# A Fast Projected Conjugate Gradient Algorithm for Training Support Vector Machines

Tong Wen, Alan Edelman, and David Gorsich

Abstract. Support Vector Machines (SVMs) are of current interest in the solution of classification problems. However, serious challenges appear in the training problem when the training set is large. Training SVMs involves solving a linearly constrained quadratic programming problem. In this paper, we present a fast and easy-to-implement projected Conjugate Gradient algorithm for solving this quadratic programming problem.

Compared with the exiting ones, this algorithm tries to be adaptive to each training problem and each computer's memory hierarchy. Although written in a high-level programming language, numerical experiments show that the performance of its MATLAB implementation is competitive with that of benchmark C/C++ codes such as SVM$^{light}$ and SvmFu. The parallelism of this algorithm is also discussed in this paper.

## Introduction

Support Vector Machines (SVMs) have attracted recent attention as a technique to attack classification problems, where prior statistical knowledge about the underlying classes is not available. Two examples are the face detection problem [**13**] [**18**] and the handwritten-digit recognition problem [**4**] [**14**] [**15**] [**16**]. In summary, SVMs tell the difference between two classes by learning from examples of these two classes. The learning procedure involves solving a linearly constrained quadratic programming problem, which is challenging because its size can be large. In this paper, we present a fast and easy-to-implement projected Conjugate Gradient algorithm for solving this quadratic programming problem.

To establish our notation, we first give a brief introduction to SVMs in Section 1. We then present this algorithm in the language of linear algebra in Section 2. Also in this section, addressed are the issues of how to be adaptive to each training problem and each computer's memory architecture. The MATLAB implementation of this algorithm is compared numerically with two benchmark C/C++ codes SVM$^{light}$ and SvmFu in Section 3. The timing results based on two representative

2000 *Mathematics Subject Classification.* Primary 65F10 90C20; Secondary 65Y20 68T10.

This paper is supported by U.S. Army TACOM under the contract TCN 00-130 awarded by Battelle-Research Triangle Park and NSF under the grant DMS-9971591.

We would like to thank Ryan Rifkin at M.I.T. for many valuable discussions.

training sets of size $O(10,000)$ show that the performance of this MATLAB implementation is very competitive. Parallelism is discussed in Section 4. Finally, we conclude this paper with Section 5 exploring future work.

## 1. Support Vector Machines

The goal of this introduction is to set the ground for our later discussion. For more exhaustive treatments of SVMs, we refer readers to **[5]** **[6]** **[21]** **[24]**. As a matter of notation, we use bold typeface for vectors, and normal typeface for scalars such as vector and matrix components. Matrices are indicated by capital letters.

SVMs tell the difference between two classes by learning from instances or examples of these two classes. Geometrically, SVMs estimate the optimal separating boundary between these two classes by a pair of parallel hyperplanes. If the training examples of these two classes are linearly separable as indicated in Figure 1, the maximal-margin SVM hyperplanes are the pair that separates these training examples with the maximum gap. Each training example is represented by a pair $(\boldsymbol{x}_i, y_i)$, where $\boldsymbol{x}_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$, for $i = 1, \ldots, m$. The vector $\boldsymbol{x}_i$ is the description of the $i$th example, while the scalar $y_i$ is the label indicating which class this example belongs to. Let $\boldsymbol{w}$ be the vector defining the normal direction of an oriented hyperplane. We can represent any two parallel (oriented) hyperplanes by the following equation:

$$\boldsymbol{w}^T\boldsymbol{x} + b = \pm 1, \text{ where } b \in \mathbb{R}. \tag{1.1}$$

It is easy to see that the distance between these two parallel hyperplanes is

$$d = \frac{2}{\|\boldsymbol{w}\|_2}. \tag{1.2}$$

For $i = 1, \ldots, m$, the following inequality enforces that the hyperplanes must separate all the training examples:

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1. \tag{1.3}$$

Therefore, if the training examples are linearly separable, the pair of maximal-margin hyperplanes can be computed by solving the following linearly constrained quadratic programming (QP) problem:

$$\underset{\boldsymbol{w},b}{\text{minimize}} \quad f(\boldsymbol{w}, b) \equiv \frac{1}{2}\|\boldsymbol{w}\|_2^2 \tag{1.4}$$

subject to

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1, \quad \text{for } i = 1, \ldots, m. \tag{1.5}$$

The assumption of linear separability in Figure 1 is not general, because given $m$ points in $\mathbb{R}^n$, they may not be separable by hyperplanes. To separate linearly non-separable examples, a map $\phi(\cdot)$ is used to map them to a higher dimensional space $\mathbb{R}^{n'}$ $(n' > n)$, so that in $\mathbb{R}^{n'}$, the mapped examples can be separated by a pair of parallel hyperplanes. Note that our hyperplane model still holds here. The only difference is that $\boldsymbol{x}_i$ is replaced by $\phi(\boldsymbol{x}_i)$. The underlying idea is simple: if a linear function is not enough to separate the examples, then a nonlinear one is used. Since only inner products are involved in computing the SVM separating hyperplanes as we will see later, a positive definite kernel function $k(\cdot)$ is used instead of $\phi(\cdot)$, where

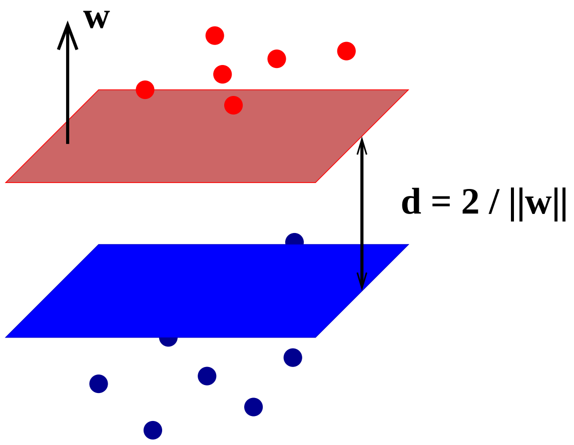



FIGURE 1. A pair of parallel (oriented) hyperplanes separating examples of two classes labeled by $\pm 1$, where point $\boldsymbol{x}_i$ along with its label $y_i$ represents one training example $(\boldsymbol{x}_i, y_i)$. $\boldsymbol{w}$ gives the normal direction of these two hyperplanes, which points to the positive (gray) points. The positive and the negative (dark) points are linearly separable.

$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$. The kernel function $k(\cdot)$ is preferred due to the curse of dimensionality.

Given $m$ training examples, we can always separate them by choosing a certain kernel function $k(\cdot)$. However, to avoid overfitting we may not separate the training examples exactly. Remember that the goal here is to distinguish the two classes as accurately as possible rather than to separate a particular set of training examples correctly. To incorporate the case where examples are not exactly separable, error terms (nonnegative slack variables) $\varepsilon_i$ are introduced. The generalized QP problem is:

$$\underset{w,b,\varepsilon}{\text{minimize}} \quad f(\boldsymbol{w}, b, \boldsymbol{\varepsilon}) \equiv \frac{1}{2}\|\boldsymbol{w}\|_2^2 + c\sum_i \varepsilon_i \tag{1.6}$$

subject to

$$y_i(\boldsymbol{w}^T \phi(\boldsymbol{x}_i) + b) \geq 1 - \varepsilon_i \text{ and } \varepsilon_i \geq 0, \text{ for } i = 1, \dots, m. \tag{1.7}$$

The pair of hyperplanes determined by the above problem is referred as the soft-margin SVM hyperplanes.

If the training point $\phi(\boldsymbol{x}_i)$ is correctly separated by the soft-margin hyperplanes, then $\varepsilon_i = 0$. To make the SVM hyperplanes represent the general difference between two classes, a large gap (a small $\|\boldsymbol{w}\|_2$) is preferred, whereas the goal to separate the training examples correctly (the goal to minimize the error term $\sum_i \varepsilon_i$) tends to reduce the gap. The constant $c$ (specified beforehand) determines the trade-off between the size of the gap and how well the examples are separated.

Problem (1.6) subject to (1.7) is referred as the primal SVM QP problem, which as we have seen has a clear geometric meaning. However, in practice its dual problem is solved because of its simplicity:

$$\underset{\boldsymbol{\alpha}}{\text{maximize}} \quad F(\boldsymbol{\alpha}) \equiv \boldsymbol{\alpha}^T \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^T H \boldsymbol{\alpha} \tag{1.8}$$

subject to

$$\boldsymbol{y}^T \boldsymbol{\alpha} = 0, \tag{1.9}$$

$$\mathbf{0} \leq \boldsymbol{\alpha} \leq \boldsymbol{c}, \tag{1.10}$$

where $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_m]^T$, $\boldsymbol{y} = [y_1, \ldots, y_m]^T$, $\mathbf{1} = [1, \ldots, 1]^T$ and $\boldsymbol{c} = c\mathbf{1}$. $H$ is a $m \times m$ symmetric positive semi-definite matrix with

$$\begin{aligned} H_{ij} &= y_i(\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j))y_j \\ &= y_i k(\boldsymbol{x}_i, \boldsymbol{x}_j) y_j. \end{aligned}$$

Each dual variable (Lagrange multiplier) $\alpha_i$ corresponds to one example $(\boldsymbol{x}_i, y_i)$. To simplify our notation, we use $\boldsymbol{x}_i$ in place of $\phi(\boldsymbol{x}_i)$ in our remaining discussion, considering that the hyperplane model always holds. At optimality, the primal variables $\boldsymbol{w}$ and $b$ are determined by $\boldsymbol{\alpha}$ as the following:

$$\boldsymbol{w} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i, \tag{1.11}$$

and if $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1$ then

$$b = y_i - \boldsymbol{w}^T \boldsymbol{x}_i = y_i(1 - \boldsymbol{e}_i^T H \boldsymbol{\alpha}), \tag{1.12}$$

where $\boldsymbol{e}_i$ is the $i$th column of the $m \times m$ identity matrix $I_m$. Given an unseen point $\boldsymbol{x}$, the SVM decision rule for predicting its class is

$$\begin{aligned} h_{\text{SVM}}(\boldsymbol{x}) &= \operatorname{sgn}(\boldsymbol{w}^T \boldsymbol{x} + b) \\ &= \operatorname{sgn}(\sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i^T \boldsymbol{x} + b). \end{aligned}$$

At optimality, the values of $\alpha_i$ partition the set of training points

$$S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$$

into three parts $S_1$, $S_2$ and $S_3$ with the following property:

$$\begin{cases} \alpha_i = 0 & \Longleftrightarrow \boldsymbol{x}_i \in S_1,\ S_1 = \{\boldsymbol{x}_i |\ y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) > 1\ (\varepsilon_i = 0)\}; \\ 0 < \alpha_i < c & \Longleftrightarrow \boldsymbol{x}_i \in S_2,\ S_2 = \{\boldsymbol{x}_i |\ y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1\ (\varepsilon_i = 0)\}; \\ \alpha_i = c & \Longleftrightarrow \boldsymbol{x}_i \in S_3,\ S_3 = \{\boldsymbol{x}_i |\ y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1 - \varepsilon_i\ \ (\varepsilon_i > 0)\}. \end{cases}$$

$S_2 \cup S_3$ contain the Support Vectors (SVs). It follows that at optimality the equality $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1 - \varepsilon_i$ holds only when $\boldsymbol{x}_i$ is a SV. In other words, only the SVs determine the pair of SVM separating hyperplanes.

An important property of SVMs is that $\boldsymbol{\alpha}$ is generally sparse because usually we have $m > n$. The intuition explaining the sparsity is that in $\mathbb{R}^n$ to determine a hyperplane with $n+1$ unknowns we only need $n+1$ equations. This is the property that people exploit to develop a fast algorithm.

## 2. Solving the SVM Dual Quadratic Programming Problem

The more examples SVMs learn from, the better predicting performance they tend to have. However, when the training set $S_T = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\}$ is large, it becomes expensive to store and access the $m \times m$ Hessian matrix $H$. Hence, it becomes hard to solve the dual QP problem for standard QP solvers that take $H$ explicitly as an input. For instance, when $m = 10,000$, it needs almost 1 gigabyte to store $H$. Many modern computers may not even have that much memory. Another concern is that when $m > n$, $H$ is rank deficient, which may bring numerical difficulties for certain solvers where the positive definiteness of $H$ is assumed.

In this section, we describe a fast, memory efficient and easy-to-implement algorithm for solving the SVM dual QP problem. This projected Conjugate Gradient algorithm is presented in the language of linear algebra. One of the advantages of building our algorithm on basic linear algebra (BLA) operations is that these operations have been well implemented across computer platforms, so it is easy to implement this algorithm by leveraging off previous work. Also addressed are issues related to performance improvement such as how to make this algorithm adaptive to each training problem and each computer's memory architecture. Although programmed in a high-level language, as you will see in the next section, the performance of our MATLAB implementation of this algorithm is competitive with that of benchmark C/C++ codes such as SVM$^{light}$ [**10**] and SvmFu [**20**].

### 2.1. A Projected Conjugate Gradient Algorithm.

An important property of the dual QP problem is that its solution is sparse. Ignoring the training examples $(\boldsymbol{x}_i, y_i)$ with $\alpha_i = 0$ does not change the solution. For example, to compute the pair of maximal-margin hyperplanes shown in Figure 2, solving the right-hand-side separation problem based on the SVs is equivalent to solving the left-hand-side one, but it is much cheaper. Knowing which training points are the SVs in advance enables us to solve a much smaller problem. One of the main themes of this section is to exploit the sparsity property so as to solve the dual QP problem fast and with less memory usage.

To illustrate a better way to compute the maximal-margin hyperplanes, as an example, let us solve the SVM separation problem indicated in Figure 2 in the following way. Initially, we randomly choose a small set, say $G = \{\boldsymbol{x}_3, \boldsymbol{x}_4, \boldsymbol{x}_5\}$ as our guess of the SVs. Then we solve the $3 \times 3$ separation problem based on the working set $G$, and we use its solution (indicated by the doted lines) to test the training points that are not in $G$. For any $\boldsymbol{x}_i \in \overline{G}$ that is separated correctly by the current solution, it is still considered to be a non-SV; otherwise, we add it to $G$. For this problem, $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_6\}$ are added to $G$. Finally, solving the updated $6 \times 6$ separation problem gives us the optimal solution with the SVs $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3\}$. Since solving these two smaller subproblems is cheaper than solving the original separation problem as a whole, this idea works. By controlling how many points to add to the working set $G$ at each time, we can control the size of each subproblem. For instance, we can add only one point to $G$ to formulate the second subproblem.

The above idea is quite natural from the learning perspective. We first build our knowledge of the difference between two classes based on a small number of examples. We then improve it by learning from future mistakes in distinguishing these two classes. In summary, the procedure is

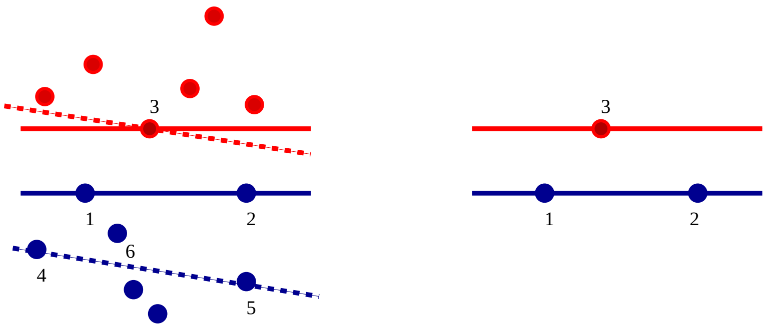



FIGURE 2. An example of a two-dimensional maximal-margin SVM separation problem. The optimal solution is the pair of lines that separate the positive (gray) and negative (dark) points with the largest gap. The SVs $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3\}$ and the optimal solution are indicated in the right-hand-side figure. The two dotted lines are the maximal-margin solution to separate points $\boldsymbol{x}_3$ from $\boldsymbol{x}_4$ and $\boldsymbol{x}_5$.

(1) Initially, set $\boldsymbol{\alpha} = \mathbf{0}$ and a guess of the SVs $G$ is chosen.
(2) The subproblem based on the working set $G$ is constructed and solved.
(3) If the solution from Step 2 separates all the points in $\overline{G}$ correctly, **stop**; otherwise, a certain number of points in $\overline{G}$ with the largest separation errors are added to $G$, and at the same time, the points in $G$ that are not SVs for the current solution can be dropped to $\overline{G}$, then go to step 2.

When $\boldsymbol{\alpha}$ is sparse, solving the sequence of smaller subproblems determined by the above procedure is much cheaper than solving the dual QP problem directly.

As indicated by Figure 2, this strategy can be easily appreciated from the primal problem point of view. Our goal is to show how to use this strategy to solve the dual QP problem efficiently, where some training examples may not be separated correctly by the final solution. In this section, issues such as how to construct, update, and solve the subproblems are addressed.

Other methods can be found in [**17**] [**11**] [**12**] [**19**]. The basic idea underlying all these methods is the same, that is, to decompose the original large problem into a sequence of smaller and easy-to-solve subproblems. But they differ in their decomposition strategies and their solutions to the subproblems. Some of our ideas are inspired by the Constrained Conjugate Gradient Algorithm described in [**12**], while [**11**] and [**19**] describe respectively the ideas underlying SVM$^{light}$ and more or less the ideas underlying SvmFu. To our knowledge, SVM$^{light}$ is currently the most popular training code.

2.1.1. *Constructing a Subproblem.*

First, let us consider in general how to project a linearly constrained QP problem onto an affine space. Suppose a new constraint is added to the dual QP problem requiring that the solution must lie in a subspace, say the column space of a matrix $P \in \mathbb{R}^{n\times p}$, where $p \le n$. Under this restriction, $\boldsymbol{\alpha}$ can be expressed as $\boldsymbol{\alpha} = P\hat{\boldsymbol{\alpha}}$, where $\hat{\boldsymbol{\alpha}} \in \mathbb{R}^p$. More generally, $\boldsymbol{\alpha}$ can be written as $\boldsymbol{\alpha} = \boldsymbol{\alpha}_0 + P\hat{\boldsymbol{\alpha}}$, where $\boldsymbol{\alpha}_0$ is the initial feasible value of $\boldsymbol{\alpha}$. For this case, $\boldsymbol{\alpha}$ is restricted in the affine space determined by $\boldsymbol{\alpha}_0$ and $P$. By substituting $\boldsymbol{\alpha}$ with $\boldsymbol{\alpha}_0 + P\hat{\boldsymbol{\alpha}}$ in the dual QP problem, we have

$$\text{(2.1)} \qquad \underset{\hat{\boldsymbol{\alpha}}}{\text{maximize}} \quad \hat{F}(\hat{\boldsymbol{\alpha}}) \equiv \hat{\boldsymbol{\alpha}}^T P^T \boldsymbol{r}_0 - \frac{1}{2}\hat{\boldsymbol{\alpha}}^T P^T H P \hat{\boldsymbol{\alpha}}$$

subject to

$$\text{(2.2)} \qquad \boldsymbol{y}^T P \hat{\boldsymbol{\alpha}} = 0,$$

$$\text{(2.3)} \qquad \boldsymbol{0} \leq \boldsymbol{\alpha}_0 + P\hat{\boldsymbol{\alpha}} \leq \boldsymbol{c},$$

where $\boldsymbol{r}_0$ is the gradient of $F(\boldsymbol{\alpha})$ at $\boldsymbol{\alpha}_0$. Note that $\boldsymbol{y}^T\boldsymbol{\alpha}_0 = 0$ because $\boldsymbol{\alpha}_0$ is feasible. So far, $P$ is still a general matrix. It will be specified later either directly or indirectly by its orthogonal complement $Q$, where $P^TQ = 0$.

Consider $\boldsymbol{\alpha}_0$ as an intermediate solution of the dual QP problem and $P\hat{\boldsymbol{\alpha}}$ as an update. To compute an update, we choose $P = E_p$, where the columns of $E_p$ are all the $\boldsymbol{e}_i$ satisfying $0 < \boldsymbol{e}_i^T\boldsymbol{\alpha}_0 < c$ (the constraint $0 \leq \alpha_i \leq c$ is not active at $\boldsymbol{\alpha}_0$). If $\boldsymbol{e}_i^T\boldsymbol{\alpha}_0 = 0$ or $\boldsymbol{e}_i^T\boldsymbol{\alpha}_0 = c$ (the constraint $0 \leq \alpha_i \leq c$ is active at $\boldsymbol{\alpha}_0$), then respectively $\boldsymbol{e}_i$ or $-\boldsymbol{e}_i$ becomes one column of $Q$. Note that $P$ and $Q$ are simply the matrix representations of the indices of the points in $G$ and $\overline{G}$, or equivalently the indices of the inactive and active constraints at $\boldsymbol{\alpha}_0$. We extend the definition of $\overline{G}$ to incorporate the points $\boldsymbol{x}_i$, where $\boldsymbol{e}_i^T\boldsymbol{\alpha}_0 = c$. That is, if $\boldsymbol{x}_i \in \overline{G}$ then $\boldsymbol{\alpha}_0(i)$ can be either 0 or $c$. Define $\hat{H} = P^THP$, $\hat{\boldsymbol{r}_0} = P^T\boldsymbol{r}_0$, $\hat{\boldsymbol{\alpha}_0} = P^T\boldsymbol{\alpha}_0$, and so on so forth. With the above choice of $P$, $\hat{\boldsymbol{\alpha}}$ is determined by Problem (2.1) subject to (2.2) and (2.3), which is the dual QP problem projected onto the affine subspace determined by $\boldsymbol{\alpha_0}$ and $E_p$:

$$\text{(2.4)} \qquad \underset{\hat{\boldsymbol{\alpha}}}{\text{maximize}} \quad \hat{F}(\hat{\boldsymbol{\alpha}}) \equiv \hat{\boldsymbol{\alpha}}^T \hat{\boldsymbol{r}}_0 - \frac{1}{2}\hat{\boldsymbol{\alpha}}^T \hat{H} \hat{\boldsymbol{\alpha}}$$

subject to

$$\text{(2.5)} \qquad \hat{\boldsymbol{y}}^T \hat{\boldsymbol{\alpha}} = 0,$$

$$\text{(2.6)} \qquad \boldsymbol{0} \leq \hat{\boldsymbol{\alpha}}_0 + \hat{\boldsymbol{\alpha}} \leq \boldsymbol{c}.$$

Note that $\boldsymbol{0}$ and $\boldsymbol{c}$ are of size $p$. Since $P\hat{\boldsymbol{\alpha}}$ is orthogonal to the column space of $Q$, it does not affect any active constraints at $\boldsymbol{\alpha}_0$. In other words, the update determined by the above QP problem only improves the part of $\boldsymbol{\alpha}_0$ whose values are strictly between 0 and $c$. When $p$ is small, it is cheap to solve the above problem.

To deal with the equality constraint (2.5), Problem (2.4) is projected again onto the hyperplane determined by this constraint. We choose $Q = \hat{\boldsymbol{y}}$ and $P = I - \frac{\hat{\boldsymbol{y}}\hat{\boldsymbol{y}}^T}{p}$ for this projection. It is easy to see that the column spaces of $P$ and $Q$ are orthogonal complements to each other, and that $I - P$ is the orthogonal projector onto the column space of $Q$. To reuse the notation $\hat{\boldsymbol{\alpha}}$, in (2.4), (2.5) and (2.6), $\hat{\boldsymbol{\alpha}}$ is replaced with $P\hat{\boldsymbol{\alpha}}$, where $P = I - \frac{\hat{\boldsymbol{y}}\hat{\boldsymbol{y}}^T}{p}$. The resulting subproblem has a simpler form:

$$\text{(2.7)} \qquad \underset{\hat{\boldsymbol{\alpha}}}{\text{maximize}} \quad \hat{F}(\hat{\boldsymbol{\alpha}}) \equiv \hat{\boldsymbol{\alpha}}^T P^T \hat{\boldsymbol{r}}_0 - \frac{1}{2}\hat{\boldsymbol{\alpha}}^T P^T \hat{H} P \hat{\boldsymbol{\alpha}}$$

subject to

$$\text{(2.8)} \qquad \boldsymbol{0} \leq \hat{\boldsymbol{\alpha}}_0 + P\hat{\boldsymbol{\alpha}} \leq \boldsymbol{c}.$$

If $\hat{\boldsymbol{\alpha}}$ is the solution to the above problem, then the update is

$$E_p(I - \frac{\hat{\boldsymbol{y}}\hat{\boldsymbol{y}}^T}{p})\hat{\boldsymbol{\alpha}}.$$

Remember that in Problem (2.4) subject to (2.5) and (2.6), $P = E_p$ and $E_p\hat{\boldsymbol{\alpha}}$ gives the update (the notation $\hat{\boldsymbol{\alpha}}$ is reused in Problem (2.7) subject to (2.8)).

2.1.2. *Solving the Subproblem by the Conjugate Gradient Method.*

If the box constraint (2.8) turns out to be loose, i.e., not active, then the above problem is equivalent to a linear system

$$P^T\hat{H}P\hat{\boldsymbol{\alpha}} = P^T\hat{\boldsymbol{r}}_0,$$

or equivalently

$$P\hat{H}P\hat{\boldsymbol{\alpha}} = P\hat{\boldsymbol{r}}_0 \ (P \text{ is symmetric}). \tag{2.9}$$

Since $P\hat{H}P$ is symmetric and usually positive definite under the above assumption, this linear system could be solved numerically by the Conjugate Gradient (CG) method [**8**] [**22**]:

$$\begin{aligned}
\hat{\boldsymbol{\alpha}}^{(0)} &= 0, \quad \boldsymbol{\gamma}^{(0)} = P\hat{\boldsymbol{r}}_0, \quad \boldsymbol{\rho}^{(0)} = \boldsymbol{\gamma}^{(0)} \\
\lambda &= \frac{{\boldsymbol{\gamma}^{(l-1)}}^T\boldsymbol{\gamma}^{(l-1)}}{{\boldsymbol{\rho}^{(l-1)}}^T P\hat{H}P\boldsymbol{\rho}^{(l-1)}} \\
\hat{\boldsymbol{\alpha}}^{(l)} &= \hat{\boldsymbol{\alpha}}^{(l-1)} + \lambda\boldsymbol{\rho}^{(l-1)} \\
\boldsymbol{\gamma}^{(l)} &= \boldsymbol{\gamma}^{(l-1)} - \lambda P\hat{H}P\boldsymbol{\rho}^{(l-1)} \\
\mu &= \frac{{\boldsymbol{\gamma}^{(l)}}^T\boldsymbol{\gamma}^{(l)}}{{\boldsymbol{\gamma}^{(l-1)}}^T\boldsymbol{\gamma}^{(l-1)}} \\
\boldsymbol{\rho}^{(l)} &= \boldsymbol{\gamma}^{(l)} + \mu\boldsymbol{\rho}^{(l-1)}.
\end{aligned}$$

From the fact $P^2 = P$, it follows that the above CG iterations can be simplified as

$$\begin{aligned}
\hat{\boldsymbol{\alpha}}^{(0)} &= 0, \quad \boldsymbol{\gamma}^{(0)} = P\hat{\boldsymbol{r}}_0, \quad \boldsymbol{\rho}^{(0)} = \boldsymbol{\gamma}^{(0)} \\
\lambda &= \frac{{\boldsymbol{\gamma}^{(l-1)}}^T\boldsymbol{\gamma}^{(l-1)}}{{\boldsymbol{\rho}^{(l-1)}}^T \hat{H}\boldsymbol{\rho}^{(l-1)}} \\
\hat{\boldsymbol{\alpha}}^{(l)} &= \hat{\boldsymbol{\alpha}}^{(l-1)} + \lambda\boldsymbol{\rho}^{(l-1)} \\
\boldsymbol{\gamma}^{(l)} &= \boldsymbol{\gamma}^{(l-1)} - \lambda P(\hat{H}\boldsymbol{\rho}^{(l-1)}) \\
\mu &= \frac{{\boldsymbol{\gamma}^{(l)}}^T\boldsymbol{\gamma}^{(l)}}{{\boldsymbol{\gamma}^{(l-1)}}^T\boldsymbol{\gamma}^{(l-1)}} \\
\boldsymbol{\rho}^{(l)} &= \boldsymbol{\gamma}^{(l)} + \mu\boldsymbol{\rho}^{(l-1)}.
\end{aligned}$$

Note that $\hat{H}$ is a $p \times p$ matrix. Since $P = I - \frac{\hat{\boldsymbol{y}}\hat{\boldsymbol{y}}^T}{p}$, the multiplication of $P$ with a vector is cheap. It only involves two Level-1 (vector-vector) operations. The most expensive operation above is the Level-2 (matrix-vector) operation $\hat{H}\boldsymbol{\rho}^{(l-1)}$. When $p$ is small, it is cheap to compute the above operations and the storage requirement (one $p \times p$ matrix and four $p$-vectors) is also small.

However, to solve Problem (2.7) subject to (2.8) with the above algorithm, the box constraint (2.8) has to be considered. If $0 \leq (\hat{\boldsymbol{\alpha}_0})_i + (P\hat{\boldsymbol{\alpha}})_i \leq c$ becomes active during the Conjugate Gradient iterations, then its global index representation $\boldsymbol{e}_{l_i}$ is added to $Q$. At the same time, some old active constraints can be relaxed by moving their index representations from $Q$ to $P$, if doing so improves the objective function. After $P$ and $Q$ is updated, the subproblem (2.7) subject to (2.8) is reformulated correspondingly and solved again. In the following section, the stopping criteria and the criteria for relaxing active constraints are given. An important fact worth to point out is that $\hat{H}$ is not guaranteed to be in full rank. Therefore, we need to control the steps of the CG iterations to keep the solution from blowing up.

2.1.3. *The Optimality Conditions.*

Note that Problem (2.7) subject to (2.8) is the compact version of Problem (2.1) subject to (2.2) and (2.3) with

$$\begin{aligned} Q &= [\pm\boldsymbol{e}_{i_1}, \pm\boldsymbol{e}_{i_2}, \ldots, \pm\boldsymbol{e}_{i_q}, \boldsymbol{y}] \\ &= [E_q, \boldsymbol{y}] \end{aligned}$$

and

$$P = I - Q(Q^TQ)^{-1}Q^T,$$

where $E_q$ is the orthogonal complement of $E_p$ $(m = p + q)$, and the column space of $P$ is the orthogonal complement of the column space of $Q$. In other words, the previous two projections are incorporated in the current definition of $P$. Let us define

$$\boldsymbol{d} = P\boldsymbol{r}$$

and

$$\boldsymbol{\beta} = -[I_{q\times q}, \boldsymbol{0}](Q^TQ)^{-1}Q^T\boldsymbol{r},$$

where $\boldsymbol{r} = \boldsymbol{1} - H\boldsymbol{\alpha}$ and $P$ is a $n \times n$ matrix. The optimality conditions [**1**] [**3**] tell that $\boldsymbol{\alpha}$ is optimal iff

$$\boldsymbol{d} = \boldsymbol{0} \text{ and } \boldsymbol{\beta} \geq \boldsymbol{0}. \tag{2.10}$$

Note that $\boldsymbol{d}$ is the projected gradient onto the search space spanned by $P$. If $\boldsymbol{d} = \boldsymbol{0}$ , then we have reached a stationary point. The condition $\boldsymbol{\beta} \geq 0$ implies that relaxing the active constraints represented by $Q$ does not improve the objective function nearby. It follows that if (2.10) is true then $\boldsymbol{\alpha}$ is a local optimum, thus a global optimum due to the convex property of a linearly constrained QP problem.

Since $(Q^TQ)^{-1}$ has a simple closed form:

$$\begin{bmatrix} I + \frac{E_q^T\boldsymbol{y}\boldsymbol{y}^TE_q}{p} & \frac{-E_q^T\boldsymbol{y}}{p} \\ \frac{-\boldsymbol{y}^TE_q}{p} & \frac{1}{p} \end{bmatrix},$$

it is easy to compute $d$ and $\boldsymbol{\beta}$. For instance,

$$\boldsymbol{\beta} = E_q^T\boldsymbol{r} + \frac{E_q^T\boldsymbol{y}((E_q^T\boldsymbol{y})E_q^T\boldsymbol{r} - \boldsymbol{y}^T\boldsymbol{r})}{p}, \tag{2.11}$$

where $E_q$ is an indexing operator. Each time when active constraints are to be relaxed, we start with the one that has the most negative $\beta_i$.

2.1.4. *The Algorithm.*

As a summary of the previous discussion, the algorithm is outlined briefly as the following:

$\boldsymbol{\alpha} = \mathbf{0}$;
initialize $E_p$;
**while** $\boldsymbol{d} \neq \mathbf{0}$ or $\boldsymbol{\beta} < \mathbf{0}$
  at most $k$ steps of the CG iterations for Problem (2.7) subject to (2.8);
  update $\boldsymbol{\alpha}$ and $\boldsymbol{r}$:

$$\boldsymbol{\alpha} = \boldsymbol{\alpha} + E_p(I - \frac{\hat{\boldsymbol{y}}\hat{\boldsymbol{y}}^T}{p})\hat{\boldsymbol{\alpha}};$$

$$\boldsymbol{r} = \boldsymbol{r} - HE_p(I - \frac{\hat{\boldsymbol{y}}\hat{\boldsymbol{y}}^T}{p})\hat{\boldsymbol{\alpha}};$$

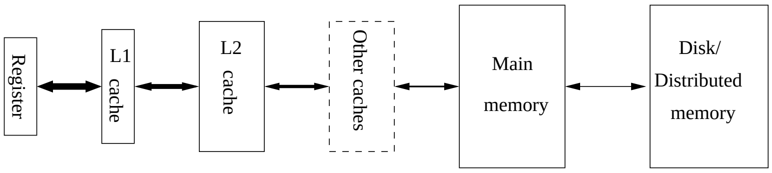
  compute $\boldsymbol{\beta}$;
  relax at most $l$ active constraints with the most negative $\beta_i$;
  update $E_p$;
  compute the columns of $H$ corresponding to the relaxed constraints;
  update $HE_p$ and $\hat{H}$;
**end**

During each iteration of this algorithm, only $p$ columns of $H$ ($HE_p$) are used. Thus, there is no need to precompute $H$ so that a lot of flops (floating point operations) and memory can be saved. In our implementation of this algorithm, two matrices $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$ and $HE_p$ are stored in memory. Since $n$ and $p$ are generally much smaller than $m$, most modern computers can meet the memory requirement for problems of size $m = O(10,000)$. Later in this paper, we will show an alternative which consumes less memory by only using part of $HE_p$, but taking more steps to converge.

Note that the $i$th column of $H$ is computed only when $\alpha_i = 0$ or $\alpha_i = c$ is relaxed. It turns out that computing the columns of $H$ is the most expensive operation in this algorithm. When $l$ is large, it can be considered as a Level-3 (matrix-matrix multiplication) operation. Although Level-1 and Level-2 operations are cheaper in terms of flops, Level-3 operations can be implemented more efficiently by exploiting the computer memory hierarchy. More discussions will be given on the implementation of these operations in the next section.

The setting of parameters $k$ and $l$ affects the performance. We will discuss more on how to choose $k$ and $l$ in the next section. In our implementation, $k$ and $l$ are determined adaptively based on each training set. To initialize $G$, we can randomly choose a small number of training points from the training set. An alternative is to compute an initial guess of the pair of separating hyperplanes and choose the points near the two hyperplanes as the candidates for $G$.

Note that the residual or the gradient $\boldsymbol{e}_i^T\boldsymbol{r} = 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i$ tells how well the current solution separates $\boldsymbol{x}_i$. When (2.10) is true, the optimality is achieved. As we have seen, this algorithm successfully exploits the sparsity property of SVMs. When the final solution is sparse, it solves the dual QP problem with much less flops and memory consumption. Meanwhile, this algorithm can be easily implemented




| *level* | *clock cycles* |
|---|---|
| register | 1 |
| L1 cache | $2-3$ |
| L2 cache | $6-12$ |
| near main memory | $6-100$ |
| far main memory | $100-250$ |
| distributed main memory | $O(100)$ |
| message-passing | $O(1000)-O(10000)$ |

FIGURE 3. A model of the modern computer memory hierarchy. The statistics in the table above are cited from [**7**].

by MATLAB. For details of our MATLAB implementation, please refer to the downloadable codes [**25**].

If each subproblem is solved exactly, then the objective function is improved strictly each time. However, from the performance point of view, it is better off to solve each subproblem approximately except the last one. Issues related to further speeding up the convergence of this algorithm are addressed in the following section.

### 2.2. Speed Considerations.

The sparsity property of SVMs enables us to avoid unnecessary memory consumption and flops. To achieve better performance, in this section we discuss how to make this algorithm adaptive to each computer's memory hierarchy and each training problem.

2.2.1. *Being Adaptive to the Computer Memory Hierarchy.*

When large data sets are involved in computations, the number of flops is not an accurate indicator of the running time. The cost of accessing data must also be taken into account. In Figure 3, a model of modern computer memory hierarchy is given. Loading data from main memory to caches and writing data from caches back to main memory are expensive operations. To reduce the cost of accessing data, computations should be arranged so that more data when needed can be found at the top (left) levels of the hierarchy. Decomposing a large problem into smaller subproblems is a good strategy. It is well known that Level-3 operations can be implemented more efficiently than Level-1 and Level-2 operations because of the above model.

In general, fast codes are developed by explicitly using the information of each configuration of the above model. To achieve portable performance, we want our implementation to be adaptive to each machine's memory hierarchy as much as possible. Since the basic operations in our algorithm are BLA operations, it is important to implement them adaptively. To achieve this goal, we use ATLAS BLAS [**26**] [**27**] [**28**], a software package for BLA operations that is automatically tuned for each machine. Thanks to MATLAB's inclusion of ATLAS BLAS starting from its version 6. Better performance and portability can be achieved now from MATLAB.

2.2.2. *Setting **k** and **l** Adaptively.*

As mentioned in the previous section, the running time of this algorithm depends on the setting of $k$ and $l$. In this section, we discuss how to set these two parameters adaptively for each training problem.

The convergence rate of this algorithm is related to the spectrum of $H$ and the value of $c$. Generally, the leading eigenvalues of $H$ are well separated. The number of these leading eigenvalues is a good indicator of the size of $S_1$. While the value of $c$ affects the size of $S_2$. The smaller $c$ is, the larger $S_2$. For the extreme case when the training set is separable and $c$ is set very large (such that the constraints $\alpha_i \leq c$ never become active), the spectrum of $H$ is an important factor determining the running time. Based on this observation, we set $k$ and $l$ adaptively using the information of the spectrum of $H$.

From the convergence property of the CG method, we know that when using the CG method to solve a symmetric positive definite linear system with $j$ well-separated leading eigenvalues, the residual tends to be reduced quickly during the first $j$ iterations. According to this property, we estimate the number of leading eigenvalues by $\hat{k}$, the number of iterations for the CG method to reduce the residual norm of the first subproblem to a scale of 0.01.

We use $k$ to control the steps of the CG iterations, because there is no need to compute Problem (2.7) subject to (2.8) exactly if it is not the final subproblem. Another reason to not solve every subproblem exactly is due to the rank deficiency of $H$. The CG method converges slowly or may not even converge for ill-conditioned matrices. $k$ should be bounded by the rank of $H$. For example, if $\boldsymbol{x}_i \in \mathbb{R}^2$ then $\hat{H}$ has two nonzero eigenvalues in general. It makes sense to solve the subproblems with only two steps of the CG iterations before the solution blows up. In our algorithm, $k$ is set to be $\hat{k}$ obtained from the initial subproblem.

When $l$ active constraints are relaxed, $[\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{i_l}]^T X$ is computed to generate the corresponding columns of $H$ ($H$ is not precomputed). Considering that $[\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{i_l}]^T X$ is a Level-3 operation when $l$ is large, we tend to relax all the active constraints with negative $\beta_i$ (because Level-3 operations can be implement efficiently). However, with this strategy we increase the probability that some previously relaxed constraints become active again, which may slow down the convergence. Our observation is that when the number of the SVs in $S_1$ is small, it is better off to set $l$ small. Since the number of $H$'s leading eigenvalues provides information about the size of $S_1$, in our algorithm $l$ is also set to be $\hat{k}$.

## 3. Numerical Experiments

The MATLAB implementation of this algorithm is named “FMSvm”, where “FM” stands for “Fast MATLAB”. In this section, FMSvm [**25**] is compared with two bench mark C and C++ codes, SVM$^{light}$ and SvmFu respectively.

We use two training sets Digit17 and Face to compare the convergence rates of these three training codes. Digit17 is a sparse training set containing 13007 samples of handwritten digits 1 and 7, which are obtained from the MNIST handwritten-digit database. This database has become a standard for testing and comparing different learning algorithms. The other training set is a face detection training set obtained from MIT AI Lab, which is dense and contains 31022 examples. The face

| Training sets: Digit17-6000 and Digit17 | | | | | | | |
|---|---|---|---|---|---|---|---|
| l(1) | l(0.1) | p2(0.1) | p2(0.001) | r10(10) | r10(1) | r3(1) | r3(0.1) |
| Training sets: Face-6000, Face-13901 and Face | | | | | | | |
| l(4) | l(1) | l(0.1) | p2(1) | p2(0.001) | r10(10) | r3(10) | r3(1) |

TABLE 1. Choices of kernels and $c$. Here, "l" stands for the linear kernel, "p2" stands for the polynomial kernel with degree 2 and "r$x$" stands for the radial basis function kernel with $\sigma = x$. The number in the parenthesis is the value of $c$. The definitions of these kernel functions are listed in the appendix.

detection problem is interesting because it represents a class of similar problems, such as how to detect cancers in medical images and how to detect enemy tanks in satellite images. Digit17 is easy to separate and it is balanced (6742 positive examples vs. 6265 negative examples). While Face is harder to separate and unbalanced (2901 positive examples vs. 28121 negative examples). The training points in Digit17 and Face are of dimensions 784 and 361 respectively. The two training set are downloadable at [**25**].

Both the SVM$^{light}$ and SvmFu training functions have parameters whose setting affects their running time[1]. Bad choices can make the convergence very slow. Unfortunately, the optimal setting of these parameters are not known a priori. If they are not set, the default setting is taken. The FMSvm training function also has two parameters $k$ and $l$, but users are not required to set them. Instead, it tries to estimate the optimal setting by itself based on each particular training problem. In this section we compare FMSvm with SVM$^{light}$ and SvmFu using both default and estimated optimal settings.

From Digit17 and Face, we create five training sets: Digit17-6000, Digit17, Face-6000, Face-13902 and Face, with sizes 6000, 13007, 6000, 13902 and 31022 respectively. The two smallest training sets are used to obtain an estimate of the optimal settings for SVM$^{light}$ and SvmFu. The estimated optimal setting for each training function is the setting among ten trials which gives the best performance. Then on larger training sets Digit17, Face-13902 and Face, FMSvm is compared against SVM$^{light}$ and SvmFu with the estimated optimal settings. For each training set, different kernel functions and choices of $c$ as listed in Table 1 are used for the comparison.

In Figure 4, we can see that for the easy-to-separate training set Digit17-6000, there is no big difference in performance between FMSvm and the other two with default settings. But for Face-6000 's case l(4), FMSvm does much better than both SVM$^{light}$ and SvmFu. The timing results of FMSvm against SVM$^{light}$ and SvmFu plotted in the right column of Figure 4 are for the case where the estimated optimal settings are used for all of them. The estimated optimal settings obtained here for SVM$^{light}$ and SvmFu are used for later comparisons shown in Figure 5 and Figure 6. These estimated optimal settings are listed in the appendix.

[1]The convergence time is measured by cpu seconds. For SVM$^{light}$ and SvmFu, the time spent in loading the input data file is excluded. For instance, SvmFu needs around 24 cpu seconds to load the training set Face, while SVM$^{light}$ needs around 115 cpu seconds. All the timing results in this paper are obtained on the sever `newton.mit.edu`, which is a linux machine with 2 1.2 *GHZ* Athlon MP processors and 2 *GB* memory.

**Using Default Settings** **Using Estimated Optimal Settings**

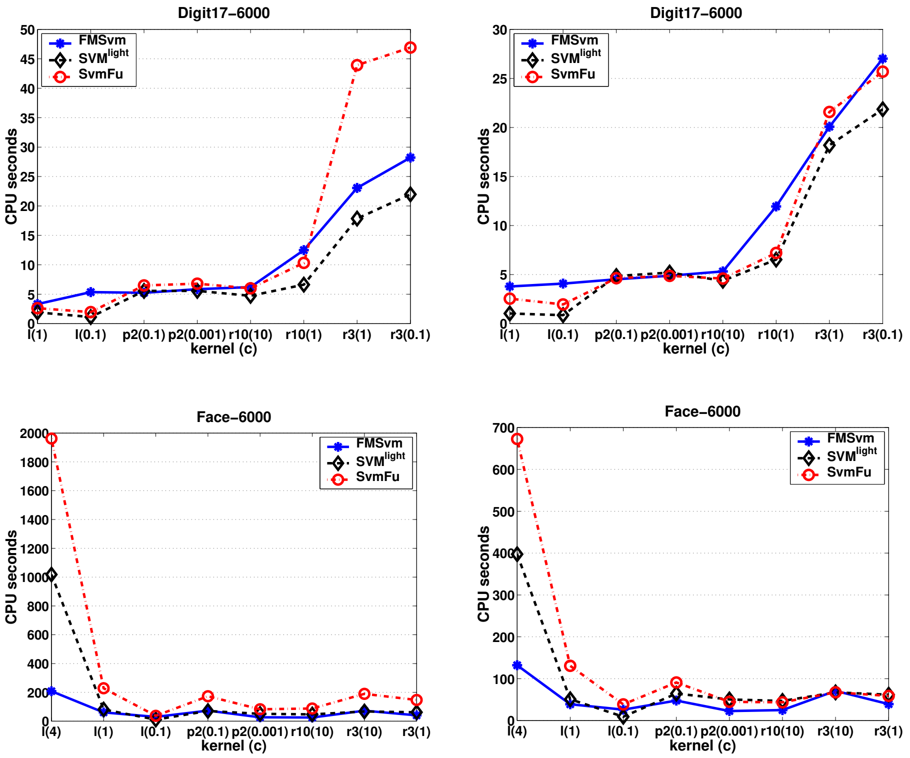



FIGURE 4. Timing results on the two training subsets of size 6000. Plots in the left column are the timing results of FMSvm against SVM$^{light}$ and SvmFu with default parameter settings. For problems of size larger than 2000, the default setting for SvmFu is "(h, c)" $=$ $(2000, 0)$ and the default setting for SVM$^{light}$ is "(q, n, m)" $=$ $(10, 10, 40)$. However, for training set Face-6000 we use $(20, 10, 40)$ as SVM$^{light}$'s default setting, following the hint to choose $n < q$ to prevent zig-zagging, where $(20, 10, 40)$ is the estimated optimal setting for Digit17-6000. Note that in SVM$^{light}$ and SvmFu, $h, c$ and $q, n, m$ have different meanings. Plots in the right column are the timing results of FMSvm against SVM$^{light}$ and SvmFu, where the estimated optimal settings are used for all of them. With default settings, for the easy-to-separate training set Digit17-6000, there is no big difference in performance between the FMSvm training function and the other two. But for Face-6000 's case l(4), FMSvm does much better than both SVM$^{light}$ and SvmFu.

**Comparison of FMSvm with SVM$^{light}$ and SvmFu with the estimated optimal parameter settings**

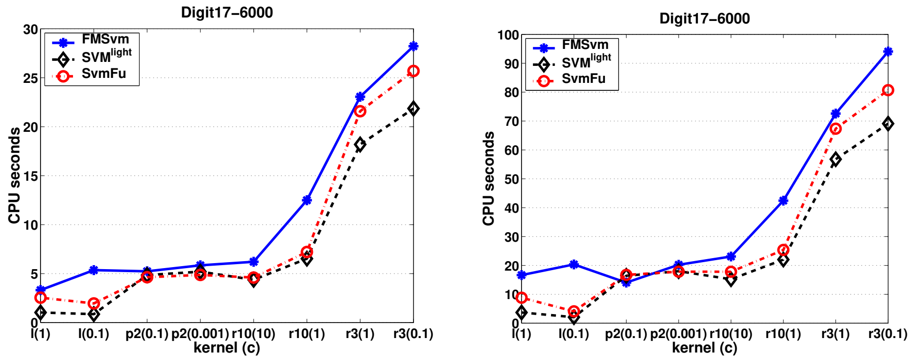



FIGURE 5. FMSvm VS. SVM$^{light}$ and SvmFu. The estimated optimal parameter settings are used for SVM$^{light}$ and SvmFu. For those parameters that seem to depend on the size of the training set, their values are adjusted proportionally as the size increases. All the settings are listed in the appendix. We can see that FMSvm (with parameters set automatically) matches the estimated best performance of SVM$^{light}$ and SvmFu on the training sets Digit17-6000 and Digit17.

Since the optimal parameter settings for all the three training functions are not known a priori, the default setting or a setting from an arbitrary guess may cause very slow convergence. FMSvm tries to avoid this situation by being adaptive to each training problem. Shown in Figure 5 and Figure 6 are the timing results of FMSvm against SVM$^{light}$ and SvmFu, where the estimated optimal settings are used for SVM$^{light}$ and SvmFu. We can see that FMSvm matches the estimated best performance of SVM$^{light}$ and SvmFu on the two training sets Digit17 and Face.

If we measure the difference between a setting $A$ and the estimated optimal setting $O$ by the ratio of relative slowdown:

$$s = \frac{\mathrm{time}(A) - \mathrm{time}(O)}{\mathrm{time}(O)},$$

then in Figure 4, for each training function we get 16 such ratios that measure the difference between the setting used in the left column and the estimated optimal setting used in the right column. The mean and standard deviation of these ratios are respectively $(0.16, 0.14)$, $(0.24, 0.43)$ and $(0.76, 0.60)$ for FMSvm, SVM$^{light}$ and SvmFu. We can see that the parameter setting used by FMSvm is closer to the corresponding estimated optimal.

In summary, FMSvm's idea of being adaptive to each training problem works, which helps preventing the slow convergence caused by bad parameter settings. Based on training sets Digit17 and Face, we can see that FMSvm matches the best performance of SVM$^{light}$ and SvmFu, while having the features provided by a high-level programming language.

**Comparison of FMSvm with $\mathrm{SVM}^{light}$ and SvmFu with the estimated optimal parameter settings**

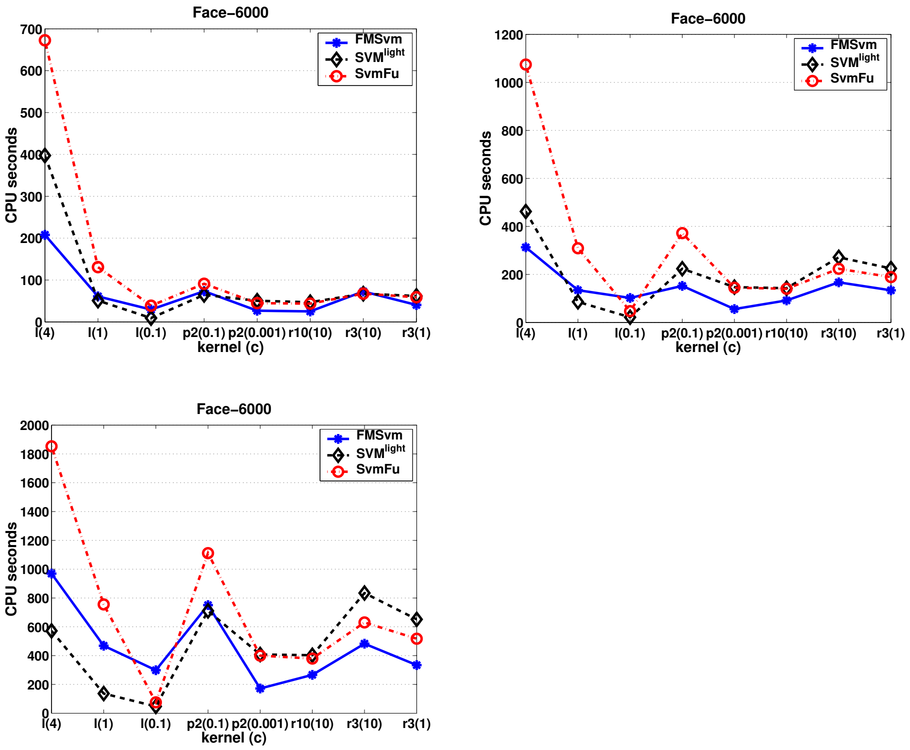



FIGURE 6. FMSvm VS. $\mathrm{SVM}^{light}$ and SvmFu. The estimated optimal parameter settings are used for the training functions of $\mathrm{SVM}^{light}$ and SvmFu. For those parameters that seem to depend on the size of the training set, their values are adjusted proportionally as the size increases. All the settings are listed in the appendix. We can see that FMSvm (with parameters set automatically) matches the estimated best performance of $\mathrm{SVM}^{light}$ and SvmFu on the training sets Face-6000, Face-13901 and Face.

## 4. Distributing Large Training Problems

The basic memory requirement for our algorithm is to store the training example matrix $X$ and the computed part of the Hessian matrix $HE_p$. Instead of $E_p^T HE_p$, $HE_p$ is stored because we need to update the residual

$$r = r_0 - HE_p(I - \frac{\hat{\boldsymbol{y}}\hat{\boldsymbol{y}}^T}{p})\hat{\alpha}.$$

To control memory consumption, we can update part of the residual so that only the corresponding rows of $HE_p$ are needed. From the primal problem point of view, this is equivalent to testing part of the training points in $\overline{G}$ using the current solution. This approach can be easily extended to distribute large training problems. The idea is to use the master process to compute the subproblems, where only $E_p^T H E_p$ is needed, and let the slave processes to update the residual in parallel. Updating the residual is expensive because the corresponding part of $H$ has to be computed.

To make our MATLAB code support distributed computations, we use an extended MATLAB environment called MATLAB*P [**9**], where an almost transparent interface to distributed matrices and the operations on them is provided. MATLAB*P consists of two parts, a MATLAB front end and a server living in a supercomputer. Distributed matrices and the operations on them are stored and executed respectively on the server. Unlike MATLAB's transparent support to sparse matrices, small modification is required to make ordinary MATLAB codes to support MATLAB*P's distributed matrices. But once MATLAB*P knows that $X$ is a distributed matrix, all the operations involving $X$ are automatically executed by the server. We expect that extending FMSvm to support MATLAB*P's distributed matrices should not be a difficult task.

## 5. Conclusion and Future work

While successfully exploiting the sparse property of SVMs, this algorithm also provides the capability to be adaptive to the training problem and the computer memory hierarchy. The resulting implementation is not only efficient but also highly portable and easy-to-use. We expect that further performance improvement can be achieved using preconditioning techniques. This could be a feature for the next version of FMSvm.

A rigorous analysis of the convergence behavior of this algorithm is subject to future study. The techniques and heuristics employed here to speed up the convergence make this analysis hard. Although FMSvm works well on the training sets we have tried, so far we can not guarantee that this algorithm converges for all the cases. In our future work, we want to better understand the convergence behavior of this algorithm and to make it more adaptive. We also want to test this algorithm with more training problems and on more computer platforms.

## Appendix A. List of Kernel Functions

- Linear kernel: $k(\boldsymbol{s},\boldsymbol{t}) = \boldsymbol{s}^T\boldsymbol{t}$, $\boldsymbol{s},\boldsymbol{t} \in \mathbb{R}^n$.
- Polynomial kernel: $k(\boldsymbol{s},\boldsymbol{t}) = (\boldsymbol{s}^T\boldsymbol{t} + b)^d$, $\boldsymbol{s},\boldsymbol{t} \in \mathbb{R}^n$ and $b,d \in \mathbb{R}$.
- Radial basis function kernel: $k(\boldsymbol{s},\boldsymbol{t}) = e^{-\|\boldsymbol{s}-\boldsymbol{t}\|^2/(2\sigma^2)}$, $\boldsymbol{s},\boldsymbol{t} \in \mathbb{R}^n$ and $\sigma \in \mathbb{R}$.

## Appendix B. List of estimated optimal settings for SVM$^{light}$ and SvmFu

- Table 2: The Estimated Optimal Settings for Training Sets Digit17-6000 and Digit17.
- Table 3: The Estimated Optimal Settings for Training Sets Face-6000, Face-13901 and Face.

## References


[1] Mordecai Ariel. *Nonlinear Programming: Analysis and Methods.* Prentice-Hall, N.J., 1976.

[2] D. Bertsekas. *Nonlinear Programming.* Athena Scientific, Belmont, 1995.

[3] John C. G. Boot. *Quadratic Programming.* Rand McNally & Company, Chicago, 1964.

| *Kernels* | *Digit17*-6000 | | *Digit17* | |
|---|---|---|---|---|
| ($c$) | SVM$^{light}$3.5 | SvmFu3.0 | SVM$^{light}$3.5 | SvmFu3.0 |
| l(1) | $(20, 10, 40)$ | $(2000, 100)$ | $(20, 10, 80)$ | $(2000, 200)$ |
| l(0.1) | $(20, 10, 40)$ | $(2000, 0)$ | $(20, 10, 80)$ | $(2000, 0)$ |
| p2(0.1) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13007, 0)$ |
| p2(0.001) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13007, 0)$ |
| r10(10) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13007, 0)$ |
| r10(1) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13007, 0)$ |
| r3(1) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13007, 0)$ |
| r3(0.1) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13007, 0)$ |

TABLE 2. The Estimated Optimal Settings for Training Sets Digit17-6000 and Digit17.

| *Kernels* | *Face*-6000 | | *Face*-13901 | | *Face* | |
|---|---|---|---|---|---|---|
| ($c$) | SVM$^{light}$3.5 | SvmFu3.0 | SVM$^{light}$3.5 | SvmFu3.0 | SVM$^{light}$3.5 | SvmFu3.0 |
| l(4) | $(60, 20, 40)$ | $(6000, 0)$ | $(60, 20, 80)$ | $(13901, 0)$ | (60,20,200) | (31022,0) |
| l(1) | $(60, 20, 40)$ | $(6000, 0)$ | $(60, 20, 80)$ | $(13901, 0)$ | (60,20,200) | (31022,0) |
| l(0.1) | $(40, 10, 40)$ | $(2000, 40)$ | $(40, 10, 80)$ | $(2000, 80)$ | (40,10,200) | (2000,200) |
| p2(1) | $(40, 10, 40)$ | $(6000, 0)$ | $(40, 10, 80)$ | $(13901, 0)$ | (40,10,200) | (31022,0) |
| p2(0.001) | $(60, 20, 40)$ | $(6000, 0)$ | $(60, 20, 80)$ | $(13901, 0)$ | (60,20,200) | (31022,0) |
| r10(10) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13901, 0)$ | (20,10,200) | (31022,0) |
| r3(10) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13901, 0)$ | (20,10,200) | (31022,0) |
| r3(1) | $(20, 10, 40)$ | $(6000, 0)$ | $(20, 10, 80)$ | $(13901, 0)$ | (20,10,200) | (31022,0) |

TABLE 3. The Estimated Optimal Settings for Training Sets Face-6000, Face-13901 and Face.


[4] L. Bottou, C. Cortes, J. Denker, H. Drucher, I. Guyon, L. Jackel, Y. LeGun, U. Müller, E. Säckinger, P. Simard and V. Vapnik. Comparison of Classifier Methods: a Case Study in Handwritten Digit Recognition. *Proceedings of the 12th International Conference on Pattern Recognition and Neural Networks, Jerusalem*, 77-87. IEEE Computer Society Press, 1994.

[5] Christopher J.C. Burges. *A Tutorial on Support Vector Machines for Pattern Recognition*. Knowledge Discovery and Data Mining, 2(2), 1998.

[6] Nello Cristianini, John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[7] Craig C. Douglas, Gundolf Haase, Jonathan Hu, Markus Kowarschik, Ulrich Rüde and Christian Weiss. *Portable Memory Hierarchy Techniques For PDE Solvers: Part I*. SIAM NEWS, Volume 33/Number 5, June 2000.

[8] Gene H. Golub, Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, 1996.

[9] Parry Husbands. *Interactive Supercomputing*. Ph.D Thesis, M.I.T., 1999.

[10] Thorsten Joachims. $SVM^{light}$ 3.5. `http://svmlight.joachims.org`.

[11] Thorsten Joachims. Making Large-Scale Support Vector Machine Learning Practical. *Advances in Kernel Methods: Support Vector Learning*, edited by Bernhard Schölkopf, Christopher J.C. Burges and Alexander J. Smola, The MIT Press, Cambridge, 1998.

[12] Linda Kaufman. Solving the Quadratic Programming Problem Arising in Support Vector Classification. *Advances in Kernel Methods: Support Vector Learning*, edited by Bernhard Schölkopf, Christopher J.C. Burges and Alexander J. Smola, The MIT Press, Cambridge, 1998.

[13] M. Kirby, L. Sirovich. Application of the Karhunen-Loève Procedure for the Characterization of Human Faces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):103-108, 1990.

[14] U. Krebel. The Impact of the Learning-Set Size in Handwritten Digit Recognition. *Artificial Neural Networks – ICANN'91*, edited by T. Kohonen, 1685-1689, Amsterdam, 1991. North-Holland.

[15] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1:541-551, 1989.

[16] Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Müller, E. Säckinger, P. Simard and V. Vapnik. Comparison of Learning Algorithms for Handwritten Digit Recognition. *Proceedings ICANN'95 – International Conference on Artificial Neural Networks*, edited by F. Fogelman-Soulié and P. Gallinari. Volume II, 53-60, Nanterre, France, 1995.

[17] E. Osuna, R. Freund and F. Girosi. An Improved Training Algorithm for Support Vector Machines. *Neural Networks for Signal Processing VII – Proceeding of the 1997 IEEE Workshop*, edited by J. Principe, L. Gile, N. Morgan and E. Wilson, 511-520, New York, 1997. IEEE.

[18] E. Osuna, R. Freund and F. Girosi. Training Support Vector Machines: An Application to Face Detection. *Proceedings, Computer Vision and Pattern Recognition'97*, 130-136, 1997.

[19] John C. Platt. Fast Training of Support Vector Machines Using Sequential Minimal Optimization. *Advances in Kernel Methods: Support Vector Learning*, edited by Bernhard Schölkopf, Christopher J.C. Burges and Alexander J. Smola, The MIT Press, Cambridge, 1998.

[20] Ryan Rifkin. *SvmFu* 3.0. `http://fpn.mit.edu/SvmFu/`.

[21] Edited by Bernhard Schölkopf, Christopher J.C. Burges and Alexander J. Smola. *Advances in Kernel Methods: Support Vector Learning.* The MIT Press, Cambridge, 1999.

[22] Lloyd N. Trefethen, David Bau, III. *Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[23] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory.* Springer Verlag, New York, 1995.

[24] Vladimir N. Vapnik. *Statistical Learning Theory.* John Wiley & Sons, Inc, 1998.

[25] Tong Wen. *FMSvm* 1.0. `http://math.mit.edu/~tonywen/FMSvm/`.

[26] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). *SC '89 Proceedings* (Electronic Publication). IEEE Publication.

[27] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, Volume 27, Numbers 1-2, pp 3-25, 2001.

[28] R.Clint Whaley, Antoine Petitet, Jack J. Dongarra. *Automated Empirical Optimization of Software and the ATLAS Project.* `http://math-atlas.sourceforge.net/`.

Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139

*E-mail address*: `tonywen@math.mit.edu`

Department of Mathematics & Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139

*E-mail address*: `edelman@math.mit.edu`

U.S. Army Tank-automotive & Armaments Command, Automotive Research Center, Warren, MI 48397

*E-mail address*: `gorsichd@tacom.army.mil`