Large Linear Classification When Data Cannot Fit In Memory

Hsiang-Fu Yu, Cho-Jui Hsieh, Kai-Wei Chang and Chih-Jen Lin,
Department of Computer Science, National Taiwan University

Recent advances in linear classification have shown that for applications such as document classification, the training process can be extremely efficient. However, most of the existing training methods are designed by assuming that data can be stored in the computer memory. These methods cannot be easily applied to data larger than the memory capacity due to the random access to the disk. We propose and analyze a block minimization framework for data larger than the memory size. At each step a block of data is loaded from the disk and handled by certain learning methods. We investigate two implementations of the proposed framework for primal and dual SVMs, respectively. Because data cannot fit in memory, many design considerations are very different from those for traditional algorithms. We discuss and compare with existing approaches which are able to handle data larger than memory. Experiments using data sets 20 times larger than the memory demonstrate the effectiveness of the proposed method.

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

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Block minimization methods, large-scale learning, linear classification, support vector machines

ACM Reference Format:
Yu, H.-F., Hsieh, C.-J., Chang, K.-W., Lin, C.-J. 2011. Large Linear Classification When Data Cannot Fit In Memory. ACM Trans. Knowl. Discov. Data. 0, 0, Article 0 (0), 23 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Linear classification[1] is useful in many applications, but training large-scale data remains an important research issue. For example, a category of PASCAL Large Scale Learning Challenge[2] at ICML 2008 is designed to compare *linear* SVM implementations. The competition evaluates the running time after data have been loaded into the memory, but many participants find that loading time costs more. Thus, some have concerns about the evaluation.[3] This result indicates a landscape shift in large-scale linear classification because time spent on reading/writing between memory and disk becomes the bottleneck. A more challenging situation for large linear classification is to deal with data sets that cannot fit in memory. Existing training algorithms often

[1]By linear classification we mean that data remain in the input space and kernel methods are not used.
[2]`http://largescale.first.fraunhofer.de/workshop`
[3]`http://hunch.net/?p=330`

This work was supported in part by the National Science Council of Taiwan via the grant 98-2221-E-002-136-MY3.
Authors' addresses: H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin, Department of Computer Science, National Taiwan University, Taipei 106, Taiwan; Email:{b93107, b92085, b92084, cjlin}@cise.ntu.edu.tw.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

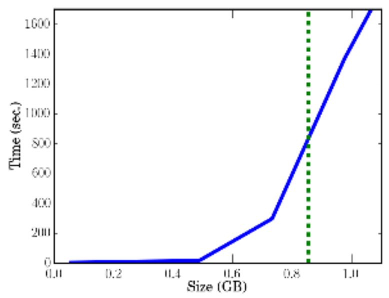


Fig. 1: Data size versus training time by directly applying LIBLINEAR on a machine with 1GB memory (the actual available memory is about 0.853GB). When data size is larger than the memory capacity, the running time grows rapidly.

need to iteratively access data, so without enough memory, the training time will be huge. To see how serious the situation is, Figure 1 presents the running time by applying an efficient linear classification package LIBLINEAR [Fan et al. 2008] to train data with different scales on a computer with 1 GB memory. Clearly, the time grows sharply when the data size is beyond the memory capacity.

We model the training time to contain two parts:

$$\begin{aligned} \text{training time} = {} & \text{time to run data in memory} + \\ & \text{time to access data from disk.}^4 \end{aligned} \tag{1}$$

Traditional training algorithms, assuming that the second part is negligible, focus on the first part by minimizing the number of CPU operations. Linear classification, especially when applied to document classification, is in a situation that the second part may be more significant. Recent advances on linear classification (e.g., Joachims [2006], Bottou [2007], Hsieh et al. [2008], Shalev-Shwartz et al. [2011] and the recent survey by Yuan et al. [2011]) have shown that training one million instances takes only a few seconds (without counting the loading time). Therefore, some have said that linear classification is essentially a *solved* problem if the memory is enough. However, handling data beyond the memory capacity remains a challenging research issue.

According to Langford et al. [2009], existing approaches to handle large data can be roughly categorized to two types. The first approach solves problems in distributed systems by parallelizing batch training algorithms (e.g., Chang et al. [2008] and Zhu et al. [2009]). However, not only is writing programs on a distributed system difficult, but also the data communication/synchronization may cause significant overheads. The second approach considers online learning algorithms. Because data may be used only once, this type of approaches can effectively handle the memory issue. However, even with an online setting, an implementation over a distributed environment is still complicated; see the discussion in Section 2.1 of Langford et al. [2009]. Moreover, existing implementations (including those in large Internet companies) may lack important functions such as evaluations by different criteria, parameter selection, or feature selection.

In machine learning practice, Tong [2010] argues that keeping algorithms simple and robust is crucial. Therefore, a simple system constructed according to users' need is

[4]These two parts are not necessarily disjoint because we may run some data in memory and simultaneously read other data from disk. For simplicity, here we assume they are disjoint.

more favorable. This paper aims to construct large linear classifiers for ordinary users who can access only a single machine rather than a distributed system. We consider one assumption and one requirement:

— Assumption: Data cannot be stored in memory, but can be stored in the disk of one computer. Moreover, sub-sampling data to fit in memory causes lower accuracy.
— Requirement: The method must be simple so that support for multi-class classification, parameter selection and other functions can be easily done.

Our assumption holds only for certain data, because sub-sampling is useful in some occasions. In particular, if informative instances are retained in the selected subset, the resulting accuracy may be similar to that of using the full set. A study by Yu et al. [2003] selects important instances by reading data from disk only once.

In this work, we discuss a simple and effective block minimization framework for applications satisfying the above assumption. We focus on batch learning though extensions to online or incremental/decremental learning are straightforward. While many existing online learning studies claim to handle data beyond the memory capacity, most of them conduct simulations with enough memory and check the number of passes to access data (e.g., Shalev-Shwartz et al. [2011] and Bottou [2007]). In contrast, we conduct experiments in a real environment without enough memory. We show that the proposed methods are competitive with a well-developed online learning package Vowpal_Wabbit [Langford et al. 2007].

An earlier linear-SVM study [Ferris and Munson 2003] has specifically addressed the situation that data are stored in disk, but it assumes that the number of features is much smaller than data points. Our approach allows a large number of features, a situation often occurred for document data sets.

This paper is organized as follows. In Section 2, we consider SVM as our linear classifier and propose a block minimization framework. Two implementations of the proposed framework for primal and dual SVM problems are respectively described in Sections 3 and 4. Techniques to minimize the training time modeled in Eq. (1) are in Section 5. Section 6 discusses the implementation of cross validation, multi-class classification, and incremental/decremental settings. Section 7 discusses related approaches for training linear classifiers when data are larger than memory capacity. We show experiments in Section 8 and give conclusions in Section 9.

A preliminary version of this work appears in a conference paper [Yu et al. 2010].

2. BLOCK MINIMIZATION FOR LINEAR SVMS

We consider linear SVM in this work because it is one of the most used linear classifiers. Given a training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^l$, $\boldsymbol{x}_i \in R^n,\ y_i \in \{-1,+1\}$, SVM solves the following unconstrained optimization problem.[5]

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\max(1-y_i\boldsymbol{w}^T\boldsymbol{x}_i, 0), \tag{2}$$

where $C > 0$ is a penalty parameter. This formulation considers L1 loss, though our approach can be easily extended to L2 loss. Problem (2) is often referred to as the primal form of SVM. One may instead solve its dual problem.

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le C, i = 1, \dots, l, \end{aligned} \tag{3}$$

[5]The standard SVM comes with a bias term b. Here we do not consider this term for simplicity.

ALGORITHM 1: A block minimization framework for linear SVM

```
1. Split $\{1,\ldots,l\}$ to $B_1,\ldots,B_m$ and store data into $m$ files accordingly.
2. Set initial $\boldsymbol{\alpha}$ or $\boldsymbol{w}$.
3. For $k = 1, 2, \ldots$    (outer iteration)
    — For $j = 1,\ldots,m$    (inner iteration)
        3.1. Read $\boldsymbol{x}_r, \forall r \in B_j$ from disk.
        3.2. Conduct operations on $\{\boldsymbol{x}_r \mid r \in B_j\}$.
        3.3. Update $\boldsymbol{\alpha}$ or $\boldsymbol{w}$.
```

where $\boldsymbol{e} = [1,\ldots,1]^T$ and $Q_{ij} = y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$.

Because data cannot fit in memory, the training method must avoid random accesses of data. In Figure 1, LIBLINEAR randomly accesses one instance at a time, so frequent moves of the disk head result in lengthy running time. A viable method must satisfy the following conditions:

1. Each optimization step reads a *contiguous* chunk of training data.
2. The optimization procedure converges toward the optimum even though each step uses only a subset of training data.
3. The number of optimization steps (iterations) should not be too large. Otherwise, the same data point may be accessed from the disk too many times.

Obtaining a method having all these properties is not easy. We will propose block minimization methods to achieve them to a certain degree.

In unconstrained optimization, block minimization is a classical method (e.g., Bertsekas [1999, Chapter 2.7]). Each step of this method updates a block of *variables*, but, to apply it here, we hope each block corresponds to a contiguous chunk of data. Let $\{B_1,\ldots,B_m\}$ be a partition of all data indices $\{1,\ldots,l\}$. According to the memory capacity, we can decide the block size so that instances associated with B_j can fit in memory. These m blocks, stored as m files, are loaded when needed. Then at each step, we conduct some operations using one block of data, and update $\boldsymbol{w}$ or $\boldsymbol{\alpha}$ according to if the primal or the dual problem is considered. We assume that $\boldsymbol{w}$ or $\boldsymbol{\alpha}$ can be stored in memory. The block minimization framework is summarized in Algorithm 1. We refer to the step of working on a single block as an inner iteration, while the m steps of going over all blocks as an outer iteration. Algorithm 1 can be applied on both the primal form (2) and the dual form (3), where two implementations are shown in Sections 3 and 4, respectively.

We discuss some implementation considerations for Algorithm 1. For convenience, assume $B_1,\ldots,B_m$ have a similar size $|B| = l/m$. The total cost of Algorithm 1 is

$$(T_m(|B|) + T_d(|B|)) \times \frac{l}{|B|} \times \text{\#outer-iters}, \tag{4}$$

where

— $T_m(|B|)$ is the cost of operations at each inner iteration, and
— $T_d(|B|)$ is the cost to read a block of data from disk. In general,

$$T_d(B) = \text{initial cost} + O(|B|), \tag{5}$$

where $O(|B|)$ indicates the transfer time proportional to the data size.

The two terms $T_m(|B|)$ and $T_d(|B|)$ respectively correspond to the two parts in Eq. (1) for modeling the training time.

Many studies have applied block minimization to train SVM or other machine learning problems, but we are not aware of any work that considers this in the disk level.

Currently, the major approach to train nonlinear SVM (i.e., SVM with nonlinear kernels) has been block minimization, which is often called decomposition methods in the SVM community. We discuss the difference between ours and existing studies in two aspects:

— variable selection for each block, and
— block size.

Existing SVM packages assume data in memory, so they can use flexible ways to select each B_j. They do not restrict $B_1, \ldots, B_m$ to be a split of $\{1, \ldots, l\}$. Moreover, to decide indices of one single B_j, they may access the whole set, an impossible situation for us. We are more constrained here because data associated with each B_j must be predetermined and stored in a contiguous chunk of the disk before running Algorithm 1.

Regarding the block size, we now go back to analyze Eq. (4). If data can fit in memory, $T_d(|B|) = 0$. Generally, we have

$$|B| \nearrow \text{ implies } T_m(|B|) \nearrow \text{ and \#outer-iters} \searrow .^6 \tag{6}$$

$T_m(|B|)$ is more than linear to $|B|$; see, for example, the theoretical complexity analysis by Boyd and Vandenberghe [2004, Chapter 11].[7] Therefore, $T_m(|B|) \times l/|B|$ in Eq. (4) is increasing along with $|B|$. In contrast, the #outer-iters may not decrease as quick. Therefore, nearly all existing SVM packages use a small $|B|$. For example, $|B| = 2$ in LIBSVM [Chang and Lin 2011] and 10 in SVM^{light} [Joachims 1998]. With $T_d(|B|) > 0$, the situation is now very different. At each outer iteration, the cost is

$$T_m(|B|) \times \frac{l}{|B|} + T_d(|B|) \times \frac{l}{|B|}. \tag{7}$$

The second term is for reading l instances. Because Eq. (5) indicates that reading each block of data takes some initial time, a smaller number of blocks is better. That is, the second term in Eq. (7) is a decreasing function of $|B|$. While the first term is increasing following the earlier discussion, as reading data from the disk is slow, the second term is likely to dominate. Therefore, contrary to existing SVM software, in our case the block size should not be too small. We will investigate this issue by experiments in Section 8.

The remaining issue is to decide operations at each inner iteration. The second and the third conditions mentioned earlier in this section should be considered. We discuss two implementations in the next two sections.

3. SOLVING DUAL SVM BY LIBLINEAR FOR EACH BLOCK

A nice property of the SVM dual problem (3) is that each variable corresponds to a training instance. Thus, we can easily devise an implementation of Algorithm 1 by updating a block of variables at a time. Let $\bar{B}_j = \{1, \ldots, l\} \backslash B_j$ and $\boldsymbol{d}_{\bar{B}_j}$ be the sub-vector

[6] We are not aware of any theoretical result on the decrease of the number of outer iterations, but empirically this is generally true. See, for example, Table 1 of Serafini and Zanni [2005].
[7] Most existing analyses consider inter-point methods, in which a sequence of linear systems must be solved. Because solving a linear system is cubic to the number of variables, the overall complexity is at least as large.

ALGORITHM 2: An implementation of Algorithm 1 for solving dual SVM

```
We only show details of steps 3.2 and 3.3:
3.2 Exactly or approximately solve the sub-problem (8) to obtain $d^*_{B_j}$.
3.3 $\alpha_{B_j} \leftarrow \alpha_{B_j} + d^*_{B_j}$
    Update $w$ by Eq. (11).
```

of $\boldsymbol{d}$ comprising d_i, $i \in \bar{B}_j$.[8] At each inner iteration we solve the following sub-problem.

$$\begin{aligned} \min_{\boldsymbol{d}_{B_j}} \quad & f(\boldsymbol{\alpha}+\boldsymbol{d}) \\ \text{subject to} \quad & \boldsymbol{d}_{\bar{B}_j} = \boldsymbol{0} \text{ and } 0 \le \alpha_i + d_i \le C,\ \forall i \in B_j. \end{aligned} \tag{8}$$

That is, we update $\boldsymbol{\alpha}_{B_j}$ using the solution of sub-problem (8), while fix $\boldsymbol{\alpha}_{\bar{B}_j}$. Then, Algorithm 1 reduces to the standard block minimization procedure, so the convergence to the optimal function value of problem (3) holds [Bertsekas 1999, Proposition 2.7.1].

We must ensure that at each inner iteration, only one block of data is needed. With the constraint $\boldsymbol{d}_{\bar{B}_j} = \boldsymbol{0}$ in Eq. (8),

$$f(\boldsymbol{\alpha}+\boldsymbol{d}) = \frac{1}{2}\boldsymbol{d}_{B_j}^T Q_{B_j B_j} \boldsymbol{d}_{B_j} + (Q_{B_j,:}\boldsymbol{\alpha} - \boldsymbol{e}_{B_j})^T \boldsymbol{d}_{B_j} + f(\boldsymbol{\alpha}), \tag{9}$$

where $Q_{B_j,:}$ is a sub-matrix of Q including elements Q_{ri}, $r \in B_j, i = 1, \ldots, l$. Clearly, $Q_{B_j,:}$ in Eq. (9) involves all training data, a situation violating the requirement of Algorithm 1. Fortunately, some (e.g., Zhang [2002] and Hsieh et al. [2008]) have proposed a trick to conquer this difficulty. By initializing and maintaining

$$\boldsymbol{w} \equiv \sum_{i=1}^{l} \alpha_i y_i \boldsymbol{x}_i, \tag{10}$$

we have

$$Q_{r,:}\boldsymbol{\alpha} - 1 = y_r \boldsymbol{w}^T \boldsymbol{x}_r - 1, \forall r \in B_j.$$

Therefore, if $\boldsymbol{w}$ is available in memory, only instances associated with the block B_j are needed. To maintain $\boldsymbol{w}$, if $\boldsymbol{d}^*_{B_j}$ is an optimal solution of sub-problem (8), we consider Eq. (10) and use

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \sum_{r \in B_j} d^*_r y_r \boldsymbol{x}_r. \tag{11}$$

This operation again needs only the block B_j. The procedure is summarized in Algorithm 2.

For solving the sub-problem (8), because all the information is available in the memory, any bound-constrained optimization method can be applied. We consider a dual coordinate descent method (i.e., block minimization with a single element in each block) by Hsieh et al. [2008]. It is implemented as one of the many solvers in the software LIBLINEAR [Fan et al. 2008]. Then, Algorithm 2 becomes a two-level block minimization method. The two-level setting had been used previously for SVM or other applications (e.g., Memisevic [2006], Pérez-Cruz et al. [2004] and Rüping [2000]), but we are not aware of any work that associates the inner level with memory and the outer level with disk.

[8]Following the use of $\boldsymbol{d}_{\bar{B}_j}$ to represent a sub-vector of $\boldsymbol{d}$, we denote other sub-vectors in this paper by the same way.

Algorithm 2 converges if each sub-problem (8) is exactly solved. Practically we often obtain an approximate solution by imposing a stopping criterion. We therefore must address two issues:

1. The stopping criterion for solving the sub-problem must be satisfied after a finite number of operations, so we can move on to the next sub-problem.
2. We need to prove the convergence.

Next, we show that these two issues can be resolved if we use LIBLINEAR to solve the sub-problem. Let $\{\boldsymbol{\alpha}^k\}$ be the sequence generated by Algorithm 2, where k is the index of outer iterations. Because each outer iteration contains m inner iterations, we can further consider a sequence

$$\{\boldsymbol{\alpha}^{k,j}\}_{k=1,j=1}^{\infty,m+1} \text{ with } \boldsymbol{\alpha}^{k,1} = \boldsymbol{\alpha}^k \text{ and } \boldsymbol{\alpha}^{k,m+1} = \boldsymbol{\alpha}^{k+1}.$$

From $\boldsymbol{\alpha}^{k,j}$ to $\boldsymbol{\alpha}^{k,j+1}$, LIBLINEAR coordinate-wisely updates variables in B_j to approximately solve the sub-problem (8).

If the coordinate descent updates satisfy certain conditions, we can prove the convergence of $\{\boldsymbol{\alpha}^{k,j}\}$:

THEOREM 3.1. *If a coordinate descent method is applied to solve sub-problem* (8) *and it possesses the following properties:*

1. *each $\alpha_i,\ i \in B_j$ is updated at least once, and*
2. *the number of coordinate-descent updates $t_{k,j}$ for solving a sub-problem is uniformly bounded (i.e., $\exists T > 0$ such that $t_{k,j} < T\ \forall k, j$),*

then $\{\boldsymbol{\alpha}^{k,j}\}$ generated by Algorithm 2 globally converges to an optimal solution $\boldsymbol{\alpha}^*$. The convergence rate is at least linear: there are $0 < \mu < 1$ and an iteration k_0 such that

$$f(\boldsymbol{\alpha}^{k+1}) - f(\boldsymbol{\alpha}^*) \leq \mu \left(f(\boldsymbol{\alpha}^k) - f(\boldsymbol{\alpha}^*) \right), \forall k \geq k_0. \tag{12}$$

The proof is in Appendix A. With Theorem 3.1, the condition 2 mentioned in the beginning of Section 2 holds. For condition 3 on the convergence speed, the theoretical linear convergence shown in (12) is not very fast. However, for problems like document classification, some (e.g., Hsieh et al. [2008]) have shown that in practice a small number of iterations is enough to get a reasonable model. Though Hsieh et al. [2008] differs from us by restricting $|B| = 1$, we hope to enjoy the same property of not needing many iterations. Experiments in Section 8 confirm that for some document data this property holds.

Next, we discuss various ways to fulfill the two properties in Theorem 3.1.

3.1. Loosely Solving the Sub-problem

A simple setting to satisfy Theorem 3.1's two properties is to go through all variables in B_j a fixed number of times. Then, not only is $\{t_{kj}\}$ uniformly bounded, but also the finite termination for solving each sub-problem holds. A small number of passes to go through B_j means that we loosely solve the sub-problem (8). The cost per block is thus cheap, although the number of outer iterations may become large. Through experiments in Section 8, we discuss how the number of passes affects the running time. A special case is to go through all $\alpha_i, i \in B_j$ only once. Then, Algorithm 2 becomes a standard (one-level) coordinate descent method, though data are loaded by a block-wise setting.

For each pass to go through data in one block, we can sequentially update variables in B_j. However, as mentioned in Hsieh et al. [2008], using a random permutation of B_j's elements as the order of updates usually leads to faster convergence in practice.

3.2. Accurately Solving the Sub-problem

Alternatively, we can accurately solve the sub-problem. The cost per inner iteration is higher, but the number of outer iterations may be reduced. Because an upper bound on the number of iterations does not reveal how accurate the solution is, most optimization software considers the gradient information for the stopping condition. We check the setting in LIBLINEAR. Its gradient-based stopping condition (details shown in Appendix B) guarantees the finite termination in solving each sub-problem (8). Thus, the procedure can move on to the next sub-problem without getting into an infinite loop. Regarding the convergence, to use Theorem 3.1, we must show that $\{t_{k,j}\}$ is uniformly bounded:

THEOREM 3.2. *If coordinate descent steps with* LIBLINEAR*'s stopping condition are used to solve sub-problem* (8)*, then Algorithm 2 either terminates in a finite number of outer iterations or*

$$t_{k,j} \leq 2|B_j| \; \forall j \text{ after } k \text{ is large enough.}$$

Therefore, if LIBLINEAR's dual coordinate descent implementation is used to solve sub-problem (8), then Theorem 3.1 implies the convergence.

4. SOLVING PRIMAL SVM BY PEGASOS FOR EACH BLOCK

Instead of solving the dual problem, in this section we check if the framework in Algorithm 1 can be used to solve the primal SVM. Because the primal variable $\boldsymbol{w}$ does not correspond to data instances, we cannot use a standard block minimization setting to have a sub-problem like (8). In contrast, existing stochastic gradient descent methods possess a nice property that at each step only certain data points are used. In this section, we study how a stochastic method Pegasos [Shalev-Shwartz et al. 2011] can by used for implementing Algorithm 1.

Pegasos considers a scaled form of the primal SVM problem.

$$\min_{\boldsymbol{w}} \quad \frac{1}{2lC}\boldsymbol{w}^T\boldsymbol{w} + \frac{1}{l}\sum_{i=1}^{l}\max(1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i, 0).$$

At the tth update, Pegasos chooses a block of data B and updates the primal variable $\boldsymbol{w}$ by a stochastic gradient descent step:

$$\bar{\boldsymbol{w}} = \boldsymbol{w} - \eta^t\nabla^t, \tag{13}$$

where $\eta^t = lC/t$ is the learning rate, ∇^t is the sub-gradient

$$\nabla^t = \frac{1}{lC}\boldsymbol{w} - \frac{1}{|B|}\sum_{i\in B^+} y_i\boldsymbol{x}_i, \tag{14}$$

and $B^+ \equiv \{i \in B \mid y_i\boldsymbol{w}^T\boldsymbol{x}_i < 1\}$. Then, Pegasos obtains $\boldsymbol{w}$ by scaling $\bar{\boldsymbol{w}}$:

$$\boldsymbol{w} \leftarrow \min(1, \frac{\sqrt{lC}}{\|\bar{\boldsymbol{w}}\|})\bar{\boldsymbol{w}}. \tag{15}$$

Clearly, we can directly consider B_j in Algorithm 1 as the set B in the above update. Alternatively, we can conduct several Pegasos updates on a partition of B_j. Algorithm 3 gives details of the procedure. Here, we consider two settings for an inner iteration:

1. Using one Pegasos update on the whole block B_j.
2. Splitting B_j to $|B_j|$ sets, where each one contains an element in B_j and then conducting $|B_j|$ Pegasos updates.

ALGORITHM 3: An implementation of Algorithm 1 for solving primal SVM. Each inner iteration is performed by Pegasos.

```
1. Split $\{1,\dots,l\}$ to $B_1,\dots,B_m$ and store data into $m$ files accordingly.
2. $t = 0$ and set initial $\boldsymbol{w} = \mathbf{0}$.
3. For $k = 1, 2, \dots$
   — For $j = 1,\dots,m$
      3.1. Find a partition of $B_j$: $B_j^1,\dots,B_j^{\bar{r}}$.
      3.2. For $r = 1,\dots,\bar{r}$
         — Use $B_j^r$ as $B$ to conduct the updates (13)-(15).
         — $t \leftarrow t + 1$
```

Different from dual SVM, we should not solve the sub-problem of primal SVM accurately. Otherwise, the model will converge to a solution that only learns on the set B_j.

For the convergence, Pegasos is proved to converge if all instances $\{\boldsymbol{x}_1,\dots,\boldsymbol{x}_l\}$ are used for updating the model at each step; see Shalev-Shwartz et al. [2011, Corollary 1]. It is also shown that the same convergence results hold in expectation if each update is conducted on a subset chosen i.i.d. from the entire data set; see Shalev-Shwartz et al. [2011, Lemma 3]. Although Algorithm 3 is a special case of Pegasos, it splits the data into blocks $B_j, \forall j$ and updates on a subset of B_j at a time. Therefore, we are not able to apply their convergence proof. However, empirically we observe that Algorithm 3 converges without problems.

5. TECHNIQUES TO REDUCE THE TRAINING TIME

Many techniques have been proposed to make block minimization faster. However, these techniques may not be suitable here as they are designed by assuming that all data are in memory. Based on the complexity analysis in Eq. (7), in this section we propose three techniques to speed up Algorithm 1. One technique effectively shortens $T_d(|B|)$, while the other two aim at reducing the number of iterations.

5.1. Data Compression

The loading time $T_d(|B|)$ is a bottleneck of Algorithm 1 due to the slow disk access. Except some initial cost, Eq. (5) indicates that $T_d(|B|)$ is proportional to the length of data. Hence, we can consider a compression strategy to reduce the loading time of each block. However, this strategy introduces two additional costs: the compression time in the beginning of Algorithm 1 and the decompression time when a block is loaded. The former is minor as we only do it once. For the latter, we must ensure that the loading time saved is more than the decompression time. The balance between compression speed and ratio has been well studied in the area of backup and networking tools [Morse 2005]. We choose a widely used compression library zlib for our implementation.[9] Experiments in Section 8 show that if the reading speed of the disk is slow, the compression strategy effectively reduces the training time.

Because of using compression techniques, all blocks are stored in a binary format instead of a plain text form.

5.2. Random Permutation of Sub-problems

In Algorithm 1, we sequentially work on blocks $B_1, B_2, \dots, B_m$. We can consider other ways such as using a permutation of blocks. In LIBLINEAR's coordinate descent implementation, the authors randomly permute all variables at each pass of going through

[9]`http://www.zlib.net`

ALGORITHM 4: Splitting data into blocks

```
1. Decide $m$ and create $m$ empty files.
2. For $i = 1, \ldots$
    2.1. Convert $\boldsymbol{x}_i$ to a binary format $\bar{\boldsymbol{x}}_i$.
    2.2. Randomly choose a number $j \in \{1, \ldots, m\}$.
    2.3. Append $\bar{\boldsymbol{x}}_i$ into the end of the $j$th file.
```

data and report faster convergence. We adopt a permutation strategy here as the loading time is similar regardless of the order of sub-problems.

5.3. Split of Data

An important step of Algorithm 1 is to split training data into m files. We need a careful design as data cannot fit in memory. To begin, we find the size of data and decide the value m based on the memory capacity. This step does not have to go through the whole data set because the operating system provides information such as file sizes. Then, we can sequentially read data instances and save them to m files. This approach is simple and seems to work well in the first glance. However, data in the same class are often stored together in the training set, so we may get a block of data with the same label. This situation clearly causes slow convergence.[10] Thus, for each instance being read, we randomly decide which file it should be saved to. Algorithm 4 summarizes our procedure. It goes through data only once.

6. OTHER FUNCTIONALITY

A learning system only able to solve an optimization problem (2) or (3) is not practically useful. Other functions such as multi-class classification or cross validation (for parameter selection) are very important. We discuss how to implement these functions based on the design in Section 2.

6.1. Multi-class Classification

Existing multi-class approaches either solve one single optimization problem (e.g., Crammer and Singer [2002]) or train several two-class problems (e.g., one-against-one and one-against-the rest). For data beyond the memory capacity, we discuss how to solve the optimization problem by Crammer and Singer [2002] and how to apply the one-against-the rest strategy.

If data can fit in memory, the optimization problem by Crammer and Singer [2002] can be solved by a dual coordinate descent method [Keerthi et al. 2008], which, available in LIBLINEAR, is an extension of the coordinate descent method by Hsieh et al. [2008] for the standard SVM dual problem. For data larger than memory, because the dual form of Crammer and Singer's formulation still possesses the property that variables correspond to data instances, the block minimization framework in Algorithm 1 can still be applied. Then, for each block, the sub-problem can be solved by the method of Keerthi et al. [2008].

To apply the one-against-the rest approach for a K-class problem, we must train K classifiers, where each one separates a class from the rest. If we sequentially train K models, the disk accessing time is K times more. An implementation to save the disk access time is to train K models together. We split each block B_j to $B_j^1, \ldots, B_j^K$ according to the class information. Then, we solve K sub-problems simultaneously. That is, we use B_j^t as positive data and $B_j \setminus B_j^t$ as negative data to update vectors $\boldsymbol{w}^t$

[10]Note that this is also an issue with online/stochastic gradient methods if they split data and randomly select a file at a time to work on.

ALGORITHM 5: An block minimization framework for the one-against-the rest multi-class approach. We assume the K class labels are $1, \ldots, K$.

```
1. Split $\{1, \ldots, l\}$ to $B_1, \ldots, B_m$, and store data into $m$ files accordingly.
2. Set initial $\boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^K$ and $\boldsymbol{w}^1, \ldots, \boldsymbol{w}^K$, where $K$ is the number of classes.
3. For $k = 1, 2, \ldots$ (outer iteration)
   — For $j = 1, \ldots, m$ (inner iteration)
      3.1. Read $\boldsymbol{x}_r, \forall r \in B_j$ from disk.
      3.2. For $t = 1, \ldots, K$
           — Use $B_j^t \equiv \{\boldsymbol{x}_r \mid r \in B_j \text{ and } y_r = t\}$ as positive data and $B_j \setminus B_j^t$ as negative
             data.
           — Conduct certain training operations, and update $\boldsymbol{\alpha}^t$ and $\boldsymbol{w}^t$.
```

and $\boldsymbol{\alpha}^t$. The details are in Algorithm 5. The one-against-one approach is less suitable as it needs $K(K-1)/2$ vectors to store $\boldsymbol{w}$, which may be memory consuming. For one-against-the rest and the approach in Crammer and Singer [2002], they both need only K vectors.

6.2. Cross Validation

Assume we conduct v-fold cross validation. Due to the use of m blocks, a straightforward implementation is to split m blocks to v groups. Each time one group of blocks is used for validation, while all the remaining groups are for training. Similar to the situation in multi-class classification, the loading time is v times more than training a single model. To save the disk accessing time, a more sophisticated implementation is to train v models together. For example, if $v = 3$, we split each block B_j to three parts B_j^1, B_j^2, and B_j^3. Then $\cup_{j=1}^m (B_j^1 \cup B_j^2)$ is the training set to validate $\cup_{j=1}^m B_j^3$. We maintain three vectors $\boldsymbol{w}^1, \boldsymbol{w}^2$, and $\boldsymbol{w}^3$. Each time when B_j is loaded, we solve three sub-problems to update $\boldsymbol{w}$ vectors. This implementation effectively saves the data loading time, but the memory must be enough to store v vectors $\boldsymbol{w}^1, \ldots, \boldsymbol{w}^v$. The overall procedure is similar to Algorithm 5 for multi-class classification.

6.3. Incremental/ Decremental Setting

Many practical applications retrain a model after collecting enough new data. Our approach can be extended to this scenario. We make a reasonable assumption that each time several blocks are added or removed. Using LIBLINEAR to solve the dual form as an example, to possibly save the number of iterations, we can reuse the vector $\boldsymbol{w}$ obtained earlier. Algorithm 2 maintains $\boldsymbol{w} = \sum_{i=1}^l y_i \alpha_i \boldsymbol{x}_i$, so the new initial $\boldsymbol{w}$ can be

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \sum_{i:\boldsymbol{x}_i \text{ being added}} y_i \alpha_i \boldsymbol{x}_i - \sum_{i:\boldsymbol{x}_i \text{ being removed}} y_i \alpha_i \boldsymbol{x}_i. \tag{16}$$

For data being added, α_i is simply set to zero, but for data being removed, their corresponding α_i are not available. To use Eq. (16), we must store $\boldsymbol{\alpha}$. That is, before and after solving each sub-problem, Algorithm 2 reads and saves $\boldsymbol{\alpha}$ from/to disk.

If we solve the primal problem by Pegasos for each block, Algorithm 3 can be directly applied for incremental or decremental settings.

7. RELATED APPROACHES FOR LARGE-SCALE DATA

In this section, we discuss related approaches for training linear classifiers when data cannot fit in the memory. The comparisons between these approaches and our block minimization framework are in Section 8.3.

Table I: Data statistics: We assume a sparse storage. Each non-zero feature value needs 12 bytes (4 bytes for the feature index and 8 bytes for the value). However, this 12-byte structure consumes 16 bytes on a 64-bit machine due to data structure alignment.

Data set	l	n	#nonzeros	Memory (Bytes)
yahoo-korea	460,554	3,052,939	156,436,656	2,502,986,496
kddcup10	19,264,093	29,890,095	566,345,790	9,061,532,640
webspam	350,000	16,609,143	1,304,697,446	20,875,159,136
epsilon	500,000	2,000	1,000,000,000	16,000,000,000

7.1. Data Sub-sampling

In many cases, sub-sampling training data does not downgrade the prediction accuracy much. Therefore, by using only a portion of the training data to fit in the memory, we can employ standard training techniques. This approach usually works well when the data quality is good. However, in some situations, using the full training set may still be necessary. In Section 8.3, we demonstrate the relationship between testing performance and sub-sampling size.

7.2. Aggregating Models Trained on Subsets of Data

Bagging [Breiman 1996] is a classical classification method. In the training phase, a bagging method randomly draws m subsets of samples from the entire data set. Then, it trains m models $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m$ on these subsets. In the testing phase, the prediction of a testing instance is based on the decisions from the m models. If each subset can be stored in memory, then training is efficient. Similar to the block generation in our framework, this method needs to get subsets in the beginning.

Although a bagging method is scalable to large data sets and may achieve an accurate model (e.g., Zinkevich et al. [2010] and Chakrabarti et al. [2008]), its solution is not the same as the model from solving problem (2). In Section 8.3, we compare the proposed block optimization framework with a bagging method, which averages m models trained on $B_j, \forall j$.

7.3. Online Learning Approaches

Online methods can easily deal with large-scale data. An online learning algorithm loads several data points at a time, so it avoids storing the whole data in the memory. In the following, we discuss an online learning package Vowpal_Wabbit [Langford et al. 2007].

Vowpal_Wabbit minimizes an un-regularized problem and supports several loss functions. Here we consider L1 loss. For any instance $\boldsymbol{x}$, it updates the weight vector $\boldsymbol{w}$ by a sub-gradient descent direction. Vowpal_Wabbit supports the setting to pass over data several times. During the first pass, it saves the data points into a cache file. This is similar to our data compression strategy discussed in Section 5.1. In Section 8.3, we compare Vowpal_Wabbit with the proposed block optimization framework.

8. EXPERIMENTS

In this section, we first conduct experiments to analyze the performance of the proposed block minimization framework. Then, we investigate several implementation issues discussed in Section 5. Finally, we compare the proposed method with other approaches that can handle data beyond the memory capacity.

Table II: Number of blocks and initial time to split and compress data into blocks. Time is in seconds.

Data set	#Blocks	Initial time
yahoo-korea	5	228
kddcup10	40	842
webspam	40	1,594
epsilon	30	1,237

We consider two document data sets yahoo-korea and webspam, an artificial data set epsilon, and an education data set kddcup10 from a data mining challenge.[11] Table I summarizes the data statistics. All data sets except yahoo-korea are publicly available at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

Except kddcup10, we randomly split each data set to $4/5$ for training and $1/5$ for testing, and all feature vectors are instance-wisely scaled to unit-length (i.e., $\|\boldsymbol{x}_i\| = 1, \forall i$). For epsilon, each feature of the training set is normalized to have mean zero and variance one, and the testing set is modified according to the same scaling factors. This feature-wise scaling is conducted before the instance-wise scaling. For the kddcup10 data set, we directly use the same training and testing split in Yu et al. [2011] without any further scaling.

We conduct experiments on a 64-bit machine with 1GB RAM. Due to the space consumed by the operating system, the available memory that we can use is 0.853GB. The reading speed of the disk is 102.36 MB/sec.[12] Our methods are implemented in C/C++ with double precision.

8.1. Comparison of Sub-problem Solvers

In this section, we compare various settings introduced in Sections 3–4 for operations on a block of data. The value C in problem (2) is set to one.

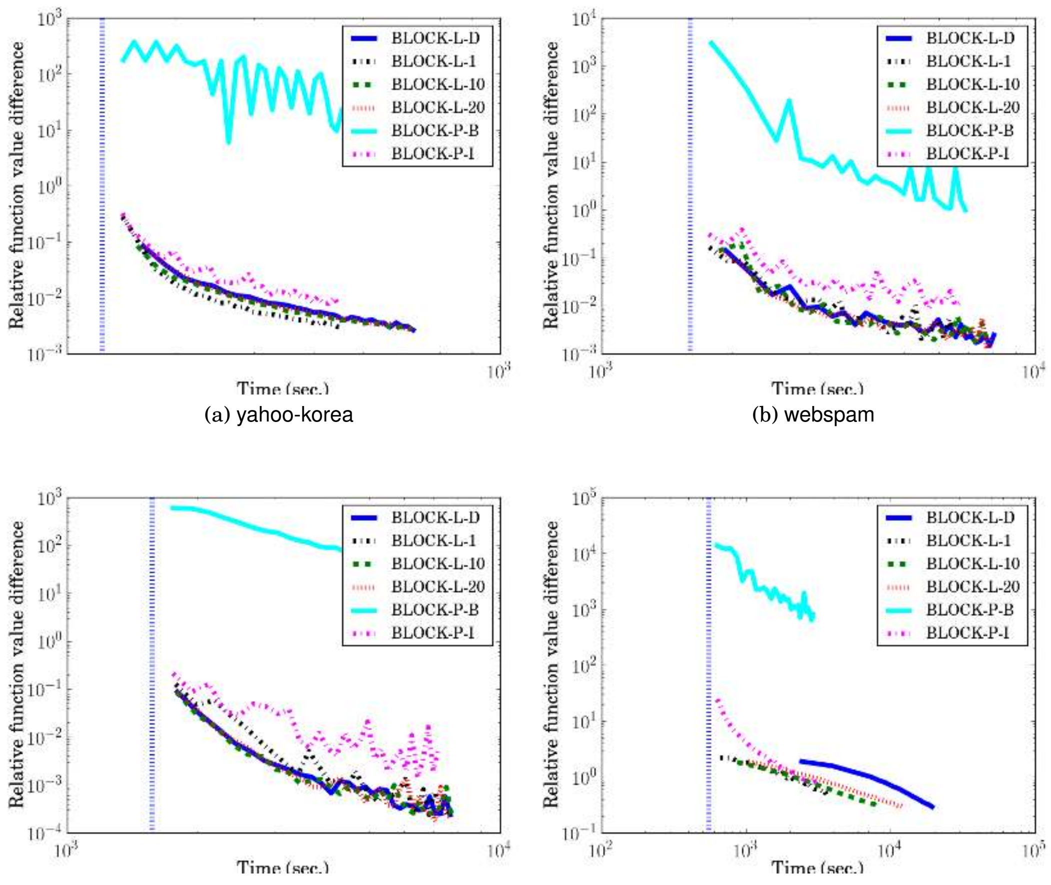
— BLOCK-L-N: Algorithm 2 with LIBLINEAR to solve each sub-problem. LIBLINEAR goes through each block of data N rounds, where we consider $N = 1, 10$, and 20.
— BLOCK-L-D: Algorithm 2 with LIBLINEAR to solve each sub-problem. LIBLINEAR's default stopping condition is adopted.
— BLOCK-P-B: Algorithm 3 with $\bar{r} = 1$. That is, we apply one Pegasos update on each block.
— BLOCK-P-I: Algorithm 3 with $\bar{r} = |B_j|$. That is, we apply $|B_j|$ Pegasos updates, each of which uses an individual data instance.

We do not include any standard linear classifier for comparison because Yu et al. [2010] have shown that these classifiers suffer from severe disk swapping.

We make sure that no other jobs are running on the same machine and report *wall clock* time in all experiments. We include all data loading time and the initial time to split and compress data into blocks. Table II lists the number of blocks and the initial time.

We are interested in both how fast these methods reduce the objective function value in Eq. (2) and how quickly they obtain a reasonable model. Figures 2 and 3 respectively present two results:

[11] We use a preprocessed version of the second data set bridge_to_algebra_2008_2009 in KDD Cup 2010.
[12] The reading speed of the disk is given by the program hdparm under the Linux environment.



(a) yahoo-korea (b) webspam (c) epsilon (d) kddcup10

Fig. 2: This figure shows the relative function value difference to the minimum. Time (in seconds) is log-scaled. The blue dotted vertical line indicates time spent by Algorithm 1-based methods for the initial split of data to blocks.

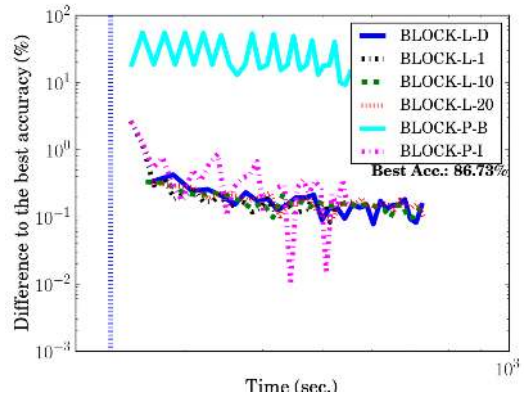
1. Training time versus the relative difference to the optimal function value

$$\left|\frac{f^P(\boldsymbol{w}) - f^P(\boldsymbol{w}^*)}{f^P(\boldsymbol{w}^*)}\right|,$$

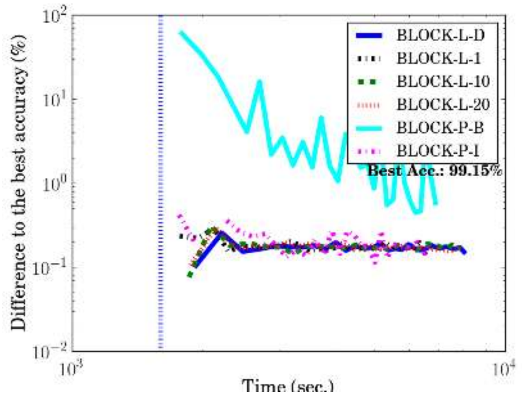
   where f^P is the primal objective function in Eq. (2) and $\boldsymbol{w}^*$ is the optimal solution. Since $\boldsymbol{w}^*$ is not really available, we spend enough training time to get a reference solution.
2. Training time versus the difference to the best testing accuracy

$$(\text{acc}^* - \text{acc}(\boldsymbol{w})) \times 100\%,$$

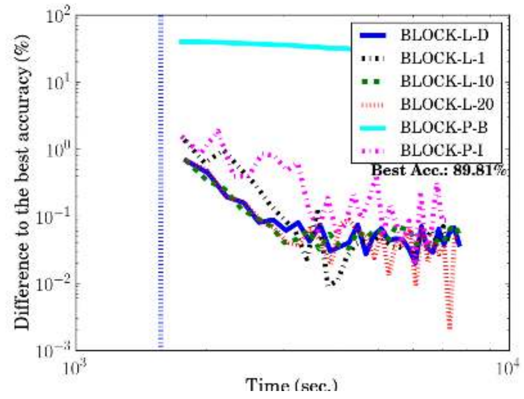
   where $\text{acc}(\boldsymbol{w})$ is the testing accuracy using the model $\boldsymbol{w}$ and acc^* is the final testing accuracy.



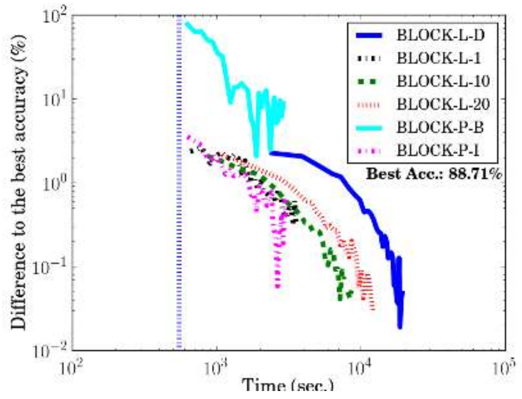
(a) yahoo-korea



(b) webspam



(c) epsilon



(d) kddcup10

Fig. 3: This figure shows the accuracy difference to the best testing accuracy. Time (in seconds) is log-scaled. The blue dotted vertical line indicates time spent by Algorithm 1-based methods for the initial split of data to blocks.

From Figure 2, BLOCK-L-$*$ methods (using LIBLINEAR) are faster than BLOCK-P-$*$ methods (using Pegasos) in most cases. One of the possible reasons is that for BLOCK-P-$*$, the information of each block is underutilized. In particular, BLOCK-P-B suffers from very slow convergence because for each block this method conducts only one very simple update. However, it may not be always needed to use the block of data in an exhaustive way. For example, in Figures 2a and 2d, BLOCK-L-1 (for each block LIBLINEAR goes through all data only once) is slightly faster than BLOCK-L-D (for each block LIBLINEAR is run with the default stopping condition). Nevertheless, as reading each block from the disk is expensive, in general we should make proper efforts to use it.

The numbers of instances and features in kddcup10 are very large. In such a situation, all the methods converge slowly; see Figure 2d. To store both w and α, BLOCK-L-$*$ methods requires 400MB memory. However, BLOCK-P-$*$ only need 160MB to store the model w. Because of using less memory, BLOCK-P-I is more likely to store w in

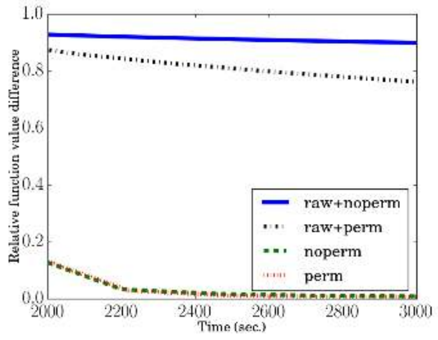


Fig. 4: Effectiveness of two implementation techniques: *raw*: no random assignment in the initial data splitting. *perm*: a random order of blocks at each outer iteration. BLOCK-L-D is used to train the data set webspam.

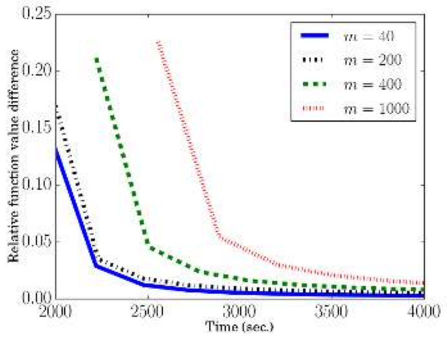


Fig. 5: Convergence speed of using different m (number of blocks). BLOCK-L-D is used to train the data set webspam.

a higher level of the memory hierarchy such as L2 cache. Therefore, it is competitive with BLOCK-L-$*$ in this case. This result also indicates that when determining the number of blocks in Algorithms 1, both l (size of $\boldsymbol{\alpha}$) and n (size of $\boldsymbol{w}$) need to be taken into consideration.

Regarding testing accuracy, if we consider a 0.5% difference to the best testing accuracy is satisfactory, all BLOCK-$*$ methods except BLOCK-P-B take about only four outer iterations to achieve reasonable accuracy values. Therefore, we do not need to read the training set many times.

8.2. Investigation of Some Implementation Issues

We investigate the usefulness of implementation techniques proposed in Section 5.

8.2.1. Initial Data Splitting and Random Permutation of Sub-problems. Section 5.3 proposes randomly assigning data to blocks in the beginning of Algorithm 1. It also suggests that a random order of $B_1, \ldots, B_m$ at each outer iteration is useful. Figure 4 presents the result of running BLOCK-L-D on webspam. We assume the worst situation that data of the same class are grouped together in the input file. If data are not randomly split to blocks, clearly the convergence is very slow. Further, the random permutation of blocks at each outer iteration slightly improves the training time.

8.2.2. Block Size. In Figure 5, we present the training speed of BLOCK-L-D by using various block sizes (equivalently, numbers of blocks). The training time of using $m = 40$ blocks is smaller than that of $m = 400$ or $1,000$. This result is consistent with the discussion in Section 2. When the number of blocks is smaller (i.e., larger block size), from Eq. (6), the cost of operations on each block increases. However, as we read fewer files, the total time is shorter. Furthermore, the initial time for data splitting is longer as m increases. Therefore, contrary to traditional SVM software which uses a small block size, now for each inner iteration we should consider a large block. In Figure 5, we do not check $m = 20$ because the memory is not enough to store a block of data.

8.2.3. Data Compression. We check if compressing each block of data saves time. By running 10 outer iterations of BLOCK-L-D on the training set of webspam with $m =$

40, the implementation takes 3,230 seconds with compression, while 4,660 seconds without compression. Thus, the compression technique is very useful in this case.

The data loading time depends heavily on the disk reading speed. For a fast disk, compressing data may even slow down the training process.

8.3. Comparison of Existing Methods for Large-scale Data

In Section 7, we discussed existing approaches for training large-scale data. In this section, we first show that the sub-sampling strategy may downgrade the performance on the data sets we used. Then, we compare the proposed block minimization methods with other approaches for large-scale data.

To compare with the method of random sub-sampling, we shuffle each data set and train problem (2) by LIBLINEAR on subsets with different sizes. Figure 6 presents the performance of models trained on subsets of data. Results show that for our four data sets, using only a portion of data that can fit in memory may fail to obtain a model as good as using the entire data. In this situation, a method that considers the whole data set is still useful.

Next, we compare the following approaches which are able to train data larger than memory:

— BLOCK-L-10: This is the most stable one among all settings in Section 8.1 for the block minimization method.
— AvgBlock: A bagging approach introduced in Section 7.2. We average the models trained by LIBLINEAR with the default stopping condition on each block of data $B_j, j = 1, \ldots, m$. Although AvgBlock can be trained on a distributed system with multiple machines, here, we run it on a single computer.
— Vowpal_Wabbit: An online method mentioned in Section 7.3. The package (latest version 5.1) is available at `https://github.com/JohnLangford/vowpal_wabbit/wiki`. We use the default parameters.

For BLOCK-L-10 and AvgBlock, we use the same block splits as in Section 8.1 and select the parameter C in problem (2) by five-fold cross validation on the training set. Note that Vowpal_Wabbit considers an un-regularized problem, so these methods may give slightly different final testing accuracy values.

Similar to methods under the block minimization framework, Vowpal_Wabbit compresses data samples and stores them into a cache file. The time to generate the cache file is included in the training time measurements. Because Vowpal_Wabbit has a different implementation of compression, its initial time is different from that of the block minimization methods.

In yahoo-korea and kddcup10, data samples are sorted based on some pattens. Vowpal_Wabbit faces a slow convergence problem on these data sets. In contrast, block minimization methods solve this problem by implementing the random split algorithm in Section 5.3. Because Vowpal_Wabbit does not support this functionality, in the experiments, we randomly shuffle each of these two data sets and run Vowpal_Wabbit on the permuted data. The time to shuffle data is not included in Vowpal_Wabbit's training time. We also would like to note that Vowpal_Wabbit considers two tricks to speed up the training process. First, it uses single floating point arithmetic, although this decision may cause numerical inaccuracy. Second, it uses two threads for training. One is for loading data samples from the compressed file and the other is for updating the model.

We are interested in both whether BLOCK-L-10 and Vowpal_Wabbit can obtain a reasonable model quickly and how fast their final convergence is. Therefore, we show in Table III both the results after running the first and the tenth outer iterations. To show more details, we demonstrate the testing performance along the training time in

Table III: Training time and testing accuracy after the first and the tenth outer iterations. Time is in seconds. For each method, time for its initialization is included. For example, initially BLOCK-L-10 and AvgBlock must split data to files.

		yahoo-korea		kddcup10		webspam		epsilon	
		$C = 4$		$C = 0.1$		$C = 64$		$C = 1$	
method	#iter	acc.	time	acc.	time	acc.	time	acc.	time
BLOCK-L-10	1	85.97	259	88.49	862	99.32	1,944	89.12	1,802
	10	87.29	456	89.89	3,153	99.51	4,475	89.78	3,773
Vowpal_Wabbit	1	82.05	139	87.05	492	96.86	1,321	88.04	1,136
	10	85.97	345	86.54	1,891	98.30	1,979	89.50	1,758
AvgBlock	1	86.08	628	89.64	6,809	98.40	4,722	88.83	1,999

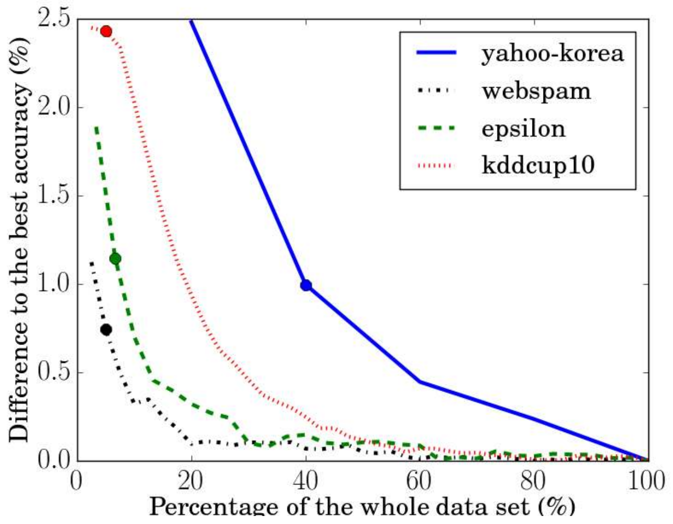
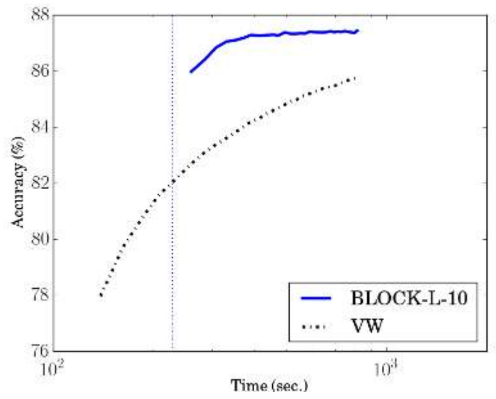


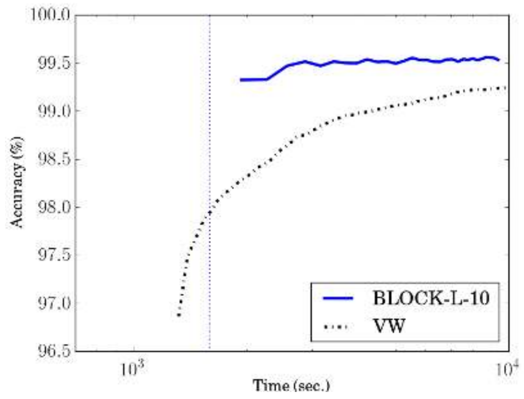
Fig. 6: Data size versus difference to the best testing accuracy. The marker on each curve indicates the size of the subset that can fit in memory. Results show that training only sub-sampled data may not be enough to achieve the best testing performance.

Figure 7. We omit AvgBlock in Figure 7, because it cannot be conducted in an iterative manner.

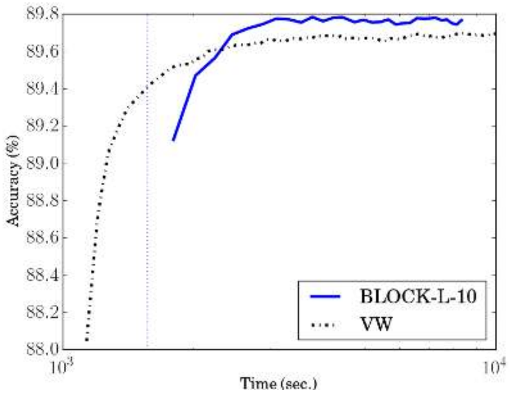
The results indicate that BLOCK-L-10 efficiently obtains a reasonably good model by using only one outer iteration. After 10 iterations, BLOCK-L-10 achieves an accuracy value almost the same as that of the final model. Vowpal_Wabbit takes less training time per iteration; however, because of solving an un-regularized problem instead of problem (2), it sometimes converges to a model with lower testing accuracy. On some data sets such as kddcup10, AvgBlock achieves similar accuracy values to BLOCK-L-10. However, on other data sets, BLOCK-L-10 is slightly better because it solves problem (2) using all the training data. The training time of AvgBlock, if divided by m, is very competitive. Thus, AvgBlock is potentially useful on a distributed environment.



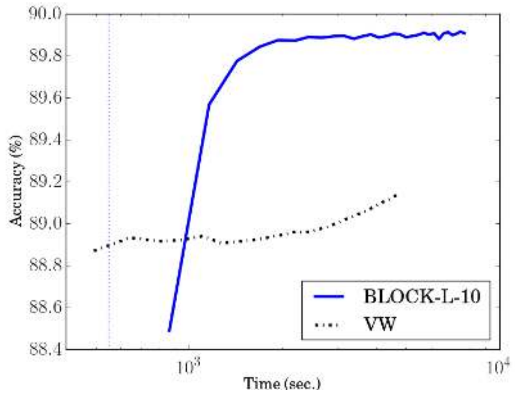
(a) yahoo-korea



(b) webspam



(c) epsilon



(d) kddcup10

Fig. 7: This figure compares BLOCK-L-10 and Vowpal_Wabbit by showing testing accuracy versus training time. Time (in seconds) is log-scaled. The blue dotted vertical line indicates time spent by BLOCK-L-10 for the initial split of data to blocks.

9. DISCUSSION AND CONCLUSIONS

The proposed block minimization framework can be extended in several directions. For examples, recently, Chang and Roth [2011] propose an algorithm based on the block minimization framework. At each step, their method updates the model using data consisting of a new data block loaded from disk and a block of samples cached in memory from previous steps. Because of using more informative data points at each step, the convergence is faster. Another possible extension is to combine the proposed framework with some data reduction techniques. For example, the hashing technique by Li and König [2010] can approximate the original data using a smaller number of features, so at each step we are able to include more instances in a block.

The discussion in Section 6 shows that implementing cross validation or multi-class classification may require extra memory space and some modifications of Algorithm 1. Thus, constructing a complete disk-level learning tool is certainly more complicated than implementing Algorithm 1. These challenges should be addressed in future research.

In summary, we propose and analyze a block minimization method for large linear classification when data cannot fit in memory. Experiments show that the proposed method can effectively handle data 20 times larger than the memory size.

Our code is available at

`http://www.csie.ntu.edu.tw/~cjlin/liblinear/exp.html`

ACKNOWLEDGMENTS

The authors thank anonymous reviewers of both this paper and the earlier conference version for very helpful comments.

A. PROOF OF THEOREM 3.1

If each sub-problem involves a finite number of coordinate descent updates, then Algorithm 1 can be regarded as a coordinate descent method. We apply Theorem 2.1 of Luo and Tseng [1992] to obtain the convergence results. The theorem requires that problem (3) satisfies certain conditions and in the coordinate descent method there is an integer t such that every α_i is iterated at least once every t successive updates (called almost cyclic rule in Luo and Tseng [1992]). Following the same analysis in the proof of Hsieh et al. [2008, Theorem 1], problem (3) satisfies the required conditions. Moreover, the two properties on $t_{j,k}$ imply the almost cyclic rule. Hence, both global and linear convergence results are obtained.

B. PROOF OF THEOREM 3.2

To begin, we discuss the stopping condition of LIBLINEAR. Each run of LIBLINEAR to solve a sub-problem generates $\{\boldsymbol{\alpha}^{k,j,v} \mid v = 1, \ldots, t_{k,j}+1\}$ with

$$\boldsymbol{\alpha}^{k,j} = \boldsymbol{\alpha}^{k,j,1} \text{ and } \boldsymbol{\alpha}^{k,j+1} = \boldsymbol{\alpha}^{k,j,t_{k,j}+1}.$$

We further let $i_{j,v}$ denote the index of the variable being updated by $\boldsymbol{\alpha}^{k,j,v+1} = \boldsymbol{\alpha}^{k,j,v} + d^* \boldsymbol{e}_{i_{j,v}}$, where d^* is the optimal solution of

$$\min_d f(\boldsymbol{\alpha}^{k,j,v} + d\boldsymbol{e}_{i_{j,v}}) \quad \text{subject to } 0 \le \alpha^{k,j,v}_{i_{j,v}} + d \le C, \tag{17}$$

and $\boldsymbol{e}_{i_{j,v}}$ is an indicator vector for the $(i_{j,v})$th element. All $t_{k,j}$ updates can be further separated to several rounds, where each one goes through all elements in B_j. LIBLINEAR checks the following stopping condition in the end of each round:

$$\max_{v \in \text{a round}} \nabla^P_{i_{j,v}} f(\boldsymbol{\alpha}^{k,j,v}) - \min_{v \in \text{a round}} \nabla^P_{i_{j,v}} f(\boldsymbol{\alpha}^{k,j,v}) \le \epsilon, \tag{18}$$

where ϵ is a tolerance and $\nabla^P f(\boldsymbol{\alpha})$ is the projected gradient:

$$\nabla^P_i f(\boldsymbol{\alpha}) = \begin{cases} \nabla_i f(\boldsymbol{\alpha}) & \text{if } 0 < \alpha_i < C, \\ \max(0, \nabla_i f(\boldsymbol{\alpha})) & \text{if } \alpha_i = C, \\ \min(0, \nabla_i f(\boldsymbol{\alpha})) & \text{if } \alpha_i = 0. \end{cases} \tag{19}$$

The reason that LIBLINEAR considers Eq. (18) is that from the optimality condition, $\boldsymbol{\alpha}^*$ is optimal if and only if $\nabla^P f(\boldsymbol{\alpha}^*) = \mathbf{0}$.

Next we prove the theorem by showing that for all $j = 1, \ldots, m$ there exists k_j such that

$$\forall k \ge k_j,\ t_{k,j} \le 2|B_j|. \tag{20}$$

Suppose that (20) does not hold. We can find a j and a sub-sequence $R \subset \{1, 2, \ldots\}$ such that

$$t_{k,j} > 2|B_j|, \forall k \in R. \tag{21}$$

Since $\{\boldsymbol{\alpha}^{k,j} \mid k \in R\}$ are in a compact set, we further consider a sub-sequence $M \subset R$ such that $\{\boldsymbol{\alpha}^{k,j} \mid k \in M\}$ converges to a limit point $\bar{\boldsymbol{\alpha}}$.

Let $\sigma \equiv \min_i Q_{ii}$. Following the explanation in Hsieh et al. [2008, Theorem 1], we only need to analyze indices with $Q_{ii} > 0$. Therefore, $\sigma > 0$. Lemma 2 of Hsieh et al. [2008] shows that

$$f(\boldsymbol{\alpha}^{k,j,v}) - f(\boldsymbol{\alpha}^{k,j,v+1}) \geq \frac{\sigma}{2}\|\boldsymbol{\alpha}^{k,j,v} - \boldsymbol{\alpha}^{k,j,v+1}\|^2, \quad \forall v = 1, \ldots, 2|B_j|. \tag{22}$$

The sequence $\{f(\boldsymbol{\alpha}^k) \mid k = 1, \ldots\}$ is decreasing and bounded below as the feasible region is compact. Hence

$$\lim_{k\to\infty} f(\boldsymbol{\alpha}^{k,j,v}) - f(\boldsymbol{\alpha}^{k,j,v+1}) = 0, \quad \forall v = 1, \ldots, 2|B_j|. \tag{23}$$

Using (23) and taking the limit on both sides of (22), we have

$$\begin{aligned} &\lim_{k\in M, k\to\infty} \boldsymbol{\alpha}^{k,j,2|B_j|+1} = \lim_{k\in M, k\to\infty} \boldsymbol{\alpha}^{k,j,2|B_j|} = \cdots \\ &= \lim_{k\in M, k\to\infty} \boldsymbol{\alpha}^{k,j,1} = \bar{\boldsymbol{\alpha}}. \end{aligned} \tag{24}$$

From the continuity of $\nabla f(\boldsymbol{\alpha})$ and (24), we have

$$\lim_{k\in M, k\to\infty} \nabla f(\boldsymbol{\alpha}^{k,j,v}) = \nabla f(\bar{\boldsymbol{\alpha}}),\ \forall v = 1, \ldots, 2|B_j|.$$

Hence there are ϵ and $\bar{k}$ such that $\forall k \in M$ with $k \geq \bar{k}$

$$|\nabla_i f(\boldsymbol{\alpha}^{k,j,v})| \leq \frac{\epsilon}{4} \text{ if } \nabla_i f(\bar{\boldsymbol{\alpha}}) = 0, \tag{25}$$

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v}) \geq \frac{3\epsilon}{4} \text{ if } \nabla_i f(\bar{\boldsymbol{\alpha}}) > 0, \tag{26}$$

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v}) \leq -\frac{3\epsilon}{4} \text{ if } \nabla_i f(\bar{\boldsymbol{\alpha}}) < 0, \tag{27}$$

for any $i \in B_j$, $v \leq 2|B_j|$.

When we update $\boldsymbol{\alpha}^{k,j,v}$ to $\boldsymbol{\alpha}^{k,j,v+1}$ by changing the ith element (i.e., $i = i_{j,v}$) in the first round, the optimality condition for (17) implies that one of the following three situations occurs:

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v+1}) = 0, \tag{28}$$

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v+1}) > 0 \text{ and } \alpha_i^{k,j,v+1} = 0, \tag{29}$$

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v+1}) < 0 \text{ and } \alpha_i^{k,j,v+1} = C. \tag{30}$$

From (25)-(27), we have that

$$i \text{ satisfies } \begin{cases} (28) \\ (29) \\ (30) \end{cases} \Rightarrow \begin{cases} \nabla_i f(\bar{\boldsymbol{\alpha}}) = 0 \\ \nabla_i f(\bar{\boldsymbol{\alpha}}) \geq 0 \\ \nabla_i f(\bar{\boldsymbol{\alpha}}) \leq 0 \end{cases}. \tag{31}$$

In the second round, assume α_i is changed at the v'th update. From (31) and (25)-(27), we have

$$|\nabla_i f(\boldsymbol{\alpha}^{k,j,v'})| \leq \frac{\epsilon}{4}, \tag{32}$$

or

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v'}) \geq -\frac{\epsilon}{4} \text{ and } \alpha_i^{k,j,v'} = 0, \tag{33}$$

or

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v'}) \leq \frac{\epsilon}{4} \text{ and } \alpha_i^{k,j,v'} = C. \tag{34}$$

Using (32)-(34), the projected gradient defined in (19) satisfies

$$|\nabla_i^P(\boldsymbol{\alpha}^{k,j,v'})| \leq \frac{\epsilon}{4}.$$

This result holds for all $i \in B_j$. Therefore,

$$\begin{aligned}&\max_{v \in \text{2nd round}} \nabla^P_{i_{j,v}}(\boldsymbol{\alpha}^{k,j,v}) - \min_{v \in \text{2nd round}} \nabla^P_{i_{j,v}}(\boldsymbol{\alpha}^{k,j,v}) \\ \leq\ & \frac{\epsilon}{4} - (-\frac{\epsilon}{4}) = \frac{\epsilon}{2} < \epsilon.\end{aligned}$$

Thus, (18) is valid in the second round. Then $t_{k,j} = 2|B_j|$ violates (21). Hence (20) holds and the theorem is obtained.

REFERENCES

BERTSEKAS, D. P. 1999. *Nonlinear Programming* Second Ed. Athena Scientific, Belmont, MA 02178-9998.

BOTTOU, L. 2007. Stochastic gradient descent examples. `http://leon.bottou.org/projects/sgd`.

BOYD, S. AND VANDENBERGHE, L. 2004. *Convex Optimization*. Cambridge University Press.

BREIMAN, L. 1996. Bagging predictors. *Machine Learning 24,* 2, 123–140.

CHAKRABARTI, D., AGARWAL, D., AND JOSIFOVSKI, V. 2008. Contextual advertising by combining relevance with click feedback. In *Proceeding of the 17th international conference on World Wide Web*. 417–426.

CHANG, C.-C. AND LIN, C.-J. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology 2*, 27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

CHANG, E., ZHU, K., WANG, H., BAI, H., LI, J., QIU, Z., AND CUI, H. 2008. Parallelizing support vector machines on distributed computers. In *Advances in Neural Information Processing Systems 20*, J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds. MIT Press, Cambridge, MA, 257–264.

CHANG, K.-W. AND ROTH, D. 2011. Selective block minimization for faster convergence of limited memory large-scale linear models. In *Proceedings of the Seventeenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

CRAMMER, K. AND SINGER, Y. 2002. On the learnability and design of output codes for multiclass problems. *Machine Learning* 2–3, 201–233.

FAN, R.-E., CHANG, K.-W., HSIEH, C.-J., WANG, X.-R., AND LIN, C.-J. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research 9*, 1871–1874.

FERRIS, M. AND MUNSON, T. 2003. Interior point methods for massive support vector machines. *SIAM Journal on Optimization 13,* 3, 783–804.

HSIEH, C.-J., CHANG, K.-W., LIN, C.-J., KEERTHI, S. S., AND SUNDARARAJAN, S. 2008. A dual coordinate descent method for large-scale linear SVM. In *Proceedings of the Twenty Fifth International Conference on Machine Learning (ICML)*.

JOACHIMS, T. 1998. Making large-scale SVM learning practical. In *Advances in Kernel Methods – Support Vector Learning*, B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds. MIT Press, Cambridge, MA, 169–184.

JOACHIMS, T. 2006. Training linear SVMs in linear time. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

KEERTHI, S. S., ANDKAI WEI CHANG, S. S., HSIEH, C.-J., AND LIN, C.-J. 2008. A sequential dual method for large scale multi-class linear SVMs. In *Proceedings of the Forteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

LANGFORD, J., LI, L., AND STREHL, A. 2007. Vowpal Wabbit. `https://github.com/JohnLangford/vowpal_wabbit/wiki`.

LANGFORD, J., LI, L., AND ZHANG, T. 2009. Sparse online learning via truncated gradient. *Journal of Machine Learning Research 10*, 771–801.

LANGFORD, J., SMOLA, A., AND ZINKEVICH, M. 2009. Slow learners are fast. In *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, Eds. 2331–2339.

LI, P. AND KÖNIG, A. C. 2010. b-Bit minwise hashing. In *Proceedings of the Nineteenth International Conference on World Wide Web*. 671–680.

LUO, Z.-Q. AND TSENG, P. 1992. On the convergence of coordinate descent method for convex differentiable minimization. *Journal of Optimization Theory and Applications 72,* 1, 7–35.

MEMISEVIC, R. 2006. Dual optimization of conditional probability models. Tech. rep., Department of Computer Science, University of Toronto.

MORSE, JR., K. G. 2005. Compression tools compared. *Linux Journal*.

PÉREZ-CRUZ, F., FIGUEIRAS-VIDAL, A. R., AND ARTÉS-RODRÍGUEZ, A. 2004. Double chunking for solving SVMs for very large datasets. In *Proceedings of Learning 2004, Spain*.

RÜPING, S. 2000. mySVM - another one of those support vector machines. Software available at `http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/`.

SERAFINI, T. AND ZANNI, L. 2005. On the working set selection in gradient projection-based decomposition techniques for support vector machines. *Optimization Methods and Software 20*, 583–596.

SHALEV-SHWARTZ, S., SINGER, Y., AND SREBRO, N. 2011. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming 127,* 1, 3–30.

TONG, S. 2010. Lessons learned developing a practical large scale machine learning system. Google Research Blog. `http://googleresearch.blogspot.com/2010/04/lessons-learned-developing-practical.html`.

YU, H., YANG, J., AND HAN, J. 2003. Classifying large data sets using SVMs with hierarchical clusters. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, New York, NY, USA, 306–315.

YU, H.-F., HSIEH, C.-J., CHANG, K.-W., AND LIN, C.-J. 2010. Large linear classification when data cannot fit in memory. In *Proceedings of the Sixteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

YU, H.-F., LO, H.-Y., HSIEH, H.-P., LOU, J.-K., MCKENZIE, T. G., CHOU, J.-W., CHUNG, P.-H., HO, C.-H., CHANG, C.-F., WEI, Y.-H., WENG, J.-Y., YAN, E.-S., CHANG, C.-W., KUO, T.-T., LO, Y.-C., CHANG, P. T., PO, C., WANG, C.-Y., HUANG, Y.-H., HUNG, C.-W., RUAN, Y.-X., LIN, Y.-S., LIN, S.-D., LIN, H.-T., AND LIN, C.-J. 2011. Feature engineering and classifier ensemble for KDD Cup 2010. In *JMLR Workshop and Conference Proceedings*. To appear.

YUAN, G.-X., HO, C.-H., AND LIN, C.-J. 2011. Recent advances of large-scale linear classification. *Proceedings of IEEE*. Submitted.

ZHANG, T. 2002. On the dual formulation of regularized linear systems with convex risks. *Machine Learning 46,* 1–3, 91–129.

ZHU, Z. A., CHEN, W., WANG, G., ZHU, C., AND CHEN, Z. 2009. P-packSVM: Parallel primal gradient descent kernel SVM. In *Proceedings of the IEEE International Conference on Data Mining*.

ZINKEVICH, M., WEIMER, M., SMOLA, A., AND LI, L. 2010. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems 23*, J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds. 2595–2603.

Received Mar 2011; revised July 2011; accepted October 2011