# Stochastic Gradient Boosted Distributed Decision Trees

Jerry Ye, Jyh-Herng Chow, Jiang Chen, Zhaohui Zheng
Yahoo! Labs
Sunnyvale, CA
{jerryye, jchow, jiangc, zhaohui}@yahoo-inc.com

## ABSTRACT

Stochastic Gradient Boosted Decision Trees (GBDT) is one of the most widely used learning algorithms in machine learning today. It is adaptable, easy to interpret, and produces highly accurate models. However, most implementations today are computationally expensive and require all training data to be in main memory. As training data becomes ever larger, there is motivation for us to parallelize the GBDT algorithm. Parallelizing decision tree training is intuitive and various approaches have been explored in existing literature. Stochastic boosting on the other hand is inherently a sequential process and have not been applied to distributed decision trees. In this work, we present two different distributed methods that generates exact stochastic GBDT models, the first is a MapReduce implementation and the second utilizes MPI on the Hadoop grid environment.

## Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Retrieval Models; I.2.6 [**Learning**]: Concept Learning

## General Terms

Algorithms

## 1. INTRODUCTION

Gradient tree boosting constructs an additive regression model, utilizing decision trees as the weak learner [5]. Although it is arguable for GBDT, decision trees in general have an advantage over other learners in that it is highly interpretable. GBDT is also highly adaptable and many different loss functions can be used during boosting. More recently, adaptations of GBDT utilizing pairwise and ranking specific loss functions have performed well at improving search relevance [2, 13]. In addition to its advantages in interpretability, GBDT is able to model feature interactions and inherently perform feature selection. Besides utilizing shallow decision trees, trees in stochastic GBDT are trained on a randomly selected subset of the training data and is less prone to over-fitting [6]. However, as we attempt to incorporate increasing numbers of features and instances in training data and because existing methods require all training data to be in physical memory, we focus our attention on distributing the GBDT algorithm.

In this paper, we present a scalable distributed stochastic GBDT algorithm that produces trees identical to those trained by the non-distributed algorithm. Our distributed approach is also generalizable to GBDT derivatives. We focus on stochastic boosting and adapting the boosting framework to distributed decision tree learning. In our work, we explore two different techniques at parallelizing stochastic GBDT on Hadoop[1]. Both methods rely on improving the training time of individual trees and not on parallelizing the actual boosting phase. Our initial attempt focused on an original MapReduce implementation and our second approach utilizes a novel way of launching MPI jobs on Hadoop.

## 2. RELATED WORK

Since decision trees were introduced by Qinlan [9], they have become a highly successful learning model and are used for both classification and regression. Friedman furthered the usage of decision trees in machine learning with the introduction of stochastic gradient boosted decision trees [6], using regression trees as weak learners.

Existing literature has explored parallelization of decision trees, but focused only on construction of single trees. In order to utilize larger training sets, parallelization of decision tree learning generally fell into one of two areas. The first focused on approximate methods to reduce training time [12]. The second approach partitions the training data and distributes the computation of the best split across multiple machines, aggregating splits to find a global best split [10]. However, none of the methods have focused on distributed learning of stochastic gradient boosted decision trees.

## 3. BACKGROUND

Stochastic GBDT is an additive regression model consisting of an ensemble of regression trees. Figure 1 shows a GBDT ensemble of binary decision trees with an arrow showing possible paths that a sample might traverse during testing. Each decision tree node is a split on a feature at a specific value, with a branch for each of the possible binary outcomes. Samples propagate through the tree depending on its specific feature value and the split points of the nodes that it visits. Each shaded terminal node returns a response for the tree and the sum of the responses from the trees is the score of the sample.

In our work, we utilize a Message Passing Interface (MPI) implementation from OpenMPI[2] as well as MapReduce. MPI has been in development for decades and works especially well for communication between machines. Recent work in grid computing such

[1]http://hadoop.apache.org
[2]http://www.openmpi.org

as the Apache Hadoop project have allowed for large scale deployments of cheap, interchangeable compute nodes [4]. Hadoop is an open source implementation of MapReduce [3] and it allows for streaming jobs where users can specify tasks for each compute node independent of MapReduce. In our work, we used MPI on Hadoop directly by writing a customized launcher.

The next section describes how we distributed the training process for stochastic GBDT.

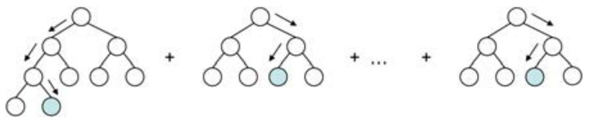

*Figure 1:* **A gradient boosted decision tree ensemble.**

## 4. METHOD

In this section, we describe how to parallelize the stochastic GBDT learning process. Chen et. al. [2] presented a good overview of regression trees and gradient boosting in Section 3 of TRADA. We describe various ways of partitioning the training data, our MapReduce implementation, and finally our MPI implementation on Hadoop. Before we proceed further, it is worth reiterating that we are only interested in learning exact models. The models are identical to those trained with the non-distributed version of the algorithm. This guided us as we made important design decisions. Alternatives to boosting and other tree ensemble methods such as random forests [11] were not attempted for this reason.

### 4.1 Distributed Training Data

In order to parallelize our decision tree training process, we must distribute the training data among machines. We are only interested in methods that would partition the data onto different machines rather than simply replicating it to reduce memory usage. There are already several existing approaches to distributed tree construction that aimed to improve scalability in terms of memory usage and improving training performance [8]. Caregea et. al. outlined algorithms for distributed tree training and presented different methods for partitioning training data, either horizontally or vertically [1]. In our work, we distributed our data using both vertically and horizontally partitioned methods.

### 4.2 MapReduce Implementation

Our initial implementation of distributed decision trees tried to frame the problem in the MapReduce paradigm and used a horizontal partitioning approach. Gehrke et. al. presented the concept of aggregating attribute, value, class label pairs and we utilized this in our MapReduce implementation [7]. Mappers would collect sufficient statistics [1] for tree construction, where each computes the candidate cutpoints by aggregating the unique attribute-value pairs. Algorithm 1 details the mapper and reducer code for finding candidate split points. During the map phase, the sufficient statistics consist of a key $(f, v)$, containing the feature $f$ and the feature value $v$, and the corresponding value $(r_i, w_i)$ consisting of the current residual and the weight of sample $i$. The reduce phase aggregates the residual and weight sums for each key. Given the output file, we then perform a single pass over the sorted cutpoints and the global best cut can be found.

This MapReduce method reduces the complexity of finding the optimal cutpoint for each feature from the dimension of the number of samples to the number of unique sample values. This method scales particularly well on datasets with categorical or Boolean features (e.g. click data). The entire process requires another map

**Algorithm 1** Aggregating candidate splits

```
$map(key, value)$:
F $\Leftarrow$ set of features
sample $\Leftarrow$ split(value,delim)
for $f$ in $F$ do
    key = (f, sample[f])
    value = (sample[residual], sample[weight])
    emit(key, value)
end for

$reduce(key, values)$:
residual_sum $\Leftarrow$ 0
weight_sum $\Leftarrow$ 0
for v in values do
    residual_sum $\Leftarrow$ residual_sum + v.residual
    weight_sum $\Leftarrow$ weight_sum + v.weight
end for
emit(key, (residual_sum,weight_sum))
```

**Algorithm 2** Partitioning a Node $n$

```
$map(key, value)$:
sample $\Leftarrow$ split(value,delim)
if sample[n.feature] $<$ n.splitpoint then
    residual = sample[residual]+ n.left_response
else
    residual = sample[residual]+ n.right_response
end if
emit(key, value)
```

task to partition the data for each node and a final one to apply the current ensemble after training an entire tree. Algorithm 2 shows pseudo code for updating the residuals for each sample. The partitioner writes samples out to different output files depending on which side of the split the sample ends up. The applier code is trivial to write in MapReduce and is omitted.

The MapReduce implementation is relatively straight forward to implement and requires few lines of code. However, because we essentially use HDFS for communication by writing out multiple files when splitting a node, we suffer from high system overhead. Hadoop is currently just not a good fit for this class of algorithms. Due to high communication overhead, we shift the focus of the remaining parts of this section to our MPI approach.

### 4.3 Learning a Distributed Regression Tree with MPI on Hadoop

Our second approach tries to optimize communication by using MPI on Hadoop streaming rather than MapReduce. For this implementation, we chose vertical partitioning since it minimizes the communication overhead of computing a tree node. For the rest of the paper, we will be working with vertically partitioned data, unless otherwise noted. Load balancing was performed to reduce time spent waiting for stragglers. For our implementation, we used our own MPI launcher for Hadoop.

#### *4.3.1 MPI on Hadoop*

In order to utilize existing Hadoop clusters, we modified OpenMPI to launch using Hadoop streaming. The main advantage of our approach is that we can use existing clusters without having to build out a dedicated MPI cluster. Technical challenges such as determining and communicating the master node, SSH-less job launching, and fault tolerance had to be solved in the process.

### 4.3.2 *Finding the Best Split for a Node*

The best split for a node is the split $c_{i,j}$ that maximizes $gain(c)$ across all unique cut points $j$ and feature $i \in F$, where $F$ is the set of all features.

For vertical partitioning, each machine works on a subset of the feature space $F_L$ and have only enough information to compute the best local split $S'_{i,j}$ for a unique cut point $j$ and feature $i \in F_L$.

$$S'_{i,j} = \text{argmax}_{i,j}\{gain(c_{i,j})\}$$

Each machine computes the best gain among its subset of features and sends this information to all of the other machines using an MPI broadcast. Each machine then determines the global best split $S^*_{i,j}$ by aggregating the local splits from each machine and determining which split $S'_{i,j}$ corresponds to the cut point $c'_{i,j}$ that maximizes gain.

$$S^*_{i,j} = \text{argmax}_{i,j}\{gain(c'_{i,j})\}$$

Every machine now knows the global best cut and waits for the machine with the split feature in memory to partition its samples and then transmit the updated post-split indices.

### 4.3.3 *Partitioning the Data*

During traditional decision tree construction, samples are partitioned into two subsets after a split point is learned. In distributed partitioning, only one machine has the feature in memory to partition the dataset. Therefore, only the machine with the best split can partition the data, updating other machines after it is finished.

In stochastic GBDT, we start off with a random subset of the training data for each tree. Since the targets for each sample in the next tree is the gradient of the loss function, we need the current score for every sample in the training set during boosting. This poses a unique problem in the distributed case where we have to apply the current ensemble on samples with features distributed on multiple machines. To remedy this, we modify our partition process to operate over all samples in the training data and maintain an incremental score index

$$\text{score}_m(x) = \text{score}_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$$

where the current score at tree $m$ depends on the score of the sample from training the previous $m-1$ trees, the indicator function, $I$ and the response $\gamma_{jm}$ of the corresponding region if $x$ is in region $R_{jm}$. Since the response is the residual mean of samples at the inner nodes of the tree, we can incrementally update the score index as we train new nodes. The additional overhead at each tree is inversely linear with the sampling rate.

### 4.3.4 *Finding the Best Node for Splitting*

We employ the same greedy tree growing procedure that splits the node with the highest gain among current leaf nodes. Although this does not guarantee the optimal tree, it is an efficient linear process and follows the method used in our non-distributed version.

Other tree growing methods can be implemented with additional parallelism, such as growing by level, but in the interest of attaining identical trees, we opted to use the greedy process.

## 4.4 Stochastic Gradient Boosting with Distributed Decision Trees

Gradient Boosted Decision Trees is an additive regression model consisting of numerous weak decision tree learners $h_i(x)$.

$$H_k(x) = \sum_{i=1}^{k} \gamma_i h_i(x)$$

where $\gamma_i$ denotes the learning rate. In Stochastic Gradient Boosting, each weak learner $h_i(x)$ is trained on a subsample of the training data.

Normally, during sequential training, the target $t_i$ of sample $x_i$ for the $k$-th tree is the gradient of the loss function $L(y_i, s_i)$, usually with parameters $y_i$ being the true label of the sample and the score $s_i = H_{k-1}(x_i)$. The loss function could be as generic as least squares or as specialized as GBRank [13].

A unique challenge occurs when training distributed decision trees where data is partitioned across multiple nodes. Since the loss function depends on the score of every sample in the training data for the ensemble thus far, we need to retrieve non-local feature values from other machines or keep track of the sample score throughout training. Our method implements the latter to minimize communication between machines. Therefore, during training, we modify our applier function to be

$$H_k(x) = s_k(x)$$

where $s_k(x)$ is the index described in Section 4.3.3, with $s_0(x) = 0$. Boosting then follows as shown below:

1. Randomly sample the training data with replacement to get subset $S_k$.
2. Set the target $r_i$ of examples in $S_k$ to be $r_i = L(y_i, s_k(x_i))$ where $y_i$ is the true label of the sample
3. Train the k-th tree $h_k(x)$ with the samples $(x_i, ri)$, where $x_i \in S_k$.

## 5. EXPERIMENTS

This section overviews the results of our experiments. Since our distributed GBDT construction process is designed to produce exactly the same results as the non-distributed version, we do not need to evaluate for differences in accuracy between learned models. We focus solely on evaluating performance and scalability of our systems. We first describe our dataset and experimental setup, then the results of our MapReduce implementation. Finally, we discuss the results of our implementation utilizing MPI on Hadoop.

## 5.1 Experimental Setup

Our experiments focused on evaluating the scalability of our systems when more machines are added to the process or when the feature or sample size is varied. All runs were made on a Hadoop cluster with only the number of machines allocated as needed.

The dataset $D$ used in our experiments consisted of 1.2 million (query, uri, grade) samples with 520 numeric features. To stress the system, we randomly generated, while respecting the existing feature distributions, additional datasets that were multiples of the dimension of $D$.

## 5.2 Experimental Results

### 5.2.1 *MapReduce Implementation*

Figure 2 is a log-log plot of training times for training a decision tree node. For each of the datasets, we trained a tree and averaged the training time for splitting a node, partitioning the samples, and updating the residuals for its children. As expected, the larger

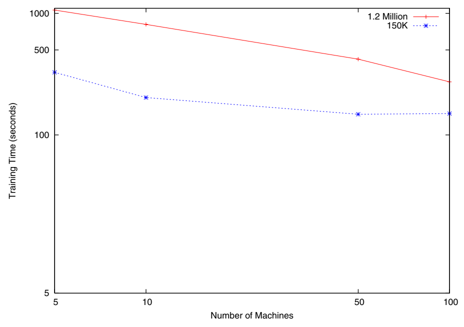

*Figure 2:* **Training time per node using the MapReduce implementation shown in log scale.**

dataset saw the largest improvement in training time as more mappers were added. Due to communication overhead, there were no further performance gains on the smaller dataset after 50 machines were used.

Communication costs were the main concern with this implementation. We discovered that this process was restricted by high cost of communication across HDFS (Hadoop currently does not have support for inter-node communication). Scheduling of multiple MapReduce jobs also proved to be a costly proposition. Given these factors, we were looking at minutes per node. This implementation was actually slower than the non-distributed version. From our results, we believe that MapReduce is simply not a good fit for communications heavy algorithms such as GBDT and highly iterative machine learning algorithms in general.

Although our MapReduce implementation scales well with even larger datasets, we were primarily interested in improving overall training time and changed our focus to using MPI on Hadoop. Comparatively, the MPI implementation trains an entire 20 terminal node tree in only 9 seconds with 20 machines on the 1.2M sample dataset while the MapReduce implementation took 273 seconds to train a single node with 100 machines.

### 5.2.2 *MPI Implementation.*

To evaluate the scalability of our MPI system, we used three different datasets of varying sizes consisting of 1.2 million samples, 500K samples, and 100K samples each. There were 520 features across all three datasets. For each dataset, we learned 10 trees with each tree trained until there were 20 leaf nodes. We repeated this experiment multiple times as we varied the number of machines during our distributed training process.

Figure 3 shows a log-log plot of the average training time as a function of the number of machines. The graph shows that the speedups depend on the size of the training data and that improvements start to taper off as more machines are added. In our dataset with 100K samples, the overhead from distributed tree training was too high for our implementation to be useful. For the 500K dataset, training time was reduced in half after using 2 machines but continued to improve sub-linearly until 5 machines were used and having no further improvements from additional machines.

The biggest advantage to using our implementation came with the larger 1.2 million sample dataset. We see that training time is halved by adding an additional machine and that improvements continue up to 20 machines with the final improvement from 70 seconds per tree to 9 seconds per tree. Figure 3 appears to indicate that we should be able to obtain even better scaling as our dataset grows larger.

We gained a lot in terms of performance using the MPI approach, but we lost some of the scalability afforded to us in the MapReduce implementation. Because we focused on minimizing inter-machine communication, we decided to go with a vertical partitioning of our dataset. Although most datasets have many features and will benefit from the scalability of this implementation, we do reach an upper limit when the size of one feature cannot fit in main memory on one machine.

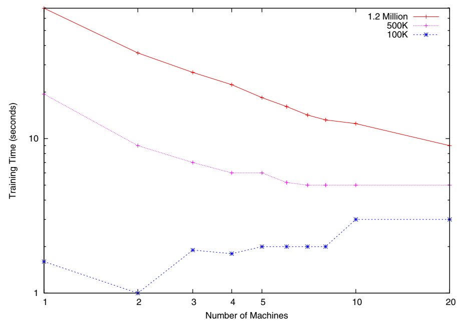

*Figure 3:* **Average training time per tree over 10 trees shown in log scale.**

## 6. CONCLUSIONS

We have shown two different methods of parallelizing stochastic gradient boosted decision trees. Our first implementation follows the MapReduce paradigm and ran on Hadoop. This approach required a very limited amount of code and scaled well. However, communication costs of reading from HDFS was too expensive for this method to be useful. In fact, it was slower than the sequential version. We believe that the main factor behind the results was that our communication intensive implementation is not well suited for the MapReduce paradigm. Our second method utilizes MPI and Hadoop streaming to run on the grid. This approach proved to be successful, obtained near ideal speedups, and scales well with large datasets.

## 7. REFERENCES

[1] CARAGEA, D., SILVESCU, A., AND HONAVAR, V. A framework for learning from distributed data using sufficient statistics and its application to learning decision trees. *International Journal of Hybrid Intelligent Systems 1*, 2 (2004).

[2] CHEN, K., LU, R., WONG, C. K., SUN, G., HECK, L., AND TSENG, B. L. Trada: tree based ranking function adaptation. In *CIKM* (2008), pp. 1143–1152.

[3] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[4] FOUNDATION, A. Apache hadoop project. *lucene.apache.org/hadoop*.

[5] FRIEDMAN, J. H. Greedy function approximation: A gradient boosting machine. *Annals of Statistics 29* (2001), 1189–1232.

[6] FRIEDMAN, J. H. Stochastic gradient boosting. *Comput. Stat. Data Anal. 38*, 4 (February 2002), 367–378.

[7] GEHRKE, J., RAMAKRISHNAN, R., AND GANTI, V. Rainforest - a framework for fast decision tree construction of large datasets. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA* (1998), A. Gupta, O. Shmueli, and J. Widom, Eds., Morgan Kaufmann, pp. 416–427.

[8] PROVOST, F., KOLLURI, V., AND FAYYAD, U. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery 3* (1999), 131–169.

[9] QUINLAN, J. R. Induction of decision trees. In *Machine Learning* (1986), pp. 81–106.

[10] SHAFER, J. C., AGRAWAL, R., AND 0002, M. M. Sprint: A scalable parallel classifier for data mining. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India* (1996), T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds., Morgan Kaufmann, pp. 544–555.

[11] STATISTICS, L. B., AND BREIMAN, L. Random forests. In *Machine Learning* (2001), pp. 5–32.

[12] SU, J., AND ZHANG, H. A fast decision tree learning algorithm. In *AAAI* (2006).

[13] ZHENG, Z., CHEN, K., SUN, G., AND ZHA, H. A regression framework for learning ranking functions using relative relevance judgments. *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval* (2007), 287–294.