Indexed Block Coordinate Descent for Large-Scale Linear Classification with Limited Memory

Ian E.H. Yen
National Taiwan University
r00922017@csie.ntu.edu.tw

Chun-Fu Chang
National Taiwan University
r99725033@ntu.edu.tw

Ting-Wei Lin
National Taiwan University
b97083@csie.ntu.edu.tw

Shan-Wei Lin
National Taiwan University
b99023@csie.ntu.edu.tw

Shou-De Lin
National Taiwan University
sdlin@csie.ntu.edu.tw

ABSTRACT

Linear Classification has achieved complexity linear to the data size. However, in many applications, data contain large amount of samples that does not help improve the quality of model, but still cost much I/O and memory to process. In this paper, we show how a Block Coordinate Descent method based on Nearest-Neighbor Index can significantly reduce such cost when learning a dual-sparse model. In particular, we employ truncated loss function to induce a series of convex programs with superior dual sparsity, and solve each dual using Indexed Block Coordinate Descent, which makes use of Approximate Nearest Neighbor (ANN) search to select active dual variables without I/O cost on irrelevant samples. We prove that, despite the bias and weak guarantee from ANN query, the proposed algorithm has global convergence to the solution defined on entire dataset, with sublinear complexity each iteration. Experiments in both sufficient and limited memory conditions show that the proposed approach learns many times faster than other state-of-the-art solvers without sacrificing accuracy.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Experimentation

1. INTRODUCTION

Linear classification has become one of the standard approaches dealing with large-scale analysis in pattern recognition and data mining. Recent advances in training linear model has achieved complexity linear to the data size [10, 17, 8], where problem with several gigabytes of data can be solved in reasonable time. With the development of more and more efficient algorithms, the learning bottleneck has shifted from computation to the I/O between disk and memory [24]. The situation becomes especially critical when data cannot fit into memory, where repeated data access through comparatively expensive I/O could encumber the performance of any efficient algorithm. How to reduce such I/O cost becomes a focus in recent research on large-scale linear classification.

One popular approach that addresses the memory limitation is solving optimization in an online fashion, where samples are removed from memory after each parameter update. However, recent studies [24, 2] have shown that online methods (e.g. [16, 17]) need large number of iterations to obtain reasonable accuracy, and since the sample are reloaded for each single update, the training time of online solver is dominated by disk I/O. To balance the time spent on I/O and computation, Yu et al. proposed a Block Minimization framework that better utilizes data in memory by splitting data into several blocks and solving one block at a time. The algorithm yields better trade-off between I/O and computation, but still needs tens of rounds of data loading before converges to a reasonable model. In [2], Kai et al. point out that, when taking Block Minimization as Block Coordinate Descent on the dual problem, some dual variables are more important than others. Therefore, caching informative samples in memory based on gradient information can result in faster convergence. However, the caching technique only applies to samples already read into memory, so a Block Coordinate Descent with cache still needs to traverse the whole dataset several times before convergence. For a dual-sparse model, this appears to be not cost-effective since only a few informative samples are relevant to improve model, while most I/O time are spent on useless ones. This motivates us to think that if one can organize data beforehand so learning only needs to read relevant samples into memory.

This paper aims to demonstrate how a Nearest-Neighbor index can improve the I/O efficiency in large-scale learning, especially when memory is limited. In practice, this is beneficial since many state-of-the-art indexing methods for Approximate Nearest Neighbor (ANN) search (e.g. Locality-Sensitive Hashing, Metric Tree etc.) do not require repeated data access, and thus only need small memory and one pass of data loading to be built. Furthermore, an index can often be reused for models trained on the same data. Scenarios such as parameter tuning, cross-validation, multi-class classification, feature selection, and data incremental learning all require executing the training algorithm multiple times.

In [6], Dhillon et al. are the first to apply Nearest-Neighbor search on sparse optimization problem. They show using ANN search to replace brute-force selection can significantly improve the efficiency of Greedy Coordinate Descent. However, their experiments also indicate that, when minimization over single coordinate can be solved cheaply, ANN search becomes the bottleneck of learning, and a simple cyclic coordinate descent can be more efficient than greedy approach due to the saved search cost. In this paper, we address this issue by proposing Indexed Block Coordinate Descent, a Nearest-Neighbor-based Block Coordinate Descent algorithm that balances the cost of search and minimization. In our experiment that considers both I/O and CPU time, even for problem with cheap coordinate minimization step, the proposed algorithm can be orders of magnitude faster than cyclic method. In addition, Indexed Block Coordinate Descent has global convergence that does not require guarantees from ANN search. This milder condition of convergence is crucial, since most ANN search algorithms are not designed for learning, and only answer maximum normalized-inner-product query [15] or nearest-to-hyperplane query [9], that does not aim to find samples with largest gradient as required by greedy approaches in [6].

The sparsity of underlying problem is also crucial to the efficiency of Nearest-Neighbor-based approaches. Although the solution of SVM is known to be dual sparse, when data are non-separable, the number of support vectors under standard L1 (hinge) or L2-loss is actually linear to the size of data [19]. In [5], Collobert et al. propose using ramp-loss to achieve better dual-sparsity, which is also known to be more robust than L1/L2-loss in noisy setting [22][21]. Although learning ramp-loss SVM is a non-convex problem, Collobert et al. point out the computational advantage of trading convexity for sparsity. They decompose the non-convex problem into a series of convex ones using Concave-Convex Procedure (CCCP) [25], where one can learn a non-linear SVM much more efficiently due to the superior dual-sparsity.

While CCCP can also solve linear SVM with ramp-loss by integrating with state-of-the-art linear solvers such as LIBLINEAR [7], *Pegasos* [17] or SVM^{perf} [10], there is little efficiency gain like that in nonlinear case, since the complexity of linear solver has little dependency on sparsity. However, in this paper, combining the sparsity induced by ramp-loss with Nearest-Neighbor-based coordinate descent, we obtain a practical algorithm that can solve large-scale problem in sublinear time.

The paper is organized as follows. In Section 2, we propose a new relaxation method that solves not only ramp-loss, but general truncated-loss problem by a series of convex programs with superior sparsity. In Section 3, we introduce Indexed Block Coordinate Descent to solve each dual-sparse convex program, and present how this framework can be combined with state-of-the-art primal or dual solvers. Section 4 gives some implementation details including the Indexing method used in our implementation. Section 5 conduct experiments on four large-scale data sets under sufficient and limited memory conditions. In both conditions, our algorithm learns times faster than other state-of-the-art solvers without sacrificing accuracy.

2. TRUNCATED LOSS AND RELAXATION

The general truncated-loss function is defined as:

$$R_s(z) = \min(L(z), 1+s)L \tag{1}$$

where $L(z)$ can be L_1-loss $\max(1-z,0)$, L_2-loss $\max(1-z,0)^2$, or other convex loss function. When $L(z)$=$\max(1-z,0)$, (1) is also called ramp-loss. Unlike standard convex loss functions that give outlier loss that can potentially grow to infinity, (1) take sample with error more than s as outliers, and assign at most $1+s$ loss to them. The truncated loss function (1) is more like 0/1-loss as in accuracy, and thus is more robust to noise [3]. More importantly, none of the outliers under (1) will become support vectors. Thus the number of support vectors does not grow linearly with data size [5]. However, the learning problem

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{l\in D} R_s(y_l\boldsymbol{w}^T\boldsymbol{x}_l) \tag{2}$$

is non-convex. When $L(z)$=$\max(1-z,0)$, (2) can be divided into a convex part $J_{vex}(\boldsymbol{w})$ and a concave part $J_{cav}(\boldsymbol{w})$

$$\begin{aligned} J(\boldsymbol{w}) \quad &= J_{vex}(\boldsymbol{w}) + J_{cav}(\boldsymbol{w}) \\ &= \left\{\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{l\in D}\max(1-y_l\boldsymbol{w}^T\boldsymbol{x}_l,0)\right\} \\ &+ \left\{-C\sum_{l\in D}\max(-s-y_l\boldsymbol{w}^T\boldsymbol{x}_l,0)\right\} \end{aligned} \tag{3}$$

and solved with Concave-Convex Procedure (CCCP) [25] by a series of convex programs

$$\boldsymbol{w}^{t+1} = \underset{\boldsymbol{w}}{argmin}\left\{J_{vex}(\boldsymbol{w}) + \nabla J_{cav}(\boldsymbol{w}^t)^T\boldsymbol{w}\right\} \tag{4}$$

[25] shows that this algorithm strictly decrease the original objective $J(\boldsymbol{w})$, and more recently, [18] shows this algorithm globally converges to a stationary point of $J(\boldsymbol{w})$. However, the formulation (3) is specific to hinge-loss. For other loss functions, it is not obvious how to derive the convex part and concave part. Another concern is, outliers are not excluded from the formulation (4), and thus cannot be filtered during training.

Here we propose a new convex relaxation method for general truncated-loss (1) that excludes outliers from each subproblems. First, we define the sets of outliers and non-outliers as $OUT(\boldsymbol{w}) = \{l|L(y_l\boldsymbol{w}^T\boldsymbol{x}_l) > 1+s\}$ and $IN(\boldsymbol{w}) = \{l|L(y_l\boldsymbol{w}^T\boldsymbol{x}_l) \leq 1+s\}$. In each iteration, this method solves

$$\begin{aligned} \boldsymbol{w}^{t+1} = \underset{\boldsymbol{w}}{argmin} \quad &\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{IN(\boldsymbol{w}^t)} L(y_l\boldsymbol{w}^T\boldsymbol{x}_l) \\ &+ C\sum_{OUT(\boldsymbol{w}^t)} 1+s \end{aligned} \tag{5}$$

which is very intuitive: solve convex problem of the original loss $L(z)$ for only non-outliers, and fix loss of outliers to 1+s. The formulation is motivated by the observation that truncated loss $R_s(z)$ is upper-bounded both by $L(z)$ and

$1+s$, so

$$\begin{aligned}
&\frac{1}{2}\|\boldsymbol{w}^{(t)}\|^2 + C\sum_{l\in\mathcal{D}} R(y_l\boldsymbol{w}^{(t)T}\boldsymbol{x}_l)\\
=&\frac{1}{2}\|\boldsymbol{w}^{(t)}\|^2 + C\sum_{IN(\boldsymbol{w}^{(t)})} L(y_l\boldsymbol{w}^{(t)T}\boldsymbol{x}_l) + C\sum_{OUT(\boldsymbol{w}^{(t)})} 1+s\\
\geq&\frac{1}{2}\|\boldsymbol{w}^{(t+1)}\|^2 + C\sum_{IN(\boldsymbol{w}^{(t)})} L(y_l\boldsymbol{w}^{(t+1)T}\boldsymbol{x}_l) + C\sum_{OUT(\boldsymbol{w}^{(t)})} 1+s\\
\geq&\frac{1}{2}\|\boldsymbol{w}^{(t+1)}\|^2 + C\sum_{l\in\mathcal{D}} R(y_l\boldsymbol{w}^{(t+1)T}\boldsymbol{x}_l)
\end{aligned}$$

where $\mathcal{D} = IN\boldsymbol{w} + OUT\boldsymbol{w}$. In other words, (5) minimizes an upper bound of (2), which strictly decrease (2) when $IN(\boldsymbol{w}^{t+1}) \neq IN(\boldsymbol{w}^t)$ (or, equivalently, $OUT(\boldsymbol{w}^{t+1}) \neq OUT(\boldsymbol{w}^t)$) [1]. When $IN(\boldsymbol{w}^{t+1}) = IN(\boldsymbol{w}^t)$, (2) equals to (5), and thus we get a minimum of (2). Since $IN(\boldsymbol{w}^t)$ would not be the same as $IN(\boldsymbol{w}^0)...IN(\boldsymbol{w}^{t-1})$, (5) converges in finite iterations as CCCP.

For $L(z)$ being hinge-loss, we can further show that the sequence $\{\boldsymbol{w}^t\}_{t=0}^{\infty}$ produced by (5) has linear convergence rate by following theorem. The reasoning is similar to the CCCP convergence rate proof in [23].

THEOREM 2.1. *The sequence $\{\boldsymbol{w}^t\}_{t=0}^{\infty}$ produced by (5) converges to a stationary point of (2) with at least linear convergence rate.*

PROOF SKETCH. Since the objective in (5) is an upper bound for (2), a sample changing from $IN(\boldsymbol{w}^t)$ to $OUT(\boldsymbol{w}^{t+1})$ or $OUT(\boldsymbol{w}^t)$ to $IN(\boldsymbol{w}^{t+1})$ can be seen as a process of descent. Therefore, we interpret (5) as an alternating minimization (block coordinate descent) between $\boldsymbol{d} \in \mathbb{R}^m$ and $(\boldsymbol{w}, \boldsymbol{\xi}) \in (\mathbb{R}^n, \mathbb{R}^m)$ on the problem

$$\begin{aligned}
\min_{\boldsymbol{w},\boldsymbol{\xi},\boldsymbol{d}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{l\in\mathcal{D}} d_l\xi_l + (1-d_l)(1+s)\\
s.t. \quad & y_l\boldsymbol{w}^T\boldsymbol{x}_l \leq 1-\xi_l \qquad (6)\\
& \xi_l \geq 0\\
& 0 \leq d_l \leq 1, \quad l = 1..m
\end{aligned}$$

which satisfies the form of non-smooth, separable problem studied in [20]. Thus we can find an equivalent Coordinate Gradient Descent procedure introduced in [20], which produces the same sequence as that produced by block coordinate descent on (6). By theorem 1,2 and 4 of [20], the sequence converges to a stationary point of (6) with at least linear convergence rate. A detailed version of this proof is in appendix. [2] □

One advantage of (5) is it ignores outliers and solves the original problem of $L(z)$ on non-outliers. Therefore, any solver for the original loss can be utilized to solve (5). Furthermore, the relaxation has no assumption on $L(z)$. Thus it is a general approach for solving the truncated version of any convex loss function. Even in problems such as regression or clustering, one can use a truncated-version of loss function to obtain a sparser, outlier-free model for indexed learning.

[1]The decrease is strict since $R(y_l\boldsymbol{w}^T\boldsymbol{x}_l) < 1+s$ for $l \notin OUT(\boldsymbol{w})$, and $R(y_l\boldsymbol{w}^T\boldsymbol{x}_l) < L(y_l\boldsymbol{w}^T\boldsymbol{x}_l)$ for $l \notin IN(\boldsymbol{w})$.

[2] http://www.csie.ntu.edu.tw/ r00922017/kdd2013appendix

The problem (2) is non-convex, so initialization $\boldsymbol{w}^0$ of (5) will affect convergence result. A good choice is solving the original convex loss problem to get $\boldsymbol{w}_L^*$ and set $\boldsymbol{w}^0 = \boldsymbol{w}_L^*$. Then since (5) guarantees to decrease the objective function (2) with loss function more similar to 0/1 loss, we are likely to get a solution $\boldsymbol{w}^*$ with higher accuracy. However, this kind of initialization cannot gain learning efficiency since it needs to solve the original problem. A practical choice for indexed learning is to solve the convex loss problem on random samples to give an initial $\boldsymbol{w}^0$ with reasonable accuracy.

3. INDEXED LEARNING

Without outliers, we obtain convex problem (5) with superior dual-sparsity. In this section, we first introduce the framework of Indexed Block Coordinate Descent. Then in Sec.3.2, we introduce the ANN search technique to find informative samples for the learning algorithm. In Sec.3.3 and 3.4, we demonstrate how to integrate our framework with state-of-the-art primal and dual solvers.

3.1 Indexed Block Coordinate Descent

The Indexed Block Coordinate Descent algorithm 1 solves dual form of the convex program (5)

$$\begin{aligned}
\min_{\boldsymbol{\alpha}\in\mathbb{R}^N} \quad & f(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^T\bar{Q}\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha}\\
s.t. \quad & 0 \leq \alpha_l \leq U, \quad l \in IN(\boldsymbol{w}^t) \qquad (7)\\
& \alpha_l = 0, \quad l \in OUT(\boldsymbol{w}^t)
\end{aligned}$$

where $\bar{Q} = Q + D$, Q is N by N matrix with $Q_{ij} = y_iy_j\boldsymbol{x}_i^T\boldsymbol{x}_j$, D is diagonal matrix. For L1-loss SVM, $D_{ii} = 0$ and $U = C$. For L2-loss SVM, $D_{ii} = 1/(2C)$ and $U = \infty$. $\boldsymbol{e}$ is N by 1 vector $[1, 1..., 1]^T$.

In the linear case, we can maintain a relation between primal and dual variables

$$\boldsymbol{w} = \sum_{l=1}^{n} y_l\alpha_l\boldsymbol{x}_l \qquad (8)$$

so the gradient of each dual variable can be computed efficiently as

$$\nabla_l f(\boldsymbol{\alpha}) = (Q\boldsymbol{\alpha})_l - 1 + D_{ii}\alpha_l = y_l\boldsymbol{w}^T\boldsymbol{x}_l - 1 + D_{ii}\alpha_l \qquad (9)$$

The active dual variables are defined by coordinates with non-zero projected gradient:

$$\nabla_l^P f(\boldsymbol{\alpha}) = \begin{cases} \nabla_l f(\boldsymbol{\alpha}) & if \quad 0 < \alpha_l < U\\ \min(0, \nabla_l f(\boldsymbol{\alpha})) & if \quad \alpha_l = 0\\ \max(0, \nabla_l f(\boldsymbol{\alpha})) & if \quad \alpha_l = U \end{cases}$$

In Algorithm 1, we maintain a set $\mathbf{S}$ that aims to include all active dual variables, and any sample l with $\alpha_l > 0$ are put in $\mathbf{S}$. Instead of loading all samples in $IN(\boldsymbol{w}^t)$ into memory, we request active variables by *queryIndex*$(\boldsymbol{w}^{(t,k)}; \boldsymbol{w}^t, s, n)$, which searches for n samples $\notin \mathbf{S}^{(k)}$ satisfying

$$y_l\boldsymbol{w}^t\boldsymbol{x}_l \geq L^{-1}(1+s) \quad \text{AND} \quad y_l\boldsymbol{w}^{(t,k)}\boldsymbol{x}_l < 1 \qquad (10)$$

[3]It guarantees sample to be non-outlier with non-zero projected gradient, since α_l=0 for all $\alpha_l \notin \mathbf{S}$ and $\nabla_l f(\boldsymbol{\alpha}) = y_l\boldsymbol{w}^T\boldsymbol{x}_l - 1 + D_{ii}\alpha_l < 0$ for $y_l\boldsymbol{w}^T\boldsymbol{x}_l < 1$. All samples returned by *queryIndex(.)* are put into $\mathbf{S}$. Note *queryIndex(.)* does not specify which search method to use. Any search method that can find n samples satisfying (10), when there exist,

Algorithm 1 Indexed Block Coordinate Descent

```
**Input:** $\boldsymbol{w}^{(t,0)} = \boldsymbol{w}^t$, $\mathbf{S}^{(0)} = \mathbf{S}^t \setminus OUT(\boldsymbol{w}^t)$
**Output:** $\boldsymbol{w}^{t+1} = \boldsymbol{w}^{(t,k)}$, $\mathbf{S}^{t+1} = \mathbf{S}^{(k)}$
**repeat**
    $(\mathbf{N}, n_e) \leftarrow$ queryIndex( $\boldsymbol{w}^{(k)}$; $\boldsymbol{w}^t$, $s$, $n$ )
    $\mathbf{B} \leftarrow [\ \mathbf{N},\ \mathbf{S}^{(k)}[\ r : r + n_e - |\mathbf{N}|\ ]\ ]$
    Solve block minimization problem (11) over $\mathbf{B}$.
    $\mathbf{S}^{(k+1)} \leftarrow [\ \mathbf{S}^{(k)},\ \mathbf{N}\ ]$
    $k \leftarrow k + 1$; $r \leftarrow (r + n_e + 1) \bmod |\mathbf{S}^{(k)}|$
**until** problem (13) defined on $\mathbf{S}^{(k)}$ reach $\epsilon_S$ and $|\mathbf{N}| < n_\epsilon$
```

yields convergence. However, search method with more efficiency and less bias can result in faster convergence. We will introduce an ANN search method in Sec.3.2 that tries to find $\boldsymbol{w}^T \boldsymbol{x}_l$ close 0, in which case, the query result tends to have large gradient no matter labels y_l are 1 or -1.

In algorithm 1, we denote fresh samples returned by *queryIndex(.)* as $\mathbf{N}$, where n_e is the number of traversed instances for finding $|\mathbf{N}| \le n$ samples satisfying (10). Though samples from $\mathbf{N}$ have larger gradient, they come from cost of search. To balance cost between search and optimization, we compose a block $\mathbf{B}$ with $\mathbf{N}$ and the other $min(n_e - |\mathbf{N}|, |S^{(k)}|, M)$ samples from reader r that cyclically traverses $\mathbf{S}^{(k)}$, where M is the sample size limited by memory. Then we solve the block minimization problem defined by $\mathbf{B}$ with $\bar{\mathbf{B}} = \mathcal{D} \setminus \mathbf{B}$

$$\begin{aligned} \min_{\alpha_B} \quad & f(\boldsymbol{\alpha_B}; \boldsymbol{\alpha}^{(t,k)}_{\bar{\mathbf{B}}}) \\ s.t. \quad & 0 \le \alpha_l \le U, \qquad l \in \mathbf{B} \\ & \alpha_l = \alpha_l^{(t,k)}, \qquad l \in \bar{\mathbf{B}} \end{aligned} \tag{11}$$

Then the cost for each iteration is

$$T_{search}(n_e) + T_{opt}(|\mathbf{B}|), \quad n_e = \frac{n}{prec[n]} \ge |\mathbf{B}| \tag{12}$$

where $T_{search}(n_e)$ is the time for search method to traverse n_e examples. $T_{opt}(|\mathbf{B}|)$ is the time for solving (11). For a ANN search method with precision $prec[n] = n/n_e$, n_e and $|\mathbf{B}|$ are sublinear to the data size. The hope is that neither $T_{search}(n_e)$ nor $T_{opt}(|\mathbf{B}|)$ becomes the bottleneck. Practically, both T_{search} and T_{opt} may involve disk I/O when samples used are not cached in memory. We will discuss details of memory management in Sec.4.2.

The Indexed Block Coordinate Descent algorithm 1 can be viewed as a reverse process of the well-known Shrinking strategy. In Shrinking strategy, the problem size is shrunk in each iteration by removing inactive variables that tends to be unchanged. However, since the strategy often needs to traverse data many times before shrinking problem to a smaller size, it is not suitable to be applied when data cannot fit into memory. On the other hand, the Indexed Block Coordinate Descent algorithm tries to maintain a set of most active dual variables $\mathbf{S}$, and increases $\mathbf{S}$ until it contains all variables with non-zero projected gradient. When combined with Nearest-Neighbor index, this strategy can avoid much I/O, especially when memory is limited.

Although the block minimization (11) decreases objective in (7) each iteration, it does not imply global convergence. In the following, we give the convergence theorem for algorithm 1, beginning with a lemma.

Lemma 3.1. *Let α^* be the optimal solution of (7). Let $\mathbf{V} = \{l | \alpha_l^* > 0\} \subseteq \mathbf{S} \subseteq IN(\boldsymbol{w}^t)$ and $\bar{\mathbf{S}} = \mathcal{D} \setminus \mathbf{S}$. Then the optimal solution of following problem*

$$\begin{aligned} \underset{\alpha}{argmin} \quad & f(\boldsymbol{\alpha}) \\ s.t. \quad & 0 \le \alpha_l \le U, \quad l \in \mathbf{S} \\ & \alpha_l = 0, \quad l \in \bar{\mathbf{S}} \end{aligned} \tag{13}$$

is also the optimal solution of (7).

The proof for above lemma is simple and thus omitted due to space limitation.[4] The next theorem gives the convergence of algorithm 1.

Theorem 3.2. *The sequence $\{\boldsymbol{\alpha}^{(t,j)}\}_{j=k}^{\infty}$ produced by algorithm 1 linearly converges to the problem (13) with $\mathbf{S} = \mathbf{S}^{(k)}$, and the sequence $\mathbf{S}^{(1)}$, $\mathbf{S}^{(2)}$, ...$\mathbf{S}^{(k)}$ converges to $\mathbf{S}^*$ in no more than $T_n + T_\epsilon$ iterations, where $T_n \le \frac{|\mathbf{S}^*|}{n}$, $T_\epsilon \le \frac{|\mathbf{S}^*| - nT_n}{n_\epsilon}$, and there are at most $n_\epsilon - 1$ active dual variables not in $\mathbf{S}^*$.*

Proof. As shown in [8], the formulation (13) satisfies the form of convex, smooth problem studied in [13]. Since each iteration after k, all variables in $\mathbf{S}^{(k)}$ are cyclically traversed, so the sequence $\{\boldsymbol{\alpha}^{(t,j)}\}_{j=k}^{\infty}$ satisfies *Gauss-Seidal* update rule of Block Coordinate Descent. By the extended version of theorem 2.1 in [13], the sequence $\{\boldsymbol{\alpha}^{(t,j)}\}_{j=k}^{\infty}$ has linear convergence rate to the optimal solution of (13) defined by $\mathbf{S} = \mathbf{S}^{(k)}$. And since $|\mathbf{S}^{(k)}|$ monotonically increases by $|\mathbf{N}|$ each iteration, we have

$$|\mathbf{S}^*| = nT_n + \sum_{i=1}^{T_\epsilon} n_i \ge nT_n + n_\epsilon T_\epsilon \tag{14}$$

where $n_i = |\mathbf{N}|$ for iterations that have $|\mathbf{N}| < n$, and $n_i \ge n_\epsilon$ for $i = 1..T_\epsilon$ since the algorithm stops when $|\mathbf{N}| < n_\epsilon$. By (14), we have $T_n \le \frac{|\mathbf{S}^*|}{n}$, $T_\epsilon \le \frac{|\mathbf{S}^*| - nT_n}{n_\epsilon}$. $\square$

The size of final set of active variables $|\mathbf{S}^*|$ is at most $|\mathcal{D}|$. However, we observed $|\mathbf{S}^*| \ll |\mathcal{D}|$ for a dual-sparse problem. Setting $n_\epsilon = 1$, Theorem 3.2 proves our algorithm converges to the optimal solution of (7). However, in practice, including n_ϵ more active dual variables only decrease the objective (2) by at most $n_\epsilon C(1+s)$, which approximately increase training accuracy only by $\frac{n_\epsilon}{|\mathcal{D}|}$. To get within ϵ tolerance of the best training accuracy Acc, we can set $n_\epsilon = \epsilon(1 - Acc)|\mathcal{D}|$. For example, to reach $\epsilon = 1\%$ tolerance for a model with 99% training accuracy on a data with $1,000,000$ samples, n_ϵ can be set as 100. A more practical stopping condition uses sampling to replace the stopping condition in algorithm 1, in which the algorithm stops when $|\mathbf{N}|/n_e < n_\epsilon/|\mathcal{D}|$. The sampling-based stopping condition is practical since it avoids algorithm 1 from traversing the whole index through expensive I/O. In the next section, we will introduce the ANN search method for the *queryIndex(.)* (10), and in Sec.3.3 and Sec.3.4, we describe the dual and primal optimization methods we use for the block minimization (11).

[3]Here we assume $L(z)$ is monotonically decreasing for $L(z) > 0$ so $L^{-1}(1+s)$ exists.

[4]http://www.csie.ntu.edu.tw/ r00922017/kdd2013appendix

3.2 Informative Sample as Nearest-Neighbor

Here we consider how informative samples can be obtained via ANN search. The *queryIndex(.)* function searches for samples satisfying $\left\{l|L^{-1}(1+s) \leq y_l \boldsymbol{w}^T \boldsymbol{x}_l \leq 1\right\}$, that is, samples near to the decision boundary. In [9], a Locality Sensitive Hashing (LSH) method EH-Hash was proposed to handle the nearest-to-hyperplane query occurs in Active Learning, where their goal is to minimize the search time for obtaining a new informative sample in active learning. Here, we apply a similar technique that transforms the nearest-to-hyperplane problem into standard ANN query. The technique costs cheaper than EH-Hash when combined with the Tree-based index in our implementation.

Given a query vector $\boldsymbol{q}$, the nearest neighbor is defined as the vector $\boldsymbol{x}_l$ most similar to $\boldsymbol{q}$, that is,

$$\underset{l}{argmax} \quad \hat{\boldsymbol{q}}^T \hat{\boldsymbol{x}}_l$$

where $\hat{v}$ means normalized v. However, sample closest to a hyperplane of normal vector $\boldsymbol{w}$ is defined as

$$\underset{l}{argmin} \quad |\hat{\boldsymbol{w}}^T \boldsymbol{x}_l|$$

We connect these two problems using kernel of degree-2 polynomial feature expansion $\boldsymbol{V(x)}$=[x_1^2, $\sqrt{2}x_1x_2$, ..., $\sqrt{2}x_1x_d$, x_2^2, $\sqrt{2}x_2x_3$, ..., x_d^2] such that the nearest-to-hyperplane problem in original space can be reduced to ANN problem in embedded space. We first considers the case when data are normalized:

$$\begin{aligned}\underset{l}{argmin} \quad |\hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_l| &= \underset{l}{argmin} \quad (\hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_l)^2 \\ &= \underset{l}{argmin} \quad \boldsymbol{V(\hat{w})}^T \boldsymbol{V(\hat{x}_l)} \\ &= \underset{l}{argmax} \quad (-\boldsymbol{V(\hat{w})})^T \boldsymbol{V(\hat{x}_l)}\end{aligned}$$

Where $-\boldsymbol{V(\hat{w})}$ is the transformed nearest-neighbor query, and (-$\boldsymbol{V(\hat{w})})^T\boldsymbol{V(\hat{x}_l)}$ can be computed efficiently by $-(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_l)^2$, which is not like the case in EH-Hash, where quadratic cost was required to generate Gaussian-distributed hashing function in the embedded space. The above transformation guarantees $-(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_l)^2$ to be a normalized inner product in some Hilbert space, and thus, can use indexing structure designed for nearest-neighbor problem.

For unnormalized sample $\boldsymbol{x}_l$ with length $\|\boldsymbol{x}_l\|_2$. Let $R = min_l\|\boldsymbol{x}_l\|_2$, since query (10) defined by $queryIndex(.)$ is a range query, we can search for a larger range

$$\left\{l|L^{-1}(1+s)/R \leq y_l\hat{w}^T\hat{\boldsymbol{x}}_l \leq 1/R\right\}$$

that includes the target samples required by (10), while the search algorithm for unnormalized data will be the same as normalized ones.

3.3 Solving Dual for each Block

In this paper, for both L_1-loss and L_2-loss cases, we use coordinate descent solver proposed by [8] to solve the dual of block minimization problem (11)

$$\begin{aligned}\underset{\boldsymbol{\alpha}_{\mathbf{B}}}{min} \quad & \frac{1}{2}\boldsymbol{\alpha}^T\bar{Q}\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha} \\ s.t. \quad & 0 \leq \alpha_l \leq U, \quad l \in \mathbf{B} \\ & \alpha_l = \alpha_l^{(t,k)}, \quad l \in \bar{\mathbf{B}}\end{aligned} \tag{15}$$

In each inner iterations, the algorithm traverses all variables in block $\mathbf{B}$ in a random permuted order, and solve each coordinate minimization problem via the closed form solution

$$\alpha_l^* = \min\left(\max\left(\alpha_l - \frac{\nabla_l f(\alpha)}{\bar{Q}_{ll}}, 0\right), C\right)$$

where $\nabla_l f(\alpha)$ is computed via (9), and maintains primal vector $\boldsymbol{w}$ by

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + (\alpha_l^* - \alpha_l)y_l\boldsymbol{x}_l$$

As shown in [8], the algorithm converges linearly to the optimal solution of (11). However, for some ill-conditioned problems, linear convergence can be very slow [7]. For loss with second-order information, a primal solver like [11] with super-linear convergence is more suitable. The next section introduce the primal problem derived from block (11).

3.4 Solving Primal for each Block

To apply primal solver on problem (11), we derive the Lagrangian dual of (11) with $\alpha_l \in \bar{\mathbf{B}}$ fixed at $\alpha_l^{(t,k)}$, which results in the primal formulation of (11) [5]

$$\underset{\boldsymbol{w}}{min} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{l\in\mathbf{B}} L(y_l\boldsymbol{w}^T\boldsymbol{x}_l) - \boldsymbol{w}^T\mathbf{v}_{\bar{\mathbf{B}}} \tag{16}$$

where

$$\mathbf{v}_{\bar{\mathbf{B}}} = \sum_{l\in\bar{\mathbf{B}}} \alpha_l^{(t,k)} y_l \boldsymbol{x}_l$$

. For L_2-loss $L(z) = max(1-z,0)^2$, we have *generalized Hessian* and *gradient* of (16) as

$$\begin{aligned}H(w) &= \mathcal{I} + 2CX_{I,:}^T X_{I,:}, \\ g(w) &= \boldsymbol{w} + 2CX_{I,:}^T(X_{I,:}\boldsymbol{w} - \mathbf{y_I}) - \mathbf{v}_{\bar{\mathbf{B}}}\end{aligned} \tag{17}$$

where $X = \left[\boldsymbol{x}_1, ...\boldsymbol{x}_{|\mathcal{D}|}\right]^T$, $\mathbf{y} = \left[y_1, ..., y_{|\mathcal{D}|}\right]^T$, $\mathcal{I}$ is identity matrix and $I = \left\{l|1 - y_l\boldsymbol{w}^T\boldsymbol{x}_l > 0\right\}$. With (17), one can apply any primal solver that uses second-order information to get faster convergence. In our implementation, we use the Trust-Region Quasi-Newton Method proposed in [11], which is also included in the LIBLINEAR package [7].

When setting $\mathbf{B} = \mathbf{S}^{(k)}$, we have $\mathbf{v}_{\bar{\mathbf{B}}} = 0$ since $\alpha_l^{(t,k)} = 0$ for $l \in \bar{\mathbf{S}}^{(k)}$. In this case, (16) becomes a L_2-loss problem defined on $\mathbf{S}^{(k)}$. It means, each time we solve a L_2-loss problem on a subset of data $\mathbf{S}^{(k)}$, we get descent in terms of dual objective, and by Theorem 3.2, after $T_n + T_\epsilon$ iterations, we obtain all samples of non-zero loss and get a global optimum of (7).

4. IMPLEMENTATION ISSUES

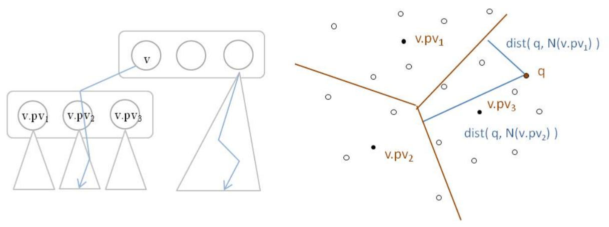
In this section, we address some details of implementing an indexed optimization algorithm. First, we discuss challenges of building index for learning problem, as opposed to other problem like retrieval. Then we discuss the Caching, Shrinking strategy for learning with less memory and time.

4.1 Indexing for Learning

There are two practical challenges of building index for selecting informative training samples. First, the index for ANN search is biased towards the reference points (or hash functions) selected. While such bias is acceptable for applications like multimedia retrieval, training on biased samples

[5]http://www.csie.ntu.edu.tw/ r00922017/kdd2013appendix

Figure 1: Metric Tree with $k = 3$.



will seriously slow down the convergence of algorithm (1). Second, to be reused for different models, the index should be built on disk. However, not all indexing methods are cost-effective to be built on disk. For example, Locality-sensitive hashing (LSH), one of the most popular methods recent years, requires a sample to be stored in different hash tables, which can significantly increase the storage cost [14].

Here we propose Metric Forest that addresses these issues. Unlike LSH, a tree-structured index has size comparable to the original data. In [12], a tree-based indexing, called Metric Tree, is modified to solve ANN problem, and is shown to be more efficient than LSH. A Metric Tree organizes points as a binary tree, in which each subtree $N(\boldsymbol{v})$ rooted on a node v partitioned into points closer to $\boldsymbol{v.lpv}$ and those closer to $\boldsymbol{v.rpv}$, where $\boldsymbol{v.lpv}$ and $\boldsymbol{v.rpv}$ are pivot points selected from data, and the metric is defined by angle distance in the embedded space: $d(\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2) = \cos^{-1}(\boldsymbol{V(\hat{x}_1)}^T \boldsymbol{V(\hat{x}_2)}) = \cos^{-1}((\hat{\boldsymbol{x}}_1{}^T \hat{\boldsymbol{x}}_2)^2)$. Here we propose Metric Forest, which shares the basic idea of Metric tree, but is different in three ways:

Bias Reduction.

Since reference points closer to the root of a Metric Tree are used more frequently, query results are often biased toward those points. [12] propose a variant called Spill-Tree to allow overlap between different partitions, thus alleviate the bias problem. However, the overlap between partitions significantly increases storage, and under limited memory, even building a tree is time-consuming. In our design, data are split into R random blocks which can be put into memory, and a Metric Tree is built for each block. When performing search, all trees are traversed in an order determined randomly. Such design warrants the query results not biased to a small number of reference points.

Partitioning.

While binary tree works well in memory, it is inefficient on disk because the seeking time on disk is more expensive. Here we use, instead, K-way partitioning such that the tree depth is smaller and each tree node contains more points. For each subtree $N(\boldsymbol{v})$ rooted on node $\boldsymbol{v}$, points are partitioned into $N(\boldsymbol{v.pv_1})...N(\boldsymbol{v.pv_K})$ s.t. $\boldsymbol{v.pv_k}$ is nearest to the points in $N(\boldsymbol{v.pv_k})$ among $\boldsymbol{v.pv_1}...\boldsymbol{v.pv_K}$.

Searching.

In [12], Defeatist Search is used to obtain efficient approximate search, in which only the nearest partition $N(\boldsymbol{v.pv_k})$ to a query point is visited. However, in our algorithm, we need to increase, or decrease, search range according to the current margin $1/\|w\|_2$. Therefore, we exploit Best-Bin-First search strategy, as proposed in [1], in which each partition $N(\boldsymbol{v.pv_k})$ is put in a priority queue, and ranked by their distance to the query, where the distance is measured by their nearest boundary to the query point, as shown in Figure 1. Let $\boldsymbol{v.pv_q}$ be the closest pivot point to the query, the nearest boundary of partition $N(\boldsymbol{v.pv_k})$ is defined by the hyperplane passing $(\boldsymbol{v.pv_k} + \boldsymbol{v.pv_q})/2$ with normal vector $\boldsymbol{v.pv_k} - \boldsymbol{v.pv_q}$.

4.2 Memory Management

In our implementation, we use Least-Recently-Used (LRU) cache for frequently-used tree nodes to minimize I/O between memory and index. This is effective since, in *queryIndex(.)*, we only traverse branches of Metric Tree near decision boundary. As model changes over time, samples away from the boundary will not be traversed again, and thus, are removed when cache is full. Since we keep samples with larger projected gradient in memory, I/O of unnecessary data is minimized. When sample are added into active set $\mathbf{S}$, we moved the memory quota from cache to $\mathbf{S}$, until LRU cache has memory quota less than a threshold m_{LRU}, when we will write some samples in $\mathbf{S}$ into disk in a sliding window fashion, and read those samples back when they are traversed.

4.3 Shrinking on Index

In search of *queryIndex(.)*, samples not satisfying (9) for $\boldsymbol{w}^{(t,k)}$ are likely not satisfying (9) for $\boldsymbol{w}^{(t,k+1)}$, $\boldsymbol{w}^{(t,k+2)}$,...either. This is like the case in most SVM solver, where many bounded variables, that is, $\alpha_l = 0$ or $\alpha_l = C$, tends to be unchanged if their gradients are large to the bounded direction. Here we apply a similar technique to shrink variables that are not likely to satisfy (10) in later iterations, which includes variables with (i) $y_l \boldsymbol{w}^{(t,k)T} \boldsymbol{x}_l > 1 + T$, (ii) $y_l \boldsymbol{w}^{(t)T} \boldsymbol{x}_l < L^{-1}(1 + s)$, (iii) $l \in S^{(k)}$, where T is a threshold. The shrunk variables would not be traversed in a Metric Tree, and a tree node is not traversed if all of its descendants are shrunk. When the shrunken problem converges, we recover all of shrunken variables and solve the non-shrunk problem to check convergence. This is repeated until the original non-shrunk problem is solved.

5. EXPERIMENT

In this section, we conduct experiments that compare our algorithm (Index-L1-Dual, Index-L2-Dual, and Index-L2-Primal) with state-of-the-art linear SVM solvers LIBLINEAR (L1-Dual, L2-Dual, and L2-Primal), online *Pegasos* (Online-L1 and Online-L2), and truncated-loss batch solver (Trunc-L1-Dual, Trunc-L2-Dual, and Trunc-L2-Primal) that uses the same truncated-loss function (1) as our method, but employs LIBLINEAR as inner procedure for each convex relaxation (5). There are, of course, other state-of-the-art solvers. However, as our contribution is an indexed learning framework, a comparison between methods with/without our technique is more essential than that between different solvers. In limited memory condition, we compare Indexed Block Coordinate Descent with online *Pegasos* and LIBLINEAR-CDBLOCK (Block-L1-Dual, Block-L2-Dual), a limited-memory version of LIBLINEAR proposed in [24], and refined in [2]. The initialization for both truncated-loss solvers uses 10,000 random samples solved by the corresponding convex loss solver in LIBLINEAR. In our experiments, both I/O and Initialization are included into training time.

Table 1: Statistics of Data.

Dataset	#samples	#features	Storage (KB)
Covtype	581,012	54	69,516
Kddcup1999	4,898,431	126	725,180
PAMAP	3,850,505	104	2,198,880
Mnist8m	8,100,000	784	19,042,640

Table 2: Statistics of Index.

Dataset	Storage (KB)	Tree Size	Tree Width	Build Time (s)
Covtype	446,444	2,000	10	11
Kddcup1999	1,476,580	100,000	100	163
PAMAP	4,554,208	100,000	10	301
Mnist8m	20,704,784	10,000	10	1,539

Our experiments conducted on 4 large-scale public datasets[6] of increasing size: Covtype, Kddcup1999, PAMAP and Mnist8m. Their statistics are summarized in Table 1.

Table 2 shows the statistics of index built. The construction time and storage size are generally linear to the data size. In our experiment, we found that the parameter *tree size* (e.g. #samples / tree) and *tree width* (e.g. #pivot-points) do not affect result significantly. However, *tree size* should be large enough s.t. each tree contains useful samples, and small enough s.t. each tree can be built in memory. *Tree width* should be large enough s.t. the average depth=$log_{width}(Size)$ is reasonably small (3 to 5 in our experiment). Since KDDCUP1999 and PAMAP have more irrelevant samples than other datasets, their *tree size* are set to be larger than that of others.

We scale features in all dataset to between 0 and 1. For Covtype and KDDCUP1999, we randomly selected 1/3 of samples for testing and the remaining 2/3 samples for training. For PAMAP and Mnist8m, to test on limited memory, we only used 1/10 for testing and left 9/10 for training. All of these data have multi-classes, and thus the index built can be reused for different models. Due to space limitation, we only show figures for one of those 1-against-all models for each dataset. For Covtype, the given result is on the model of *class 2* against the other 6 classes, which follows the choice in [4]. For KDDCUP1999, we show result of the class *normal* against all the other 22 *abnormal* classes. For PAMAP, we classify action "sitting" from the other 23 actions. For mnist8m, we show result of class *digit-1* against all the other 9 digits. Throughout the experiments, we set parameters $n = 1000$ for Indexed Block Coordinate Descent, $s = 1$ for truncated-loss, and $c = 1$ for SVM model.

Figure 2, 3, 4 show the testing error of L_1-*Dual*, L_2-*Dual*, and L_2-*Primal* solvers under sufficient memory, where Indexed Block Coordinate Descent saves much I/O time by selecting only relevant samples into memory. Unlike online solver, it converges to much more accurate solution as that produced by truncated-loss batch solver. Though truncated-loss learning problem is non-convex, where different solvers may converges to different solutions, the indexed solver achieves similar accuracy as the truncated-loss

[6]Covtype, Kddcup1999, PAMAP can be found at UCI Machine Learning Repository, while Mnist8m can be found in LIBSVM Datasets.

Figure 2: L1-loss Dual Solvers

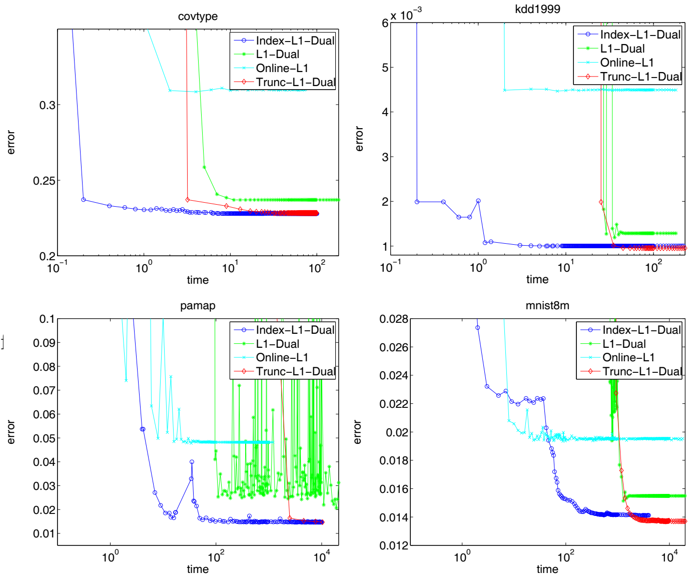


Figure 3: L2-loss Dual Solvers

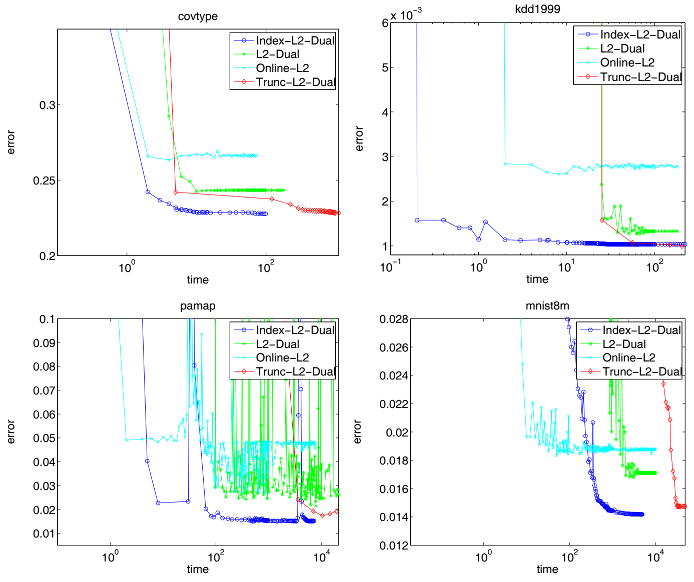
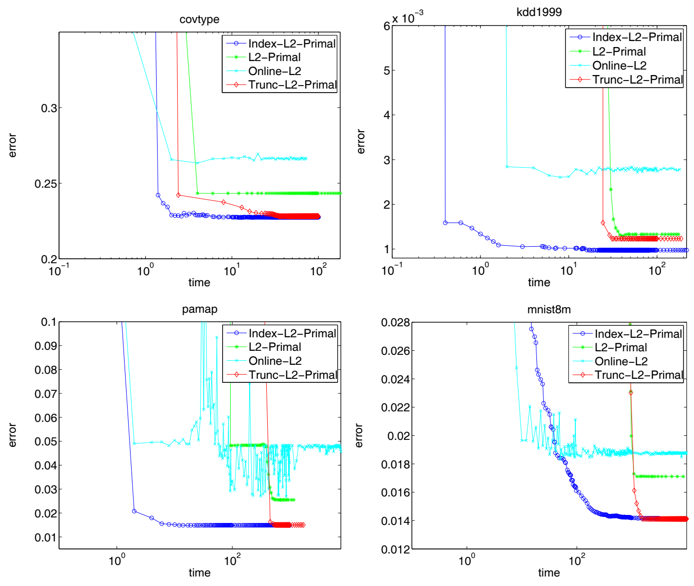


Figure 4: L2-loss Primal Solvers



batch solver in most of cases, but indexed solver was orders of magnitude faster than the batch one. In Figure 6, we shows result of our algorithm for a spectrum of parameter $c = 0.01, 0.1, 10, 100$ on KDDCUP1999 dataset.

Figure (5) shows the limited-memory experiments conducted on PAMAP and Mnist8m. We compare Indexed Block Coordinate Descent with Online Pegasos and LIBLINEAR-CDBLOCK, where data size is 10 times larger than memory space, under which the batch version of LIBLINEAR and truncated-loss solvers suffer from severe swaps and can hardly progress. For Mnist8m, we limit memory size to 2GB, and uses 20 blocks with 1GB cache for LIBLINEAR-CDBLOCK. For PAMAP, we limit memory size to 400MB, and uses 10 blocks with 200MB cache for LIBLINEAR-CDBLOCK. In Figure (5), the Indexed Block Coordinate Descent has almost the same performance as in sufficient-memory condition, where it achieves higher accuracy by selecting informative samples under truncated-loss into memory. Though datasize was 10 times larger than the memory size, the indexed solver is not affected much since the memory is still large enough for maintaining only relevant samples.

Finally, in Figure 7, we show the number of support vectors, and corresponding testing error of our algorithm and LIBLINEAR for a spectrum of s, which shows the significant effect of truncated-loss on increasing the dual-sparsity of SVM. When $s = 1$, it means outlier should have error more than the current margin. This choice yields great performance for most data. However, in Mnist8m dataset, $s = 2$ is a better choice, and for PAMAP dataset, $s = 0.5$ was better.

6. DISCUSSION AND CONCLUSION

In many applications, large-scale data contain only some relevant samples that can effectively improve the accuracy of model. While random sampling, or online learning can overlook those rare but crucial samples, batch learners generally cost too much memory and I/O time. In this paper, we

Figure 5: Limited-Memory Solvers. For Mnist8m, we limit memory size to 2GB, and uses 20 blocks with 1GB cache for LIBLINEAR-CDBLOCK. For PAMAP, we limit memory size to 400MB, and uses 10 blocks with 200MB cache for LIBLINEAR-CDBLOCK.

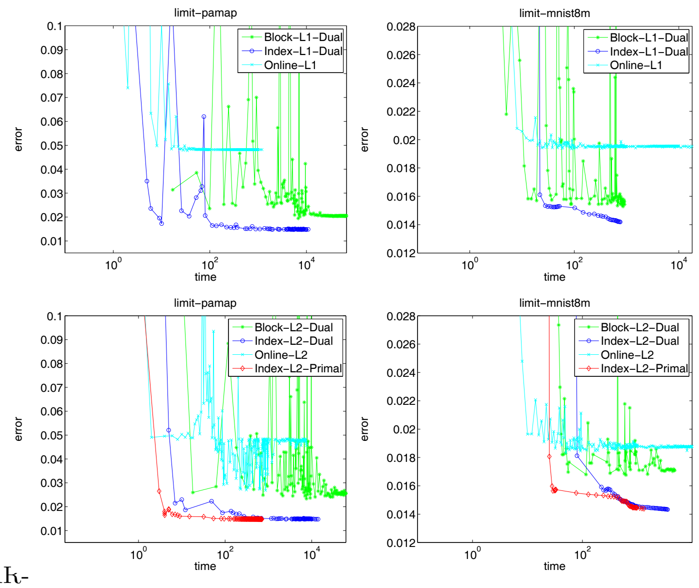


Figure 6: L1-Dual Solvers on KDDCUP1999 for c = 0.01, 0.1, 10, 100.

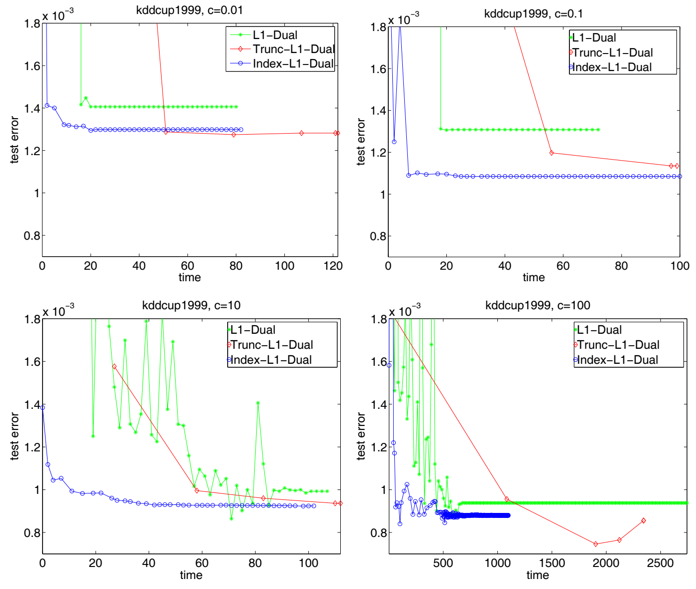
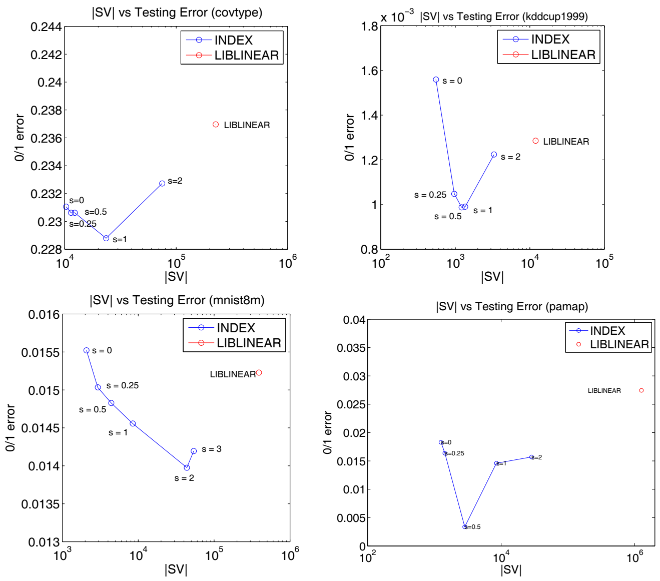


Figure 7: Testing Error vs. #SV for a spectrum of s. (L1-Dual Solver)



propose Indexed Block Coordinate Descent algorithm that makes use of Approximate Nearest Neighbor (ANN) search to select active dual variables without I/O cost on irrelevant samples. Though building index takes time linear to the data size, in practice, this is beneficial since people often learn several models from the same data. Scenarios such as parameter tuning, model selection, cross-validation, multi-class classification, feature selection, and data incremental learning all require multiple passes of training. In the case of limited memory, our approach can save much I/O cost since building index do not require much memory and only requires a pass of data reading. This indexed learning approach can be potentially apply to a general class of large-scale learning problem, through defining new truncated-loss function for convex-loss problems in classification, regression or clustering.

7. ACKNOWLEDGEMENT

This work is supported by National Science Council, National Taiwan University and Intel Corporation under Grants NSC 101-2911-I-002-001, NSC 101-2628-E-002-028-MY2 and NTU 102R7501.

8. REFERENCES

[1] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, 1997.

[2] K.-W. Chang and D. Roth. Selective block minimization for faster convergence of limited memory large-scale linear models. In *SIGKDD*. ACM, 2011.

[3] O. Chapelle, C. B. Do, Q. V. Le, A. J. Smola, and C. H. Teo. Tighter bounds for structured estimation. In *NIPS*, 2008.

[4] R. Collobert, S. Bengio, and Y. Bengio. A parallel mixture of SVMs for very large scale problems. *Neural Computation*, 14, 2002.

[5] R. Collobert, F. Sinz, J. Weston, and L. Bottou. Trading convexity for scalability. In *ICML*, 2006.

[6] I. S. Dhillon, P. D. Ravikumar, and A. Tewari. Nearest neighbor based greedy coordinate descent. In *NIPS*, 2011.

[7] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *JMLR*, 9, 2008.

[8] 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.

[9] P. Jain, S. Vijayanarasimhan, and K. Grauman. Hashing hyperplane queries to near points with applications to large-scale active learning. In *NIPS*, 2010.

[10] T. Joachims. Training linear SVMs in linear time. In *SIGKDD*, 2006.

[11] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region newton method for logistic regression, 2008.

[12] T. Liu, A. W. Moore, A. G. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.

[13] Z.-Q. Luo and P. Tseng. On the convergence of coordinate descent method for convex differentiable minimization. *Optim. Theory*, 72, 1992.

[14] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In *ICVLDB*, 2007.

[15] P. Ram and A. G. Gray. Maximum inner-product search using cone trees. In *KDD*, 2012.

[16] S. Shalev-Shwartz, K. Crammer, O. Dekel, and Y. Singer. Online passive-aggressive algorithms. In *NIPS*, 2003.

[17] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated sub-gradient SOlver for SVM. In *ICML*, 2007.

[18] B. K. Sriperumbudur and G. R. G. Lanckriet. On the convergence of the concave-convex procedure. In *NIPS*, 2009.

[19] I. Steinwart. Sparseness of support vector machines. *JMLR*, 4, 2003.

[20] P. Tseng and S. Yun. A coordinate gradient descent method for nonsmooth separable minimization. *Math. Program*, 117, 2009.

[21] L. Wang, H. Jia, and J. Li. Letters: Training robust support vector machine with smooth ramp loss in the primal space. *Neurocomput.*, 71, 2008.

[22] Z. Wang and S. Vucetic. Fast online training of ramp loss support vector machines. In *ICDM*, 2009.

[23] I. E. Yen, N. Peng., P. Wang, and S. Lin. On convergence rate of concave-convex procedure. In *NIPS*, 2012.

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

[25] A. L. Yuille and A. Rangarajan. The concave-convex procedure. *Neural Computation*, 15, 2002.