# Efficient Large Scale Linear Programming Support Vector Machines

Suvrit Sra

Dept. of Comp. Sciences, The University of Texas at Austin
Austin, TX 78712, USA

**Abstract.** This paper presents a decomposition method for efficiently constructing $\ell_1$-norm Support Vector Machines (SVMs). The decomposition algorithm introduced in this paper possesses many desirable properties. For example, it is provably convergent, scales well to large datasets, is easy to implement, and can be extended to handle support vector regression and other SVM variants. We demonstrate the efficiency of our algorithm by training on (dense) synthetic datasets of sizes up to 20 million points (in $\mathbb{R}^{32}$). The results show our algorithm to be several orders of magnitude faster than a previously published method for the same task. We also present experimental results on real data sets—our method is seen to be not only very fast, but also highly competitive against the leading SVM implementations.

## 1 Introduction

Traditionally Support Vector Machines (SVMs) are constructed by maximizing an $\ell_2$-norm margin, which is achieved by solving an associated quadratic program. Researchers have also looked at maximizing margins measured using other $\ell_p$ norms [1]—most notably the $\ell_1$ and $\ell_\infty$ norms, both of which lead to linear programming formulations[1]. The book chapter by Bennett [1] lists some further useful references related to such formulations. Some recent relevant papers studying the $\ell_1$-norm SVM are [2, 13, 19].

Most work on SVMs, however, ends up focusing on the details of the $\ell_2$-norm SVM, relegating the solution of the $\ell_1$ (or $\ell_\infty$) norm SVM to an off-the-shelf linear programming (LP) solver such as CPLEX$^{\text{TM}}$. For real world data, especially for large-scale data, such an approach can be very expensive, if not impractical. The $\ell_2$-SVM has on the other hand witnessed a lot of research and efficient implementations for solving it are available (e.g., SVM$^{\text{light}}$ [10], SMO [16], LIBSVM [4]). It is desirable that some of the algorithmic progress made for the $\ell_2$-SVM be carried over to the $\ell_1$-SVM too.

We are aware of one previous work, namely that of Bradley and Mangasarian [2] that attempts to make learning $\ell_1$-SVMs practical for large data sets. These authors introduce a method called Linear Programming Chunking (LPC) that

[1] In general, maximizing margin using an $\ell_q$ norm can be done by minimizing $\|\boldsymbol{w}\|_p$, where $\frac{1}{p} + \frac{1}{q} = 1$.

decomposes a linear program into smaller chunks and solves them using any LP solver. However, despite its efficiencies, LPC can be prohibitively slow for large problems. Thus, an efficient method for solving LP based SVM formulations is needed, and this paper presents such a method. Our approach yields a scalable and efficient decomposition method for solving the $\ell_1$-SVM, which is several orders of magnitude faster than the LPC approach and makes learning large scale $\ell_1$-SVMs practical. Furthermore, our method is simple to implement, provably convergent, easily extensible to solve other SVM variants, and yields accuracies competitive with well established SVM software.

Much of the speed of our algorithm lies in the fact that we solve the primal as opposed to the dual formulation of the $\ell_1$-SVM.[2] Recently Chapelle [5] has provided motivation for reconsidering the solution of the primal formulation of the SVM. Since many years, owing to its simplicity and extensibility to nonlinear cases, researchers have focused on solving the dual problem for SVMs. For example, when number of training points greatly exceeds the dimensionality of a single point, it is advantageous to solve the primal rather than the dual (despite the novel efficiencies introduced in [14]).

## 2 The $\ell_1$-Norm SVM

As per the standard two-class classification problem, we assume the input to be the set $\{(\boldsymbol{x}_i, y_i) : 1 \le i \le N\}$, where $\boldsymbol{x}_i \in \mathbb{R}^M$ are the training points, and $y_i \in \{\pm 1\}$ are their associated class labels. The aim is to learn a function or classifier $f(\boldsymbol{x})$ such that given a new data point $\boldsymbol{x}$, we can accurately predict its class label. A linear classifier is commonly constructed by computing a function $f(\boldsymbol{x}) = \operatorname{sgn}(\boldsymbol{w}^T\boldsymbol{x} + b)$. The corresponding $\ell_1$-SVM problem may be written as

$$\begin{aligned} &\min_{\boldsymbol{w},b} \ \|\boldsymbol{w}\|_1 + C\sum_i \xi_i, \\ \text{subject to} \quad & y_i(\langle \boldsymbol{x}_i, \boldsymbol{w}\rangle + b) \ge 1 - \xi_i, \qquad \xi_i \ge 0, \quad 1 \le i \le N. \end{aligned} \tag{2.1}$$

The parameter $C$ is a cost (penalty) parameter and is provided as input (normally after having been determined using cross-validation). Observe that in (2.1) we seek to minimize $\|\boldsymbol{w}\|_1$ instead of $\|\boldsymbol{w}\|_2^2$, as is done in the traditional $\ell_2$-SVM. Minimizing $\|\boldsymbol{w}\|_1$ leads to sparser solutions, which in turn imply better dimension reduction, greater robustness, and faster classifiers [1, 2, 19].

We introduce an auxiliary variable $\boldsymbol{f} = |\boldsymbol{w}|$ (elementwise) to write (2.1) as the linear program (LP)

$$\begin{aligned} &\min_{\boldsymbol{w},b,\boldsymbol{f},\boldsymbol{\xi}} \ \mathbf{1}^T(\boldsymbol{f} + C\boldsymbol{\xi}) \\ &y_i(\langle \boldsymbol{x}_i, \boldsymbol{w}\rangle + b) \ge 1 - \xi_i, \quad 1 \le i \le N, \\ &-\boldsymbol{w} - \boldsymbol{f} \le \mathbf{0}, \quad \boldsymbol{w} - \boldsymbol{f} \le \mathbf{0}, \quad \boldsymbol{\xi} \ge \mathbf{0}. \end{aligned} \tag{2.2}$$

[2] Our approach has a more *primal-dual* flavor, but since we never form the dual, we continue referring to it as a *primal* approach.

For large-scale data solving the LP (2.2) using off-the-shelf software can be very expensive. Bradley and Mangasarian [2] described a method called Linear Programming Chunking for efficiently solving (2.2). However, despite their "chunking" approach, their method can still be extremely slow for large data sets. In Section 3 below, we describe a fast decomposition procedure for solving (2.2). We remark that our techniques can be easily adapted to solve the $\ell_\infty$-norm SVM. Furthermore, nonlinear SVMS via the LP-machines [8] can also be handled with equal ease. We omit the details due to space limitations (these will be published elsewhere).

## 3 Algorithm

It may not seem obvious how to solve (2.2) using a simple decomposition method. Problem (2.2) lacks strict convexity, a necessary ingredient for the application of many decomposition techniques. Fortunately, we can exploit a very useful result of Mangasarian [12, Theorem 2.1-a-i] (adapted as Theorem 1 below) that permits us to transform (2.2) into an equivalent quadratic program that has the necessary strict convexity.

**Theorem 1 ($\ell_1$ SVM).** *Let $\boldsymbol{g} = [\boldsymbol{w}; b; \boldsymbol{f}; \boldsymbol{\xi}]$ and $\boldsymbol{c} = [\boldsymbol{0}; 0; \boldsymbol{1}; C\boldsymbol{1}]$ be partitioned conformally. If* (2.2) *has a solution, then there exists an $\epsilon_0 > 0$, such that for all $\epsilon \le \epsilon_0$,*

$$\operatorname*{argmin}_{\boldsymbol{g}\in G} \ \|\boldsymbol{g} + \epsilon^{-1}\boldsymbol{c}\|_2^2 \quad = \quad \operatorname*{argmin}_{\boldsymbol{g}\in G^\star} \ \|\boldsymbol{g}\|_2^2, \tag{3.1}$$

*where $G$ is the feasible set for* (2.2) *and $G^\star$ is the set of optimal solutions to* (2.2). *The minimizer of* (3.1) *is unique.*

Theorem 1 essentially states that the solution of (3.1) yields the minimum $\ell_2$-norm solution out of all the possible solutions of (2.2). This seemingly counter-intuitive replacement of a linear program by a corresponding quadratic program lies at the heart of building a decomposition method for the $\ell_1$-SVM.

### 3.1 Decomposition

To permit a clearer description we rewrite (3.1) in the more explicit form

$$\min_{\boldsymbol{g}=[\boldsymbol{w};b;\boldsymbol{f},\boldsymbol{\xi}]} \ \tfrac{1}{2}\|\boldsymbol{g} - (-\tfrac{1}{\epsilon})\boldsymbol{c}\|^2, \tag{3.2}$$

$$\begin{array}{lrcll}
\text{subject to} & -y_i\boldsymbol{x}_i^T\boldsymbol{w} - y_i b + \boldsymbol{0}^T\boldsymbol{f} - \xi_i & \le & -1 & \text{(TR)}\\
& -w_i + 0b - f_i + 0\xi_i & \le & 0 & \text{(A1)}\\
& w_i + 0b + f_i + 0\xi_i & \le & 0 & \text{(A2)}\\
& 0w_i + 0b + 0f_i - \xi_i & \le & 0, & \text{(XI)}
\end{array}$$

where $\boldsymbol{g}$ and $\boldsymbol{c}$ are as in Theorem 1, and $1 \le i \le N$. Let $\boldsymbol{z}$ denote the vector of dual variables associated with the training (TR), absolute value (A1), (A2), and soft-margin (XI) constraints. Further, let $\boldsymbol{A}$ denote the matrix of all these constraints put together.

Let $L(\boldsymbol{g}, \boldsymbol{z})$ denote that Lagrangian for (3.2). A first order necessary condition of optimality is

$$\frac{\partial}{\partial \boldsymbol{g}} L(\boldsymbol{g}, \boldsymbol{z}) = \boldsymbol{g} + \epsilon^{-1}\boldsymbol{c} + \boldsymbol{A}^T\boldsymbol{z} = 0, \qquad \boldsymbol{z} \geq \mathbf{0}. \tag{3.3}$$

The decomposition procedure that we use consists of the following main steps:

1. Start with a dual feasible solution, and obtain a corresponding primal solution so that the first-order necessary conditions (3.3) are satisfied. For example, $\boldsymbol{z} = \mathbf{0}$ and $\boldsymbol{g} = -\epsilon^{-1}\boldsymbol{c}$ is a valid initialization.
2. Go through each constraint individually and enforce it. Enforcing each constraint is equivalent to updating the corresponding dual variable $z_j$ $(1 \leq j \leq 4N)$ so that $z_j \geq 0$ is maintained, while recomputing $\boldsymbol{g}$ to ensure that (3.3) remains satisfied.
3. Repeat Step 2 until some convergence condition is satisfied (such as small net violation of all the KKT constraints, change below a certain threshold to the objective function etc.).

This decomposition procedure is based upon Bregman's method [3][3], which is a generic decomposition method for minimizing a strictly convex function subject to linear inequality constraints. This procedure generates a sequence of primal $(\{\boldsymbol{g}^t\}_{t\geq 0})$ and dual $(\{\boldsymbol{z}^t\}_{t\geq 0})$ iterates that converge to the optimal solution of the associated problem (see [3, Chapter 6] for a proof). Pseudo-code and associated implementation details for this algorithm are omitted from this paper due to space limitations.

So far we have not remarked upon the selection of the parameter $\epsilon$. Two approaches are possible. One can test the accuracy on a hold-out subset of the training data and perform a search for a good value of $\epsilon$. One can also pick an $\epsilon$ from within a predefined range of values. The former approach might increase the running time (albeit minimally), whereas the latter is simple and fast. We tried both approaches, and found that usually a value of $\epsilon$ in [0.1–100] worked well (i.e., resulted in high training and test accuracy, as well as rapid convergence).

## 4 Experimental Results

In this section we describe some of the experiments that we performed to assess the quality of our implementation. We consider two types of experiments. The first type is on real data (Section 4.1), while the second is on synthetic data (Section 4.2). The purpose of the former is to illustrate that our implementation performs competitively on real world data sets when compared to some of the

[3] For the quadratic case, Bregman's method reduces to Hildreth's method [9], however we continue using the name Bregman's method to indicate that the same ideas could be applied to handle other convex penalties too.

leading SVM implementations. The latter set of experiments show two things: i) the efficiency of our $\ell_1$-norm SVM implementation, and ii) the importance of solving the *primal* problem for data with highly skewed dimensions.

We implemented our algorithm in C++ using the the sparse matrix library SSLib [18]. The experiments reported in Section 4.1 were performed on a Pentium 4, 3GHz Linux machine equipped with 1GB RAM, whereas those in Section 4.2 were performed on a Pentium Xeon 3.2GHz Linux machine with 8GB RAM.

## 4.1 Classification Experiments on Real Data

Below we report classification results for several real data sets. Table 1 presents these results, wherein we report both training and test accuracies, as well as the respective running times of the algorithms tested (excluding I/O). As attested to by the results, our $\ell_1$-SVM performs competitively against standard SVM packages such as SVM$^{\text{light}}$ and LIBSVM (version 2.82). The results given in Table 1 are merely illustrative. More extensive parameter tuning would definitely lead to better accuracies than reported.

We selected some of the data sets made available on the LIBSVM [4] webpage and the UCI machine learning repository [7]. The datasets used were

1. Liver-UCI [7]—345 $\times$ 7. No test set.
2. W7A [16]—24,692 training points with 300 features; 25,057 test points.
3. Ijcnn [17]—49,990 training points with 22 features; 91,701 test points.
4. RCV1 [11]—20,242 training points with 47,236 features; 677,399 test points.

**Table 1.** Training accuracies and associated running times for our $\ell_1$-SVM implementation with both soft and hard margins ($C = \infty$), and comparative numbers for SVM$^{\text{light}}$ and LIBSVM. The running times reported are exclusive of the time spent in I/O. We added timing computation code to LIBSVM. For the data sets that came with a separate test collection, we also report test accuracies and test times. Our implementation was overall faster in both training and testing, with a marginal sacrifice in terms of accuracies.

| Data set | $\ell_1$-SVM (hard) | $\ell_1$-SVM (soft) | SVM$^{\text{light}}$ | LIBSVM |
|---|---|---|---|---|
| Liver-UCI | 69.6% (0.02s) | 71.2% (0.04s) | 74.8% (0.12s) | 70.4% (0.03s)) |
| W7A | 97.7% (1.49s) | 98.6% (3.3s) | 98.7% (6.2s) | 98.7% (12.6s) |
| W7A (test) | 97.6% (0s) | 98.6% (0s) | 98.7% (0.01s) | 98.7% (3.4s) |
| Ijcnn | 90.3% (0.04s) | 92.1% (1.5s) | 92.4% (22.4s) | 92.4% (84s) |
| Ijcnn (test) | 90.1% (0.01s) | 91.7% (0.01s) | 92.1% (0.08s) | 92.1% (85.1s) |
| RCV1 | 99.6% (0.08s) | 98.8% (2.2s) | 98.9% (19s) | 98.9% (318s) |
| RCV1 (test) | 96.0% (0.29s) | 96.3% (0.30s) | 96.3% (0.90s) | 96.3%($\sim$ 82min) |

For all the implementations tested, we used the same value for the cost parameter $C$. Further, for both SVM$^{\text{light}}$ and LIBSVM we used the linear kernels. Observe that our $\ell_1$-SVM outperforms both SVM$^{\text{light}}$ and LIBSVM in terms of training and testing speed, with a small drop in accuracy—except for the RCV1 dataset, where the $\ell_1$-norm SVM has higher training accuracy.

## 4.2 Scalability Experiments on Synthetic Data

We performed a series of experiments on synthetically generated data to test the scalability of our $\ell_1$-SVM. We sampled an equal number of points from two multidimensional von Mises-Fisher distributions [15], with overlapping means, so that the data were not linearly separable. We compare our implementation

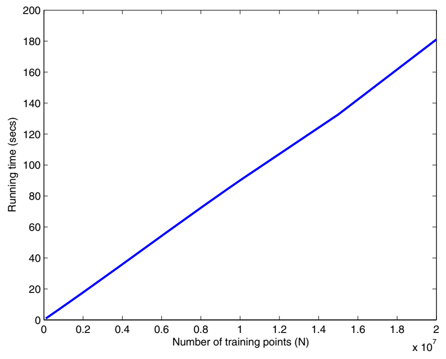

**Fig. 1.** Running time of our $\ell_1$-SVM to demonstrate its scalability. The plot ranges from $N = 100{,}000$ to 20 million, and the corresponding times range from 0.89s to 181s. The run time can be seen to grow linearly in the number of training points (because the number of training points $N \gg M$, no effect of dimensionality is discernible).

against the Linear Programming Chunking (LPC) approach of [2]. Empirically, our method for solving the $\ell_1$-SVM is several orders of magnitude faster than the LPC method. An exact number representing the speedup is not possible to provide since the authors of LPC ran their experiments on a different platform than ours. Nevertheless, we offer a conservative estimate of speed to permit a rough comparison. LPC was run on a cluster with 64 Sun UltraSPARC II processors, with a total of 8GB of RAM. We ran our $\ell_1$-SVM code on an Intel Xeon 3.2GHz Linux machine with 8GB RAM. Conservatively assuming that both machines run at approximately the same speed (i.e., disregarding the fact that the cluster had 64 processors)[4] we can give a crude comparison between the two implementations. Note that these numbers are merely indicative of the speedup, since LPC and $\ell_1$-SVM were run on different machines. The LPC method consumed 6.94, 25.91, and 231.32 hours, for 200,000, 500,000, and 1 million points, respectively. In comparison, our method took 1.76, 4.43, and 8.8 seconds for the same sized datasets. These numbers are compelling and show that our $\ell_1$-SVM runs several orders of magnitude faster than the LPC algorithm while trying to solve the same problem. Hence, for such large scale datasets, it should be the

[4] This estimate is conservative because if one compares the performance of the cluster with that of a single Xeon based machine, the cluster is a few times faster—as can be ascertained by going through CPU/System comparison benchmarks.

method of choice. Our speed gains come from two sources: i) the decomposition approach, and ii) solving the primal problem instead of the dual.

We mention in passing that Mangasarian and Musicant [14] solve a particular version of the $\ell_2$-norm SVM problem by attacking the dual formulation using an active set approach. Our $\ell_1$-norm implementation can be modified to solve the primal version of their problem and once again outperforms the dual. Mangasarian and Musicant [14] reported running times of (on a 400MHz Pentium II Xeon processor with 2GB RAM) of 38 minutes for a problem size of 4 million points (in $\mathbb{R}^{32}$), and 96.57 minutes for 7 million points. Comparative numbers for our implementation can be obtained from Figure 1. Interpolation yields the estimates 36 seconds for 4 million points and 63 seconds for 7 million points.

## 5 Discussion

In this paper we treated the solution of a linear programming based $\ell_1$-norm SVM problem by converting it into a quadratic program, which was then solved by an efficient decomposition method. As far as we know, nobody has applied this idea to the solution of $\ell_1$-SVMs before. We saw that the decomposition method permits efficient solution of extremely large-scale problems. Furthermore, our results corroborate the non-surprising, but often overlooked fact that for problems where the number of training points vastly outnumbers their dimensionality, solving the primal problem is more efficient.

Since the decomposition procedures that we invoked are quite general, they can easily be adapted to solve related problems such as SV-regression, linear programming SVMs [6, 8], and the so-called $\epsilon$- and $\nu$-SVM variants.

### 5.1 Future Work

The $\ell_1$-SVM presented in this paper is a preliminary piece of work. Many further refinements need to be incorporated into it to make it a highly competitive and accurate SVM training engine. Notable extensions to it that are currently under preparation include:

- Automatic determination of a good values for the cost parameter $C$ and the control parameter $\epsilon$.
- Improved methods for automatically determining early stopping criteria so that the algorithm takes minimum amount of running time without sacrificing too much accuracy.
- Additional improvements to the algorithm itself to improve its rate of convergence, and decrease errors due to numerical difficulties.

## Acknowledgments

I thank Brian Kulis for initial discussions about the $\ell_1$-SVM.

## References

[1] K. Bennett. Combining support vector and mathematical programming methods for classification. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods*, pages 307–326. MIT Press, 1999.

[2] P. S. Bradley and O. L. Mangasarian. Massive data discrimination via linear support vector machines. *Optimization Methods and Software*, 13(1), 2000.

[3] Y. Censor and S. A. Zenios. *Parallel Optimization: Theory, Algorithms, and Applications*. Oxford University Press, 1997.

[4] C-C. Chang and C-J. Lin. LIBSVM: a libary for support vector machines, 2001. *http://www.csie.ntu.edu.tw/˜cjlin/libsvm*.

[5] O. Chapelle. Training a support vector machine in the primal. Technical report, Max Planck Institute for Biological Cybernetics, 2006.

[6] N. Cristianini and J. Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[7] C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998.

[8] T. Graepel, R. Herbrich, B. Schölkopf, A. Smola, P. Bartlett, K.-R. Müller, K. Obermayer, and R. Williamson. Classification on proximity data with lp-machines. In *9th Int. Conf. on Artificial Neural Networks: ICANN*, 1999.

[9] C. Hildreth. A quadratic programming procedure. *Naval Res. Logist. Quarterly*, 4, 1957.

[10] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods*, pages 42–56. MIT Press, 1999.

[11] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5:361–397, 2004.

[12] O. L. Mangasarian. Normal solutions of linear programs. *Mathematical Programming Study*, 22:206–216, 1984.

[13] O. L. Mangasarian. Exact 1-norm support vector machines via unconstrained convex differentiable minimization. *Journal of Machine Learning Research*, 2006.

[14] O. L. Mangasarian and D. R. Musicant. Active support vector machine classification. In *NIPS*, 2001.

[15] K. V. Mardia and P. Jupp. *Directional Statistics*. John Wiley and Sons Ltd., second edition, 2000.

[16] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods*, pages 185–208. MIT Press, 1999.

[17] D. Prokhorov. Slide presentation in IJCNN'01, Ford Research Laboratory. IJCNN 2001 neural network competition, 2001.

[18] S. Sra and S. S. Jegelka. SSLib: Sparse Matrix Manipulation Library. *http://www.cs.utexas.edu/users/suvrit/work/progs/sparselib.html*, 2006.

[19] J. Zhu, S. Rosset, T. Hastie, and R. Tibshirani. 1-norm support vector machines. In *NIPS*, 2004.