XGBoost: A Scalable Tree Boosting System

Tianqi Chen and Carlos Guestrin, University of Washington

XGBoost

- eXtreme Gradient Boosting
- 29 Kaggle challenges with winners in 2015
  - 17 used XGBoost
  - 8 of these solely used XGBoost; the others combined XGBoost with DNNs
- KDDCup 2015
  - Every single top 10 finisher used XGBoost

dmlc
XGBoost eXtreme Gradient Boosting

XGBoost Applications

- Store sales prediction
- High energy physics event classification
- Web text classification
- Customer behavior prediction
- Motion detection
- Ad click through rate prediction
- Malware classification
- Product categorization
- Hazard risk prediction
- Massive on-line course dropout rate prediction

dmlc
XGBoost eXtreme Gradient Boosting

Properties of XGBoost

- Single most important factor in its success: ***scalability***
- Due to several important systems and algorithmic optimizations

1. Highly scalable end-to-end tree boosting system
2. Theoretically justified weighted quantile sketch for efficient proposal calculation
3. Novel sparsity-aware algorithm for parallel tree learning
4. Effective cache-aware block structure for out-of-core tree learning

What is “tree boosting”?

- Given a dataset (n examples, m features)

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\} \ (|\mathcal{D}| = n, \mathbf{x}_i \in \mathbb{R}^m, y_i \in \mathbb{R})$$

- Tree ensemble uses K additive functions to predict output

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

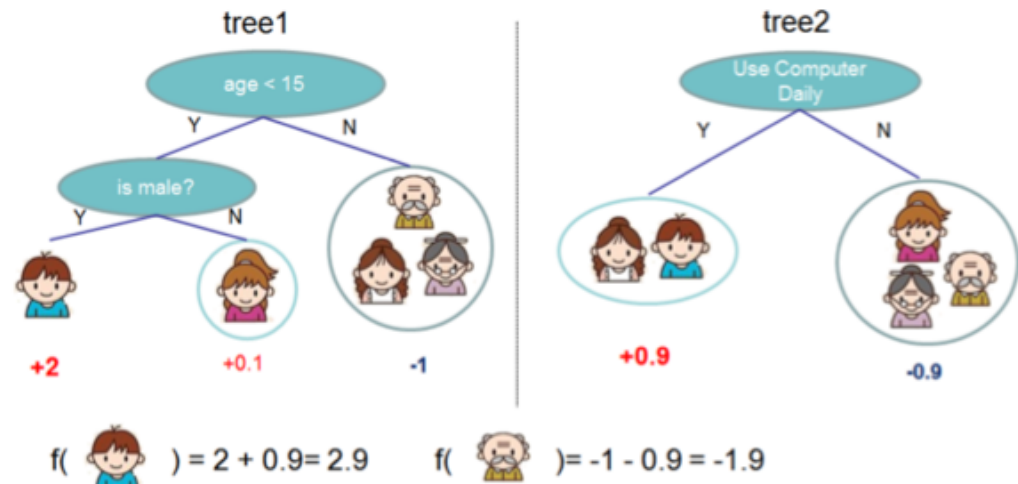


Figure 1: Tree Ensemble Model. The final prediction for a given example is the sum of predictions from each tree.

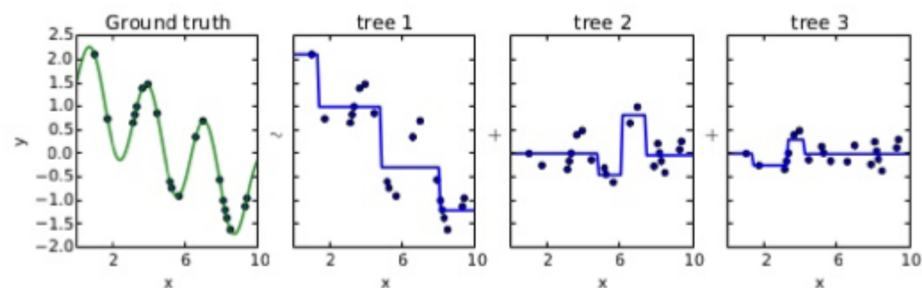
What is “gradient boosting”?

Gradient Boosting [J. Friedman, 1999]

Statistical view on boosting

- ⇒ Generalization of boosting to arbitrary loss functions

Residual fitting



sklearn.ensemble.GradientBoostingClassifier|Regressor

Regularized objective function

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

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

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

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

Regularized objective function

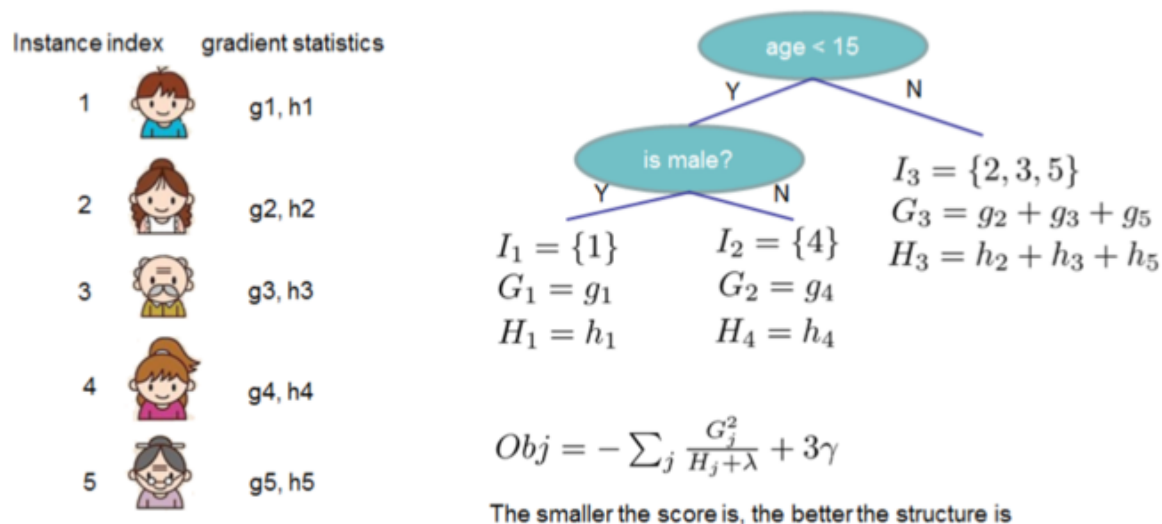


Figure 2: Structure Score Calculation. We only need to sum up the gradient and second order gradient statistics on each leaf, then apply the scoring formula to get the quality score.

Split-finding algorithms

- 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

- Two variants
  - Global
  - Local

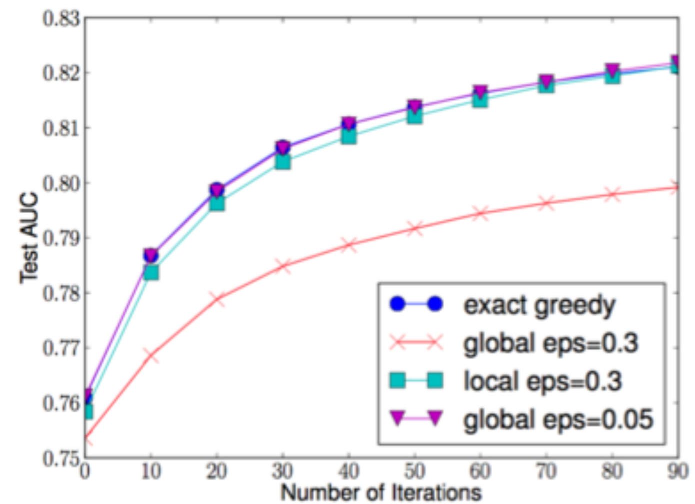


Figure 3: Comparison of test AUC convergence on Higgs 10M dataset. The eps parameter corresponds to the accuracy of the approximate sketch. This roughly translates to 1 / eps buckets in the proposal. We find that local proposals require fewer buckets, because it refine split candidates.

Shrinkage and column subsampling

- Shrinkage
  - Scales newly added weights by a factor η
  - Reduces influence of each individual tree
  - Leaves space for future trees to improve model
  - Similar to learning rate in stochastic optimization
- Column subsampling
  - Subsample features
  - Used in Random Forests
  - Prevents overfitting more effectively than row-sampling

Sparsity-aware split finding

- Equates sparsity with missing values
- Defines a "default" direction: follow the observed paths
- Compare to "dense" method

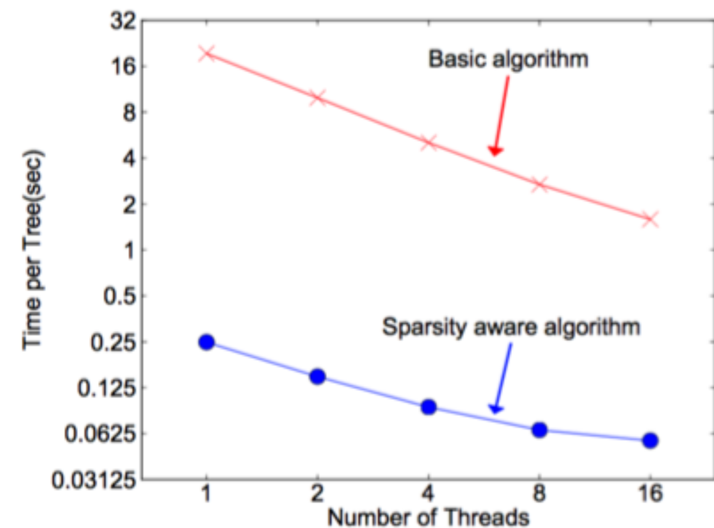


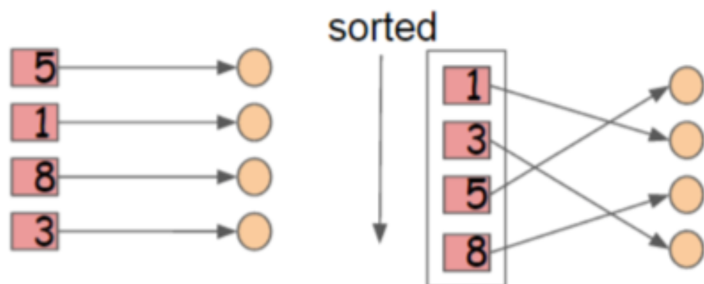
Figure 5: Impact of the sparsity aware algorithm on Allstate-10K. The dataset is sparse mainly due to one-hot encoding. The sparsity aware algorithm is more than 50 times faster than the naive version that does not take sparsity into consideration.

How does this work?

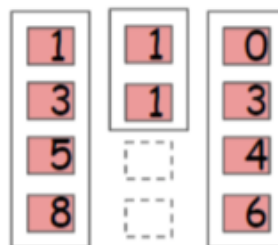
- Features need to be in sorted order to determine splits
- Concept of *blocks*
  - Compressed column (CSC) format
  - Each column sorted by corresponding feature value
- Exact greedy algorithm: all the data in a single block
- Data are sorted once before training and used subsequently in this format

Feature transformations in blocks

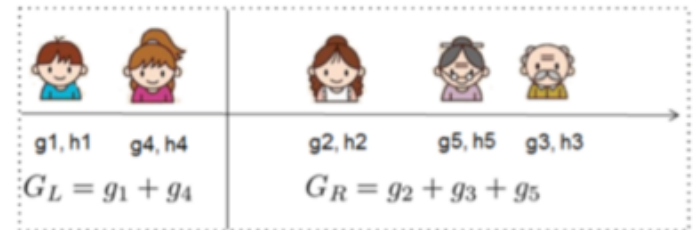
Layout Transformation of one Feature (Column)

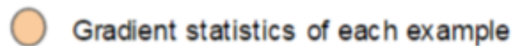


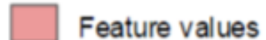
The Input Layout of Three Feature Columns

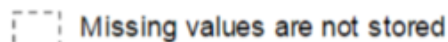


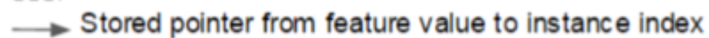
Linear scan over presorted columns to find best split











More on blocks

- 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
  - Cuts down on disk I/O
- **Block Sharding**
  - Data is split across multiple disks (i.e. cluster)
  - Pre-fetcher is assigned to each disk to read data into memory

Cache-aware access

Exact Greedy Algorithm

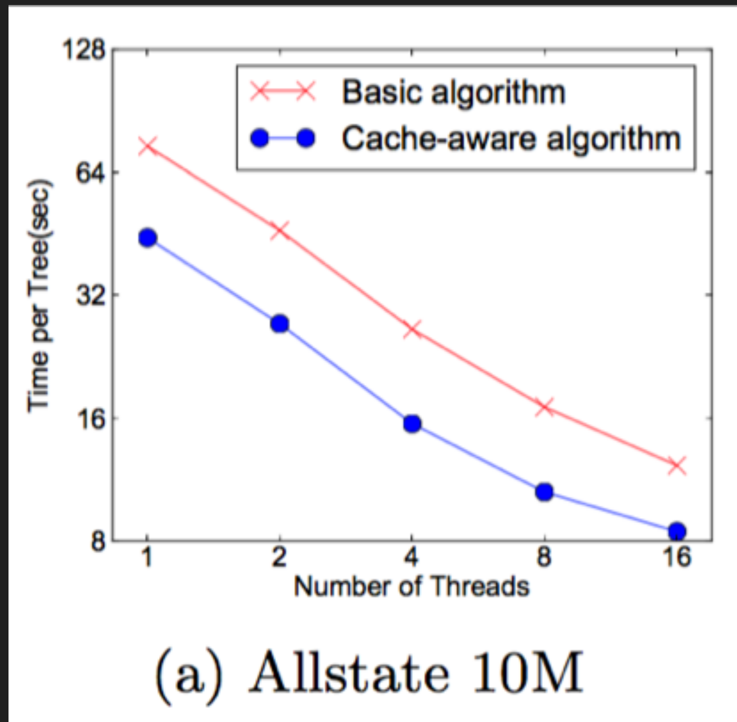
- Allocate an internal buffer in each thread
- Fetch gradient statistics
- Perform accumulation in mini-batch
- Reduces runtime overhead when number of rows is large

Approximate Algorithms

- Choice of block size is critical
- 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

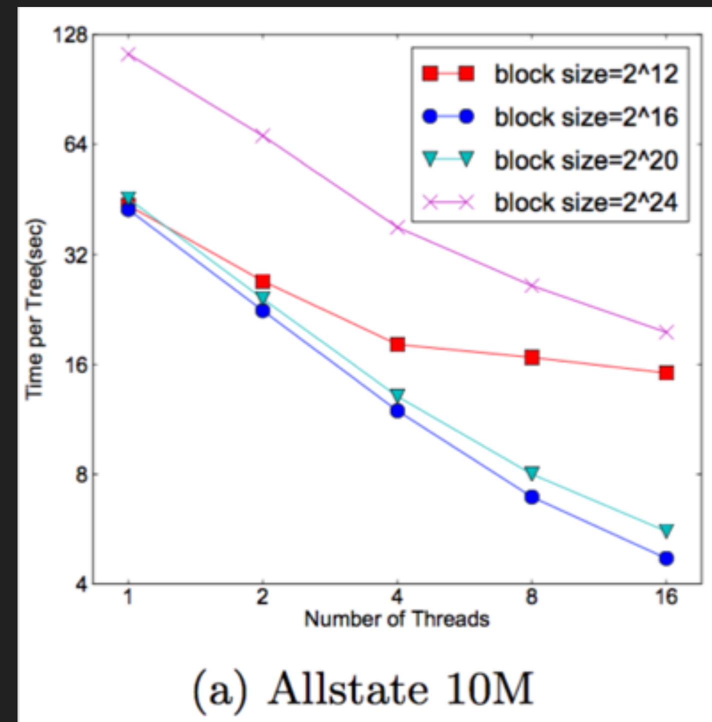
Cache-aware access

Exact



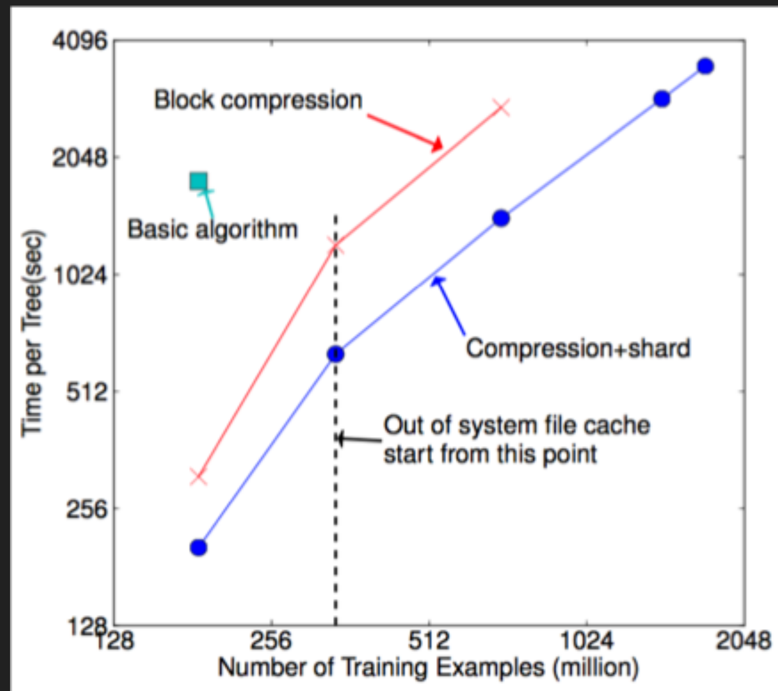
(a) Allstate 10M

Approximate

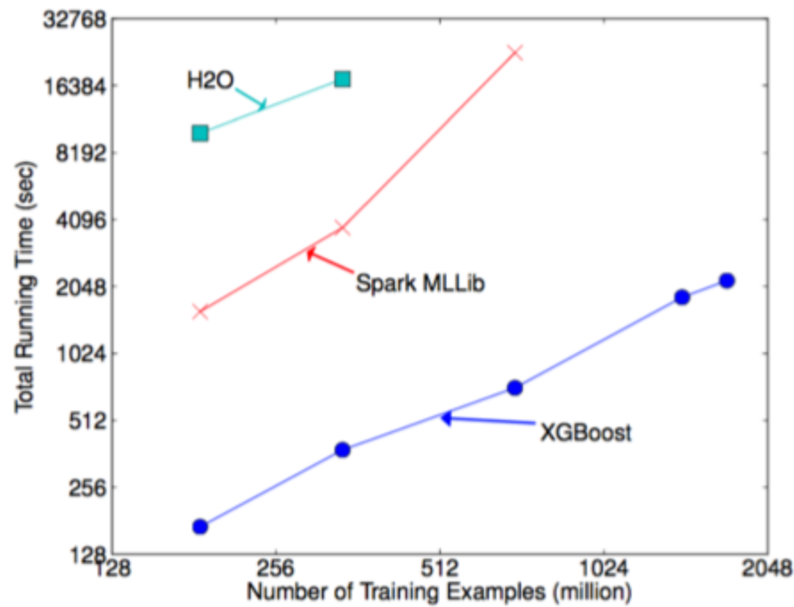


(a) Allstate 10M

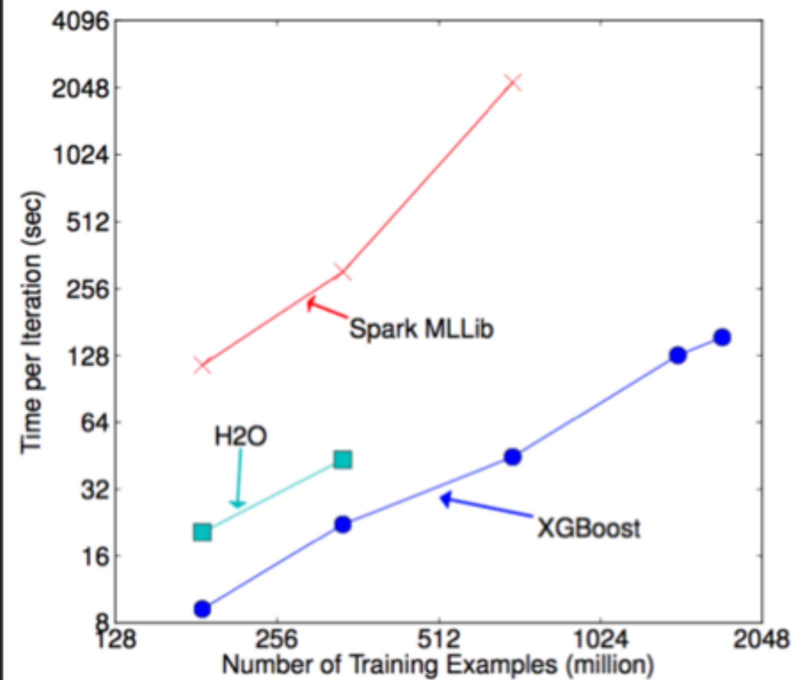
Results: out of core



Results: distributed



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



(b) Per iteration cost exclude data loading

Results: scalability

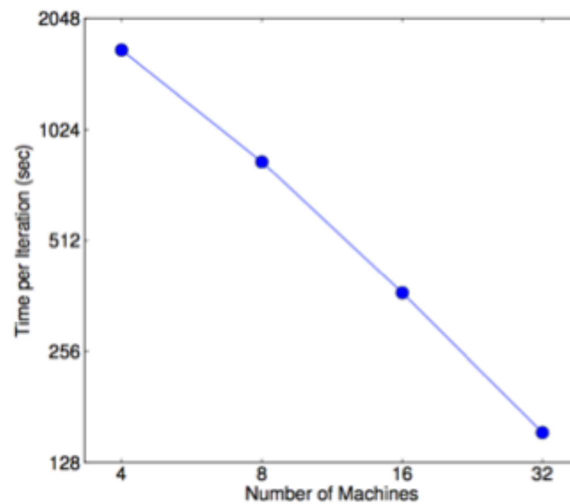


Figure 13: Scaling of XGBoost with different number of machines on criteo full 1.7 billion dataset. Using more machines results in more file cache and makes the system run faster, causing the trend to be slightly super linear. XGBoost can process the entire dataset using as little as four machines, and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

Demonstration

https://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html

Conclusions

- Novel sparsity-aware algorithm for handling sparse data
- Theoretical guarantees for weighted quantile sketching for approximate learning
- Cache access patterns, data compression, and data sharding techniques

arXiv.org > cs > arXiv:1603.02754

Computer Science > Learning

XGBoost: A Scalable Tree Boosting System

Tianqi Chen, Carlos Guestrin

(Submitted on 9 Mar 2016 (v1), last revised 10 Jun 2016 (this version, v3))

Tree boosting is a highly effective and widely used machine learning method. In this paper, we describe a scalable end-to-end tree boosting system called XGBoost, which is used widely by data scientists to achieve state-of-the-art results on many machine learning challenges. We propose a novel sparsity-aware algorithm for sparse data and weighted quantile sketch for approximate tree learning. More importantly, we provide insights on cache access patterns, data compression and sharding to build a scalable tree boosting system. By combining these insights, XGBoost scales beyond billions of examples using far fewer resources than existing systems.

Comments: KDD'16 changed all figures to type1
Subjects: **Learning (cs.LG)**
DOI: 10.1145/2939672.2939785
Cite as: arXiv:1603.02754 [cs.LG]
(or arXiv:1603.02754v3 [cs.LG] for this version)

Submission history

From: Tianqi Chen [view email]
[v1] Wed, 9 Mar 2016 01:11:51 GMT (592kb,D)
[v2] Mon, 23 May 2016 22:11:40 GMT (625kb,D)
[v3] Fri, 10 Jun 2016 23:23:51 GMT (837kb,D)

http://arxiv.org/abs/1603.02754