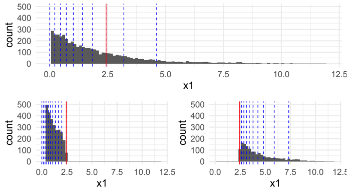
Introduction to Machine Learning

XGBoost



Learning goals

- XXX
- XXX

MOTIVATION

Chen and Guestrin (2016):
XGBoost (short for eXtreme Gradient Boosting) is an open-source software library with R/Python/Julia/Spark interface.

A scalable regularized tree boosting system:

- Efficient implementation in *C++*.
- Parallel (approximate) split finding.
- Handling of *sparse data*.
- Support of cluster-computing frameworks.

State-of-the-art machine learning method:

- Many winning solutions at Kaggle use XGBoost.
- KDDCup 2015 Top 10 used XGBoost.

LOSS MINIMIZATION

XGBoost uses a risk function with 3 regularization terms:

$$\mathcal{R}_{\text{reg}}^{[m]} = \sum_{i=1}^{n} L\left(y^{(i)}, f^{[m-1]}(\mathbf{x}^{(i)}) + b^{[m]}(\mathbf{x}^{(i)})\right) + \lambda_1 J_1(b^{[m]}) + \lambda_2 J_2(b^{[m]}) + \lambda_3 J_3(b^{[m]}),$$

with $J_1(b^{[m]}) = T^{[m]}$ to regularize tree depth, where $T^{[m]}$ is the number of leaves in the tree.

$J_2(b^{[m]}) = \left\|\mathbf{c}^{[m]}\right\|_2^2$ and $J_3(b^{[m]}) = \left\|\mathbf{c}^{[m]}\right\|_1$ are $L2$ and $L1$ regularizers of the terminal region scores $c_t^{[m]}, t = 1, \dots, T^{[m]}$.

Note: In the following we define
$J(b^{[m]}) := \lambda_1 J_1(b^{[m]}) + \lambda_2 J_2(b^{[m]}) + \lambda_3 J_3(b^{[m]})$ to simplify the notation.

LOSS MINIMIZATION

To approximate the loss in iteration m, use a second-order Taylor expansion around $f^{[m-1]}(\mathbf{x})$:

$$
\begin{aligned}
L(y, f^{[m-1]}(\mathbf{x}) + b^{[m]}(\mathbf{x})) \approx \\
L(y, f^{[m-1]}(\mathbf{x})) + g^{[m]}(\mathbf{x}) b^{[m]}(\mathbf{x}) + \frac{1}{2} h^{[m]}(\mathbf{x}) b^{[m]}(\mathbf{x})^2,
\end{aligned}
$$

with gradient

$$g^{[m]}(\mathbf{x}) = \frac{\partial L(y, f^{[m-1]}(\mathbf{x}))}{\partial f^{[m-1]}(\mathbf{x})}$$

and Hessian

$$h^{[m]}(\mathbf{x}) = \frac{\partial^2 L(y, f^{[m-1]}(\mathbf{x}))}{\partial f^{[m-1]}(\mathbf{x})^2}.$$

Note: $g^{[m]}(\mathbf{x})$ are the negative pseudo-residuals $-\tilde{r}^{[m]}$ we use in standard gradient boosting to determine the direction of the update.

LOSS MINIMIZATION

Since $L(y, f^{[m-1]}(\mathbf{x}))$ is a constant, the optimization criterion simplifies to

$$\begin{aligned}\mathcal{R}_{\text{reg}}^{[m]} &\approx \sum_{i=1}^{n} g^{[m]}(\mathbf{x}^{(i)}) b^{[m]}(\mathbf{x}^{(i)}) + \frac{1}{2} h^{[m]}(\mathbf{x}^{(i)}) b^{[m]}(\mathbf{x}^{(i)})^2 + J(b^{[m]}) + const \\ &\propto \sum_{t=1}^{T^{[m]}} \sum_{\mathbf{x}^{(i)} \in R_t^{[m]}} g^{[m]}(\mathbf{x}^{(i)}) c_t^{[m]} + \frac{1}{2} h^{[m]}(\mathbf{x}^{(i)}) (c_t^{[m]})^2 + J(b^{[m]}).\end{aligned}$$

Defining $G_t^{[m]}$ and $H_t^{[m]}$ as the sum of the gradients and Hessians, respectively, in terminal node t yields

$$= \sum_{t=1}^{T^{[m]}} G_t^{[m]} c_t^{[m]} + \frac{1}{2} H_t^{[m]} (c_t^{[m]})^2 + J(b^{[m]}).$$

LOSS MINIMIZATION

Expanding $J(b^{[m]})$, we get

$$\begin{aligned}\mathcal{R}_{\text{reg}}^{[m]} &= \sum_{t=1}^{T^{[m]}} \left(G_t^{[m]} c_t^{[m]} + \frac{1}{2} H_t^{[m]} (c_t^{[m]})^2 + \frac{1}{2}\lambda_2 (c_t^{[m]})^2 + \lambda_3 |c_t^{[m]}| \right) + \lambda_1 T^{[m]} \\ &= \sum_{t=1}^{T^{[m]}} \left(G_t^{[m]} c_t^{[m]} + \frac{1}{2} (H_t^{[m]} + \lambda_2)(c_t^{[m]})^2 + \lambda_3 |c_t^{[m]}| \right) + \lambda_1 T^{[m]}.\end{aligned}$$

Note: The factor $\frac{1}{2}$ is added to the $L2$ regularization to simplify the notation as shown in the second step. This does not impact estimation since we can just define $\lambda_2 = 2\tilde{\lambda}_2$.

LOSS MINIMIZATION

Since

$$\frac{\partial \mathcal{R}_{\text{reg}}^{[m]}}{\partial c_t^{[m]}} = \begin{cases} (G_t^{[m]} - \lambda_3) + (H_t^{[m]} + \lambda_2) c_t^m & \text{for } c_t^m < 0 \\ (G_t^{[m]} + \lambda_3) + (H_t^{[m]} + \lambda_2) c_t^m & \text{for } c_t^m > 0, \end{cases}$$

the optimal weights $\tilde{c}_1^{[m]}, \dots, \tilde{c}_{T^{[m]}}^{[m]}$ can then be calculated as

$$\tilde{c}_t^{[m]} = -\frac{t_{\lambda_3}\left(G_t^{[m]}\right)}{H_t^{[m]} + \lambda_2}, t = 1, \dots T^{[m]},$$

with

$$t_{\lambda_3}(x) = \begin{cases} x - \lambda_3 & \text{for } x > \lambda_3 \\ x + \lambda_3 & \text{for } x < -\lambda_3 \\ 0 & \text{for } |x| \leq \lambda_3. \end{cases}$$

LOSS MINIMIZATION - SPLIT FINDING

To evaluate the performance of a candidate split that divides the instances in region $R_t^{[m]}$ into a left and right node we use the **risk reduction** achieved by that split:

$$\tilde{S}_{LR} = \frac{1}{2}\left[\frac{t_{\lambda_3}\left(G_{tL}^{[m]}\right)^2}{H_{tL}^{[m]} + \lambda_2} + \frac{t_{\lambda_3}\left(G_{tR}^{[m]}\right)^2}{H_{tR}^{[m]} + \lambda_2} - \frac{t_{\lambda_3}\left(G_{t}^{[m]}\right)^2}{H_{t}^{[m]} + \lambda_2}\right] - \lambda_1,$$

where the subscripts L and R denote the left and right leaves after the split.

LOSS MINIMIZATION - SPLIT FINDING

Algorithm (Exact) Algorithm for split finding

```
 1: Input $I$: instance set of current node
 2: Input $p$: dimension of feature space
 3: $gain \leftarrow 0$
 4: $G \leftarrow \sum_{i \in I} g(\mathbf{x}^{(i)}), H \leftarrow \sum_{i \in I} h(\mathbf{x}^{(i)})$
 5: for $j = 1 \rightarrow p$ do
 6:     $G_L \leftarrow 0, H_L \leftarrow 0$
 7:     for $i$ in sorted($I$, by $x_j$) do
 8:         $G_L \leftarrow G_L + g(\mathbf{x}^{(i)}), H_L \leftarrow H_L + h(\mathbf{x}^{(i)})$
 9:         $G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$
10:         compute $\tilde{S}_{LR}$
11:     end for
12: end for
13: Output Split with maximal $\tilde{S}_{LR}$
```

APPROXIMATE SPLIT-FINDING ALGORITHMS

Three different algorithms to search for these splits are implemented in XGBoost.

The **exact greedy algorithm** iterates over all possible splits of all features.

The **global approximate algorithm** iterates over percentiles of the empirical distribution of each feature.

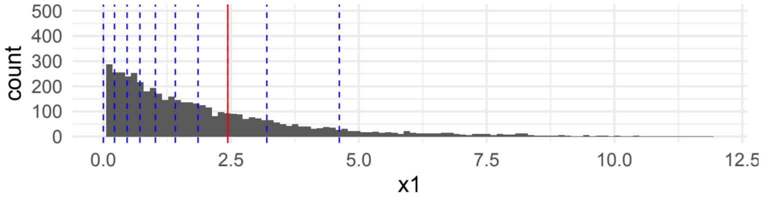
The **local approximate algorithm** does the same as the global, but recalculates the percentiles after each split.

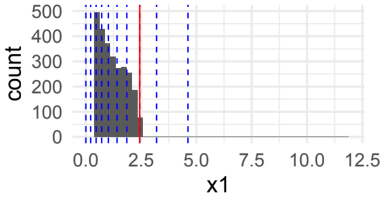
APPROXIMATE SPLIT-FINDING ALGORITHMS

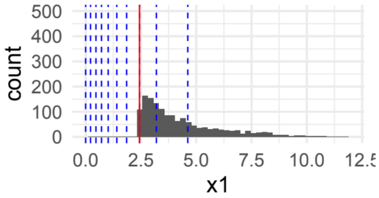
Algorithm Approximate algorithm for split finding

```
1: for $j = 1 \to p$ do
2:     Define possible split proposals $S_j = \{s_j^{(1)}, s_j^{(2)}, \ldots, s_j^{(l)}\}$ by percentiles on feature $j$.
3:     Proposal can be done once per tree (global), or in each node (local).
4: end for
5: for $j = 1 \to p$ do
6:     $G_{kv} \leftarrow \sum_{i \in \{i | s_j^{(v)} \geq x_j^{(i)} > s_k^{(v-1)}\}} g(\mathbf{x}^{(i)})$
7:     $H_{kv} \leftarrow \sum_{i \in \{i | s_j^{(v)} \geq x_j^{(i)} > s_k^{(v-1)}\}} h(\mathbf{x}^{(i)})$
8: end for
9: Follow same steps as exact algorithm to find max score only among proposed splits.
```

APPROXIMATE SPLIT-FINDING ALGORITHMS



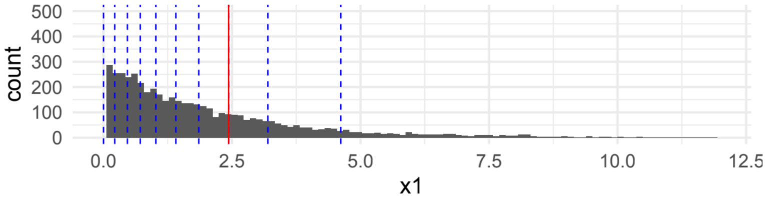


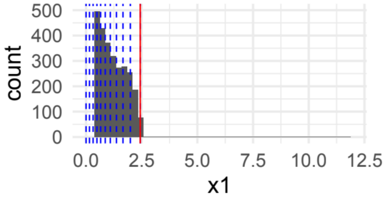


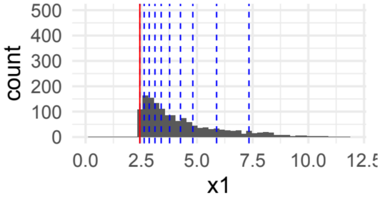
Approximate **global** split finding.

Blue lines indicate percentiles, red line is the chosen split. Proposals are not recomputed, allowing the right node only two possibilities.

APPROXIMATE SPLIT-FINDING ALGORITHMS







Approximate **local** split finding.
Blue lines indicate percentiles, red line is the chosen split. Percentiles are recalculated after each split.

APPROXIMATE SPLIT-FINDING ALGORITHMS

Both approximate splitting algorithms are useful for large datasets as iteration over all splits becomes very expensive.

XGBoost also allows for sparsity-aware split finding, which means that for a large number of zero (or missing) values, a default direction in each node is introduced.

The calculation of the percentiles in the feature distributions can be parallelized, allowing parallel split finding for the approximate algorithms.

EVEN MORE REGULARIZATION

Besides the regularization terms discussed above, XGBoost also incorporates a step length ν for the updates.

Furthermore, it uses **feature subsampling** similar to the random forest. Only a random subset of all features is considered for each split.

Stochastic gradient boosting, as introduced before, is utilized as well, which means that in each iteration only a subset of the observations is used.

A final possibility for regularization is DART (Dropout Additive Regression Trees). For each iteration only a fraction of previously fitted trees is considered. After an update the joint contribution of the dropped tree and the tree new to the ensemble are scaled to avoid overshooting.

STORAGE AND OUT-OF-MEMORY CALCULATION

The data is stored in **blocks.** Each of these blocks is stored in a compressed column format.

Depending on the split finding algorithm, these blocks contain the complete (sorted) data, or subsets of rows.

These blocks can be stored out-of-memory (on disk) to save memory and distributed across multiple machines for parallel computations.

For more details on this see *Chen and Guestrin (2016)*.

SUMMARY

XGBoost is an extremely powerful method, but also hard to configure correctly. Overall, eight hyperparameters have to be set, which is difficult to do in practice and almost always requires tuning.

Different split finding algorithms can be selected, which allows XGBoost to be efficient even on very large datasets.

A large number of different regularization strategies is included to prevent overfitting.

COMPARISON OF MAJOR BOOSTING SYSTEMS

System	Exact algo.	Approx. algo.	Sparsity-aware	Variable importance	Parallel	Language
ada	yes	no	no	no	no	R
GBM	yes	no	partially	yes	no	R
mboost	yes	no	no	no	no	R
compboost	yes	no	yes	yes	yes	R
H2O	no	yes	partially	yes	yes	R (Java)
XGBoost	yes	yes	yes	yes	yes	R + Python
lightGBM	no	yes	yes	yes	yes	R + Python
catboost	no	yes	no	yes	yes	R + Python
scikit-learn	yes	no	no	yes	no	Python
pGBRT	no	no	no	no	yes	Python
Spark MLLib	no	yes	partially	yes	yes	R, Python, Java, Scala

Note: H2O is a commercial software written in Java with a solid R interface. In the free version only two CPUs can be used.