A parallel training algorithm for large scale support vector machines

Elad Yom-Tov
IBM Haifa Research Labs
Haifa 31905
Israel
email: yomtov@il.ibm.com

November 28, 2004

1 Abstract

Support vector machines (SVMs) are an extremely successful class of classification and regression algorithms. Building an SVM entails the solution of a constrained convex quadratic programming problem which is quadratic in the number of training samples. Previous parallel implementations of SVM solvers sequentially solved subsets of the complete problem, which is problematic when the solution requires many support vectors. In this article we introduce a parallel implementation of a sequential SVM solver which overcomes these problems and makes it possible to solve extremely large SVM problems, with up to several million training points.

Keywords: Support vector machines, pattern classification, quadratic programming.

2 Introduction

Support vector machine (SVM) is an algorithm which can be used, among others, for classification[8], regression[7], and ranking[5]. Building a support vector machine problem entails the solution of a quadratic programming (QP) problem which is of the size of the number of training patterns. Thus, although there arise many problems where there are large data sets (OCR, BioInformatics, imaging, etc), it is possible to apply SVMs to them due to the difficulty in solving such large QP problems. In this article we present a highly efficient parallel implementation of a previously-proposed solver for SVM, and show that it can solve SVM problems with hundreds of thousands and even millions of training samples.

There are several formulations of the SVM problem, depending on the specific application of the SVM (Classification, regression, etc). In the framework of classification consider a training set

$$D = \{(\mathbf{x}_i, y_i), \quad i = 1, ..., N, \quad \mathbf{x}_i \in \Re^m, \quad y_i \in \{-1, 1\}\}. \tag{1}$$

The goal of the SVM is to learn a mapping from $\mathbf{x}_i$ to y_i such that the error in mapping, as measured on a new dataset, would be minimal. SVMs learn to find a separating function that maximizes the margin between the data from the two classes.

Finding the weights of the linear function can be done directly, or through the solution of the dual problem. In the following we use the notation of [10]. The dual problem is thus:

$$Maximize \;\; L_D(h) = \sum_i h_i - \frac{1}{2} h \cdot D \cdot h \tag{2}$$

$$subject\ to \;\; 0 \leq h_i \leq C, \;\; i = 1, ..., N \tag{3}$$

where D is a matrix such that $D_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) + \lambda^2 y_i y_j$, and $K(\cdot, \cdot)$ is either an inner product of the samples or a function of these samples. In the latter case this function is known as the kernel function, which can be any function which complies with the Mercer conditions[8], for example polynomial functions, radial-basis (Gaussian) functions, or hyperbolic tangents. If the data is not seperable, C is a tradeoff between maximizing the margin and reducing the number of misclassifications. Finally, λ is a small constant which is not part of the basic SVM formulation. This parameter was added in [10] as an additional degree of freedom, and is set as in [10].

The resulting classifier is computed as:

$$f(\mathbf{x}) = sign\left(\sum_{i \in SV} h_i y_i K(\mathbf{x}_i, \mathbf{x}) + h_i y_i \lambda^2\right) \tag{4}$$

The matrix D is not a sparse matrix, and when the size of the problem is large solving the QP problem is difficult.

There are several main methods for finding a solution to the SVM problem[8]. Interior point algorithms solve the optimization problem by simultaneously satisfying the primal and dual feasibility conditions. These algorithms work by iteratively solving a set of equations. However, it is known that interior point algorithms have a difficulty solving large-scale SVM problems due to the need to invert large matrices. A solution, albeit only approximate, can be obtained by using a low rank approximation of the kernel matrix [3].

Most other methods for solving the SVM problem use subset selection for reducing the problem size. The initial idea for subset selection, known as chunking [9], worked by storing in memory part of the data, finding the support vectors for this partial problem, and replacing all the points which are not support vectors with new data, until convergence is met. This approach works well if the

whole set of support vectors can be kept in memory, but when this is not the case chunking will converge extremely slowly, as we show below.

A slightly different approach are the Working Set algorithms. These algorithms perform gradient descent on a subset of the variables, known as the working set, while freezing other variables. A working set algorithm was parallelized in [11]. The working set approach is taken farthest in Platt's sequential minimal optimization (SMO) algorithm[6], where the working set is comprised of two samples and the analytic update to the variables is computed. The main limit of the Working Set algorithms is that if the support vectors cannot be held in memory the algorithm might not converge because of a phenomena known as thrashing, the corrections made according to one working set will be canceled by the corrections of another, and vice-versa.

Finally, some on-line SVM solvers exist. These solvers iterate through the training data and adjust the solution based on each sample. These are efficient algorithms for linear SVMs, but cannot be used for nonlinear SVMs.

Most SVM solvers (Surveyed in [8]) are not readily parallelable. In fact, so far little has been done to parallelize SVMs, a few note exceptions being [11, 2]. However, we believe that the only currently feasible method for solving very large scale SVM problems is through parallel solvers.

3 Parallel sequential support vector machine solver

In [10] a sequential version of the SVM solver is proposed. A pseudocode of the algorithm is described below:

Step 1: Initialize $h_i = 0$.
Step 2: For each pattern $i = 1, ..., N$ compute:

1. For each sample $j = 1, ..., N$ compute $d(j) = y_i y_j \left(K(\mathbf{x}_i, \mathbf{x}_j) + \lambda^2 \right)$
2. $E_i = \sum_{j=1}^{N} h_j d_j$
3. $\delta h_i = min \{ max [\gamma (1 - E_i), -h_i], C - h_i \}$.
4. $h_i = h_i + \delta h_i$

Step 3: If training has converged (See below), then stop. Otherwise goto step 2.

In [10] it is shown that the learning rate γ is bounded by $\gamma \leq 2/max_i \{D_{ii}\}$ and that the training data must be scaled so that $\|\omega\|^2 > 1/4$.

It is possible to parallelize the sequential SVM algorithm by running step 2 of the algorithm on different processing elements for each pattern or for each group of patterns. The main limitation is that each processing element has to be able to hold in memory (and compute) $d(j)$ of at least one training sample. In our (Matlab) implementation this limits the number of training samples to about $5,000,000/Data\ dimension$. If the number of training samples is smaller than

the maximum it is useful to compute more than one sample on each machine in order to reduce the number of data transfer operations.

Training was terminated (Step 3) when the largest update to any Lagrange multiplier h_i was small enough or when the target function ($L_D(h) = \sum_i h_i - \frac{1}{2} h \cdot D \cdot h$) value decreased compared to the previous iteration.

We implemented the parallel sequential SVM solver in Matlab, a matrix language especially suitable for the computations of the algorithm steps.

4 Hardware setup

One 2.4Ghz Pentium-based system running the Windows 2000 operating system was used as the server for the system. Between one and three 2.4Ghz Pentium-based systems running Linux 9 were used as clients. All machines had 1Gb memory. An AFS server was used as the joint storage space.

The server ran one copy of Matlab that generated jobs for processing by the clients. The jobs were represented as separate files. Any common variables (For example, the training data) were saved in a different file and read only once by each client application. Each client ran one or two copies of Matlab (Thus there was a maximum of 6 clients working at a given time). Each client checked the joint storage space and if a job for processing was available it read this job, deleted it from the storage space, processed and returned the answer as a file in the joint storage space.

After spawning all jobs for a particular command the server started collecting the answers returned by the clients until all answers were collected.

As a comparison to parallel SVM, a copy of SVMlight [4] was run on the server machine. SVMlight is a working set-type solver.

In all our experiments we used a radial-basis classifier with a radius of 1 and a cost function (C) of 1.

5 Results

We used two of the largest databases from the UCI Repository [1] in our study. The first database was the 20,000 sample Letter OCR database (17 attributes, distinguishing between the letters A-M and N-Z). The second database consisted of the first 200,000 examples from the Covertype dataset (54 attributes, distinguishing between Lodgepole pine and all other cover types).

The parallel sequential SVM solver was compared to SVMlight solver by training an SVM classifier using between 5% and 90% of the data (In steps of 5%) and testing on the remaining 10% of the data.

The relative difference in the test set classification rates between the SVM classifiers found by the two methods, defined as $|C_{SVMlight} - C_{Parallel}| / C_{Parallel}$, was less than 1%.

Figure 1 shows the time taken by the parallel SVM solver and by SVMlight to convergence to a solution. The left figure shows the time taken for the

Covertype dataset, which is a non-separable dataset (As indicated by the classification error, which is in the order of 35%). The right figure shows the time to convergence for the Letters dataset, which is seperable (The classification error is in the order of 2%). Note that when the dataset is seperable both solvers scale similarly, in contrast to the scaling performance when the dataset is unseparable. In the latter case the parallel SVM solver scales significantly better compared to SVMlight. The reason for this is that in non-separable data it is increasingly difficult to hold the support vectors in memory as the number of training points grows. This in turn causes Working Sets solvers to thrash, and convergence (when reached) is extremely slow. In contrast the sequential SVM solver is not limited by such problems, and thus its convergence curves are less affected by the size of the dataset.

Note that the training of SVMlight was terminated after 70,000 training points since the time to converge was prohibitively large. For example, training an SVMlight classifier with 80,000 points (the next step in the figure) would have taken approximately 7 hours according to the regression curves.

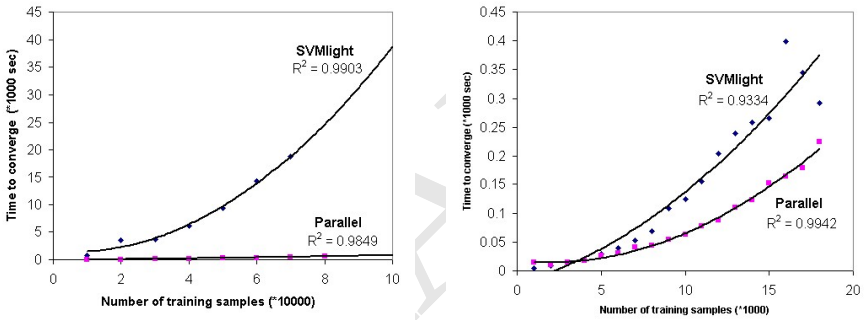

Figure 1: Convergence time of the parallel SVM solver compared to that of the SVMlight solver. The left figure shows the time taken to converge when classifying the non-separable Covertype dataset. The right figure shows the time to converge when classifying the separable Letters dataset. Second-order polynomial regression curves are shown on the graph, together with the R^2 regression coefficients. These figures show that when the dataset is separable there is little difference between the performance of the two solvers. However, when the dataset is not separable, the parallel SVM solver scales significantly better.

Figure 2 shows how the time to convergence of the parallel SVM classifier depends on the number of clients. This figure demonstrates the time to converge given 18,000 samples of the Letters dataset. We also measured these times when training with 9,000 samples of the Letters dataset and 20,000 samples of the Covertype dataset. Assuming that there is a certain processing time for each sample and a constant read/write time of the data and the results, we fit a

regression model of the form

$$t = \alpha_{RW} \cdot N_{Sub} + \alpha_{Pr} \cdot t_{Sub} \cdot N_{Sub}/N_{Clients} \tag{5}$$

where t is the time to converge, $N_{Clients}$ the number of active clients, N_{Sub} the number of sub-problems (i.e.: the number of blocks the data was divided into for processing), α_{RW} the read/write regression parameter, and α_{Pr} the client processing regression parameter. The parameter N_{Sub} depends quadratically on the number of training points. The results show that this model fits the Letters data with $R^2 = 0.89$ ($\alpha_{RW} = 69.9, \alpha_{Pr} = 9.66$) and the Covertype data with $R^2 = 0.999$ ($\alpha_{RW} = 96.7, \alpha_{Pr} = 27.4$). This suggests that such a model is indeed applicable to the curve. The implications of this model is that improvement in performance is limited by the read and write times of the network as well as by the number of clients, which for the Letters database saturates at about 4 clients.

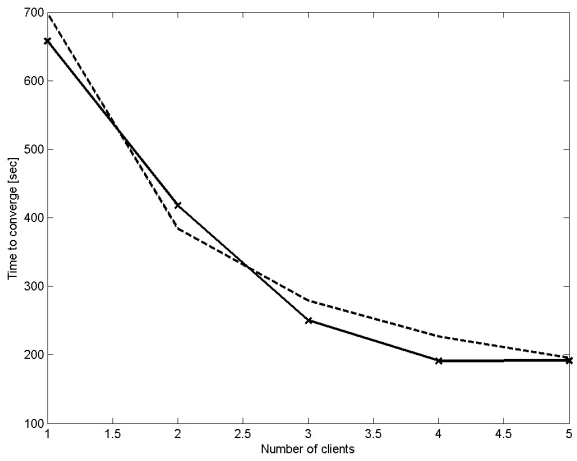

Figure 2: Convergence time of the parallel SVM solver versus the number of clients. This is the convergence time for the Letters dataset, using 18,000 samples. The dotted line shows the predicted values according to the model described in the text.

The speedup of the sequential SVM is computed as the ratio of the time needed to converge using a single processing element (T_S) to the time needed to converge using p processing elements (T_p),

$$sp_r(p) = \frac{T_S}{T_p} \tag{6}$$

Figure 3 shows the relative efficiency ($eff(p) = sp_r(p)/p$) for the three examined setups. This figure shows that the parallel algorithm is efficient on the

small number of processors measured. Notice how, for the separable Letters dataset, as more training data is available, so does the efficiency reach a maximum at a higher number of processors. In contrast, efficiency does not change significantly for the non-separable Covertype dataset.

Finally, we report the computation of Kuck's function, defined as:

$$f_K(p) = eff_r(p)sp_r(p) = \frac{sp_r^2(p)}{p} \tag{7}$$

This function reaches a maximum in the optimal number of processors for a given problem. As one can see in Figure 4, this maximum is reached using 3-4 processing elements.

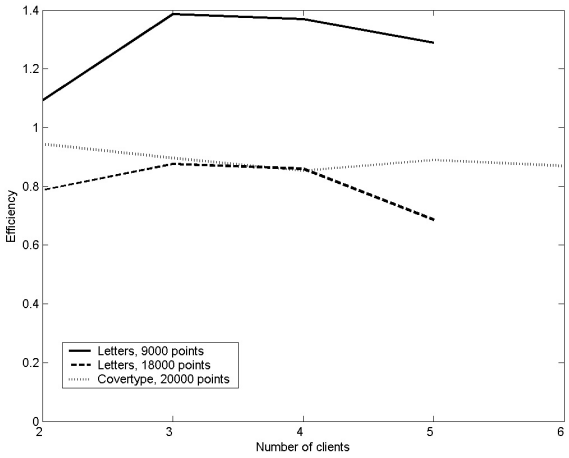

Figure 3: Relative efficiency versus the number of client processors.

6 Discussion

This paper presents a parallel implementation of the sequential support vector solver suggested in [10]. In difference from the popular Working Set algorithms, this algorithm solved the quadratic programming problem associated with SVMs in batch mode. That is, the update to every Lagrange multiplier is computed at each iteration. This overcomes the problems which occur in other support vector solvers that arise when the set of support vectors is too large to be held in memory.

The proposed implementation makes it possible to solve extremely large classification and regression problems, which have so far been outside the scope of SVMs. Currently problems with as many as 5,000,000 samples are handled by the parallel sequential SVM solver.

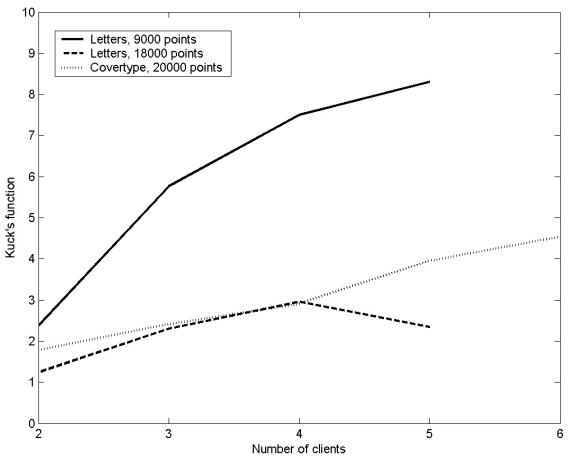

Figure 4: Kuck's function versus the number of client processors. Note that the optimal number of processors is only achieved for one problem.

Our tests on two large datasets show that the time to converge with the (parallel) sequential SVM solver is on-par with the popular SVMlight package when the dataset is separable, and is significantly better than SVMlight when the data is not separable (and thus requires many support vectors to be used for the solution).

As we have demonstrated, the current speed of the parallel implementation is limited by both the network access times as well as by the number of processing elements. Therefore more processing clients are not enough to improving the processing speed of the algorithm. Better network nodes and better management mechanisms are also essential.

References

[1] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[2] R. Collobert, S. Bengio, and Y. Bengiou. A parallel mixture of svms for very large scale problems. In *Advances in Neural Information Processing Systems*. MIT Press, 2002.

[3] S. Fine and K. Scheinberg. Efficient svm training using low-rank kernel representations. *Journal of Machine Learning Research*, 2:243–264, 2001.

[4] T. Joachims. Making large-scale svm learning practical. In *Advances in Kernel Methods - Support Vector Learning*, 1999.

[5] T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, 2002.

[6] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. In *Advances in Kernel Methods - Support Vector Learning*, pages 185–208, 1999.

[7] B. Scholkopf and A.J. Smola. A tutorial on support vector regression. In *NeuroCOLT2 Technical Report NC2-TR-1998-030*, 1998.

[8] B. Scholkopf and A.J. Smola. *Leaning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT Press, Cambridge, MA, USA, 2002.

[9] V. Vapnik. *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, 1982.

[10] S. Vijayakumar and S. Wu. Sequential support vector classifiers and regression. In *International conference on soft computing*, pages 610–619, 1999.

[11] G. Zanghirati and L. Zanni. A parallel solver for large quadratic programs in training support vector machines. *Parallel computing*, 29:535–551, 2003.