ABSTRACT

Modern computer hardware offers an elaborate hierarchy of storage subsystems with different speeds and capacities. Furthermore, processors are now inherently parallel offering the execution of several diverse threads simultaneously. This paper proposes a first SVM optimization algorithm which takes advantage of these properties by integrating caching with optimization. It works by performing updates in the dual, thus obviating the need to rebalance frequently visited instances. Furthermore we trade off file I/O with data expansion on the fly by generating features on demand. This significantly increases throughput. Experiments show that it outperforms other linear SVM solvers, including the award winning work of [?], by orders of magnitude and produces more accurate solutions within a given amount of time.

1. INTRODUCTION

In this paper we propose an efficient method for large scale single machine Support Vector estimation, yielding the fastest currently available solver which runs on general purpose hardware by over an order of magnitude. To our knowledge, this is the first work on SVMS which takes explicit advantage of different storage subsystems on a computer to achieve very high data throughput. In particular, our optimization algorithm is co-designed to take advantage of the fact that memory access is much faster than disk access. The approach is entirely generic and can be extended to a more fine-grained storage hierarchy and a larger class of convex optimization problems.

1.1 Active Set Solvers

Support Vector Machines (SVMs) have arguably revolutionized machine learning over the past decade. Fueled by their impressive performance in a number of real-world applications, there have been numerous efforts to scale SVMS to large data sets. Some notable contributions include Sequential Minimal Optimization (SMO) [24, 8], SVMLight [15], LaSVM [2], and gradient projection based solvers [34].

Much of the initial success of SVMs was attributed to the so-called kernel trick wherein training data is implicitly mapped to a high dimensional feature space, and a margin maximizing linear classifier is learned in this mapped space. In contrast, linear SVMs do not employ the kernel trick explicitly. As massive, high-dimensional data sets are becoming commonplace, there is a recent surge of interest in linear SVMS. Some recent papers [16, 14, 13, 31] tackle this problem with great success, and provide algorithms with convergence guarantees. The above solvers either assume that the data resides in memory or that a cluster of machines with appropriate communication and synchronization facilities are available.

In the award winning paper [?] revisited the problem of training SVMs when the data does not fit into memory [25, 23, 15]. In a nutshell, the key idea is to split the data into manageable blocks, compress and store each block on disk, and perform dual coordinate descent by loading each block sequentially. This basic idea was improved upon by [?] who observed that the block minimization (BM) algorithm of [?] does not retain important points before discarding each block. They therefore, propose to retain some important points from the previous blocks in the RAM. This simple idea leads to significant speed-ups, and [?] demonstrate that their selective block minimization (SBM) algorithm outperforms BM.

A closely related alternative to dual coordinate descent is stochastic gradient descent (SGD). Recently there have been numerous variants of which have been studied both theoretically [3, 26] as well as empirically [4].

1.2 The Storage Hierarchy

Our algorithm takes advantage of the different characteristics inherent in the storage hierarchy of modern computers. That is, while hard disks excel at storing large amounts of data, they have typically mediocre data transfer rates and are outright slow at random access operations. Compared to that, main memory comes at a hundredfold premium in terms of space but offers two to three orders of magnitude faster data transfer rates. CPU caches are yet faster again. Similar considerations hold for solid state drives, PCI interconnects, and graphics subsystems.

This suggests that algorithms which require streaming through data from disk should take advantage of the data...
they already have in main memory while waiting for more data to arrive from disk. Obviously, the same rationale applies to a sequence of storage systems with different capacity/bandwidth characteristics. To make things somewhat more explicit we list a range of such systems below:

<table>
<thead>
<tr>
<th>System</th>
<th>Capacity</th>
<th>Bandwidth</th>
<th>IOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>3TB</td>
<td>150MB/s</td>
<td>$10^4$</td>
</tr>
<tr>
<td>SSD</td>
<td>256GB</td>
<td>500MB/s</td>
<td>$5 \cdot 10^4$</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB</td>
<td>30GB/s</td>
<td>$10^8$</td>
</tr>
<tr>
<td>Cache</td>
<td>16MB</td>
<td>100GB/s</td>
<td>$10^9$</td>
</tr>
</tbody>
</table>

In the present paper we focus on two parts of this hierarchy — disk and memory. This already affords quite dramatic improvements in terms of speed relative to sequential algorithms. In a nutshell our algorithm does the following:

Iterate over the data in main memory while streaming data from disk. Evict primarily instances from main memory that are uninformative.

A naive approach which takes, e.g. stochastic gradient descent steps based on the importance of instances would likely run afoot of data weighting problems — informative instances need not have extended statistical weight but rather only a higher influence on the choice of objective function. One way of dealing with this issue is to resort to dual updates. That is, we consider the dual optimization problem to SVMs (and related problems) and judiciously update the associated Lagrange multipliers. This leads to the following algorithm:

**Reader**

while not converged do
  read instance $(x, y)$ from disk
  if buffer full then evict random $(x', y')$ from memory
  insert new $(x, y)$ into ring buffer in memory
end while

**Optimization**

while not converged do
  cyclically pick instance $(x, y)$ from memory
  update dual parameter $\alpha$
  update weight vector $w$
  evict stale $(x, y)$ from memory
end while

1.3 Enterprise Scale Solvers

Industrial data sets regularly exceed the capacities offered by single computers in terms of both storage and computation. This means that distributed inference techniques are required. Unfortunately, large server centers often come with rather severe restrictions on reliability, inter-machine latency, communication trees, delays, etc. such that it is desirable to find algorithms which compute estimates using a bare minimum of communication. Note that not all estimation problems are amenable to efficient high-latency scenarios. For instance, latent variable models typically require excellent communication and great care needs to be taken to obtain fast estimates [1, 28]. A common attribute of these models is that they have a large degree of symmetry and non-convexity in their parametrization.

Fortunately much of what is commonly known as generalized linear models can be addressed with convex solvers. That is, the estimation problems can be decomposed efficiently into parts which are guaranteed to yield very similar solutions. In statistical learning terminology this is known as stability of the solution space and there exists a rich body of research [5, 22, 7, 12] extolling the desirable theoretical properties of convexly penalized estimation problems. It is therefore natural to take advantage of these properties in terms of implementations. For instance, [35] show that it is possible to perform stochastic gradient descent on individual processors independently, using random sub-samples of the data and to average the parameter estimates afterwards and simultaneously reaping the benefits of parallelization. Note that in previous work [21] suggested a similar averaging strategy, however their theoretical analysis only showed that averaging does not hurt, rather than actually accelerate convergence. It is in the spirit of [35] that we approach the problem of estimation:

1. Decompose (possibly with oversampling) the data for several machines.
2. Solve the estimation problem per machine as efficiently as possible.
3. Average the solutions between machines to obtain a final estimate.

Much work in the analysis of [35] was invested into proving that the stochastic gradient descent solutions on subsets are sufficiently independent for averaging to be actually beneficial. If we treat the optimization step as a standard batch problem this obstacle disappears. In this case we can appeal directly to the asymptotic analysis of [22] to see that averaging is beneficial: in particular, [22] show that the parameter distribution of a penalized empirical risk minimizer $w^*|X, Y$ conditioned on some data $X, Y$ is asymptotically normal. This means that if we obtain such estimates based on various subsets of data via $w^*|X_i, Y_i$, we will be able to aggregate this to an improved joint estimate via $\frac{1}{n} \sum_{i=1}^n w^*|X_i, Y_i$.

Consequently, in the present paper we are primarily concerned with step 2 of the above approach — to find the most efficient way of solving a convex optimization problem on a single machine.

**Outline.** We begin by giving an overview of dual descent algorithms for linear Support Vector Machines in section 2. Subsequently Section 3 gives a detailed description of the nested loop used in traversing through data in core memory and streaming from disk. Experimental results are provided in Section 5 and we conclude with a discussion in Section 6.
2. DUAL COORDINATE DESCENT

2.1 Support Vector Classification

In the following we assume that we are given \( m \) instances \( x_i \in \mathcal{X} \) and labels \( y_i \in \{\pm 1\} \) drawn independently and identically from some distribution \( p(x,y) \). It is our goal to find some function \( f : \mathcal{X} \rightarrow \mathbb{R} \) which minimizes the misclassification error, by minimizing the probability that \( yf(x) \leq 0 \). This constitutes the most basic of all estimation problems, namely that of binary classification. \(^2\) We simplify things further by assuming that \( \mathcal{X} = \mathbb{R}^n \). This assumption will be relaxed subsequently when we discuss how to expand features on the fly. The primal formulation of a linear SVM can be written as follows \(^3\) [10]:

\[
\begin{align*}
\text{minimize} \quad & \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{m} \max\{0, 1 - y_i w^\top x_i\} \\
\text{subject to} \quad & w^\top x_i \leq 1, \quad w^\top x_i \geq -1.
\end{align*}
\]

(1)

Using standard convex optimization tools [6] the above problem can be rewritten in its dual form:

\[
\begin{align*}
\text{minimize} \quad & D(\alpha) := \frac{1}{2} \alpha^\top Q \alpha - \alpha^\top 1 \\
\text{subject to} \quad & 0 \leq \alpha \leq C.
\end{align*}
\]

(2a)

Here, \( Q \) is an \( m \times m \) matrix whose entries are given by \( Q_{ij} = y_i y_j x_i^\top x_j \), and \( 1 \) is the vector of all ones. The minimizer \( w^* \) of (1) and the minimizer \( \alpha^* \) of (2) are related by the primal dual connection: \( w^* = \sum_{i=1}^{m} \alpha^*_i y_i x_i \). The dual problem (2) is a Quadratic Program (QP) with box constraints, and the \( i \)-th coordinate \( \alpha_i \) corresponds to the \( i \)-th instance \( (x_i, y_i) \).

2.2 Dual Updates

[14] showed that the following coordinate descent scheme can be used to minimize (2):

- Initialize \( \alpha^1 = 0 \).
- At iteration \( t \) select coordinate \( i_t \).
- Update \( \alpha^t \) to \( \alpha^{t+1} \) via:

\[
\begin{align*}
\alpha_{i_t}^{t+1} &= \arg\min_{0 \leq \alpha \leq C} D(\alpha^t + (\alpha - \alpha_{i_t}) e_{i_t}) \quad (3a) \\
\alpha_i^{t+1} &= \alpha_i^t \quad \forall \ i \neq i_t. \quad (3b)
\end{align*}
\]

Here, \( e_i \) denotes the \( i \)-th standard basis vector. Since \( D(\alpha) \) is a QP, the one-variable subproblem (3a) can be solved exactly (see [14] for details):

\[
\alpha_{i_t}^{t+1} = \min \left\{ \max\left\{ 0, \alpha_{i_t}^t - \nabla_{i_t} D(\alpha) \right\}, C \right\}. 
\]

(4)

Here, \( \nabla_{i_t} D(\alpha) \) denotes the \( i \)-th coordinate of the gradient. The above updates are also closely related to implicit updates of [17, 18, 19]. If we maintain \( w^t := \sum_{i=1}^{m} \alpha^t_i y_i x_i \), then the gradient \( \nabla_{i_t} D(\alpha) \) can be computed efficiently using

\[
\nabla_{i_t} D(\alpha) = e_{i_t}^\top (Q \alpha - 1) = w^t_i y_i x_i - 1
\]

(5)

\(^2\)We limit ourselves to this case for the purpose of making the algorithm more explicit. That said, our method is entirely generic and would also be applicable problems of structured estimation such as those discussed in [30].

\(^3\)We omit an explicit bias term since this greatly simplifies the dual problem while remaining entirely general since the bias can be introduced easily as an additional coordinate in the data.

and kept related to \( \alpha^{t+1} \) by computing

\[
w^{t+1} = w^t + (\alpha_{i_t}^{t+1} - \alpha_{i_t}^t)y_i x_i. 
\]

(6)

A naive choice for \( i_t \) is to take examples periodically, which means \( i_t \) is defined as \( (t - 1 \mod m) + 1 \). In this case, data set is accessed sequentially and that procedure is iterated. This iteration from \( \alpha^{t(m+1)} \) to \( \alpha^{t(m+1)} \) is called outer iteration, contrated with inner iteration from \( \alpha^t \) to \( \alpha^{t+1} \). A practical choice is to randomly access examples by permutation of the indices. In that case, \( i_t \) is defined as \( \sigma_k^t((t - 1 \mod m) + 1) \) where \( \sigma^k \in \Sigma_m \) is a permutation.

2.3 Shrinking

A remarkable property of this problem is that we can potentially reduce the size of optimization problem by carefully choosing the subset of the data set. We aim to utilize the property and solve the entire problem efficiently. First we look at the following fact:

FACT 1. Let \( \alpha^* \) be the optimal solution on (2). Define \( A^m_{\text{in}}, A^m_{\text{out}} \) as \( \{i|\alpha^*_i = 0\} \), \( \{i|0 < \alpha^*_i < C\} \) and \( \{i|\alpha^*_i = C\} \) respectively. Any optimal solution of

\[
\text{max } D(\alpha) 
\]

s.t. \( \alpha_i = 0 \ (i \in A^m_{\text{in}}), \alpha_i \in \mathbb{R} (i \in A^m_{\text{out}}), \alpha_i = C (i \in A^m_{\text{out}}) \)

is optimal in (2).

\( A^m_{\text{in}} \) is called active set. In the context of dual coordinate descent, it indicates that such an \( i \)-th coordinate that \( \alpha_i = 0 \) is not necessary to be chosen in the entire process of optimization, and it is also the case when \( \alpha_i = C \) once hold. We can call examples \( x_i \) corresponding to \( \alpha_i = 0 \) or \( \alpha_i < C \) and \( \alpha_i = C \) holds as inactive data, active data and bounded data respectively. From KKT conditions, we can see that \( w^t_i x_i y_i > 1 \) implies the data is inactive. Conversely, \( w^t_i x_i y_i < 1 \) implies the example is bounded. Intuitively, inactive data is an easy example and bounded data is a difficult one. As these examples often occupies large amount of the whole data set for the linear SVM problems, the size of subproblem we have to solve intrinsically can be expected to be small. Then the key for the acceleration is to estimate the active set.

To eliminate inactive or bounded examples from the original optimization problem, some heuristics have been considered so far. Shrinking is one of heuristics which can reduce the time for the entire optimization based on the estimation of the active set. In \( k \)-th outer iteration, a coordinate is chosen from not the set of all coordinates \( \{1, \ldots, m\} \), but a working set \( A_k \subset \{1, \ldots, m\} \). And \( k \)-th iteration ends when every coordinate in \( A_k \) is chosen at once and only at once. At each update, the procedure of shrinking decides if the chosen variable is fixed and eliminated from the working set or remains in the working set. As some data in \( A_k \) may be eliminated in \( k \)-th iteration, it can be seen that the shrinking generates a new working set \( A_{k+1} \) and decreasing sequence of subsets

\[
A_0 \supset A_1 \supset \cdots \supset A_k \supset A_{k+1} \supset 
\]

where \( A_0 = \{1, \ldots, m\} \). Let \( t_k \) be the total number of inner iteration taken by the end of \( k \)-th outer iteration. As \( k \)-th outer iteration spends \(|A_k| \) times of inner iteration, \( t_k = \sum_{j=1}^{k} |A_j| \) holds. Shrinking determines the next working set based on the present gradient and projected gradients gotten
by the previous iteration. Formally, shrinking defines $A_{k+1}$ in the process of update as the following equation:

$$A_{k+1} := A_k \setminus \{ i_t | t_k < t \leq t_{k+1}, \alpha_i^t = 0, \nabla_i D(\alpha^t) > \epsilon^t_+ \} \cup \{ i_t | t_k < t \leq t_{k+1}, \alpha_i^t = C, \nabla_i D(\alpha^t) < \epsilon^t_- \}$$

Here $\epsilon^t_+$ is maximum and minimum projected gradient obtained by the previous outer iteration, defined by

$$\epsilon^t_+ = \max_{t_k < t \leq t_{k+1}} \nabla_i D(\alpha^t), \quad \epsilon^t_- = \min_{t_k < t \leq t_{k+1}} \nabla_i D(\alpha^t).$$

Projected gradient $\nabla_i^u D(\alpha^t)$ is identical to $\nabla_i D(\alpha^t)$ if there exists $\delta > 0$ such that $0 < \alpha_i^t - \delta \nabla_i D(\alpha^t) < C$, and 0 otherwise.

The criterion of shrinking for regarding an example as inactive or bounded can be explained as follows. As (7) holds, such an example that the value of $\nabla_i f(\alpha^t)$ is large locates far away from the present decision boundary. Also, an example with a large negative gradient locates far away but the opposite side to which it should be. When the present weight vector is close to the optimal, the value of margin is also close to that at optimal. Then it will remain positive at optimal when the value of margin is large enough at last stage of optimization. Then an example with sufficiently large positive margin and $\alpha_i^t = 0$ is supposed to be inactive. Also, an example with sufficiently large negative margin and $\alpha_i^t = C$ is supposed to be bounded.

On the other hand, we can estimate how far from the optimal solution the present weight vector is by using the project gradients. The following fact implies their relations;

**Fact 2.** $\nabla_i^u D(\alpha^k) \to 0$ for any $i$ and then $\epsilon^k_+ \to 0$ when $\alpha^k \to \alpha^*$.

When the present weight vector is close to the optimal, then the project gradient for all data is small. Then it the $\epsilon$ can be seen as a indicator of optimality. Actually, [14] uses $\epsilon_+ - \epsilon_-$ as a stop condition.

Then, at first stage of optimization, the absolute values of $\epsilon_+$ are large and then a small amount of data, which the distance from the decision boundary is large enough, is eliminated. And at the last stage of optimization, absolute values of $\epsilon_-$ are small and then most of inactive or bounded examples are eliminated.

The criterion for the discrimination of the active data described above is not guaranteed to be always correct. Then we need unshrinking phase after achieving the sub-optimality of the problem associated with some subset $A_K$. First we check the optimality of the solution as the solution of the entire subproblem. To do this, setting $A_{K+1} = \{ t_k = \{ 1, \cdots, m \}, \epsilon_+ = \epsilon_-, \epsilon_+ = \epsilon_- \}$, the same procedure is iterated at the beginning. The next outer iteration gives the stop condition with respect to the entire problem and stops if the condition is satisfied and continues the procedure of shrinking otherwise. As the solution obtained in $K$-th outer iteration is often very close to the solution of the entire problem and $t_K$ is much smaller than $Km$, the total number of inner iteration can be reduced drastically.

### 3. Proposed Method

Learning algorithms in general tend to have serious slowdown when we don’t have an enough memory to store entire data set. This is because each data has to be accessed iteratively and a time to access data from disk containing I/O time will grow up drastically. Then the I/O time reduction has to be considered when we treat large scale data which cannot fit in memory. As a coordinate descent also needs to iteratively access data, the training time will be huge without enough memory and then the same issue has to be considered when we take dual coordinate descent method.

We propose a novel reader-trainer scheme for dual coordinate descent in large scale linear SVM, in which a reader thread and a trainer thread which are running on each processor almost asynchronously. As one thread can consume one core, we can make use of multi-core processors more efficiently. And the reader thread can continue to read from the HDD sequentially then passing through the whole data speedily. The trainer thread can access data with random access and then a problem when we sequentially accessing to the data can be avoided. Moreover, our scheme doesn’t need to do any preprocess, either to stop learning to load a new data in the middle of optimization. Then we can say our scheme solves the problems which the previous method haven’t solved. Then the total time of training can be expected much shorter than the previous method.

As we assume data cannot fit in memory, we assume there exists a number $\Omega < m$, which is an upper bound for the number of data capable to stored in memory. We maintain working set $A$, the set of indices to which data corresponding are in the memory. $|A| \leq \Omega$ must hold during the whole process of the optimization. We identify an element of $A$ as the data set itself within a scope that doesn’t make an ambiguity.

The reading thread performs loading an example on $A$ continuously. When there already have been $\Omega$ data in $A$, one element of $A$ is randomly chosen and discarded. Because reading the data from the disk is still the most time-consuming operation through learning, the reading thread is going to concentrate on this task. The algorithm of the reading thread is denoted by Algorithm 1.

The training thread performs updates of $\mathbf{w}$ and $\alpha$ according to the formula of coordinate descent (4) and (6) with respect to an example $i_t$ at $t$-th update. Here, $i_t$ is uniformly sampled from the working set $A$. We define $t_k$ as the total number of updates performed by reading thread finishing passing all the data set $k$ times. As the training thread can perform updates independently of the reading thread and without accessing the disk, it is expected to perform many updates while the reading thread reads one data from the disk.

As discussed in section 2, the coordinate descent method can reduce much time to obtain the optimal solution by carefully choosing the working set so that we perform more updates with respect to active data. Considering that the present gradient and the statistics of the projected gradient can make an useful criterion, in our scheme, we also use these information to eliminate a seemingly-inactive/bounded data every time data is sampled from $A$.

As in the case in shrinking, an example with large positive margin and $\alpha_i^t = 0$ or large negative margin with $\alpha_i^t = C$ is eliminated from the working set in our scheme. Then the condition for $i$ to be eliminated can be written as follows:

$$\alpha_i^t = 0 \text{ and } \nabla_i D(\alpha^t') > \epsilon^t_+, \quad \text{or } \alpha_i^t = C \text{ and } \nabla_i D(\alpha^t') < -\epsilon^t_+.$$  

Here, $\epsilon^t_+$ is the maximum absolute value of projected gradient
computed among previous \(m\) updates, mo
\[
\tilde{\epsilon}^t = \max_{t = m \mod 1} |\nabla_i^\Pi D(\alpha^*)| \quad (t \mod m = 1)
\]
\[
\tilde{\epsilon}^{t+1} = \tilde{\epsilon}^t \quad \text{(otherwise)}
\]

There are two significant differences from shrinking in general case. One is that the eliminated data will be loaded on \(A\) again, because the reading thread is continuously loading data independent of the trainer’s decision. It means that the criterion for eliminating the data can be more encouraged. Another thing is that there is an upper bound for the data to be stored. Then when the elimination criterion is not severe, the size of data supposed to remain in cache in the current criterion will be much larger than \(\Omega\). These points imply that when the number of data currently stored in \(A\) is nearly \(\Omega\) and still the absolute values of \(\epsilon_+\) are large, discrimination of inactivity or boundedness will work better with more severe criterion. Then we propose the shrunken threshold considering the case \(\Omega\) is small, which uses \(\tilde{\epsilon}^t\) defined as
\[
\tilde{\epsilon}^t = \tilde{\epsilon}^t \quad (t \mod m = 1 \land |A(t)| \leq 0.9\Omega), \quad (9a)
\]
\[
\tilde{\epsilon}^t = 0.9\tilde{\epsilon}^t \quad (t \mod m = 1 \land |A(t)| > 0.9\Omega), \quad (9b)
\]
\[
\tilde{\epsilon}^t = \tilde{\epsilon}^{t-1} \quad (t \mod m \neq 1 \land |A(t)| \leq 0.9\Omega), \quad (9c)
\]
\[
\tilde{\epsilon}^t = 0.9\tilde{\epsilon}^{t-1} \quad (t \mod m \neq 1 \land |A(t)| > 0.9\Omega), \quad (9d)
\]

where \(A(t)\) denotes the working set from which \(i_t\) is sampled.

As for the stop criterion of our scheme, we calculate the maximum/minimum value of projected gradient among the range of \(\{i_t|t_k < t \leq t_{k+1}\}\). Given tolerance \(\delta\), learning will end when the condition
\[
\max_{t_k < t \leq t_{k+1}} \nabla_i^\Pi D(\alpha^*) - \min_{t_k < t \leq t_{k+1}} \nabla_i^\Pi D(\alpha^*) < \delta \quad (10)
\]
holds after reading thread read data set \(k\) times. The algorithm of the training thread is denoted by Algorithm 2.

Algorithm 1 read

\[
\begin{array}{l}
\text{for } k = 1, \ldots, \text{max_iter} \text{ do} \\
\quad \text{for } i = 1, \ldots, m \text{ do} \\
\quad \quad \text{if } |A| = \Omega \text{ then} \\
\quad \quad \quad \text{Sample } i \in A \\
\quad \quad \quad \quad A = A \setminus \{i\} \\
\quad \quad \text{end if} \\
\quad \text{Read } y_i, x_i \\
\quad \text{Calculate } Q_i = x_i^\top x_i \\
\quad \text{Load } y_i, Q_i, x_i \text{ on Cache} \\
\quad A = A \cup \{i\} \\
\quad \text{end for} \\
\quad \text{if stop_condition == true then} \\
\quad \quad \text{reading_thread_is_running =false} \\
\quad \quad \text{break;} \\
\quad \text{end if} \\
\text{end for}
\end{array}
\]

3.1 Proof of Convergence (sketch)

Our analysis builds upon the results of [20] and [14].

Definition 1 (Luo and Tseng Problem). Consider the following minimization problem
\[
\min_{\alpha} g(E(\alpha)) + b^\top \alpha \quad (11)
\]
\[
s.t. \quad L_i \leq \alpha_i \leq U_i \quad (12)
\]
where \(\alpha\) and \(b\) are \(n\)-dimensional vectors, \(E\) is a \(d \times n\) dimensional fixed matrix, \(L_i \in [-\infty, \infty)\) and \(U_i \in (-\infty, \infty]\) are upper and lower bounds respectively. The above optimization problem is a Luo and Tseng problem if the following conditions hold:

1. \(E\) has no zero columns
2. the set of optimal solutions \(A\) is non-empty
3. the function \(g\) is strictly convex and twice continuously differentiable
4. for all optimal solutions \(\alpha^* \in A\), \(\nabla^2 g(E(\alpha^*))\) is positive definite.

The following result was implicitly shown in the proof of Theorem 1 in [14].

Lemma 1. The SVM dual (2) is a Luo and Tseng problem.

Proof. Set \(E_i = y_i x_i^\top, b = 1, L_i = 0, U_i = C\) and \(g(z) = \frac{1}{2}|z|^2\). \(E\) has no column because we can assume \(x_i \neq 0\). \(A\) is obviously non-empty. \(g\) is obviously strictly convex and twice differentiable and \(\nabla^2 g\) is positive definite everywhere. □
All that remains to establish the convergence our coordinate descent scheme is to show it satisfies the almost cyclic rule. If the training thread sequentially accesses the cached training data, and the following conditions hold:

- The training thread is at most \( \kappa \geq 1 \) times faster than the reading thread, that is, the training thread performs at most \( \kappa \) coordinate updates in the time that it takes the reader thread to read one training data from disk.
- A point is never deleted from the cache unless the \( a_i \) corresponding to that point has not been updated.

Then it is easy to see that this corresponds to an almost cyclic rule with \( B = \kappa m \).

4. RELATED WORK

Before proceeding to the experiments we would like to discuss some related work and place our contributions in perspective. Perhaps the closest in spirit to our work is the online version of SBM described by [19]. Here, data is assumed to arrive in a streaming fashion, and a dual coordinate descent procedure is used on blocks of data. However, there are some important differences between the two methods. First, online SBM only looks at the data once, while our algorithm StreamSVM performs multiple passes through the data. Second, the reading and the training in SBM happen in a synchronous fashion, while our reader thread asynchronously reads and caches data. As we will see in the experiments, StreamSVM also achieves near-optimal generalization performance after just one or two passes through the data.

In order to speed up linear SVMs where the feature mapping can be computed explicitly, [20] introduced a computational framework for linear SVMs (COFFIN). Along the same lines, one can use hash functions to map sparse high-dimensional features into dense low-dimensional features. This idea, first described in [27] and improved upon in [33] is one of the important reasons why the Vowpal Wabbit learning framework is fast. Both the above techniques compute a feature mapping \( \phi(x) \) on the fly, and are very naturally incorporated into our framework either at the reader level or at the trainer level. In fact, we will demonstrate the use of the feature expansion technique in our experiments.

We would also like to mention the recent flurry of activity on developing parallel stochastic gradient descent solvers [19, 35, 11]. We view our research as complimentary to these efforts. For instance, in a distributed setting each node can train individually using our idea of asynchronous reader and trainer threads, and the weight vector can be periodically synchronized with a master.

5. EXPERIMENTS

We performed experiments on some of the largest publicly available data sets for binary classification to evaluate the performance of our algorithm, and to compare it with two existing methods namely SBM and BM. Since SBM was shown to be competitive with other methods such as Vowpal Wabbit (VW), Stochastic Gradient Descent (SGD), or Pegasos we do not explicitly compare against them here.

Datasets.
Table 1 summarizes the datasets used in our experiments. The webspam trigram dataset webspam-t\(^3\), as well as the the KDD cup 2010 dataset kddb\(^6\) are from the LibSVM binary data collection\(^7\). The dna and ocr datasets were all obtained from the Pascal Large Scale Learning Workshop website [29]. For all the datasets we randomly selected 80\% of the labeled data for training and the remaining 20\% for testing. Besides being massive, note that our datasets cover a variety of real-life scenarios. The dna dataset is dense, but is heavily imbalanced in terms of the number of positive vs negative examples. The ocr dataset is dense and balanced while both kddb and webspam-t are very sparse but very high dimensional.

Implementation Details.
Our code is implemented in portable C++ and uses POSIX threads (pthreads) for multi-threaded programming. In order to cache the data efficiently we use a thread safe in-memory hash table (StashDB) provided by Kyoto Cabinet\(^8\). We empirically find that the use of Kyoto Cabinet reduces our memory footprint by approximately 25\% compared to SBM and BM. Our program utilizes two threads namely the reader and trainer. The reader thread, as the name implies, is tasked with reading the data from disk and storing it into the cache. If the cache is full then the reader randomly replaces a cached point with the new data. The trainer thread randomly selects training points in the cache and performs updates. The trainer also deletes points from the cache based on the shrinking criterion. It is also responsible for computing the duality gap. When the gap is lower than the tolerance threshold or the number of iterations exceeds the maximum, then both the threads exit. Open-source code as well as all the scripts needed to reproduce our experimental results will be made available for download from http://www.stat.purdue.edu/~vishy.

Hardware.
All experiments were conducted on the Rossmann computing cluster at Purdue University\(^9\), where each node has two 2.1 GHz 12-core AMD 6172 processors with 48 GB physical memory per node.

Experimental Setup.

\(^2\)http://hunch.net/~vw/
Table 1: Summary of the datasets used in our experiments. \( n \) is the total # of examples, \( d \) is the # of features, \( s \) is the feature density (% of features that are non-zero), \( n_+ : n_- \) is the ratio of the number of positive vs negative examples, Datasize is the size of the data file on disk, \( \Omega \) is the number of points in the cache, and #Blocks is the data split used for SBM. M denotes a million.

<table>
<thead>
<tr>
<th>dataset</th>
<th>( n )</th>
<th>( d )</th>
<th>( s(%) )</th>
<th>( n_+ : n_- )</th>
<th>Datasize</th>
<th>( C )</th>
<th>( \Omega )</th>
<th>#Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ocr</td>
<td>3.5 M</td>
<td>1156</td>
<td>100</td>
<td>0.96</td>
<td>45.28 GB</td>
<td>100</td>
<td>150,000</td>
<td>40</td>
</tr>
<tr>
<td>dna</td>
<td>50 M</td>
<td>800</td>
<td>25</td>
<td>3e-3</td>
<td>63.04 GB</td>
<td>1</td>
<td>700,000</td>
<td>60</td>
</tr>
<tr>
<td>webspam-t</td>
<td>0.35 M</td>
<td>16.61 M</td>
<td>0.022</td>
<td>1.54</td>
<td>20.03 GB</td>
<td>100</td>
<td>15,000</td>
<td>20</td>
</tr>
<tr>
<td>kddb</td>
<td>20.01 M</td>
<td>29.89 M</td>
<td>1e-4</td>
<td>6.18</td>
<td>4.75 GB</td>
<td>0.1</td>
<td>2,000,000</td>
<td>6</td>
</tr>
</tbody>
</table>

We focused our study on the following two aspects: How does StreamSVM compare with SBM and BM for different values of \( C \)? How does the cache size affect the performance of StreamSVM? We also conducted an experiment where we expanded the features on the fly to demonstrate the scalability and flexibility of our framework. Since the objective functions we are minimizing are strongly convex, all optimizers will converge the same solution (within numerical precision) and produce the same generalization performance eventually. Therefore, what we are specifically interested in is the rate at which the objective function decreases and generalization performance increases.

5.1 Results

Varying \( C \)

As \( C \) increases the effect of the regularizer decreases and the non-smooth hinge loss dominates the primal objective. Intuitively, this means that the problem becomes harder to solve for large values of \( C \). Therefore, it is important to test an optimization algorithm across a range of \( C \) values.

We used \( C \in \{0.0001, 0.001, \ldots, 1000.0\} \) to test the performance of StreamSVM and contrast it with SBM and BM. For this experiment we set the maximum number of iterations for all algorithms to be 100; for StreamSVM an iteration is defined as one complete pass through the data by the reader thread. The duality gap tolerance was set to be \( 10^{-3} \). A job is killed by the queue manager on the cluster if it does not finish within 48hrs. Besides the space required to store the weight vector \( w \) and the coefficients \( \alpha \), both algorithms were allowed to use up to 2GB of extra RAM.

We plot the relative objective function value as a function of wall clock time in Figures BUGBUG. The relative objective function is defined as \( (D^* - D^t)/D^* \) where \( D^t \) is the dual objective function output by the optimizer at the \( t \)-th iteration, and \( D^* \) is the largest dual objective function value produced by either SBM, BM, or StreamSVM for the same parameter settings. Note that since we are plotting the y-axis on a log scale some points with relative objective function value of 0 are not displayed.

Note that in almost all cases, StreamSVM outperforms SBM comprehensively. In particular, for values of \( C \) larger than 1 we find that StreamSVM is consistently able to decrease the objective function faster than SBM. In many cases, StreamSVM has produced a solution with relative objective function value of \( 10^{-3} \) or better even before SBM has managed to read and compress the data during the first pass.

Varying \( \Omega \)

Next we study the effect of varying the cache size \( \Omega \) on the performance of StreamSVM. The same setup as the previous experiment was used here but with two notable changes. First, we fixed the value of \( C \) to be 1.0, 0.1, 100.0 and 64.0 for dna, kddb, ocr, and webspam-t datasets respectively. The \( C \) values for kddb and webspam-t follow those used by \cite{1}, and for dna we used the value reported by \cite{2}. For ocr we used cross-validation to pick the value of \( C \) which gives the best generalization performance. Second, we set the cache size to 256MB, 1GB, 4GB, and 16GB respectively.

On all datasets, the performance with 256MB cache was inferior compared to higher cache sizes. On the dna and webspam-t datasets it is clear that increasing the cache size speeds up convergence. On the other hand, the convergence rate on ocr does not seem to depend on the cache size. We investigated this further and found that at the optimal solution only 1450 points were active, that is, their \( 0 < \alpha_i < C \). Consequently having a large cache does not help. On the kddb dataset both the number of data points and dimension of the problem are very large, but the training data is extremely sparse. Even though we can cache a large number of training points, the major bottleneck here is because we need to frequently access random elements of \( w \) (29.89 million entries) and \( \alpha \) (20.01 million entries), both of which are stored as dense vectors. Therefore, the increase in cache size does not significantly improve convergence speed.

Expanding Features on the Fly.

6. DISCUSSION
Figure 2: Relative objective function value as a function of wall clock time for various datasets as the cache size Ω is varied.
7. REFERENCES


**APPENDIX**