XGBoost: A Scalable Tree Boosting System

Tianqi Chen, Carlos Guestrin

Kalmutskiy Kirill

November 2020

Ensemble methods

Ensemble methods aim at improving predictability in models by combining several models to make one very reliable model.

The most popular ensemble methods:

- Bagging: sampling technique where samples are derived from the whole population (set) using the replacement procedure
- Stacking: technique for averaging predictions of different models
- Boosting: technique that learns from previous predictor mistakes to make better predictions in the future

Why XGBoost?

- Highly scalable end-to-end tree boosting system
- Effective regularization, which allows achieving high metrics
- Important algorithmic optimizations
- Novel sparsity-aware algorithm for parallel tree learning
- Effective cache-aware block structure for out-of-core tree learning

Tree Boosting

For a given data set with n examples and m features $\mathcal{D} = \{(_i, y_i)\}$ $(|\mathcal{D}| = n, _i \in \mathbb{R}^m, y_i \in \mathbb{R})$, a tree ensemble model uses K additive functions to predict the output.

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

where $\mathcal{F} = \{f(x) = w_{q(x)}\}(q : \mathbb{R}^m \to T, w \in \mathbb{R}^T)$ is the space of regression trees.

Regularized Learning Objective

To learn the set of functions used in the model, it is necessary to to minimize the following *regularized* objective.

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

Here l is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and the target y_i.
The second term Ω penalizes the complexity of the model (i.e., the regression tree functions).

Gradient Tree Boosting

The tree ensemble model includes functions as parameters and cannot be optimized using traditional optimization methods in Euclidean space.

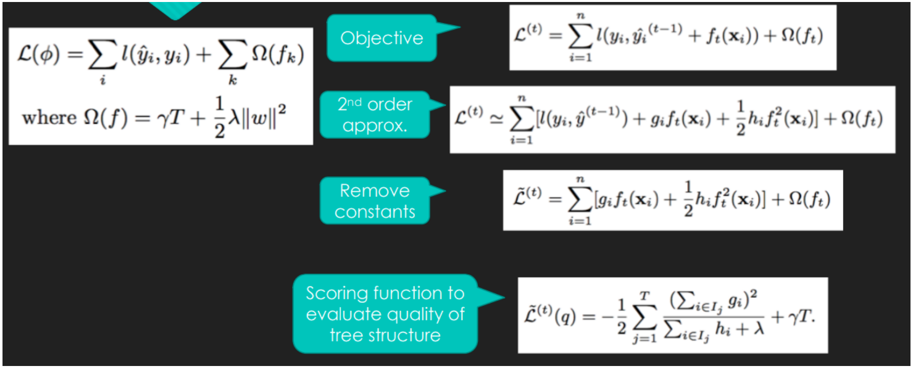
Let $\hat{y}_i^{(t)}$ be the prediction of the i-th instance at the t-th iteration, we will need to add f_t to minimize the following objective.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i{}^{(t-1)} + f_t(_i)) + \Omega(f_t)$$

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(_i) + \frac{1}{2} h_i f_t^2(_i)] + \Omega(f_t)$$

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 first and second order gradient statistics on the loss function.

Gradient Tree Boosting



$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

Objective

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

2nd order approx.

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

Remove constants

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

Scoring function to evaluate quality of tree structure

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

Example

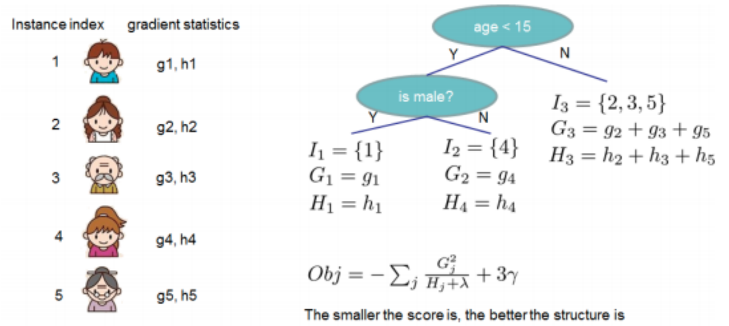


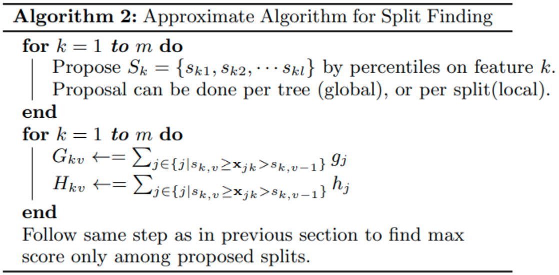
Figure 2: Structure Score Calculation. We only need to sum up the gradient and second order gradient statistics on each leaf, then apply the scoring formula to get the quality score.

Shrinkage and column subsampling

XGBoost uses the following techniques to prevent overfitting

- Shrinkage
  - Scales newly added weights by a factor ν
  - Reduces influence of each individual tree
  - Leaves space for future trees to improve model
  - Similar to learning rate in stochastic optimization
  - Can greatly affect the complexity of the model: high values lead to fewer trees and vice versa
- Column subsampling
  - Subsample features (columns)
  - Idea taken from the Random Forest algorithm
  - Prevents overfitting more effectively than object subsampling (rows)

Split-finding algorithms



Algorithm 2: Approximate Algorithm for Split Finding

```
for $k = 1$ to $m$ do
    Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature $k$.
    Proposal can be done per tree (global), or per split(local).
end
for $k = 1$ to $m$ do
    $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$
    $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$
end
Follow same step as in previous section to find max
score only among proposed splits.
```

- Exact
  - Computationally demanding
  - Enumerate all possible splits for continuous features
- Approximate
  - Algorithm proposes candidate splits according to percentiles of feature distributions
  - Maps continuous features to buckets split by candidate points
  - Aggregates statistics and finds best solution among proposals

Comparison of split-finding

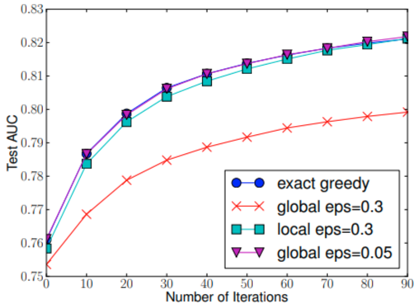


Figure 3: Comparison of test AUC convergence on Higgs 10M dataset. The eps parameter corresponds to the accuracy of the approximate sketch. This roughly translates to 1 / eps buckets in the proposal. We find that local proposals require fewer buckets, because it refine split candidates.

Sparsity-aware split finding

In many real-world problems, it is quite common for the input x to be sparse. There are multiple possible causes for sparsity:

- Presence of missing values in the data
- Frequent zero entries in the statistics
- Feature engineering such as one-hot encoding

How can this be dealt with? Define a “default” direction: when a value is missing in the sparse matrix x, the instance is classified into the default direction

Sparsity-aware split finding

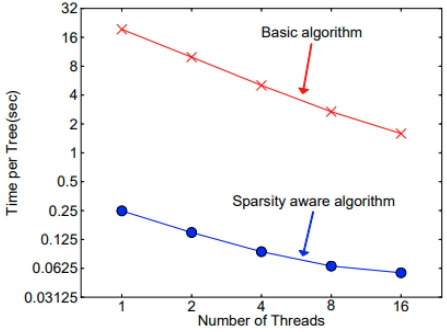


Figure 5: Impact of the sparsity aware algorithm on Allstate-10K. The dataset is sparse mainly due to one-hot encoding. The sparsity aware algorithm is more than 50 times faster than the naive version that does not take sparsity into consideration.

Column Block for Parallel Learning

The most time consuming part of tree learning is to get the data into sorted order. The block structure can help us with this. Data in each block is stored in the CSC format, with each column sorted by the corresponding feature value

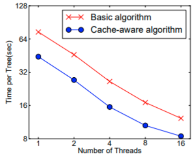
- Data is stored on multiple blocks, and these blocks are stored on disk
- Independent threads pre-fetch specific blocks into memory to prevent cache misses
- Block Compression
  - Each column is compressed before being written to disk, and decompressed on-the-fly when read from disk into a prefetched buffer
- Block Sharding
  - Data is split across multiple disks / clusters
  - Pre-fetcher is assigned to each disk to read data into memory

Cache-aware access

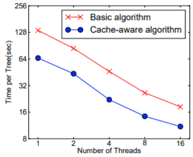
While the proposed block structure helps optimize the computation complexity of split finding, the new algorithm requires indirect fetches of gradient statistics by row index, since these values are accessed in order of feature. This is a non-continuous memory access.

- Exact Greedy Algorithm
  - Allocate an internal buffer in each thread
  - Fetch gradient statistics into this buffer
  - Perform accumulation in mini-batch
  - As a result, reduce runtime overhead when number of rows is large
- Approximate Algorithms
  - Solve this problem by choosing a correct block size
  - Small block size results in small workloads for each thread
  - Large block size results in cache misses as gradient statistics do not fit in cache

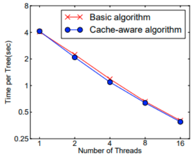
Cache-aware access



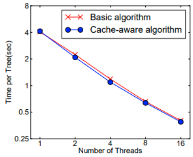
(a) Allstate 10M



(b) Higgs 10M

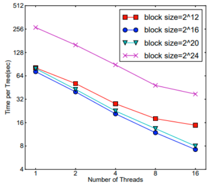


(c) Allstate 1M



(d) Higgs 1M

Figure 7: Impact of cache-aware prefetching in exact greedy algorithm. We find that the cache-miss effect impacts the performance on the large datasets (10 million instances). Using cache aware prefetching improves the performance by factor of two when the dataset is large.

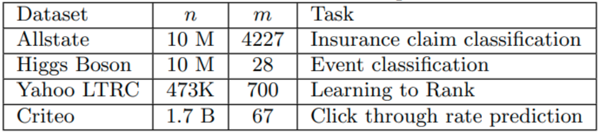


(b) Higgs 10M

Figure 9: The impact of block size in the approximate algorithm. We find that overly small blocks results in inefficient parallelization, while overly large blocks also slows down training due to cache misses.

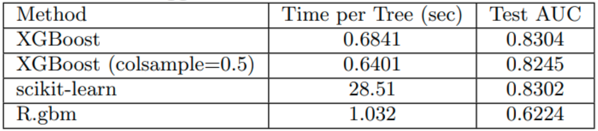
Datasets and results

Table 2: Dataset used in the Experiments.



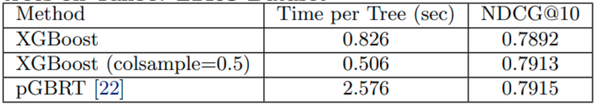
Dataset	n	m	Task
Allstate	10 M	4227	Insurance claim classification
Higgs Boson	10 M	28	Event classification
Yahoo LTRC	473K	700	Learning to Rank
Criteo	1.7 B	67	Click through rate prediction

Table 3: Comparison of Exact Greedy Methods with 500 trees on Higgs-1M data.



Method	Time per Tree (sec)	Test AUC
XGBoost	0.6841	0.8304
XGBoost (colsample=0.5)	0.6401	0.8245
scikit-learn	28.51	0.8302
R.gbm	1.032	0.6224

Table 4: Comparison of Learning to Rank with 500 trees on Yahoo! LTRC Dataset



Method	Time per Tree (sec)	NDCG@10
XGBoost	0.826	0.7892
XGBoost (colsample=0.5)	0.506	0.7913
pGBRT [22]	2.576	0.7915

Out-of-core experiment

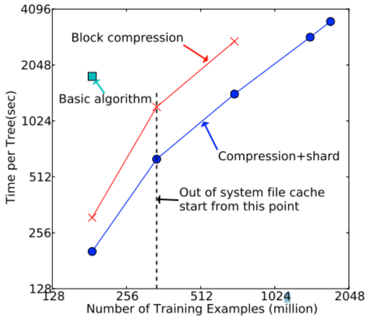
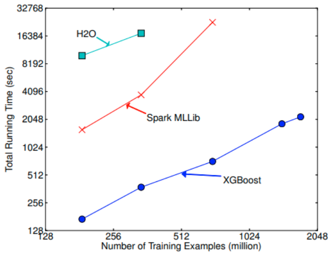
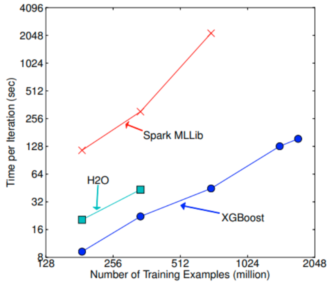


Figure 11: Comparison of out-of-core methods on different subsets of criteo data. The missing data points are due to out of disk space. We can find that basic algorithm can only handle 200M examples. Adding compression gives 3x speedup, and sharding into two disks gives another 2x speedup. The system runs out of file cache start from 400M examples. The algorithm really has to rely on disk after this point. The compression+shard method has a less dramatic slowdown when running out of file cache, and exhibits a linear trend afterwards.

Time Per Tree comparison



(a) End-to-end time cost include data loading



Distributed experiment

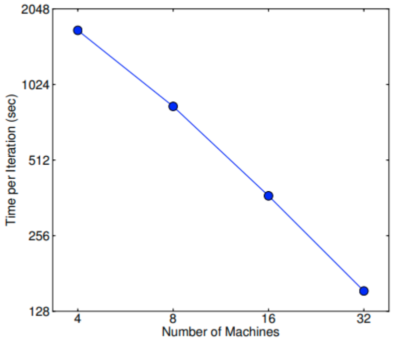
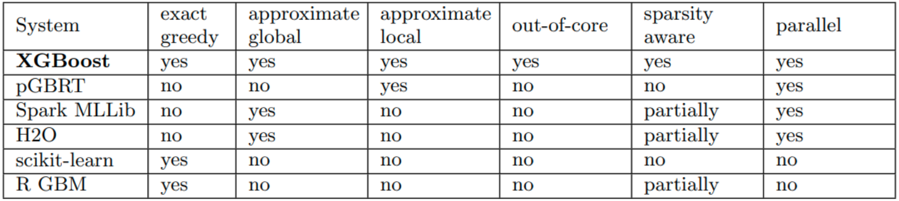


Figure 13: Scaling of XGBoost with different number of machines on criteo full 1.7 billion dataset. Using more machines results in more file cache and makes the system run faster, causing the trend to be slightly super linear. XGBoost can process the entire dataset using as little as four machines, and scales smoothly by utilizing more available resources.

Instead of a conclusion



System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost	yes	yes	yes	yes	yes	yes
pGBRT	no	no	yes	no	no	yes
Spark MLLib	no	yes	no	no	partially	yes
H2O	no	yes	no	no	partially	yes
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

XGBoost at the time of creation was a truly unique algorithm that was able to take gradient boosting to a new level of quality.

Among the 29 challenge winning solutions 3 published at Kaggle's blog during 2015, 17 solutions used XGBoost. For comparison, the second most popular method, deep neural nets, was used in 11 solutions.

The success of the system was also witnessed in KDDCup 2015, where XGBoost was used by every winning team in the top-10.