Large-scale Logistic Regression and Linear Support Vector Machines Using Spark

Chieh-Yen Lin
Dept. of Computer Science
National Taiwan Univ., Taiwan
r01944006@csie.ntu.edu.tw

Cheng-Hao Tsai
Dept. of Computer Science
National Taiwan Univ., Taiwan
r01922025@csie.ntu.edu.tw

Ching-Pei Lee *
Dept. of Computer Science
Univ. of Illinois, USA
clee149@illinois.edu

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

***Abstract*—Logistic regression and linear SVM are useful methods for large-scale classification. However, their distributed implementations have not been well studied. Recently, because of the inefficiency of the MapReduce framework on iterative algorithms, Spark, an in-memory cluster-computing platform, has been proposed. It has emerged as a popular framework for large-scale data processing and analytics. In this work, we consider a distributed Newton method for solving logistic regression as well linear SVM and implement it on Spark. We carefully examine many implementation issues significantly affecting the running time and propose our solutions. After conducting thorough empirical investigations, we release an efficient and easy-to-use tool for the Spark community.**

I. Introduction

Logistic regression (LR) and linear support vector machine (SVM) [1], [2] are popular methods in machine learning and data mining. They are useful for large-scale data classification. Many efficient single-machine methods for training these models are well studied [3]. However, the extremely large amount of data available nowadays is far beyond the capacity of a single machine. We therefore need more machines to store the data in memory for efficiently computations. MapReduce [4] is a popular framework for distributed applications with fault tolerance. Nevertheless, the heavy disk I/O of reloading the data at each MapReduce operation is an obstacle of developing efficient iterative machine learning algorithms on this platform.

Recently, Spark [5] has been considered as a great alternative of MapReduce in overcoming the disk I/O problem. Spark is an in-memory cluster-computing platform that allows the machines to cache data in memory instead of reloading the data repeatedly from disk. Although in-memory computing is a solution to reduce the disk I/O, developing an efficient and stable solver of LR and linear SVM on Spark is never easy. In the next paragraph, we list some important issues that should be considered.

First, for supporting fault tolerance, Spark applies read-only resilient distributed dataset (RDD) [6]. Without considering the properties of RDDs carefully, the price of fault tolerance may be expensive. Second, Spark is new and still under development, so the performance of its APIs is not clear. We therefore must carefully design criteria and experiments to analyze the performance of different possible implementations. Third, in contrast to traditional low-level communication interface like MPI [7] that supports both all-reduce operations and master-slave implementations, Spark only provides the master-slave structure. Thus, two types of communications are required as follows. The master machine first assigns the tasks and ships the necessary variables to the slave machines. The slave machines then send the computed results back to the master machine. To decrease the overheads of this structure, a discreet design is required.

In this paper, we consider a distributed version of the trust region Newton method (TRON) proposed by [8] to solve LR and linear SVM. Distributed TRON has recently been shown to be efficient with MPI in [9], but has not been studied on fault-tolerant distributed platforms such as Spark. We detailedly check the above issues to make our method efficient on Spark.

By thoroughly studying essential issues on efficiency and stability, we release our Spark implementation Spark LIBLINEAR[1] for LR and L2-loss SVM as an extension of the software LIBLINEAR [10].

This paper is organized as follows. Section II briefly introduces Spark. The formulation of LR and linear SVM, and their distributed training by TRON are discussed in Section III. In Section IV, we detailedly survey and analyze important implementation issues. Experimental results are shown in Section VI. Section VII concludes this work. A supplementary file including additional results is available at http://www.csie.ntu.edu.tw/~cjlin/papers/spark-liblinear/supplement.pdf.

II. Apache Spark

Traditional MapReduce frameworks such as Hadoop [11] have inefficient performance when conducting iterative computations because it requires disk I/O of reloading the data at each iteration. To avoid extensive disk I/O, distributed in-memory computing platforms become popular. Apache Spark [5] is a general-purpose in-memory cluster-computing platform. This platform allows users to develop applications by high-level APIs. Spark enables the machines to cache data and intermediate results in memory instead of reloading them from disk at each iteration. In order to support parallel programming, Spark provides resilient distributed datasets and parallel operations. This section discusses the details of these techniques.

* This work was done when Ching-Pei Lee was at National Taiwan University.

[1] http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/distributed-liblinear/

A. Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) [12] is a distributed file system designed for storing and manipulating large-scale data. HDFS provides a robust and convenient file system for Spark. It is highly fault-tolerant because of two useful strategies. First, data replication ensures that replicas (copies) of data are placed in several nodes. When one node is lost, replications of data stored on other nodes can still be accessible. Second, HDFS applies heartbeats to check the availability of the nodes. One node is set to be the name node and the rest are set to be data nodes. The name node manages the namespace of HDFS and the data nodes store data as blocks. HDFS heartbeats are periodically sent from data nodes to the name node. When the name node does not receive the heartbeat from a specific data node d_i, it marks d_i as a dead node and recovers the tasks done by d_i.

B. Resilient Distributed Datasets

In Spark, a partition is a distributed segment of data. When a driver program loads data into memory, a partition is a basic loading unit for the cache manager of Spark. If there is enough memory in the slave nodes, partitions will be cached in memory, and otherwise in disk. In Spark, a training data set is represented by a resilient distributed dataset (RDD) [6] which consists of partitions. An RDD is created from HDFS files or by transforming other RDDs.

The usage of RDDs is an important technique to realize parallel computing not only outside but also inside a slave node. A slave node only needs to maintain some partitions of the training data set. Instead of handling the original whole data set, every slave node can focus on its partitions simultaneously. This mechanism achieves parallelism if the number of partitions is enough. Assume the number of the slave machines is s and the number of partitions is p. The parallel computing can be fully enabled by specifying $p > s$. Therefore, the p partitions can be operated in parallel on the s slave machines. Users can specify the appropriate value of p according to their applications. In fact, the Spark system decides a $p_{\min}$ based on the size of the training data. If users do not set the value of p, or the user-specified p is smaller than $p_{\min}$, the Spark system will adopt $p = p_{\min}$.

Spark provides two types of parallel operations on RDDs: transformations and actions. Transformations, including operations like map and filter, create a new RDD from the existing one. Actions, such as reduce and collect, conduct computations on an RDD and return the result to the driver program.

C. Lineage and Fault Tolerance of Spark

The key mechanism in Spark for supporting fault tolerance is through read-only RDDs. If any partition is lost, the Spark system will apply transformations on the original RDD that creates this partition to recompute it. The transformations operations are maintained as a lineage, which records how RDDs are derived from other RDDs. Lineages are maintained in the master machine as the centralized metadata. They make the recomputation of RDDs efficient. The RDDs are made read-only to ensure that after reconducting the operations recorded in the lineage, we can obtain the same results.

III. Logistic Regression, Support Vector Machines and Distributed Newton Method

To illustrate how to apply a distributed version of TRON in solving LR and linear SVM, we begin with introducing their optimization formulations. We then discuss a trust region Newton method with its distributed extension.

A. Logistic Regression and Linear SVM

Given a set of training label-instance pairs $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^l$, $\boldsymbol{x}_i \in \mathbf{R}^n$, $y_i \in \{-1, 1\}$, $\forall i$, most linear classification models consider the following optimization problem.

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) \equiv \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum\nolimits_{i=1}^{l} \xi(\boldsymbol{w}; \boldsymbol{x}_i, y_i), \qquad (1)$$

where $\xi(\boldsymbol{w}; \boldsymbol{x}_i, y_i)$ is a loss function and $C > 0$ is a user-specified parameter. Commonly used loss functions include

$$\xi(\boldsymbol{w}; \boldsymbol{x}_i, y_i) \equiv \begin{cases} \max(0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i), & (2) \\ \max(0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i)^2, \quad \text{and} & (3) \\ \log\left(1 + \exp\left(-y_i\boldsymbol{w}^T\boldsymbol{x}_i\right)\right). & (4) \end{cases}$$

Problem (1) is referred as L1-loss and L2-loss SVM if (2) and (3) is used, respectively. When (4) is considered, (1) becomes LR. It is known that (3) and (4) are differentiable while (2) is not and is thus more difficult to optimize. Therefore, we focus on solving LR and L2-loss SVM in the rest of the paper.

B. A Trust Region Newton Method

Truncated Newton methods have been an effective method to solve (1) and other optimization problems. Here we consider a special version called trust region Newton methods (TRON). TRON has been successfully applied in [13] to solve LR and linear SVM under the single-core setting. We briefly introduce TRON in the rest of this sub-section.

At the t-th iteration, given the current iterate $\boldsymbol{w}^t$, TRON obtains the truncated Newton step by approximately solving

$$\min_{\boldsymbol{d}} \quad q_t(\boldsymbol{d}), \quad \text{subject to} \quad \|\boldsymbol{d}\| \le \Delta_t, \qquad (5)$$

where $\Delta_t > 0$ is the current size of the trust region, and

$$q_t(\boldsymbol{d}) \equiv \nabla f(\boldsymbol{w}^t)^T\boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T\nabla^2 f(\boldsymbol{w}^t)\boldsymbol{d} \qquad (6)$$

is the second-order Taylor approximation of $f(\boldsymbol{w}^t + \boldsymbol{d}) - f(\boldsymbol{w}^t)$. Because $\nabla^2 f(\boldsymbol{w})$ is too large to be formed and stored, a Hessian-free approach of applying CG (Conjugate Gradient) iterations is used to approximately solve (5). At each CG iteration we only need to obtain the Hessian-vector product $\nabla^2 f(\boldsymbol{w})\boldsymbol{v}$ with some vector $\boldsymbol{v} \in \mathbf{R}^n$ generated by the CG procedure. The details of the CG method is described in Algorithm 1. For LR,

$$\nabla^2 f(\boldsymbol{w}) = I + CX^T DX,$$

where I is the identity matrix,

$$X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l]^T$$

is the data matrix, and D is a diagonal matrix with

$$D_{i,i} = \frac{\exp(-y_i\boldsymbol{w}^T\boldsymbol{x}_i)}{(1 + \exp(-y_i\boldsymbol{w}^T\boldsymbol{x}_i))^2}.$$

Without explicit forming $\nabla^2 f(\boldsymbol{w})$, each CG iteration calculates

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + CX^T\left(D\left(X\boldsymbol{v}\right)\right). \tag{7}$$

Notice that when L2-loss SVM is considered, since it is not twice-differentiable, we follow [13], [14] to use a generalized Hessian when computing the Hessian-vector products.

After (5) is solved, TRON adjusts the trust region and the iterate according to how good the approximation is. Note that at each TRON iteration, the function value and the gradient are evaluated only once, while it is clear from Algorithm 1 that we need to compute the Hessian-vector products for several different vectors in the iterative CG procedure until (5) is solved. More details can be found in [13].

Algorithm 1 CG procedure for approximately solving (5)

1: Given $\xi_t < 1, \Delta_t > 0$. Let $\bar{\boldsymbol{d}}^0 = 0, \boldsymbol{r}^0 = -\nabla f(\boldsymbol{w}^t)$, and $\boldsymbol{s}^0 = \boldsymbol{r}^0$.
2: For $i = 0, 1, \ldots$ (inner iterations)
3: If $\|\boldsymbol{r}^i\| \leq \xi_t \|\nabla f(\boldsymbol{w}^t)\|$, output $\boldsymbol{d}^t = \bar{\boldsymbol{d}}^i$ and stop.
4: Compute

$$\boldsymbol{u}^i = \nabla^2 f(\boldsymbol{w}^t)\boldsymbol{s}^i. \tag{8}$$

5: $\alpha_i = \|\boldsymbol{r}^i\|^2 / ((\boldsymbol{s}^i)^T \boldsymbol{u}^i)$.
6: $\bar{\boldsymbol{d}}^{i+1} = \bar{\boldsymbol{d}}^i + \alpha_i \boldsymbol{s}^i$.
7: If $\|\bar{\boldsymbol{d}}^{i+1}\| \geq \Delta_t$, compute τ such that $\|\bar{\boldsymbol{d}}^i + \tau \boldsymbol{s}^i\| = \Delta_t$, then output the vector $\boldsymbol{d}^t = \bar{\boldsymbol{d}}^i + \tau \boldsymbol{s}^i$ and stop.
8: $\boldsymbol{r}^{i+1} = \boldsymbol{r}^i - \alpha_i \boldsymbol{u}^i$.
9: $\beta_i = \|\boldsymbol{r}^{i+1}\|^2 / \|\boldsymbol{r}^i\|^2$.
10: $\boldsymbol{s}^{i+1} = \boldsymbol{r}^{i+1} + \beta_i \boldsymbol{s}^i$.

C. Distributed Algorithm

From the discussion in the last section, the computational bottleneck of TRON is (8), which is the product between the Hessian matrix $\nabla^2 f(\boldsymbol{w}^t)$ and the vector $\boldsymbol{s}^i$. This operation can possibly be parallelized in a distributed environment as parallel vector products. Based on this observation, we discuss a method of running TRON distributedly. To simplify the discussion, we only demonstrate the algorithm for solving LR.

We first partition the data matrix X and the labels Y into disjoint p parts.

$$X = [X_1, \ldots, X_p]^T,$$
$$Y = \text{diag}(y_1, \ldots, y_l) = \begin{bmatrix} Y_1 & & \\ & \ddots & \\ & & Y_p \end{bmatrix},$$

where $\text{diag}(\cdot)$ represents a diagonal matrix. For easier description, we introduce the following component-wise function

$$\sigma(\boldsymbol{v}) \equiv [1 + \exp(-v_1), \ldots, 1 + \exp(-v_n)]^T,$$

and the operations on this function are also component-wise. We then reformulate the function, the gradient and the Hessian-vector products of (1) as follows.

$$f(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum\nolimits_{k=1}^{p} f_k(\boldsymbol{w}), \tag{9}$$

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + C\sum\nolimits_{k=1}^{p} \nabla f_k(\boldsymbol{w}), \tag{10}$$

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + C\sum\nolimits_{k=1}^{p} \nabla^2 f_k(\boldsymbol{w})\boldsymbol{v}, \tag{11}$$

where

$$f_k(\boldsymbol{w}) \equiv \boldsymbol{e}_k^T \log\left(\sigma\left(Y_k X_k \boldsymbol{w}\right)\right), \tag{12}$$

$$\nabla f_k(\boldsymbol{w}) \equiv (Y_k X_k)^T \left(\sigma\left(Y_k X_k \boldsymbol{w}\right)^{-1} - \boldsymbol{e}_k\right), \tag{13}$$

$$\nabla^2 f_k(\boldsymbol{w})\boldsymbol{v} \equiv X_k^T\left(D_k\left(X_k\boldsymbol{v}\right)\right), \tag{14}$$

$$D_k \equiv \text{diag}\left(\left(\sigma\left(Y_k X_k \boldsymbol{w}\right) - \boldsymbol{e}_k\right) / \sigma\left(Y_k X_k \boldsymbol{w}\right)^2\right),$$

and $\boldsymbol{e}_k$ is the vector of ones. Note that $\log(\cdot)$ is used as a component-wise function in (12). The functions $f_k, \nabla f_k$ and $\nabla^2 f_k$ are the map functions operating on the k-th partition. We can observe that for computing (12)-(14), only the data partition X_k is needed in computing. Therefore, the computation can be done in parallel, with the partitions being stored distributedly. After the map functions are computed, we need to reduce the results to the machine performing the TRON algorithm in order to obtain the summation over all partitions. This algorithm requires two phrases of communications. The first one is shipping $\boldsymbol{w}$ and $\boldsymbol{v}$ to all slave machines. The second one is reducing the results of those map functions to the master machine. The details are described in Algorithm 2. Note that except the computation of (8), Algorithm 1 is conducted on the master machine and hence not parallelized. However, since the cost of each CG step excluding the calculation of (8) is only $O(n)$, the CG procedure will not become a bottleneck even we use many nodes.

Algorithm 2 A distributed TRON algorithm for LR and SVM

1: Given $\boldsymbol{w}^0, \Delta_0, \eta, \epsilon$.
2: For $t = 0, 1, \ldots$
3: The master ships $\boldsymbol{w}^t$ to every slave.
4: Slaves compute $f_k(\boldsymbol{w}^t)$ and $\nabla f_k(\boldsymbol{w}^t)$ and reduce them to the master.
5: If $\|\nabla f(\boldsymbol{w}^t)\| < \epsilon$, stop.
6: Find $\boldsymbol{d}^t$ by solving (5) using Algorithm 1.
7: Compute $\rho_t = \frac{f(\boldsymbol{w}^t + \boldsymbol{d}^t) - f(\boldsymbol{w}^t)}{q_t(\boldsymbol{d}^t)}$.
8: Update $\boldsymbol{w}^t$ to $\boldsymbol{w}^{t+1}$ according to

$$\boldsymbol{w}^{t+1} = \begin{cases} \boldsymbol{w}^t + \boldsymbol{d}^t & \text{if } \rho_t > \eta, \\ \boldsymbol{w}^t & \text{if } \rho_t \leq \eta. \end{cases}$$

9: Obtain Δ_{t+1} by rules in [13].

IV. Implementation Design

In this section, we study implementation issues for our software. We name our distributed TRON implementation Spark LIBLINEAR because algorithmically it is an extension of the TRON implementation in the software LIBLINEAR [10]. Because Spark is implemented in Scala, we use the same language. Scala [15] is a functional programming language that runs on the Java platform. Furthermore, Scala programs are compiled to JVM bytecodes. The implementation of Spark LIBLINEAR involves complicated design issues resulting from Java, Scala and Spark. For example, in contrast to traditional languages like C and C++, similar expressions in Scala may easily behave differently. It is hence easy to introduce overheads in developing Scala programs. A careful design is necessary. We analyze the following different implementation

issues for efficient computation, communication and memory usage.

- Programming language:
  - Loop structure
  - Data encapsulation
- Operations on RDD:
  - Using mapPartitions rather than map
  - Caching intermediate information or not
- Communication:
  - Using broadcast variables
  - The cost of the reduce function

The first two issues are related to Java and Scala, while the rest four are related to Spark. After the discussion in this section, we conduct empirical comparisons in Section VI.

A. Loop Structure

From (12)-(14), clearly the computational bottleneck at each node is on the products between the data matrix X_k (or X_k^T) and a vector $\boldsymbol{v}$. To compute this matrix-vector product, a loop to conduct inner products between all $\boldsymbol{x}_i \in X_k$ and $\boldsymbol{v}$ is executed. This loop, executed many times, is the main computation in our algorithm. Although a for loop is the most straightforward way to implement an inner product, unfortunately, it is known that in Scala, a for loop may be slower than a while loop.[2] To study this issue, we discuss three methods to implement the inner product: for-generator, for-range and while.

Building a for-generator involves two important concepts: collection and generator. A collection is a container which stores data variables of the same type. It allows convenient batch operations. In Scala, a syntax such as "element $\leftarrow$ collection" is called a generator for iterating through all elements of a collection. The approach for-generator involves a generator to create an iterator for going through elements of a collection. This method is notated by

for(element $\leftarrow$ collection) { ... }

Another way to have a for loop is by going through a range of indices, where the range is created by a syntax like "3 to 18." This method, denoted as for-range, can be implemented by

for(i $\leftarrow$ 0 to collection.length) { ... }

Finally, we consider the approach of using a while loop to implement the inner product:

```
i = 0;
while(condition) {
        ...
        i = i + 1;
}
```

From the viewpoint of writing a program, a for rather than a while loop should be used. However, the performance of a for loop is not as efficient as a while loop. Actually, there is only the syntax of "for expression" instead of "for loop" in Scala. It turns out that the Scala compiler translates the for expression into a combination of higher-order operations. For example [16], the for-generator approach is translated into:

collection.foreach { **case** element => operations }

The translation comes with overheads and the combination becomes complicated when more operations are applied. Further, the optimization of a for expression has not been a focus in Scala development because this expression is too imperative to consist with the functional programming principle [15]. In contrast, a while loop is a loop rather than an expression, so it acquires relatively efficient optimization. Therefore, we choose the while loop to implement our software.

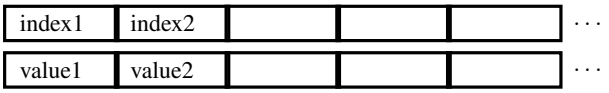
B. Data Encapsulation

We follow LIBLINEAR [10] to represent data as a sparse matrix, where only non-zero entries are stored. This strategy is important to handle large-scale data. For example, for a given 5-dimensional feature vector $(2, 0, 0, 8, 0)$, only two index-value pairs of non-zero entries are stored.

We investigate how to store the index-value information such as "1:2" and "4:8" in memory. The discussion is based on two encapsulation implementations: the Class approach (CA) and the Array approach (AA).

CA encapsulates a pair of index and feature value into a class, and maintains an array of class objects for each instance:

In contrast, AA directly uses two arrays to store indices and feature values of an instance:



One advantage of CA is the readability of source code. However, this design causes overheads on both memory usage and accessing time. For memory usage, CA requires additional memory than AA because each class object maintains an object header in memory. Note that the number of object headers used is proportional to the number of non-zero values in the training data. For the accessing time, AA is faster because it directly accesses indices and values, while the CPU cache must access the pointers of class objects first if CA is applied.

C. Using mapPartitions Rather Than map

The second term of the Hessian-vector product (7) can be represented as the following form.

$$\sum_{i=1}^{l} \boldsymbol{x}_i D_{i,i} \boldsymbol{x}_i^T \boldsymbol{v} = \sum_{i=1}^{l} a(\boldsymbol{x}_i, y_i, \boldsymbol{w}, \boldsymbol{v}) \boldsymbol{x}_i,$$

where $a(\boldsymbol{x}_i, y_i, \boldsymbol{w}, \boldsymbol{v}) = D_{i,i} \boldsymbol{x}_i^T \boldsymbol{v}$. Then map and reduce operations can be directly applied; see Algorithm 3. However, considerable overheads occur in the map operations because for each instance $\boldsymbol{x}_i$, an intermediate vector $a(\boldsymbol{x}_i, y_i, \boldsymbol{w}, \boldsymbol{v}) \boldsymbol{x}_i$ is created.

In addition to the above-mentioned overheads, the reduce function in Algorithm 3 involves complicated computation.

[2] https://issues.scala-lang.org/browse/SI-1338

Algorithm 3 map implementation

```
1: data.map(new Function() {
2:     call(x, y) { return a(x, y, w, v)x }
3: }).reduce(new Function() {
4:     call(a, b) { return a + b }
5: })
```

Algorithm 4 mapPartitions implementation

```
1: data.mapPartitions(new Function() {
2:     call(partition) {
3:         partitionHv = new DenseVector(n)
4:         for each (x, y) in partition
5:             partitionHv += a(x, y, w, v)x
6:     }
7: }).reduce(new Function() {
8:     call(a, b) { return a + b }
9: })
```

Because we consider sparse storage, intermediate vectors may also be represented as sparse vectors. Consequently, the reduce function involves additions of sparse vectors. When adding two sparse vectors, the vector sum easily contains more non-zero values, so the creation of longer index and value arrays is inevitable. Because numerous temporary arrays may be created in the reduce function, the cost is not negligible.

To avoid the overheads and the complicated additions of sparse vectors, we consider the mapPartitions operation in Spark; see Algorithm 4. Spark provides mapPartitions to apply a function on each partition of an RDD. In Algorithm 4, all instances in a partition share only one intermediate vector called partitionHv, which saves $\sum_{i \in X_k} a(\boldsymbol{x}_i, y_i, \boldsymbol{w}, \boldsymbol{v})\boldsymbol{x}_i$, where X_k is the k-th part of data matrix X. This setting ensures that computing a Hessian-vector product involves only p intermediate vectors. Thus, the overheads of using mapPartitions is less than that of using map with l intermediate vectors. Moreover, additions of sparse vectors are replaced by simply accumulating sparse vectors on a dense vector. The dense vector contains all entries of the features, so the values in sparse vectors can be directly added to the dense vector. Thus no creation of new index and value arrays is involved. One may question that a dense vector requires more memory than a sparse vector, but this disadvantage is negligible unless only few non-zero elements are in the dense vector. In fact, the partitionHv vectors tend to be dense because gradients and Hessian-vector products are usually dense even when the data is sparse. Hence, The memory usage of dense vectors is not an issue.

We note that the technique of using mapPartitions can also be applied to compute the gradient.

D. Caching Intermediate Information or not

The calculations of (12)-(14) all involve the vector $\sigma(Y_k X_k \boldsymbol{w})$.[3] Instinctively, if this vector is cached and shared between different operations, the training procedure can be more efficient. In fact, the single-machine package LIBLINEAR uses this strategy in their implementation of TRON. Because our data is represented by an RDD, one straightforward thinking of implementing this strategy is to save the vector in this data RDD. However, because RDDs are read-only, we should not store variables that would change with iterations in the data RDD. One may then want to create a new RDD containing only $\sigma(Y_k X_k \boldsymbol{w})$ at each Newton iteration to share the cached information between operations. Nevertheless, this created RDD is useless. Spark does not allow any single operation to gather information from two different RDDs and run a user-specified function such as (12), (13) or (14). Therefore, we must create one new RDD per iteration to store both the training data and the information to be cached. The new RDDs are transformed from the original data RDD. Unfortunately, this approach incurs severe overheads in copying the training data from the original RDD to the new one. Furthermore, the dependency of the new RDDs on the original data RDD lengthens the lineage. When a slave machine fails during training, Spark traces back through the lineage to recover the task. The longer lineage may cause longer time of recovery.

Another possibility of caching $\sigma(Y_K X_k \boldsymbol{w})$ is to store them in the master machine after the function (12) is computed, and then ship them back to the slaves in computing (13) and (14). This approach requires additional communication of collecting and distributing $\sigma(Y_k X_k \boldsymbol{w})$ in the cluster. The cost of this communication is proportional to $O(l)$, so this approach may be feasible when l is not too large. However, when the data size is large, this additional communication cost may counterbalance the computation time saved by caching $\sigma(Y_k X_k \boldsymbol{w})$.

Based on the above discussion, we decide not to cache $\sigma(Y_k X_k \boldsymbol{w})$ because recomputing them is more cost-effective. In the study of [9] with an implementation based on MPI (see Section V-B), this issue does not occur. Therefore, this example demonstrates that specific properties of a parallel programming framework may strongly affect the implementation.

E. Using Broadcast Variables

In Algorithm 2, communication occurs at two places. The first one is sending $\boldsymbol{w}$ and $\boldsymbol{v}$ from the master machine to the slave machines, and the second one is reducing the results of (12)-(14) to the master machine. We discuss the first case here, and postpone the second one to Section IV-F.

In Spark, when an RDD is split into partitions, one single operation on this RDD is divided into tasks working on different partitions. Take Algorithm 2 as an example. The data matrix X forms an RDD, and it is split into $X_1, \ldots, X_p$. The task of computing (9) is divided into computing $f_k(\boldsymbol{w})$ on the partition X_k in (12) for $k = 1, \ldots, p$. In this algorithm, when an operation of computing (9), (10) or (11) is conducted, by default, Spark sends a copy of $\boldsymbol{w}$ and $\boldsymbol{v}$ to each partition. As mentioned in Section II-B, the number of partitions p is set to be larger than the number of slaves s. Under this setting, many redundant communications occur because we just need to send a copy to each slave machine but not each partition. When the vector dimension n is large, these redundant communications may incur high cost. Besides, sending $\boldsymbol{w}$ once per TRON

[3] For L2-loss SVM, the vector to be considered is $Y_k X_k \boldsymbol{w}$.

iteration instead of sending it for each operation of computing (9), (10) or (11) is a more efficient strategy. In such a case where each partition shares the same information from the master, it is recommended to use broadcast variables.[4] Broadcast variables are read-only shared variables that are cached in each slave machine and can be accessed by all the partitions of the RDD running on this slave. Because $p > s$, broadcast variables may effectively reduce the communication cost.

It is clear from Algorithm 2 that at the same iteration of TRON, the functions (12)-(14) share the same $\boldsymbol{w}$. Thus, we only need to broadcast $\boldsymbol{w}$ once per TRON iteration.[5] Then slaves can used the cached broadcast variable $\boldsymbol{w}$ for computing (12)-(14). Assume there are c_i times of CG involved in iteration i. By using broadcast variables, we can avoid the c_i times of sending the same $\boldsymbol{w}$ to the slaves during the CG procedure.

F. The Cost of the reduce Function

We continue to discuss the second place incurring communications. Because (12) is a scalar and the communication cost of reducing it is negligible, we focus on reducing the vectors (13) and (14) of size n from slaves to the master. In the reduce operation, the slaves send the output of each partition to the master separately. Thus in Algorithm 4, totally p vectors with size n are sent to the master. If we locally combine the outputs of partitions on the same slave machine before sending them to the master, only s vectors are communicated. We adopt the coalesce function in our implementation to approach this goal. The coalesce function returns a new RDD with p_c partitions, where $p_c < p$ is user-specified. The new partition is formed by merging the original RDD partitions with consideration of locality. In our implementation, we execute the coalesce function with $p_c = s$ between the mapPartitions and the reduce functions. If the locality of partitions is fully utilized in the coalesce function, the partitions of the original RDD on the same slave machine are merged into one partition. Consequently, the input of the reduce function has only s partitions, and hence the number of vectors sent from slaves to the master is decreased from p to s.

V. Related Works

Among the existing tools for distributed training of LR, we discuss two publicly available ones that will be used in our experimental comparison in Section VI.

A. LR Solver in MLlib

MLlib[6] is a machine learning library implemented in Apache Spark. It provides a stochastic gradient (SG) method for LR. In this SG implementation, at the t-th iteration and given the current iterate $\boldsymbol{w}^t$, the model is updated by

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \frac{\eta_0}{\sqrt{t}} \nabla f_{I_t}(\boldsymbol{w}^t),$$

where $\eta_0 > 0$ is the initial learning rate specified by the user, I_t is a randomly drawn subset of the training data, and

$$f_{I_t}(\boldsymbol{w}^t) = \frac{1}{2}\|\boldsymbol{w}^t\|^2 + C \sum\nolimits_{i \in I_t} \log(1 + \exp(-y_i \boldsymbol{w}^T \boldsymbol{x}_i)).^7$$

The computation of $\nabla f_{I_t}(\boldsymbol{w}^t)$ is similar to that of (10), except that X_k and Y_k in (13) are replaced by their intersection with I_t. Their implementation also follows the master-slave structure. Therefore, communications similar to our method are also required: sending $\boldsymbol{w}$ to slaves and gathering the gradient from slaves. Hence, the communication problems described in Section IV-E and IV-F also occur in MLlib but have not been solved.

B. MPI LIBLINEAR

MPI LIBLINEAR [9] is an MPI implementation of a distributed TRON algorithm that is similar to the one discussed in Section III-C. We investigate some important differences between MPI LIBLINEAR and our Spark LIBLINEAR.

First, MPI LIBLINEAR is implemented in C++ while Spark LIBLINEAR is in Scala. The C++ implementation is simpler because issues such as the difference between for and while loops mentioned in Section IV-A do not exist. Second, because MPI does not involve high-level APIs such as map and mapPartitions in Spark, the issue discussed in Section IV-C also does not exist in MPI LIBLINEAR. Third, Spark LIBLINEAR can recover from failure of machines. The principle of read-only RDDs is essential to support fault tolerance. Thus, the discussion on caching $\sigma(Y_k X_k \boldsymbol{w})$ in Section IV-D is an important issue for efficiency. In contrast, MPI LIBLINEAR does not support fault tolerance, so the read-only principle is not considered in MPI LIBLINEAR. Finally, there is no master machine in MPI LIBLINEAR. It only has a program to activate the distributed tasks. Machines directly share all information with each other by all-reduce operations. In contrast, Spark only supports the master-slave structure. As a consequence, MPI LIBLINEAR only requires one communication (slave to the other slaves) while Spark LIBLINEAR needs two (master to slaves in Section IV-E and slaves to master in Section IV-F) each time computing (9), (10) or (11).

VI. Experiments

In this section, first we experimentally investigate implementation issues examined in Section IV. We then show the scalability of Spark LIBLINEAR. Finally, we compare the performance between MLlib, MPI LIBLINEAR and Spark LIBLINEAR. We consider some real-world data sets listed in Table I. All data sets except yahoo-japan and yahoo-korea are available at the LIBSVM data set page.[8] Because of space limitation, we present results of only LR here and leave the results of L2-loss SVM in the supplementary materials.

[4] http://spark.apache.org/docs/latest/programming-guide.html

[5] We cache $\boldsymbol{w}^t + \boldsymbol{d}$ used in line 7 as the new broadcast variables $\boldsymbol{w}^{t+1}$ for the next iteration. Thus in general we broadcast $\boldsymbol{w}$ once per iteration unless $rho_t \leq \eta$. In our experiments, the case $\rho_t \leq \eta$ occurs very seldom.

[6] http://spark.apache.org/docs/1.0.0/mllib-guide.html

[7] In the implementation of MLlib, the following objective function is used.

$$\tilde{f}(\boldsymbol{w}) = \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{1}{l}\sum\nolimits_{i=1}^{l} \log(1 + \exp(-y_i \boldsymbol{w}^T \boldsymbol{x}_i)).$$

We then use $\lambda = 1/Cl$ and multiply $\tilde{f}$ by Cl to make the function equal to (1) with the logistic loss.

[8] http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/. Note that we use the unigram version of webspam in our experiments.

TABLE I. DATA INFORMATION: DENSITY IS THE AVERAGE RATIO OF NON-ZERO FEATURES PER INSTANCE.

Data set	#instances	#features	density	#nonzeros	p_s	p_e
ijcnn	49,990	22	59.09%	649,870	1	
real-sim	72,309	20,958	0.25%	3,709,083	2	
rcv1	20,242	47,236	0.16%	1,498,952	1	
news20	19,996	1,355,191	0.03%	9,097,916	2	
covtype	581,012	54	22.00%	6,901,775		32
webspam	350,000	254	33.52%	29,796,333	6	32
epsilon	400,000	2,000	100.00%	800,000,000		183
rcv1t	677,399	47,236	0.16%	49,556,258		32
yahoo-japan	176,203	832,026	0.02%	23,506,415	5	32
yahoo-korea	460,554	3,052,939	0.01%	156,436,656		34

Four smaller and two larger sets are chosen for the experiments of loops and encapsulation in Sections VI-A and VI-B. These two issues are strongly related to Scala, so we run experiments only on one and two nodes to reduce the effect of communication. To fit data in the memory of one or two nodes, the training data sets must be small. Larger sets are used for other experiments to check the efficiency on real distributed environments.

The number of partitions used in our experiments is also listed in Table I. The value p_s is used for the experiments of loops and encapsulation, and p_e is applied for the rest. In the experiments of loops and encapsulation, we adopt the Spark-calculated $p_{\min}$ as p_s to reduce the effect of combining partitions. As mentioned in Section II-B, $p_{\min}$ is a constant for a given data set. For other experiments, $p_{\min}$ is probably not suitable because when it is not large enough, the parallel function of Spark will not be enabled. One may want to choose a large p to exploit parallelism. However, a too large p leads to extra overheads of maintaining and handling numerous partitions. Eventually, to check the performance of 16 nodes with multiple cores, we choose $p_e = \max(32, p_{\min})$. This setting ensures that there are at least two partitions for each slave to handle while p will not be too large.

We evaluate the performance by the relative difference to the optimal function value:

$$|\frac{f(\boldsymbol{w}) - f(\boldsymbol{w}^*)}{f(\boldsymbol{w}^*)}|.$$

The optimal $f(\boldsymbol{w}^*)$ is obtained approximately by running optimization methods with a very tight stopping condition. All experiments use $C = 1$ and are conducted on a cluster where each machine has 16 GB of RAM and 8 virtual CPUs.

A. *Different Loop Structures*

The performance of different loop implementations is evaluated in Figure 1. The while loop is significantly more efficient in almost all cases. Therefore, while loops are adopted in our final implementation and all subsequent experiments. Notice that if two nodes are used, results in Figures 1(g)-1(l) show that the difference between for and while loop reduces for small-scale data sets. The reason is that when data is split between two nodes, each node requires conducting fewer loops.

B. *Encapsulation*

We now check the encapsulation issue discussed in Section IV-B. The approach CA encapsulates the index and value of each feature into a class, while the other approach AA directly uses two arrays to store the indices and values separately. Results of using one and two nodes are in Figure 2. One can observe that AA is more efficient when one node is used, and is faster or equally good when two nodes are used. This confirms our conjecture in Section IV-B that AA has less overheads than CA. Therefore, we apply the AA encapsulation in our implementation.

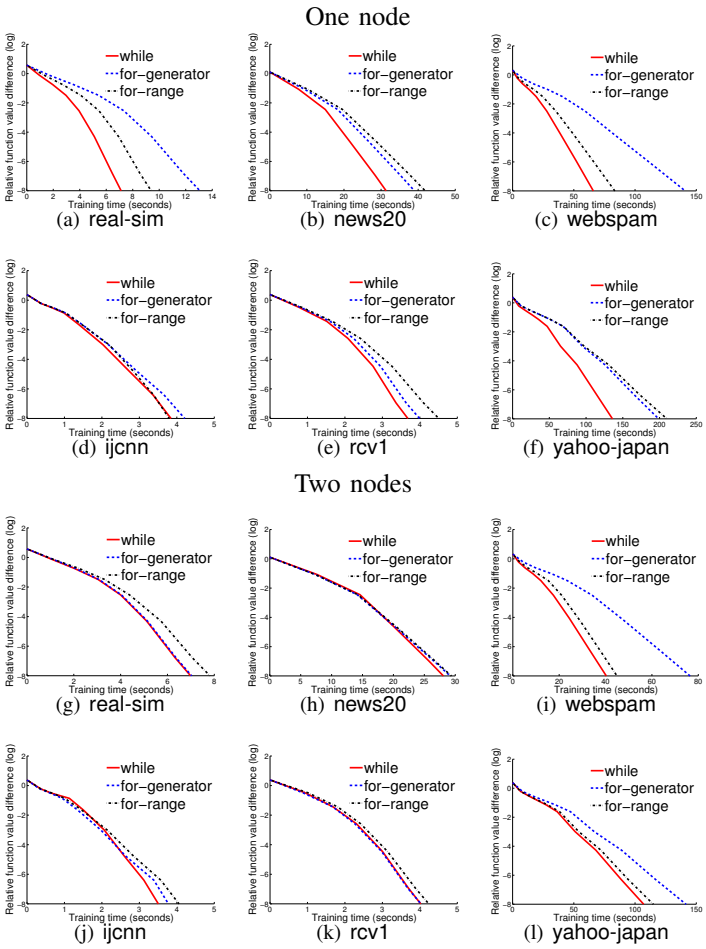


Fig. 1. Loop implementation comparison: running time (in seconds) versus the relative objective value difference. We run LR with $C = 1$. Top: using one node; bottom: using two nodes.

C. mapPartitions *and* map

In Figure 3, we show the comparison between mapPartitions and map using 16 nodes. We mentioned in Section IV-C that the communication cost of the two implementations are basically the same, so the longer running time of map implies its higher computation cost. This result is consistent with the analysis in Section IV-C. As the consequence of its better efficiency, we apply mapPartitions in Spark LIBLINEAR.

D. *Broadcast Variables and the* coalesce *Function*

The implementations with and without broadcast variables and the coalesce function are compared in Figure 4. The results indicate that using broadcast variables and the coalesce function significantly reduces the running time on data sets with higher feature dimensions. The observation validates our

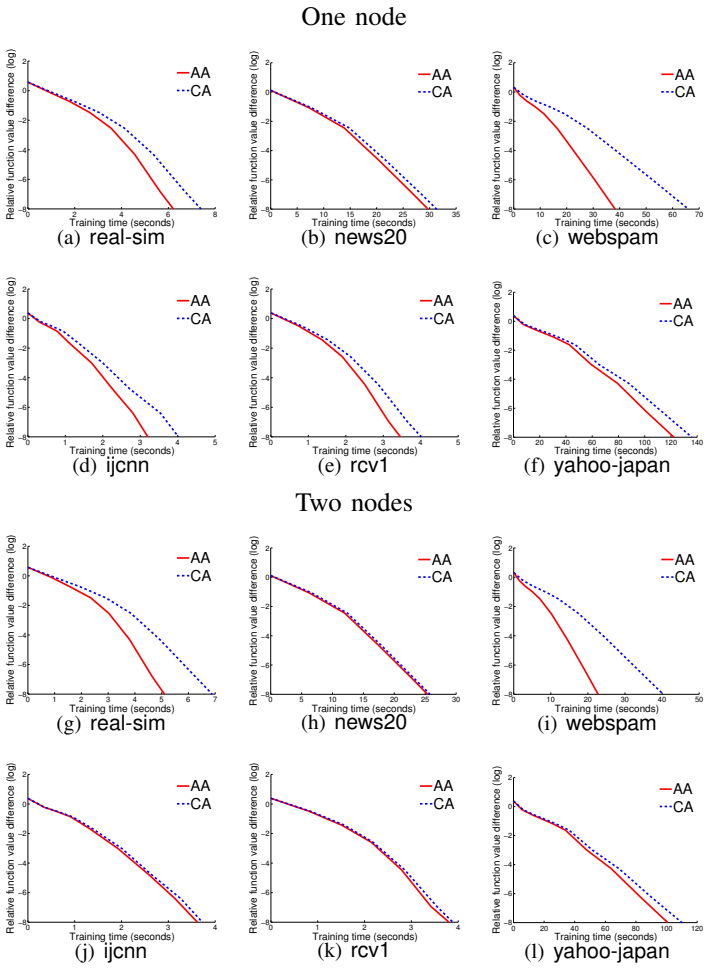


Fig. 2. Encapsulation implementations: We present running time (in seconds) versus the relative objective value difference. Top: using one node; bottom: using two nodes.

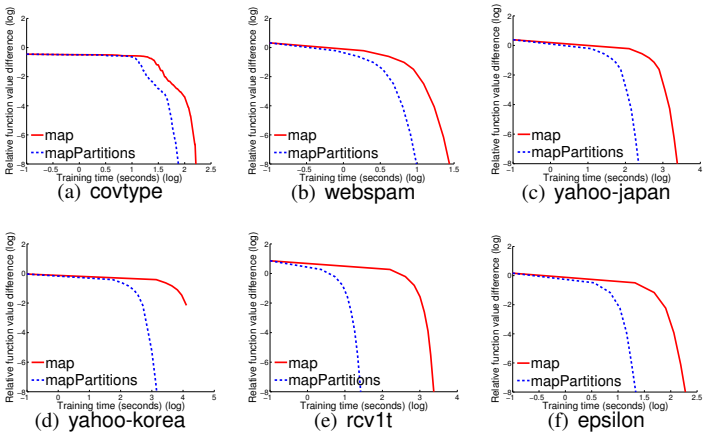


Fig. 3. map and mapPartitions: We present running time (in seconds, log scaled) versus the relative objective value difference. We run LR with $C = 1$ on 16 nodes.

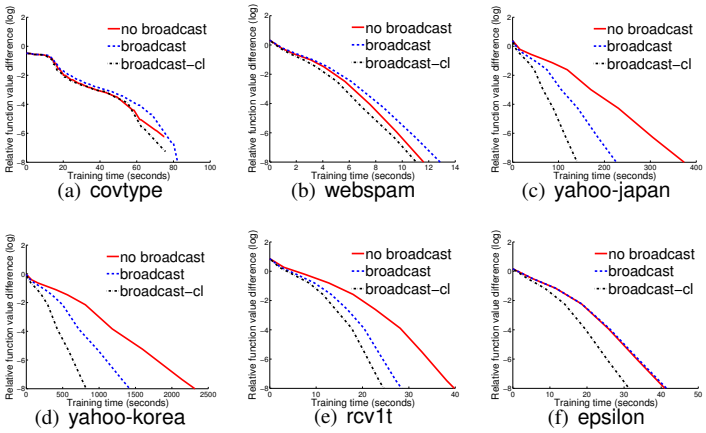


Fig. 4. Broadcast variables and coalesce: We present running time (in seconds) versus the relative objective value difference. We run LR with $C = 1$ on 16 nodes. Note that broadcast-cl represents the implementation with both broadcast variables and the coalesce function.

discussion in Sections IV-E and IV-F, which indicates high communication cost for data with many features. However, for data with few features like covtype and webspam, adopting broadcast variables slightly degrades the efficiency because the communication cost is low and broadcast variables introduce some overheads. Regarding the coalesce function, it is beneficial for all data sets.

E. Analysis on Scalability

To examine the relation between the training speed and the number of nodes used, we vary the number of nodes from 2 to 4, 8 and 16 to compare the training time. The results are shown as N-2, N-4, N-8 and N-16 in Figure 5. To check the speed-up of using different number of nodes, we run the experiments with one virtual CPU per node. N-2 does not appear in Figure 5(f) because the epsilon data set is too large to fit in the memory of two nodes. We observe that the performances of covtype, webspam and epsilon improve when the node size increases.

For data sets yahoo-japan, yahoo-korea and rcv1 t respectively shown in Figures 5(c), 5(d),and 5(e), as the number of nodes increases (i.e., 4 to 8 nodes, 8 to 16 nodes and 8 to 16 nodes), the training efficiency degrades. The reason is that these data sets possess higher feature dimension, so the communication cost grows faster when the number of nodes increases. In Section VI-G, we observe that MPI LIBLINEAR also encounters this problem. The communication bottleneck may be improved by the feature-wise distributed approach discussed in [9]. The feasibility of applying this approach on Spark will be investigated in the near future.

F. Comparing with MLlib

After studying implementation issues discussed in IV, we compare Spark LIBLINEAR with some existing packages. We first examine the performance of Spark LIBLINEAR with the LR solver in MLlib. Note that 16 nodes are used in this experiment. We follow the default setting of MLlib to set I_t as the whole training data, so the SG method becomes a gradient descent (GD) method in our experiments. We present the result with the fastest function value decrease among using different initial learning rates η_0 from $\{10^{-5}, 10^{-4}, \ldots, 10^{5}\}$. The results are shown in Figure 6. We can see that the convergence of MLlib is rather slow in comparison with Spark LIBLINEAR. The reason is that the GD method is known to

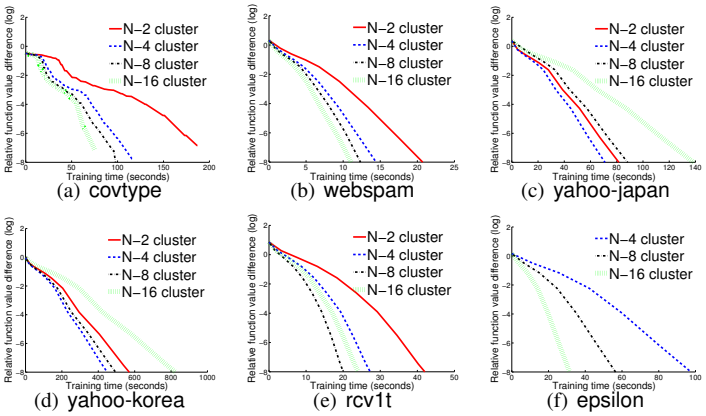


Fig. 5. Scalability: We present running time (in seconds) versus the relative objective value difference. We run LR with $C = 1$.

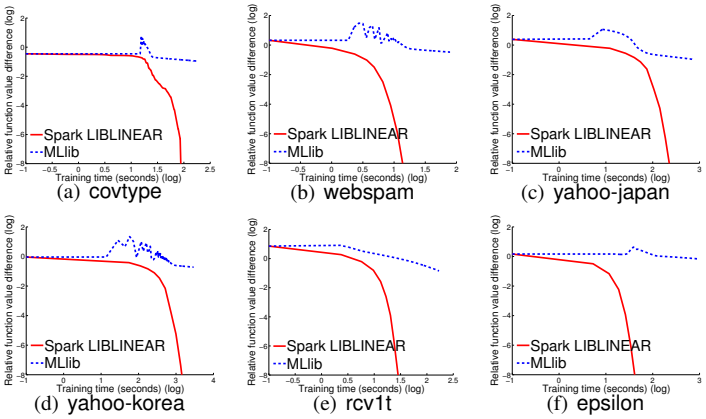


Fig. 6. Comparison with MLlib: We present running time (in seconds, log scale) versus the relative objective value difference. We run LR with $C = 1$ on 16 nodes.

have slow convergence, while TRON enjoys fast quadratic local convergence for LR [13]. Note that as MLlib requires more iterations to converge, the communication cost is also higher. This then exacerbates its inefficiency.

G. Comparison of Spark LIBLINEAR and MPI LIBLINEAR

We continue to compare the performance of Spark LIBLINEAR and MPI LIBLINEAR. In Figure 7, we show the results of using 2, 4, and 8 nodes for each data set. We do not present results on epsilon with two nodes because of the memory capacity issue mentioned in Section VI-E. Note that Spark LIBLINEAR-m denotes using Spark LIBLINEAR with multiple cores on each node, and the other two approaches apply one core per node. Spark realizes multi-core parallelism by handling multiple partitions in the same node simultaneously. To fully exploit parallelism, the number of cores used on each node should not exceed p/s. Therefore we use four cores per node for the multi-core approach. In contrast, there is no multi-core parallelization mechanism in MPI, so we must modify MPI LIBLINEAR with additional libraries to activate multi-core computation. We therefore do not consider MPI LIBLINEAR with multiple cores in this experiment.

For different settings of Spark LIBLINEAR, we observe that using multiple cores is not beneficial on yahoo-japan and yahoo-korea. A careful profiling shows that the bottleneck of the training time on these data sets is communication and using more cores does not reduce this cost. MPI LIBLINEAR also suffers from the increase of training time for larger clusters in the high dimensional data yahoo-japan. See also the discussion in the end of Section VI-E.

In the comparison between MPI LIBLINEAR and Spark LIBLINEAR with one core, MPI LIBLINEAR is faster in almost all cases. The reason is that to support fault tolerance, Spark LIBLINEAR incurs additional cost in maintaining RDDs. Also, the different programming languages used in the two packages affect their efficiency. The only exception is the data set epsilon. In Figure 7(f), one-core Spark LIBLINEAR is competitive with MPI LIBLINEAR, while multi-core Spark LIBLINEAR is significantly faster than the other two approaches. A possible reason is that in the training of the dense data epsilon, most of the training time is spent on the matrix-vector products, so other factors become less significant and using more cores is effectively accelerates this computation. We also notice that when only two nodes are used and therefore the communication cost is low, the training time of multi-core Spark LIBLINEAR is similar to that of MPI LIBLINEAR in Figures 7(b) and 7(e). Although MPI LIBLINEAR is generally faster than Spark LIBLINEAR, Spark LIBLINEAR has the important advantage of supporting fault tolerance.

VII. Discussions and Conclusions

In this work, we consider a distributed TRON algorithm on Spark for training LR and linear SVM with large-scale data. Many important implementation issues affecting the training time are thoroughly studied with careful empirical examinations. Our implementation is efficient with fault tolerance provided by Spark. It is possible to use hybrid methods that generate a better initial $\boldsymbol{w}$ for TRON such as the algorithm considered in [17] to reduce the training time. We leave it as an interesting future work to examine the performance of such approaches. In the experiments, we observe that the communication cost becomes the training bottleneck when the feature dimension is huge. We plan to investigate feature-wise distributed algorithms on Spark to tackle this problem in the near future, as it is shown in [9] that such an approach may be more efficient in training high dimensional data. Based on this research, we release an efficient and easy to use tool for the Spark community.

Acknowledgments

This work was supported in part by the National Science Council of Taiwan. The authors thank Xiangrui Meng and Mingfeng Huang for helpful discussion.

References

[1] B. E. Boser, I. Guyon, and V. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. ACM Press, 1992, pp. 144–152.

[2] C. Cortes and V. Vapnik, "Support-vector network," *Machine Learning*, vol. 20, pp. 273–297, 1995.

[3] G.-X. Yuan, C.-H. Ho, and C.-J. Lin, "Recent advances of large-scale linear classification," *Proceedings of the IEEE*, vol. 100, no. 9, pp. 2584–2603, 2012. [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/papers/survey-linear.pdf

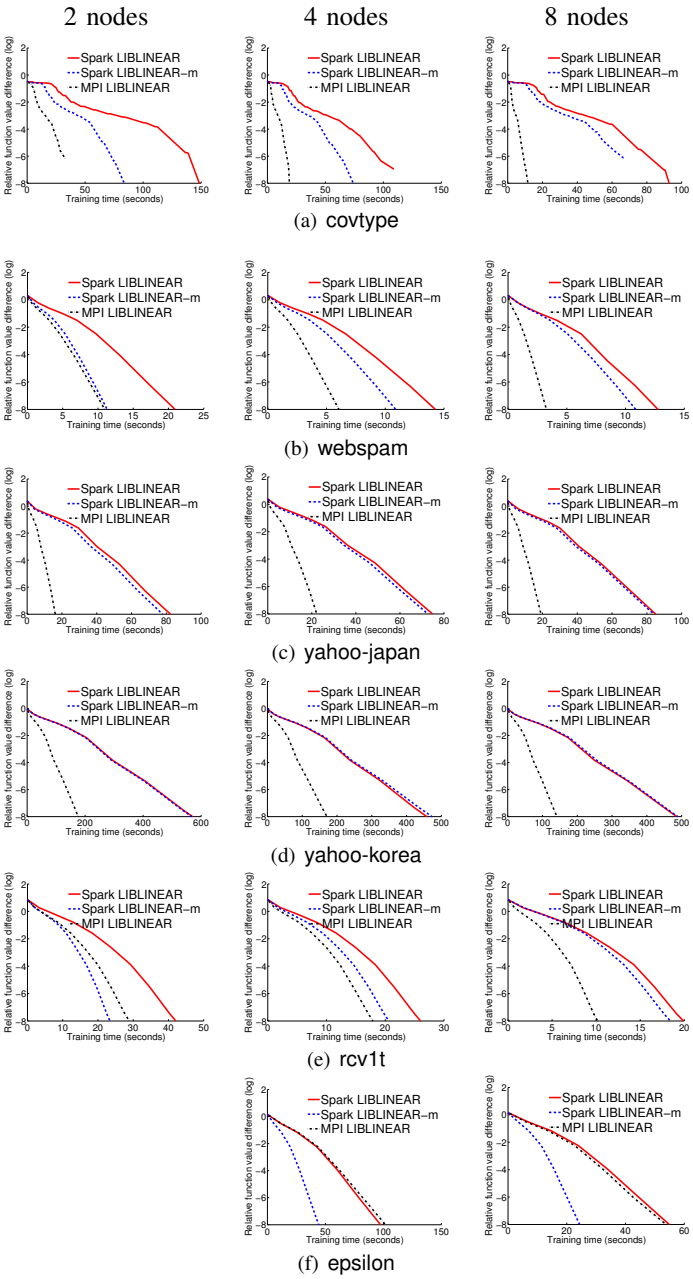


Fig. 7. Comparison with MPI LIBLINEAR: We present running time (in seconds) versus the relative objective value difference. We run LR with $C = 1$.

[4] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[7] M. Snir and S. Otto, *MPI-The Complete Reference: The MPI Core*. Cambridge, MA, USA: MIT Press, 1998.

[8] C.-J. Lin and J. J. Moré, "Newton's method for large-scale bound constrained problems," *SIAM Journal on Optimization*, vol. 9, pp. 1100–1127, 1999.

[9] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin, "Distributed Newton method for regularized logistic regression," Department of Computer Science and Information Engineering, National Taiwan University, Tech. Rep., 2014.

[10] 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*, vol. 9, pp. 1871–1874, 2008. [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf

[11] T. White, *Hadoop: The definitive guide*, 2nd ed. O'Reilly Media, 2010.

[12] D. Borthakur, "HDFS architecture guide," 2008.

[13] C.-J. Lin, R. C. Weng, and S. S. Keerthi, "Trust region Newton method for large-scale logistic regression," *Journal of Machine Learning Research*, vol. 9, pp. 627–650, 2008. [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/papers/logistic.pdf

[14] O. L. Mangasarian, "A finite Newton method for classification," *Optimization Methods and Software*, vol. 17, no. 5, pp. 913–929, 2002.

[15] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima, 2008.

[16] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "The Scala language specification," 2004.

[17] A. Agarwal, O. Chapelle, M. Dudik, and J. Langford, "A reliable effective terascale linear learning system," *Journal of Machine Learning Research*, vol. 15, pp. 1111–1133, 2014.