Selective Block Minimization for Faster Convergence of Limited Memory Large-Scale Linear Models

Kai-Wei Chang and Dan Roth
Dept. of Computer Science
University of Illinois at Urbana-Champaign, IL USA
{kchang10, danr}@illinois.edu

ABSTRACT

As the size of data sets used to build classifiers steadily increases, training a linear model efficiently with limited memory becomes essential. Several techniques deal with this problem by loading blocks of data from disk one at a time, but usually take a considerable number of iterations to converge to a reasonable model. Even the best block minimization techniques [1] require many block loads since they treat all training examples uniformly. As disk I/O is expensive, reducing the amount of disk access can dramatically decrease the training time.

This paper introduces a selective block minimization (SBM) algorithm, a block minimization method that makes use of selective sampling. At each step, SBM updates the model using data consisting of two parts: (1) new data loaded from disk and (2) a set of informative samples already in memory from previous steps. We prove that, by updating the linear model in the dual form, the proposed method fully utilizes the data in memory and converges to a globally optimal solution on the entire data. Experiments show that the SBM algorithm dramatically reduces the number of blocks loaded from disk and consequently obtains an accurate and stable model quickly on both binary and multi-class classification.

Categories and Subject Descriptors

I.5.2 [**Pattern Recognition**]: Design Methodology—*Classifier design and evaluation*

General Terms

Algorithms, Performance, Experimentation

1. INTRODUCTION

In the past few years, the size of data sets used to build classifiers has steadily increased. Significantly larger amounts of annotated data are available now (e.g., for web applications such as spam filtering), and it has been argued that using more samples and adding more features (e.g, using n-gram in a bag-of-words model) actually boosts performance [2]. In some applications, unlimited amounts of annotated data can be generated (e.g., see Sec. 7). Linear classification models have been shown to handle large amounts of data well, and several optimization techniques (e.g., [3, 4, 5]) have been applied to efficiently train linear models. However, when the data cannot fit into memory, batch learners, which load the entire data during the training process, suffer severely due to disk swapping [1]. In these cases, training techniques that deal well with memory limitations become crucial.

A popular scheme that addresses the memory problem processes the data in an online fashion in multiple rounds (e.g., [6, 7, 4, 8]). However, an online learner, which performs a simple weight update over a single training example at a time, usually requires a large number of iterations to converge. Therefore, when an online learner loads data from disk at each step, disk I/O overhead becomes the bottleneck of the training process. Sec. 7 shows an example where the online learner MIRA [9] takes more than one hour loading data but spends less than one minute updating the model. As discussed in [1], training time consists of (1) the time used to update the model's parameters given the data in memory and (2) the time needed to load data from disk. The machine learning literature focuses on the first issue and neglects the second. Consequently, most algorithms aim at reducing CPU time and the number of passes over the data required to obtain an accurate model (e.g, [4, 5]). However, the cost of disk access could be orders of magnitude higher. *In this paper, we focus on the second issue – saving I/O time by reducing disk access.* To our knowledge, this is the first work that targets minimizing disk access when training a linear model.

There are two ways to reduce the expensive I/O overhead. One is by applying data compression to reduce loading time. The other, orthogonal direction, is algorithmic. It suggests placing more effort on learning from data that is already in memory (e.g, [7, 8]). Recently, [1] proposed a block minimization framework which makes progress in both directions. Their solver splits the data into blocks and stores them in compressed files. By loading and training on a block of samples at a time, it employs a better weight update scheme and achieves faster convergence than online algorithms. However, they assume that all training samples are equally important and treat them uniformly. Consequently, their proposed block minimization method still requires a considerable number of iterations to converge to a reasonable model.

KDD'11, August 21–24, 2011, San Diego, CA, USA.

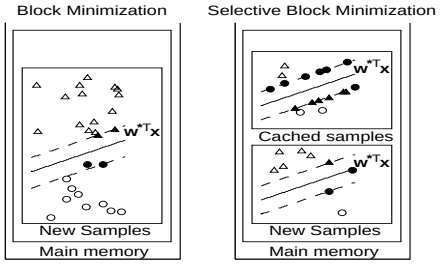


Figure 1: The figure illustrates how a block minimization method wastes resources by considering a lot of unimportant samples (blank circles/triangles), while the selective block minimization (SBM) algorithm focuses on training on informative samples (solid circles/triangles) by caching. The circles/triangles represent the samples drawn from the entire data set. We treat samples closer to the final separator ($\boldsymbol{w}^*$) as more informative (see Sec. 2.2).

This paper develops the selective block minimization (SBM) algorithm, which is shown to significantly improve the I/O cost. SBM can be applied to various linear models, such as SVM and Logistic Regression, when the model can be learned in the dual form. At each step, the proposed algorithm solves the minimization problem on a block of data consisting of new samples loaded from disk along with samples that are already stored in memory, selected from previous steps. The cached samples are selected based on gradient information computed while solving the sub-problem. Figure 1 illustrates the comparison between the proposed algorithm and the block minimization framework (details are given in Sec. (2)). By caching informative samples in memory, the selective block minimization algorithm fully utilizes the memory capacity. The proposed method enjoys the following properties:

- SBM caches informative samples in memory, thus requiring fewer data loads before reaching a given level of performance relative to existing approaches. For example, in one document classification experiment (Sec. 7.2), SBM obtained a model that was as accurate as the optimal one with just a single pass over the data set. For the same data, the current best method [1] requires loading the data from disk 10 times to reach the same performance. That is, SBM saves I/O access by reducing the number of required iterations.
- SBM caches data selectively and thus treats samples non-uniformly, but it still keeps the convergence properties of the block minimization approach.

We present experimental results that exhibit the dramatic reduction in training time achieved by SBM when solving large binary and multi-class problems with limited memory. Moreover, when SBM is required to load each sample into memory once, it is shown to outperform online learners.

The rest of this paper is organized as follows. The selective block minimization algorithm is presented in Sec. 2. Extensions to multi-class classification and streaming data are discussed in Sec. 3 and 4. We address implementation issues in Sec. 5 and analyze SBM's relations with other algorithms for learning with large scale data in Sec. 6. Experimental results are presented in Sec. 7, and Sec. 8 concludes with some discussion. Proofs, additional experiments, and code can be found at
`http://cogcomp.cs.illinois.edu/page/publication_view/660`.

Algorithm 1 A Selective Block Minimization Algorithm for Linear Classification

1. Split data D into subsets $B_j, j = 1 \dots m$.
2. Initialize $\boldsymbol{\alpha} = \mathbf{0}$.
3. Set cache set $\Omega^{1,1} = \emptyset$.
4. For $t = 1, 2, \dots$ (outer iteration)
   For $j = 1, \dots, m$ (inner iteration)
   (a) Read $B_j = (\boldsymbol{x}_i, y_i)_{i=1 \dots m_D}$ from disk.
   (b) Obtain $d^{t,j}$ by exactly or loosely solving (3) on $W^{t,j} = B_j \cup \Omega^{t,j}$, where $\Omega^{t,j}$ is cache.
   (c) Update $\boldsymbol{\alpha}$ by (4).
   (d) *Select cached data* $\Omega^{t,j+1}$ *for next round from* $W^{t,j}$ *(see Sec. 2.2).*

2. SELECTIVE BLOCK MINIMIZATION

In this section, we describe the selective block minimization method by exemplifying it in the context of a L1-loss linear SVM for binary classification. Given a sequence of data $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^l, \boldsymbol{x}_i \in R^n, y_i = \{1, -1\}$, linear SVM considers the following minimization problem:

$$\min_{\boldsymbol{w}} 1/2\|\boldsymbol{w}\|^2 + C\sum\nolimits_{i=1}^{l} \max(1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i, 0), \quad (1)$$

where $C > 0$ is a penalty parameter. Instead of solving (1), in this paper we solve its equivalent dual problem:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) = (1/2)\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} - \boldsymbol{e}^T \boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le C, i = 1, \dots, l, \end{aligned} \quad (2)$$

where $\boldsymbol{e} = [1, \dots, 1]^T$ and $Q_{ij} = y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$. Eq. (2) is a box constraint optimization problem, and each variable corresponds to one training instance. Let $\boldsymbol{\alpha}^*$ be the optimal solution of (2). We call an instance $(\boldsymbol{x}_i, y_i)$ a support vector if the corresponding parameter satisfies $\alpha_i^* > 0$, and an instance i is an unbounded support vector if $0 < \alpha_i^* < C$. We show in Sec. 2.2 that unbounded support vectors are more important during the training process.

With abundant memory, one can load the entire data set and solve (1) or (2) with a batch learner. However, when memory is limited, only part of the data can be considered at a given time. [1] applies a block minimization technique to deal with this situation. We first introduce their method for solving (2). In [1] the data is split into subsets $B_j, j = 1 \dots m$, and a two-level iterative procedure is considered, as follows. Initializing $\boldsymbol{\alpha}$ to zero vectors, the block minimization algorithm generates a sequence of solutions $\{\boldsymbol{\alpha}^{t,j}\}_{t=1 \dots \infty, j=1 \dots m}$. We refer to the step from $\boldsymbol{\alpha}^t = \boldsymbol{\alpha}^{t,1}$ to $\boldsymbol{\alpha}^{t+1} = \boldsymbol{\alpha}^{t,m+1}$ as an *outer* iteration, and $\boldsymbol{\alpha}^{t,j}$ to $\boldsymbol{\alpha}^{t,j+1}$ as an *inner* iteration. For updating from $\boldsymbol{\alpha}^{t,j}$ to $\boldsymbol{\alpha}^{t,j+1}$, [1] considers solving a sub-problem on B_j. After the sub-problem is solved, they update the models and removed all the samples $i, i \in B_j$ from memory, and then load another block into it. As each sample only appears in one B_j, their process treats all the samples equally. Since informative samples are likely to be present in different blocks, the solver wastes time on unimportant samples.

Intuitively, we should select only informative samples and use them to update the model. However, trying to select only informative samples and using only them in training may be too sensitive to noise and initialization and does not yield the appropriate convergence result we show later. Instead, we propose a selective block minimization algorithm. On one hand, as we show, our method maintains the convergence property of block minimization by loading a block of new data B_j from disk at each iteration. On the other hand,

it maintains a cache set $\Omega^{t,j}$, such that informative samples are kept in memory. At each inner iteration, SBM solves the following sub-problem in the dual on $W^{t,j} = B_j \cup \Omega^{t,j}$:

$$\begin{aligned} \boldsymbol{d}^{j,t} = \arg\min_{\boldsymbol{d}} \quad & f(\boldsymbol{\alpha} + \boldsymbol{d}) \\ \text{subject to} \quad & \boldsymbol{d}_{\bar{W}^{t,j}} = 0, 0 \le \alpha_i + d_i \le C, \forall i, \end{aligned} \tag{3}$$

where $\bar{W}^{t,j} = \{1 \dots l\} \setminus W^{t,j}$. Then it updates the model by

$$\boldsymbol{\alpha}^{t,j+1} = \boldsymbol{\alpha}^{t,j} + \boldsymbol{d}^{t,j}. \tag{4}$$

In other words, it updates α_W, while fixing $\alpha_{\bar{W}}$. [1] shows that, by maintaining a temporary vector $\boldsymbol{w} \in R^n$, solving equation (3) only involves the training samples in $W^{t,j}$. After updating the model, SBM selects samples from $W^{t,j}$ to the cache $\Omega^{t,j+1}$ for the next round.

We discuss the cache selection in Sec. 2.2 and show that by solving the sub-problem mentioned above with coordinate descent methods, SBM reduces to a dual coordinate descent method [3] and converges to the global solution of (2), regardless of the cache selection strategy. Note that the samples in cache $\Omega^{t,j+1}$ are only chosen from $W^{t,j}$ and are already in memory. If a sample is important, though, it may stay in the cache for many rounds, as $\Omega^{t,j} \subset W^{t,j}$. Algorithm 1 summarizes the procedure.

We assume here that the cache size and the data partition are fixed. One can adjust the ratio between $|\Omega|$ and $|W|$ during the training procedure. However, if the partition varies during training, we lose the advantage of storing B_j in a compressed file. In the rest of this section, we discuss the design details of the selective block minimization method.

2.1 Solving using Dual Coordinate Decent

While [1] considers solving the linear SVM both in the primal and in the dual forms, here we only consider solving it in the dual form. The reason is that primal methods directly update $\boldsymbol{w}$, which corresponds to the entire data set [4], so they usually assume that samples are treated uniformly. In Algorithm 1, a cached sample appears in several sub-problems within an outer iteration. Therefore, primal methods cannot directly apply to our model. In the dual, on the other hand, each α_i corresponds to a sample. Therefore, it allows us to put emphasis on informative samples, and train the corresponding α_i.

Following the discussion in [1], we use LIBLINEAR [10] as the solver for the sub-problem. LIBLINEAR implements a dual coordinate descent method [3], which iteratively updates variables in $W^{t,j}$ to solve (3) until the stopping condition is satisfied. The following theorem shows the convergence of $\{\boldsymbol{\alpha}^{t,j}\}$ in Algorithm 1:

Theorem 1
Assume that a coordinate descent method is applied to solving (3). *For each inner iteration* (t, j) *assume that*

- *the number of updates,* $\{t^{t,j}\}$, *used in solving the sub-problem is uniformly bounded, and*
- *each* $\alpha_i, \forall i \in W^{t,j}$, *is updated at least once.*

Then, the vector $\{\boldsymbol{\alpha}^{t,j}\}$ *generated by algorithm 1 globally converges to an optimal solution* $\boldsymbol{\alpha}^*$ *at a linear rate or better. That is, there exist* $0 < \mu < 1$ *and* k_0 *such that*

$$f(\boldsymbol{\alpha}^{k+1}) - f(\boldsymbol{\alpha}^*) \le \mu \left(f(\boldsymbol{\alpha}^k) - f(\boldsymbol{\alpha}^*) \right), \forall k \ge k_0.$$

The proof can be modified from [1, Theorem 1].

Sec. 3.1 and Sec. 3.2 in [1] show that the sub-problem can be solved tightly until the gradient-based stopping condition holds, or solved loosely by going through the samples in $W^{t,j}$ a fixed number of rounds. Either way, the first condition in Theorem 1 holds (see [1, Theorem 2]). Moreover, as LIBLINEAR goes through all the data at least once at its first iteration, the second condition is satisfied too. Therefore, Theorem 1 implies the convergence of Algorithm 1. Notice that Theorem 1 holds regardless of the cache selection strategy used.

2.2 Selecting Cached Data

The idea of training on only informative samples appears in the selective sampling literature [11], in active set methods for non-linear SVM (see Sec. 6), and in shrinking techniques in optimization [12]. In this section we discuss our strategy for selecting $\Omega^{t,j}$ from $W^{t,j}$.

During the training, some α_is may stay on the boundary ($\alpha_i = 0$ or $\alpha_i = C$) until the end of the optimization process. Shrinking techniques consider fixing such α_is, thus solving a problem with fewer variables. In the following, we show that those unbounded support vectors with $0 < \alpha_i^* < C$ are more important during the training process. We first show in the following theorem that, if we have an oracle of support vectors, solving a sub-problem using only the support vectors provides the same result as if training is done on the entire data.

Theorem 2
Let $\boldsymbol{\alpha}^*$ *be the optimal solution of* (2), $V = \{i \mid \alpha_i^* = 0\}$ *and* $\bar{V} = \{1 \dots l\} \setminus V$. *If* $\boldsymbol{\alpha}'$ *is the optimal solution of the following problem:*

$$\begin{aligned} & \min_{\boldsymbol{\alpha}} \ (1/2)\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} - \boldsymbol{e}^T \boldsymbol{\alpha} \\ & \textit{subject to } \boldsymbol{\alpha}_{\bar{V}} = 0, 0 \le \alpha_i \le C, i = 1, \dots, l, \end{aligned}$$

then $f(\boldsymbol{\alpha}') = f(\boldsymbol{\alpha}^*)$ *and* $\boldsymbol{\alpha}'$ *is an optimal solution of* (2).

The proof is omitted due to space limitations. Similarly, if we know in advance that $\alpha_i^* = C$, we can solve a smaller problem by fixing $\alpha_i = C$. Therefore, those unbounded support vectors are more important and should be kept in memory. However, we do not know the unbounded support vectors before we solve (2). Therefore, we consider caching unbounded variables ($0 < \alpha_i < C$), which are more likely to eventually be unbounded support vectors.

In dual coordinate descent, α_i is updated in the direction of $-\nabla_i f(\boldsymbol{\alpha})$. Therefore, if $\alpha_i = 0$ and $\nabla_i f(\boldsymbol{\alpha}) > 0$ or if $\alpha_i = C$, and $\nabla_i f(\boldsymbol{\alpha}) < 0$, then α_i is updated toward the boundary. Theorem 2 in [3] implies such α_i will probably stay on the boundary until convergence.

Based on the shrinking strategy proposed in [3], we define the following score to indicate such situations:

$$g(\alpha_i) = \begin{cases} -\nabla_i f(\boldsymbol{\alpha}) & \alpha_i = 0, \nabla_i f(\boldsymbol{\alpha}) > 0, \\ \nabla_i f(\boldsymbol{\alpha}) & \alpha_i = C, \nabla_i f(\boldsymbol{\alpha}) < 0, \\ 0 & \text{Otherwise.} \end{cases}$$

$g(\alpha_i)$ is a non-positive score with the property that the lower its value is, the higher the probability is that the corresponding α_i stays on the boundary. We calculate $g(\alpha_i)$ for each sample i, and select the l_c samples with the higher scores into $\Omega^{t,j+1}$, where l_c is the cache size.

In some situations, there are many unbounded variables, so we further select those with the higher gradient value:

$$g(\alpha_i) = \begin{cases} -\nabla_i f(\alpha) & \alpha_i = 0, \\ \nabla_i f(\alpha) & \alpha_i = C \\ |\nabla_i f(\alpha)| & \text{Otherwise.} \end{cases}$$

To our knowledge, this is the first work that applies a shrinking strategy at the disk level. Our experimental results indicate that our selection strategy significantly improves the performance of a block minimization framework.

3. MULTI-CLASS CLASSIFICATION

Multi-class classification is important in a large number of applications. A multi-class problem can be solved by either training several binary classifiers or by directly optimizing a multi-class formulation [13]. One-versus-all takes the first approach and trains a classifier $\boldsymbol{w}^i$ for each class i to separate it from the union of the other labels. To minimize disk access, [1] proposed an implementation that updates $\boldsymbol{w}^1, \boldsymbol{w}^2 \dots \boldsymbol{w}^k$ simultaneously when B_j is loaded into memory. With a similar technique, Algorithm 1 can be adapted to multi-class problems and maintain separate caches $\Omega_u, u = 1 \dots k$ for each sub-problem. In one-versus-all, the k binary classifiers are trained independently and the training process can be easily parallelized in a multi-core machine. However, in this case there is no shared information among the k binary classifiers so the solver has little global information when selecting the cache. In the following, we investigate an approach that applies SBM in solving a direct multi-class formulation [14] in the dual:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f_M(\boldsymbol{\alpha}) = 1/2 \sum_{u=1}^{k} \|\boldsymbol{w}^u\|^2 + \sum_{i=1}^{l} \sum_{u=1}^{k} e_i^u \alpha_i^u \\ \text{subject to} \quad & \sum_{u=1}^{k} \alpha_i^u = 0, \forall i = 1, \dots, l \\ & \alpha_i^u \le C_{y_i}^u, \forall i = 1, \dots, l, u = 1, \dots, k, \text{ where} \end{aligned} \tag{5}$$

$$\boldsymbol{w}^u = \sum_{i=1}^{l} \alpha_i^u \boldsymbol{x}_i,\ C_{y_i}^u = \begin{cases} 0 & \text{if } y_i \neq u, \\ C & \text{if } y_i = u, \end{cases}\ e_i^u = \begin{cases} 1 & \text{if } y_i \neq u, \\ 0 & \text{if } y_i = u. \end{cases}$$

The decision function is

$$\arg\max_{u=1,\dots,k} (\boldsymbol{w}^u)^T \boldsymbol{x}.$$

The superscript u denotes the class label, and the underscript i represents the sample index. Each sample corresponds to k variables $\alpha_i^1 \dots \alpha_i^k$, and each $\boldsymbol{w}^u$ corresponds to one class. Similar to Algorithm 1, we first split data into m blocks. Then, at each step, we load a data block B_j and train a sub-problem on $W^{t,j} = B_j \cup \Omega^{t,j}$

$$\begin{aligned} \min_{\boldsymbol{d}} \quad & f(\boldsymbol{\alpha} + \boldsymbol{d}) \\ \text{subject to} \quad & \sum_{u=1}^{k} (\alpha_i^u + d_i^u) = 0,\ (\alpha_i^u + d_i^u) \le C_{y_i}^u, \forall i, u \\ & d_i^u = 0, \text{ if } i \notin W^{t,j}, \end{aligned} \tag{6}$$

and update the $\boldsymbol{\alpha}$ by

$$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + \boldsymbol{d},$$

To apply SBM, we need to solve (6) only with the data block $W^{t,j}$. [15] proposed a sequential dual method to solve (5). The method sequentially picks a sample i and solves for the corresponding k variables $\alpha_i^u, u = 1, 2, \dots, k$. In their update, i is the only sample needed. Therefore, their methods can be applied in SBM. We use the implementation of [15] in LIBLINEAR.

In Sec. 2.2, we mentioned the relationship between shrinking and cache selection. For multi-class SBM, [10, Appendix E.5] describes a shrinking technique implemented in LIBLINEAR. If α_i^u is on the boundary (i.e., $\alpha_i^u = C_i^u$) and satisfies certain conditions, then it is shrunken. To select a cache set, we consider a sample 'informative' if a small fraction of its corresponding variables $\alpha_i^u, u = 1, 2, \dots, k$ are shrunken. We keep such samples in the cache $\Omega^{t,j}$. Sec. 7 demonstrates the performance of SBM in solving (5).

4. LEARNING FROM STREAMING DATA

Learning over streaming data is important in many applications when the amount of data is just too large to keep in memory; in these cases, it is difficult to process the data in multiple passes and it is often treated as a stream that the algorithm needs to process in an online manner. However, online learners make use of a simple update rule, and a single pass over the samples is typically insufficient to obtain a good model. In the following, we introduce a variant of SBM that considers Algorithm 1 with only one iteration. Unlike existing approaches, we accumulate the incoming samples as a block, and solve the optimization problem one data block at a time. This strategy allows SBM to fully utilize its resources (learning time and memory capacity) and to obtain a significantly more accurate model than in standard techniques that load each sample from disk only once.

When SBM collects a block of samples B from the incoming data, it sets the corresponding $\alpha_i, i \in B$ to 0. Then, as in Algorithm 1, it updates the model by solving (3) on $W = B \cup \Omega$, where Ω is the cache set collected from the previous step. In Sec. 2, we described SBM as processing the data multiple times to solve Eq. (2) exactly. This required maintaining the whole $\boldsymbol{\alpha}$. When the number of instances is large, storing $\boldsymbol{\alpha}$ becomes a problem (see Sec. 5.2). However, when dealing with streaming data each sample is loaded once. Therefore, if a sample i is removed from memory, the corresponding α_i will not be updated later, and keeping $\alpha_i, i \in W$ is sufficient. Without the need to store the whole $\boldsymbol{\alpha}$, the solver is able to keep more data in memory.

The SBM for streaming data thus obtains a feasible solution that approximately solves (2). Compared to other approaches for streaming data, SBM has two advantages:

- By adjusting the cache size and the tightness of solving the sub-problems, we can obtain models with different accuracy levels. Therefore, the solver can adjust to the available resources.
- As we update the model on both a set of incoming samples and a set of cached samples, the model is more stable than the model learned by other online learners.

We study its performance experimentally in Sec. 7.2.

5. IMPLEMENTATION ISSUES

[1] introduces several techniques to speed up block minimization algorithms. Some techniques such as data compression and loading blocks in a random order can also be applied to Algorithm 1. In the following, we further discuss implementation issues that are relevant to SBM.

5.1 Storing Cached Samples in Memory

In Step 4(d) of Algorithm 1, we update the cache set $\Omega^{t,j}$ to $\Omega^{t,j+1}$ by selecting samples from $W^{t,j}$. As memory is limited, non-cached samples $i \notin \Omega^{t,j+1}$ are removed from memory. If data is stored in a continuous array[1] we need a careful design to avoid duplicating samples.

Assume that $W^{t,j}$ is stored in a continuous memory chunk MEM_W. The lower part of MEM_W, MEM_Ω, stores the samples in $\Omega^{t,j}$, and the higher part stores B_j. In the following we discuss a way to update the samples in MEM_Ω from $W^{t,j} = \Omega^{t,j} \cup B_j$ to $W^{t,j+1} = \Omega^{t,j+1} \cup B_{j+1}$ without using extra memory. We want to update MEM_W and store

[1]This operation can be performed easily if the data is stored in a linked list. However, accessing data in a linked list is slower than in an array.

$\Omega^{t,j+1}$ in its lower part. However, as $\Omega^{t,j+1}$ is selected from $\Omega^{t,j} \cup B_j$, which is currently in Mem_W, we need to move the samples in such an order that the values of samples in $\Omega^{t,j+1}$ will not be covered. Moreover, as data is stored in a sparse format, the memory consumption for storing each sample is different. We cannot directly swap the samples in memory. To deal with this problem, we first sort the samples in $\Omega^{t,j+1}$ by their memory location. Then, we sequentially move each sample in $\Omega^{t,j+1}$ to a lower address of MEM_W and cover those non-cached samples. This way, we form a continuous memory chunk $\Omega^{t,j+1}$. Finally, we load B_{j+1} after the last cached sample in $\Omega^{t,j+1}$; MEM_W stores $W^{t,j+1}$ now.

5.2 Dealing with a Large α

When the number of examples is huge, storing $\boldsymbol{\alpha}$ becomes a problem. This is especially the case in multi-class classification, as the size of $\boldsymbol{\alpha}$ grows rapidly since each sample corresponds to k elements in $\boldsymbol{\alpha}$. In the following we introduce two approaches to cope with this situation.

1. Storing $\alpha_i, i \notin W$ in a hash table: In real world applications, the number of support vectors is sometimes much less than the number of samples. In these situations, many α_is will stay at 0 during the whole training process. Therefore, we can store the α_is in a sparse format. That is, we store non-zero $\alpha_i > 0$ in a hash table. We can keep a copy of the current working $\alpha_i, i \in W$ in an array, as accessing data from a hash table is slower than from an array. The hash strategy is widely used in developing large scale learning systems. For example, VW uses a hash table to store the model $\boldsymbol{w}$ when n is huge.

2. Storing $\alpha_i, i \notin W$ in disk: In solving a sub-problem, $\alpha_i, i \notin W^{t,j}$ are fixed. Therefore, we can store them in disk and load them when the corresponding samples $B_j, i \in B_j$ are loaded. We maintain m files to save those α_is which are not involved in the current sub-problem. Then, we save the α_i to the jth file when the corresponding sample $i \in B_j$ is removed from memory. To minimize disk access, the α_is can be stored in a sparse format, and only the α_is with non-zero value are stored.

6. RELATIONS WITH OTHER METHODS

In Sec. 1 and Sec. 2 we discussed the differences between the block minimization framework and our Algorithm 1. In the following, we review other approaches to large scale classification and their relations to Algorithm 1.

Online learning: Online learning is a popular approach to learning from huge data sets. An online learner iteratively goes over the entire data and updates the model as it goes. A commonly held view was that online learners efficiently deal with large data sets due to their simplicity. In the following, we show that the issues are different when the data does not fit in memory. At each step, an online learner performs a simple update on a single data instance. The update done by an online learner is usually cheaper than the one done by a batch learner (e.g., Newton-type method). Therefore, even though an online learner takes a large number of iterations to converge, it could obtain a reasonable model quickly [16, 17]. However, when the data cannot fit into memory, we need to access each sample from disk again every round. An online method thus requires large I/O due to the large number of iterations required. One may consider a block of data at a time to reduce the number of iterations.

Some online learning algorithms have considered solving a block of data at a time. For example, Pegasos [4] allows a block-size update when solving (1) with projected stochastic gradient descent. However, as they assume data points are randomly drawn from the entire data, they cannot do multiple updates on one block. Therefore, it still cannot fully utilize the data in memory. Some online learning methods have considered a caching technique. [18] suggested periodically updating the model on samples the model gets wrong. However, their caching technique is ad-hoc and has no convergence guarantees. To our knowledge, SBM is the first method that uses caching and *provably converges* to (1).

Parallelizing batch algorithm: Although parallel algorithms such as [19, 20] can scale up to huge problems, the communication across machines still incurs a large overhead. Moreover, the aforementioned algorithms still require that training is done on a single machine, so we do not consider parallel solutions in this work. Recently there have been proposals for solving large problems by parallelizing online methods using multi-core machines [21]. However, these methods only shorten the learning time and do not solve the I/O problem.

Reducing data size: Sub-sampling and feature pruning techniques as in [22, 23] reduce the size of the data. In some situations, training on a subset of the data is sufficient to generate a reasonable model. But in other situations, the huge amount of data is already preprocessed and further pruning may degrade the performance. In fact, in SBM we start from a model trained on a randomly drawn data block, and iteratively improve the model by loading other sample blocks. Therefore, its performance should be at least as good as random sub-sampling.

Bagging models trained on a subset of the data: Bagging [24] is an algorithm that improves test performance by combining models trained on subsets of data. To train on large data, a bagging algorithm randomly picks k-subsets of the entire data, and trains k models $\{w^1, w^2, \ldots, w^k\}$ separately on each subset. The final model is obtained by averaging the k models. Some analysis was provided recently in [25]. Although a bagging algorithm can sometimes learn a good model, it does not solve Eq. (1). Therefore, in practice, it is not clear if a bagging model can obtain as good a classifier as obtained by solving (1).

Selective sampling: Selective sampling and active learning methods select informative samples for learning a model [26, 27]. Even though their goal is different, these methods can be applied when training on large data sets (See Sec. 6 in [22]). Starting from a subset of the data, selective sampling methods iteratively select the samples that are close to the current margin, and update the model on those samples. To compute the margin accurately, a selective sub-sampling method may require access to the entire data from disk, which is expensive. In contrast, SBM loads a data block not only for selection but also for training. It selects samples into the cache only if they are already in memory. Therefore, it suffers a very small overhead when selecting samples. Moreover, selective sampling is not guaranteed to converge to (1), while SBM converges linearly.

Related methods for non-linear models: In solving non-linear models, data is usually mapped into a high dimensional feature space via a kernel trick. When the kernel matrix goes beyond memory capacity, an active set method (e.g., [28, 29, 30]) maintains an active set A and updates only the variables $\alpha_i, i \in A$ each time. If $|A|$ is small, the

Table 1: Data statistics: We assume data is stored in a sparse representation. Each feature with non-zero value takes 16 bytes on a 64-bit machine due to data structure alignment. l, n, and #nonzeros are number of instances, features, and non-zero values in each training set, respectively. l_t is the number of instances in the test set. Memory is the memory consumption needed to store the entire data. #nBSV shows the number of unbounded support vectors ($0 < \alpha_i < C$) in the optimal solution. #classes is the number of class labels.

Data set	l	l_t	n	#nonzeros	Memory (Bytes)	C	#nBSV	#classes
webspam	280,000	70,000	16,609,143	1,043,792,623	16,700,681,968	64	7071	2
kddcup10	19,264,093	748,401	29,890,095	566,345,790	9,061,532,640	0.1	2,775,946	2
prep	60,000,000	5,710,649	11,148,531	918,704,364	14,699,269,824	0.1	-	10

corresponding kernel entries can be stored in memory and the active set method yields fast training. Active set methods are related to our work as they also select samples during training. However, the memory problem of dealing with non-linear models is different from the one we face. Non-linear solvers usually assume that samples are stored in memory, while the entire kernel cannot be stored. Therefore, there is no restriction on accessing data. In our case, the samples can only be loaded sequentially from disk.

Another algorithm, Forgetron [31], is related to our work. However, the focus there is different. When training a model with a non-linear kernel, the model is represented by some samples (support vectors). When they have too many samples to store in memory, Forgetron selects the samples which can represent the current model best. In the linear case, $\boldsymbol{w}$ can be directly stored. In our case, on the other hand, the selection process is done in order to speed up the training rather than to just represent the model more concisely.

7. EXPERIMENTAL STUDY

In this section, we first show that SBM provides an improvement over the block minimization algorithm (BMD) in [1]. Then, we demonstrate that SBM converges faster than an online learner. Finally, we investigate the setting in which each sample is loaded from disk only once and show that SBM deals better with streaming data.

We consider two large binary classification data sets, webspam and kddcup10, and one multi-class data set, prep. webspam is a document collection; kddcup10 is taken from educational applications and has been used in the data mining challenge.[2] prep is a 10-classes classification task, aiming at predicting the correct preposition in an English sentence with a bag-of-words model [32].[3] In kddcup10 and webspam, the feature vectors are scaled to unit length, so that $\forall i, \|\boldsymbol{x}_i\| = 1$. prep takes binary features, and we prune the features which appear less than 5 times in all samples. Table 1 lists the data statistics. As shown in [33], applying sub-sampling or bagging downgrades the test performance on kddcup10 and webspam. Each data set is stored as an ASCII file before the training process.

We tune the penalty parameter C using a five-fold cross-validation on the training data and report the performance of solving (2) with the best parameter. This setting allows SBM for linear SVM to be compared with other learning packages such as VW. The implementation of BMD and SBM is in C/C++[4] with double precision in a 64-bit machine. We restrict each process to use at most 2GB memory.

7.1 Training Time and Testing Accuracy

Comparison between SBM and BMD: We compare SBM with the block minimization algorithm in the dual (BMD) [1], as both the algorithms consider a block of data at a time. For a fair comparison, we take the largest block size ($|D_{\text{BM}}| = \max_j |B_j|$) in BMD as a reference, and adjust SBM to use the same amount of memory during training. We consider three variants of Algorithm 1: SBM-1/2, SBM-1/3, and SBM-1/2-rand. SBM-1/2 and SBM-1/2-rand load half the memory with new data and the rest is kept for the cache ($|B_j| = |\Omega| = 1/2|D_{\text{BM}}|$). SBM-1/3 reserves 1/3 of the memory capacity for cache and loads the rest with new data ($|\Omega| = 1/3|D_{\text{BM}}|$ and $|B_j| = 2|\Omega|$). Both SBM-1/2 and SBM-1/3 select the cached sample using the gradient information as described in Sec. 2.2 and 3. SBM-1/2-rand selects the cached sample by picking randomly from the samples in memory (W). All sub-problems in BMD and SBM-* are solved by a dual coordinate method with 10 rounds ($t^{i,j} = 10$). We split both webspam and kddcup10 into 12 blocks ($m = 12$)[5] and split prep into 15 blocks in BMD. The cache size and block size of SBM-* are adjusted according to BMD. Each block is stored in a compressed file. The compression introduces additional cost in storing compressed files but it reduces the load time of each block (see [1, Sec. 5.1]). We report the *wall clock* time in all experiments including the initial time to generate the compressed files (the time to read data from an ASCII file, split it, and store the compressed data in disk.)

The number of α_i^us in prep is 600 million. Therefore, storing all α_i^u takes 4.8GB memory which exceeds the memory restriction. Following the discussion in Sec. 5.2, we store the unused α_i^u in files for BMD and SBM-*.

To check the convergence speed of each method, we compare the relative difference between the dual objective function value (2) and the optimum ($f(\boldsymbol{\alpha}^*)$),

$$|(f(\boldsymbol{\alpha}) - f(\boldsymbol{\alpha}^*))/f(\boldsymbol{\alpha})|\,, \tag{7}$$

over time. We are also interested in the time it takes each method to obtain a reasonable and stable model. Therefore, we compare the difference between the performance of the current model (acc($\boldsymbol{w}$)) and the best test accuracy among all methods (acc*)

$$(\text{acc}^* - \text{acc}(\boldsymbol{w}))\%. \tag{8}$$

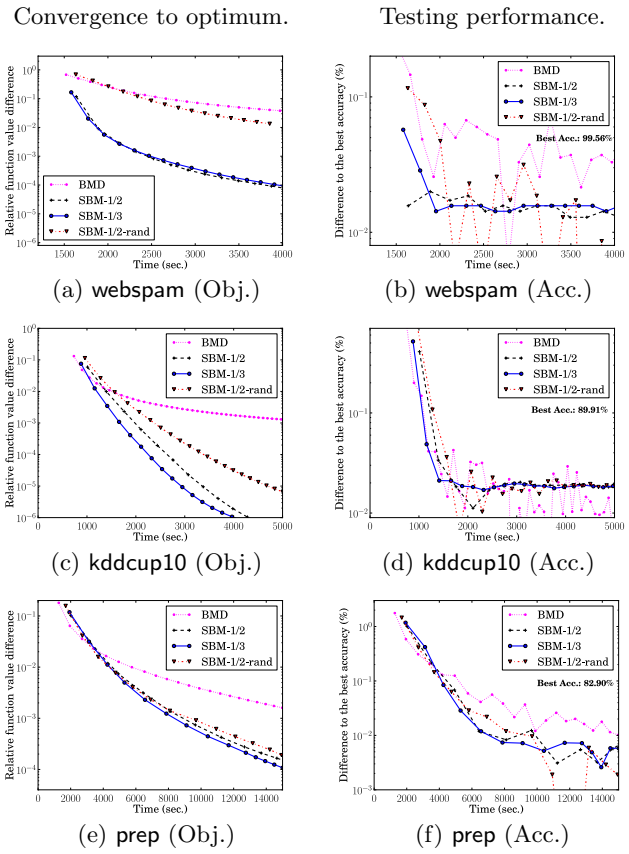
Note that tightly solving (1) is not guaranteed to have the best test performance. Some solvers may reach a higher accuracy before the model converges.

[2] kddcup10 is preprocessed from the second data set in KDD Cup 2010. webspam and kddcup10 can be downloaded at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

[3] prep is extracted from Wikipedia and New York Times. As the annotation can be automatically generated from articles (assuming the given text is correct), the supervised data can be very large.

[4] The experimental code is modified from [1] `http://www.csie.ntu.edu.tw/~cjlin/liblinear/exp.html`.

[5] Although the data size of webspam is larger than kddcup10. kddcup10 has a huge number of samples and features, and it takes about 500MB to store the model.



(a) webspam (Obj.) (b) webspam (Acc.)

(c) kddcup10 (Obj.) (d) kddcup10 (Acc.)

(e) prep (Obj.) (f) prep (Acc.)

Figure 2: Comparison between the proposed methods (SBM-*) and the block minimization algorithm (BMD) [1]. Left column: the relative function value difference (7) to the minimum of (2). Right column: the test accuracy difference (8) from the best achievable accuracy. Time is in seconds, and the y-axis is in log-scale. Each marker indicates one outer iteration. As all the solvers access all samples from disk once at each outer iteration, the markers indicate the amount of disk access. It is shown that SBM-* converges faster and obtains a more accurate and stabler model quickly.

Figure 2 shows that all the SBM-* outperform BMD significantly. By selecting cached samples using the gradient information, our key methods, SBM-1/2 and SBM-1/3 perform even better. In particular, when the unbounded support vectors can fit into memory, SBM-1/2 and SBM-1/3 keep most of them in cache during the training process and thus enjoy fast convergence.[6] Consequently, the models generated are more stable in test. The performance of SBM-1/2 and SBM-1/3 are similar. This indicates that the model is not sensitive to the ratio between block size and cache size. Note that SBM is more general than BMD, as it is reduced to BMD when the cache size is zero.

It's interesting to note that even when the cached set is selected randomly, SBM-1/2-rand is still faster than BMD. One possible reason is that partitioning the data affects the performance of the solver. BMD uses a fixed partition $B_j, \forall j$, throughout the training process. For SBM, although, $\forall j, B_j$ is fixed as well, the cached set Ω varies at each iteration.

[6]E.g., in figure 2(a), more than 98% of the unbounded support vectors are stored in cache after 2 iterations.

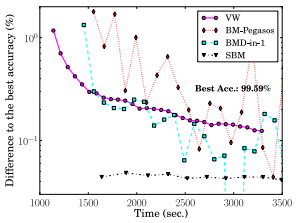


Figure 3: A comparison between SBM and online learning algorithms on webspam. We show the difference between test performance and the best accuracy achievable (8). Time is in seconds. The markers indicate the time it takes each solver to access data from disk once. Except for VW, each solver takes about 90 sec. in each round to load data from the compressed files.

Therefore, SBM avoids wasting time on a bad data block. When the number of bounded support vectors is large, the selection of Ω is less important since drawing a sample randomly already has a high probability of including bounded support vectors in the cache (see figure 2(c)). SBM-* works better in binary classification than in multi-class classification, as sub-problems of the multi-class formulation take more learning time. Thus, the I/O overhead plays a smaller role at training time.

Comparison between SBM and online learners: In Sec. 6, we mentioned that people in the community believe that online learners are fast for large data sets since they make use of a simple update rule. However, as we show, online learners use more I/O time when the data is not in memory. We compare the following solvers:

- VW[7]: an online learning package (version 5.1).[8] It implements a stochastic gradient descent method [34].
- BM-Pegasos: A version of Pegasos implemented under a block minimization framework [1, 4].
- BMD-in-1: Similar to BMD but the inner solver goes through the samples in each sub-problem only once.
- SBM: The same as SBM-1/2 in the previous experiment.

We implemented Pegasos under the block minimization framework such that it saves considerable I/O time by loading data from a compressed file. Although VW also considers a hinge loss function, it does not converge to (2). Therefore, its final accuracy is different from that of the others. Except for SBM, all the methods update the model on each sample once per outer iteration. BMD-in-1 takes the same update rule as the passive-aggressive online algorithm [9], but it keeps $\boldsymbol{\alpha}$ and its solution converges to (2). Except for VW, all methods take about 90 seconds to access each sample from the compressed files. VW has a faster implementation[9] for loading data and takes about 70 seconds per outer iteration.

Figure 3 shows the test performance over time (8) on webspam. SBM converges faster than other online methods to the final performance. *The reason is that an online learner spends most of its time loading data instead of learning a*

[7] `https://github.com/JohnLangford/vowpal_wabbit`

[8]We use the latest version 5.1, and set the parameter as -b 24 --loss_function hinge --initial_t 80000 --power_t 0.5 -l 1000 --compressed as the author suggested.

[9] VW uses single floating point precision while our implementations use double precision. Although using single precision reduces the training time, numerical inaccuracy may occur. In addition, VW is a multi-threaded program. It uses one thread for loading data and another thread for learning.

Table 2: Training time (*wall clock time*) and testing accuracy for each solver *when each sample is loaded into memory only once*. We show both the loading time (the time to read data from an ASCII file) and the learning time. The reference model is obtained by solving (1) until it converges (see text). We show that by updating the model on one data block at a time, SBM obtains an accurate model for streaming data. Note that the time reported here is different from the time required to finish the first iteration in Figure 2 (see text). Some entries in the table are missing. BM-Pegasos and VW do not support multi-class classification. Also, it is difficult to access the I/O and learning time of VW since it is a multi-threaded program. The top three methods are implemented in Java, and the rest are in C/C++.

Method	webspam				kddcup10				prep			
	Training Time (sec.)			Acc.	Training Time (sec.)			Acc.	Training Time (sec.)			Acc.
	I/O	Learning	Total		I/O	Learning	Total		I/O	Learning	Total	
Perceptron	5078	31	5108	98.66	1403	42	1448	84.93	1843	903	2753	76.10
MIRA [9]	5392	27	5419	98.73	1265	44	1312	86.14	1829	816	2652	76.81
CW [8, 35]	4698	59	4757	99.49	1439	99	1535	89.07	1820	1121	2949	77.70
VW	-	-	507	98.42	-	-	223	85.25	-	-	-	-
BM-Pegasos	470	9	479	97.87	187	20	207	87.48	-	-	-	-
BMD	472	60	532	99.48	181	201	382	88.62	265	649	914	80.06
SBM-in-3	484	81	565	**99.55**	181	151	332	89.54	321	778	1099	81.61
SBM	473	177	650	**99.55**	180	442	634	**89.67**	354	1956	2310	**81.80**
Reference	-	-	-	99.55	-	-	-	89.90	-	-	-	82.90

model (see Sec. 6). For example, BM-Pegasos spends only 15 seconds updating the model. Therefore, it spends 85% of the training time loading data. In contrast, the learning time per iteration in SBM is 180 seconds. Thus, SBM uses 2/3 of the training time to update the model and consequently converges faster.

7.2 Experiments on Streaming Data

In Sec. 4, we introduced a variation of SBM that deals with streaming data. Now we investigate its performance.

In addition to the solvers BMD, SBM, BM-Pegasos and VW introduced in previous experiments, we further compare them to the implementations in [35].[10] The package currently supports the following solvers for data streams:

- MIRA: an online learning algorithm proposed in [9].
- CW: a confidences weighted online algorithm [8, 35].
- Perceptron: an online learning algorithm.

In addition, we consider a setting of SBM, SBM-in-3, in which we only allow 3 rounds when solving each sub-problem (the general setting allows 10 rounds). The training time of this version of SBM is similar to the training time of BMD. Note that MIRA, CW and Perceptron are implemented in JAVA, and the rest of the solvers are implemented in C/C++. For multi-class problems, we used the top-1 approach [35] in MIRA, CW and Perceptron.

Table 2 lists the training time and test accuracy when each sample is loaded from disk only once. To see if loading data once can achieve a result similar to running over the data multiple times, we show a reference model in the table, which solves Eq. (1) until the model converges. We provide both the time to load data from an ASCII file (I/O) and the update time of the model (Learning). For each solver we focus on the ratio of loading to learning time rather than on comparing the absolute times across different solvers. Note that the time reported in Table 2 is different from the time to complete the first round in Figure 2. Figure 2 includes the time it takes to generate compressed files from an ASCII file. However, for streaming data, each sample is loaded into memory once and there is no need to store data in compressed files, resulting in shorter I/O time.

The results show that SBM is able to utilize the memory capacity and spend more learning time to achieve an accurate model. It even achieves a model that is almost as accurate as the reference model on webspam. As the I/O is expensive, for all the methods, the loading time takes a large fraction of the total training time. Online methods such as MIRA and Perceptron use a simple update rule. Their learning time is indeed small, but the model learned with a single pass over the samples is far from converging. CW uses a better update rule and achieves a more accurate model, but the performance is still limited. In contrast, both SBM and BMD are able to update on a data block at a time, thus taking more information into account. Therefore, they achieve a better result. Comparing SBM-in-3 to BMD we see that SBM-in-3 has a similar training time to that of BMD, but learns a model with better accuracy. As shown, by keeping informative samples in memory, SBM utilizes the memory storage better. When the resources are limited, users can adjust the test performance of SBM by adjusting the memory usage and the accuracy level of solving the sub-problems.

8. DISCUSSION AND CONCLUSION

In summary, we presented a selective block minimization algorithm for training large-scale linear classifiers under memory restrictions. We also analyzed the proposed algorithm and provided extensive experimental results. To reduce the amount of disk access, we suggested caching informative samples in memory and updating the model on the important data more frequently. We show that, even though the proposed method treats data points non-uniformly, it provably converges to the global optimum on the entire data. Our experimental results on binary, multi-class and streaming data, show the significant advantage of SBM over block minimization and online algorithms. SBM converges quicker to a more accurate and stable model.

We note that these methods can be applied more broadly. For example, it can be applied for solving L2-loss SVM,

[10] `http://www.cs.jhu.edu/~mdredze/`. The original package takes a very long time to train on the webspam data set. It might be due to the deep class hierarchy in the package. We fixed the problem and sped up the package.

regularized least square problems, and logistic regression, by applying the corresponding dual solvers [3, 36]. Moreover, note that in this work we considered sparse data stored in row format. Therefore, we could split the data into blocks and access particular instances. When data is stored in a column format, splitting can only be done feature by feature. It would be interesting to investigate how to extend our method to this scenario.

Acknowledgments This research is supported by the Defense Advanced Research Projects Agency (DARPA) Machine Reading Program under Air Force Research Laboratory (AFRL) prime contract no. FA8750-09-C-0181 and by the DARPA under the Bootstrap Learning Program. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of the DARPA, AFRL, ARL or the US government.

9. REFERENCES

[1] H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin, "Large linear classification when data cannot fit in memory," in *ACM KDD*, 2010.

[2] G. Loosli, S. Canu, and L. Bottou, "Training invariant support vector machines using selective sampling," in *Large Scale Kernel Machines* (L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, eds.), pp. 301–320, Cambridge, MA.: MIT Press, 2007.

[3] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A dual coordinate descent method for large-scale linear SVM," in *ICML*, 2008.

[4] S. Shalev-Shwartz, Y. Singer, and N. Srebro, "Pegasos: primal estimated sub-gradient solver for SVM," in *ICML*, 2007.

[5] T. Joachims, "Training linear SVMs in linear time," in *ACM KDD*, 2006.

[6] J. Langford, L. Li, and T. Zhang, "Sparse online learning via truncated gradient," *JMLR*, vol. 10, pp. 771–801, 2009.

[7] P. Rai, H. Daumé III, and S. Venkatasubramanian, "Streamed learning: One-pass SVMs," in *IJCAI*, 2009.

[8] M. Dredze, K. Crammer, and F. Pereira, "Confidence-weighted linear classification," in *ICML*, 2008.

[9] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online passive-aggressive algorithms," *JMLR*, 2006.

[10] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *JMLR*, vol. 9, pp. 1871–1874, 2008.

[11] G. Loosli, S. Canu, and L. Bottou, "Training invariant support vector machines using selective sampling," in *Large Scale Kernel Machines*, pp. 301–320, Cambridge, MA.: MIT Press, 2007.

[12] T. Joachims, "Making large-scale SVM learning practical," in *Advances in Kernel Methods - Support Vector Learning*, MIT Press, 1998.

[13] S. Har-Peled, D. Roth, and D. Zimak, "Constraint classification for multiclass classification and ranking," in *NIPS*, 2003.

[14] K. Crammer and Y. Singer, "On the learnability and design of output codes for multiclass problems," in *COLT*, 2000.

[15] S. S. Keerthi, S. Sundararajan, K.-W. Chang, C.-J. Hsieh, and C.-J. Lin, "A sequential dual method for large scale multi-class linear SVMs," in *ACM KDD*, 2008.

[16] L. Bottou and Y. L. Cun, "Large scale online learning," in *NIPS*, 2004.

[17] S. Shalev-Shwartz and N. Srebro, "SVM optimization: inverse dependence on training set size," in *ICML*, 2008.

[18] M. Collins and B. Roark, "Incremental parsing with the perceptron algorithm," in *ACL*, 2004.

[19] K. Woodsend and J. Gondzio, "Hybrid mpi/openmp parallel linear support vector machine training," *JMLR*, vol. 10, pp. 1937–1953, 2009.

[20] E. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui, "Parallelizing support vector machines on distributed computers," in *NIPS 21*, 2007.

[21] J. Langford, A. J. Smola, and M. Zinkevich, "Slow learners are fast," in *NIPS*, 2009.

[22] H. Yu, J. Yang, and J. Han, "Classifying large data sets using SVMs with hierarchical clusters," in *ACM KDD*, 2003.

[23] J. L. Balcázar, Y. Dai, and O. Watanabe, "A random sampling technique for training support vector machines," in *ALT*, 2001.

[24] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, pp. 123–140, August 1996.

[25] M. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized stochastic gradient descent," in *NIPS 23*, pp. 2595–2603, 2010.

[26] F. Pérez-Cruz, A. R. Figueiras-Vidal, and A. Artés-Rodríguez, "Double chunking for solving SVMs for very large datasets," in *Proceedings of Learning 2004, Spain*, 2004.

[27] S. Tong and D. Koller, "Support vector machine active learning with applications to text classification," *JMLR*, vol. 2, pp. 45–66, 2002.

[28] S. V. N. Vishwanathan, A. J. Smola, and M. N. Murty, "Simplesvm," in *ICML*, 2003.

[29] K. Scheinberg, "An efficient implementation of an active set method for svms," *JMLR*, vol. 7, pp. 2237–2257, 2006.

[30] A. Bordes, S. Ertekin, J. Weston, and L. Bottou, "Fast kernel classifiers with online and active learning," *JMLR*, vol. 6, pp. 1579–1619, 2005.

[31] O. Dekel, S. Shalev-Shwartz, and Y. Singer, "The forgetron: A kernel-based perceptron on a budget," *SIAM Journal on Computing*, vol. 37, pp. 1342–1372, January 2008.

[32] A. Rozovskaya and D. Roth, "Generating confusion sets for context-sensitive error correction," in *EMNLP*, 2010.

[33] H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin, "Large linear classification when data cannot fit in memory," tech. rep., National Taiwan University, 2011. `http://www.csie.ntu.edu.tw/~r96050/papers/bm.pdf`.

[34] Y. Freund, R. E. Schapire, Y. Singer, and M. K. Warmuth, "Using and combining predictors that specialize," in *STOC*, 1997.

[35] K. Crammer, M. Dredze, and A. Kulesza, "Multi-class confidence weighted algorithms," in *EMNLP*, 2009.

[36] H.-F. Yu, F.-L. Huang, and C.-J. Lin, "Dual coordinate descent methods for logistic regression and maximum entropy models," *Machine Learning*, 2011. To appear.

[37] Z.-Q. Luo and P. Tseng, "On the convergence of coordinate descent method for convex differentiable minimization," *J. Optim. Theory Appl.*, vol. 72, no. 1, pp. 7–35, 1992.

APPENDIX

A. PROOF OF THEOREM

Proof of Theorem 1

We follow the proof in [1, Theorem 1]. Theorem 2.1 of [37] analyzes the convergence of coordinate descent methods. The theorem requires several conditions on (1) and an almost cyclic update rule. [3, Theorem 1]) have proved that Eq. (1) satisfies the required conditions. To satisfy the second condition, we need to prove that, in the coordinate decent method, there exists an integer T such that every variable is updated at least once between the rth iteration and the $(r+T)$th iteration, for all r. From Algorithm 1, for each α_i, there exists an index j such that $\alpha_i \in B_j$ and $B_j \subset W^{t,j}$. Therefore, the two assumptions imply α_i is updated at least once and the number of updates is bounded at each outer iteration. Therefore, Algorithm 1 enjoys both global and linear convergence.

Proof of Theorem 2

Let $\boldsymbol{\alpha}^*$ be the optimal solution of (2), and $\boldsymbol{\alpha}'$ is the optimal solution of the following optimization problem:

$$\begin{aligned} &\min_{\boldsymbol{\alpha}} \ \frac{1}{2}\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha} \\ &\text{subject to } \boldsymbol{\alpha}_{\bar{V}} = 0, 0 \le \alpha_i \le C, i = 1, \ldots, l. \end{aligned} \tag{9}$$

The solution space of (2) is larger than (9). Therefore, $f(\boldsymbol{\alpha}')$ is a feasible solution of (2). Since $\boldsymbol{\alpha}^*$ is an optimum of (2),

$$f(\boldsymbol{\alpha}^*) \le f(\boldsymbol{\alpha}'). \tag{10}$$

Since $\boldsymbol{\alpha}^*$ satisfies the constraint $\boldsymbol{\alpha}_{\bar{V}} = 0$, $\boldsymbol{\alpha}^*$ is a feasible solution of (9). As $\boldsymbol{\alpha}'$ is an optimum of (9),

$$f(\boldsymbol{\alpha}') \le f(\boldsymbol{\alpha}^*). \tag{11}$$

Combining (10) and (11), we obtain

$$f(\boldsymbol{\alpha}') = f(\boldsymbol{\alpha}^*). \tag{12}$$

B. ADDITIONAL EXPERIMENTS

In this section, we demonstrate two additional experiments. We first show the performance of SBM, BMD and an online leanring package, VW under the single precision floating point arithmetic. Then, we investigate the influence of the cache size on the convergence of SBM.

B.1 Experiments Under The Single Precision Floating Point Arithmetic

Some packages (e.g., VW) consider storing floating point values in single precision. Using single precision has the following advantages:

- Each feature with non-zero value needs only 8 bytes. Therefore, the size of compressed cached files and the memory usage are reduced.
- Calculations with single precision are usually faster than with double precision.

However, using single precision sometimes induces numerical inaccuracy. For the sake of robustness, some packages (e.g., LIBLINEAR) use double precision to store the data and the model.

In Section 7.1, we follow the setting in [1] to implement SBM and BM-* in double precision and keep VW using single precision. Therefore, Figure 3 shows that the time cost per iteration of VW is smaller than the other methods implemented in double precision. In this section, we study the test performance along time of each method under the single precision arithmetic on webspam data set. Besides the solvers showed in Figure 3, we include a dual block minimization method that solving the sub-problem with 10 rounds (BMD-in-10). Figure 4 shows the results. By using single precision, all the methods under the block minimization framework (SBM and BM-*) reduce about 25% training time. In the figure, VW, BM-Pegasos and BMD-in-1 take small effort at each iteration, but they require considerable number of iterations to achieve a reasonable model. In contrast, the cost of BMD-in-10 and SBM per iteration is high, but they converge faster.

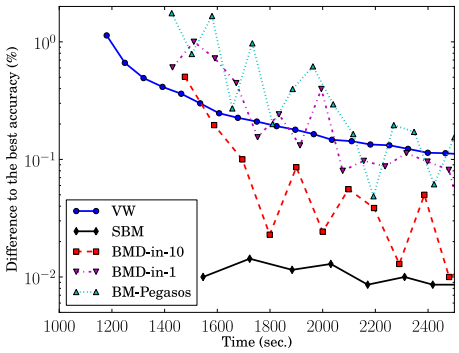
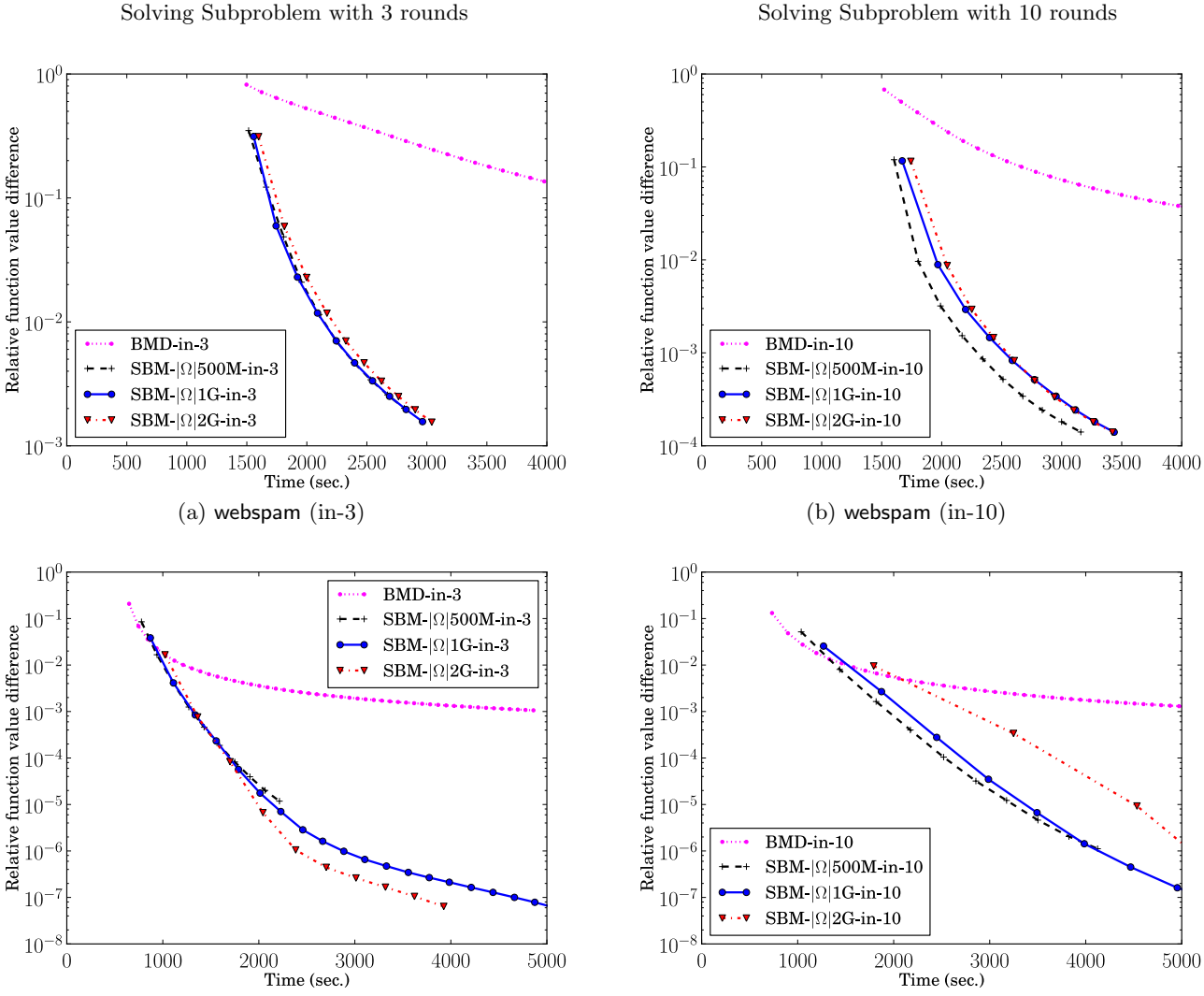


Figure 4: A comparison among SBM, BMD and online learning algorithms on webspam. All the methods are implemented with single precision. We show the difference between test performance and the best accuracy achievable (8). Time is in seconds. The markers indicate the time it takes each solver to access data from disk once.

B.2 Performance of SBM with Various Cache Size

In this section, we investigate how is SBM sensitive to the cache size. We experiment on the two binary classification problems, webspam and kddcup10. We set cache size of SBM to 0B (equivalent to BMD), 500MB, 1GB and 2GB. We demonstrate the results of SBM when the sub-problem is solved with 3 rounds and 10 rounds.

Figure 5 shows the results. For webspam data set, as the number of unbounded support vectors is small, most of unbounded support vectors can be stored in cache even though the cache size is small (e.g., 500MB). As a result, SBM does not take advantage of large cache capacity. For kddcup10 data set, the number of unbounded support vectors is too large to store in cache. In this situation, 5(c) shows that when sub-problem is solved loosely, large cache size yields faster convergence. However, if sub-problem is solved tightly (see 5(d)), SBM takes too much effort on each sub-problem and suffers from long training time. How to choose the cache size and inner stopping condition of SBM can be a further research issue.



(a) webspam (in-3)

(b) webspam (in-10)

(c) kddcup10 (in-3)

(d) kddcup10 (in-10)

Figure 5: The relative function value difference (7) to the minimum of (2). Time is in seconds, and the y-axis is in log-scaled. We show the performance of SBM with various cache size. BMD is equivalent to SBM with setting cache size to zero. We demonstrate the situations that SBM solves the sub-problem with 3 rounds and 10 rounds.