XGBoost

Introduction to Boosted Trees:

Tree boosting is a highly effective and widely used machine learning method. Let us understand the concepts of Regression Tree, Ensemble and gradient boosting before we jump into the widely popular XGBoost.

Regression Tree is also known as classification and regression tree has the same decision rules as in decision tree but in addition it has one score in each leaf value.

Ensemble as we know use multiple learning algorithms to obtain better predictive performance that can be obtained from any of the constituent learning algorithms. Many of the data mining competitions are won by using some variants of tree ensemble methods.

Model: Assuming we have K trees

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

, where F is the space of all Regression trees

Objective:

$$Obj = \underbrace{\sum_{i=1}^{n} l(y_i, \hat{y}_i)}_{\text{Training loss}} + \underbrace{\sum_{k=1}^{K} \Omega(f_k)}_{\text{Complexity of the Trees}}$$

Gradient Boosting:

As we are using trees rather than numerical vectors we cannot use SGD (Stochastic Gradient Descent). Hence we use Additive Training (Boosting)

- Start from constant prediction, add a new function each time

$$\begin{aligned}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
&\dots \\
\hat{y}_i^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i) \longleftarrow \textbf{New function}
\end{aligned}$$

Our objective remains which f to add at each stage so that the *Obj* is minimized. Taking the Taylor expansion and removing the constants out new objective becomes

$$\sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

- where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), \quad h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$

A

We define tree by a vector of scores in leafs, and a leaf index mapping function that maps an instance to a leaf

$$f_t(x) = w_{q(x)}, \quad w \in \mathbf{R}^T, q : \mathbf{R}^d \to \{1, 2, \cdots, T\}$$

where w is the leaf weight of the tree and q is the structure of the tree.

Define complexity as (this is not the only possible definition)

$$\Omega(f_t) = \underbrace{\gamma T}_{\text{Number of leaves}} + \underbrace{\frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2}_{\text{L2 norm of leaf scores}}$$

- Define the instance set in leaf j as $I_j = \{i | q(x_i) = j\}$
- Regroup the objective by each leaf

$$\begin{aligned} Obj^{(t)} &\simeq \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \\ &= \sum_{i=1}^{n} \left[g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2 \right] + \gamma T + \lambda \frac{1}{2} \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2 \right] + \gamma T \end{aligned}$$

- This is sum of T independent quadratic functions
- Two facts about single variable quadratic function

$$argmin_x \; Gx + \frac{1}{2}Hx^2 = -\frac{G}{H}, \; H > 0 \qquad \min_x \; Gx + \frac{1}{2}Hx^2 = -\frac{1}{2}\frac{G^2}{H}$$

- Let us define $G_j = \sum_{i \in I_j} g_i \quad H_j = \sum_{i \in I_j} h_i$

$$\begin{aligned} Obj^{(t)} &= \sum_{j=1}^{T} \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2 \right] + \gamma T \\ &= \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma T \end{aligned}$$

- Assume the structure of tree (q(x)) is fixed, the optimal weight in each leaf, and the resulting objective value are

$$w_j^* = -\frac{G_j}{H_j + \lambda} \qquad Obj = -\frac{1}{2} \sum_{j=1}^{T} \underbrace{\frac{G_j^2}{H_j + \lambda}}_{\text{This measures how good a tree structure is!}} + \gamma T$$

Though in algorithm we need to enumerate all the possible tree structures q it is not feasible because there can be infinitely many tree structures so,

- In practice, we grow the tree greedily
  - Start from tree with depth 0
  - For each leaf node of the tree, try to add a split. The change of objective after adding the split is

$$Gain = \frac{1}{2}\left[\underbrace{\frac{G_L^2}{H_L + \lambda}}_{\text{the score of left child}} + \underbrace{\frac{G_R^2}{H_R + \lambda}}_{\text{the score of right child}} - \underbrace{\frac{(G_L + G_R)^2}{H_L + H_R + \lambda}}_{\text{the score of if we do not split}} \right] - \underbrace{\gamma}_{\text{The complexity cost by introducing additional leaf}}$$

  - Remaining question: how do we find the best split?

A

Ideally or the exact Greedy Algorithm for split finding is

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
```

Though the exact algorithm is very powerful it becomes impossible to do it efficiently when the data doesn't fit entirely into memory. Therefore an approximation algorithm is used.

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.
```

Also in real world it is quite common that input **x** to be sparse (missing values too). To make the algorithm aware of the sparsity in the data a **default direction** is added at each tree node. The algorithm is shown below:

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
```

A

Column block for Parallel Learning:

The most time consuming part is to get the data in sorted order. In order to reduce the cost of sorting instead of storing the entire dataset in a single block and running the split search we store the data in in-memory units. In each such unit data is stored in the compressed column format (CSC) . Collecting statistics from each column can be parallelized, giving us a parallel algorithm for split finding.

Apart from these XGBoost also has capabilities like **Cache-aware Access** (to prevent non-continuous memory access) and **Out-of-Core Computation** to fully utilize a machine’s resources.

A

For a simple interface in R we use **xgboost()** and for more a advanced interface we use **xgboost.train()**

xgboost

Usage: xgboost(data = NULL, label = NULL, missing = NULL, params = list(), nrounds, verbose = 1, print.every.n = 1L, early.stop.round = NULL, maximize = NULL, ...)

The list and description of some important arguments

data takes matrix, dgCMatrix, local data file. Label is the response variable, missing to represent missing values (used only when input is dense).

objective: “reg:linear” or “binary:logistic”

eta: stepsize for boosting

max.depth : maximum depth of the tree

nthreads: number of threads used in training (if not set default uses all the threads)

nrounds: number of iterations

xgboost.train

In case of xgboost.train() we have many other parameters for advanced interfacing like

Subsample: subsample ratio of training instance

Min_child_weight: minimum sum of instance weight(hessian) needed in a child.

For objectives: “multi:softmax” or “multi:softprob” etc

obj for customized objective etc.

xgb.plot.tree

To plot a boosted tree model