Large Linear Classification When Data Cannot Fit In Memory

Hsiang-Fu Yu
Dept. of Computer Science
National Taiwan University
Taipei 106, Taiwan
b93107@csie.ntu.edu.tw

Cho-Jui Hsieh
Dept. of Computer Science
National Taiwan University
Taipei 106, Taiwan
b92085@csie.ntu.edu.tw

Kai-Wei Chang
Dept. of Computer Science
National Taiwan University
Taipei 106, Taiwan
b92084@csie.ntu.edu.tw

Chih-Jen Lin
Dept. of Computer Science
National Taiwan University
Taipei 106, Taiwan
cjlin@csie.ntu.edu.tw

ABSTRACT

Recent advances in linear classification have shown that for applications such as document classification, the training 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. As data cannot fit in memory, many design considerations are very different from those for traditional algorithms. 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

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 compares *linear* SVM implementations. The competition evaluates the 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. Existing training algorithms often 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 [1] 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:

$$\begin{aligned}\text{training time} = {} & \text{time to run data in memory} + \\ & \text{time to access data from disk.}\end{aligned} \quad (1)$$

Traditional training algorithms, assuming that the second

[1]By linear classification we mean that data remain in the input space and kernel methods are not used.

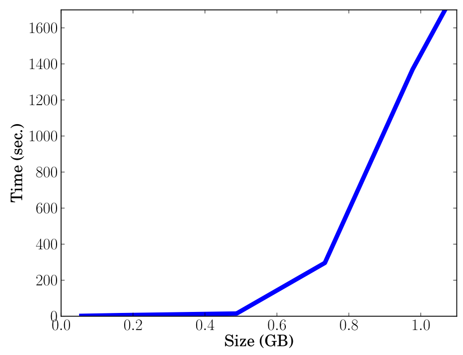


Figure 1: Data size versus training time on a machine with 1GB memory.

[2]`http://largescale.first.fraunhofer.de/workshop`
[3]`http://hunch.net/?p=330`

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., [2, 3, 4, 5]) 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 [6], 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., [7, 8]). However, not only writing programs on a distributed system is difficult, but also the data communication/synchronization may cause significant overheads. The second approach considers online learning algorithms. Since 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 [9]. Existing implementations (including those in large Internet companies) may lack important functions such as evaluations by different criteria, parameter selection, or feature selection.

This paper aims to construct large linear classifiers for ordinary users. 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.

If sub-sampling does not downgrade the accuracy, some (e.g., [10]) have proposed approaches to select 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., [3, 4]). In contrast, we conduct experiments in a real environment without enough memory. An earlier linear-SVM study [11] 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 in Sections 3 and 4. Techniques to minimize the training time modeled in (1) are in Section 5. Section 6 discusses the implementation of cross validation, multi-class classification, and incremental/decremental settings. We show experiments in Section 7 and give conclusions in Section 8.

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 $\boldsymbol{w}$
```

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 data 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:[4]

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

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:

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

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

As 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 *continuous* 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 methods to achieve them to a certain degree.

In unconstrained optimization, block minimization is a classical method (e.g., [12, Chapter 2.7]). Each step of this method updates a block of *variables*, but here we need a connection to 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

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

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). We show two implementations in Sections 3 and 4, respectively.

We discuss some implementation considerations for Algorithm 1. For the 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.

These two terms respectively correspond to the two parts in (1) for modeling the training time.

Many studies have applied block minimization to train SVM or other machine learning problems, but we might be the first to consider it in the disk level. Indeed 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 confined here as data associated with each B_j must be pre-stored in a file before running Algorithm 1.

Regarding the block size, we now go back to analyze (4). If data are all in memory, $T_d(|B|) = 0$. For $T_m(|B|)$, people observe that if $|B|$ linearly increases, then

$$|B| \nearrow, T_m(|B|) \nearrow, \text{ and } \#\text{outer-iters} \searrow. \tag{5}$$

$T_m(|B|)$ is generally more than linear to $|B|$, so $T_m(|B|) \times l/|B|$ 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 [13] and 10 in SVM^{light} [14]. With $T_d(|B|) > 0$, the situation is now very different. At each outer iteration, the cost is

$$T_m(|B|) \times \frac{l}{|B|} + T_d(|B|) \times \frac{l}{|B|}. \tag{6}$$

The second term is for reading l instances. As reading each block of data takes some initial time, a smaller number of blocks reduces the cost. Hence the second term in (6) 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 7.

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
        (7) to obtain $d^*_{B_j}$
    3.3 $\alpha_{B_j} \leftarrow \alpha_{B_j} + d^*_{B_j}$
        Update $w$ by (10)
```

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. Assume $\bar{B}_j = \{1, \ldots, l\} \backslash B_j$, at each inner iteration we solve the following sub-problem.

$$\begin{aligned} \min_{\boldsymbol{d}_{B_j}} \quad & f(\boldsymbol{\alpha} + \boldsymbol{d}) \\ \text{subject to} \quad & \boldsymbol{d}_{\bar{B}_j} = \mathbf{0} \text{ and } 0 \le \alpha_i + d_i \le C, \ \forall i \in B_j. \end{aligned} \tag{7}$$

That is, we change $\boldsymbol{\alpha}_{B_j}$, while fix $\boldsymbol{\alpha}_{\bar{B}_j}$. We then update $\boldsymbol{\alpha}_{B_j}$ using the solution of (7). Then Algorithm 1 reduces to the standard block minimization procedure, so the convergence to the optimal function value of (3) holds [12, Proposition 2.7.1].

We must ensure that at each inner iteration, only one block of data is needed. With the constraint $\boldsymbol{d}_{\bar{B}_j} = \mathbf{0}$ in (7),

$$f(\boldsymbol{\alpha} + \boldsymbol{d}) = \frac{1}{2}\boldsymbol{d}_{B_j}^T Q_{B_j B_j} \boldsymbol{d}_{B_j} + (Q_{B_j,:}\boldsymbol{\alpha} - \boldsymbol{e}_{B_j})^T \boldsymbol{d}_{B_j} + f(\boldsymbol{\alpha}), \tag{8}$$

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 (8) involves all training data, a situation violating the requirement in Algorithm 1. Fortunately, by maintaining

$$\boldsymbol{w} = \sum_{i=1}^{l} \alpha_i y_i \boldsymbol{x}_i, \tag{9}$$

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 (7), we consider (9) and use

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \sum_{r \in B_j} d^*_r y_r \boldsymbol{x}_r. \tag{10}$$

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

For solving the sub-problem (7), as all the information is available in the memory, any bound-constrained optimization method can be applied. We consider LIBLINEAR [1], which implements a coordinate descent method (i.e., block minimization with a single element in each block). Then Algorithm 2 becomes a two-level block minimization method. The two-level setting had been used before for SVM or other applications (e.g., [15, 16, 17]), but ours might be the first to associate the inner level with memory and the outer level with disk.

Algorithm 2 converges if each sub-problem is exactly solved. Practically we often obtain an approximate solution by imposing a stopping criterion. We then 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 using LIBLINEAR for solving the sub-problem. Let $\{\boldsymbol{\alpha}^k\}$ be the sequence generated by Algorithm 2, where k is the index of outer iterations. As 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 (7) and we let $t_{k,j}$ be the number of updates.

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

Theorem 1
If applying a coordinate descent method to solve (7) *with the following properties:*

1. *each α_i, $i \in B_j$ is updated at least once, and*

2. *$\{t_{k,j}\}$ is uniformly bounded,*

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

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

The proof is in appendix. With Theorem 1, the condition 2 mentioned in the beginning of Section 2 holds. For condition 3 on the convergence speed, block minimization does have fast convergence rates. However, for problems like document classification, some (e.g., [5]) have shown that we do not need many iterations to get a reasonable model. Though [5] differs from us by restricting $|B| = 1$, we hope to enjoy the same property of not needing many iterations. Experiments in Section 7 confirm that for some document data this property holds.

Next we discuss various ways to fulfill the two properties in Theorem 1.

3.1 Loosely Solving the Sub-problem

A simple setting to satisfy Theorem 1's two properties is to go through all variables in B_j a fixed number of times. Then not only t_{kj} is 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 very loosely solve the sub-problem (7). While the cost per block is cheaper, the number of outer iterations may be large. Through experiments in Section 7, we discuss how the number of passes affects the running time. A special case is to go through all $\alpha_i, i \in B_j$ exactly 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, using a random permutations of B_j's elements as the order for update 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. As an upper bound on the number of iterations does not reveal how accurate the solution is, most optimization software consider the gradient information. We check the setting in LIBLINEAR. Its gradient-based stopping condition (details shown in appendix) guarantees the finite termination in solving each sub-problem (7). Thus the procedure can move on to the next sub-problem without problem. Regarding the convergence, to use Theorem 1, we must show that $\{t_{k,j}\}$ is uniformly bounded:

Theorem 2
If coordinate descent steps with LIBLINEAR's stopping condition are used to solve (7)*, then Algorithm 2 either terminates in a finite number of outer iterations or*

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

Therefore, if LIBLINEAR is used to solve (7), then Theorem 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 problem. Since 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 (7). In contrast, existing stochastic gradient descent methods possess a nice property that at each step only certain data are used. In this section, we study how Pegasos [3] can by used for implementing an 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:

$$\bar{\boldsymbol{w}} = \boldsymbol{w} - \eta^t \nabla^t, \tag{11}$$

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

$$\nabla^t = \frac{1}{lC}\boldsymbol{w} - \frac{1}{|B|}\sum_{i \in B^+} y_i \boldsymbol{x}_i, \tag{12}$$

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

$$\boldsymbol{w} \leftarrow \min(1, \frac{\sqrt{lC}}{\|\bar{\boldsymbol{w}}\|})\bar{\boldsymbol{w}}. \tag{13}$$

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:

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

```
1. Split $\{1,\ldots,l\}$ to $B_1,\ldots,B_m$ and store data into $m$ files accordingly.
2. $t = 0$ and initial $\boldsymbol{w} = \mathbf{0}$.
3. For $k = 1, 2, \ldots$
     For $j = 1, \ldots, m$
       3.1. Find a partition of $B_j$: $B_j^1, \ldots, B_j^{\bar{r}}$.
       3.2. For $r = 1, \ldots, \bar{r}$
              • Use $B_j^r$ as $B$ to conduct the update (11)-(13).
              • $t \leftarrow t + 1$
```

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.

For the convergence, though Algorithm 3 is a special case of Pegasos, we cannot apply its convergence proof [3, Corollary 1], which requires that all data $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$ are used at each update. 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 (6), 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, $T_d(|B|)$ 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 [18]. We choose a widely used compression library zlib for our implementation.[5] Experiments in Section 7 show that 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 $B_1, B_2, \ldots$, B_m. We can consider other ways such as a permutation of blocks to decide the order of sub-problems. In LIBLINEAR's coordinate descent implementation, the authors randomly permute all variables at each iteration and report faster convergence. We adopt a permutation strategy here as the loading time is similar regardless of the order of sub-problems.

[5]http://www.zlib.net

Algorithm 4 Splitting data into blocks

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

5.3 Split of Data

An important step of Algorithm 1 is to split training data to m files. We need a careful design as data cannot be loaded into memory. To begin, we find the size of data and decide the value m based on the memory size. This step does not have to go through the whole data set as the operating system provides information such as file sizes. Then we can sequentially read data instances and save them to m files. 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. 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 (3) is not practically useful. Other functions such as cross validation (for parameter selection) or multi-class classification are very important. We discuss how to implement these functions based on the design in Section 2.

6.1 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 remaining groups are for training. However, the loading time is v times more than training a single model. To save the disk accessing time, a more complicated 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 $\boldsymbol{w}^1, \ldots, \boldsymbol{w}^v$.

6.2 Multi-class Classification

Existing multi-class approaches either train several two-class problems (e.g., one-against-one and one-against-the rest) or solve one single optimization problem (e.g., [19]). Take one-against-the rest for a K-class problem as an example. We train K classifiers, where each one separates a class from the rest. Similar to the situation in cross validation, the disk access time is K times more if we sequentially train K models. Using the same technique, we split each blocks B_j to $B_j^1, \ldots, B_j^K$ according to the class information, Then K sub-problems are solved to update vectors $\boldsymbol{w}^1, \ldots, \boldsymbol{w}^K$. Finally we obtain K models simultaneously. The one-against-one

Table 1: 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
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

approach is less suitable as it needs $K(K-1)/2$ vectors for $\boldsymbol{w}$, which may consume too much memory. For one-against-the rest and the approach in [19], they both need K vectors.

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

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \sum_{i:\boldsymbol{x}_i \text{ being added}} y_i\alpha_i\boldsymbol{x}_i - \sum_{i:\boldsymbol{x}_i \text{ being removed}} y_i\alpha_i\boldsymbol{x}_i. \tag{14}$$

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 (14), 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 solving the primal problem by Pegasos for each block, Algorithm 3 can be directly applied for incremental or decremental settings.

7. EXPERIMENTS

In this section, we conduct experiments to analyze the performance of the proposed approach. We also investigate several implementation issues discussed in Section 5.

7.1 Data and Experimental Environment

We consider two document data sets yahoo-korea[6] and webspam, and an artificial data epsilon.[7] Table 1 summarizes the data statistics.

We randomly split 4/5 data for training and 1/5 for testing. 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. The value C in (2) is set to one.

We conduct experiments on a 64-bit machine with 1GB RAM. Due to the space consumed by the operating system, the real memory capacity we can use is 895MB.

7.2 A Related Method

For the comparison we include another method StreamSVM [20], which performs only a single pass over data. The method initiates with a single data point. When a new data point is read, it checks whether the point is contained in a ball enclosing past data. If so, it continues to next data point. If not, it updates the center and radius of the ball to cover the point. Because this method is very different from our approach, we omit its details here.

7.3 Training Time and Testing Accuracy

We compare the following methods:

- BLOCK-L-N: Algorithm 2 with LIBLINEAR to solve each sub-problem. LIBLINEAR goes through the 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 the whole 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.
- LIBLINEAR: The standard LIBLINEAR without any modification to handle the situation if data cannot fit in memory.
- StreamSVM

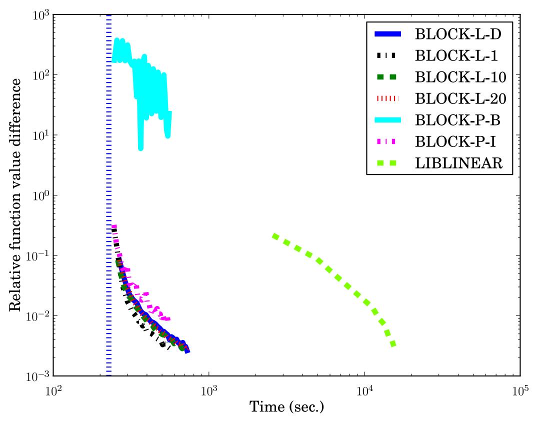
For all methods under the framework of Algorithms 1, the number of blocks is 5 for yahoo-korea, 40 for webspam and 30 for epsilon. We make sure that no other jobs are running on the same machine and report *wall clock* time in all experiments. We include all data loading time and, for Algorithm 1, the initial time to split and compress data into blocks. It takes around 228 seconds to split yahoo-korea, 1,594 seconds to split webspam and 1,237 seconds to split epsilon. For LIBLINEAR, the loading time for yahoo-korea is 103 seconds, 829 seconds for webspam and 560 seconds for epsilon.

Figure 2 presents two results:

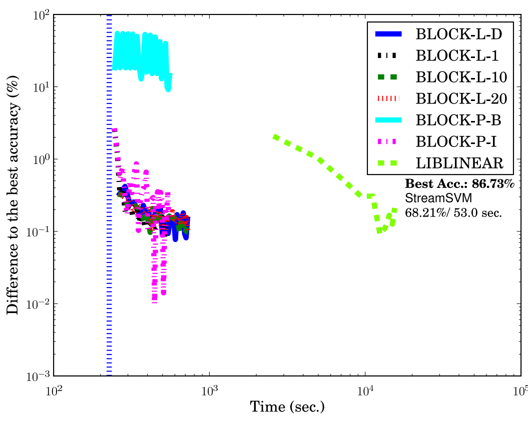
1. Training time versus the relative difference to the optimum
$$\left|\frac{f^P(\boldsymbol{w}) - f^P(\boldsymbol{w}^*)}{f^P(\boldsymbol{w}^*)}\right|,$$
where f^P is the primal objective function in (2) and $\boldsymbol{w}^*$ is the optimal solution. Since $\boldsymbol{w}^*$ is not really available, 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 best testing accuracy among all methods.

[6]This data set is not publicly available

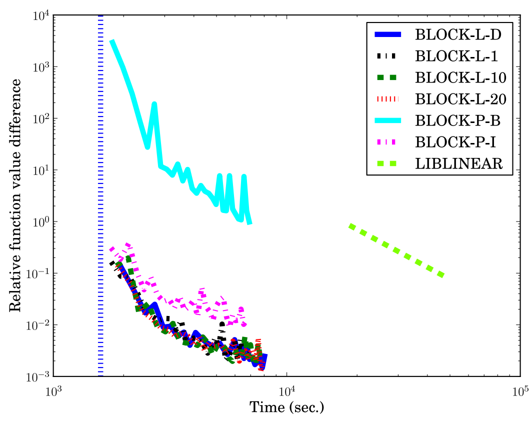
[7]webspam and epsilon can be downloaded at `http://largescale.first.fraunhofer.de/instructions/`



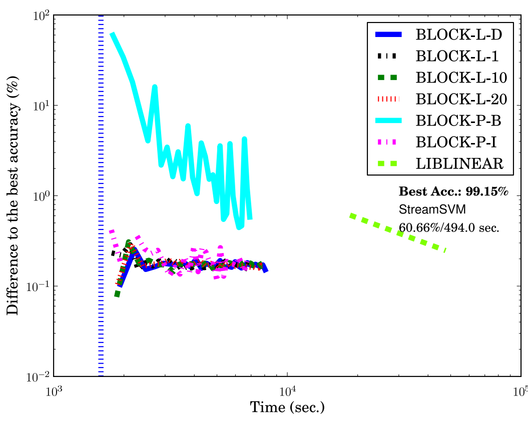
(a) yahoo-korea: Relative difference to optimum.



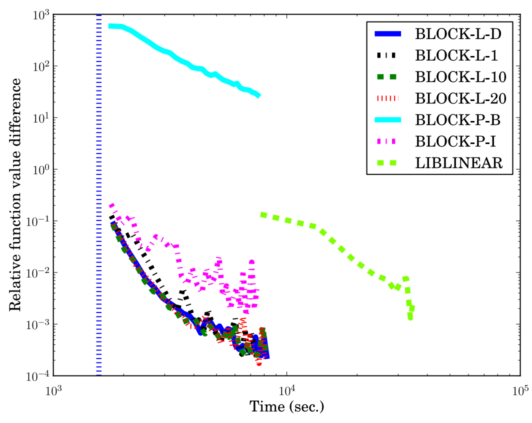
(b) yahoo-korea: Difference to the best accuracy.



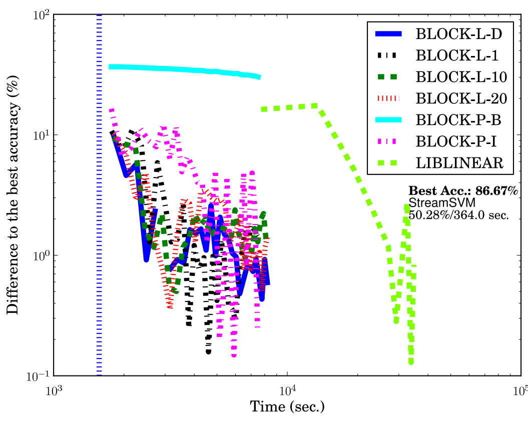
(c) webspam: Relative difference to optimum.



(d) webspam: Difference to the best accuracy.



(e) epsilon: Relative difference to optimum.



(f) epsilon: Difference to the best accuracy.

Figure 2: This table shows the relative function value difference to the minimum and the accuracy difference to the best testing accuracy. Time (in seconds) is log scaled. The blue dotted vertical line indicates time spent by Algorithms 1-based methods for the initial split of data to blocks. StreamSVM goes through data only once, so we present only one accuracy value. Note that in Figure 2(f), the curve of BLOCK-L-D is not connected, where the missing point corresponds to the best accuracy.

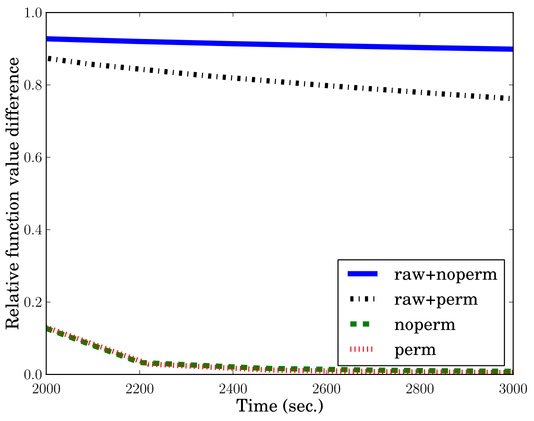


Figure 3: Effectiveness of two implementation techniques: *raw*: no random assignment in the initial data splitting. *perm*: a random order of blocks at each outer iteration. BLOCK-L-D is used.

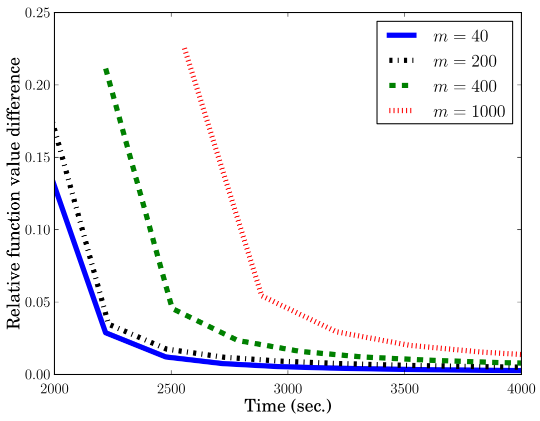


Figure 4: Convergence speed of using different m (number of blocks). BLOCK-L-D is used.

Clearly, LIBLINEAR suffers from slow disk swapping due to the random access of data. For Algorithm 1-based methods, BLOCK-L-$*$ methods (using LIBLINEAR) are faster than BLOCK-P-$*$ (using Pegasos) methods. The reason seems to be that for BLOCK-P-$*$, the information of each block is underutilized. In particular, BLOCK-P-B suffers from very slow convergence as for each block it 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 Figure 2(a), BLOCK-L-1 (for each block LIBLINEAR goes through all data only once) is slightly faster than BLOCK-L-D (for each block running LIBLINEAR 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. For StreamSVM, because of passing data only once, its accuracy is lower than others.

Note that the objective values of BLOCK-P-$*$ methods may not be decreasing as Pegasos does not have this property. All BLOCK-$*$ methods except BLOCK-P-B needs around four iterations to achieve reasonable accuracy values. This number of iterations is small, so we do not need to read the training set many times.

7.4 Initial Block Split and Random Permutation of Sub-problems

Section 5 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 iteration is useful. We are interested in their effectiveness. Figure 3 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 iteration slightly improves the training time.

7.5 Block Size

In Figure 4, we present the training speed of BLOCK-L-D by using various block sizes (equivalently, numbers of blocks). The data webspam is considered. The training time of using $m = 40$ blocks is smaller than that of $m = 400$ or 1000. This result is consistent with the discussion in Section 2. When the number of blocks is smaller (i.e., larger block size), from (5), the cost of operations on each block increases. However, as we read less files, the total time is shorter. Furthermore, the initial split time is longer as m increases. Therefore, contrary to traditional SVM software which use small block sizes, now for each inner iteration we should consider a large block. We do not check $m = 20$ because the memory is not enough to load a block of data.

7.6 Data Compression

We check if compressing each block saves time. By running 10 outer iterations of BLOCK-L-D on the training set of webspam with $m = 40$, the implementation with compression takes 3,230 seconds, but without compression needs 4,660 seconds. Thus the compression technique is very useful.

8. DISCUSSION AND CONCLUSIONS

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 learning tool is certainly more complicated than implementing Algorithm 1. There are many new and challenging future research issues.

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`

9. ACKNOWLEDGMENTS

The authors thank anonymous reviewers for helpful comments. This work was supported in part by the National Science Council of Taiwan via the grant 98-2221-E-002-136-MY3.

10. REFERENCES

[1] 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.

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

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

[4] L. Bottou, "Stochastic gradient descent examples," 2007. `http://leon.bottou.org/projects/sgd`.

[5] 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.

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

[7] 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.

[8] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen, "P-packSVM: Parallel primal gradient descent kernel SVM," in *ICDM*, 2009.

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

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

[11] M. Ferris and T. Munson, "Interior point methods for massive support vector machines," *SIAM Journal on Optimization*, vol. 13, no. 3, pp. 783–804, 2003.

[12] D. P. Bertsekas, *Nonlinear Programming*. Belmont, MA 02178-9998: Athena Scientific, second ed., 1999.

[13] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

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

[15] R. Memisevic, "Dual optimization of conditional probability models," tech. rep., Department of Computer Science, University of Toronto, 2006.

[16] 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.

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

[18] K. G. Morse, Jr., "Compression tools compared," *Linux Journal*, 2005.

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

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

[21] 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

Proof of Theorem 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 [21] to obtain the convergence results. The theorem requires that (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 [21]). Following the same analysis in the proof of [5, Theorem 1], (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.

Proof of Theorem 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

$$\min_d f(\boldsymbol{\alpha}^{k,j,v} + d\boldsymbol{e}_{i_{j,v}}) \quad \text{subject to } 0 \le \alpha^{k,j,v}_{i_{j,v}} + d \le C, \tag{15}$$

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:

$$\max_{v \in \text{a round}} \nabla^P_{i_{j,v}} f(\boldsymbol{\alpha}^{k,j,v}) - \min_{v \in \text{a round}} \nabla^P_{i_{j,v}} f(\boldsymbol{\alpha}^{k,j,v}) \le \epsilon, \tag{16}$$

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

$$\nabla^P_i f(\boldsymbol{\alpha}) = \begin{cases} \nabla_i f(\boldsymbol{\alpha}) & \text{if } 0 < \alpha_i < C, \\ \max(0, \nabla_i f(\boldsymbol{\alpha})) & \text{if } \alpha_i = C, \\ \min(0, \nabla_i f(\boldsymbol{\alpha})) & \text{if } \alpha_i = 0. \end{cases} \tag{17}$$

The reason that LIBLINEAR considers (16) 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

$$\forall k \ge k_j, \; t_{k,j} \le 2|B_j|. \tag{18}$$

Suppose that (18) does not hold. We can find a j and a sub-sequence $R \subset \{1, 2, \ldots\}$ such that

$$t_{k,j} > 2|B_j|, \forall k \in R. \tag{19}$$

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 [5, Theorem 1], we only need to analyze indices with $Q_{ii} > 0$. Therefore, $\sigma > 0$. Lemma 2 of [5] shows that

$$\begin{aligned} &f(\boldsymbol{\alpha}^{k,j,v}) - f(\boldsymbol{\alpha}^{k,j,v+1}) \ge \frac{\sigma}{2}\|\boldsymbol{\alpha}^{k,j,v} - \boldsymbol{\alpha}^{k,j,v+1}\|^2, \\ &\forall v = 1, \ldots, 2|B_j|. \end{aligned} \tag{20}$$

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

$$\begin{aligned}&\lim_{k\to\infty} f(\boldsymbol{\alpha}^{k,j,v}) - f(\boldsymbol{\alpha}^{k,j,v+1}) = 0,\\ &\forall v = 1, \ldots, 2|B_j|.\end{aligned} \tag{21}$$

Using (21) and taking the limit on both sides of (20), we have

$$\begin{aligned}&\lim_{k\in M, k\to\infty} \boldsymbol{\alpha}^{k,j,2|B_j|+1} = \lim_{k\in M, k\to\infty} \boldsymbol{\alpha}^{k,j,2|B_j|} = \cdots\\ =&\lim_{k\in M, k\to\infty} \boldsymbol{\alpha}^{k,j,1} = \bar{\boldsymbol{\alpha}}.\end{aligned} \tag{22}$$

From the continuity of $\nabla f(\boldsymbol{\alpha})$ and (22), 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 $k \geq \bar{k}$

$$|\nabla_i f(\boldsymbol{\alpha}^{k,j,v})| \leq \frac{\epsilon}{4} \text{ if } \nabla_i f(\bar{\boldsymbol{\alpha}}) = 0, \tag{23}$$

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v}) \geq \frac{3\epsilon}{4} \text{ if } \nabla_i f(\bar{\boldsymbol{\alpha}}) > 0, \tag{24}$$

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v}) \leq -\frac{3\epsilon}{4} \text{ if } \nabla_i f(\bar{\boldsymbol{\alpha}}) < 0, \tag{25}$$

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 (15) implies that one of the following three situations occurs:

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v+1}) = 0, \tag{26}$$

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v+1}) > 0 \text{ and } \alpha_i^{k,j,v+1} = 0, \tag{27}$$

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v+1}) < 0 \text{ and } \alpha_i^{k,j,v+1} = C. \tag{28}$$

From (23)-(25), we have that

$$i \text{ satisfies } \begin{cases}(26)\\(27)\\(28)\end{cases} \Rightarrow \begin{cases}\nabla_i f(\bar{\boldsymbol{\alpha}}) = 0\\ \nabla_i f(\bar{\boldsymbol{\alpha}}) \geq 0\\ \nabla_i f(\bar{\boldsymbol{\alpha}}) \leq 0\end{cases}. \tag{29}$$

In the second round, assume α_i is not changed again until the v'th update. From (29) and (23)-(25), we have

$$|\nabla_i f(\boldsymbol{\alpha}^{k,j,v'})| \leq \frac{\epsilon}{4}, \tag{30}$$

or

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v'}) \geq -\frac{\epsilon}{4} \text{ and } \alpha_i^{k,j,v'} = 0, \tag{31}$$

or

$$\nabla_i f(\boldsymbol{\alpha}^{k,j,v'}) \leq \frac{\epsilon}{4} \text{ and } \alpha_i^{k,j,v'} = C. \tag{32}$$

Using (30)-(32), the projected gradient defined in (17) satisfies

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

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

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

Thus (16) is valid in the second round. Then $t_{k,j} = 2|B_j|$ violates (19). Hence (18) holds and the theorem is obtained.