reduced cost remained nonnegative for a given number of consecutive iterations (say 100). According to this strategy the sets  $I_0'$  and  $I_c'$  are large during the earlier iterations and become gradually smaller during the course of the algorithm. This nicely correlates with the fact that the size of  $I_s$  is very small in the earlier iterations  $I_s$  gets gradually larger during the course of the algorithm. It is often the case that maximum of  $|I_s|(|I_0'| + |I_c'|)$  over all iteration is 3 or 4 times smaller than  $\max\{|I_s|\} \times \max\{(|I_0'| + |I_c'|)\}$ . At the end one still has to price all the dual variables for  $I_0''$  and  $I_c''$ , but only a few of such iterations are usually needed.

The second strategy is called *sprint* in Linear Programming literature and was introduced by Forrest (1989). Sprint (sometimes also called *sifting*) has been proven to be very effective in practice for problems that contain large number of inactive constraints (see Bixby et al., 1992). Following the sprint strategy we select a relatively small subset of dual variables with the smallest (including the most negative) reduced costs and we form  $I_0'$  and  $I_c'$  from the indices of those variables. Once the problems was solved for  $I_0'$  and  $I_c'$  the remaining constraints are priced again and the next relatively small sets of candidates are selected. Pricing all remaining variables and choosing the next small subset is called a *major iteration*. According to this strategy  $I_0'$  and  $I_c'$  are always kept small, but the sets  $I_0''$  and  $I_c''$  have to be considered regularly throughout the algorithm. As long as the ratio of major iterations to the number of “cheap” iterations is small, the implementation will be efficient.

Table 3 below shows that *sprint* outperforms *shrinking* in most cases, especially on larger, more difficult problems.

## 4.2 Memory Saving Version

We now discuss the memory saving version. SVM<sup>light</sup> has an elegant scheme, where the kernel values are stored in cache according to their most recent usage. The size of the cache is dictated by

Name	$n$	$k$	$n_s$	$n_c$	SVM-QP <sub>shr</sub>	SVM-QP
web_100_100	24692	300	980	453	537	65
web_40_10	24692	300	1037	568	281	68
web_40_100	24692	300	1214	313	416	84
web_100_10	24692	300	679	835	124	40
letter_100_100	20000	16	241	39	6	3
letter_40_1	20000	16	250	266	6	5
letter_40_100	20000	16	346	8	7	4
letter_100_10	20000	16	193	146	6	4
letter_40_10	20000	16	320	57	7	4
letter_lin_100	20000	16	17	1056	20	35
ocr9_256_100	7291	256	378	0	7	5
ocr0_256_100	7291	256	309	0	6	4
abalone_4_100	4177	10	64	1863	4	5
abalone_p5_100	4177	10	304	1520	31	22
spam_300_100	4601	57	1417	181	80	64
spam_lin_100	4601	57	58	822	15	11
adult_100_1	16100	123	97	5996	89	81
adult_100_100	16100	123	871	4823	253	175
adult_200_1	16100	123	168	5785	95	85
adult_200_100	16100	123	483	5219	107	140
adult_50_10	16100	123	615	5143	120	120

Table 3: Sprint vs. Shrinking.

Name	$n$	$k$	$n_s$	$n_c$	SVM-QP	SVM-QP <sub>mem</sub>	SVM <sub>mem</sub> <sup>light</sup>
web_100_100	24692	300	980	453	65	428	1097
web_40_10	24692	300	1037	568	68	447	553
web_40_100	24692	300	1214	313	84	524	1201
web_100_10	24692	300	679	835	40	230	348
letter_100_100	20000	16	241	39	3	14	26
letter_40_1	20000	16	250	266	5	17	9
letter_40_100	20000	16	346	8	4	17	15
letter_100_10	20000	16	193	146	4	14	14
letter_40_10	20000	16	320	57	4	20	11
letter_lin_100	20000	16	17	1056	35	72	1052
ocr9_256_100	7291	256	378	0	5	24	13
ocr0_256_100	7291	256	309	0	4	13	9
abalone_4_100	4177	10	64	1863	5	13	78
abalone_p5_100	4177	10	304	1520	22	44	-
adult_100_1	16100	123	97	5996	81	171	228
adult_100_100	16100	123	871	4823	175	624	1541
adult_200_1	16100	123	168	5785	85	174	253
adult_200_100	16100	123	483	5219	140	173	1360
adult_50_10	16100	123	615	5143	120	355	593

Table 4: Comparison for memory saving mode.

the user. In the experiments discussed above we allowed the size of the cache to be 500MB, which is at least as much memory as was used by SVM-QP.

We did not implement such sophisticated memory handling mechanism in our code. Luckily *sprint* provides a natural setting for a memory saving mode. Instead of storing the whole  $Q_s$  we only store the elements of  $Q$  whose columns are in  $I_s$  and whose rows are in  $I_s \cup I'_0 \cup I'_c$ . The size of  $I'_0 \cup I'_c$  can be regulated according to the available storage space. At each major iteration all the elements of  $Q_s$  whose row indices are in  $I''_0 \cup I''_c$  have to be recomputed. This can be a costly step. To further try to reduce the computational cost of that step, we apply *shrinking* to  $I''_0 \cup I''_c$ . That is, if during a few consecutive major iterations a certain reduced cost remained nonnegative then the appropriate variable is removed from  $I''_0 \cup I''_c$  and is ignored until the later stage of the algorithm. In Table 4 below we present our results. We chose the size of  $I'_0$  and  $I'_c$  to be 50 each, this way the total storage for the elements of  $Q_s$  (including  $Q_{ss}$ ) did not exceed 20MB. We compare our CPU time to that of SVM<sup>light</sup> with 20MB of cache limit. We also list the CPU times for the version of SVM-QP that stores the full  $Q_s$ , to demonstrate the trade-off between the CPU time and memory requirement.

### 4.3 Warm Start

One of the significant advantages of the active set methods over the interior point methods is that the former can benefit very well from warm starts. For instance, if some additional labeled training data become available, the old optimal solution is used as a starting point for the active set algorithm



and the new optimal solution is typically obtained within a few iterations. This will be explored in more detail in the subsection on the incremental mode of our algorithm.

Another situation where warm start arises, is when one wants to explore the path of optimal solutions for various values of penalty parameter  $C$ . In Hastie et al. (2004) the whole solution path is generated using an active set method similar to ours. There are some differences between the two methods, however. The method in Hastie et al. (2004) is a parametric active set method, which in practice is usually slower than a purely primal or dual active set method, such as ours. Also their method requires that at each iteration an optimal solution of a parametric problem is available, hence there does not seem to be any possibility to use *sprint* or *shrinking*. It remains to be seen whether a good implementation of the algorithm in Hastie et al. (2004) can match the performance of our algorithm.

Our algorithm is not suitable directly for generating the entire parametric path, but using the warm starts one can easily use it to generate solutions for a selection of the values of parameter  $C$ .

The warm starts can also be used when one wants to explore different values of kernel parameters, but the efficiency of such application needs a separate computational study.

Here we investigate the use of warm start to increase the efficiency of the algorithm itself. It has been noticed (see, for example, Fine and Scheinberg, 2001) that for many SVM problems the matrix  $Q$  has eigenvalues decaying to zero. It was suggested in Fine and Scheinberg (2001) to use a low rank approximation of  $Q$  and solve the approximate problem with an interior point method using product form Cholesky factorizations, which benefit from the low rank of  $Q$ . Such approximations, however, are not always very accurate. The idea we explore here is to use the solution of the approximate problem to warm start the active set method.

If  $k$  is the rank of the approximation of  $Q$ , then per iteration complexity of the IPM is  $O(nk^2)$ . There is a trade-off in choosing the right value for  $k$ : if  $k$  is chosen to be too large, then the IPM will not be efficient and if  $k$  is too small then the solution produced by the IPM is too far from the optimal solution of the true problem. We chose  $k = 50$ , which is reasonably small to make the IPM part fast and sufficiently large to hope for a good warm start. The results in Table 5 are not as dramatic as one might hope. Often the active set method itself is so fast that it outperforms the IPM even for  $k = 50$ , for instance on **letter\_x\_x** problems. In other cases the approximation does not produce a good enough warm start. There are also cases where  $Q$  itself has very low rank and, hence, the problem can be solved to optimality just by the IPM; see **letter\_lin\_100**, for instance. There are examples however, where the combined method achieves better timing results than either method, when used separately. This seems to happen for the problems with relatively large  $I_c$  sets, such as the **adult\_x\_x** problems. We have to note that we are using a rather crude implementation of the IPM for SVM. One might achieve better results with a more efficient implementation of an IPM.

#### 4.4 Incremental Mode

Incremental mode is used when the training data is available one point (or a few points) at a time. Our algorithm applies naturally and almost without change to the incremental mode. Whenever more data points become available, their indices get placed in set  $I_0$ , then Step 2(i) is applied to price the corresponding variables, and if a negative reduced cost is found, then the algorithm proceeds in the usual manner. The only difference with the batch case (when all data is available at once) is that the pricing in Step 2(i) cannot be applied to the data that is not available yet. Hence, the constraints

Name	$n$	$k$	$n_s$	$n_c$	SVM-QP <sub><math>p</math></sub>	SVM-QP
web_100_100	24692	300	980	453	113	65
web_40_10	24692	300	1037	568	112	68
web_40_100	24692	300	1214	313	142	84
web_100_10	24692	300	679	835	82	40
letter_100_100	20000	16	241	39	36	3
letter_40_1	20000	16	250	266	38	5
letter_40_100	20000	16	346	8	49	4
letter_100_10	20000	16	193	146	33	4
letter_40_10	20000	16	320	57	44	4
letter_lin_100	20000	16	17	1056	7	35
ocr9_256_100	7291	256	378	0	21	5
ocr0_256_100	7291	256	309	0	15	4
abalone_4_100	4177	10	64	1863	5	5
abalone_p5_100	4177	10	304	1520	10	22
spam_300_100	4601	58	1417	181	40	64
spam_lin_100	4601	58	58	822	9	11
adult_100_1	16100	123	97	5996	66	81
adult_100_100	16100	123	871	4823	125	175
adult_200_1	16100	123	168	5785	68	85
adult_200_100	16100	123	483	5219	92	140
adult_50_10	16100	123	615	5143	95	120

Table 5: Warm starting SVM-QP by an IPM.

Name	$n$	$k$	$n_s$	$n_c$	SVM-QP	SVM-QP <sub>inc</sub>
web_100_100	24692	300	980	453	65	1388
web_40_10	24692	300	1037	568	68	1017
web_40_100	24692	300	1214	313	84	1190
web_100_10	24692	300	679	835	40	1101
letter_100_100	20000	16	241	39	3	24
letter_40_1	20000	16	250	266	5	37
letter_40_100	20000	16	346	8	4	34
letter_100_10	20000	16	193	146	4	24
letter_40_10	20000	16	320	57	4	39
letter_lin_100	20000	16	17	1056	35	43
ocr9_256_100	7291	256	378	0	5	27
ocr0_256_100	7291	256	309	0	4	16
abalone_4_100	4177	10	64	1863	5	13
abalone_p5_100	4177	10	304	1520	22	137
spam_300_100	4601	58	1417	181	64	414
spam_lin_100	4601	58	58	822	11	647
adult_100_1	16100	123	97	5996	81	372
adult_100_100	16100	123	871	4823	175	5882
adult_200_1	16100	123	168	5785	85	330
adult_200_100	16100	123	483	5219	140	1819
adult_50_10	16100	123	615	5143	120	2274

Table 6: Incremental mode.

with sufficiently negative reduced costs are not included, until their data points are added to the problem. As we show in Table 6, this results in a dramatic increase of CPU time.

Notice, that the sifting does not make sense in the incremental mode, since it selects the sets  $I_0'$  and  $I_c'$  based on the entire data set. However, shrinking can be easily applied, since its selection of  $I_0'$  and  $I_c'$  is only based on the past behavior of each individual constraint.

## 5. Concluding Remarks

Traditional active set methods for convex QPs were considered impractical for large-scale SVM problems. However, they have theoretical appeal for many reasons. In this paper we studied in details an active set method SVM and show that an efficient implementation can outperform other state-of-the-art SVM software.

Future direction of this work lies in a more comprehensive theoretical analysis of the behavior and complexity of the method for SVM problems.

## Acknowledgments

The author is grateful to Alexandre Belloni for pointing out the paper (Frangioni, 1996) for handling the singular reduced QP system as well as for many fruitful discussions. The author is also grateful to John Forrest for suggesting the *sprint* technique.

## References

- J. Balcazar, Y. Dai, and O Watanabe. Provably fast training algorithms for support vector machines. In *Proc. 1st IEEE International Conference on Data Mining*, pages 43–50. IEEE, 2001.
- R. E. Bixby, J. W. Gregory, I. J. Lustig, R. E. Marsten, and D. F. Shanno. Very large-scale linear programming: A case study in combining interior point and simplex methods. *Operations Research*, 40(5):885–897, 1992.
- C. L. Blake and C. J Merz. UCI repository of machine learning databases, 1998. URL <http://www.ics.uci.edu/~mllearn/MLRepository.html>.
- B. Boser, I. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In D. Haussler, editor, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152. ACM Press, 1992.
- G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 409–415. MIT Press, 2001.
- N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge University Press, 2000.
- M. C. Ferris and T. S. Munson. Interior point methods for massive support vector machines. Technical Report 00-05, Computer Sciences Department, University of Wisconsin, Madison, WI, 2000.
- S. Fine and K. Scheinberg. Efficient svm training using low-rank kernel representations. *Journal of Machine Learning Research*, 2:243–264, 2001.
- R. Fletcher. A general quadratic programming algorithm. *Journal of the Institute of Mathematics and Its Applications*, pages 76–91, 1971.
- J. J. Forrest. Mathematical programming with a library of optimization subroutines. Presented at the ORSA/TIMS Joint National Meeting, 1989.
- A. Frangioni. Solving semidefinite quadratic problems within nonsmooth optimization algorithms. *Computers in Operations Research*, 23(11):1099–1118, 1996.
- D. Goldfarb. Extension of newton’s method and simplex method for solving quadratic problems. In F. A. Lootsma, editor, *Numerical Methods for Nonlinear Optimization*, pages 239–254. Academic Press, London, 1972.
- D. Goldfarb and A. Idnani. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical Programming*, 27:1–33, 1983.

- G. H. Golub and Ch. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, 3rd edition, 1996.
- T. Hastie, S. Rosset, R. Tibshirani, and J. Zhu. The entire regularization path. *Journal of Machine Learning Research*, 5:1391–1415, 2004.
- T. Joachims. Making large-scale support vector machine learning practical. In B. Schölkopf, C. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods*, chapter 12, pages 169–184. MIT Press, 1999.
- L. Kaufman. Solving the quadratic programming problem arising in support vector classification. In B. Schölkopf, C. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods Support Vector Learning*, pages 147–168. MIT Press, 1998.
- K. C. Kiwiel. A dual method for certain positive semidefinite quadratic programming problems. *SIAM Journal on Scientific and Statistical Computing*, 10:175–186, 1989.
- J Nocedal and S. Wright. *Numerical Optimization*. Springer-Verlag, 1999.
- E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. In *Proceedings of the IEEE Neural Networks for Signal Processing VII Workshop*, pages 276–285. IEEE, 1997.
- J. C. Platt. Fast training support vector machines using sequential minimal optimization. In B. Schölkopf, C. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods*, chapter 12, pages 185–208. MIT Press, 1999.
- R. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, New York, NY, 2nd edition, 2001.