XGBoost: A Scalable Tree Boosting System

Authors – Tianqi Chen, Carlos Guestrin
(University of Washington)

Ashley Lau

Tree Boosting in a Nutshell

What is tree boosting?

- Tree boosting is a supervised machine learning technique that involves the combination of several decision trees to form a single predictive model.
- Decision trees are built in sequence, with each new tree attempting to correct the errors made by the previous tree.
- Tree boosting can be used for classification and regression.

Regularized Learning Objective

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

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

Optimize the objective function using second order approximation

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

How to prevent overfitting

2 additional techniques to prevent overfitting:

- 1. Shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model
- 2. Column (feature) subsampling aims to reduce the number of features (columns) used in a model by selecting a subset of the features from the training data and using only those features to train the model

Split Finding Algorithms

Exact Greedy algorithm

- Find max score amongst all possible splits on all features
- Computationally demanding

Approximate algorithm (weighted quantile sketch)

- The algorithm proposes splitting points based on feature percentiles, discretizes continuous features into buckets using these points, aggregates statistics, and selects the best solution among proposals.

XGBoost System Design

Sparsity-aware split finding

- XGBoost can efficiently handle sparse data by classifying missing values into default directions of tree nodes so that algo focuses on non-missing values only

Column blocking for parallel compute

- To reduce time sorting data, XGBoost stores data in compressed column blocks allowing split finding algo to run in parallel
- Major time save!!

XGBoost System Design

Cache-awareness

- Depending on split finding algo used, XGBoost does data prefetching or chooses correct block size to reduce the amount of time that the processor spends waiting for data to be retrieved from main memory

Out-of-core computation

- To handle data that does not fit in main memory, data is divided into multiple blocks and stored separately on disks. Disk Reading takes up majority of compute time so 2 techniques are used to combat this:
  - 1. Block Compression
  - 2. Block Sharding

Table 1: Comparison of major tree boosting systems.

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

Experimentation

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

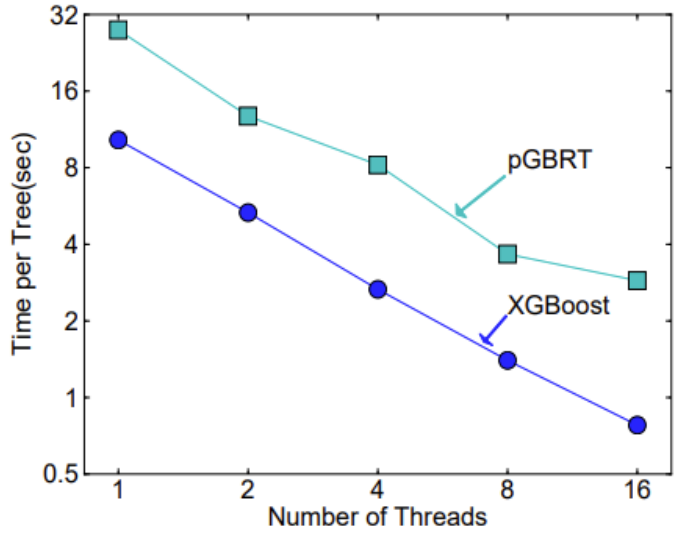
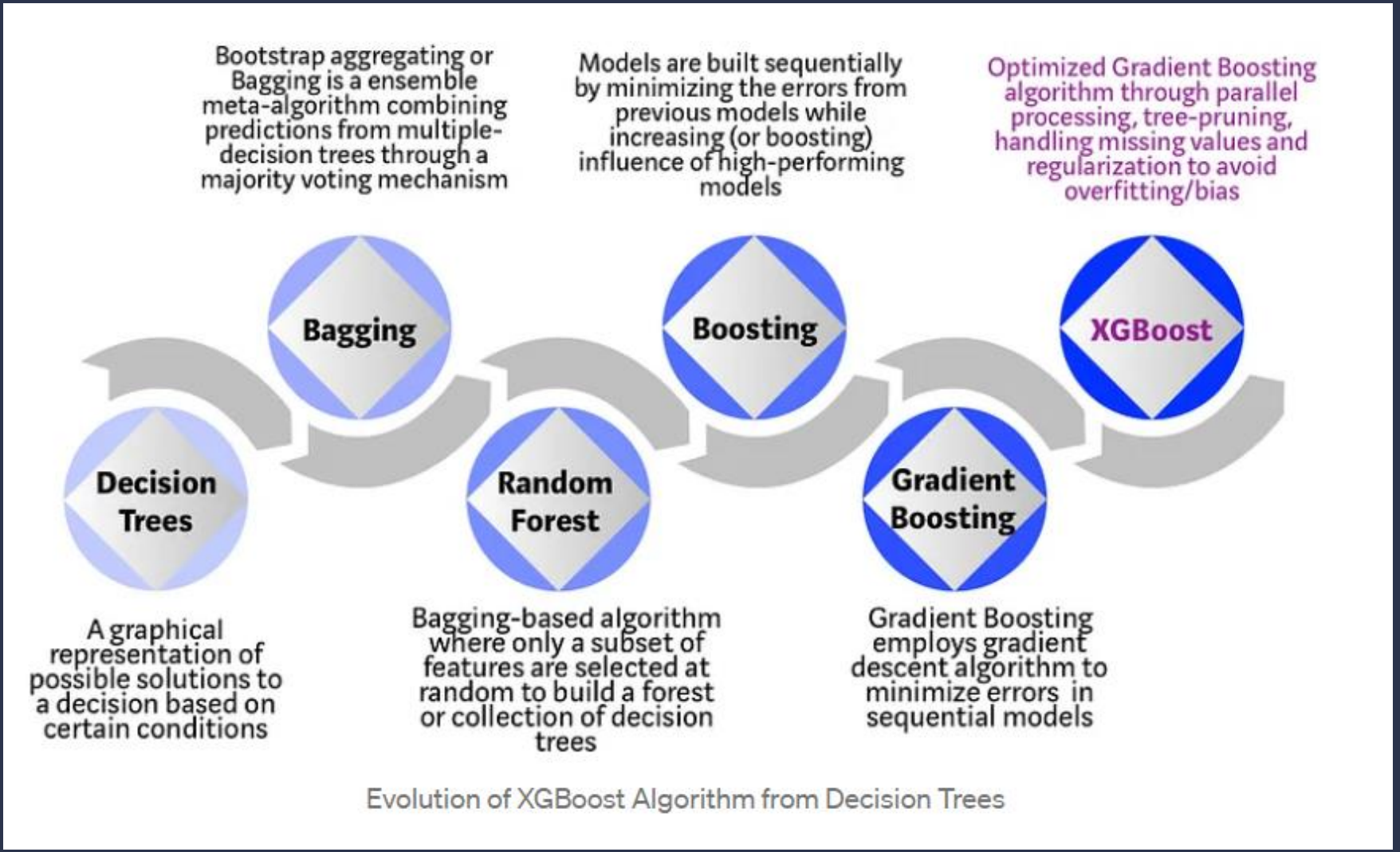


Figure 10: Comparison between XGBoost and pGBRT on Yahoo LTRC dataset.

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

Evolution of Tree Boosting

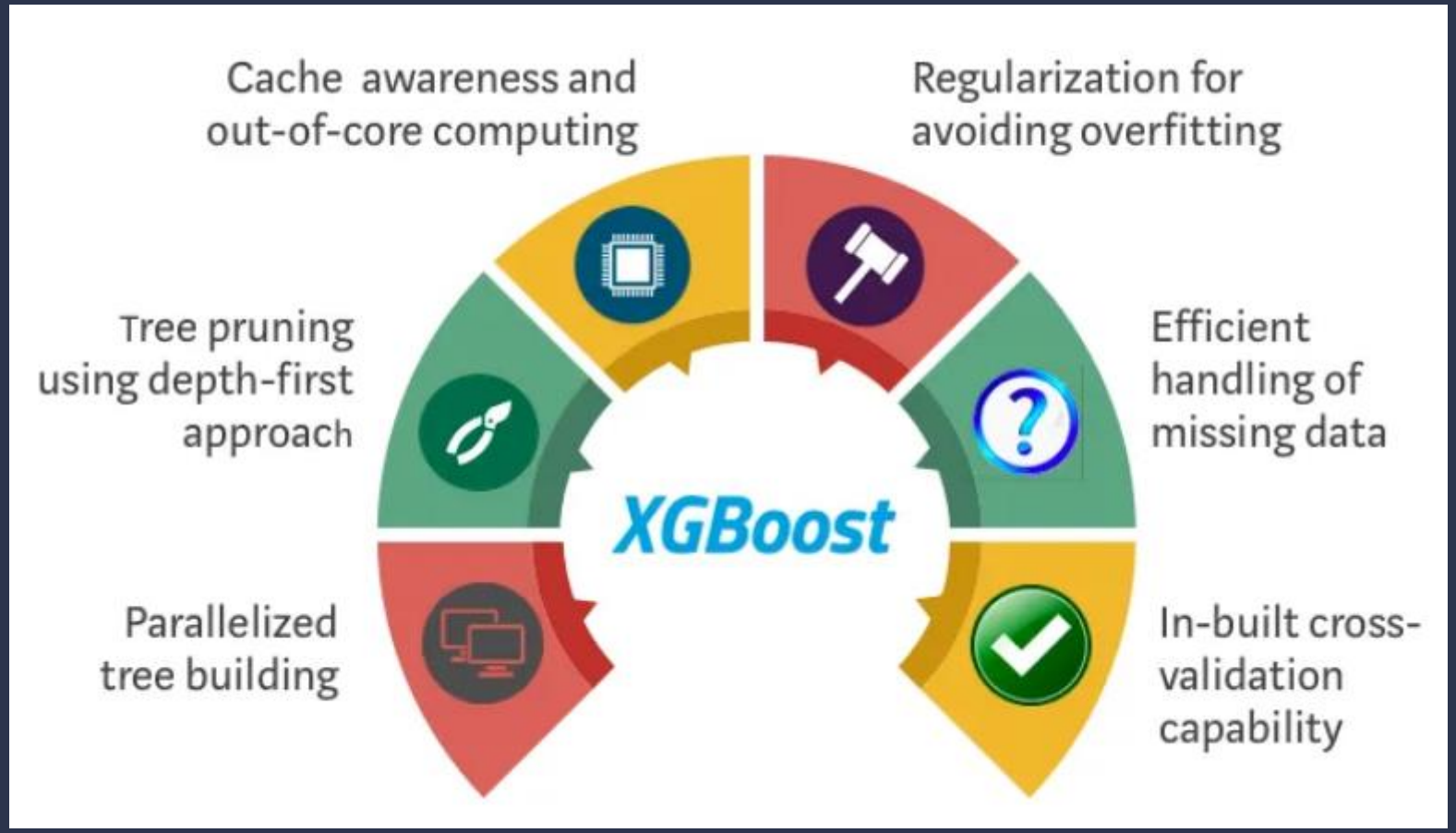


Evolution of XGBoost Algorithm from Decision Trees

How does XGBoost enhance field of Deep Learning?

- Award-winning algo of choice in previous Kaggle competitions
- Widely used across industry
- Open-source package: Github link

Thank you


Cache awareness and out-of-core computing
Regularization for avoiding overfitting
Tree pruning using depth-first approach
Efficient handling of missing data
XGBoost
Parallelized tree building
In-built cross-validation capability

Appendix

- [1]Chen, Tianqi, and Carlos Guestrin. Arxiv, 2016, *XGBoost: A Scalable Tree Boosting System*, arxiv.org/pdf/1603.02754.pdf. Accessed 10 Mar. 2023.
- [2]Dmlc. "DMLC/Xgboost: Scalable, Portable and Distributed Gradient Boosting (GBDT, GBRT or GBM) Library, for Python, R, Java, Scala, C++ and More. Runs on Single Machine, Hadoop, Spark, Dask, Flink and Dataflow." *GitHub*, https://github.com/dmlc/xgboost.
- [3]Morde, Vishal. "XGBoost Algorithm: Long May She Reign!" Medium, Towards Data Science, 8 Apr. 2019, https://towardsdatascience.com/https-medium-com-vishalmorde-xgboost-algorithm-long-she-may-rein-edd9f99be63d.

Algorithm Explainations

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
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 **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted*$(I,$ *by* $\mathbf{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

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.

Algorithm 3: Sparsity-aware Split Finding

Input: I, instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: d, feature dimension
Also applies to the approximate setting, only collect statistics of non-missing entries into buckets
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I}, g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  // enumerate missing value goto right
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for j *in sorted(I_k, ascent order by* $\mathbf{x}_{jk}$*)* **do**
    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
  // enumerate missing value goto left
  $G_R \leftarrow 0,\ H_R \leftarrow 0$
  for j *in sorted(I_k, descent order by* $\mathbf{x}_{jk}$*)* **do**
    $G_R \leftarrow G_R + g_j,\ H_R \leftarrow H_R + h_j$
    $G_L \leftarrow G - G_R,\ H_L \leftarrow H - H_R$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split and default directions with max gain