self-study: Gradient Boost Trees and XGBoost

Tianpei Xie

Jun. 24th., 2022

Contents

1 Tree-based methods

In most of applications for boosting methods, the basis function class $\mathcal{F}$ is the family of **tree-based models**. A tree-based model can be represented as weighted sum of region indicators:

$$f_{\mathcal{T}}(\boldsymbol{x}) = \sum_{m=1}^{T} w_m \mathbb{1}\left\{\boldsymbol{x} \in \mathcal{R}_m\right\}$$

where $\mathcal{R}_m$ is the region in $\mathcal{R}^k$ partioned according to a binary tree $\mathcal{T}$.The partition $\{\mathcal{R}_m, m = 1, \ldots, T\}$ is generated by branching at one input feature x^j at a time. Each **terminal point** or **leaves** of the tree represent a partitioned regin. Tree-based ensemble is a piece-wise constant function, therefore the class of ensemble trees can also be represented as

$$\mathcal{F} \equiv \left\{f_{\mathcal{T}}(\boldsymbol{x}) = \boldsymbol{w}_{q(\boldsymbol{x})} : q \in \mathcal{T}, \forall \mathcal{T}\right\}$$

where $q : \mathbb{R}^d \to 1, 2, \ldots, T$ is the tree-structured mapping that maps input data to a region index and $\boldsymbol{w} = [w_1, \ldots, c_T] \in \mathbb{R}^T$. Given the region partitions $\mathcal{R}_m$, c_m is easy to inference by simpling taking the mean of the target for samples in the region in **regression tree** or by taking the misclassification error or cross entropy in **classification tree**.

Note that tree-based ensemble did partition the input space into several regions similar to clustering, but the partition is selected to optimize the loss with respect to target y, while for the clustering method the partition is selected to optimize some metrics on the input data $\boldsymbol{x}$. There are several advantages when using tree-based methods [Hastie et al., 2009]:

- Making predictions is **fast** (no complicated calculations, just looking up constants in the tree)
- Its easy to understand / **interpret** what variables are important in making the prediction (look at the tree)
- If some data is *missing*, we might not be able to go all the way down the tree to a leaf, but we can still make a prediction by averaging all the leaves in the sub-tree we do reach
- The model gives a jagged response, so it can work when the true regression surface is not smooth. If it is smooth, though, the piecewise-constant surface can approximate it arbitrarily closely (with enough leaves)
- There are fast, reliable algorithms to learn these trees

Learning a tree ensemble f can be done by optimizing the objective function

$$\begin{aligned}\mathcal{L}(f) &:= \sum_i \mathcal{L}(y_i, \hat{y}_i) + \gamma\Omega(\mathcal{T}) + \frac{\alpha}{2}\|\boldsymbol{w}\|_2^2 \\ &= \sum_{m=1}^{T}\sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} \mathcal{L}(y_i, w_m) + \gamma\Omega(\mathcal{T}) + \frac{\alpha}{2}\sum_{m=1}^{T} w_m^2\end{aligned}$$

where $\mathcal{L}$ is the regression loss function such as mean squared error, $\hat{y}_i = f(\boldsymbol{x}_i)$ is the predicted output of the ensemble given i-th sample $\boldsymbol{x}_i$. The second term $\Omega(\cdot)$ measure the complexity of the tree $\mathcal{T}$. The last term is a smoothing factor. Since the objective function is non-smooth, the solution is based on a ***recursive greedy algorithm in top-down manner***:

- begin with the root node, finding one feature that split the input space into two nodes/regions;

- in order to find the optimal split, for each region, estimate w_j by taking average of responses within the region
- compute the loss / residual for the esimate in each region; the optimal split is the one tha minimize the objective function
- for each region, repeat previous steps to further partition
- stop growing the tree when some criteria are met

Usually we can make sure that the tree stop growing when some minimal node size has been reached. When tree grow too large, we will reduce the tree complexity by using ***cost-complexity pruning***. Let $\mathcal{T} \subset \mathcal{T}_0$ be a tree obtained after prunning $\mathcal{T}_0$, i.e. by collapsing any number of non-terminal nodes (internal nodes) in $\mathcal{T}_0$. Let $\mathcal{R}_m, m = 1, \ldots, T$ be regions partitioned via terminal nodes.

For **regression tree**, we can define cost complexity criterion as

$$C_\gamma(\mathcal{T}) = \sum_{m=1}^{T} \sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} (y_i - \hat{w}_m)^2 + \gamma T$$

where $\hat{w}_m = \frac{1}{|\{\boldsymbol{x}_i \in \mathcal{R}_m\}|} \sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} y_i$. $Q_m(\mathcal{T}) := \sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} (y_i - \hat{w}_m)^2 / N_m$ is the variance or residual of the loss [Hastie et al., 2009]. Then the idea is to find, for each γ, $\mathcal{T}_\lambda \subseteq \mathcal{T}_0$ to minimize the $C_\gamma(\mathcal{T})$. γ is the tuning parameters which balance the goodness-of-fit and the model complexity. It can be shown that for given γ there exists a *unique* tree $\mathcal{T}_\gamma$ that minimize the cost complexity criterion. This solution can be found via ***weakest link pruning***: i.e. we successively collaps two internal nodes that produce the smallest per-node increase in residual. We continue this process until we reach out to the root. The process gives us a sequence of trees and the optimal $\mathcal{T}_\gamma$ must exist within this sequence, i.e. the tree that minimize the cost among trees in the sequence is the optimial solution.

For **classification tree**, the *residual* $Q_m(\mathcal{T}) := \sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} (y_i - \hat{w}_m)^2 / N_m$ in each region can be replaced by some *impurity measure*. Note that $\hat{p}_{mk} := \sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} \mathbb{1}\{y_i = k\} / N_m$ is the proportion of label k in the region. The classification within the region can be done via majority vote $k_m := \operatorname{argmax}_k \hat{p}_{mk}$. Some commonly used impurity measure are

- misclassification error: $1 - \hat{p}_{mk_m} := \sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} \mathbb{1}\{y_i \neq k_m\} / N_m$
- *Gini index*: $\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'} = \sum_k^K \hat{p}_{mk}(1 - \hat{p}_{mk})$
- cross-entropy: $-\sum_{k=1}^K \hat{p}_{mk} \log \hat{p}_{mk}$

Both cross-entropy and Gini Index are more sensitive to node probabililty change than misclassification error.

Tree-based methods can handle *missing value* more flexibly. First, it can treat the missing value as additional category in categorical features. This allows us to encode the pattern of missing value itself into the model. Second, it can construct surrogate variables. When considering a predictor for split, we use only observations for which the predictor is not missing. Having choosing the optimal primary predictor and split, we form a list of *surrogate predictors and split points*. The first surrogate is the predictor and corresponding split point that best mimic the split of training data done by the primary predictor. The second is the one secondly best mimic the primary split. Then in prediction phase, let the observations go through the surrogate predictors and splits *in* *order*. The higher the **correlation** between missing/primary predictor and surrogate predictors, the smaller the loss of information due to missing value.

Tree-based methods are known to be of **higher variance** due to the greedy splits it generates. A small change of data will result in a different series of splits, making interpretation more precautious. The major reason is due ot the hierarchical nature of the tree-building process: the effect of error in root split will be propagated to all splits below. Ensemble methods such as boosting will help to reduce the variance for tree-based model.

2 Boosting

The basic idea of ***boosting*** is to combine multiple *weak* learners in sequential manner in order to boost the performance of joint model. Boosting belongs to Generalized Additive Models

$$y = \sum_{j=1}^{p} f_j(\boldsymbol{x}_i) + \epsilon$$

where $f_j : \mathbb{R}^d \to \mathbb{R}$ or $f_j : \mathbb{R}^d \to \{0, 1\}$ are classifiers belong to a family of simple functions $\mathcal{F}$. Similar to classical Fourier transform, each f_j are able to capure a simple relationship between covariate variables and the response variables, but combining them together, the model is expected to cover both broad and detailed aspects of the target distribution. Boosting belongs to *Ensemble Methods*, which all seeks to improve the performance of machine learning model by joining multiple models together.

2.1 Boosting as Forward Stagewise Additive Modeling

Learning in boosting is done stagewise sequentially. At each stage, a new model f_j is learned and added to the ensemble, so that the overall objective is minimized. We can use the following function to define the objective of boosting model:

$$\begin{aligned}
\mathcal{L}(f) &:= \sum_i \mathcal{L}\left(y_i, \hat{y}_i\right) + \gamma \sum_j \Omega(\mathcal{T}_j) + \frac{\alpha}{2} \sum_j \|\boldsymbol{w}_j\|_2^2 \\
&= \sum_i \mathcal{L}\left(y_i, \sum_{j=1}^{p} \beta_j f_j(\boldsymbol{x}_i)\right) + \gamma \sum_j \Omega(\mathcal{T}_j) + \frac{\alpha}{2} \sum_j \|\boldsymbol{w}_j\|_2^2
\end{aligned}$$

where $\hat{y}_i = \sum_{j=1}^{p} f_j(\boldsymbol{x}_i)$ and $f_j \in \mathcal{F}$ of tree ensembles. Then at stage k, the objective can be modified as

$$\begin{aligned}
\mathcal{L}_k(f_k|f^{k-1}) &:= \sum_i \mathcal{L}\left(y_i, \sum_{j=1}^{k-1} \beta_j f_j(\boldsymbol{x}_i) + \beta_k f_k(\boldsymbol{x}_i)\right) + \gamma \sum_j^{k-1} \Omega(\mathcal{T}_j) + \frac{\alpha}{2} \sum_j^{k-1} \|\boldsymbol{w}_j\|_2^2 \\
&= \sum_i \mathcal{L}\left(y_i, f^{k-1}(\boldsymbol{x}_i) + \beta_k f_k(\boldsymbol{x}_i)\right) + \gamma \sum_j^{k-1} \Omega(\mathcal{T}_j) + \frac{\alpha}{2} \sum_j^{k-1} \|\boldsymbol{w}_j\|_2^2 \qquad (1)
\end{aligned}$$

where $f^{k-1}(\boldsymbol{x}_i) = \sum_{j=1}^{k-1} \beta_j f_j(\boldsymbol{x}_i)$ is the ensemble learned at previous stage.

For **squared loss** $\mathcal{L}(y, f) := (y - f(x))^2$, the above equation (1) can be represented as

$$\begin{aligned}
\mathcal{L}_k(f_k|f^{k-1}) &= \sum_i \Big(y_i - f^{k-1}(\boldsymbol{x}_i) - \beta_k f_k(\boldsymbol{x}_i)\Big)^2 \\
&= \sum_i \left(r_{i,k} - \beta_k f_k(\boldsymbol{x}_i)\right)^2
\end{aligned}$$

where $r_{i,k} := y_i - f^{k-1}(\boldsymbol{x}_i)$ is the *residual* of the ensemble at previous stage at current observation. So for squared loss, the new model f_j is learned to be the best fit to the residual of previous ensemble. $ensemble \rightarrow residual \rightarrow new\_ensemble$.

For **exponential loss** $\mathcal{L}(y, f) := \exp(-y\, f(x))$, the (1) becomes AdaBoost.

$$\begin{aligned}
\mathcal{L}_k(f_k|f^{k-1}) &= \sum_i \exp\Big(-y_i(f^{k-1}(\boldsymbol{x}_i) + \beta_k f_k(\boldsymbol{x}_i))\Big) \\
&= \sum_i \exp\Big(-y_i f^{k-1}(\boldsymbol{x}_i)\Big) \exp\left(-y_i \beta_k f_k(\boldsymbol{x}_i)\right) \\
&= \sum_i D_{i,k-1} \exp\left(-y_i \beta_k f_k(\boldsymbol{x}_i)\right)
\end{aligned}$$

For tree-based model f, we can further decompose the equation (1) as

$$\begin{aligned}
\mathcal{L}_k(f_k|f^{k-1}) &:= \sum_i \mathcal{L}\Big(y_i, f^{k-1}(\boldsymbol{x}_i) + \beta_k f_k(\boldsymbol{x}_i)\Big) + \Omega(\mathcal{T}) \\
&= \sum_{m=1}^{T} \sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} \mathcal{L}\Big(y_i, f^{k-1}(\boldsymbol{x}_i) + \beta_k * w_m\Big) + \Omega(\mathcal{T}) \\
\mathcal{L}_k(\{\theta_{k,m}, \mathcal{R}_m\} | f^{k-1}) &:= \sum_{m=1}^{T} \sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} \mathcal{L}\Big(y_i, f^{k-1}(\boldsymbol{x}_i) + \theta_{k,m}\Big) + \Omega(\mathcal{T}) \qquad (2)
\end{aligned}$$

Note that the continuous score $\theta_{k,m}$ can be learned by minimizing

$$\sum_{i:\boldsymbol{x}_i \in \mathcal{R}_m} \mathcal{L}\Big(y_i, f^{k-1}(\boldsymbol{x}_i) + \theta_{k,m}\Big)$$

within each region $\mathcal{R}_m$. We can choose $\beta_k = \beta$ and call it the ***learning rate*** of boosting algorithm.

2.2 Gradient Boost Trees

The objective function (1) can be *approximated* via its first and second order approximation at f^{k-1}.

$$\begin{aligned}
\mathcal{L}_k(f_k|f^{k-1}) = \mathcal{L}_k(f) &:\approx \mathcal{L}_k(f^{k-1}) + \nabla_{\mathcal{L}}|_{f^{k-1}}^{T}\, \Delta f + \frac{1}{2} H_{\mathcal{L}}|_{f^{k-1}}\, \|\Delta f\|_2^2 \\
&= \sum_i \mathcal{L}\Big(y_i, f^{k-1}(\boldsymbol{x}_i)\Big) + \sum_i g_{i,k} f_k(\boldsymbol{x}_i) + \frac{1}{2}\sum_i h_{i,k} f_k^2(\boldsymbol{x}_i)
\end{aligned}$$

$$= const. + \sum_i g_{i,k} f_k(\boldsymbol{x}_i) + \frac{1}{2}\sum_i h_{i,k} f_k^2(\boldsymbol{x}_i) \tag{3}$$

$$= const. + \frac{1}{2}\sum_i h_{i,k}\left(f_k(\boldsymbol{x}_i) + \frac{g_{i,k}}{h_{i,k}}\right)^2 \tag{4}$$

where

$$g_{i,k} := \left[\frac{\partial \mathcal{L}(y_i, f(\boldsymbol{x}_i))}{\partial f(\boldsymbol{x}_i)}\right]_{f=f^{k-1}(\boldsymbol{x}_i)}$$

$$h_{i,k} = \left[\frac{\partial^2 \mathcal{L}(y_i, f(\boldsymbol{x}_i))}{\partial^2 f(\boldsymbol{x}_i)}\right]_{f=f^{k-1}(\boldsymbol{x}_i)}$$

are gradient and Hessian of the loss function w.r.t. functions f. Since these methods use gradient approximation to build boosting trees, it is called ***gradient boost methods***. Example of loss function and their first and second order derivatives:

- **squared loss** $\mathcal{L}(f) := (y-f)^2/2$: $g := (f-y)$, $h := -y$
- **binary cross-entropy loss (logistic loss)** $\mathcal{L}(f) := -(y\log(f) + (1-y)\log(1-f))$, $g := \frac{f-y}{(1-f)f}$, $h := \frac{y}{f^2} + \frac{1-y}{(1-f)^2}$

For tree-based models, equation (3) can be simplified as

$$\begin{aligned}
&\sum_{m=1}^{T}\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m}\left(g_{i,k}w_m + \frac{1}{2}h_{i,k}w_m^2\right) + \Omega(\mathcal{T})\\
&= \sum_{m=1}^{T}\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m}\left(g_{i,k}w_m + \frac{1}{2}h_{i,k}w_m^2\right) + \gamma T + \alpha\frac{1}{2}\sum_{m=1}^{T}\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m} w_m^2\\
&= \sum_{m=1}^{T}\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m}\left(g_{i,k}w_m + \frac{1}{2}(h_{i,k}+\alpha)w_m^2\right) + \gamma T. \qquad (5)
\end{aligned}$$

So if we fixed the tree structure, i.e. $\mathcal{R}_m, \forall m$, then finding the optimal weight (score) w_m can be done easily by minimizing the above equation. We have

$$\begin{aligned}
w_m^* &:= \text{argmin}_{w_m}\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m}\left(g_{i,k}w_m + \frac{1}{2}(h_{i,k}+\alpha)w_m^2\right)\\
&= \left(\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m} g_{i,k}\right)w_m + \frac{1}{2}\left(\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m}(h_{i,k}+\alpha)\right)w_m^2\\
&= -\frac{\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m} g_{i,k}}{\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m}(h_{i,k}+\alpha)} \qquad (6)
\end{aligned}$$

and the optimal objective value is

$$\hat{\mathcal{L}_k}^*(f) = -\frac{1}{2}\sum_{m=1}^{T}\frac{\left(\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m} g_{i,k}\right)^2}{\sum_{i:\boldsymbol{x}_i\in\mathcal{R}_m}(h_{i,k}+\alpha)} + \gamma T \tag{7}$$

Here, we can use the equation (7) to measure the **impurity of the split**. Note that we can measure the impact of the split $\mathcal{I}_o = \mathcal{I}_L \cup \mathcal{I}_R$ via the loss reduction

$$\frac{1}{2}\left[\frac{\left(\sum_{i\in\mathcal{I}_L} g_{i,k}\right)^2}{\sum_{i\in\mathcal{I}_L}(h_{i,k}+\alpha)} + \frac{\left(\sum_{i\in\mathcal{I}_R} g_{i,k}\right)^2}{\sum_{i\in\mathcal{I}_R}(h_{i,k}+\alpha)} - \frac{\left(\sum_{i\in\mathcal{I}_o} g_{i,k}\right)^2}{\sum_{i\in\mathcal{I}_o}(h_{i,k}+\alpha)}\right] - \gamma T. \tag{8}$$

This equation (8) is used in practice to choose split candidate [Chen and Guestrin, 2016].

Gradient boost methods can be seen as searching for ***gradients in functional space*** $\mathcal{F}$, since the the optimal solution for a new model f_k should fit the **negative gradient values** of the ensemble $-\boldsymbol{g} = -\nabla_{\mathcal{L}}|_{f^{k-1}}$. For instance, when the loss function is squared loss, one fits the tree $\mathcal{T}$ according to the residual of the ensemble in previous iterations via least square. Both gradient boost methods and the recursive greedy algorithm to fit the tree are appoximation procedures. From (4), we can see that, similar to AdaBoost, the gradient boost method also reweights samples. Unlike Adaboost, which use the mis-classification error to upweight/downweight samples, the gradient boosting uses the **curvature of error surface** to upweight/downweight samples. It **downweight** samples located in *flat* surface and **upweight** the samples reside in ***steep curvature*** of the error surface.

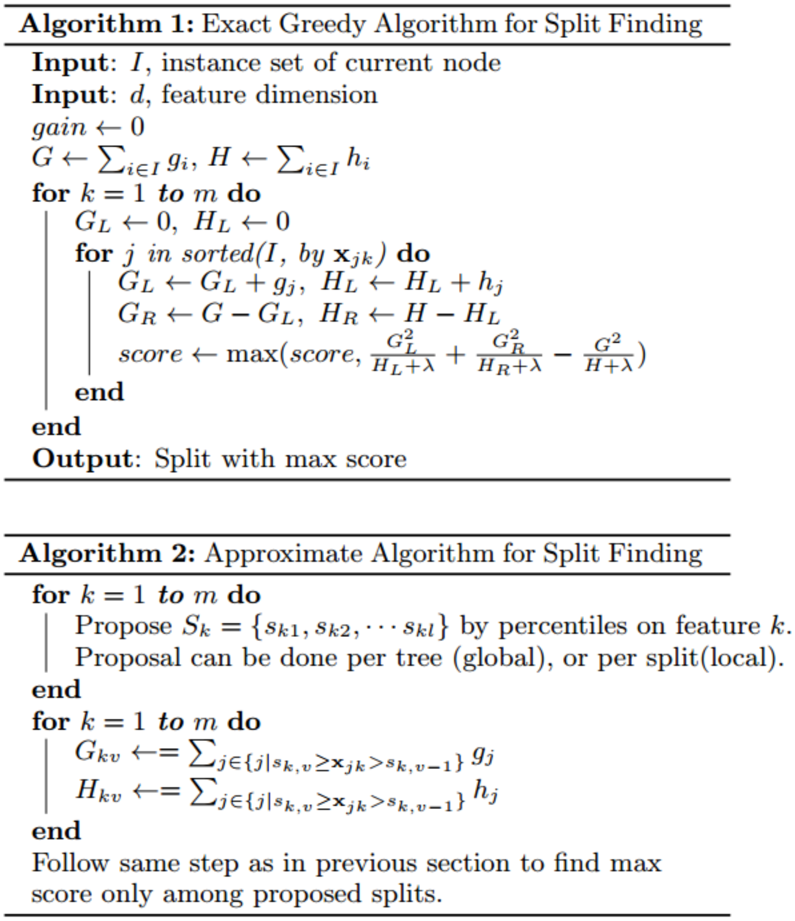
2.3 Split finding algorithm

One of the key problems in tree learning is to find the best split as indicated in (8). There are two ways to find it: the **exact greedy search** and the **approximate greedy search**. In exact greedy search, one enumerates all possible split for all continous features. In order to do so efficiently, the algorithm must first ***sort*** the data according to feature values and visit the data in sorted order to *accumulate* the gradient statistics for the structure score in (8). The drawbacks for exact greedy search is that it needs to fit all data into the memory. It is not efficient for distributed computing as well. In order to support these two settings, the appoximate greedy search is proposed. See figure 1 for descriptions of the algorithms.

The appoximate greedy search first propose a set of candidate splitting points based on the **percentiles** of feature distribution. The algorithm then maps the continous features into the **buckets** defined by these spiltting point, aggregates statistics and find the optimal solution based on the aggregated statistics. There are two *variants*: **global approximate** or **local approximate**. The global approximate proposes all candidates at the intital step of tree construction and uses the same proposal for split finding for all levels below. The local approximate re-proposes candidates after each split. The global method requires less proposal steps than the local method. However, usually more candidate points are needed for the global proposal because candidates are not refined after each split. The local proposal refines the candidates after splits, and can potentially be more appropriate for deeper trees [Chen and Guestrin, 2016].

The appoximate greedy search use the percentile of feature distribution to find candidte split points. A ***weighted*** quantile is proposed in [Chen and Guestrin, 2016] to improve this procedure. Let $(x_{i,d}, h_i)$ be the sample-hessian pair, where $x_{i,d}$ be the feature d for i-th sample and h_i is the second-order derivatives for the loss w.r.t. f. For all sample-hessian pair of feature d in training set as $\mathcal{D}_d := [(x_{i,d}, h_i)]_i$. We can define a rank function $r_d : \mathcal{R} \to [0, \infty)$:

$$r_d(t) := \frac{\sum_{i:x_i<t} h_i}{\sum_i h_i}.$$



```
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.
```

Figure 1: Split finding algorithm. The exact split finding algorithm search all featues and sort the data according to feature value. It then accumulate gradient statistics for the structure score in (8). [Chen and Guestrin, 2016]

It measures the propotion of instances whose feature d is less than t, **weighted by second-order derivatives** h_i at sample i. We then find the candidate split points as $[s_{0,d}, \dots, s_{l,d}]$ where $s_{0,d} = \min_i x_{i,d}$ and $s_{l,d} = \max_i x_{i,d}$ and $|s_{i,d} - s_{i+1,d}| <= \epsilon$ whiich split the interval between min and max of feature value into l buckets, each size no greater than ϵ. The reason weighted by h_i is that the approximate in (3) can be seen as quadratic form $h_i(f + g_i/h_i)^2$.

3 XGBoost

XGBoost [Chen and Guestrin, 2016] is an open source package for the gradient boosting tree algorithm. It has been popular among industry and competitions. XGBoost provides additional optimization on the gradient boosting algorithm to make is successful in real data.

Features of XGBoost:

- Support both single and **distributed systems**(Hadoop, Spark). As discussed above, XGBoost use global/local approximate split finding algorithm to handle distributed data. It also stores data in blocks and compressed columns.

- XGBoost is used in supervised learning(regression and classification problems). Due to generic setting on gradient boosting, it can be used with cross-entropy loss, squared loss etc.

- Supports **parallel processing**. It compressed data in each column and stored samples separately in in-memory *blocks* (a.k.a. ***column blocks*** since each column store the value of features). Data in blocks can be reused and distributed via multiple machines. In approximate split finding, using *sorted data structure* within each blocks, only linear scan is needed. Morever, collecting statstics in (8) can be paralleled too since each column is treated independently.

- **Column subsampling**. Similar to Random Forest, XGBoost can select a subset of features in each block/batch to reduce the variance and to prevent overfitting.

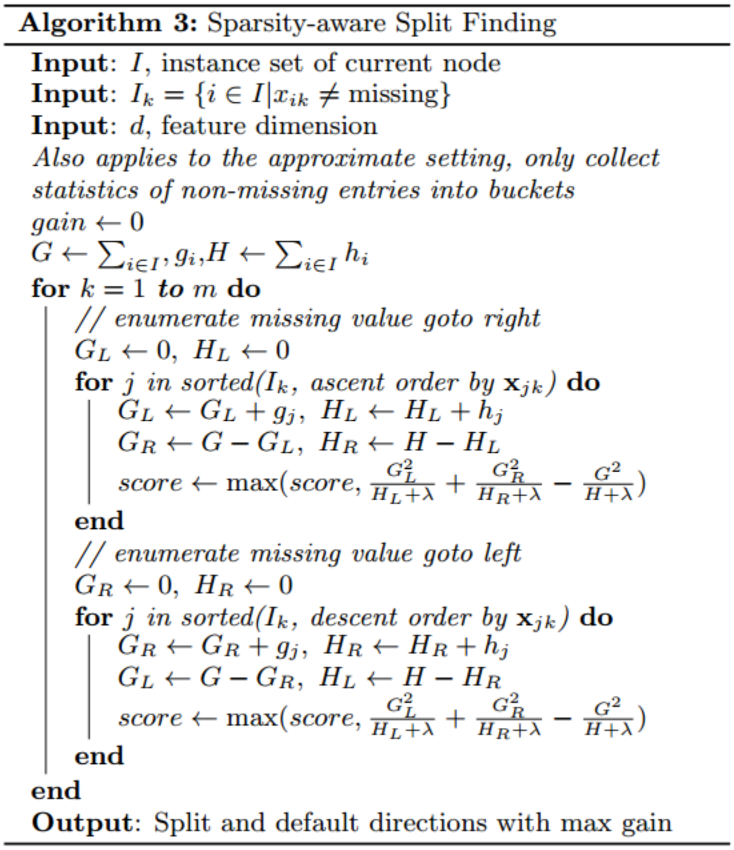
- Cache optimization. While the proposed block structure helps optimize the computation complexity of split finding, the new algorithm requires indirect fetches of gradient statistics by row index, since these values are accessed in order of feature. This is a non-continuous memory access.For the exact greedy algorithm, we can alleviate the problem by a **cache-aware prefetching algorithm**. For approximate algorithms, we solve the problem by choosing a correct block size.

- Efficient memory management for large datasets exceeding RAM. It uses techniques such as **block compression** and **block sharding**.

- Has a variety of regularizations which helps in reducing overfitting. For example, XGBoost uses **shrinkage** to scale down newly added weights by a factor η after each step of tree boosting. η is called the *learning rate.*

- Auto tree pruning Decision tree will not grow further after certain limits internally. Using the criteria defined in (7).

- Can handle **missing values**. It implements **sparsity-aware split finding**. See Figure 2. There are multiple possible causes for sparsity: 1) presence of missing values in the data; 2) frequent zero entries in the statistics; and, 3) artifacts of feature engineering such as one-hot



```
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
```

Figure 2: Sparsity-aware split finding algorithm. The approximate split finding algorithm only the non-missing entries and search for different choice of default directions at each branch. [Chen and Guestrin, 2016]

encoding. In order to handle sparsity, a default direction is computed. When a value is missing in the sparse matrix X, the instance is classified into the default direction. There are two choices of default direction at each branch, and the optimal default directions are learned from the data. Finally, only **non-missing entries** are visited to define buckets.

- Has inbuilt Cross-Validation.

- Takes care of outliers to some extent. Because it uses weighted quantile for each feature in appoximate split finding, the rare extreme values does not affect the split candidate proposals.

References

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.