XGBoost: Reliable Large-scale Tree Boosting System

Tianqi Chen and **Carlos Guestrin**
University of Washington
{tqchen, guestrin}@cs.washington.edu

Abstract

Tree boosting is an important type of machine learning algorithms that is widely used in practice. In this paper, we describe XGBoost, a reliable, distributed machine learning system to scale up tree boosting algorithms. The system is optimized for fast parallel tree construction, and designed to be fault tolerant under the distributed setting. XGBoost can handle tens of millions of samples on a single node, and scales beyond billions of samples with distributed computing.

1 Introduction

Machine learning and data mining have become a part of our daily lives. Our emails are protected by smart spam classifiers, which learn from massive amounts of spam data and user feedback. As we shop online, a recommender system helps us find products that match our taste by learning from our shopping history. Besides improving personal life, machine learning also plays a key role in helping companies make smart decisions and generate revenue: advertising systems match the right ads with the right users at the right time. Demand forecasting systems predict product demand in advance, allowing sellers to be prepared. Fraud detection systems protect banks from malicious attackers. There are two important factors behind the success of these applications — effective machine learning models that can capture the complex dependence between variables and scalable learning systems that effectively learn the model of interest with large amount of collected data.

Tree boosting [6] is one of the most important and widely used machine learning models. Variants of the model have been applied to problems such as classification [5, 8] and ranking [4]. These types of models are used by many winning solutions for machine learning challenges. They are also deployed into real world production systems, such as online advertising [7]. Despite its great success, the existing public practice of tree boosting algorithms are still limited to million scale datasets. While there is some discussion on how to parallelize this type of algorithm [9, 3, 11], there is little discussion about optimizing system and algorithm jointly in order to build a reliable tree boosting system that handles billion scale problems. In this paper, we introduce XGBoost, a novel machine learning system that reliably scales tree boosting algorithms to billions of samples with fault tolerance guarantees.

The major contribution of this paper is as follows: 1) We introduce a novel block based data layout to speedup tree construction in both in-memory and external memory setting; 2) We propose a new distributed approximation algorithm for tree searching; 3) We build a general fault-tolerant Allreduce protocol for reliable distributed communication. The system can handle tens of millions of data samples on a single machine, and reliably scales beyond billions of samples in distributed setting. We released the system as a machine learning open source software. It has been widely adopted and become one of the default machine learning packages that data scientists and industry practitioners use in daily practice.

Tree Boosting in a Nutshell We first briefly review the learning objective in tree boosting. For a given data set with n examples and m features $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ ($|\mathcal{D}| = n, \mathbf{x}_i \in \mathbb{R}^m$), a tree ensemble

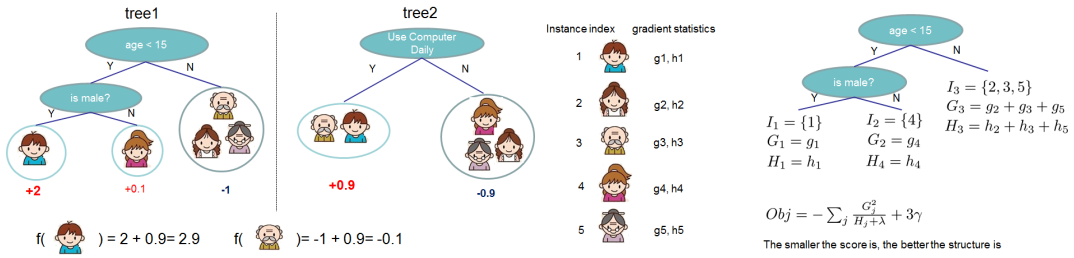


Figure 1: *Left:* The tree ensemble model. *Right:* The structure score calculation of a single tree.

model (shown in Fig. 1 left) uses K additive functions to predict the output.

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F},$$

where $\mathcal{F} = \{f(x) = w_{q(x)}\}(q : \mathbb{R}^m \to T, w \in \mathbb{R}^m)$ is the space of regression trees. Here q represents the structure of each tree that maps an example to the leaf index and w is the weight vector of each leaf. For a given example, we will use the decision rules in the trees (given by q) to classify them into the leaves and calculate the final prediction by summing up the scores in the corresponding leaves (given by w). To learn a tree ensemble, we optimize the following *regularized* objective

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k), \quad \Omega(f) = \gamma T + \lambda \|w\|^2.$$

Here l is a differentiable convex loss function that measures the quality of prediction $\hat{y}_i$ on training data, the second term Ω measures the complexity of the model to avoid over-fitting. The model is trained in an additive matter. At each iteration t, we add a new tree to optimize the objective. We can derive a score to measure the quality of a given tree structure q (the derivation is omitted due to space constraints).

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \tag{1}$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ are the gradient and second order gradient statistics. This score is like the impurity score for evaluating decision trees, except that it is derived for wider range of objective functions. Fig. 1 (right) shows how the score can be calculated: we only need to calculate the sum statistics of examples in each leaf node, then apply Eq (1) to get the quality of the tree. A greedy algorithm will iteratively search over the possible split candidates and find the best split to add to the tree until a maximum depth is reached.

2 Parallel Tree Learning Algorithm

The key question in tree learning algorithms is how to find the best split, as indicated by Eq (1). In order to do so, we need to enumerate over all possible splits over all the features. Algorithm 1 shows how we can do such split search on a single machine. The idea is to visit the data in sorted order on the feature type of interest, and accumulate the gradient histogram to calculate the structure score in Eq (1). In this section, we will discuss several improvements we have made to Algorithm 1.

Input Data Layout The most time consuming part of the tree learning algorithm is getting the data in sorted order. This makes the time complexity of learning each tree $O(n \log n)$. In order to reduce the cost of sorting, we propose to restructure the data into an in-memory unit which we called *block*. The data in each block is stored in a Compressed Column Storage (CSC) format, with each column sorted by the feature value. Fig 2 left shows how we can transform a dataset into the block-based format. This input data layout only needs to be computed once before training, and can be reused in later iterations. We can store the entire dataset into a single block, and run the split search algorithm by linearly scanning over the pre-sorted entries. This reduces the time complexity of the tree construction to $O(n)$. As we will show in the next section, the proposed layout will also help the algorithm for multiple blocks.

Algorithm 1: Parallel Tree Split Finding Algorithm on Single Machine

Input: I, instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m *in parallel* **do**
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for j *in sorted(I_k, ascent order by* $\mathbf{x}_{jk}$*)* **do**
    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
    $gain \leftarrow \max(gain, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split and default direction with max gain

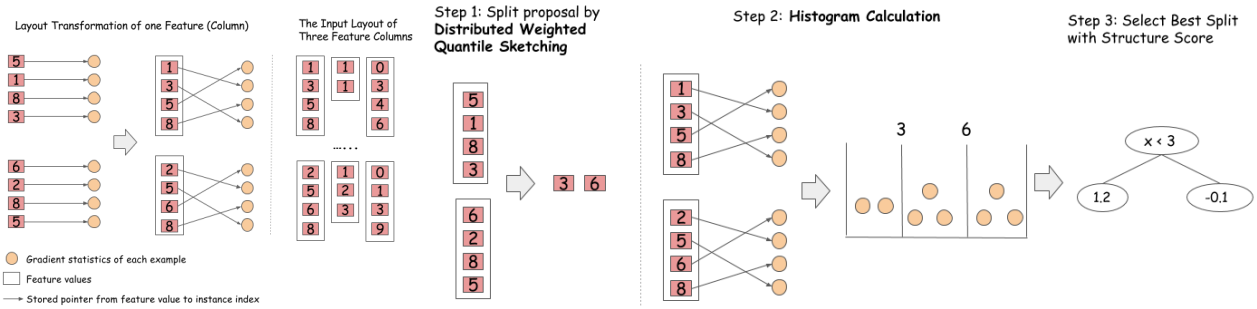


Figure 2: *Left:* The Input Data Layout. *Right:* Split Finding with Multiple Blocks.

Approximation Algorithm with Multiple Blocks While the single block layout works well when the data fits in the memory of a single machine, it cannot be directly applied to the distributed setting or external memory setting (where the data do not fit into the memory). For these general cases, we can store the data into several blocks. In this scenario, it is no longer possible to use Alg. 1 directly, because we cannot enumerate the entire dataset in sorted way. We will instead resort to an approximation algorithm. The algorithm starts by proposing potential good split points and then accumulating histograms to only evaluate the solution on these proposed splits. With a small number of proposal splits, the histogram accumulation can be done efficiently in the setting of multiple blocks. However, it is very important to find good proposals. We use an approximate quantile finding algorithm to propose the percentiles of the feature distribution on the branch to be splitted. The procedure is shown on Fig. 2 right. Importantly, the new input layout also enables *linear time* construction of both the quantile sketch and histogram. The new layout also enables easy parallelization of histogram estimation over features without introducing race conditions.

Cache-aware Prefetch Both Alg 1 and the proposed improved algorithm involve an indirect fetch of gradient statistics. This is a non-continuous memory access. A naive implementation of histogram accumulation will introduce read/write dependency between accumulation and non-continuous memory fetch operation. This can slow down the computation due to the cache misses. To alleviate the problem, we allocate an internal buffer in each thread, fetch the gradient statistics into it, and then perform accumulation in a mini-batch manner. This pre-fetching helps us to reduce the runtime overhead when the input data blocks are large.

Distributed and Out of Core Computing When there are multiple machines, we can distribute the data by rows, and generate local data blocks in each machine. The quantile estimation and histogram calculation are synchronized using an Allreduce style protocol, which we will elaborate in the next section. Notably, the same algorithm can be used in out-of-core setting, where only part of data is loaded into memory. In the out of core setting, we store the data blocks on disk, and use a pre-fetch thread to load the data block iteratively. This enables learning from datasets that are larger than a single machine's memory. The out-of-core computation can also benefit the distributed learning by allowing the algorithm to run with fewer memory resources.

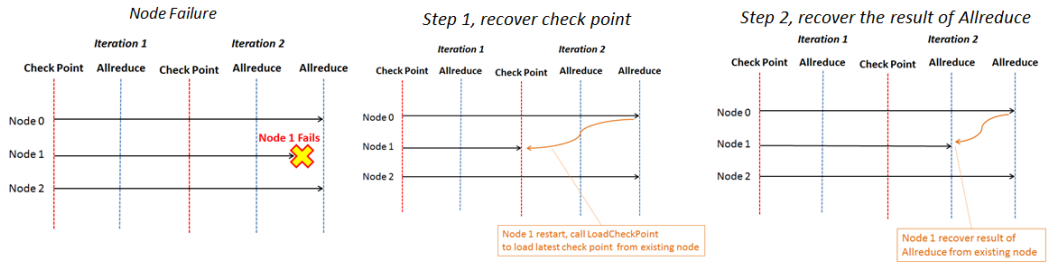


Figure 3: Allreduce Fault Recovery Protocol

3 Fault Tolerant Allreduce Protocol

In order to implement the distributed algorithm described in last section, we will need to have a communication protocol to do distributed quantile estimation and histogram aggregation. We choose to use Allreduce as the synchronization protocol. Allreduce has been successfully applied to other machine learning algorithms such as linear models [1]. The major advantage of using Allreduce is that the transition from single machine implementation to the distributed case is straightforward. Allreduce also preserves the program state between reductions naturally inside program itself. This is a valuable property for complicated machine learning algorithms, since every iteration of a machine learning algorithm can involve several rounds of reductions and the program state in the intermediate stage are hard to capture by an explicit checkpoint. However, most existing implementations of Allreduce are not fault tolerant between iterations, making it vulnerable to machine failures. The existing protocols, such as the one proposed in Agarwal et.al [1] is only fault-tolerant during startup stage of the program. We proposed a fault tolerant Allreduce protocol to address this problem.

Recovery Protocol One of the key properties of Allreduce is that all the callers get the same result after the call. Since most machine learning algorithm only need to use Allreduce and broadcast. We can make use of this property to construct a recovery protocol. Specifically, the result of Allreduce is recorded in some of the nodes after the call complete. When a node fails and restarts, the computed result can be forwarded to the restarted node, instead of doing the computation all over again. Fig. 3 gives an example of the protocol. When an error happens during the Allreduce stage, the alive nodes will reset their state to the state right before Allreduce, and continue the recovery step.

In order to implement the protocol, we need a consensus protocol to make all the nodes agree on which step needs to be recovered. Interestingly, this consensus protocol can be implemented using (a non-fault tolerant) Allreduce as well, by using Allreduce to find the minimum of step counters proposed by each node propose. The data is then transferred to the recovered node by a message passing algorithm. The recovery protocol is built on top of these two low level primitives that do not need to be fault tolerant[1]. This design also allows us to potentially swap the low level primitives to other native implementations in the runtime platform.

Virtual Checkpoint Another property of the tree boosting algorithm is that each node holds a replica of the model. The failed node can directly "load" the checkpoint from another node. This checkpoint is only required to be kept in memory before next iteration starts. This allows the training process to quickly restart from the most recent iteration. The recovery protocol can also safely discard all the recorded results of Allreduce before the checkpoint, making the final system memory efficient.

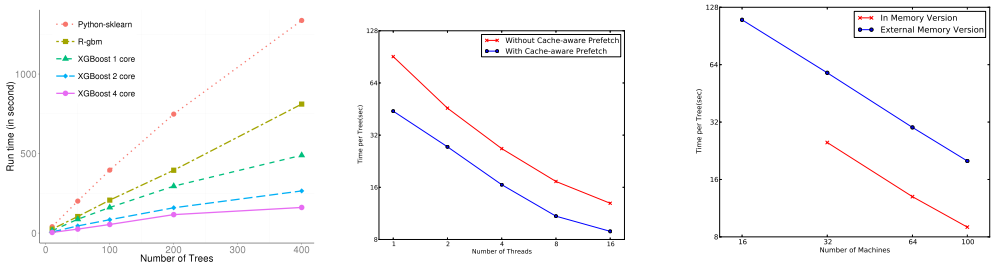
Rabit We implemented this protocol as a library named rabit [2](reliable Allreduce and broadcast interface). It can work on various platforms including MPI, SGE and YARN. XGBoost's communication layer is based on top of rabit, which enables it to handle machine failures gracefully and being able to work on the mentioned platforms.

4 Results and Discussions

Preliminary Experiment Results We benchmark the system against two major tree boosting packages on a single machine on the higgs boson challenge dataset. The result is shown in Fig. 4(a).

[1]We still need the low level primitives to be able to detect connection failures and notify the caller about the event, but do not require it to perform recovery.

[2]https://github.com/dmlc/rabit



(a) Speed Comparison to Other Packages (b) Impact of Prefetch on Allstate Claim Dataset (c) Speed of Distributed XGBoost on Billion Ads Dataset

Figure 4: Performance of XGBoost

As we can see even with a single core, our system runs twice as fast as R's gbm and four times as fast as scikit-learn. These improvements are bought by the optimizations mentioned in Sec. 2. We also conduct an experiment specifically to evaluate the gain of cache aware prefetch, on Allstate claim prediction dataset, which contains 10 million insurance claim records. The results are shown in Fig. 4(b). We can find that we can get around two times speedup by using cache aware prefetch on this setting. Finally we run an experiment on four billion criteo [3] Ad click through data set. The experiment is configured with c3.8xlarge EC2 instances, each with 32 cores and 64G of RAM. We place two workers on each of the machine, with each worker using 16 cores. The results are shown in Fig. 4(c). We can find that the distributed XGBoost can easily handle billion scale dataset, and gives near linear speedup with more machines.

Adoption and Industry Practice We have released our system as open source software on github [4]. Since its initial release, it has been widely adopted by data scientists and machine learning practitioners. XGBoost has been used as a major system in winner solutions of more than ten machine learning challenges in the past year, most of which are highly competitive. These include highly impactful ones in the field, such as KDDCup [2]. It has also been widely adopted by industry users, including Google, Alibaba and Tencent, and various startup companies. According to our contacts, xgboost can scale with hundreds of workers (with each worker utilizing multiple processors) smoothly and solve machine learning problems involve Terabytes of real world data.

Portability The system described in this paper can run without the support of any existing distributed frameworks, it is also designed to *work with and be embedded into existing distributed platforms*. Most existing distributed systems for data processing such as Spark [12] and Reef [10] take a bottom up approach, to provide common programming abstractions for building data processing and machine learning algorithms. Specific implementations are usually required to implement new learning algorithms on each of these platforms. XGBoost takes a top down approach, by building a scalable tree boosting system on top of a few primitives for which the implementation can be easily replaced. This allows the system to be ported to any platform that implements these primitives. The integration can also happen at different level, either on resource allocation, or the communication layer. So far, we have ported the system to common platforms such as Hadoop YARN, MPI and SGE with no change to the source code. We believe that it can also be easily ported to other platforms that support the minimum primitives required by XGBoost.

Acknowledgments

We would like to thank Marco Tulio Ribeiro and Sameer Singh for their valuable feedback. We would also like to thank Tong He, Bing Xu, Michael Benesty, Yuan Tang and other contributors of the opensource xgboost package. This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

[3] http://labs.criteo.com/downloads/download-terabyte-click-logs/

[4] https://github.com/dmlc/xgboost

References

[1] Alekh Agarwal, Oliveier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 15:1111–1133, 2014.

[2] Ron Bekkerman. The present and the future of the kdd cup competition: an outsiders perspective.

[3] Ron Bekkerman, Mikhail Bilenko, and John Langford. Scaling up machine learning: Parallel and distributed approaches. In *Proceedings of the 17th ACM SIGKDD International Conference Tutorials*, KDD '11 Tutorials.

[4] C. Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11:23–581, 2010.

[5] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *The annals of statistics*, 28(2):337–407, 2000.

[6] J.H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

[7] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD'14, 2014.

[8] P. Li. Robust Logitboost and adaptive base class (ABC) Logitboost. In *Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI'10)*, pages 302–311, 2010.

[9] S. Tyree, K.Q. Weinberger, K. Agrawal, and J. Paykin. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th international conference on World wide web*, pages 387–396. ACM, 2011.

[10] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, et al. Reef: Retainable evaluator execution framework. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1343–1355. ACM, 2015.

[11] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09.

[12] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, 2010.