Efficient Distributed Linear Classification Algorithms via the Alternating Direction Method of Multipliers

Caoxie Zhang
Department of EECS
University of Michigan
Ann Arbor, MI 48109, USA
caoxiezh@umich.edu

Honglak Lee
Department of EECS
University of Michigan
Ann Arbor, MI 48109, USA
honglak@eecs.umich.edu

Kang G. Shin
Department of EECS
University of Michigan
Ann Arbor, MI 48109, USA
kgshin@eecs.umich.edu

Abstract

Linear classification has demonstrated success in many areas of applications. Modern algorithms for linear classification can train reasonably good models while going through the data in only tens of rounds. However, large data often does not fit in the memory of a single machine, which makes the bottleneck in large-scale learning the disk I/O, not the CPU. Following this observation, Yu *et al.* (2010) made significant progress in reducing disk usage, and their algorithms now outperform LIBLINEAR. In this paper, rather than optimizing algorithms on a single machine, we propose and implement distributed algorithms that achieve parallel disk loading and access the disk only once. Our large-scale learning algorithms are based on the framework of alternating direction methods of multipliers. The framework derives a subproblem that remains to be solved efficiently for which we propose using dual coordinate descent and trust region Newton method. Our experimental evaluations on large datasets demonstrate that the proposed algorithms achieve significant speedup over the classifier proposed by Yu *et al.* running on a single machine. Our algorithms are faster than existing distributed solvers, such as Zinkevich *et al.* (2010)'s parallel stochastic gradient descent and Vowpal Wabbit.

1 Introduction

Large-scale linear classification has proven successful in many areas, such as machine learning, data mining, computer vision, and security. With the burgeoning of social media, which provides an unprecedented amount of user-provided supervised information, there will likely be more extreme-scale data requiring classification. Training on these large data can be done on a single machine or a distributed system. Many algorithms for a single machine, such as LIBLINEAR [6] and PEGASOS [16], have been developed and extensively used in both academia and industry. These algorithms can sequentially train on the entire data in a few tens of rounds to obtain a good model and utilize the sparsity of the data. However, training with large-scale data on a single machine becomes slow because of the disk I/O, not the CPU. Since the disk bandwidth is 2-3 orders of magnitude slower than memory bandwidth and CPU speed, much of the training phase is spent on loading data rather than processing. This scenario worsens when the data cannot fit in main memory, causing severe disk swaps. Yu *et al.* [19] addressed this problem by reading blocks of data and processing them in batches. Their algorithm reduced the disk I/O time via pre-fetching and achieved a significant speedup over the original LIBLINEAR, which is one of the state-of-art libraries for linear classification [6]. However, since all single-machine algorithms go through the entire data many times, they have to load the same data from the disk *multiple times* when the data cannot fit in memory. If the data requires more than tens of gigabytes of storage (greater than the typical RAM capacity), those algorithms are still inefficient since the disk needs to be accessed more than once and the training takes hours to complete.

On the other hand, distributed systems composed of commercial servers (nodes) are becoming more prevalent both in research labs and medium-scale companies. A large Internet company may have thousands of nodes. Algorithms running on distributed systems can now load the data in parallel to the distributed memory and train without using disk any more, thereby significantly reducing the cumbersome disk loading time. One important challenge in the field is to design new algorithms in the parallel setting, since the current linear classification algorithms are inherently sequential.

In this paper, we propose distributed algorithms for linear classification. The system architecture is illustrated in Fig. 1. We use the alternating direction method of multipliers (ADMM) as the framework for solving distributed convex optimization; see Boyd *et al.* [2] for a review of ADMM. This framework introduces additional variables to regularize the difference

Appearing in Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS) 2012, La Palma, Canary Islands. Volume XX of JMLR: W&CP XX.

Copyright 2012 by the authors.

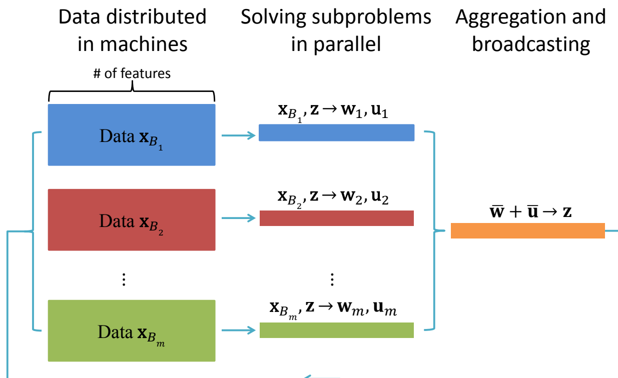

Figure 1: An illustration of the distributed linear classification algorithm. The data is split by instances across machines. Each machine loads the local data in parallel and keeps it in the main memory. Then each machine uses efficient algorithms (e.g., Algorithm 1 in Sec. 3.1) to solve a subproblem with the input being the local data and a shared vector $\mathbf{z}$ to generate the output of a weight vector and an auxiliary vector $\mathbf{u}$. After solving the subproblems, all machines aggregate these two vectors using averaging to generate $\mathbf{z}$ and broadcast it for the next round of iterations.

among the models solved by the distributed machines. The ADMM framework provides the freedom to propose efficient methods for solving the subproblems in distributed machines. We propose a dual coordinate descent method that achieves linear run-time complexity in the number of samples and takes advantage of the feature vector's sparsity. We also use the trust region Newton method to handle the dense data matrix.

Early work on distributed algorithms for kernel SVM was done by Chang *et al.* [3], who used an approximate matrix factorization to solve the convex optimization. However, the run-time complexity of kernel SVMs is at least quadratic in the number of samples. The authors in [2, 7] used the ADMM framework to solve a SVM problem in a single machine. However, they used a general convex optimization method for the subproblem, which has super linear complexity in the number of samples; moreover, they did not evaluate performance on large data in realistic distributed environments.

Our contributions are as follows. First, we propose an efficient distributed linear classification algorithm that achieves parallel disk loading and linear run-time and space complexity. Second, our experimental evaluations on large datasets in distributed environments show that the proposed approach is significantly faster than other existing distributed approaches, such as (i) parallel stochastic gradient descent using averaging to aggregate solutions as proposed by Zinkevich *et al.* [20] and (ii) Vowpal Wabbit (VW) [11]. Our proposed algorithm is also significantly faster than single-machine based algorithms, such as Block LIBLINEAR [19].

2 A Distributed Framework for Linear Classification

In this section, we apply ADMM to linear classification to yield the distributed algorithm. ADMM is a general framework for solving distributed optimization. The first work on ADMM may date back to the 1970s [8] and most of the theoretical results have been established in the 1990s [5]. However, until recently, ADMM was not widely known in the field. For completeness, we provide a review of its application to linear classification.

Given a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^{l}$ ($\mathbf{x}_i \in \mathbb{R}^n, y_i \in \{-1, +1\}$), we consider L2-regularized L2-loss (squared hinge loss) SVM as the linear classification model. Our algorithms also apply to hinge loss and squared loss as we show in Sec. 5. Here, we focus on L2-loss for the sake of presentation.

$$\min_{\mathbf{w}} f_1(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{i=1}^{l} \max(1 - y_i\mathbf{w}^T\mathbf{x}_i, 0)^2, \quad (1)$$

where C is a hyperparameter. For simplicity, we ignore the bias term, although one can append a constant to the feature vector. To make the problem amenable to decomposition, we first let $\{B_1, \dots, B_m\}$ be a partition of all data indices $\{1, \dots, l\}$. Then, we write down an equivalent problem as follows:

$$\begin{aligned} \min_{\mathbf{w}_1,\dots,\mathbf{w}_m,\mathbf{z}} \quad & \frac{1}{2}||\mathbf{z}||_2^2 + C\sum_{j=1}^{m}\sum_{i\in B_j} \max(1 - y_i\mathbf{w}_j^T\mathbf{x}_i, 0)^2 \\ & + \sum_{j=1}^{m} \frac{\rho}{2}||\mathbf{w}_j - \mathbf{z}||_2^2, \\ \text{subject to} \quad & \mathbf{w}_j - \mathbf{z} = 0, \quad j = 1, \dots, m, \qquad (2) \end{aligned}$$

where ρ is a constant step size for later iterations. Here we introduce a new weight vector $\mathbf{w}_j$ that is associated with data B_j, and a regularization vector $\mathbf{z}$. The term $\sum_{j=1}^{m} \frac{\rho}{2}||\mathbf{w}_j - \mathbf{z}||_2^2$ helps the algorithm converge more robustly.

Let us denote $w := \{\mathbf{w}_1, \dots, \mathbf{w}_m\}$ and $\lambda := \{\lambda_1, \dots, \lambda_m\}$, $\lambda_j \in \mathbb{R}^n, j = 1, \dots, m$. We can have the Lagrangian of problem (2) as follows:

$$\begin{aligned} \mathcal{L}(w, \mathbf{z}, \lambda) = & \frac{1}{2}||\mathbf{z}||_2^2 + C\sum_{j=1}^{m}\sum_{i\in B_j} \max(1 - y_i\mathbf{w}_j^T\mathbf{x}_i, 0)^2 \\ & + \sum_{j=1}^{m}\left(\frac{\rho}{2}||\mathbf{w}_j - \mathbf{z}||_2^2 + \lambda_j^T(\mathbf{w}_j - \mathbf{z})\right), \quad (3) \end{aligned}$$

where λ are the dual variables. ADMM consists of the following iterations:

$$w^{k+1} = \arg\min_{w} \mathcal{L}(w, \mathbf{z}^k, \lambda^k) \quad (4)$$

$$\mathbf{z}^{k+1} = \arg\min_{\mathbf{z}} \mathcal{L}(w^{k+1}, \mathbf{z}, \lambda^k) \quad (5)$$

$$\lambda_j^{k+1} = \lambda_j^k + \rho(\mathbf{w}_j^{k+1} - \mathbf{z}^{k+1}), \quad j = 1, \dots, m. \quad (6)$$

Since the Lagrangian $\mathcal{L}$ is separable in $\mathbf{w}_j$, we can solve

problem (4) in parallel:

$$\mathbf{w}_j^{k+1} = \arg\min_{\mathbf{w}'} C \sum_{i \in B_j} \max(1 - y_i \mathbf{w}'^T \mathbf{x}_i, 0)^2 + \frac{\rho}{2} ||\mathbf{w}' - \mathbf{z}||_2^2 + \lambda_j^T (\mathbf{w}' - \mathbf{z}),\ j = 1, ..., m. \quad (7)$$

Also, $\mathbf{z}^{k+1}$ has a closed form solution:

$$\mathbf{z}^{k+1} = \frac{\rho \sum_{j=1}^m \mathbf{w}_j^{k+1} + \sum_{j=1}^m \lambda_j^k}{m\rho + 1}. \quad (8)$$

A simple change of variables, letting $\lambda_j = \rho \mathbf{u}_j$, can make the quadratic form of $\mathbf{w}_j$ in Eq. (8) more compact. We now have the new ADMM iterations as:

$$\mathbf{w}_j^{k+1} = \arg\min_{\mathbf{w}} C \sum_{i \in B_j} \max(1 - y_i \mathbf{w}^T \mathbf{x}_i, 0)^2 + \frac{\rho}{2} ||\mathbf{w} - \mathbf{z}^k + \mathbf{u}_j^k||_2^2, \quad j = 1, \ldots, m. \quad (9)$$

$$\mathbf{z}^{k+1} = \frac{\sum_{j=1}^m \left(\mathbf{w}_j^{k+1} + \mathbf{u}_j^k\right)}{m + 1/\rho} \quad (10)$$

$$\mathbf{u}_j^{k+1} = \mathbf{u}_j^k + \mathbf{w}_j^{k+1} - \mathbf{z}^{k+1}, \quad j = 1, \ldots, m. \quad (11)$$

Here, each machine j solves the subproblem (9) in parallel, which is only associated with data $\mathbf{x}_{B_j} \triangleq \{\mathbf{x}_i : i \in B_j\}$. Machine j also loads the data $\mathbf{x}_{B_j}$ from the disk only once and stores them in the memory in the ADMM iterations. Each machine only needs to communicate $\mathbf{w}_j$ and $\mathbf{u}_j$ without passing the data.

The ADMM iterations also give the following theoretical guarantees. For any $\rho > 0$ we have:

1. If we define the residual variable $\mathbf{r}^k = [\mathbf{w}_1^k - \mathbf{z}^k, \ldots, \mathbf{w}_m^k - \mathbf{z}^k]$, then $\mathbf{r}^k \to \mathbf{0}$ as $k \to \infty$.
2. The objective function in problem (2) converges to the optimal objective function of problem (1).

To show the above, since $\frac{1}{2}||\mathbf{z}||_2^2$ and $\max(1 - y_i \mathbf{w}^T \mathbf{x}_i, 0)^2$ are both closed and proper convex functions and problem (1) possesses strong duality, they meet the conditions to guarantee the convergence results [2]. These results imply that eventually $\mathbf{w}_j^k$ would agree upon a consensus vector $\mathbf{z}^k$, which would converge to the solution of the original SVM problem (1). The auxiliary variables $\mathbf{u}_j^k$ convey to machine j how different its $\mathbf{w}_j^k$ is from $\mathbf{z}^k$ and serve as the signal to pull $\mathbf{w}_j^{k+1}$ into consensus when machine j solves the subproblem (9).

ADMM is essentially a subgradient-based method that solves the dual problem of (2). However, it has a better convergence result as compared to other distributed approaches, such as the dual decomposition method [2]. ADMM is a first order method, so it would seem that it would take many iterations to achieve high accuracy; however, our empirical experience suggests that ADMM usually takes only tens of iterations to achieve good accuracy. This few number of the iterations can be enough for large-scale classification tasks since the SVM (or logistic regression) uses loss functions that approximate the misclassification errors, and it may not be necessary to minimize these objectives exactly [1, 17]. In our experiments, we show that our algorithm achieves fast convergence in training optimality and test accuracy within only a few tens of ADMM iterations for large-scale datasets.

3 Efficient Algorithms for Solving the Subproblem

Although ADMM provides a parallel framework, the issue of solving the subproblem (9) efficiently still remains. In this subproblem, each machine contains a portion of the data, which can still be quite large to solve. In this section, we present both a dual method and a primal method. The dual method is a coordinate descent method similar to the one in [9]. To obtain an ϵ-optimal dual solution, our method has a computational complexity of $O(n_{nz} \log(1/\epsilon))$, where n_{nz} is the total number of non-zero features in the data. For well-scaled data, the term $\log(1/\epsilon)$ (including the constant) usually becomes a few of tens to achieve good accuracy, namely, the algorithm only needs to sequentially pass the data in a few of tens rounds. The primal method is a trust region Newton method similar to the one in [12]. This method obtains an approximated Hessian using the conjugate gradient. The computational complexity is $O(n_{nz} \times$ number of conjugate gradient iterations $\times$ number of Newton iterations). The primal method can be more efficient if the data matrix is dense. We first detail the dual coordinate descent method and then briefly describe the trust region Newton method.

3.1 A Dual Coordinate Descent Method

We rewrite the subproblem (9) in a more readable way:

$$\min_{\mathbf{w}} f_2(\mathbf{w}) = \frac{\rho}{2} ||\mathbf{w} - \mathbf{v}||_2^2 + C \sum_{i=1}^s \max(1 - y_i \mathbf{w}^T \mathbf{x}_i, 0)^2, \quad (12)$$

where $\mathbf{v} = \mathbf{z}^k - \mathbf{u}_j^k$ at the k-th ADMM iterations, and $\{\mathbf{x}_1, \ldots, \mathbf{x}_s\}$ denotes the data in $\mathbf{x}_{B_j}$ for some machine j. The above problem is different from traditional SVM in its regularization term, which also considers the consensus to other machines' solutions. Therefore, we still need an efficient special solver for this problem.

The dual problem of (12) can be written as a quadratic programming:

$$\min_{\alpha} \quad f_3(\alpha) = \frac{1}{2\rho} \alpha^T \bar{Q} \alpha - b^T \alpha \quad \text{subject to} \quad \alpha_i \geq 0, \quad \forall i, \quad (13)$$

where $\bar{Q} = Q + D$, $Q_{ij} = y_i y_j \mathbf{x}_i^T \mathbf{x}_j$, D is a diagonal matrix, $D_{ii} = \rho/(2C)$ and $b = [1 - y_1 \mathbf{v}^T \mathbf{x}_1, \ldots, 1 - y_s \mathbf{v}^T \mathbf{x}_s]^T$.

We solve this problem using dual coordinate descent, which optimizes one variable in α at a time and then circularly moves to the next variable and so on. For problem (13), the one-variable optimization has a close form solution since it is a quadratic minimization. In other words, for any i, we can optimize α_i while fixing other variables. Let $G_i^{(t)}$ be the partial derivative of f_3 with respect of α_i at the t-th iteration, then we have

$$G_i^{(t)} = \sum_{j=1}^{s} \alpha_j^{(t)} \bar{Q}_{ij} - b_i. \tag{14}$$

Thus, the optimal α_i will be the root of $G_i^{(t)}$ projected on $[0, \infty)$. We can update α_i as

$$\begin{aligned} \alpha_i^{(t+1)} &= \max\left(\frac{b_i - \sum_{j \neq i} \alpha_j^{(t)} \bar{Q}_{ij}}{\bar{Q}_{ii}}, 0\right) \\ &= \max\left(\alpha_i^{(t)} - G_i^{(t)}/\bar{Q}_{ii}, 0\right) \end{aligned} \tag{15}$$

We can also use the projected partial derivative, denoted as $PG_i^{(t)}$, to determine the stopping criteria of coordinate descent:

$$PG_i^{(t)} = \begin{cases} \min(0, G_i^{(t)}) & \text{if } \alpha_i^{(t)} = 0, \\ G_i^{(t)} & \text{otherwise.} \end{cases} \tag{16}$$

If $PG_i^{(t)} = 0$, we do not need to update $\alpha_i^{(t)}$.

To get $G_i^{(t)}$ in Eq. (14), the main computation is $O(s)$. However, due to the special structure of the problem, we can reduce to $O(\bar{n})$, where $\bar{n}$ is the average number of non-zero features in the data, to make the computational complexity independent of the size of the data, s. The key idea here is to maintain an intermediate vector $\mathbf{w}^{(t)}$ at each iteration as

$$\mathbf{w}^{(t)} = \sum_{j=1}^{s} y_j \alpha_j^{(t)} \mathbf{x}_j + \mathbf{v}. \tag{17}$$

Then, we can express $G_i^{(t)}$ as

$$G_i^{(t)} = y_i {\mathbf{w}^{(t)}}^T \mathbf{x}_i - 1 + D_{ii} \alpha_i^{(t)} \tag{18}$$

The main computation is then the dot product of $\mathbf{w}^{(t)}$ and $\mathbf{x}_i$, which is $O(\bar{n})$ if we save $\mathbf{x}_i$ as a sparse form. To maintain $\mathbf{w}^{(t)}$ after $\alpha_i^{(t)}$ changes to $\alpha_i^{(t+1)}$, we update it as follows:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \left(\alpha_i^{(t+1)} - \alpha_i^{(t)}\right) y_i \mathbf{x}_i. \tag{19}$$

This operation also takes $O(\bar{n})$. The overall procedure is provided in Algorithm 1.

For theoretical results, we can easily apply results in [13] to show that Algorithm 1 converges and it takes $O(\log(1/\epsilon))$ iterations of while-loops to achieve an ϵ-accurate solution, i.e., $f_3(\alpha) \leq f_3(\alpha^*) + \epsilon$. Therefore, the total computation is $O(s\bar{n}\log(1/\epsilon))$ for an ϵ-accurate solution, where $s\bar{n}$ is the total number of non-zero features in data $\mathbf{x}_{B_j}$.

Algorithm 1 A dual coordinate descent method for solving the problem (12)

Initialize $\alpha^{(0)}$, $\mathbf{w}^{(0)} = \sum_{i=1}^{s} y_i \alpha_i^{(0)} \mathbf{x}_i + \mathbf{v}$ and $t = 0$.
while $\alpha^{(t)}$ is not optimal **do**
  for $i = 1 \dots s$ **do**
    $t = t + 1$
    $G_i^{(t)} = y_i {\mathbf{w}^{(t)}}^T \mathbf{x}_i - 1 + D_{ii} \alpha_i^{(t)}$
    $PG_i^{(t)} = \begin{cases} \min(0, G_i^{(t)}) & \text{if } \alpha_i^{(t)} = 0, \\ G_i^{(t)} & \text{otherwise.} \end{cases}$
    if $|PG_i^{(t)}| \neq 0$ **then**
      $\alpha_i^{(t+1)} = \max\left(\alpha_i^{(t)} - G_i^{(t)}/Q_{ii}, 0\right)$
      $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \left(\alpha_i^{(t+1)} - \alpha_i^{(t)}\right) y_i \mathbf{x}_i$
    else
      $\alpha^{(t+1)} = \alpha^{(t)}$
      $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)}$
    end if
  end for
end while

3.2 A Trust Region Newton Method

We use the trust region Newton method in [12] to solve the problem (12). Since the Hessian in L2 loss SVM does not exist, the authors in [12] used a generalized Hessian for approximation. With the generalized Hessian they use the conjugate gradient method to find the Newton-like direction and use the trust region Newton method to iterate. The method is general and can be directly applied to our problem (12). The only important difference is the gradient of the objective function:

$$\nabla f_2(\mathbf{w}) = (\rho + 2C \sum_{i \in I} \mathbf{x}_i \mathbf{x}_i^T)\mathbf{w} - 2C \sum_{i \in I} y_i \mathbf{x}_i - \rho \mathbf{v}, \tag{20}$$

where $I = \{i | (1 - y_i \mathbf{w}^{(t)} \mathbf{x}_i) > 0\}$.

4 Improving the Distributed Algorithms

We have discussed the use of two efficient methods to solve the subproblem (9) under the framework of ADMM. However, there is still room for further improvement of the distributed algorithms. In this section, we describe several simple but important techniques that can significantly affect efficiency.

Random permutation of data. Since each machine processes only a portion of the data and communicates its solutions to other machines, if the local data contains mostly the same label, it may take a large number of ADMM iterations to achieve consensus. A useful technique is to randomly shuffle the data to different machines to ensure that class labels in the data are balanced. This technique reduces ADMM's total number of iterations.

Warm start in solving subproblem (9). In solving subproblem (9), it is not necessary to start from some fixed (e.g., zero) vector. In particular, $\mathbf{w}_j^k$ may not change much at the very end of the ADMM iterations.

So, we use $\mathbf{w}^{k-1}$ as the starting point for obtaining $\mathbf{w}^k$. Specifically, in Algorithm 1 we save the previous α used for obtaining $\mathbf{w}^{k-1}$ and reuse it as α_0. For the primal method, we can directly use $\mathbf{w}^{k-1}$ as the starting point for Newton iteration. This technique shortens the time within each ADMM iteration.

Inexact minimization (early stopping) of subproblem (9). Solving (9) exactly, especially at the initial ADMM iterations, may not be worthwhile since the exact solutions usually require a large amount of time and might not be the direction for achieving good consensus. In fact, the problem (9) can be solved approximately. In Algorithm 1, we can limit the while-loop's maximum number of iterations to be, for example, M times. This corresponds to going through the local data only M rounds. In the trust region Newton method, we can limit the number of Newton iterations. This technique can dramatically speed up the ADMM iterations.

Over-relaxation. $\mathbf{w}_j^{k+1}$ is used for updating $\mathbf{z}$ and $\mathbf{u}$ in (10)–(11). In fact, $\mathbf{w}_j^{k+1}$ can be added with the previous value of $\mathbf{z}^k$ to improve convergence. The new $\widehat{\mathbf{w}}_j^{k+1}$ in (10)-(11) can be written as:

$$\widehat{\mathbf{w}}_j^{k+1} = \beta \mathbf{w}_j^{k+1} + (1-\beta)\mathbf{z}^k. \tag{21}$$

We used $\beta \in [1.5, 1.8]$, as reported in [4]. This technique can reduce the total number of ADMM iterations while achieving the same accuracy.

5 Implementations

Our algorithms are simple and easy to implement on distributed systems. First, we consider their implementations from a data perspective. The data is split uniformly by data instances for processing on different machines. At the start, each machine loads its own data in parallel to fit in its RAM. Data in memory is represented as a sparse matrix, so the space complexity is $O(|B_j|\bar{n})$ for machine j. Each machine j maintains its own $\mathbf{w}_j$ and $\mathbf{u}_j$ in its memory and processes them in $\mathbf{w}$-update and $\mathbf{u}$-update in (9, 11) in parallel with other machines. Machine j broadcasts $\mathbf{w}_j^{k+1} + \mathbf{u}_j^k$ and waits to collect $\sum_{j=1}^m (\mathbf{w}_j^{k+1} + \mathbf{u}_j^k)$ for $\mathbf{z}$-update in (10). For the communication of $\mathbf{w}_j^{k+1} + \mathbf{u}_j^k$, and synchronization, i.e., waiting to collect $\sum_{j=1}^m (\mathbf{w}_j^{k+1} + \mathbf{u}_j^k)$, we use the Message-Passing Interface (MPI) [14] for inter-machine coordination, which is one of the most popular parallel programming frameworks for scientific applications. MPI supports high-level message communication and eliminates most of the programming burdens for low-level synchronization.

Second, we also implement our algorithms for different loss functions such as hinge loss and square loss. These loss functions would yield different forms of subproblem (9). However, we can apply the same idea of dual coordinate descent to solving the subproblem since it is still a quadratic programming and allows us to use the sparsity trick of Eq. (17). We implemented all the algorithms, e.g., ADMM and subproblem solvers, in C/C++. Specifically, we use OpenMPI for the communication in ADMM, and also modify the LIBLINEAR library for solving the problem (9) using the dual and primal methods. To further improve the implementations, we also used the following additional techniques.

Distributed normalization and evaluation of test accuracy. Our empirical findings suggest that normalizing the feature vector to unit length can be very helpful to the convergence of classification algorithms. Therefore, when each machine loads the data, it can normalize the data in parallel with other machines. Moreover, we can also evaluate the test data in a distributed fashion if test data is too big to load in one machine. These simple ideas allow the experiments to be done more efficiently.

Cross-validation and multiclass classification. Cross-validation (hyperparameter selection) is easily carried out in our implementations. Each machine can separate its own data into training and validation sets and perform both training and validation in a distributed fashion. For multiclass classification, we implement a one-versus-the-rest (OVR) method, since its accuracy is comparable to other surrogate-loss multiclass classifications [15, 10]. The OVR has essentially N binary classifications, where N is the number of classes. Note that, in our algorithms, the N binary classifications need to load the data only once.

6 Experiments

In this section, we first show that our proposed algorithms are faster than other existing distributed approaches on four large datasets. We then demonstrate a significant improvement over a single-machine solver and provide an analysis on these gains.

We consider four large datasets: a document dataset (`webspam`), an education dataset (`kddcup10`), and two synthetic datasets (`epsilon` and `kappa`).[1] The dataset `kappa` is generated as follows: $\mathbf{x}_i$ and $\mathbf{w}$ are uniformly sampled from $[-1, 1]^n$; $y_i = \text{sgn}(\mathbf{w}^T \mathbf{x}_i)$ and y_i will flip its sign with probability 0.1. Finally, we normalize $\mathbf{x}_i$ such that $||\mathbf{x}_i||_2^2 = 1$. All datasets are split into training and test sets with an 8:2 ratio, except the `kddcup10` dataset, which has already been separated into training and test sets. We also use five-fold cross validation to choose the best hyperparameter C since we need to compare the accuracy to VW, which has a different optimization problem. The details of the datasets are summarized in Table 1.

Our distributed algorithms are evaluated by running on 8 machines. Each machine has an Intel Core i7-950 processor (at 3.06GHz) and 12 GB RAM. The training/testing data is split and distributed evenly across these nodes. The disk throughput is ∼140 MB/sec. The machines are connected in a star network through

[1]The first three datasets are available at `http://www.csie.ntu.edu.tw/cjlin/libsvmtools/datasets/`. For `kddcup10`, we used a pre-possessed version of bridge_to_algebra_2008_2009 in KDD Cup 2010 such that each feature vector has unit norm.

Table 1: Summary of the datasets. l is the number of examples, and n the number of features. We also show the total number of non-zero features in the dataset. The memory represents the actual size of the data. Each element in the data is represented as 4 bytes for index and 8 bytes for values. A 64-bit machine will align the data structure and cause each element to have 16 bytes. *Split* is the initial time to split and compress the datasets into files. *Spread* is the initial time for our algorithms to disseminate the files to the corresponding machines.

Dataset	l	n	# nonzeros	Data size (GB)	C	Split (s)	Spread (s)
`webspam`	350,000	16,609,143	1,304,697,446	20.9	32	881	26
`kddcup10`	20,012,498	29,890,095	588,310,963	9.4	1	712	148
`epsilon`	500,000	2,000	1,000,000,000	16	2	504	12
`kappa`	500,000	8,000	4,000,000,000	64	0.5	3913	299

a 1 Gigabit Ethernet with the TCP throughput of ~111 MB/sec between any two machines.

6.1 Comparison with Other Distributed Solvers

First, we compare our algorithms with three distributed solvers that can load data in parallel and access the disk only once. The first solver is an extended version of Block LIBLINEAR [19]. We set up the serialized version of linear classification to run on multiple machines. All machines load the local data in parallel. Then only one machine runs at a time using Block LIBLINEAR, passing the processed model to the next machine as the initial model in a round-robin manner. This method saves a large amount of disk loading time but leaves all machines idle except one during training.

The second solver uses parallel stochastic gradient descent (SGD), similar to Zinkevich *et al.* [20]. The original algorithm sequentially passes the data only once using SGD, and then aggregates an average model. Here, we extend the algorithm by repeating such a procedure: the averaged weight vector is used as the initial point for the next round of SGD.

The third solver is the most recent version of Vowpal Wabbit (VW).[2] VW implements a fast online learning algorithm using SGD without regularization. The current version 6.0 starts to support running on clusters by using similar ideas in parallel SGD. VW directly uses sockets for the communications and synchronization of nodes. We would like to compare our method to VW since VW mixes the disk loading and training, and therefore might be faster than the second solver.

Now we describe the algorithms' settings. Unless explicitly indicated, the loss functions are all squared hinge loss. We used four different settings for our algorithms based on ADMM:

- DUAL: This uses the dual coordinate descent method, i.e., Algorithm 1 to solve the subproblem (9). The stopping criterion for Algorithm 1 is $\max_i |PG_i| < 0.001$, where all PG_i are in the same for-loop in Algorithm 1.
- DUAL.M1: This is similar to DUAL, except that it goes through the data only once in solving (9).
- PRIMAL: This uses the trust region Newton method we described in Sec. 3.2. Its stopping criterion is that the norm of gradient is less than 0.01.
- PRIMAL.M1: This is similar to PRIMAL, but it uses only one Newton step.

For ADMM iterations, we use over-relaxation with $\beta = 1.6$ and step size $\rho = 1$ for all cases.

Experimental settings for other distributed solvers are:

- D-B-LIBLINEAR: This uses distributed systems to run the serialized Block LIBLINEAR.
- D-B-LIBLINEAR.M1: This is similar to D-B-LIBLINEAR, except that each node only passes through the data only once in each run.
- PSGD: Parallel stochastic gradient descent as in Zinkevich *et al.* [20]. We compute the aggregated model every time the algorithm passes the whole data. The learning rate is set to 10^{-3} as in [20].
- PSGD.D[x]: Rather than using a constant learning rate in the SGD updates of PSGD, it uses a decaying learning rate where $\eta(t) = x/(t+1)$. We used $x = \{10^{-4}, 10^{-3}, 10^{-2}, 0.1, 0.5, 1, 2, 5\}$ and plot the best x.
- VW-Cluster (squared): We use the square loss for VW since we empirically found that it achieves better accuracy than other loss functions, such as hinge loss. We also use VW to compress the datasets and use the cached file for training. We set the number of bits for each feature to 24.
- VW-Cluster.A (squared): This is similar to VW-Cluster, but we add flags `--adaptive` and `--exact_adaptive_norm`, such that the gradient norms will be accumulated across nodes as well. These would be used to perform the non-uniform averaging of weights across the nodes for better convergence.

We first show the initial time for splitting the data and spreading it to different machines in Table 1.[3] We then measure the training performance in two metrics: training time vs. relative (training) optimality and training time vs. test accuracy. We define the training time starting from the disk loading. The relative optimality is defined as the following relative difference between the primal function value to the minimum function value found by all algorithms:

$$(f_1 - f_{1\text{best}})/f_{1\text{best}}. \tag{22}$$

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

[3] We note that it is possible to even avoid this one-time cost by designing a distributed learning system that accumulates the data in a distributed way.

We also compare the difference between current test accuracy and best accuracy (acc*%) over time, using

$$\text{acc}^{*}\% - \text{acc}\%. \quad (23)$$

All distributed solvers enjoy the benefit of loading in parallel only once from disk, which alleviates the cumbersome disk loading. Here, we are interested in which approach can converge quickly both in primal objective and test accuracy. As shown in the left column of Fig. 2, DUAL.M1 has the fastest convergence rate in reducing primal function value. Interestingly, even though DUAL.M1 goes through data only once when solving the subproblem (9), ADMM still improves in most of the iterations despite this inexact minimization. Using the exact minimization, such as DUAL and PRIMAL, does not yield much less primal function value and takes a longer time. PSGD*[4], despite our significant efforts of tuning the learning rate, cannot reach 1% relative optimality in a reasonable amount of time for the sparse datasets `webspam` and `kddcup`. PSGD* also has a slower convergence rate in the dense datasets `epsilon` and `kappa`. We conjecture that PSGD* is slower because it does not have auxiliary variables (e.g., $\mathbf{u}$ as in ADMM) to convey the differences in local models that can more strongly pull toward the consensus. D-B-LIBLINEAR* is slower because it does not use parallel training thus leaving most machines unutilized.

As the right column of Fig. 2 shows, DUAL.M1 (using squared hinge loss) is the fastest method to achieve the best accuracy in datasets `webspam`, `kddcup` and `epsilon`, while DUAL.M1 using hinge loss outperforms others in the `kappa` dataset. We show DUAL.M1 using hinge loss in the `kappa` dataset because it yields better accuracy than squared hinge loss, though not in the rest of the datasets. Note that VW does not support squared hinge loss, but does support other loss functions, such as squared loss and hinge loss. We found that VW using squared loss is much better than hinge loss, and therefore, we show the best results of VW in the figures. Since the objective function is different, VW is not directly comparable to DUAL*, PRIMAL*, PSGD*, and D-B-LIBLINEAR*. VW-Cluster.A is faster than VW-Cluster (except for dataset `kddcup`) because it uses non-uniform averaging to improve convergence. However, the non-uniform averaging still yields slower convergence of VW than our algorithms. In summary, these results suggest that our proposed optimization methods converge faster in test accuracy with a proper choice of loss function.

6.2 Comparison to a Single-machine Solver.

Now, we study how our distributed algorithms compare against Block LIBLINEAR running in a single machine in terms of training time. We denote the single-machine solver using Block LIBLINEAR as B-LIBLINEAR and its variant that passes the data only once in each block as B-LIBLINEAR.M1. See Fig. 3 for the break-down of training time. Although Block LIBLINEAR attempts to reduce the time spent in disk, the disk loading time still occupies a significant portion since Block LIBLINER would load the same samples from disk multiple times. We also observe that both DUAL.M1 and B-LIBLINEAR.M1 spend much less time in processing data than loading, and that they are more efficient than those that use exact minimization, such as DUAL and B-LIBLINEAR. These findings reveal that, in large-scale classification, the main component in the training time is data loading, which motivated us to improve it by using a distributed system for parallel loading. In addition to performing parallel disk loading only once, our algorithms introduce coordinations between machines, such as communications and synchronization. Communications involve the passing of $\mathbf{w}_j + \mathbf{u}_j$ at each ADMM iteration, and synchronization corresponds to the waiting time to collect this message. For relatively modest dimensional features such as in `epsilon` and `kappa`, the coordination overhead is quite small. For datasets that have high dimensional features, such as `webspam` and `kddcup10`, the coordination time turns out to be greater than the processing time. Overall, DUAL.M1 achieves $7 \sim 60$ fold speedup over B-LIBLINEAR or B-LIBLINEAR.M1.

[4] "*" means the wildcard character. Here PSGD* stands for PSGD and PSGD.D[x].

7 Discussions and Conclusion

Extremely high-dimensional data, e.g., having more than billions of features, would create significant overheads for our distributed algorithms to communicate due to the network bandwidth constraint. One solution for this is to use the hash trick [18] to randomly group different features to reduce the dimension and therefore alleviate communication overheads.

We evaluated and performed experiments on 8 machines, a typical scale in academia or research labs. It would be interesting to evaluate in a kilo-node scale as in a data center. In such settings, coordinations between nodes will need to be carefully designed when calculating average; for example, nodes in the same rack can aggregate the sums and then communicate across the racks. Our current implementation requires that the data be fit in the distributed memory to ensure fast training. If the data is larger than the distributed memory, it is straightforward to apply the idea of Block LIBLINEAR to load and train a portion of data in batch when solving the subproblems.

In this paper, we proposed simple and efficient distributed algorithms for large-scale linear classification. Our algorithms provide a significant performance gain over state-of-the-art linear classifiers running in a single machine and existing state-of-the-art distributed solvers, which shows the promise of our approach in large-scale learning problems. Our code is available at: `http://eecs.umich.edu/~caoxiezh/`.

Acknowledgments

This work was supported in part by a Google Faculty Research Award and AFOSR Grant FA9550-10-1-0393.

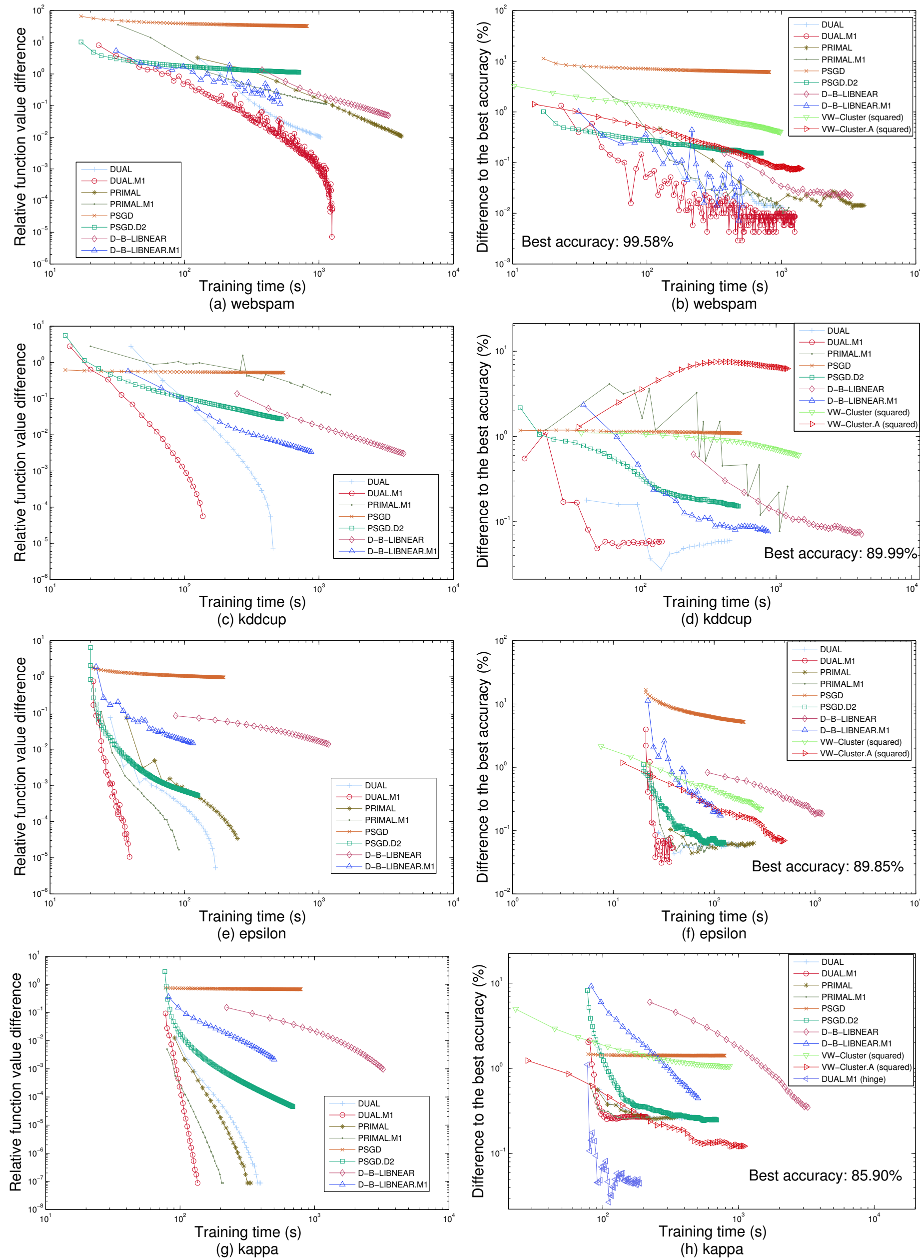

Figure 2: Performance comparisons between our algorithms, Block LIBLINEAR, PSGD and VW. Each marker indicates an iteration. We only show the results using hinge loss in DUAL.M1 for `kappa` since using hinge loss is better than squared hinge loss only for this dataset. We do not show PRIMAL in `kddcup10` dataset because it takes too long to converge. We do not show VW for the relative optimality since VW uses a different objective function.

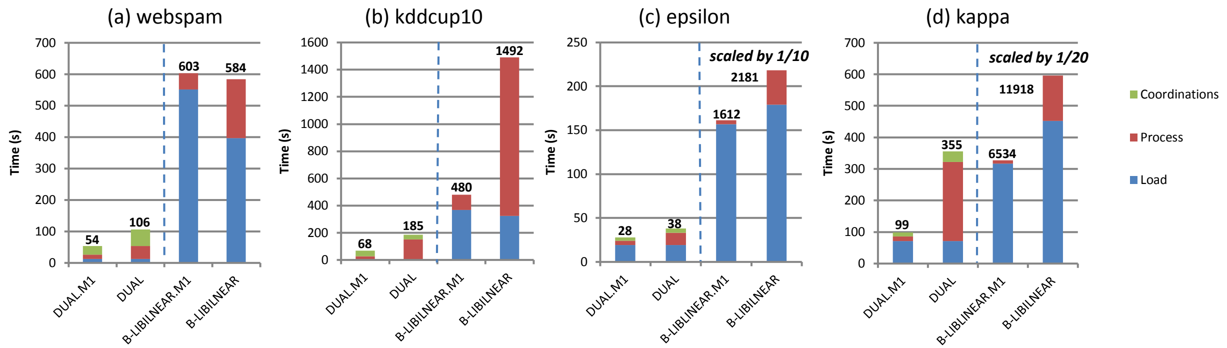

Figure 3: The break-down of training time for our distributed algorithms and Block LIBLINEAR (B-LIBLINEAR) running in a single machine. We measure the data loading time, processing time and coordination (communication and synchronization) time when both algorithms achieve 1% relative optimality with $C = 1$. Indeed, our methods spend much less time in data loading than B-LIBLINEAR.

References

[1] L. Bottou and Y. LeCun. Large scale online learning. In *Advances in Neural Information Processing Systems*, 2004.

[2] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. 3(1):1–122, 2011.

[3] E. Y. Chang, K. Zhu, H. Wang, and H. Bai. PSVM: Parallelizing support vector machines on distributed computers. In *Advances in Neural Information Processing Systems*, 2008.

[4] J. Eckstein. Parallel alternating direction multiplier decomposition of convex programs. *Journal of Optimization Theory and Applications*, 80(1):39–62, 1994.

[5] J. Eckstein and D. P. Bertsekas. On the douglas-rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55(1):293–318, 1992.

[6] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[7] P. A. Forero, A. Cano, and G. B. Giannakis. Consensus-based distributed support vector machines. *Journal of Machine Learning Research*, 11:1663–1707, 2010.

[8] D. Gabay and B. Mercier. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers and Mathematics with Applications*, 2(1):17–40, 1976.

[9] 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 *Proceedings of the 25th International Conference on Machine Learning*, 2008.

[10] S. S. Keerthi, S. Sundararajan, K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. A sequential dual method for large scale multi-class linear svms. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008.

[11] J. Langford, L. Li, and A. Strehl. Vowpal Wabbit online learning project. Technical report, 2007.

[12] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region Newton method for large-scale logistic regression. *Journal of Machine Learning Research*, 9:627–650, 2008.

[13] Z. Q. Luo and P. Tseng. On the convergence of the coordinate descent method for convex differentiable minimization. *Journal of Optimization Theory and Applications*, 72:7–35, 1992.

[14] MPI Forum. MPI: A Message-Passing Interface Standard, version 2.2, 2009.

[15] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–141, 2004.

[16] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated sub-gradient solver for SVM. In *Proceedings of the 24th International Conference on Machine Learning*, 2007.

[17] S. Shalev-Shwartz and N. Srebro. SVM optimization: inverse dependence on training set size. In *Proceedings of the 25th international Conference on Machine Learning*, pages 928–935, 2008.

[18] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10:2615–2637, 2009.

[19] H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin. Large linear classification when data cannot fit in memory. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010.

[20] M. Zinkevich, A. J. Smola, M. Weimer, and L. Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, 2010.