

## 3.5 XGBoost基本原理

CSDN学院  
2017年11月

# ► XGBoost



- XGBoost : eXtreme Gradient Boosting
- Gradient Boosting Machines(GBM) 的C++优化实现，快速有效
  - 深盟分布式机器学习开源平台 (Distributed Machine Learning Community, DMLC)的一个分支
  - DMLC也开包含流行的深度学习库mxnet

The name xgboost, though, actually refers to the engineering goal to [push the limit of computations resources for boosted tree algorithms](#). Which is the reason why many people use xgboost.

– Tianqi

*Chen, on Quora.com*

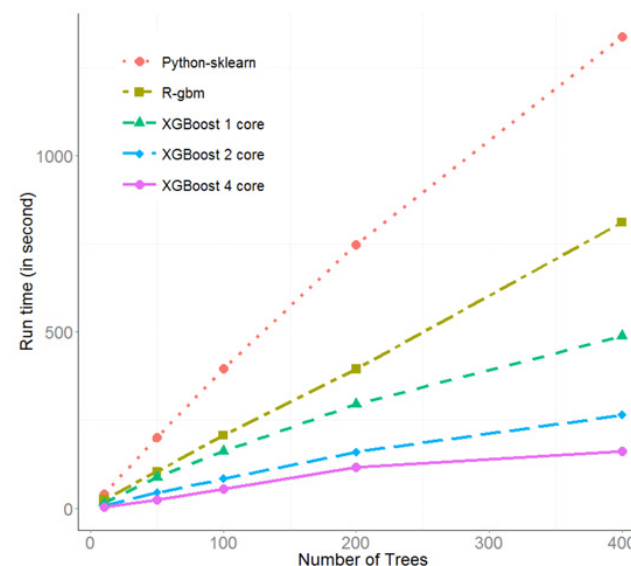


## ► XGBoost的优势

- XGBoost是近几年应用机器学习领域内一个强有力的武器
  - 执行速度：确实比其他Gradient Boosting实现快
  - 模型性能：在结构化数据集上，在分类 / 回归/排序预测建模上表现突出

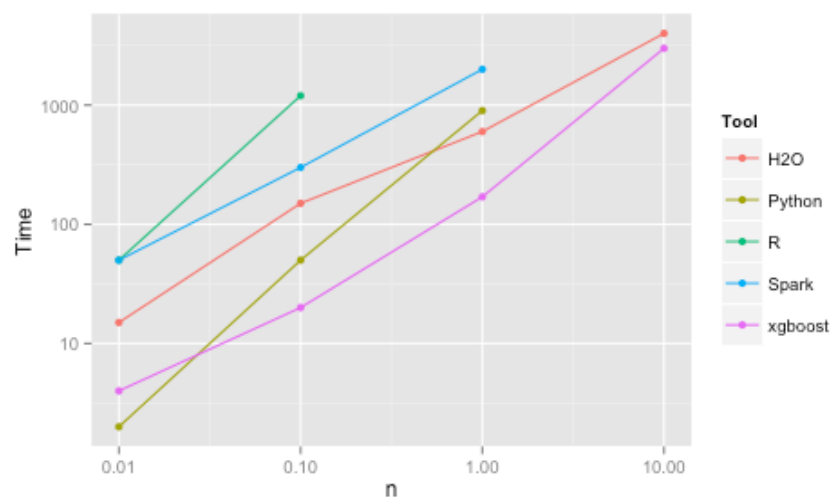
## ► 执行速度

- 在Kaggle的Higgs Boson竞赛中
  - 单线程的XGBoost比GBM的R语言包实现和Python的sklearn实现快将近一倍
  - 多线程有接近线性程度的提升

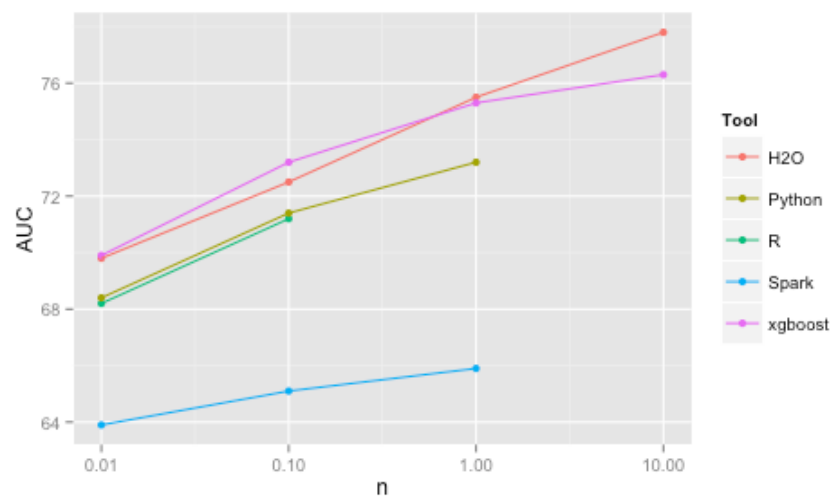


## ► 执行速度 (cont.)

- 2015年，Szilard Pafka在一些数据集上比较了XGBoost和其他Gradient Boosting / Bagged decision trees实现：又快又好



数据规模 ( M )



数据规模 ( M )

## ► XGBoost的性能

- 在Kaggle和KDD等一些数据科学竞赛平台上取了优异成绩

竞赛	名次	作者	备注
<a href="#">KDD Cup 2016 competition.</a>	1	Vlad Sandulescu, Mihai Chiru	<a href="http://arxiv.org/abs/1609.02728">http://arxiv.org/abs/1609.02728</a>
<a href="#">Dato Truly Native? competition.</a>	1	Marios Michailidis, Mathias Müller and HJ van Veen	<a href="http://blog.kaggle.com/2015/12/03/dato-winners-interview-1st-place-mad-professors/">http://blog.kaggle.com/2015/12/03/dato-winners-interview-1st-place-mad-professors/</a>
<a href="#">CERN LHCb experiment Flavour of Physics competition.</a>	1	Vlad Mironov, Alexander Guschin	<a href="http://blog.kaggle.com/2015/11/30/flavour-of-physics-technical-write-up-1st-place-go-polar-bears/">http://blog.kaggle.com/2015/11/30/flavour-of-physics-technical-write-up-1st-place-go-polar-bears/</a>
...	...	...	...

- XGBoost : eXtreme Gradient Boosting
  - 可自定义损失函数：损失函数采用二阶近似
  - 规范化的正则项：叶子节点数目、叶子结点的分数
  - 建树与剪枝：先建完全树后剪枝
    - 支持分裂点近似搜索
    - 稀疏特征处理
    - 缺失值处理
  - 特征重要性与特征选择
  - 并行计算
  - 内存缓存

## ► 损失函数的二阶近似

- Gradient Boosting算法虽然对常见损失函数适用，但除了L2损失函数，其他损失函数推导还是比较复杂
- XGBoost：对损失函数用二阶Taylor展开近似

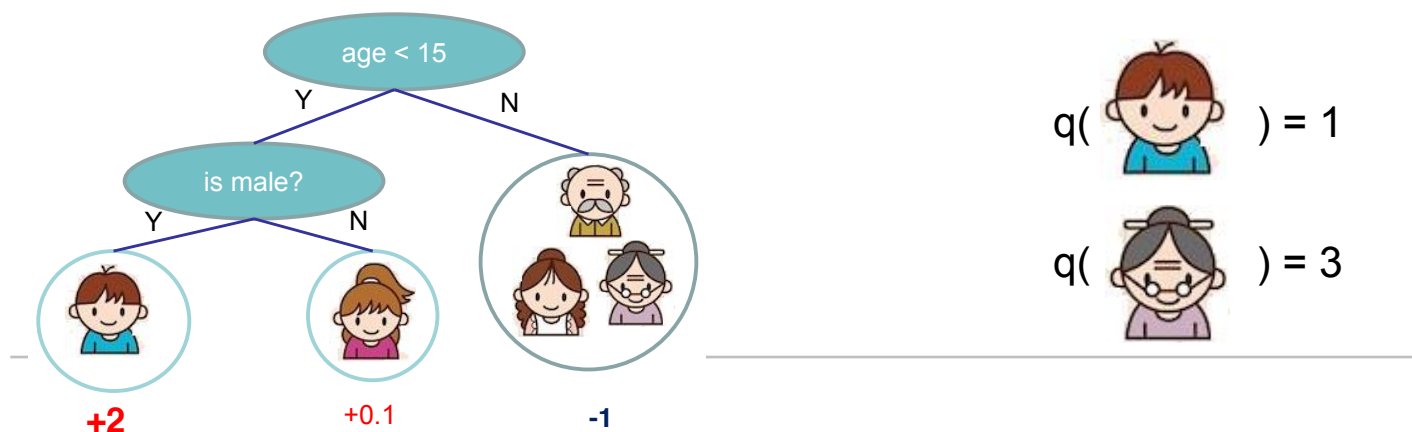


## ► 损失函数的二阶近似

$$f(x + \Delta x) \cong f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2$$

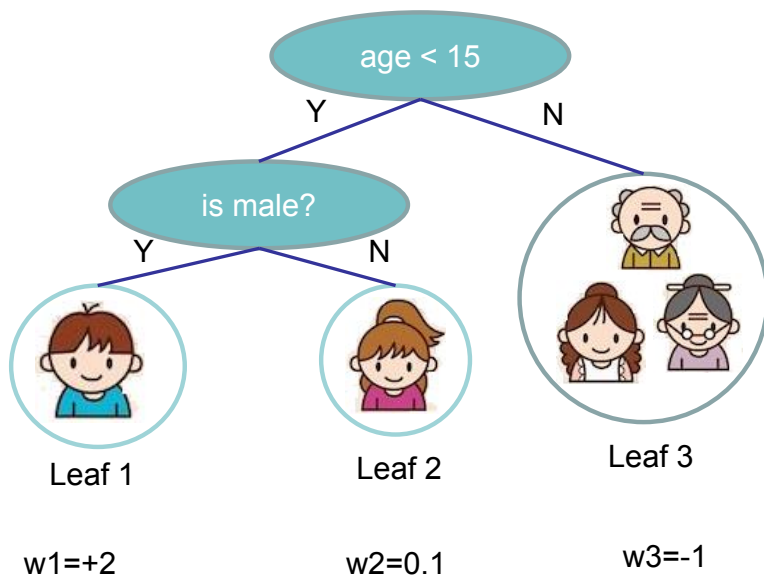
- XGBoost：对损失函数的二阶Taylor展开近似
- 在第 $m$ 步时，令  $g_{m,i} = \left[ \frac{\partial L(f(\mathbf{x}_i), y_i)}{\partial f(\mathbf{x}_i)} \right]_{\mathbf{f}=\mathbf{f}_{m-1}}$ ， $h_{m,i} = \left[ \frac{\partial^2 L(f(\mathbf{x}_i), y_i)}{\partial^2 f(\mathbf{x}_i)} \right]_{\mathbf{f}=\mathbf{f}_{m-1}}$
- $L(y_i, f_{m-1}(\mathbf{x}_i) + \phi(\mathbf{x}_i)) = \underbrace{L(f_{m-1}(\mathbf{x}_i), y_i)}_{\text{与未知量}\phi(\mathbf{x}_i)\text{无关}} + g_{m,i} \phi(\mathbf{x}_i) + \frac{1}{2} h_{m,i} \phi(\mathbf{x}_i)^2$
- 所以  $L(f_{m-1}(\mathbf{x}_i) + \phi(\mathbf{x}_i), y_i) = g_{m,i} \phi(\mathbf{x}_i) + \frac{1}{2} h_{m,i} \phi(\mathbf{x}_i)^2$
- 对L2损失， $L(f(\mathbf{x}; \boldsymbol{\theta}), y) = \frac{1}{2}(f(\mathbf{x}; \boldsymbol{\theta}) - y)^2$ ， $\nabla_f L(\boldsymbol{\theta}) = f(\mathbf{x}; \boldsymbol{\theta}) - y$ ， $\nabla_f^2 L(\boldsymbol{\theta}) = 1$
- 所以  $g_{m,i} = f_{m-1}(\mathbf{x}_i) - y_i$ ， $h_{m,i} = 1$

- Recall：树的定义：把树拆分成结构部分 $q$ 和叶子分数部分 $w$
- $\phi(\mathbf{x}) = w_{q(\mathbf{x})}$  ,  $\mathbf{w} \in R^T, q: R^D \rightarrow \{1, \dots, T\}$ 
  - 结构函数 $q$ ：把输入映射到叶子的索引号
  - $w$ ：给出每个索引号对应的叶子的分数
  - $T$ 为树中叶子结点的数目， $D$ 为特征维数



# ► 树的复杂度

- 树的复杂度定义为（不是唯一的定义方式）
- $\Omega(\phi(\mathbf{x})) = \gamma T + \frac{1}{2} \lambda \sum_{t=1}^T w_t^2$ 
  - 叶子节点的数目、叶子节点分数的L2正则



$$\Omega = \gamma 3 + \frac{1}{2} \lambda (4 + 0.01 + 1)$$

↑  
 $2^2 + 0.1^2 + (-1)^2$

## ► 目标函数






- 令每个叶子 $t$ 上的样本集合为 $I_t = \{i | q(\mathbf{x}_i) = t\}$
  - $J(\boldsymbol{\theta}) = \sum_{i=1}^N L(f(\mathbf{x}_i; \boldsymbol{\theta}), y_i) + \Omega(\boldsymbol{\theta})$
  - $\cong \sum_{i=1}^N \left( g_{m,i} \phi(\mathbf{x}_i) + \frac{1}{2} h_{m,i} \phi(\mathbf{x}_i)^2 \right) + \gamma T + \frac{1}{2} \lambda \sum_{t=1}^T w_t^2$
  - $= \sum_{i=1}^N \left( g_{m,i} w_{q(\mathbf{x}_i)} + \frac{1}{2} h_{m,i} w_{q(\mathbf{x}_i)}^2 \right) + \gamma T + \frac{1}{2} \lambda \sum_{t=1}^T w_t^2$
  - $= \sum_{t=1}^T \left[ \sum_{i \in I_t} g_{m,i} w_t + \frac{1}{2} \sum_{i \in I_t} h_{m,i} w_t^2 \right] + \frac{1}{2} \lambda \sum_{t=1}^T w_t^2 + \gamma T$
  - $= \sum_{t=1}^T \left[ \underbrace{\sum_{i \in I_t} g_{m,i}}_{G_t} w_t + \frac{1}{2} \left( \underbrace{\sum_{i \in I_t} h_{m,i}}_{H_t} + \lambda \right) w_t^2 \right] + \gamma T$
  - $= \sum_{t=1}^T \left[ G_t w_t + \frac{1}{2} (H_t + \lambda) w_t^2 \right] + \gamma T$
- T个独立的二次函数之和

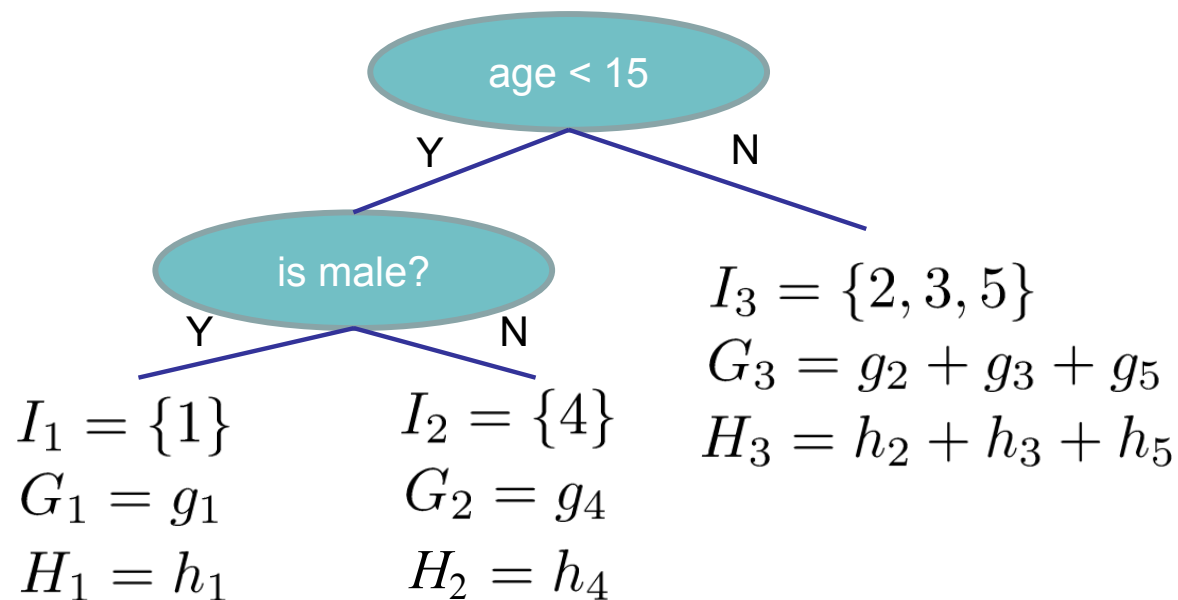
## ► 目标函数

- 假设我们已经知道树的结构 $q$ ,
- $J(\boldsymbol{\theta}) = \sum_{t=1}^T \left[ G_t w_t + \frac{1}{2} (H_t + \lambda) w_t^2 \right] + \gamma T$
- 则由  $\frac{\partial J(\boldsymbol{\theta})}{\partial w_t} = G_t + (H_t + \lambda) w_t = 0$
- 得到最佳的 $\mathbf{w}$  :  $w_t = -\frac{G_t}{H_t + \lambda}$
- 以及最佳的 $\mathbf{w}$ 对应的目标函数, 可视为树的分数 :
- $J(\boldsymbol{\theta}) = -\frac{1}{2} \sum_{t=1}^T \left[ \frac{G_t^2}{H_t + \lambda} \right] + \gamma T$       分数越小的树越好 !

# ► 例：树的分数

Instance index      gradient statistics

1		$g_1, h_1$
2		$g_2, h_2$
3		$g_3, h_3$
4		$g_4, h_4$
5		$g_5, h_5$



$$Obj = - \sum_j \frac{G_j^2}{H_j + \lambda} + 3\gamma$$

The smaller the score is, the better the structure is

- 枚举可能的树结构
- 计算结构分数
- $J(\theta) = -\frac{1}{2} \sum_{t=1}^T \left[ \frac{G_t^2}{H_t + \lambda} \right] + \gamma T$
- 选择分数最小的树结构，并且运用最优的权重/分数
- 但是，树结构有很多可能 → 贪心算法

## ► 建树 ( cont. )

- 实践中，我们贪婪的增加树的叶子结点数目：
- ( 1 ) 从深度为0的树开始
- ( 2 ) 对于树的每个叶子节点，尝试增加一个分裂点：
  - 令  $I_L$  和  $I_R$  分别表示加入分裂点后左右叶子结点的样本集合， $I = I_L \cup I_R$ ,
  - $G_L = \sum_{i \in I_L} g_{m,i}$ ， $G_R = \sum_{i \in I_R} g_{m,i}$ ， $H_L = \sum_{i \in I_L} h_{m,i}$ ， $H_R = \sum_{i \in I_R} h_{m,i}$ ，
  - 则增加分裂点后目标函数的变化为

$$Gain = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G_L^2 + G_R^2}{H_L + H_R + \lambda} - \gamma$$

- 但是：怎么找到最优的分裂点？



## ► 建树——精确搜索算法

- 对每一个结点，穷举所有特征、所有可能的分裂点
  - 对每个特征，通过特征值将实例进行排序
  - 运用线性扫描来寻找该特征的最优分裂点
  - 对所有特征，采用最佳分裂点
- 深度为 $k$ 的树的时间复杂度：
  - 对于一层排序，需要时间 $M\log(N)$ ， $N$ 为样本数目
  - 由于有 $D$ 个特征， $k$ 层，所以为 $kDM\log(N)$

---

**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**  $D$  **do** (对每维特征)

$G_L \leftarrow 0, H_L \leftarrow 0$

**for**  $j$  **in**  $sorted(I, \text{by } \mathbf{x}_{jk})$  **do** (以第 $k$ 维特征为分裂特征, 第 $j$ 个样本 $\mathbf{x}_{jk}$ 的值为阈值)

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$score \leftarrow \max(score, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

**end**

**end**

**Output:** Split with max score

---

首先, 对所有特征都按照特征的数值进行预排序。

其次, 在遍历分割点的时候用 $O(\#data)$ 的代价找到一个特征上的最好分割点。

最后, 找到一个特征的分割点后, 将数据分裂成左右子节点。

## ► 基于预排序的方法

- 这样的预排序算法的优点是能精确地找到分割点。
- 缺点也很明显：
  - 空间消耗大。算法需要保存数据的特征值，还保存了特征排序的结果（例如排序后的索引，为了后续快速的计算分割点），这里需要消耗训练数据**两倍**的内存。
  - 时间上也有较大的开销，在遍历每一个分割点的时候，都需要进行分裂增益的计算，消耗的代价大。
  - 对cache优化不友好。在预排序后，特征对梯度的访问是一种随机访问，并且不同的特征访问的顺序不一样，无法对cache进行优化。同时，在每一层长树的时候，需要随机访问一个行索引到叶子索引的数组，并且不同特征访问的顺序也不一样，也会造成较大的cache miss。

# ► 建树——近似搜索算法

- 当数据太多不能装载到内存时，不能进行精确搜索分裂，只能近似
  - 根据特征分布的百分位数，提出特征的一些候选分裂点
  - 将连续特征值映射到桶里（候选点对应的分裂），然后根据桶里样本的统计量，从这些候选中选择最佳分裂点
- 根据候选提出的时间，分为
  - 全局近似：在构造树的初始阶段提出所有的候选分裂点，然后对各个层次采用相同的候选
    - 提出候选的次数少，但每次的候选数目多（因为候选不更新）
  - 局部近似：在每次分裂都重新提出候选
    - 对层次较深的树更适合

---

**Algorithm 2:** Approximate Algorithm for Split Finding

---

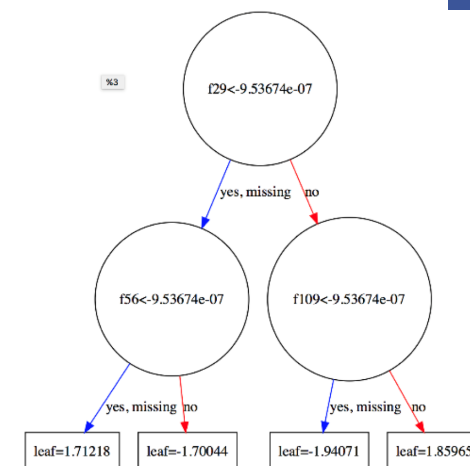
```
for  $k = 1$  to  $D$  do
    | Propose  $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$  by percentiles on feature  $k$ .
    | Proposal can be done per tree (global), or per split(local).
end
for  $k = 1$  to  $D$  do
    |  $G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$ 
    |  $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$ 
end
```

Follow same step as in previous section to find max score only among proposed splits.

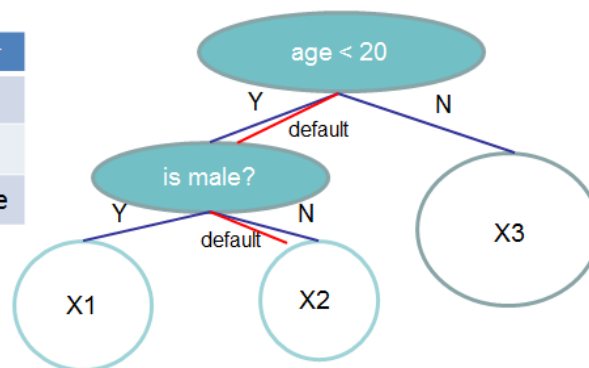
---

# ► 建树——稀疏特征

- 在实际任务中，极有可能遇到稀疏特征
  - 缺失数据
  - 人工设计的特征：如one-hot编码
- XGBoost：在树的每个结点设置一个缺省方向



Data		
Example	Age	Gender
X1	?	male
X2	15	?
X3	25	female

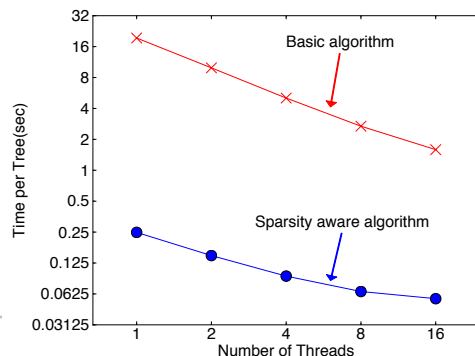


# 建树——稀疏特征

- 统一的稀疏特征处理方案：  
将稀疏特征视为缺失值
- 最佳缺省方向确定：
  - 只访问非缺失数据
  - 计算复杂度与非缺失数据数目

线性相关

在数据高度稀疏的  
Allstate-10K 数据集  
上稀疏算法比基本  
算法快近50倍



## 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**  $D$  **do**

*// enumerate missing value goto right*

$G_L \leftarrow 0, H_L \leftarrow 0$  (假设缺省方向为右边)

**for**  $j$  in sorted( $I_k$ , ascent order by  $x_{jk}$ ) **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*

$G_R \leftarrow 0, H_R \leftarrow 0$  (假设缺省方向为左边)

**for**  $j$  in sorted( $I_k$ , descent order by  $x_{jk}$ ) **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

CSDN

不止于代码



## ► 剪枝和正则

- Recall 分裂的增益：

$$Gain = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G_L^2 + G_R^2}{H_L + H_R + \lambda} - \gamma$$

- 增益可能为负：引入新叶子有复杂度惩罚
- 提前终止
  - 如果出现负值，提前停止（scikit-learn中采用的策略）
  - 但被提前终止掉的分裂可能其后续的分裂会带来好处
- 过后剪枝
  - 将树分裂到最大深度，然后再基于上述增益计算剪枝
  - 有必要：在实现时还有学习率 / 收缩，给后续轮留机会，进一步防止过拟合： $f_m(\mathbf{x}_i) = f_{m-1}(\mathbf{x}_i) + \eta \phi_m(\mathbf{x}_i)$



- `xgboost.XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True, objective='binary:logistic', nthread=-1, gamma=0, min_child_weight=1, max_delta_step=0, subsample=1, colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1, scale_pos_weight=1, base_score=0.5, random_state=0, seed=None, missing=None, **kwargs)`
- `sklearn.ensemble.GradientBoostingClassifier(loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_split=1e-07, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, presort='auto')`

## ► XGBoost小结

- What is XGBoost: GBM的优化实现
- Why XGBoost : 又快又好
- How to use XGBoost
  - 下一小节

# THANK YOU



AI100