Scalable and Parallel Boosting with MapReduce

Indranil Palit and Chandan K. Reddy, *Member*, *IEEE*

Abstract—In this era of data abundance, it has become critical to process large volumes of data at much faster rates than ever before. Boosting is a powerful predictive model that has been successfully used in many real-world applications. However, due to the inherent sequential nature, achieving scalability for boosting is nontrivial and demands the development of new parallelized versions which will allow them to efficiently handle large-scale data. In this paper, we propose two parallel boosting algorithms, ADABOOST.PL and LOGITBOOST.PL, which facilitate simultaneous participation of multiple computing nodes to construct a boosted ensemble classifier. The proposed algorithms are competitive to the corresponding serial versions in terms of the generalization performance. We achieve a significant speedup since our approach does not require individual computing nodes to communicate with each other for sharing their data. In addition, the proposed approach also allows for preserving privacy of computations in distributed environments. We used *MapReduce* framework to implement our algorithms and demonstrated the performance in terms of classification accuracy, speedup and scaleup using a wide variety of synthetic and real-world data sets.

Index Terms—Boosting, parallel algorithms, classification, distributed computing, MapReduce.

1 INTRODUCTION

In several scientific and business applications, it has become a common practice to gather information that contains millions of training samples with thousands of features. In many such applications, data are either generated or gathered everyday at an unprecedented rate. To efficiently handle such large-scale data, faster processing and optimization algorithms have become critical in these applications. Hence, it is vital to develop new algorithms that are more suitable for parallel architectures. One simple approach could be to deploy a **s**ingle inherently parallelizable data mining **p**rogram to **m**ultiple **d**ata (SPMD) on multiple computers. However, for algorithms that are not inherently parallelizable in nature, redesigning to achieve parallelization is the only alternative solution.

Ensemble classifiers [1], [2], [3] are reliable predictive models that use multiple learners to obtain better predictive performance compared to other methods [4]. Boosting is a popular ensemble method that has been successfully used in many real-world applications. However, due to its inherent sequential nature, achieving scalability for boosting is not easy and demands considerable research attention for developing new parallelized versions that will allow them to efficiently handle large-scale data. It is a challenging task to parallelize boosting since they iteratively learn weak classifiers with respect to a distribution and add them to a final strong classifier. Thus, weak learners in next iterations give more focus to the samples that previous weak learners misclassified. Such a dependent iterative setting in boosting makes it inherently a serial algorithm. The task of making iterations independent of each other and thus leveraging boosting for parallel architectures is nontrivial. In this work, we solve such an interdependent problem with a different strategy.

In this paper, we propose two novel parallel boosting algorithms, ADABOOST.PL (Parallel ADABOOST) and LOGITBOOST.PL (Parallel LOGITBOOST). We empirically show that, while maintaining a competitive accuracy on the test data, the algorithms achieve a significant speedup compared to the respective baseline (ADABOOST or LOGITBOOST) algorithms implemented on a single machine. Both the proposed algorithms are designed to work in cloud environments where each node in the computing cloud works only on a subset of the data. *The combined effect of all the parallel working nodes is a boosted classifier model induced much faster and with a good generalization capability*.

The proposed algorithms achieve parallelization in both time and space with minimal amount of communication between the computing nodes. Parallelization in space is also important because of the limiting factor posed by the memory size. Large data sets, that cannot fit into the main memory, are often required to swap between the main memory and the (slower) secondary storage, introducing latency cost which sometimes will even diminish the speedup gained by the parallelization in time. For our implementation, we used *MapReduce* [5] framework, which is a popular model for distributed computing that abstracts away many of the difficulties of cluster management such as data partitioning, scheduling tasks, handing machine failures, and intermachine communication. To demonstrate the superiority of the proposed algorithms, we compared our results to the MULTBOOST [6] algorithm, which is a variant of ADABOOST and the only other parallel boosting algorithm available in the literature that can achieve parallelization both in space and time.

- *I. Palit is with the Department of Computer Science and Engineering, University of Notre Dame, 222 Cushing Hall, Notre Dame, IN 46556. E-mail: ipalit@nd.edu.*
- *C.K. Reddy is with the Department of Computer Science, Wayne State University, 5057 Woodward Avenue, Suite 14109.4, Detroit, MI 48202. E-mail: reddy@cs.wayne.edu.*

Manuscript received 21 Mar. 2011; revised 29 Aug. 2011; accepted 4 Sept. 2011; published online 30 Sept. 2011.
Recommended for acceptance by J. Haritsa.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2011-03-0145.
Digital Object Identifier no. 10.1109/TKDE.2011.208.

The primary contributions of our work are as follows: 1) We propose a new parallel framework for boosting algorithm that achieves parallelization both in time (significantly reduced computational time) and space (large data sets are distributed among various machines and thus each machine handles far less amount of data). We achieve this by making the parallel working nodes' computations independent from each other thus minimizing the communication cost between the workers during parallelization. 2) We provide theoretical guarantees of convergence for the ADABOOST.PL algorithm. 3) We efficiently implement these algorithms using MapReduce architecture on the *Amazon EC2*[1] cloud environment and experimentally demonstrate their superiority in terms of performance metrics such as prediction accuracy, speedup, and scaleup.

The rest of the paper is organized as follows: Section 2 describes some of the earlier works related to our problem. In Section 3, we propose our ADABOOST.PL algorithm along with the proof of its convergence. Section 4 describes LOGITBOOST.PL algorithm. Section 5 explains the *MapReduce* framework and provides the implementation details of the proposed algorithms. Section 6 demonstrates the experimental results and shows the comparisons with MULTBOOST. Finally, Section 7 concludes our discussion along with some future research directions.

2 Related Work

ADABOOST is one of the earliest and most popular boosting algorithm proposed in the mid 1990s [7]. It's simple intuitive algorithmic flow combined with its dramatic improvement in the generalization performance makes it one of the most powerful ensemble methods. A clear theoretical explanation of its performance is well described in [8], where boosting in a two class setting is viewed as an additive logistic regression model. LOGITBOOST is another widely used boosting algorithm which is proposed using additive modeling and is shown to exhibit more robust performance especially in the presence of noisy data.

There had been some prior works proposed in the literature for accelerating ADABOOST. These methods essentially gain acceleration by following one of the two approaches: 1) by limiting the number of data points used to train the base learners, or 2) by cutting the search space by using only a subset of the features. In order to ensure convergence, both of these approaches increase the number of iterations. However, as the required time for each iteration is less due to smaller data (or feature) size, the overall computational time by using such methods can be significantly reduced. The basic idea of the former approach is to train a base learner only on a small subset of randomly selected data instead of the complete weighted data by using the weight vector as a discrete probability distribution [7]. FILTERBOOST [9] is a recent algorithm of the same kind, based on a modification [10] of ADABOOST designed to minimize the logistic loss. FILTERBOOST assumes an oracle that can produce unlimited number of labeled samples and in each boosting iteration, the oracle generates sample points that the base learner can either accept or reject. A small subset of the accepted points are used to train the base learner.

1. http://aws.amazon.com/ec2/.

Following the latter approach for accelerating ADABOOST, Escudero et al. [11] proposed LAZYBOOST which utilizes several feature selection and ranking methods. In each boosting iteration, it chooses a fixed-size random subset of features and the base learner is trained only on this subset. Another fast boosting algorithm in this category was proposed by Busa-Fekete and Kégl [12], which utilizes multiple-armed bandits (MAB). In the MAB-based approach, each arm represents a subset of the base classifier set. One of these subsets is selected in each iteration and then the boosting algorithm searches only this subset instead of optimizing the base classifier over the entire space. However, *none of these works described so far explore the idea of accelerating boosting in a parallel or distributed setting and thus their performance is limited by the resources of a single machine.*

The strategy of parallelizing the weak learners instead of parallelizing the ensemble itself has been investigated earlier. Recently, Wu et al. [13] proposed an ensemble of C4.5 classifiers based on MapReduce called MReC4.5. By providing a series of serialization operations at the model level, the classifiers built on a cluster of computers or in a cloud computing platform could be used in other environments. PLANET [14] is another recently proposed framework for learning classification and regression trees on massive data sets using MapReduce. *These approaches are specific to the weak learners (such as tree models) and hence do not appear as a general framework for ensemble methods such as boosting.*

Despite these efforts, there has not been any significant research to parallelize the boosting algorithm itself. Earlier versions of parallelized boosting [15] were primarily designed for tightly coupled shared memory systems and hence is not applicable in a distributed cloud computing environment. Fan et al. [16] proposed boosting for scalable and distributed learning, where each classifier was trained using only a small fraction of the training set. In this distributed version, the classifiers were trained either from random samples (r-sampling) or from disjoint partitions of the data set (d-sampling). This work primarily focused on parallelization in space but not in time. Hence, even though this approach can handle large data by distributing among the nodes, the goal of faster processing time is not achieved by this approach. Gambs et al. [6] proposed MULTBOOST algorithm which allows participation of two or more working nodes to construct a boosting classifier in a privacy-preserving setting. Though originally designed for preserving privacy of computation, MULTBOOST's algorithmic layout can fit into a parallel setting. It can achieve parallelism both in space and time by requiring the nodes to have separate data and by enabling the nodes to compute without knowing about other workers' data. Hence, we compared the performance of MULTBOOST to our algorithm in this paper.

However, the main problem of these above-mentioned approaches is that they are suitable for low latency intercomputer communication environments such as traditional shared memory architecture or single machine multiple processors systems and are not suitable for a distributed cloud environment where usually the communication cost is higher. A significant portion of the time is expended for communicating information between the computing nodes rather than the actual computation. In our approach, *we overcome this limitation by making the*

workers' computations independent from each other thus minimizing these communications.

3 Parallelization of AdaBoost

In this section, we will first review the standard ADABOOST algorithm [7] and then propose our parallel algorithm ADABOOST.PL. We will also theoretically demonstrate the convergence of the proposed algorithm.

3.1 AdaBoost

ADABOOST [7] is an ensemble learning method which iteratively induces a strong classifier from a pool of weak hypotheses. During each iteration, it employs a simple learning algorithm (called the base classifier) to get a single learner for that iteration. The final ensemble classifier is a weighted linear combination of these base classifiers where each of them casts their weighted "votes." These weights correspond to the correctness of the classifiers, i.e., a classifier with lower error rate gets higher weight. The base classifiers have to be slightly better than a random classifier and hence, they are also called as weak classifiers. Simple learners such as decision stumps (decision trees with only one nonleaf node) often perform well for ADABOOST [17]. Assuming that the attributes in the data set are real valued, we will need three parameters to express a decision stump: 1) the index of the attribute to be tested (j), 2) the numerical value of the test threshold (θ), and 3) the sign of the test $\{+1, -1\}$. For example,

$$h_{j,\theta,+}(x) = \begin{cases} +1 & \text{if } x^j < \theta \\ -1 & \text{otherwise,} \end{cases} \quad (1)$$

where, x^j is the value of the jth attribute of the data object x. For simplicity, we used decision stumps as weak learners throughout this paper, though any weak learner which produces decision in a form of real value can be fitted into our proposed parallel algorithm. From (1), the negation of h can be defined as follows:

$$-h_{j,\theta,+}(x) = h_{j,\theta,-}(x) = \begin{cases} -1 & \text{if } x^j < \theta \\ +1 & \text{otherwise.} \end{cases} \quad (2)$$

The pseudocode for ADABOOST is described in Algorithm 1. Let the data set $D_n = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where each example $x_i = (x_i^1, x_i^2, \ldots, x_i^d)$ is a vector with d attribute values and each label $y_i \in \{+1, -1\}$. The algorithm assigns weights $w^t = \{w_1^t, w_2^t, \ldots, w_n^t\}$ for all the samples in D_n, where $t \in \{1, 2, \ldots, T\}$ and T is the total number of boosting iterations. Before starting the first iteration, these weights are uniformly initialized (line 1) and are updated in every consecutive iteration (lines 7-10). It is important to note that, for all t, $\sum_{i=1}^n w_i^t = 1$. At each iteration, a weak learner function is applied to the weighted version of the data which then returns an optimal weak hypothesis $h^{(t)}$ (line 3). This weak hypothesis minimizes the weighted error given by

$$\epsilon_- = \sum_{i=1}^{n} w_i^t I\{h^{(t)}(x_i) \neq y_i\}. \quad (3)$$

Here, $I\{A\}$ denotes an indicator function whose value is 1 if A is true and 0 otherwise. The weak learner function always ensures that it will find an optimal $h^{(t)}$ with $\epsilon < 1/2$. If there exists any h with $\epsilon > 1/2$ then according to (2), $-h$ will have a weighted error of $(1 - \epsilon)$ which is less than $1/2$. Hence, the optimal weak learner will always induce $-h$ instead of h. This property of ϵ having a value less than $1/2$, increases the weight of misclassified samples and decreases the weight of correctly classified samples. Hence, for the next iteration, the weak classifier focuses more on the samples that were previously misclassified. At each iteration, a weight (α^t) is assigned to the weak classifier (line 5). At the end of T iterations, the algorithm returns the final classifier H which is a weighted average of all the weak classifiers. The sign of H is used for the final prediction.

Algorithm 1. ADABOOST(D_n, T)
Input: Training set of n samples (D_n)
Number of boosting iterations (T)
Output: The final classifier (H)
Procedure:
1: $w^1 \leftarrow (\frac{1}{n}, \ldots, \frac{1}{n})$
2: **for** $t \leftarrow 1$ to T **do**
3: $h^{(t)} \leftarrow$ LEARNWEAKCLASSIFIER(w^t)
4: $\epsilon_- \leftarrow \sum_{i=1}^n w_i^t I\{h^{(t)}(x_i) \neq y_i\}$
5: $\alpha^t \leftarrow \frac{1}{2} \ln\left(\frac{1-\epsilon_-}{\epsilon_-}\right)$
6: **for** $i \leftarrow 1$ to n **do**
7: **if** $h^{(t)}(x_i) \neq y_i$ **then**
8: $w_i^{t+1} \leftarrow \frac{w_i^t}{2\epsilon_-}$
9: **else**
10: $w_i^{t+1} \leftarrow \frac{w_i^t}{2(1-\epsilon_-)}$
11: **end if**
12: **end for**
13: **end for**
14: return $H = \sum_{t=1}^T \alpha^t h^{(t)}$

3.1.1 Computational Complexity

The computational complexity of ADABOOST depends on the weak learner algorithm in line 3. The rest of the operations can be performed in $\Theta(n)$. The cost of finding the best decision stump is $\Theta(dn)$ if the data samples are sorted in each attribute. Sorting all the attributes will take $\Theta(dn \log n)$ time and this has to be done only once before starting the first iteration. Hence, the overall cost of the T iterations is $\Theta(dn(T + \log n))$.

3.2 AdaBoost.PL

The proposed ADABOOST.PL employs two or more computing workers to construct the boosting classifiers; each of the worker has access to only a specific subset of training data. The pseudocode of ADABOOST.PL is given in Algorithm 2.

For a formal description of ADABOOST.PL, let $D_{n^p}^p = \{(x_1^p, y_1^p), (x_2^p, y_2^p), \ldots, (x_{n^p}^p, y_{n^p}^p)\}$ is the data set for pth worker where $p \in \{1, \ldots, M\}$ and n^p is the number of data points in pth worker's data set. The workers compute the ensemble classifier H^p by completing all the T iterations of the standard ADABOOST (Algorithm 1) on their respective data sets (line 2). H^p is defined as follows:

$$\{(h^{p(1)}, \alpha^{p(1)}), (h^{p(2)}, \alpha^{p(2)}), \ldots, (h^{p(T)}, \alpha^{p(T)})\},$$

where $h^{p(t)}$ is the weak classifier of the pth worker at the tth iteration and $\alpha^{p(t)}$ is the corresponding weight of that

weak classifier. The worker then reorders the weak classifiers, $h^{p(t)}$, with increasing order of $\alpha^{p(t)}$ (line 3). This new ordering H^{p^*} is expressed as follows:

$$\{(h^{p^*(1)}, \alpha^{p^*(1)}), (h^{p^*(2)}, \alpha^{p^*(2)}), \ldots, (h^{p^*(T)}, \alpha^{p^*(T)})\}.$$

If $\alpha^{p(k)} = min\{\alpha^{p(t)} | t \in \{1, 2, \ldots, T\}\}$, then $\alpha^{p^*(1)} = \alpha^{p(k)}$ and $h^{p^*(1)} = h^{p(k)}$. Now, the reordered $h^{p^*(t)}$s are considered for merging in the rounds of the final classifier. Note that the number of rounds for the final classifier is the same as the number of iterations of the workers' internal ADABOOST. However, the tth round of the final classifier does not necessarily merge the tth iteration results of the workers. For example, $h^{(t)}$ is formed by merging $\{h^{1^*(t)}, \ldots, h^{M^*(t)}\}$ (line 6) where, these weak classifiers do not necessarily come from the tth iteration of the workers. *The intuition of sorting the workers' weak classifiers with respect to their weights is to align classifiers with similar correctness in the same sorted level.* This is a critical component of the proposed framework since it will ensure that like-minded classifiers will be merged during each boosting iteration.

Algorithm 2. ADABOOST.PL($D^1_{n^1}, \ldots, D^M_{n^M}$, T)
Input: The training sets of M workers ($D^1_{n^1}, \ldots, D^M_{n^M}$)
Number of boosting iterations (T)
Output: The final classifier (H)
Procedure:
1: **for** $p \leftarrow 1$ to M **do**
2: $H^p \leftarrow \text{ADABOOST}(D^P_{N^P}, T)$
3: $H^{p^*} \leftarrow$ the weak classifiers in H^p sorted w.r.t. $\alpha^{p(t)}$
4: **end for**
5: **for** $t \leftarrow 1$ to T **do**
6: $h^{(t)} \leftarrow \text{MERGE}(h^{1^*(t)}, \ldots, h^{M^*(t)})$
7: $\alpha^t \leftarrow \frac{1}{M}\sum_{p=1}^{M} \alpha^{p^*(t)}$
8: **end for**
9: return $H = \sum_{t=1}^{T} \alpha^t h^{(t)}$

The merged classifier, $h^{(t)}$ is a *ternary classifier*, a variant of weak classifier proposed by Schapire and Singer [18] which along with $+1$ and -1 might also return 0 as a way of abstaining from answering. It takes a simple majority vote among the workers weak classifiers:

$$h^{(t)}(x) = \begin{cases} \text{sign}\left(\sum_{p=1}^{M} h^{p^*(t)}(x)\right) & \text{if } \sum_{p=1}^{M} h^{p^*(t)}(x) \neq 0 \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

The ternary classifier will answer "0" if equal number of positive and negative predictions are made by the workers' weak classifiers. Otherwise, it will answer the majority prediction. It should be noted that the ternary classifier provides the algorithm the freedom of using any number of available working nodes (odd or even) in the distributed setting. In line 7, the weights of the corresponding classifiers are averaged to get the weight of the ternary classifier. After all the ternary classifiers for T rounds are generated, the algorithm returns their weighted combination as the final classifier. The strategy of distributing the data and computations among the working nodes and making the tasks of the nodes independent of each other enables much faster processing of our algorithm while resulting in a competitive generalization performance (which is shown in our experimental results).

3.2.1 Computational Complexity

In a distributed setting, where M workers participate parallelly and the data are distributed evenly among the workers, the computational cost for ADABOOST.PL is $\Theta(\frac{dn}{M}\log\frac{n}{M} + \frac{Tdn}{M})$. The sorting of the T weak classifiers (line 3) will have an additional cost of $\Theta(T\log T)$ time, which becomes a constant term if T is fixed.

3.3 Convergence of ADABOOST.PL

ADABOOST.PL sorts the worker's classifiers with respect to the weights ($\alpha^{p(t)}$) and then merges them based on the new reordering. We will now demonstrate this merging of classifiers from different iterations will ensure algorithm's convergence. As the definition of base classifier has been changed to a ternary classifier, the definition of the weighted error described in (3) must be redefined as follows:

$$\epsilon_- = \sum_{i=1}^{n} w_i^t I\{h^{(t)}(x_i) = -y_i\}. \quad (5)$$

The weighted rate of correctly classified samples is given as follows:

$$\epsilon_+ = \sum_{i=1}^{n} w_i^t I\{h^{(t)}(x_i) = y_i\}. \quad (6)$$

Gambs et al. [6] showed that, any boosting algorithm will eventually converge if the weak classifiers of the iterations satisfy the following condition:

$$\epsilon_+ > \epsilon_-. \quad (7)$$

We will now show that ADABOOST.PL satisfies this condition when the number of workers is two. Let the ith iteration weak classifier $h^{A(i)}$ of worker A is merged with the jth iteration weak classifier $h^{B(j)}$ of worker B to form the merged classifier $h^{(k)}$ for the kth round. $w^A = \{w_1^A, w_2^A, \ldots, w_{n^A}^A\}$ is the state of the weight vector (during ith iteration) of worker A's data points. Similarly, w^B can be defined as the weight vector state during the jth iteration. So, the weighted errors and the weighted rate of correctly classified points for $h^{A(i)}$ are

$$\epsilon_-^A = \sum_{l=1}^{n^A} w_l^A I\{h^{A(i)}(x_l^A) = -y_l^A\} \quad (8)$$

$$\epsilon_+^A = \sum_{l=1}^{n^A} w_l^A I\{h^{A(i)}(x_l^A) = y_l^A\}. \quad (9)$$

ϵ_-^B and ϵ_+^B can also be defined similarly for $h^{B(j)}$. Let us also define

$$\omega_-^A = \sum_{l=1}^{n^A} w_l^A I\{h^{(k)}(x_l^A) = -y_l^A\} \quad (10)$$

$$\omega_+^A = \sum_{l=1}^{n^A} w_l^A I\{h^{(k)}(x_l^A) = y_l^A\}. \quad (11)$$

Similarly, we can define ω_-^B and ω_+^B. It should be noted that there is difference between ϵ^A and ω^A. ϵ^A is defined for A's weak classifier and ω^A is defined for the merged

classifier. Using these notations, the weighted error and the weighted rate of correctly classified points for $h^{(k)}$ can be defined as follows: $\epsilon_-^* = \omega_-^A + \omega_-^B$ and $\epsilon_+^* = \omega_+^A + \omega_+^B$. It should be noted that these values are not normalized. ϵ_-^* and ϵ_+^* might exceed 1 because both $\sum_{l=1}^{n^A} w_l^A$ and $\sum_{l=1}^{n^B} w_l^B$ are equal to 1. These weight vectors were initialized by the corresponding worker and through out all the ADABOOST iterations they always sum up to 1. Hence, the normalized weighted error and the normalized rate of correctly classified points for the merged classifier will be

$$\epsilon_- = \frac{\omega_-^A + \omega_-^B}{2} \tag{12}$$

$$\epsilon_+ = \frac{\omega_+^A + \omega_+^B}{2}. \tag{13}$$

Theorem 1. *If $h^{A(i)}$ and $h^{B(j)}$ are both optimal, then $\epsilon_+ \geq \epsilon_-$.*

Proof. According to the definition of ternary classifier, $h^{(k)}$ abstains when $h^{A(i)}$ does not agree with $h^{B(j)}$. Hence, from (2), we can say that $h^{(k)}$ abstains when $h^{A(i)}$ agrees with $-h^{B(j)}$. Weighted error of $-h^{B(j)}$ on A's data can be divided into two mutually exclusive regions of $D_{n^A}^A$: 1) the region where $h^{(k)}$ abstains and 2) the region where $h^{(k)}$ does not abstain.

In the first region $h^{A(i)}$ agrees with $-h^{B(j)}$. Hence, in this region the weighted error of $-h^{B(j)}$ is equal to the weighted error of $h^{A(i)}$ which is $(\epsilon_-^A - \omega_-^A)$.

In the second region, $h^{A(i)}$ does not agree with $-h^{B(j)}$. Hence, in this region, the weighted error of $-h^{B(j)}$ is equal to the weighted rate of correctly classified points of $h^{A(i)}$ which is ω_+^A.

Hence, the weighted error of $-h^{B(j)}$ on $D_{n^A}^A$ is $(\epsilon_-^A + \omega_+^A - \omega_-^A)$. If $\omega_+^A < \omega_-^A$, then the weighted error of $-h^{B(j)}$ on $D_{n^A}^A$ will be less than ϵ_-^A. Note that ϵ_-^A is the error for $h^{A(i)}$. This contradicts the optimality of $h^{A(i)}$ on $D_{n^A}^A$. So, it is proved that $\omega_+^A \geq \omega_-^A$. Similarly, it can be shown that, $\omega_+^B \geq \omega_-^B$. Adding the last two inequalities and dividing both sides by 2 will give us $\epsilon_+ \geq \epsilon_-$. □

According to Theorem 1, we can say that, in a two worker environment, the merged classifiers in ADABOOST.PL will satisfy $\epsilon_+ \geq \epsilon_-$. ADABOOST.PL can discard any merged classifier with $\epsilon_+ = \epsilon_-$ and thus can satisfy the sufficient condition for convergence described by the inequality given in (7). ADABOOST.PL will only fail when all the merged classifiers have $\epsilon_+ = \epsilon_-$ which is very unlikely to happen. We were not able to extend (7) when the number of workers is more than two. That is, we could not theoretically prove that the merged classifier in such cases would always satisfy the necessary condition for convergence. However, in our experiments, we observed that merged classifiers almost never violated (7). In the rare event when the merged classifier violates the condition, we can simply discard it and proceed without having it within the pool of the final classifier.

4 PARALLELIZATION OFLOGITBOOST

In this section, we describe our proposed LOGITBOOST.PL algorithm. First, we will briefly discuss the standard LOGITBOOST [8] algorithm.

4.1 LOGITBOOST

LOGITBOOST [8] is a powerful boosting method that is based on additive logistic regression model. Unlike ADABOOST, it uses regression functions instead of classifiers and these functions output real values in the same form as prediction. The pseudocode for LOGITBOOST is described in Algorithm 3. The algorithm maintains a vector of probability estimates (p) for each data point which is initialized to 1/2 (line 2) and updated during each iteration (line 8). In each iteration, LOGITBOOST computes working responses (z) and weights (w) for each data points (lines 4,5).[2] The subroutine FITFUNCTION generates a weighted least squares regression function from the working response (z) and data points (x) by using the weights w (line 6). The final classifier (F) is an additive model of these real-valued regression functions. The final prediction is the sign of F.

Algorithm 3. LOGITBOOST(D_n, T)
Input: Training set of n samples (D_n)
  Number of boosting iterations (T)
Output: The final classifier (F)
Procedure:
1: $F(x) \leftarrow 0$
2: $p(x_i) = \frac{1}{2}$ for $i = 1, 2, \ldots, n$.
3: **for** $t \leftarrow 1$ to T **do**
4: $z_i \leftarrow \frac{y_i^* - p(x_i)}{p(x_i)(1-p(x_i))}$ for $i = 1, 2, \ldots, n$.
5: $w_i \leftarrow p(x_i)(1 - p(x_i))$ for $i = 1, 2, \ldots, n$.
6: $f_t \leftarrow$ FITFUNCTION(z, x, w)
7: $F(x_i) \leftarrow F(x_i) + \frac{1}{2} f_t(x_i)$ for $i = 1, 2, \ldots, n$.
8: $p(x_i) \leftarrow \frac{e^{F(x_i)}}{e^{F(x_i)} + e^{-F(x_i)}}$ for $i = 1, 2, \ldots, n$.
9: **end for**
10: return $F = \sum_{t=1}^{T} f_t$

4.1.1 Computational Complexity

The cost of finding the best regression function is $\Theta(dn)$ if the data samples are sorted based on each attribute. Hence, the computational complexity of LOGITBOOST is $\Theta(dn(T + \log n))$.

4.2 LOGITBOOST.PL

The proposed parallel LOGITBOOST.PL algorithm is described in Algorithm 4. It follows a similar strategy to the one described in Algorithm 2. LOGITBOOST.PL also distributes the data set among the workers where each worker independently induces its own boosting model. It should be noted that the LOGITBOOST does not assign any weights for the regression functions. The main distinction from the ADABOOST.PL is that the workers' functions are sorted with respect to their unweighted error rate as shown below:

$$\epsilon = \sum_{i=1}^{n} I\{sign(f(x_i)) \neq sign(y_i)\}. \tag{14}$$

This new reordered function lists are used to get the merged functions as before. The merged function averages the output of the participating functions:

2. $y^* = (y + 1)/2$, taking values 0, 1. Here, y is the class label which belongs to $\{-1, +1\}$.

$$f^{(t)}(x) = \frac{1}{M}\sum_{i=1}^{M} f^{i^*(t)}. \tag{15}$$

The final classifier is the addition of all the T merged functions.

Algorithm 4. LOGITBOOST.PL($D^1_{n^1}, \ldots, D^M_{n^M}$, T)
Input: The training sets of M workers ($D^1_{n^1}, \ldots, D^M_{n^M}$)
Number of boosting iterations (T)
Output: The final classifier (F)
Procedure:
1: **for** $p \leftarrow 1$ to M **do**
2: $H^p \leftarrow$ LOGITBOOST($D^p_{n^p}, T$)
3: $H^{p^*} \leftarrow$ the weak classifiers in H^p sorted w.r.t. their unweighted error rate.
4: **end for**
5: **for** $t \leftarrow 1$ to T **do**
6 $f^{(t)} \leftarrow$ MERGE($f^{1^*(t)}, \ldots, f^{M^*(t)}$)
7: **end for**
8: return $F = \sum_{t=1}^{T} f^{(t)}$

4.2.1 Computational Complexity

For fixed T, the computational cost for LOGITBOOST.PL is same as that of ADABOOST.PL which is $\Theta(\frac{dn}{M}\log\frac{n}{M} + \frac{Tdn}{M})$.

5 MapReduce Framework

MapReduce is a distributed programming paradigm for cloud computing environment introduced by Dean and Ghemawat [5]. The model is capable of processing large data sets in a parallel distributed manner across many nodes. The main goal is to simplify large-scale data processing by using inexpensive cluster computers and to make this easy for users while achieving both load balancing and fault tolerance.

MapReduce has two primary functions: the Map function and the $Reduce$ function. These functions are defined by the user to meet the specific requirements. The Map function, takes a key-value pair as input. The user specifies what to do with these key-value pairs and produces a set of intermediate output key-value pairs

$$Map(key_1, value_1) \rightarrow List(key_2, value_2).$$

User can also set the number of Map functions to be used in the cloud. Map tasks are processed in parallel by the nodes in the cluster without sharing data with any other nodes. After all the Map functions have completed their tasks, the outputs are transferred to $Reduce$ function(s). The $Reduce$ function accepts an intermediate key and a set of values for that key as its input. The $Reduce$ function is also user defined. User decides what to do with these key and values, and produces a (possibly) smaller set of values

$$Reduce(key_2, List(value_2)) \rightarrow List(value_2).$$

The original *MapReduce* software is a proprietary system of *Google*, and therefore, not available for public use. For our experiments, we considered two open source implementations of *MapReduce*: *Hadoop* [19] and *CGL-MapReduce* [20]. *Hadoop* is the most widely known *MapReduce* architecture. *Hadoop* stores the intermediate results of the computations in local disks and then informs the appropriate workers to retrieve (pull) them for further processing. This strategy introduces an additional step and a considerable communication overhead. *CGL-MapReduce* is another *MapReduce* implementation that utilizes *NaradaBrokering* [21], a streaming-based content dissemination network, for all the communications. This feature of *CGL-MapReduce* eliminates the overheads associated with communicating via a file system. Moreover, *Hadoop*s *MapReduce API* does not support configuring a Map task over multiple iterations and hence, in the case of iterative problems, the Map task needs to load the data again and again in each iteration. For these reasons, we have chosen *CGL-MapReduce* for our implementation and experiments.

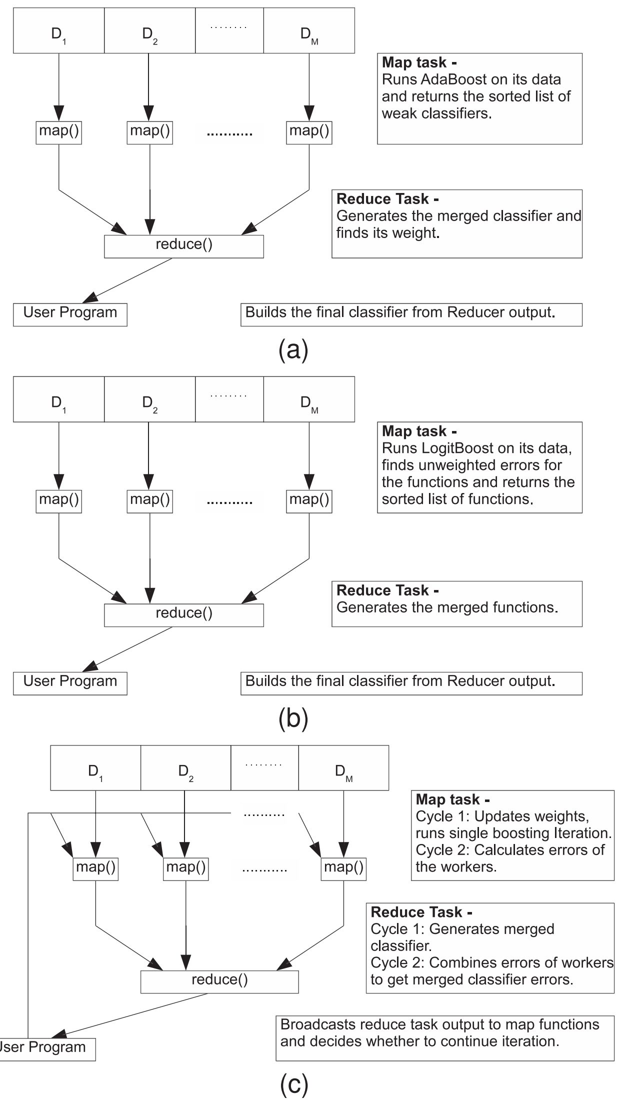


Fig. 1. MapReduce work flow for: (a) ADABOOST.PL, (b) LOGITBOOST.PL, and (c) MULTBOOST.

5.1 MapReduce-Based Implementation

Fig. 1 shows the work flows of ADABOOST.PL, LOGITBOOST.PL, and MULTBOOST in *MapReduce* framework. Each of the Map functions (Algorithms 5 and 7) represents a worker having access to only a subset of the data. For ADABOOST.PL (or LOGITBOOST.PL), each of the M Map functions runs respective ADABOOST (or LOGITBOOST) algorithm on their own subset of the data to induce the set of weak classifiers (or regression functions). LOGITBOOST.PL has an additional step of calculating the unweighted error

rates. Then, the base classifiers (or functions) are sorted. These weak classifiers (or functions) along with their weights are transmitted (not applicable for LOGITBOOST.PL) to the $Reduce$ function (Algorithms 6 and 8). After receiving the weak classifiers (or functions) from all the Map functions, the $Reduce$ function merges them at the same sorted level and averages (not required for LOGITBOOST.PL) the classifier weights to derive the weights of the merged classifiers. When all T (total number of boosting iterations) merged classifiers (or functions) are ready, they are sent to the user program and the final classifier is induced. Note that all the boosting iterations are executed in a single burst within the Map function. Hence, for ADABOOST.PL and LOGITBOOST.PL, we need only one cycle of *MapReduce* to complete the algorithms.

Algorithm 5. MAP::ADABOOST.PL(key_1, $value_1$)
Input: a $(key, value)$ pair where $value$ contains the Number of boosting iterations (T).
Output: a key and a $List(value)$ where $List$ contains the sorted T weak classifiers along with their weights.
Procedure:
1: run ADABOOST with T iterations on the Mapper's own data.
2: sort the T weak classifiers w.r.t their weights.
3: embed each weak classifier and the corresponding weight in the $List$.
4: return (key_2, $List(value_2)$)

Algorithm 6. REDUCE::ADABOOST.PL(key_2, $List(value_2)$)
Input: a key and a $List(value)$ where $List$ contains all the sorted weak classifiers of M Mappers.
Output: a $List(value)$ containing the final classifier.
Procedure:
1: **for** $t \leftarrow 1$ to T **do**
2: merge M weak classifiers each from the same sorted level of each Map output.
3: calculate the weight of this merged classifier by averaging the weights of the participating weak classifiers.
4: embed the merged classifier with the weight in the output $List$.
5: **end for**
6: return ($List(value_2)$)

Algorithm 7. MAP::LOGITBOOST.PL(key_1, $value_1$)
Input: a $(key, value)$ pair where $value$ contains the Number of boosting iterations (T).
Output: a key and a $List(value)$ where $List$ contains the sorted T regression functions
Procedure:
1: run LOGITBOOST with T iterations on the Mapper's own data.
2: calculate the unweighted error rates for each of the T regression functions.
3: sort the T regression functions classifiers w.r.t their unweighted error rates.
4: embed each regression function in the $List$.
5: return (key_2, $List(value_2)$)

During each iteration of MULTBOOST [6], each worker builds a weak classifier on its own data (see Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.208, for details). These weak classifiers are merged to a single classifier and then the workers measure the weighted errors of this merged classifier on their respective portions of the data. The errors are added to get the total error for the merged classifier and accordingly, the workers update the data points' weights. Then, the next iteration begins. Hence, in order to complete a single boosting iteration of MULTBOOST the *MapReduce* cycle needs to iterate two times. In the first cycle, the Map function receives the error rate of previously merged classifier (except for the first iteration) and updates the data point's weights. Then, it executes a single boosting iteration and generates the weak classifier. This weak classifier is output to the *Reducer*. The *Reducer* collects the weak classifiers from all the M workers, forms the merged classifier and sends it to the user program. Upon receiving the merged classifier, the user program initiates the second cycle by sending it to all of the Map functions. In this second cycle, each of the Map functions calculates the error rate of the received merged classifier on its own data and transmits it to the *Reducer*. After receiving from all the Map functions, the *Reducer* adds the M errors. This summation is the weighted error on the complete data set. It is passed to the user program and thus one MULTBOOST iteration completes. The user program keeps track of the iteration numbers completed and accordingly initiates the next iteration.

Algorithm 8. REDUCE::LOGITBOOST.PL(key_2, $List(value_2)$)
Input: a key and a $List(value)$ where $List$ contains all the sorted regression functions of M Mappers.
Output: a $List(value)$ containing the final classifier.
Procedure:
1: **for** $t \leftarrow 1$ to T **do**
2: merge M regression functions each from the same sorted level of each Map output.
3: embed the merged classifier in the output $List$.
4: **end for**
5: return ($List(value_2)$)

Note that, for ADABOOST.PL and LOGITBOOST.PL, the MapReduce framework does not need to be iterated. Thus, there are very few communications (which are often costly) between the framework components. This feature significantly contributes to the reduction of the overall execution time of the proposed algorithms.

5.2 Privacy-Preserving Aspect

For many real-world problems, it is vital to make the distributed computations secured. For the computation to be considered completely secure, the participants should learn nothing after the completion of the task, except for what can be inferred from their own input. For example, consider a scenario of making a decision if a client is qualified for receiving a loan where multiple institutions have the data about the client, but none of the institutions want to disclose the sensitive information to any other. A combined decision based on all the available data will be more knowledgable. In

such a scenario, the goal of distributed learning is to induce a classifier that performs better than the classifiers that the participants could learn using only their separate data sets, while, at the same time, disclosing as little information as possible about their data.

Verykios et al. [22] identified three different approaches for privacy-preserving data mining. Algorithms in the first approach perturb the values of selected attributes of individual records before communicating within the data sets (e.g., [23], [24]). The second approach randomize the data in a global manner by adding independent Gaussian or uniform noise to the attribute values (e.g., [25]). The third strategy [26] uses cryptographic protocols whenever sharing knowledge between the participants. Cryptographic protocols can be used in our algorithms in order to achieve robust privacy-preserving computation.

Our algorithms do not directly communicate the data sets, rather we distribute the learning procedure among the participants. The primary objective of our approach is to preserve the privacy of the participants' data while approximating the performance of the classifier as much as possible compared to the performance on the fully disclosed data available by combining all the participants' data sets.

From the *MapReduce* work flows of ADABOOST.PL and LOGITBOOST.PL, it is evident that the Map workers do not have to share their data or any knowledge derived from the data with each other. The Map workers never get any hint about the complete data set. Eventually, the $Reducer$ receives all the classifiers. Note that, we have only one $Reduce$ worker and the user program waits for the completion of the job performed by the $Reduce$ worker. Hence, we can accommodate the task of $Reducer$ within the user program and eliminate any risk of leaking knowledge to a worker. Thus, our algorithms have a great potential for being used in privacy-preserving applications. Adding cryptographic protocols will further protect it from any eavesdropping over the channel.

5.3 Communication Cost

For the communication cost analysis, let the cost of communications from the user program to Map functions, from Map functions to $Reduce$ function, and from $Reduce$ function to the user program be f, g, and h, respectively. Then, the communication cost for ADABOOST.PL and LOGITBOOST.PL will be $(f + g + h)$. MULTBOOST will take $2T(f + g + h)$ time where T is the number of iterations.

6 Experimental Results

In this section, we demonstrate the performance of our proposed algorithms in terms of various performance metrics such as classification accuracy, speedup, and scaleup. We compared our results with standard ADABOOST, LOGITBOOST, and MULTBOOST (in a parallel setting). All our experiments were performed on *Amazon EC2*[3] cloud computing environment. The computing nodes used were of type *m1.small* configured with 1.7 GHz 2006 *Intel Xeon* processor, 1.7 GB of memory, 160 GB storage, and 32 bit *Fedora Core 8*.

3. http://aws.amazon.com/ec2/.

TABLE 1
Data Sets Used in Our Experiments

Data sets	*No. of Instances (n)*	*No. of Attributes(d)*	*Size on Disk*
yeast	892	8	34 KB
wineRed	1599	12	84 KB
wineWhite	4898	12	263 KB
pendigits	7494	16	360 KB
spambase	4601	57	687 KB
musk	6598	167	4.2 MB
telescope	19020	11	1.4 MB
swsequence	3527	6349	169 MB
biogrid	4531	5367	47 MB
isolet	6238	617	24 MB
d1	200000	20	34.3 MB
d2	150000	20	25.7 MB
d3	100000	20	17.1 MB
d4	5000	2000	86 MB
d5	5000	1500	65 MB
d6	5000	1000	43 MB
alpha1	300000	700	1.8 GB
alpha2	400000	700	2.4 GB

6.1 The Data Sets

We selected a wide range of synthetic and real-world data sets with varying dimensions and sizes ranging from few Kilobytes to Gigabytes. Table 1 summarizes 10 publicly available [27] real-world data sets and eight synthetic data sets used in our experiments. The *spambase* data set classifies e-mails as spam or nonspam. The training set is a compilation of user experiences and spam reports about incoming mails. The *musk* data set contains a set of chemical properties about the training molecules and the task is to learn a model that predicts a new molecule to be either musks or nonmusks. The *telescope* data set contains scientific information collected by Cherenkov Gamma Telescope to distinguish the two classes: Gamma signals and hadron showers. The *swsequence* data [28] represents the homological function relations that exist between genes belonging to the same functional classes. The problem is to predict whether a gene belongs to a particular functional class (Class 1) or not. The *biogrid* [29] is a protein-protein interaction database that represents the presence or absence of interactions between proteins. The *pendigits* data set classifies handwritten digits collected through pressure sensitive writing pad. It was originally designed to be used for multiclass classification with a total of 10 classes (one for each digit from 0 to 9). Instead, we chose to transform it into a binary classification task by assigning the negative class to all even numbers and the positive class to the odd numbers. *Isolet* is a data set from speech recognition domain and the goal is to predict the letter name that was spoken. We also modified this 26 class problem into a binary classification problem by putting first 13 letters in one class and the rest in the other class. The biological data set, *yeast* classifies the cellular localization sites of Proteins. It is also a multiclass problem with a total of 10 classes. We retained samples only from the two most frequent classes (*CYT, NUC*). The *wineRed* and *wineWhite* data sets [30] model the wine quality based on some physicochemical tests and enumerates the quality score between 0 and 10. In this case, we assigned the negative class to all scores that are less than or equal to five and the positive class to the rest.

TABLE 2
Comparison of the 10-Fold Cross-Validation Error Rates (Standard Deviations) for the Standard ADABOOST, Best Local ADABOOST (LOCALADA), MULTBOOST, and ADABOOST.PL Algorithms Using 10 and 20 Workers

Data set	ADABOOST	10 workers LOCALADA	10 workers MULTBOOST	10 workers ADABOOST.PL	20 workers LOCALADA	20 workers MULTBOOST	20 workers ADABOOST.PL
yeast	0.3353 (0.0440)	0.3700 (0.0439)	0.3499 (0.0382)	0.3464 (0.0478)	0.3644 (0.0453)	0.3386 (0.0257)	0.3375 (0.0419)
wineRed	0.2514 (0.0221)	0.2777 (0.0318)	0.2602 (0.0374)	**0.2464 (0.0317)**	0.2927 (0.0435)	0.2739 (0.0468)	0.2564 (0.0297)
wineWhite	0.2317 (0.0212)	0.2558 (0.0145)	0.2542 (0.0140)	**0.2313 (0.0228)**	0.2695 (0.0236)	0.2517 (0.0117)	**0.2309 (0.0199)**
pendigits	0.0722 (0.0099)	0.0823 (0.0124)	0.0878 (0.0136)	**0.0699 (0.0081)**	0.0889 (0.0085)	0.1045 (0.0122)	**0.0642 (0.0102)**
spambase	0.0572 (0.0090)	0.0706 (0.0086)	0.0863 (0.0126)	0.0576 (0.0072)	0.0845 (0.0079)	0.0850 (0.0134)	0.0617 (0.0071)
musk	0.0515 (0.0067)	0.0720 (0.0090)	0.0727 (0.0121)	0.0565 (0.0094)	0.0829 (0.0110)	0.0788 (0.0103)	0.0612 (0.0111)
telescope	0.1551 (0.0063)	0.1623 (0.0059)	0.1694 (0.0119)	0.1599 (0.0086)	0.1683 (0.0066)	0.1685 (0.0100)	0.1595 (0.0132)
swsequence	0.3453 (0.0281)	0.3504 (0.0153)	0.3698 (0.0288)	**0.3334 (0.0145)**	0.3624 (0.0196)	0.3726 (0.0347)	**0.3368 (0.0174)**
biogrid	0.3136 (0.0050)	0.3475 (0.0258)	0.3625 (0.0356)	0.3180 (0.0134)	0.3740 (0.0465)	0.3746 (0.0421)	0.3291 (0.0511)
isolet	0.1577 (0.0146)	0.2139 (0.0160)	0.2050 (0.0169)	0.1601 (0.0162)	0.2502 (0.0181)	0.2358 (0.0177)	0.1661 (0.0150)
alpha1	0.0960 (0.0016)	0.1204 (0.0125)	0.1238 (0.0045)	0.0972 (0.0043)	0.1396 (0.0100)	0.1132 (0.0022)	0.1000 (0.0022)
alpha2	0.0974 (0.0027)	0.1296 (0.0010)	0.1207 (0.0014)	0.1049 (0.0014)	0.1402 (0.0108)	0.1135 (0.0017)	0.1020 (0.0024)

Similar to the real-world data sets, all our synthetic data sets are also binary in class. For the synthetic data sets *d1-d6*, we used the synthetic data generator *RDG1* available in WEKA [31] data mining tool. *RDG1* produces data randomly by a decision list consisting of a set of rules. If the existing decision list fails to classify the current instance, a new rule according to this current instance is generated and added to the decision list. The next two large data sets, *alpha1* and *alpha2*, were generated by the *QUEST* generator [32] using a perturbation factor of 0.05 and function 1 for class assignment.

6.2 Prediction Performance

Tables 2 and 3 report the 10-fold cross-validation error rates for ADABOOST.PL and LOGITBOOST.PL, respectively. For ADABOOST.PL, we compared its generalization capability with MULTBOOST, and the standard ADABOOST algorithms. In addition, to demonstrate the superior performance of ADABOOST.PL, we also compared it with that of the best individual local ADABOOST classifier trained on the data from the individual computing nodes (denoted by LOCALADA). MULTBOOST is a variation of ADABOOST and hence we did not compare LOGITBOOST.PL with MULTBOOST. In the literature, we did not find any parallelizable version of LOGITBOOST to compare against, and hence LOGITBOOST.PL is compared with standard LOGITBOOST, and the best local LOGITBOOST classifier (denoted by LOCALLOGIT).

The experiments for ADABOOST were performed using a single computing node. For ADABOOST.PL and MULTBOOST, the experiments were parallelly distributed on a cluster setup with 5, 10, 15, and 20 computing nodes. During each fold of computation, the training set is distributed equally among the working nodes (using stratification so that the ratio of the number of class samples remain the same across the workers) and the induced model is evaluated on the test set. The final result is the average of the error rates for all the 10 folds. For the LOCALADA classifier, each individual model is induced separately using the training data from each of the working node and the performance of each model on the test data is calculated and the best result is reported. For ADABOOST, the error rates are calculated in a similar manner except that, on a single node, there is no need for distributing the training set. For all the algorithms, the number of boosting iterations is set to 100. In the exact same setting, LOGITBOOST.PL is compared with standard LOGITBOOST and the LOCALLOGIT classifier.

From Table 2, we observe that ADABOOST.PL (with a single exception) always performs better than MULTBOOST and LOCALADA algorithms. Furthermore, in some cases (marked bold in ADABOOST.PL columns), our algorithm outperforms even the standard ADABOOST. In all other cases, our results are competitive to that of the standard ADABOOST. Similarly, the prediction accuracy results for LOGITBOOST.PL (Table 3) are also competitive to original LOGITBOOST (sometimes even better) and are consistently

TABLE 3
Comparison of the 10-Fold Cross-Validation Error Rates (Standard Deviations) for the Standard LOGITBOOST, Best Local LOGITBOOST (*LocalLogit*) and LOGITBOOST.PL Algorithms on 5, 10, 15, and 20 workers

Data set	LogitBoost	5 workers LocalLogit	5 workers LogitBoost.PL	10 workers LocalLogit	10 workers LogitBoost.PL	15 workers LocalLogit	15 workers LogitBoost.PL	20 workers LocalLogit	20 workers LogitBoost.PL
yeast	0.341 (0.050)	0.373 (0.038)	0.358 (0.056)	0.386 (0.048)	0.361 (0.044)	0.386 (0.027)	**0.341 (0.034)**	0.378 (0.062)	**0.340 (0.030)**
wineRed	0.238 (0.035)	0.275 (0.030)	0.247 (0.033)	0.278 (0.031)	0.255 (0.030)	0.300 (0.048)	0.241 (0.030)	0.298 (0.043)	0.258 (0.035)
wineWhite	0.234 (0.024)	0.247 (0.025)	**0.232 (0.023)**	0.267 (0.016)	**0.234 (0.021)**	0.266 (0.018)	**0.227 (0.017)**	0.274 (0.017)	**0.233 (0.017)**
pendigits	0.066 (0.011)	0.069 (0.009)	**0.059 (0.008)**	0.077 (0.009)	**0.060 (0.009)**	0.083 (0.006)	**0.063 (0.009)**	0.088 (0.008)	**0.065 (0.009)**
spambase	0.055 (0.008)	0.064 (0.006)	**0.054 (0.009)**	0.070 (0.010)	**0.054 (0.007)**	0.075 (0.013)	0.058 (0.009)	0.087 (0.012)	0.060 (0.008)
musk	0.031 (0.007)	0.047 (0.008)	0.038 (0.007)	0.056 (0.009)	0.033 (0.011)	0.070 (0.013)	0.034 (0.011)	0.081 (0.013)	0.035 (0.011)
telescope	0.149 (0.008)	0.152 (0.008)	**0.143 (0.008)**	0.158 (0.008)	**0.145 (0.008)**	0.162 (0.009)	**0.143 (0.008)**	0.167 (0.009)	**0.146 (0.008)**
swsequence	0.330 (0.022)	0.342 (0.013)	**0.324 (0.023)**	0.354 (0.021)	0.331 (0.022)	0.369 (0.025)	0.331 (0.026)	0.369 (0.042)	0.334 (0.022)
biogrid	0.308 (0.010)	0.333 (0.033)	0.312 (0.011)	0.335 (0.027)	0.320 (0.014)	0.338 (0.014)	0.327 (0.014)	0.362 (0.013)	0.325 (0.013)
isolet	0.135 (0.009)	0.177 (0.011)	**0.130 (0.011)**	0.194 (0.015)	0.139 (0.011)	0.213 (0.022)	0.147 (0.015)	0.229 (0.019)	0.161 (0.011)
alpha1	0.093 (0.004)	0.104 (0.013)	0.098 (0.010)	0.114 (0.011)	0.112 (0.010)	0.140 (0.007)	0.095 (0.005)	0.143 (0.022)	0.103 (0.009)
alpha2	0.094 (0.003)	0.120 (0.005)	0.101 (0.009)	0.121 (0.010)	0.099 (0.005)	0.127 (0.010)	0.096 (0.004)	0.129 (0.016)	0.096 (0.010)

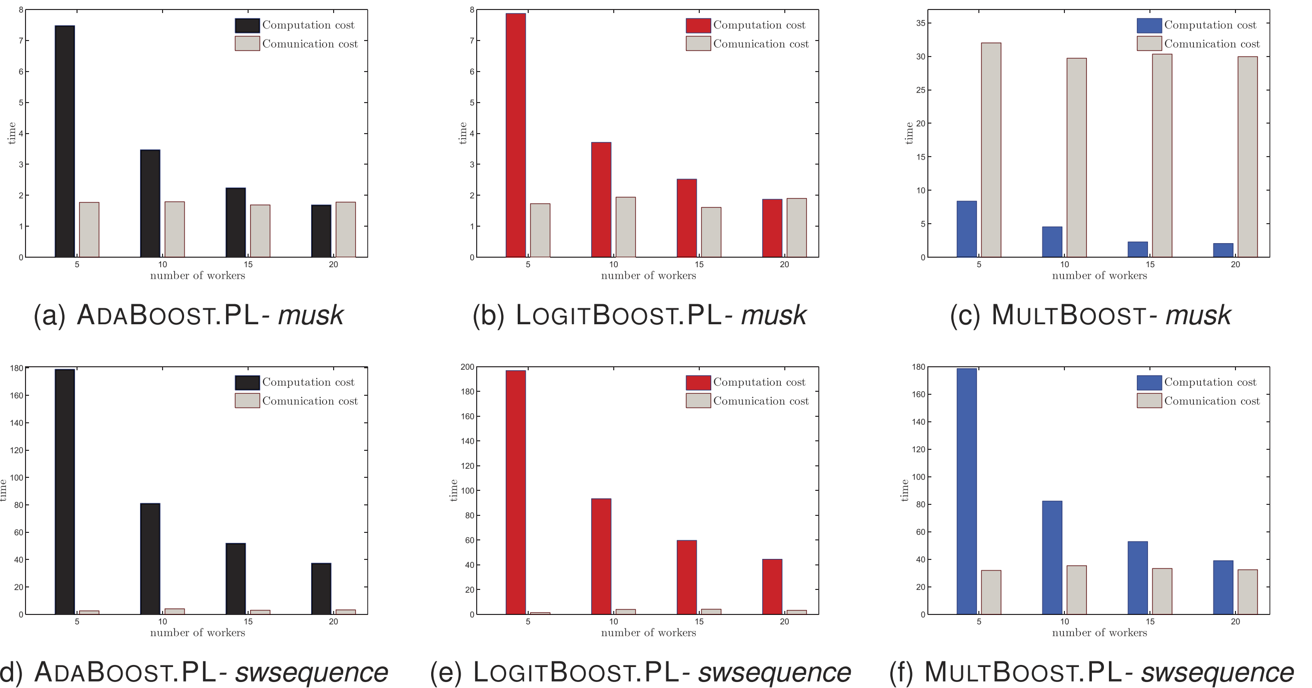


Fig. 2. The computational and communication costs of the algorithms for *musk* and *swsequence* data sets.

better than LOCALLOGIT classifier. Small compromise in the prediction performance is tolerable when the speedup in computation is significant which is really the case for our proposed algorithms (shown in the next section). Inherent parallelizable capability and insignificant difference in the prediction performance (of our algorithms compared to the respective baselines) suggests the efficacy of the proposed work in handling large-scale applications.

6.3 Results on Speedup

Speedup [33] is defined as the ratio of the execution time on a single processor to the execution time for an identical data set on p processors. In a distributed setup, we study the speedup behavior by taking the ratio of baseline (ADABOOST or LOGITBOOST) execution time (T_s) on a single worker to the execution time of the algorithms (T_p) on p workers for the same data set distributed equally. The $Speedup = T_s/T_p$. In our experiments, the values of p are 5, 10, 15, and 20. For our algorithms,

$$Speedup = \Theta\left(\frac{dn\log n + Tdn}{\frac{dn}{M}\log\frac{n}{M} + \frac{Tdn}{M}}\right) = \Theta\left(M\frac{\log n + T}{\log\frac{n}{M} + T}\right).$$

For the number of workers, $M > 1$, the inner fraction will be greater than 1. Hence, we can expect $speedup > M$ for our algorithms.

All the algorithms were run 10 times for each data set. We took the ratios of the mean execution times for calculating the speedup. The number of boosting iterations was set to 100. Fig. 3 shows the speedup gained by the algorithms on different data sets. From these plots, we observe that the larger the data set is, the better the speedups will be for both of our proposed algorithms. This is primarily due to the fact that the communication cost of the algorithms on smaller data sets tends to dominate the learning cost. For higher number of workers, the data size per workers decreases and so does the computation costs for the workers. This fact can be observed from Fig. 2. For the smaller data set *musk*, the communication costs are significantly larger compared to the computation cost, resulting in a diminishing effect on the speedup. But, for a large-scale *swsequence* data set, the computation cost is so dominant that the effect of communication cost on speedup is almost invisible. ADABOOST.PL invariably gains much better speedup than MULTBOOST for all the data sets.

6.4 Results on Scaleup

Scaleup [33] is defined as the ratio of the time taken on a single processor by the problem to the time taken on p processors when the problem size is scaled by p. For a fixed data set, speedup captures the decrease in runtime when we increase the number of available cores. Scaleup is designed to capture the scalability performance of the parallel algorithm to handle large data sets when more cores are available. We study scaleup behavior by keeping the problem size per processor fixed while increasing the number of available processors. For our experiments, we divided each data set into 20 equal splits. A single worker is given one data split and the execution time of baseline (ADABOOST or LOGITBOOST) for that worker is measured as T_s. Then, we distribute p data splits among p workers and the execution time of the parallel algorithm on p workers is measured as T_p. Finally, we calculate scaleup using this equation: $Scaleup = T_s/T_p$. In our experiments, the values of p are 5, 10, 15, and 20. The execution times were measured by averaging 10 individual runs. The number of boosting iterations for all the algorithms was 100.

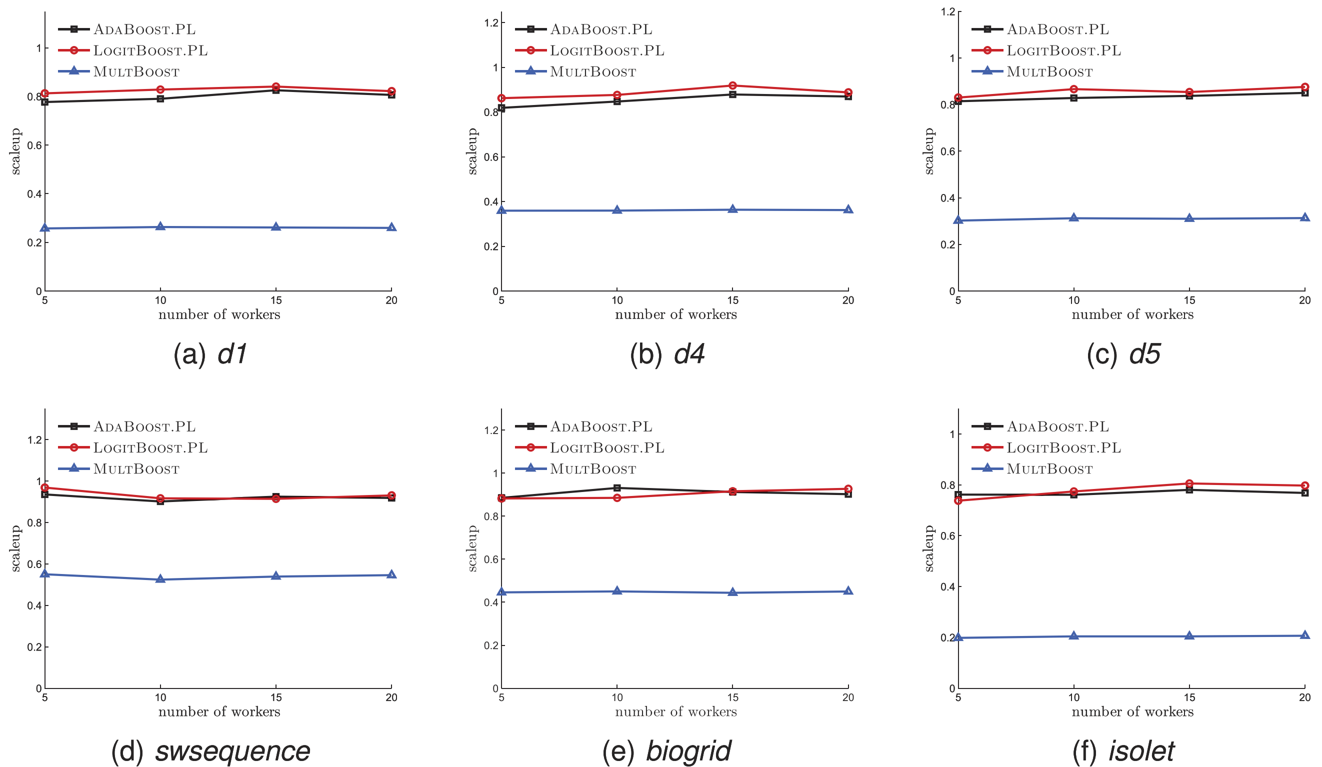
Fig. 4 shows the scaleup of the algorithms for three synthetic and three real-world data sets. Ideally, as we increase the problem size, we must be able to increase the number of workers in order to maintain the same runtime.

(a) *d1* (b) *d2* (c) *d3* (d) *d4*

(e) *d5* (f) *d6* (g) *swsequence* (h) *biogrid*

(i) *isolet* (j) *musk* (k) *telescope* (l) *spambase*

Fig. 3. The speedup comparisons for ADABOOST.PL, LOGITBOOST.PL, and MULTBOOST.



(a) *d1* (b) *d4* (c) *d5*

(d) *swsequence* (e) *biogrid* (f) *isolet*

Fig. 4. The scaleup comparisons for the ADABOOST.PL, LOGITBOOSTPL, and MULTBOOST.

The high and consistent scaleup values for ADABOOST.PL and LOGITBOOST.PL provide a strong evidence of their scalability. Regardless of the increase in the problem size, all that is needed is to increase the available resources and the algorithms will continue to effectively utilize all the workers. Nevertheless, the scaleup behavior of MULTBOOST is invariably lower compared to the proposed algorithms.

7 Conclusion and Future Work

We proposed two Parallel boosting algorithms that have good generalization performance. Due to the algorithms' parallel structure, the boosted models can be induced much faster and hence are much more scalable compared to the original sequential versions. We compared the performance of the proposed algorithms in a parallel distributed *MapReduce* framework. Our experimental results demonstrated that the prediction accuracy of the proposed algorithms is competitive to the original versions and is even better in some cases. We gain significant speedup while building accurate models in a parallel environment. The scaleup performance of our algorithms shows that they can efficiently utilize additional resources when the problem size is scaled up. In the future, we plan to explore other data partitioning strategies (beyond random stratification) that can improve the classification performance even further. We also plan to investigate the applicability of the recent work on multiresolution boosting models [34] to reduce the number of boosting iterations in order to improve the scalability of the proposed work.

Acknowledgments

This work was supported in part by the US National Science Foundation grant IIS -1242304. This work was performed while Mr. Palit completed his graduate studies at Wayne State University.

References

[1] Y. Freund and R.E. Schapire, "Experiments with a New Boosting Algorithm," *Proc. Int'l Conf. Machine Learning (ICML)*, pp. 148-156, 1996.
[2] L. Breiman, "Bagging Predictors," *Machine Learning*, vol. 24, no. 2, pp. 123-140, 1996.
[3] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5-32, 2001.
[4] D.W. Opitz and R. Maclin, "Popular Ensemble Methods: An Empirical Study," *J. Artificial Intelligence Research*, vol. 11, pp. 169-198, 1999.
[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, vol. 51, no. 1, pp. 107-113, 2008.
[6] S. Gambs, B. Kégl, and E. Aïmeur, "Privacy-Preserving Boosting," *Data Mining Knowledge Discovery*, vol. 14, no. 1, pp. 131-170, 2007.
[7] Y. Freund and R.E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *J. Computer and System Science*, vol. 55, no. 1, pp. 119-139, 1997.
[8] J. Friedman, T. Hastie, and R. Tibshirani, "Additive Logistic Regression: A Statistical View of Boosting," *The Annals of Statistics*, vol. 38, no. 2, pp. 337-407, 2000.
[9] J.K. Bradley and R.E. Schapire, "Filterboost: Regression and Classification on Large Datasets," *Proc. Advances in Neural Information and Processing Systems (NIPS)*, pp. 185-192, 2007.
[10] M. Collins, R.E. Schapire, and Y. Singer, "Logistic Regression, Adaboost and Bregman Distances," *Machine Learning*, vol. 48, nos. 1-3, pp. 253-285, 2002.
[11] G. Escudero, L. Màrquez, and G. Rigau, "Boosting Applied Toe Word Sense Disambiguation," *Proc. European Conf. Machine Learning (ECML)*, pp. 129-141, 2000.
[12] R. Busa-Fekete and B. Kégl, "Bandit-Aided Boosting," *Proc. Second NIPS Workshop Optimization for Machine Learning*, 2009.
[13] G. Wu, H. Li, X. Hu, Y. Bi, J. Zhang, and X. Wu, "Mrec4.5: C4.5 Ensemble Classification with Map-Reduce," *Proc. Fourth ChinaGrid Ann. Conf.*, pp. 249-255, 2009.
[14] B. Panda, J. Herbach, S. Basu, and R.J. Bayardo, "Planet: Massively Parallel Learning of Tree Ensembles with Mapreduce," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1426-1437, 2009.
[15] A. Lazarevic and Z. Obradovic, "Boosting Algorithms for Parallel and Distributed Learning," *Distributed and Parallel Databases*, vol. 11, no. 2, pp. 203-229, 2002.
[16] W. Fan, S.J. Stolfo, and J. Zhang, "The Application of Adaboost for Distributed, Scalable and On-Line Learning," *Proc. Fifth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, pp. 362-366, 1999.
[17] R. Caruana and A. Niculescu-Mizil, "Data Mining in Metric Space: An Empirical Analysis of Supervised Learning Performance Criteria," *Proc. 10th Int'l Conf. Knowledge Discovery and Data Mining (KDD '04)*, 2004.
[18] R.E. Schapire and Y. Singer, "Improved Boosting Algorithms Using Confidence-Rated Predictions," *Machine Learning*, vol. 37, no. 3, pp. 297-336, 1999.
[19] Apache, *Hadoop*, http://lucene.apache.org/hadoop/, 2006.
[20] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for Data Intensive Scientific Analyses," *Proc. IEEE Fourth Int'l Conf. eScience*, pp. 277-284, 2008.
[21] S. Pallickara and G. Fox, "Naradabrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids," *Proc. ACM/IFIP/USENIX Int'l Conf. Middleware (Middleware)*, pp. 41-61, 2003.
[22] V.S. Verykios, E. Bertino, I.N. Fovino, L.P. Provenza, Y. Saygin, and Y. Theodoridis, "State-of-the-Art in Privacy Preserving Data Mining," *SIGMOD Record*, vol. 33, pp. 50-57, 2004.
[23] V.S. Iyengar, "Transforming Data to Satisfy Privacy Constraints," *Proc. Eighth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, pp. 279-288, 2002.
[24] L. Sweeney, "Achieving k-Anonymity Privacy Protection Using Generalization and Suppression," *Int'l J. Uncertainty Fuzziness Knowledge-Based Systems*, vol. 10, pp. 571-588, 2002.
[25] A. Evfimievski, J. Gehrke, and R. Srikant, "Limiting Privacy Breaches in Privacy Preserving Data Mining," *Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems*, pp. 211-222, 2003.
[26] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and M.Y. Zhu, "Tools for Privacy Preserving Distributed Data Mining," *SIGKDD Explorations Newsletter*, vol. 4, pp. 28-34, 2002.
[27] A. Frank and A. Asuncion, "UCI Machine Learning Repository," http://archive.ics.uci.edu/ml, 2010.
[28] M.S. Waterman and T.F. Smith, "Identification of Common Molecular Subsequences," *J. Molecular Biology*, vol. 147, no. 1, pp. 195-197, 1981.
[29] C. Stark, B.J. Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz, and M. Tyers, "Biogrid: A General Repository for Interaction Datasets," *Nucleic Acids Research*, vol. 34, no. suppl. 1, pp. 535-539, 2006.
[30] P. Cortez, J. Teixeira, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Using Data Mining for Wine Quality Assessment," *Proc. 12th Int'l Conf. Discovery Science*, pp. 66-79, 2009.
[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten, "The Weka Data Mining Software: An Update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10-18, 2009.
[32] R. Agrawal, T. Imielinski, and A. Swami, "Database Mining: A Performance Perspective," *IEEE Trans. Knowledge and Data Eng.*, vol. 5, no. 6, pp. 914-925, Dec. 1993.
[33] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Addison-Wesley, 2003.
[34] C.K. Reddy and J.-H. Park, "Multi-Resolution Boosting for Classification and Regression Problems," *Knowledge and Information Systems*, vol. 29, pp. 435-456, 2011.

Indranil Palit received the bachelor's degree in computer science and engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh. He received the MS degree from Wayne State University. He is currently working toward the PhD degree from the Department of Computer Science and Engineering, University of Notre Dame.

Chandan K. Reddy (S'01-M'07) received the MS degree from Michigan State University and the PhD degree from Cornell University. He is currently an assistant professor in the Department of Computer Science, Wayne State University. His current research interests include data mining and machine learning with applications to social network analysis, biomedical informatics, and business intelligence. He published more than 40 peer-reviewed articles in leading conferences and journals. He was the recipient of the Best Application Paper Award in SIGKDD 2010. He is a member of the IEEE, ACM, and SIAM.

▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.