# XGBoost cheat sheet

XGBoost is an improved version of the classic GBT algorithm. This algorithm has become very popular due to its strong presence in the winning solutions of many Kaggle challenges. The algorithm also uses some tricks to be faster than classical GBT algorithm implemented in sklearn for example. (but not as fast as LightGBM). All this work is based on the author's paper: https://arxiv.org/pdf/1603.02754.pdf

Prerequisites : Having knowledge about the classic GBT algorithm.

## Theory :

As it still a gradient boosted tree algorithm, we still want to build an additive tree models $h_M$ , such as :

$$h_M(x) = h_{M-1}(x) + \delta_M(x) = \sum_{t=1}^{M} \delta_t(x)$$

First difference with the classic GBT algorithm, there is no "learning rate". Other difference, In the classical GBT algorithm we have the objective of minimizing $E(l(h_M(X), Y))$ , i.e minimizing the objective function :

$$L(h_M) = \sum_{i=1}^{n} l(y_i, h_M(x_i))$$

with $l$ a differentiable and strictly convex cost function and $\delta_t$ trees model. In XGBoost, we add a penalty on the trees model :

$$L(h_M) = \sum_{i=1}^{n} l(y_i, h_M(x_i)) + \sum_{t=1}^{M} \Omega(\delta_t) \qquad (1)$$

with

$$\Omega(\delta) = \gamma T + \tfrac{1}{2}\lambda \|\omega\|^2$$

$T$ being the number of leaves and $\omega$ the leaf weights ( $\omega \in \mathbb{R}^T$ ). The resolution of the minimization problem will also be different in its approach than in the classical GBT algorithm.

Let's suppose that $h_{M-1}$ is built and that we want to iterate our model. If we develop (1) we have :

$$L(h_M) = \sum_{i=1}^{n} l(y_i, h_{M-1}(x_i) + \delta_M(x)) + \sum_{t=1}^{M} \Omega(\delta_t)$$

we approximate it using the second order Taylor expansion :

$$L(h_M) \simeq \sum_{i=1}^{n} ( l(y_i, h_{M-1}(x_i)) + \delta_M(x_i) g_i + \tfrac{1}{2} \delta_M(x_i)^2 h_i ) + \sum_{t=1}^{M} \Omega(\delta_t) \qquad (2)$$

with

$$g_i = \nabla_{h_{m-1}(x_i)} l(y_i, h_{m-1}(x_i))$$
$$h_i = \nabla^2_{h_{m-1}(x_i)} l(y_i, h_{m-1}(x_i))$$

As $h_{M-1}$ is built, we can remove the constant term $\sum_{i=1}^{n} l(y_i, h_{M-1}(x_i))$ and $\sum_{t=1}^{M-1} \Omega(\delta_t)$ from (2) and we got this new expression to minimize :

$$L'(h_M) \simeq \sum_{i=1}^{n} ( \delta_M(x_i) g_i + \tfrac{1}{2} \delta_M(x_i)^2 h_i ) + \Omega(\delta_M)$$

which is equivalent to :

$$L'(h_M) \simeq \sum_{i=1}^{n}(\ \delta_M(x_i)\,g_i + \tfrac{1}{2}\ \delta_M\,(x_i)^2 h_i\ ) + \gamma T + \tfrac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^{\,2}$$

We define $I_j = \left\{\, i \mid \delta(x_i) = \omega_j \,\right\}$ as instance set of a leaf $j$ . From this we can write :

$$L'(h_M) \simeq \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)\,\omega_i + \tfrac{1}{2} \sum_{i\in I_j} h_i\,\omega_i^{\,2}] + \gamma T + \tfrac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^{\,2}$$

$$L'(h_M) \simeq \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)\,\omega_j + \tfrac{1}{2}(\sum_{i\in I_j} h_i)\,\omega_j^{\,2} + \tfrac{1}{2}\lambda\omega_j^{\,2}\ ] + \gamma T$$

$$L'(h_M) \simeq \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)\,\omega_j + \tfrac{1}{2}((\sum_{i\in I_j} h_i) + \lambda)\,\omega_j^{\,2}\ ] + \gamma T$$

If we consider the tree structure fixed, i.e $T$ a constant, we just need to resolve the minimum of a $f(x) = ax + bx^2$ structure function which is very easy since it is a question of solving the equation $f'(x) = 0$ which lead to $x_{min} = \frac{-a}{2b}$

The optimal weights are so :

$$\omega_j^{\,*} = \frac{-\sum_{i\in I_j} g_i}{(\sum_{i\in I_j} h_i) + \lambda} \tag{3}$$

And the corresponding objective function takes the value :

$$L'(h_M) = -\ \tfrac{1}{2}\sum_{j=1}^{T} \frac{(\sum_{i\in I_j} g_i)^2}{(\sum_{i\in I_j} h_i) + \lambda} + \gamma T \tag{4}$$

Eq (4) can be use as a scoring function to measure the quality of a tree structure. This score is like the impurity score for evaluating decision trees. We will use this formula to build the structure of the trees.

Normally it is impossible to enumerate all the possible tree structures q. A greedy algorithm that starts from a single leaf and iteratively adds branches to the tree is used instead.

Assume that $I_L$ and $I_R$ are the instance sets of left and right nodes after the split. Lettting $I = I_L \cup I_R$ , then the loss reduction after the split is given by

$$L_{split} = L_I - (L_R + L_L)$$

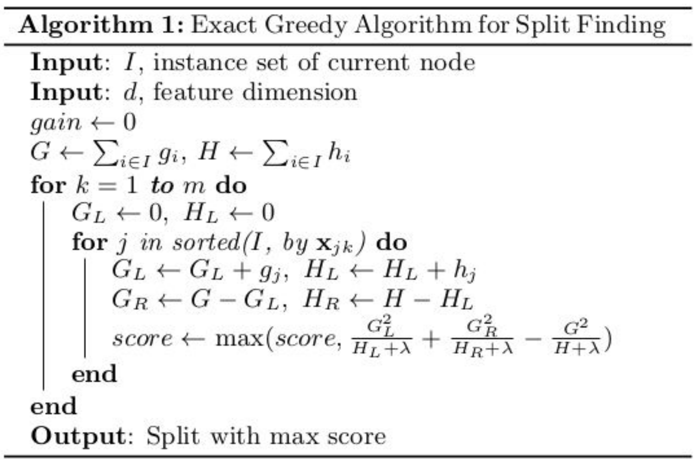
$$L_{split} = \tfrac{1}{2}(\frac{(\sum_{i\in I} g_i)^2}{(\sum_{i\in I} h_i) + \lambda} + \frac{(\sum_{i\in I_L} g_i)^2}{(\sum_{i\in I_L} h_i) + \lambda} - \frac{(\sum_{i\in I_R} g_i)^2}{(\sum_{i\in I_R} h_i) + \lambda}) - \gamma \tag{5}$$

Cleaner formula :

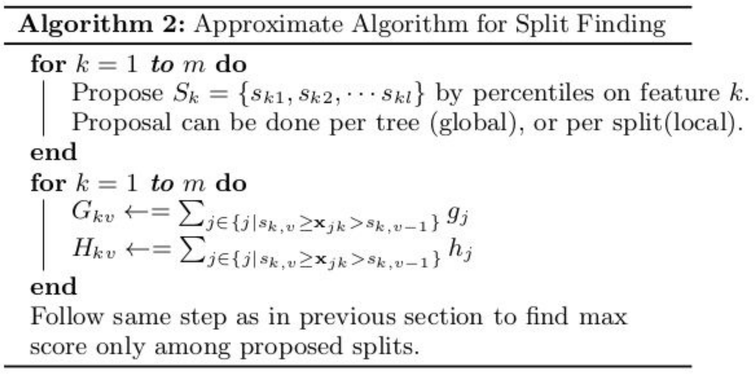
$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma$$

This formula is usually used in practice for evaluating the split candidates.

To proceed with the splits we can use two methods: exact greedy or approximate split finding, which leads to these two algorithms :



```
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.
```

**Important** : There are two variants of the approximate algorithm (or also called histogram-based algorithm), depending on when the proposal is given.

1) The global variant proposes all the candidate splits during the initial phase of tree construction, and uses the same proposals for split finding at all levels.

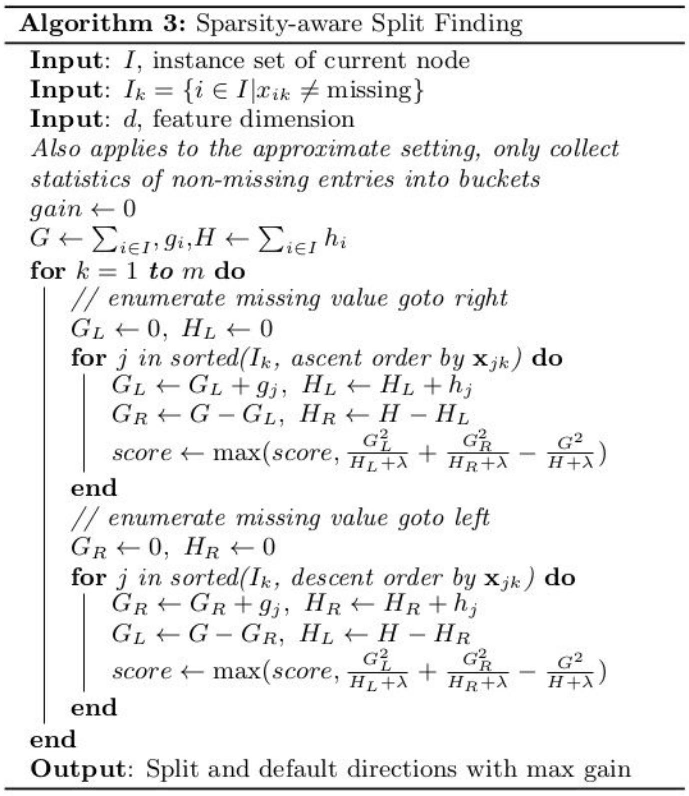
2) The local variant re-proposes after each split. The global method requires less proposal steps than the local method. Users can freely choose between the methods according to their needs

An important step in the approximate algorithm is to propose candidate split points → see the author's paper

## Tricks to improve speed :

### 1) **Sparsity-aware Split Finding :**

In many real-world problems, it is quite common for the input X to be sparse. 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 encoding. It is important to make the algorithm aware of the sparsity pattern in the data. In order to do so, we propose to add a default direction in each tree node. When a value is missing in the sparse matrix x, the instance is classified into the default direction, which lead to this algo :



```
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
```

2) **Column Block for parallel learning (or pre-sorted algorithm) :**

The most time consuming part of tree learning is to get the data into sorted order. In order to reduce the cost of sorting, we propose to store the data in in-memory units, which we called block. Data in each block is stored in the compressed column (CSC) format, with each column sorted by the corresponding feature value. This input data layout only needs to be computed once before training, and can be reused in later iterations.

→ In the exact greedy algorithm, we store the entire dataset in a single block and run the split search algorithm by linearly scanning over the pre-sorted entries. We do the split finding of all leaves collectively, so one scan over the block will collect the statistics of the split candidates in all leaf branches -> parallel learning**.**

→ The block structure also helps when using the approximate algorithm (histogram algorithm). Multiple blocks can be used in this case, with each block corresponding to a subset of rows in the dataset. Different blocks can be distributed across machines, or stored on disk in the out-of-core setting. Using the sorted structure, the quantile finding step becomes a linear scan over the sorted columns. This is especially valuable for local proposal algorithms, where candidates are generated frequently at each branch.

3) **Cache-aware Access :** see the author's paper
4) **Blocks for Out-of-core Computation** : see the author's paper