Journal of Machine Learning Research 7 (2006) 1467–1492 Submitted 10/05; Revised 2/06; Published 7/06

Parallel Software for Training Large Scale Support Vector Machines on Multiprocessor Systems*

Luca Zanni ZANNI.LUCA@UNIMO.IT
Thomas Serafini SERAFINI.THOMAS@UNIMO.IT
Department of Mathematics
University of Modena and Reggio Emilia
via Campi 213/B, Modena, 41100, Italy

Gaetano Zanghirati G.ZANGHIRATI@UNIFE.IT
Department of Mathematics
University of Ferrara
Scientific-Technological Campus, Block B
via Saragat 1, Ferrara, 44100, Italy

Editors: Kristin P. Bennett and Emilio Parrado-Hernández

Abstract

Parallel software for solving the quadratic program arising in training *support vector machines* for classification problems is introduced. The software implements an iterative decomposition technique and exploits both the storage and the computing resources available on multiprocessor systems, by distributing the heaviest computational tasks of each decomposition iteration. Based on a wide range of recent theoretical advances, relevant decomposition issues, such as the quadratic subproblem solution, the gradient updating, the working set selection, are systematically described and their careful combination to get an effective parallel tool is discussed. A comparison with state-of-the-art packages on benchmark problems demonstrates the good accuracy and the remarkable time saving achieved by the proposed software. Furthermore, challenging experiments on real-world data sets with millions training samples highlight how the software makes large scale standard nonlinear support vector machines effectively tractable on common multiprocessor systems. This feature is not shown by any of the available codes.

Keywords: support vector machines, large scale quadratic programs, decomposition techniques, gradient projection methods, parallel computation

1. Introduction

Training support vector machines (SVM) for binary classification requires to solve the following convex quadratic programming (QP) problem (Vapnik, 1998; Cristianini and Shawe-Taylor, 2000)

$$\begin{array}{rl} \min & \mathcal{F}(\alpha) = \frac{1}{2}\alpha^T G\alpha - \sum_{i=1}^{n}\alpha_i \\ \text{sub. to} & \sum_{i=1}^{n} y_i\alpha_i = 0, \\ & 0 \le \alpha_i \le C, \qquad i = 1,\ldots,n, \end{array} \tag{1}$$

*. This work was supported by the Italian Education, University and Research Ministry via the FIRB Projects *"Statistical Learning: Theory, Algorithms and Applications"* (grant RBAU01877P) and *"Parallel Algorithms and Numerical Nonlinear Optimization"* (grant RBAU01JYPN).

©2006 Luca Zanni, Thomas Serafini and Gaetano Zanghirati.

whose size n is equal to the number of examples in the given training set

$$D = \left\{(x_i, y_i),\ i = 1,\ldots,n, \quad x_i \in \mathbb{R}^M,\ y_i \in \{-1,1\}\right\},$$

and the entries of G are defined by

$$G_{ij} = y_i y_j K(x_i, x_j), \quad i,j = 1,2,\ldots,n,$$

where $K : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}$ denotes the kernel function. The main features of this problem are the density of the quadratic form and the special feasible region defined by box constraints and a single linear equality constraint. In many practical SVM applications, standard QP solvers based on the explicit storage of the Hessian matrix G may be very inefficient or, in the case of large data sets, even not applicable due to excessive memory requirements. For these reasons in recent years a lot of attention has been dedicated to this problem and several *ad hoc* strategies have been developed, which are able to solve the problem with G out of memory. Among these strategies, the decomposition techniques have been the most investigated approaches and have given rise to the state-of-the-art software for the SVM QP problem. The idea behind the decomposition techniques consists in splitting the problem into a sequence of smaller QP subproblems, sized $n_{\rm sp}$ say, that can be stored in the available memory and efficiently solved (Boser et al., 1992; Chang and Lin, 2001; Collobert and Bengio, 2001; Joachims, 1998; Osuna et al., 1997; Platt, 1998). At each decomposition step, a subset of the variables, usually called *working set*, is optimized through the solution of the subproblem in order to obtain a progress towards the minimum of the objective function $\mathcal{F}(\alpha)$. Effective implementations of this simple idea involve important theoretical and practical issues. From the theoretical point of view, the policy for updating the working set plays a crucial role since it can guarantee the strict decrease of the objective function at each step (Hush and Scovel, 2003). The most used working set selections rely on the violations of the Karush-Kuhn-Tucker (KKT) first order optimality conditions. In case of working sets of minimal size, that is sized 2, a proper selection via the *maximal-violating pair* principle (or related criteria) is sufficient to ensure asymptotic convergence of the decomposition scheme (Lin, 2002; Chen et al., 2005). For larger working sets, convergence proofs are available under a further condition which ensures that the distance between two successive approximations tends to zero (Lin, 2001a; Palagi and Sciandrone, 2005). Furthermore, based on these working set selections and further assumptions, the linear convergence rate can be also proved (Lin, 2001b). For the practical efficiency of a decomposition technique, the fast convergence and the low computational cost per iteration seem the most important features. Unfortunately, these goals are conflicting since the strategies to improve the convergence rate (as the use of large working sets or the selections based on second order information) usually increase the cost per iteration. Examples of good trade-offs between the two goals are given by the most widely used decomposition packages: LIBSVM (Chang and Lin, 2001) and SVM^{light} (Joachims, 1998).

The LIBSVM software is developed for working sets sized 2, hence it tends to minimize the computational cost per iteration. In fact, in this case the inner QP subproblem can be analytically solved without requiring a numerical QP solver and the updating of the objective gradient only involves the two Hessian columns corresponding to the updated variables. On the other hand, if only few components are updated per iteration, slow convergence is generally implied. In the last LIBSVM release (ver. 2.8) this drawback is attenuated by a new working set selection that partially exploits the second order information, thus getting only a moderate increase of the computational cost with respect to the standard selections (Fan et al., 2005).

The SVM^{light} algorithm uses a more general decomposition strategy, in the sense that it can also exploit working sets of size larger than 2. By updating more variables per iteration, such an approach is well suited for a faster convergence, but it introduces additional difficulties and costs. A generalized maximal-violating pair policy for the working set selection and a numerical solver for the inner QP subproblems are needed; furthermore, we must recall that the more variables are changed per iteration, the more expensive is the objective gradient updating. Even if SVM^{light} can run with any working set size, numerical experiences show that it effectively faces the above difficulties only in case of small sized working sets ($n_{\text{sp}} = O(10)$), where it often exhibits comparable performance with LIBSVM.

Following the SVM^{light} decomposition framework, another attempt to reach a good trade-off between convergence rate and cost per iteration was introduced by Zanghirati and Zanni (2003). This was the first approach suited for an effective implementation on multiprocessors systems. Unlike SVM^{light}, it is designed to manage medium-to-large sized working sets ($n_{\text{sp}} = O(10^2)$ or $n_{\text{sp}} = O(10^3)$), that allow the scheme to converge in very few iterations, whose most expensive tasks (subproblem solving and gradient updating) can be easily and fruitfully distributed among the available processors. Of course, several issues must be addressed to achieve good performance, such as limiting the overhead for kernel evaluations and, also important, choosing a suitable inner QP solver. Zanghirati and Zanni (2003) obtained an efficient subproblem solution by a gradient projection-type method: it exploits the simple structure of the constraints, exhibits good convergence rate and is well suited for a parallel implementation. The promising results given by this parallel scheme can be now further improved thanks to some recent studies on both the gradient projection QP solvers (Serafini et al., 2005; Dai and Fletcher, 2006) and the selection rules for large sized working sets (Serafini and Zanni, 2005). On the basis of these studies a new *parallel gradient projection-based decomposition technique* (PGPDT) is developed and implemented in software available at `http://www.dm.unife/gpdt`.

Other parallel approaches to SVMs have been recently proposed, by splitting the training data into subsets and distributing them among the processors. Some of these approaches, such as those by Collobert et al. (2002) and by Dong et al. (2003), do not aim to solve the problem (1) and then perform non-standard SVM training. Collobert et al. (2002) presented a mixture of multiple SVMs where, cyclically, single SVMs are trained on subsets of the training set and a neural network is used to assign samples to different subsets. Dong et al. (2003) used a block-diagonal approximation of the kernel matrix to derive independent SVMs and filter out the examples which are estimated to be non-support vectors; then a new serial SVM is trained on the collected support vectors. The idea to combine asynchronously-trained SVMs is revisited also by the *cascade algorithm* introduced by Graf et al. (2005). The support vectors given by the SVMs of a cascade layer are combined to form the training sets of the next layer. At the end, the global KKT conditions are checked and the process is eventually restarted from the beginning, re-inserting the computed support vectors in the training subsets of the first layer. The authors prove that this feedback loop allows the algorithm to converge to a solution of (1) and consequently to perform a standard training. Unfortunately, for all these approaches no parallel code is available yet.

This work deals with the PGPDT software and its practical behaviour. First, we give the reader an exhaustive self-contained description of the PGPDT algorithm by showing how the crucial subtasks, singly developed in very recent works, are combined with appropriate load balancing strategies newly designed to get an effective parallel tool. Second, we show how, by exploiting the resources of common multiprocessor systems, PGPDT achieves good time speedup in comparison

with state-of-the-art serial packages and makes nonlinear SVMs tractable even on millions training samples.

The paper is organized as follows: Section 2 states the decomposition framework and describes its parallelization, Section 3 compares the PGPDT with SVM^{light} and LIBSVM on medium-to-large benchmark data sets and also faces some $O(10^6)$ real-world problems, Section 4 draws the main conclusions and future developments.

2. The Decomposition Framework and its Parallelization

To describe in detail the decomposition technique implemented by PGPDT we need some basic notations. At each decomposition iteration, the indices of the variables α_i, $i = 1,\ldots,n$, are split into the set $\mathcal{B}$ of *basic* variables, usually called the *working set*, and the set $\mathcal{N} = \{1,2,\ldots,n\} \setminus \mathcal{B}$ of *nonbasic* variables. As a consequence, the kernel matrix G and the vectors $\alpha = (\alpha_1,\ldots,\alpha_n)^T$ and $y = (y_1,\ldots,y_n)^T$ can be arranged with respect to $\mathcal{B}$ and $\mathcal{N}$ as follows:

$$G = \begin{bmatrix} G_{\mathcal{B}\mathcal{B}} & G_{\mathcal{B}\mathcal{N}} \\ G_{\mathcal{N}\mathcal{B}} & G_{\mathcal{N}\mathcal{N}} \end{bmatrix}, \qquad \alpha = \begin{bmatrix} \alpha_{\mathcal{B}} \\ \alpha_{\mathcal{N}} \end{bmatrix}, \qquad y = \begin{bmatrix} y_{\mathcal{B}} \\ y_{\mathcal{N}} \end{bmatrix}.$$

Furthermore, we denote by n_{sp} the size of the working set ($n_{\text{sp}} = \#\mathcal{B}$) and by α^* a solution of (1). Finally, suppose a distributed-memory multiprocessor system equipped with N_{P} processors is available for solving the problem (1) and that each processor has a local copy of the training set.

The decomposition strategy used by the PGPDT falls within the general scheme stated in Algorithm PDT. Here, we denote by the label "Distributed task" the steps where the N_{P} processors cooperate to perform the required computation; in these steps communications and synchronization are needed. In the other steps, the processors asynchronously perform the same computations on the same input data to obtain a local copy of the expected output data.

It must be observed that algorithm PDT essentially follows the SVM^{light} decomposition scheme proposed by Joachims (1998), but it allows to distribute among the available processors the subproblem solution in step A2 and the gradient updating in step A3. Thus, two important implications can be remarked: from the theoretical viewpoint, the PDT algorithm satisfies the same convergence properties of the SVM^{light} algorithm, but, in practice, it requires new implementation strategies in order to effectively exploit the resources of a multiprocessor system. Here, we state the main convergence results of the PDT algorithm and we will describe in the next subsections how its steps have been implemented in the PGPDT software.

The convergence properties of the sequence $\{\alpha^{(k)}\}$ generated by the PDT algorithm are mainly based on the special rule (4) for the working set selection. The rule was originally introduced by Joachims (1998) following an idea similar to the Zoutendijk's feasible direction approach, to define basic variables that make possible a rapid decrease of the objective function. The asymptotic convergence of the decomposition schemes based on this working set selection was first proved by Lin (2001a) by relating the selection rule with the violation of the KKT conditions and by assuming the following strict block-wise convexity assumption on $\mathcal{F}(\alpha)$:

$$\min_{\mathcal{J}}(\lambda_{\min}(G_{\mathcal{J}\mathcal{J}})) > 0\,, \tag{5}$$

ALGORITHM PDT Parallel decomposition technique

A1. *Initialization*. Set $\alpha^{(1)} = 0$ and let $n_{\rm sp}$ and n_c be two integer values such that $n \geq n_{\rm sp} \geq n_c > 0$, n_c even. Choose $n_{\rm sp}$ indices for the working set $\mathcal{B}$ and set $k = 1$.

A2. *QP subproblem solution*. **[Distributed task]** Compute the solution $\alpha_{\mathcal{B}}^{(k+1)}$ of

$$\begin{array}{rl} \min & \frac{1}{2}\alpha_{\mathcal{B}}^T G_{\mathcal{B}\mathcal{B}}\alpha_{\mathcal{B}} + \left(G_{\mathcal{B}\mathcal{N}}\alpha_{\mathcal{N}}^{(k)} - 1_{\mathcal{B}}\right)^T \alpha_{\mathcal{B}} \\ \text{sub. to} & \sum_{i\in\mathcal{B}} y_i\alpha_i = -\sum_{i\in\mathcal{N}} y_i\alpha_i^{(k)} \ , \\ & 0 \leq \alpha_i \leq C, \quad \forall\, i \in \mathcal{B} \ , \end{array} \tag{2}$$

where $1_{\mathcal{B}}$ is the $n_{\rm sp}$-vector of all one; set $\alpha^{(k+1)} = \left(\begin{array}{cc} \alpha_{\mathcal{B}}^{(k+1)^T}, & \alpha_{\mathcal{N}}^{(k)^T} \end{array} \right)^T$.

A3. *Gradient updating*. **[Distributed task]** Update the gradient

$$\nabla\mathcal{F}\left(\alpha^{(k+1)}\right) = \nabla\mathcal{F}\left(\alpha^{(k)}\right) + \left[\begin{array}{c} G_{\mathcal{B}\mathcal{B}} \\ G_{\mathcal{N}\mathcal{B}} \end{array} \right] \left(\alpha_{\mathcal{B}}^{(k+1)} - \alpha_{\mathcal{B}}^{(k)}\right) \tag{3}$$

and terminate if $\alpha^{(k+1)}$ satisfies the KKT conditions.

A4. *Working set updating*. Update $\mathcal{B}$ by the following selection rule:

A4.1. Find the indices corresponding to the nonzero components of the solution of

$$\begin{array}{rl} \min & \nabla\mathcal{F}\left(\alpha^{(k+1)}\right)^T d \\ \text{sub. to} & y^T d = 0 \ , \\ & d_i \geq 0 \quad \text{for } i \text{ such that } \alpha_i^{(k+1)} = 0 \ , \\ & d_i \leq 0 \quad \text{for } i \text{ such that } \alpha_i^{(k+1)} = C \ , \\ & -1 \leq d_i \leq 1 \ , \\ & \#\{d_i \mid d_i \neq 0\} \leq n_c \ . \end{array} \tag{4}$$

Let $\bar{\mathcal{B}}$ be the set of these indices.

A4.2. Fill $\bar{\mathcal{B}}$ up to $n_{\rm sp}$ entries with indices $j \in \mathcal{B}$. Set $\mathcal{B} = \bar{\mathcal{B}}$, $k \leftarrow k+1$ and go to A2.

where $\mathcal{J}$ is any subset of $\{1,\ldots,n\}$ with $\#\mathcal{J} \leq n_{sp}$ and $\lambda_{\min}(G_{\mathcal{J}\mathcal{J}})$ denotes the smallest eigenvalue of $G_{\mathcal{J}\mathcal{J}}$. This condition is used to prove that there exists $\tau > 0$ such that

$$\mathcal{F}(\alpha^{(k+1)}) \leq \mathcal{F}(\alpha^{(k)}) - \frac{\tau}{2}\|\alpha^{(k+1)} - \alpha^{(k)}\|^2 \qquad \forall k\,, \tag{6}$$

from which the important property $\lim_{k\to\infty}\|\alpha^{(k+1)} - \alpha^{(k)}\| = 0$ can be derived. Of course, the assumption (5) is satisfied when G is positive definite (for example, when the Gaussian kernel is used and all the training examples are distinct), but it may not hold in other instances of the problem (1). Convergence results that do not require the condition (5) are given by Lin (2002) and Palagi and Sciandrone (2005).

For the special case $n_{sp} = 2$, where the selection rule (4) gives only the two indices corresponding to the maximal violation of the KKT conditions (the *maximal-violating pair*), Lin (2002) has shown that the assumption (5) is not necessary to ensure the convergence.

For any working set size, Palagi and Sciandrone (2005) have shown that the condition (6) is ensured by solving at each iteration the following proximal point modification of the subproblem (2):

$$\begin{array}{rl} \min & \frac{1}{2}\alpha_{\mathcal{B}}^T G_{\mathcal{B}\mathcal{B}}\alpha_{\mathcal{B}} + \left(G_{\mathcal{B}\mathcal{N}}\alpha_{\mathcal{N}}^{(k)} - 1_{\mathcal{B}}\right)^T \alpha_{\mathcal{B}} + \frac{\tau}{2}\|\alpha_{\mathcal{B}} - \alpha_{\mathcal{B}}^{(k)}\|^2 \\ \text{sub. to} & \sum_{i\in\mathcal{B}} y_i\alpha_i = -\sum_{i\in\mathcal{N}} y_i\alpha_i^{(k)}\,, \\ & 0 \leq \alpha_i \leq C, \quad \forall\, i \in \mathcal{B}\,. \end{array} \tag{7}$$

Unfortunately, this modification affects the behaviour of standard decomposition schemes in a way which is not completely understood yet. Our preliminary experiences suggest that sufficiently large values of τ can easily allow a better performance of the inner QP solvers, but those values often imply a dangerous decrease in the convergence rate of the decomposition technique. On the other hand, too small values for τ do not produce essential differences with respect to the schemes where the subproblem (2) is solved.

In the PGPDT software, besides the default setting which implements the standard PDT algorithm, two different ways to generate a sequence satisfying the condition (6) are available by user selection: (i) solving the subproblem (7) in place of (2) at each iteration or (ii) solving (7) only as emergency step, when $\alpha_{\mathcal{B}}^{(k+1)}$ obtained via (2) fails to satisfy (6). All the computational experiments of Section 3 are carried out with the default setting that generally yields the best performance. For what concerns the practical rule used in the PGPDT to stop the iterative procedure, the fulfilment of the KKT conditions within a prefixed tolerance is checked (with the equality constraint multiplier computed as suggested by Joachims, 1998). The default tolerance is 10^{-3}, as it is usual in SVM packages, but different values can be selected by the user.

Before to describe the PGPDT implementation in detail, it must be recalled that this software is designed to be effective in case of sufficiently large n_{sp}, i.e., when iterations with well parallelizable tasks are generated. For this reason, in the sequel the reader may assume n_{sp} to be of medium-to-large size.

2.1 Parallel Gradient Projection Methods for the Subproblems

The inner QP subproblems (2) and (7) can fit into the following general form:

$$\min_{w\in\Omega} \quad f(w) = \frac{1}{2}w^T A w + b^T w \tag{8}$$

where $A \in \mathbb{R}^{n_{sp} \times n_{sp}}$ is dense, symmetric and positive semidefinite, $w, b \in \mathbb{R}^{n_{sp}}$ and the feasible region Ω is defined by

$$\Omega = \{w \in \mathbb{R}^{n_{sp}}, \quad \ell \leq w \leq u, \quad c^T w = \gamma\}, \qquad \ell,\, u,\, c \in \mathbb{R}^{n_{sp}},\ \ell < u. \tag{9}$$

We recall that the size n_{sp} is such that A can fit into the available memory.

Since subproblem (8) appears at each decomposition iteration, an effective inner solver becomes a crucial tool for the performance of a decomposition technique. The standard library QP solvers can be successfully applied only in the small size case ($n_{sp} = O(10)$), since their computational cost easily degrades the performance of a decomposition technique based on medium-to-large n_{sp}. For such kind of decomposition schemes, it is essential to design more efficient inner solvers able to exploit the special features of (8) and, for the PGPDT purposes, with the additional property to be easily parallelizable. To this end, the gradient projection methods are very appealing approaches (Bertsekas, 1999). They consist in a sequence of projections onto the feasible region, that are nonexpensive operations in the case of the special constraints (9). In fact, the projection onto Ω (denoted by $P_\Omega(\cdot)$) can be performed in $O(n_{sp})$ operations by efficient algorithms, like those by Pardalos and Kovoor (1990) and by Dai and Fletcher (2006). Furthermore, the single iteration core consists essentially in an n_{sp}-dimensional matrix-vector product that is suited to be optimized (by exploiting the vector sparsity) and also to be parallelized. Thus, the simple and nonexpensive iteration motivates the interest for these approaches as possible alternative to standard solvers based on expensive factorizations (usually requiring $O(n_{sp}^3)$ operations). A general parallel gradient projection scheme for (8) is depicted in Algorithm PGPM. As in the classical gradient projection methods, at each iteration a feasible descent direction $d^{(k)}$ is obtained by projecting onto Ω a point derived by taking a steepest descent step of length ρ_k from the current $w^{(k)}$. A linesearch procedure is then applied along the direction $d^{(k)}$ to decide the step size λ_k able to ensure the global convergence. The parallelization of this iterative scheme is simply obtained by a block row-wise distribution of A and by a parallel computation of the heaviest task of each iteration: the matrix-vector product $Ad^{(k)}$.

Concerning the convergence rate, that is the key element for the PGPM performance, the choices of both the steplength ρ_k and the linesearch parameter λ_k play a crucial role. Recent works have shown that appropriate selection rules for these parameters can significantly improve the typical slow convergence rate of the traditional gradient projection approaches (refer to Ruggiero and Zanni, 2000b, for the R-linear convergence of PGPM-like schemes). From the steplength viewpoint, very promising results are actually obtained with selection strategies based on the Barzilai-Borwein (BB) rules (Barzilai and Borwein, 1988):

$$\rho_{k+1}^{\mathrm{BB1}} = \frac{{d^{(k)}}^T d^{(k)}}{{d^{(k)}}^T A d^{(k)}}, \qquad \rho_{k+1}^{\mathrm{BB2}} = \frac{{d^{(k)}}^T A d^{(k)}}{{d^{(k)}}^T A^2 d^{(k)}}.$$

The importance of these rules has been observed in combination with both monotone and nonmonotone linesearch strategies (Birgin et al., 2000; Dai and Fletcher, 2005; Ruggiero and Zanni, 2000a). In particular, for the SVM applications, the special BB steplength selections proposed by Serafini et al. (2005), for the monotone scheme, and by Dai and Fletcher (2006), for the nonmonotone method, seem very efficient.

The *generalized variable projection method* (GVPM) by Serafini et al. (2005) uses a standard limited minimization rule as linesearch technique and an adaptive alternation of the two BB formulae. It outperforms the monotone gradient projection scheme used by Zanghirati and Zanni (2003),

ALGORITHM PGPM Parallel gradient projection method for step A2 of Algorithm PDT.

B1. *Initialization.*

[Data distribution] $\forall p = 1,\ldots,N_{\mathrm{P}}$: allocate a row-wise slice $A_p = (a_{ij})_{i\in I_p,\ j=1,\ldots,n_{\mathrm{sp}}}$ of A, where I_p is the subset of row indices belonging to processor p:

$$I_p \subset \{1,\ldots,n\}, \quad \bigcup_{p=1}^{N_{\mathrm{P}}} I_p = \{1,\ldots,n\}\,, \quad I_i \cap I_j = \emptyset \ \text{ for } i \neq j.$$

Furthermore, allocate local copies of all the other input data.

Initialize the parameters for the steplength selection rule and for the linesearch strategy.

Set $w^{(0)} \in \Omega\,, \quad 0 < \rho_{\min} < \rho_{\max}\,, \quad \rho_0 \in [\rho_{\min}, \rho_{\max}]\,, \quad k = 0.$

[Distributed task] $\forall p = 1,\ldots,N_{\mathrm{P}}$: compute the local slice $t_p^{(0)} = A_p w^{(0)}$ and send it to all the other processors; assemble a local copy of the full $t^{(0)} = Aw^{(0)}$ vector.

Set $\quad g^{(0)} = \nabla f(w^{(0)}) = Aw^{(0)} + b = t^{(0)} + b\,.$

B2. *Projection.*

Terminate if $w^{(k)}$ satisfies a stopping criterion; otherwise compute the descent direction

$$d^{(k)} = P_\Omega\left(w^{(k)} - \rho_k g^{(k)}\right) - w^{(k)}\,.$$

B3. *Matrix-vector product.*

[Distributed task] $\forall p = 1,\ldots,N_{\mathrm{P}}$: compute the local slice $z_p^{(k)} = A_p d^{(k)}$ and send it to all the other processors; assemble a local copy of the full $z^{(k)} = Ad^{(k)}$ vector.

B4. *Linesearch.*

Compute the linesearch step λ_k and $w^{(k+1)} = w^{(k)} + \lambda_k d^{(k)}$.

B5. *Update.*

Compute

$$t^{(k+1)} = Aw^{(k+1)} = t^{(k)} + \lambda_k A d^{(k)} = t^{(k)} + \lambda_k z^{(k)},$$
$$g^{(k+1)} = \nabla f(w^{(k+1)}) = Aw^{(k+1)} + b = t^{(k+1)} + b,$$

and a new steplength ρ_{k+1}.

Update the parameters for the linesearch strategy, set $k \leftarrow k+1$ and go to step B2.

Initialization (step B1). Set $i_\rho = 2,\ n_{\min} = 3,\ n_{\max} = 10,\ \lambda_\ell = 0.1,\ \lambda_u = 5,\ n_\rho = 1$.

Linesearch (step B4). Compute $\lambda_k = \arg\min_{\lambda\in[0,1]} f(w^{(k)} + \lambda d^{(k)})$.

Update (step B5).

If $d^{(k)^T} A d^{(k)} = 0$ then

set $\rho_{k+1} = \rho_{\max}$

else

compute $\rho_{k+1}^{\mathrm{BB1}}$, $\rho_{k+1}^{\mathrm{BB2}}$, $\lambda_{\mathrm{opt}} = \arg\min_{\lambda} f(w^{(k)} + \lambda d^{(k)})$.

If $(n_\rho \geq n_{\min})$ and $\Big[(n_\rho \geq n_{\max})$ or $(\rho_{k+1}^{\mathrm{BB2}} \leq \rho_k \leq \rho_{k+1}^{\mathrm{BB1}})$

or $\Big((\lambda_{\mathrm{opt}} < \lambda_\ell$ and $\rho_k = \rho_k^{\mathrm{BB1}})$ or $(\lambda_{\mathrm{opt}} > \lambda_u$ and $\rho_k = \rho_k^{\mathrm{BB2}})\Big)\Big]$ then

set $i_\rho \leftarrow \mathrm{mod}(i_\rho, 2) + 1,\quad n_\rho = 0$;

end.

end.

Compute $\rho_{k+1} = \min\Big\{\rho_{\max}, \max\Big\{\rho_{\min}, \rho_{k+1}^{\mathrm{BB}i_\rho}\Big\}\Big\}$ and set $n_\rho \leftarrow n_\rho + 1$.

Figure 1: linesearch and steplength rule for the GVPM method.

that was simply based on an alternation of the BB rules every three iterations. Furthermore, the numerical experiments reported by (Serafini et al., 2005) show that the GVPM is much more efficient than the pr_LOQO (Smola, 1997) and MINOS (Murtagh and Saunders, 1998) solvers, two softwares widely used within the machine learning community. GVPM steplength selection and linesearch are described in Figure 1.

The Dai-Fletcher scheme is based on the following steplength selection:

$$\rho_{k+1}^{\mathrm{DF}} = \frac{\sum_{i=0}^{m-1} s^{(k-i)^T} s^{(k-i)}}{\sum_{i=0}^{m-1} s^{(k-i)^T} v^{(k-i)}}\ , \qquad m \geq 1\ , \tag{10}$$

where $s^{(j)} = w^{(j+1)} - w^{(j)}$ and $v^{(j)} = g^{(j+1)} - g^{(j)}$, $(g^{(j)} = \nabla f(w^{(j)}))$, $j = 0, 1, \ldots$. Observe that the case $m = 1$ reduces to the standard BB rule $\rho_{k+1}^{\mathrm{BB1}}$. In order to frequently accept the full step $w^{(k+1)} = w^{(k)} + d^{(k)}$ generated with the above steplength, a special nonmonotone linesearch is used. Figure 2 describes the version of the Dai-Fletcher method corresponding to the parameters setting suggested by Zanni (2006) for the SVM applications. It may be observed that the linesearch parameter $\lambda_k = \arg\min_{\lambda\in[0,1]} f(w^{(k)} + \lambda d^{(k)})$ is used only if $f(w^{(k)} + d^{(k)}) \geq f_{\mathrm{ref}}$ and not at each iteration, as in the GVPM. The steplength selection corresponds to the rule (10) with $m = 2$ and, for what concerns the iteration cost, no significant additional tasks are required in comparison to the GVPM ($g^{(k+1)}$ is already available in step B5).

The PGPDT software can run the PGPM with either the GVPM or the Dai-Fletcher scheme, the latter being the default due to better experimental convergence rate (Zanni, 2006).

We end this subsection with some further details about the PGPM implementation used within the PGPDT software. The starting point $w^{(0)}$ is $P_\Omega(\alpha_{\mathcal{B}}^{(k)})$ if the stopping rule of the decomposition procedure is nearly satisfied, otherwise $w^{(0)} = P_\Omega(0)$ is used. This aims to start the PGPM with

Initialization (step B1). Set $L = 2$, $f_{\text{ref}} = \infty$, $f_{\text{best}} = f_c = f(w^{(0)})$, $h = 0$, $k = 0$, $s^{(k-1)} = v^{(k-1)} = 0$.

Linesearch (step B4).

If $\left(k = 0 \text{ and } f(w^{(k)} + d^{(k)}) \geq f(w^{(k)})\right)$ or $\left(k > 0 \text{ and } f(w^{(k)} + d^{(k)}) \geq f_{\text{ref}}\right)$ then

$$w^{(k+1)} = w^{(k)} + \lambda_k d^{(k)} \qquad \text{with} \qquad \lambda_k = \underset{\lambda \in [0,1]}{\arg\min} f(w^{(k)} + \lambda d^{(k)})$$

else

$$w^{(k+1)} = w^{(k)} + d^{(k)}$$

end.

Update (step B5). Compute $s^{(k)} = w^{(k+1)} - w^{(k)}$; $v^{(k)} = g^{(k+1)} - g^{(k)}$.

If $s^{(k)^T} v^{(k)} = 0$ then

set $\rho_{k+1} = \rho_{\max}$

else

If $s^{(k-1)^T} v^{(k-1)} = 0$ then

set $\rho_{k+1} = \min\left\{\rho_{\max}, \max\left\{\rho_{\min}, \dfrac{s^{(k)^T} s^{(k)}}{s^{(k)^T} v^{(k)}}\right\}\right\}$

else

set $\rho_{k+1} = \min\left\{\rho_{\max}, \max\left\{\rho_{\min}, \dfrac{s^{(k)^T} s^{(k)} + s^{(k-1)^T} s^{(k-1)}}{s^{(k)^T} v^{(k)} + s^{(k-1)^T} v^{(k-1)}}\right\}\right\}$

end.

end.

If $f(w^{(k+1)}) < f_{\text{best}}$ then

set $f_{\text{best}} = f(w^{(k+1)})$, $f_c = f(w^{(k+1)})$, $h = 0$;

else

set $f_c = \max\left\{f_c, f(w^{(k+1)})\right\}$, $h = h + 1$;

If $h = L$ then

set $f_{\text{ref}} = f_c$, $f_c = f(w^{(k+1)})$, $h = 0$;

end.

end.

Figure 2: linesearch and steplength rule for the Dai-Fletcher method.

sparse vectors in the first decomposition steps, and to save inner solver iterations at the end of the decomposition, where slight changes in $\alpha_{\mathcal{B}}^{(k)}$ are expected. At the beginning we also set $\rho_{\min} = 10^{-10}$, $\rho_{\max} = 10^{10}$ and $\rho_0 = \min\{\rho_{\max}, \max\{\rho_{\min}, \bar{\rho}_0\}\}$, where $\bar{\rho}_0 = \|P_\Omega(w^{(0)} - (Aw^{(0)} + b)) - w^{(0)}\|_\infty^{-1}$. For the computation of $P_\Omega(\cdot)$ in step B2, the default is the following: if $n_{\text{sp}} \leq 20$ the bisection-like method described by Pardalos and Kovoor (1990) is used, else the secant-based algorithm proposed by Dai and Fletcher (2006) is chosen, that usually is faster for large size. However, the user can select one of the two projectors. Finally, we remark that the PGPM stopping rule is the same used

for the decomposition technique: the fulfilment of the KKT conditions within a prefixed tolerance. In the PGPDT, the tolerance required to the inner solver depends on the quality of the outer iterate $\alpha^{(k)}$: in the first iterations the same tolerance as the decomposition scheme is used, while a progressively lower tolerance is imposed when $\alpha^{(k)}$ nearly satisfies the outer stopping criterion. In our experience, a more accurate inner solution just from the beginning doesn't imply remarkable increase of the overall performance.

2.2 Parallel Gradient Updating

The gradient updating in step A3 is usually the most expensive task of a decomposition iteration. Since the matrix G is assumed to be out of memory, in order to obtain $\nabla \mathcal{F}(\alpha^{(k+1)})$ some entries of G need to be computed and, consequently, some kernel evaluations are involved that can be very expensive in case of large sized input space and not much sparse training examples. Thus, any strategy able to save kernel evaluations or to optimize their computation is crucial for minimizing the time consumption for updating the gradient. The updating formula (3) allows to get $\nabla \mathcal{F}(\alpha^{(k+1)})$ by involving only the columns of G corresponding to the indices for which $(\alpha_i^{(k+1)} - \alpha_i^{(k)}) \neq 0,\ i \in \mathcal{B}$. Further improvements in the number of kernel evaluations can be obtained by introducing a caching strategy, consisting in using an area of the available memory to store some elements of G to avoid their recomputation in subsequent iterations. PGPDT fills the caching area with the columns of G involved in (3); when the cache is full, the current columns substitute those that have not been used for the largest number of iterations. This simple trick seems to well combine with the working set selection used in step A4, which forces some indices of the current $\mathcal{B}$ to remain in the new working set (see the next section for more details), and remarkable reduction of the kernel evaluations are often observed. Nevertheless, the improvements implied by a caching strategy are obviously dependent on the size of the caching area. To this regard, the large amount of memory available on modern multiprocessor systems is an appealing resource for improving the performance of a decomposition technique. One of the innovative features of PGPDT is to implement a parallel gradient updating where both the matrix-vector multiplication and the caching strategy are distributed among the processors. This is done by asking each processor to perform a part of the column combinations required in (3) and to make available its local memory for caching the columns of G. In this way, the gradient updating benefits not only from a computations distribution, but also from a reduction of the kernel evaluations due to much larger caching areas. Of course, these features are not shared by standard decomposition packages, designed to exploit the resources of only one processor. The main steps of the above parallel updating procedure are summarized in Algorithm PGU.

Concerning the reduction of the kernel evaluations, it is worth to recall that the entries of G stored in the caching area can be used also for $G_{\mathcal{B}\mathcal{B}}$ in step A2. Moreover, for the computation of the linear term in (2), the equality

$$G_{\mathcal{B}\mathcal{N}}\alpha_{\mathcal{N}}^{(k)} - 1_{\mathcal{B}} = \nabla \mathcal{F}_{\mathcal{B}}(\alpha^{(k)}) - G_{\mathcal{B}\mathcal{B}}\alpha_{\mathcal{B}}^{(k)}$$

can avoid additional kernel evaluations by exploiting already computed quantities.

The gradient updating overhead within each decomposition iteration can be further reduced by optimizing the kernel computation. Even if a caching strategy can limit the number of kernel evaluations, large problems often require millions of them and their optimization becomes a need. PGPDT uses sparse vector representation of the training examples and exploits the sparseness in the dot products required by the kernel evaluations. Three kernels are available: linear, polynomial

ALGORITHM PGU Parallel gradient updating in step A3 of Algorithm PDT

i) Denote by $W_p,\ p = 1,2,\ldots,N_{\mathrm{P}}$, the caching area of the processor p and by G_i the i-th column of G. Let

$$\mathcal{B}_1 = \left\{ i \in \mathcal{B} \quad | \quad \alpha_i^{(k+1)} - \alpha_i^{(k)} \neq 0 \right\},$$
$$\mathcal{B}_n = \left\{ i \in \mathcal{B}_1 \quad | \quad G_i \notin W_p,\ \ p = 1,2\ldots,N_{\mathrm{P}} \right\}, \qquad \mathcal{B}_c = \mathcal{B}_1 \setminus \mathcal{B}_n.$$

Distribute among the processors the sets $\mathcal{B}_c$ and $\mathcal{B}_n$ and denote by $\mathcal{B}_{c,p}$ and $\mathcal{B}_{n,p}$ the sets of indices assigned to processor p. Make the distribution in such a way that

$$\mathcal{B}_c = \bigcup_{i=1}^{N_{\mathrm{P}}} \mathcal{B}_{c,i}, \qquad \mathcal{B}_{c,i} \cap \mathcal{B}_{c,j} = \emptyset \ \text{ for } \ i \neq j, \qquad \forall i \in \mathcal{B}_{c,p} \ \Rightarrow \ G_i \in W_p,$$
$$\mathcal{B}_n = \bigcup_{i=1}^{N_{\mathrm{P}}} \mathcal{B}_{n,i}, \qquad \mathcal{B}_{n,i} \cap \mathcal{B}_{n,j} = \emptyset \ \text{ for } \ i \neq j$$

and by trying to obtain a well balanced workload among the processors.

ii) $\forall p = 1,2,\ldots,N_{\mathrm{P}}$: use the columns $G_i \in W_p$, $i \in \mathcal{B}_{c,p}$, to compute

$$r_p = \begin{bmatrix} G_{\mathcal{B}\,\mathcal{B}_{c,p}} \\ G_{\mathcal{N}\,\mathcal{B}_{c,p}} \end{bmatrix} \left(\alpha_{\mathcal{B}_{c,p}}^{(k+1)} - \alpha_{\mathcal{B}_{c,p}}^{(k)} \right),$$

then compute the columns $G_i,\ i \in \mathcal{B}_{n,p}$, necessary to obtain

$$r_p \leftarrow r_p + \begin{bmatrix} G_{\mathcal{B}\,\mathcal{B}_{n,p}} \\ G_{\mathcal{N}\,\mathcal{B}_{n,p}} \end{bmatrix} \left(\alpha_{\mathcal{B}_{n,p}}^{(k+1)} - \alpha_{\mathcal{B}_{n,p}}^{(k)} \right)$$

and store in W_p as much as possible of these columns, eventually by substituting those less recently used.

iii) $\forall p = 1,2,\ldots,N_{\mathrm{P}}$: send r_p to all the other processors and assemble a local copy of

$$\nabla \mathcal{F}\left(\alpha^{(k+1)}\right) = \nabla \mathcal{F}\left(\alpha^{(k)}\right) + \sum_{i=1}^{N_{\mathrm{P}}} r_i.$$

ALGORITHM SP1 Selection procedure for step A4.1 of algorithm PDT.

i) Sort the indices of the variables according to $y_i \nabla \mathcal{F}(\alpha^{(k+1)})_i$ in decreasing order and let $I \equiv (i_1, i_2, \ldots, i_n)^T$ be the sorted list (i.e., $y_{i_1} \nabla \mathcal{F}(\alpha^{(k+1)})_{i_1} \geq y_{i_2} \nabla \mathcal{F}(\alpha^{(k+1)})_{i_2} \geq \ldots \geq y_{i_n} \nabla \mathcal{F}(\alpha^{(k+1)})_{i_n}$).

ii) Repeat the selection of a pair $(i_t, i_b) \in I \times I$, with $t < b$, as follows:

- moving down from the top of the sorted list, choose $i_t \in I_{\text{top}}(\alpha^{(k+1)})$,
- moving up from the bottom of the sorted list, choose $i_b \in I_{\text{bot}}(\alpha^{(k+1)})$,

until n_c indices are selected or a pair with the above properties cannot be found.

iii) Let $\bar{\mathcal{B}}$ be the set of the selected indices.

and Gaussian. The interested reader is referred to the available code for more details on their practical implementation. We end this section by remarking that, in case of linear kernel, the updating formula (3) can be simplified in

$$t = \sum_{i \in \mathcal{B}_1} y_i x_i \left(\alpha_i^{(k+1)} - \alpha_i^{(k)} \right), \qquad \mathcal{B}_1 = \left\{ i \in \mathcal{B} \quad | \quad \alpha_i^{(k+1)} - \alpha_i^{(k)} \neq 0 \right\},$$

$$\nabla \mathcal{F}(\alpha^{(k+1)})_j = \nabla \mathcal{F}(\alpha^{(k)})_j + y_j x_j^T t, \qquad j = 1, 2, \ldots, n, \tag{11}$$

and the importance of a caching strategy is generally negligible. Consequently, PGPDT faces linear SVMs without any caching strategy and performs the gradient updating by simply distributing the n tasks (11) among the processors.

2.3 Working Set Selection

In this section we describe how the working set updating in step A4 of the PDT algorithm is implemented within PGPDT. It consists in two phases: in the first phase at most n_c indices are chosen for the new working set by solving the problem (4), while in the second phase at least $n_{\text{sp}} - n_c$ entries are selected from the current $\mathcal{B}$ to complete the new working set. The selection procedure in step A4.1 was first introduced by Joachims (1998) and then rigorously justified by Lin (2001a). In short, by using the notation

$$I_{\text{top}}(\alpha) \equiv \left\{ i \mid (\alpha_i < C \text{ and } y_i = -1) \text{ or } (\alpha_i > 0 \text{ and } y_i = 1) \right\},$$
$$I_{\text{bot}}(\alpha) \equiv \left\{ j \mid (\alpha_j > 0 \text{ and } y_j = -1) \text{ or } (\alpha_j < C \text{ and } y_j = 1) \right\},$$

this procedure can be stated as in Algorithm SP1.

It is interesting to recall how this selection procedure is related to the violation of the first order optimality conditions. For the convex problem (1) the KKT conditions can also be written as

$$\text{a feasible } \alpha^* \text{ is optimal} \quad \Longleftrightarrow \quad \max_{i \in I_{\text{top}}(\alpha^*)} y_i \nabla \mathcal{F}(\alpha^*)_i \leq \min_{j \in I_{\text{bot}}(\alpha^*)} y_j \nabla \mathcal{F}(\alpha^*)_j .$$

ALGORITHM SP2 Selection procedure for step A4.2 of algorithm PDT.

i) Let $\bar{\mathcal{B}}$ be the set of indices selected in step A4.1.

ii) Fill $\bar{\mathcal{B}}$ up to $n_{\rm sp}$ entries by adding the most recent indices[†] $j \in \mathcal{B}$ satisfying $0 < \alpha_j^{(k+1)} < C$; if these indices are not enough, then add the most recent indices $j \in \mathcal{B}$ such that $\alpha_j^{(k+1)} = 0$ and, eventually, the most recent indices $j \in \mathcal{B}$ satisfying $\alpha_j^{(k+1)} = C$.

iii) Set $n_c = \min\{ n_c, \max\{10, J, n_{\rm new}\}\}$, where J is the largest even integer such that $J \leq \frac{n_{\rm sp}}{10}$ and $n_{\rm new}$ is the largest even integer such that $n_{\rm new} \leq \#\{ j,\ j \in \bar{\mathcal{B}} \setminus \mathcal{B}\}$;
set $\mathcal{B} = \bar{\mathcal{B}}$, $k \leftarrow k+1$ and go to step A2.

[†]We mean the indices that are in the working set $\mathcal{B}$ since the lowest number of consecutive iterations.

It means that, given a non-optimal feasible α, there exists at least a pair $(i, j) \in I_{\rm top}(\alpha) \times I_{\rm bot}(\alpha)$ satisfying

$$y_i \nabla \mathcal{F}(\alpha)_i > y_j \nabla \mathcal{F}(\alpha)_j \ .$$

Following Keerthi and Gilbert (2002), these pairs are called *KKT-violating pairs* and, from this point of view, the above selection procedure chooses indices $(i, j) \in I_{\rm top}(\alpha^{(k+1)}) \times I_{\rm bot}(\alpha^{(k+1)})$ by giving priority to those pairs which most violate the optimality conditions. In particular, at each iteration the *maximal-violating pair* is included in the working set: this property is crucial for the asymptotic convergence of a decomposition technique.

From the practical viewpoint, the indices selected via problem (4) identify steepest-like feasible descent directions: this is aimed to get a quick decrease of the objective function $\mathcal{F}(\alpha)$. Nevertheless, for fast convergence, both n_c and the updating phase in step A4.2 have a key relevance. In fact, as it is experimentally shown by Serafini and Zanni (2005), values of n_c equal or close to $n_{\rm sp}$ often yield a dangerous *zigzagging* phenomenon (i.e., some variables enter and leave the working set many times), which can heavily degrade the convergence rate especially for large $n_{\rm sp}$. This drawback suggests to set n_c sufficiently smaller than $n_{\rm sp}$ and then it opens the problem of how to select the remaining indices to fill up the new working set. The studies available in literature on this topic (see Hsu and Lin, 2002; Serafini and Zanni, 2005; Zanghirati and Zanni, 2003, and also the SVM^{light} code) suggest that an efficient approach consists in selecting these indices from the current working set. We recall in Algorithm SP2 the filling strategy recently proposed in (Serafini and Zanni, 2005) and used by the PGPDT software.

The selection policy used by Algorithm SP2 is based on two criteria: the first accords priority to the free variables over the variables at either the lower or the upper bound, the second takes into account *how long* (i.e., how many consecutive decomposition iterations) a variable has been into the working set. Roughly speaking, both the criteria aim to preserve into the working set the variables which are likely to need further optimization. The interested reader can find in the papers by Hsu and Lin (2002) and by Serafini and Zanni (2005) a deeper discussion on these criteria and the computational evidence of their benefits in terms of convergence rate. Finally, Algorithm SP2 also introduces an adaptive reduction of the parameter n_c, useful in case of large sized working sets. This trick allows the decomposition technique to start with n_c close to $n_{\rm sp}$, in order to optimize many new variables in the very first iterations, and avoids zigzagging through the progressive reduction of

n_c. The reduction takes place only if n_c is larger than an empirical threshold and it is controlled via the number of those new indices selected in step A4.1 that do not belong to the current working set.

3. Computational Experiments

The aim of this computational study is to analyse the PGPDT performance. To this end, it is also worth to show that the serial version of the proposed software (called GPDT) can train SVMs with effectiveness comparable to that of the state-of-the-art softwares LIBSVM (ver. 2.8) and SVM^{light} (ver. 6.01). Since there are no other parallel software currently available for comparison, the PGPDT will be evaluated in terms of scaling properties with respect to the serial packages.

Our implementation is an object oriented C^{++} code and its parallel version uses standard MPI communication routines (Message Passing Interface Forum, 1995), hence it is easily portable on many multiprocessor systems. Most of the experiments are carried out on an IBM SP5, which is an IBM SP Cluster 1600 equipped with 64 nodes p5-575 interconnected by a high performance switch (HPS). Each node owns 8 IBM SMP Power5 processors at 1.9GHz and 16GB of RAM (2GB per CPU). The serial packages run on this computer by exploiting only a single CPU. PGPDT has been tested also on different parallel architectures and, for completeness, we report the results obtained on a system where less memory than in the IBM SP5 is available for each CPU: the IBM CLX/1024 Linux Cluster, that owns 512 nodes equipped with two Intel Xeon processors at 3.0GHz and 1GB of RAM per CPU. Both the systems are available at the CINECA Supercomputing center (Bologna, Italy, `http://www.cineca.it`).

The considered softwares are compared on several medium, large and very large test problems generated from well known benchmark data sets, described in the next subsection.

3.1 Test Problems

We trained Gaussian and polynomial SVMs with kernel functions $K(x_i,x_j) = \exp\left(-\|x_i - x_j\|^2/(2\sigma^2)\right)$ and $K(x_i,x_j) = \left(s({x_i}^T x_j)+1\right)^d$, respectively[1].

In what follows we give some details on the databases used for the generation of the training sets, as well as on the SVM parameters we have chosen. Error rates are given as the percentage of misclassifications.

The UCI Adult data set (at `http://www.research.microsoft.com/~jplatt/smo.html`) allows to train an SVM to predict whether a household has an income greater than \$50000. The inputs are 123-dimensional binary sparse vectors with sparsity level $\approx 89\%$. We use the largest version of the data set, sized 32561. We train a Gaussian SVM with training parameters chosen accordingly to the database technical documentation, i.e., $C = 1$ and $\sigma = \sqrt{10}$, that are indicated as those maximizing the performance on a (unavailable) validation set.

The Web data set (available at `http://www.research.microsoft.com/~jplatt/smo.html`) concerns a web page classification problem with a binary representation based on 300 keyword features. On average, the sparsity level of the examples is about 96%. We use the largest version of the data set, sized 49749. We train a Gaussian SVM with the parameters suggested in the data set documentation: $C = 5$ and $\sigma = \sqrt{10}$. As before, these values are claimed to give the best performance on a (unavailable) validation set.

1. Here the notation has the usual meaning: σ is the Gaussian's variance, s is the polynomial scaling parameter and d is the polynomial degree.

The MNIST database of handwritten digits (`http://yann.lecun.com/exdb/mnist`) contains 784-dimensional nonbinary sparse vectors; the data set size is 60000 and the data sparsity is $\approx$ 81%. The provided test set is sized 10000. We train two SVM classifiers for the digit "8" with the following parameters: $C = 10$, $\sigma = 1800$ for the Gaussian kernel and $C = 3000$, $d = 4$, $s = 3 \cdot 10^{-9}$ for the polynomial kernel. This setting gives the following error rates on the test set: 0.55% for the Gaussian kernel and 0.60% for the polynomial kernel.

The Forest Cover Type data set[2] has 581012 samples with 54 attributes, distributed in 8 classes. The average sparsity level of the samples is about 78%. We train some SVM classifiers for separating class 2 from the other classes. The training sets, sized up to 300000, are generated by randomly sampling the data set. We use a Gaussian kernel with $\sigma^2 = 2.5 \cdot 10^4$, $C = 10$. For the largest training set the error rate is about 3.6% on the test set given by the remaining 281012 examples.

The KDDCUP-99 Intrusion Detection data set[3] consists in binary TCP dump data from seven weeks of network traffic. Each original pattern has 34 continuous features and 7 symbolic features. As suggested by Tsang et al. (2005), we normalize each continuous feature to the range $[0,1]$ and transform each symbolic feature to multiple binary features. In this way, the inputs are 122-dimensional sparse vectors with sparsity level $\approx 90\%$. We work with the whole training set sized 4898431 and with some smaller subsets obtained by randomly sampling the original database. We use a Gaussian kernel with parameters $\sigma^2 = (1.2)^{-1}$, $C = 2$. This choice yields error rates of about 7% on the test set of 311029 examples available in the database.

3.2 Serial Behaviour

In the first experiments set, we analyse the behaviour of the serial code on the test problems just described. In Table 1 we report the time in seconds (sec.), the decomposition iteration count (it.) and the number of kernel evaluations in millions (MKernel) required for each one of the considered SVM training packages. The values we use for the working set parameters n_{sp} and n_c are also reported: as mentioned, the LIBSVM software works only with $n_{\text{sp}} = n_c = 2$, whilst both SVM^{light} and GPDT accept larger values. For these two softwares, meaningful ranges of parameters were explored: we report the results corresponding to the pairs that gave the best training time and to the default setting ($n_{\text{sp}} = n_c = 10$ for SVM^{light}, $n_{\text{sp}} = 400$, $n_c = \lfloor n_{\text{sp}}/3 \rfloor = 132$ for GPDT). SVM^{light} is run with several values of n_{sp} in the range $[2,80]$ with both its inner solvers: the Hildreth-D'Esopo and the pr_LOQO. The best training time is obtained by using the Hildreth-D'Esopo solver with n_{sp} small and $n_c = n_{\text{sp}}/2$, generally observing a significant performance decrease for $n_{\text{sp}} > 40$.

We run the codes assigning to the caching area 512MB for the MNIST test problems and 768MB in the other cases; the default threshold $\varepsilon = 10^{-3}$ for the termination criterion is used, except for the two largest Cover Type and KDDCUP-99 test problems, where the stopping tolerance ε is set to 10^{-2}. All the other parameters are assigned default values. This means that both LIBSVM and SVM^{light} benefit from the *shrinking* (Joachims, 1998) strategy that is not implemented in the current release of GPDT.

Table 1 well emphasizes the different approach of the three softwares. In particular we see how GPDT, by exploiting large working sets, converges in far less iterations than the other softwares, but its iterations are much heavier. Looking at the computational time, GPDT seems to be very competitive with respect to both LIBSVM and SVM^{light}. Furthermore, the kernel column highlights

2. Available at `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype`.
3. Available at `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

Data set	n	n_{sp}	n_c	sec.	it.	MKernel
		GPDT				
UCI Adult	32561	400	132	94.1	162	494.2
		400	200	93.6	129	498.5
MNIST (poly)	60000	400	132	379.6	598	424.6
		600	200	345.3	221	324.4
MNIST (Gauss)	60000	400	132	359.2	136	504.8
		2000	300	341.2	22	396.4
Web Pages	49749	400	132	69.6	228	285.5
		600	200	62.2	101	252.9
Cover Type	300000	400	132	24365.5	3730	120846.5
		500	80	21561.4	5018	99880.0
KDDCUP-99	400000	400	132	10239.0	1149	56548.3
		180	60	9190.3	2248	51336.7
		LIBSVM				
UCI Adult	32561	2	2	165.9	15388	452.1
MNIST (poly)	60000	2	2	2154.4	452836	792.0
MNIST (Gauss)	60000	2	2	1081.8	20533	409.4
Web Pages	49749	2	2	64.0	13237	170.3
Cover Type	300000	2	2	17271.7	274092	53152.6
KDDCUP-99	400000	2	2	11220.8	40767	50773.8
		SVM^{light}				
UCI Adult	32561	10	10	216.7	10448	405.1
		20	10	201.1	4317	393.5
		40	20	203.8	2565	410.3
MNIST (poly)	60000	10	10	6454.1	380743	1943.8
		4	2	3090.2	420038	859.8
		8	4	3124.0	238609	905.6
MNIST (Gauss)	60000	10	10	795.6	10262	278.3
		4	2	570.3	18401	204.1
		16	8	562.8	4970	203.8
Web Pages	49749	10	10	108.6	8728	208.5
		4	2	93.8	12195	166.9
		16	8	92.7	4444	188.2
Cover Type	300000	10	10	82892.6	266632	146053.2
		8	4	29902.3	151762	44791.4
		16	8	28585.5	78026	48864.9
KDDCUP-99	400000	10	10	11356.4	21950	23941.3
		8	4	10141.8	28254	21663.6
		20	10	12308.4	20654	24966.0

Table 1: performance of the serial packages on different test problems.

Solver	SV	BSV	$\mathcal{F}_{\text{opt}}$	b	test error
MNIST (poly) test problem					
GPDT	2712	640	−2555033.8	3.54283	0.63%
LIBSVM	2715	640	−2555033.6	3.54231	0.63%
SVM^{light}	2714	640	−2555033.0	3.54213	0.62%
Cover Type test problem					
GPDT	50853	32683	−299399.7	0.22083	3.62%
LIBSVM	51131	32573	−299396.0	0.22110	3.63%
SVM^{light}	51326	32511	−299393.9	0.22149	3.62%

Table 2: accuracy of the serial solvers.

how GPDT benefits from a good optimization of the execution time for the kernel computation: compare, for instance, the results for the MNIST Gaussian test, where the kernel evaluations are very expensive. Here, in front of a number of kernel evaluations similar to LIBSVM and larger than SVM^{light}, a significant lower training time is exhibited. The same consideration holds true for the MNIST polynomial test; however in this case the good GPDT performance is also due to a lower number of kernel evaluations.

The next experiments are intended to underline how the good training time given by GPDT is accompanied by scaling and accuracy properties very similar to the other packages. From the accuracy viewpoint, this is shown for two of the considered test problems by reporting in Table 2 the number of support vectors (SV) and bound support vectors (BSV), the computed optimal value $\mathcal{F}_{\text{opt}}$ of the objective function, the bias b of the separating surface expression[4] (Cristianini and Shawe-Taylor, 2000) and the error rate on the test set.

For what concerns the scaling, Figure 3a shows, for the Cover Type test problem (the worst case for GPDT), the training time with respect to the problem size. All the packages exhibit almost the same dependence that, for this particular data set, seems between quadratic and cubic with respect to the number of examples. For completeness, the number of support vectors of these test problems is also reported in Figure 3b.

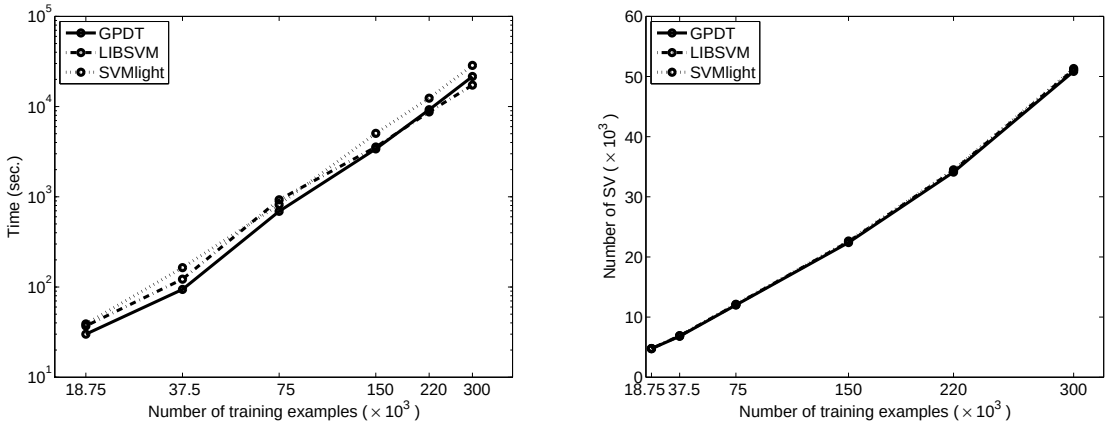
3.3 Parallel Behaviour

The second experiments set concerns with the behaviour of PGPDT. We evaluate PGPDT on the previous four largest problems and some very large problems sized $O(10^6)$ derived from the KDDCUP-99 data set.

3.3.1 LARGE TEST PROBLEMS

For a meaningful comparison against the serial version, PGPDT is run on the MNIST, Cover Type and KDDCUP-99 ($n = 400000$) test problems with the same n_{sp}, n_c and ε parameters as in the previous experiments; furthermore, the same amount of caching area (768MB) is now allocated on each CPU of the IBM SP5. Default values are assigned to the other parameters.

4. The *support vectors* are those samples in the training set corresponding to $\alpha_i^* > 0$; the samples with $\alpha_i^* = C$ are called *bound support vectors*. Roughly speaking, the support vectors are the samples characterizing the hypersurface separating the two classes and the bias b is its displacement.



(a) dependence on n.

(b) number of support vectors.

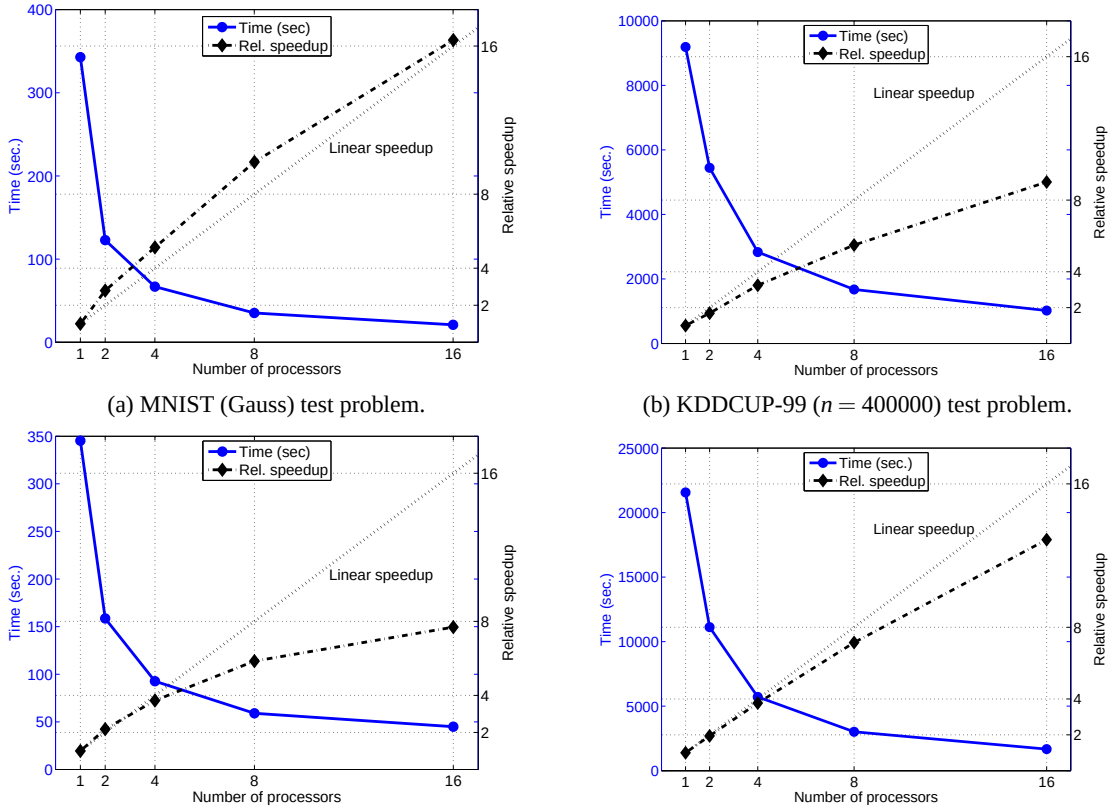
Figure 3: scaling of the serial solvers on test problems from the Cover Type data set.

N_P	sec.	sp_r	it.	MKernel	SV	BSV	$\mathcal{F}_{opt}$
	MNIST (poly) test problem						
1	345.3		221	324.2	2712	640	−2555033.8
2	158.6	2.18	212	249.2	2710	640	−2555033.8
4	100.5	3.44	214	253.9	2711	641	−2555033.8
8	59.7	5.78	212	259.8	2711	641	−2555033.7
16	47.3	7.30	217	271.4	2711	641	−2555033.8
	Cover Type test problem						
1	21561		5018	99880	50853	32683	−299399.7
2	11123	1.94	5047	98925	50786	32673	−299399.8
4	5715	3.77	5059	93597	50786	32668	−299399.9
8	3016	7.15	5086	82853	50832	32664	−299399.9
16	1673	12.89	5029	59439	50826	32697	−299399.9

Table 3: PGPDT scaling on the IBM SP5 system.

Table 3 and Figure 4 summarize the results obtained by running PGPDT on different numbers of processors. We evaluate the parallel performance by the *relative speedup*, defined as $sp_r = T_{\text{serial}}/T_{\text{parallel}}$, where T_{serial} is the training time spent on a single processor, while T_{parallel} denotes the training time on N_P processors.

Seeking clearness, in Table 3 we also report additional information on the overall PGPDT behaviour. In particular, we can see an essentially constant number of decomposition iterations (recall that only the computational burden within the decomposition iteration is distributed) and the same solution accuracy as the serial run (compare the numbers in SV, BSV and $\mathcal{F}_{opt}$ columns). Moreover, remark the lower number of total kernel evaluations needed by the parallel version, due to the growing amount of global caching memory available, which our parallel caching strategy is able to exploit. This is the motivation of the superlinear speedup observed in some situations like the MNIST (Gaussian) test problem (Figure 4a). Unfortunately, there may be cases where the bene-



(a) MNIST (Gauss) test problem.

(b) KDDCUP-99 ($n = 400000$) test problem.

(c) MNIST (poly) test problem.

(d) Cover Type test problem.

Figure 4: PGPDT scaling on the IBM SP5 system.

fits due to the parallel caching strategy are not sufficient to ensure optimal speedups. For instance, sometimes the n_{sp} values that give satisfactory serial performance are not suited for good PGPDT scaling. This is the case of the KDDCUP-99 test problem (Figure 4b), where the small working sets sized $n_{sp} = 180$ imply many decomposition iterations and consequently the fixed costs of the non-distributed tasks (working set selection and stopping rule) become very heavy. Another example is provided by the MNIST (polynomial) test problem (Figure 4c): here the subproblem solution is a dominant task in comparison to the gradient updating and the suboptimal scaling of the PGPM solver on 16 processors leads to poor speedups. However, also in these cases remarkable time reductions are observed in comparison with the serial softwares (see Table 1).

We further remark that all these considerations are quite dependent on the underlying parallel architecture. In particular, on multiprocessor systems where less memory than in the SP5 platform is available for each CPU, even better speedups can be expected due to the effectiveness of the parallel caching strategy. For instance, we report in Figure 5 what we get for the KDDCUP-99 test problem on the IBM CLX/1024 Linux Cluster, where only 400MB of caching area can be allocated on each CPU. Due to both the worse performance of this machine and the reduced caching area, larger training time is required, but an optimal PGPDT speedup is now observed up to 16 processors.

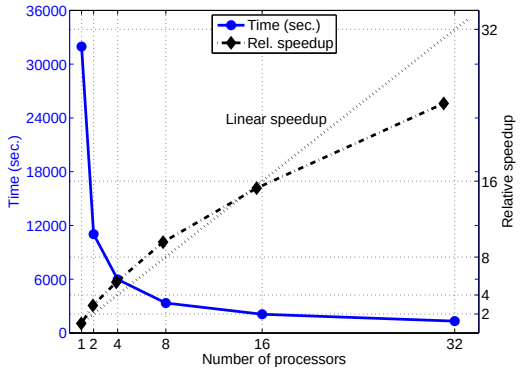
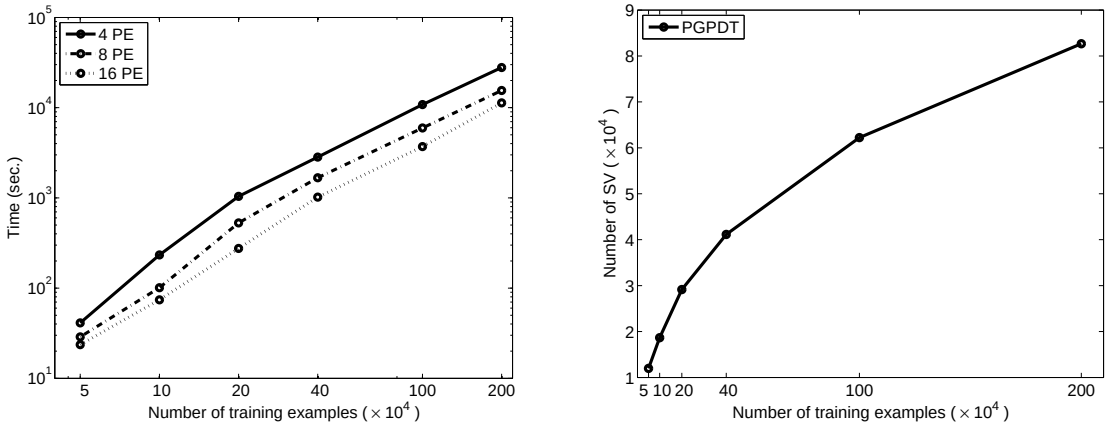


Figure 5: PGPDT scaling on the CLX/1024 system for the KDDCUP-99 ($n = 4 \cdot 10^5$) test problem.



(a) dependence on n.

(b) number of support vectors.

Figure 6: Parallel training time for different sizes of the KDDCUP-99 test probelms

3.3.2 VERY LARGE TEST PROBLEMS

In this section we present the behavior of the PGPDT code on very large test problems. In particular we considered three test problems from the KDDCUP-99 data set of size $n = 10^6$, $2 \cdot 10^6$ and 4898431, the latter being the full data set size. The test problems are obtained by training Gaussian SVMs with the parameters setting previously used for this data set.

In the two larger cases a different setting for the n_{sp}, n_c and caching area have been used. In particular, for the case $n = 2 \cdot 10^6$ we used $n_{\text{sp}} = 150$, $n_c = 40$ and 600Mb of caching area; for the full case we used $n_{\text{sp}} = 90$, $n_c = 30$ and 250Mb of caching area. The reason for reducing the caching area is that every processor can allocate no more that 1.7Gb of memory and, when the data set size increases, most of the memory is used for storing the training data and cannot be used for caching.

These $O(10^6)$ test problems are firstly used to study how the PGPDT time complexity scales with the size of the data sets. In Figure 6a the training time is reported for 4, 8 and 16 processors. Figure 6b shows the growth rate of the support vectors for these test problems. It can be observed that the scaling is close to $O(n^2)$, as often exhibited by the serial state-of-the-art decom-

position packages (Collobert and Bengio, 2001; Joachims, 1998). This result is quite natural if we remember that PGPDT is based on a parallelization of each iteration of a standard decomposition technique. Concerning the subquadratic scaling exhibited for increasing sizes, it can be motivated by the sublinear growth of the support vectors observed on these experiments; however, in different situations it may be expected a training time complexity that scales at least quadratically (see, for instance, the experiments on the Cover Type data set described in Figure 3).

Table 4 shows the PGPDT performance in terms of training time and accuracy for different number of processors. Here, the time is measured in hours and minutes and the kernel evaluations are expressed in billions. For the test problem sized $n = 2 \cdot 10^6$, the serial results concern only the GPDT because LIBSVM exceeded the time limit of 60 hours and SVM^{light} stopped without a valid solution after relaxing the KKT conditions. Due to the very large size of the problem, the amount of 600MB for the caching area seems not sufficient to prevent a huge number of kernel evaluations in the serial run. Again, this drawback is reduced in the multiprocessor runs, due to increased memory for caching. Thus, analogously to some previous experiments (see Figures 4a, 5), superlinear speedup is exhibited, in this case up to about 20 processors. The largest test problem, with size about 5 millions and more than 10^5 support vectors, can be faced in a reasonable time only with the parallel version. In this case the overall remark is that, on the considered architecture, few processors allow to train the Gaussian SVM in less than one day while few tens of processors can be exploited to reduce the training time to about 10 hours.

Finally, by observing in Table 4 the column of the objective function values, we may confirm that also in these experiments the training time saving ensured by PGPDT is obtained without damaging the solution accuracy.

These results show that PGPDT is able to exploit the resources of today multiprocessor systems to overcome the limits of the serial SVM implementations in solving $O(10^6)$ problems (see also the training time in Figure 6a). As already mentioned, there is no other available parallel software to perform a fair comparison on the same architecture and the same data; however, an indirect comparison with the results reported by Graf et al. (2005) for the cascade algorithm suggests that PGPDT could be really competitive. Furthermore, since the cascade algorithm and PGPDT exploit very different parallelization ideas (recall that the former is based on the distribution of smaller independent SVMs), promising improvements could be achieved by an appropriate combination of the two approaches.

4. Conclusions and Future Work

Parallel software to train linear and nonlinear SVMs for classification problems is presented, which is suitable for distributed memory multiprocessors systems. It implements an iterative decomposition technique based on a gradient projection solver for the inner subproblems. At each decomposition iteration, the heaviest tasks, i.e., solving the subproblem and updating the gradient, are distributed among the available processors. Furthermore, a parallel caching strategy allows to effectively exploit as much memory as available to avoid expensive kernel recomputations. Numerical comparisons with the state-of-the-art softwares LIBSVM and SVM^{light} on benchmark problems show the significant speedup that the proposed parallel package can achieve in training large scale SVMs. In short, experiments on $O(10^6)$ data sets show that nonlinear SVMs with $O(10^5)$ support vectors can be trained in few hours by exploiting some tens of processors. Thus, this parallel package, available at `http://dm.unife.it/gpdt`, can be a useful tool for overcoming the limits of the

N_P	time	it.	GKernel	SV	BSV	$\mathcal{F}_{opt}$
	$n = 2 \cdot 10^6$					
1	$54^h\ 59^m$	6192	1135.8	82521	466	−9625.9
2	$14^h\ 22^m$	6077	468.5	84565	463	−9625.8
4	$7^h\ 44^m$	6005	458.1	82193	464	−9625.7
8	$4^h\ 18^m$	6064	462.9	82723	462	−9625.8
16	$3^h\ 08^m$	6116	467.0	84100	460	−9625.9
24	$2^h\ 47^m$	6202	473.0	83626	464	−9626.0
	$n = 4898431$					
8	$19^h\ 08^m$	12300	1752.9	131041	1021	−14479.6
16	$12^h\ 16^m$	12295	1739.7	130918	1046	−14479.6
32	$9^h\ 22^m$	12310	1742.9	131736	1017	−14479.6

Table 4: PGPDT scaling on very large test problems from the KDDCUP-99 data set.

serial SVM implementation currently available. The main improvements will concern: (i) the optimization/distribution of the tasks which are not currently parallelized, to improve the scalability; (ii) the introduction of a shrinking strategy, for further reducing the number of kernel evaluations; (iii) the inner solver robustness, to better face the subproblems arising from badly scaled training data. Furthermore, work is in progress to include in a new PGPDT release a suitable data distribution and the extension to regression problems.

Acknowledgments

The authors are grateful to the staff of the CINECA Supercomputing Center (Bologna, Italy) for supporting us with their valuable expertise. The authors also thank the anonymous referees for their helpful comments.

References

Jonathan Barzilai and Jonathan M. Borwein. Two-point step size gradient methods. *IMA Journal of Numerical Analysis*, 8(1):141–148, 1988.

Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, second edition, 1999.

Ernesto G. Birgin, José Mario Martínez, and Marcos Raydan. Nonmonotone spectral projected gradient methods on convex sets. *SIAM Journal on Optimization*, 10(4):1196–1211, 2000.

Bernhard E. Boser, Isabelle Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In David Haussler, editor, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152. ACM Press, Pittsburgh, PA, 1992.

Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines, 2001. URL `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Pai-Hsuen Chen, Rong-En Fan, and Chih-Jen Lin. A study on SMO-type decomposition methods for support vector machines. Technical report, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, 2005. To appear on IEEE Transaction on Neural Networks, 2006.

Ronan Collobert and Samy Bengio. SVMTorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research*, 1:143–160, 2001.

Ronan Collobert, Samy Bengio, and Yoshua Bengio. A parallel mixture of SVMs for very large scale problems. *Neural Computation*, 14(5):1105–1114, 2002.

Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and other Kernel-Based Learning Methods*. Cambridge University Press, 2000.

Yu-Hong Dai and Roger Fletcher. New algorithms for singly linearly constrained quadratic programs subject to lower and upper bounds. *Mathematical Programming*, 106(3):403–421, 2006.

Yu-Hong Dai and Roger Fletcher. Projected Barzilai-Borwein methods for large-scale box-constrained quadratic programming. *Numerische Mathematik*, 100(1):21–47, 2005.

Jian-Xiong Dong, Adam Krzyzak, and Ching Y. Suen. A fast parallel optimization for training support vector machine. In P. Perner and A. Rosenfeld, editors, *Proceedings of 3rd International Conference on Machine Learning and Data Mining*, volume 17, pages 96–105. Springer Lecture Notes in Artificial Intelligence, Leipzig, Germany, 2003.

Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training Support Vector Machines. *Journal of Machine Learning Research*, 6: 1889–1918, 2005.

Hans Peter Graf, Eric Cosatto, Léon Bottou, Igor Dourdanovic, and Vladimir N. Vapnik. Parallel support vector machines: the Cascade SVM. In Lawrence Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2005.

Chih-Wei Hsu and Chih-Jen Lin. A simple decomposition method for support vector machines. *Machine Learning*, 46:291–314, 2002.

Don Hush and Clint Scovel. Polynomial-time decomposition algorithms for support vector machines. *Machine Learning*, 51:51–71, 2003.

Throstem Joachims. Making large-scale SVM learning practical. In Bernard Schölkopf, C.J.C. Burges, and Alex Smola, editors, *Advances in Kernel Methods – Support Vector Learning*. MIT Press, Cambridge, MA, 1998.

S. Sathiya Keerthi and Elmer G. Gilbert. Convergence of a generalized SMO algorithm for SVM classifier design. *Machine Learning*, 46:351–360, 2002.

Chih-Jen Lin. On the convergence of the decomposition method for support vector machines. *IEEE Transactions on Neural Networks*, 12:1288–1298, 2001a.

Chih-Jen Lin. Linear convergence of a decomposition method for support vector machines. Technical report, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, 2001b.

Chih-Jen Lin. Asymptotic convergence of an SMO algorithm without any assumptions. *IEEE Transactions on Neural Networks*, 13:248–250, 2002.

Message Passing Interface Forum. MPI: A message-passing interface standard (version 1.2). *International Journal of Supercomputing Applications*, 8(3/4), 1995. URL `http://www.mpi-forum.org`. Also available as Technical Report CS-94-230, Computer Science Dept., University of Tennesse, Knoxville, TN.

Bruce A. Murtagh and Michael A. Saunders. MINOS 5.5 user's guide. Technical report, Department of Operation Research, Stanford University, Stanford CA, 1998.

Edgar Osuna, Robert Freund, and Girosi Federico. Training support vector machines: an application to face detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR97)*, pages 130–136. IEEE Computer Society, New York, 1997.

Laura Palagi and Marco Sciandrone. On the convergence of a modified version of SVM^{light} algorithm. *Optimization Methods and Software*, 20:317–334, 2005.

Panos M. Pardalos and Naina Kovoor. An algorithm for a singly constrained class of quadratic programs subject to upper and lower bounds. *Mathematical Programming*, 46:321–328, 1990.

John C. Platt. Fast training of support vector machines using sequential minimal optimization. In Bernard Schölkopf, C.J.C. Burges, and Alex Smola, editors, *Advances in Kernel Methods – Support Vector Learning*. MIT Press, Cambridge, MA, 1998.

Valeria Ruggiero and Luca Zanni. A modified projection algorithm for large strictly convex quadratic programs. *Journal of Optimization Theory and Applications*, 104(2):281–299, 2000a.

Valeria Ruggiero and Luca Zanni. Variable projection methods for large convex quadratic programs. In Donato Trigiante, editor, *Recent Trends in Numerical Analysis*, volume 3 of *Advances in the Theory of Computational Mathematics*, pages 299–313. Nova Science Publisher, 2000b.

Thomas Serafini and Luca Zanni. On the working set selection in gradient projection-based decomposition techniques for support vector machines. *Optimization Methods and Software*, 20: 583–596, 2005.

Thomas Serafini, Gaetano Zanghirati, and Luca Zanni. Gradient projection methods for quadratic programs and applications in training support vector machines. *Optimization Methods and Software*, 20:353–378, 2005.

Alex J. Smola. pr_LOQO optimizer, 1997. URL `http://www.kernel-machines.org/code/prloqo.tar.gz`.

Ivor W. Tsang, James T. Kwok, and Pak-Ming Cheung. Core vector machines: fast SVM training on very large data sets. *Journal of Machine Learning Research*, 6(4):363–392, 2005.

Vladimir N. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, New York, 1998.

Gaetano Zanghirati and Luca Zanni. A parallel solver for large quadratic programs in training support vector machines. *Parallel Computing*, 29:535–551, 2003.

Luca Zanni. An improved gradient projection-based decomposition technique for support vector machines. *Computational Management Science*, 3(2):131–145, 2006.