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

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

Cheng-Hao Tsai
Dept. of Computer Science
National Taiwan Univ., Taiwan
r @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

—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 consid- ering 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.

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

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

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

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

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

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 $\|\boldsymbol{r}^i\| \le \xi_t \|\nabla f(\boldsymbol{w}^t)\|$, output $\boldsymbol{d}^t = \bar{\boldsymbol{d}}^i$ and stop.
4: Compute

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

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 $\|\bar{\boldsymbol{d}}^{i+1}\| \ge \Delta_t$, 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.

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

where $\mathrm{diag}(\cdot)$ 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.

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

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

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

where

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

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

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

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

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, ∇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 $\boldsymbol{w}^0, \Delta_0, \eta$, .
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 $\|\nabla f(\boldsymbol{w}^t)\| <$, 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

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

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 ← 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 ← 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 ← 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 programming, 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.

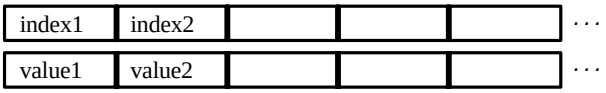
B. Data Encapsulation

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