【机器学习】决策树（下）——XGBoost、LightGBM（非常详细）

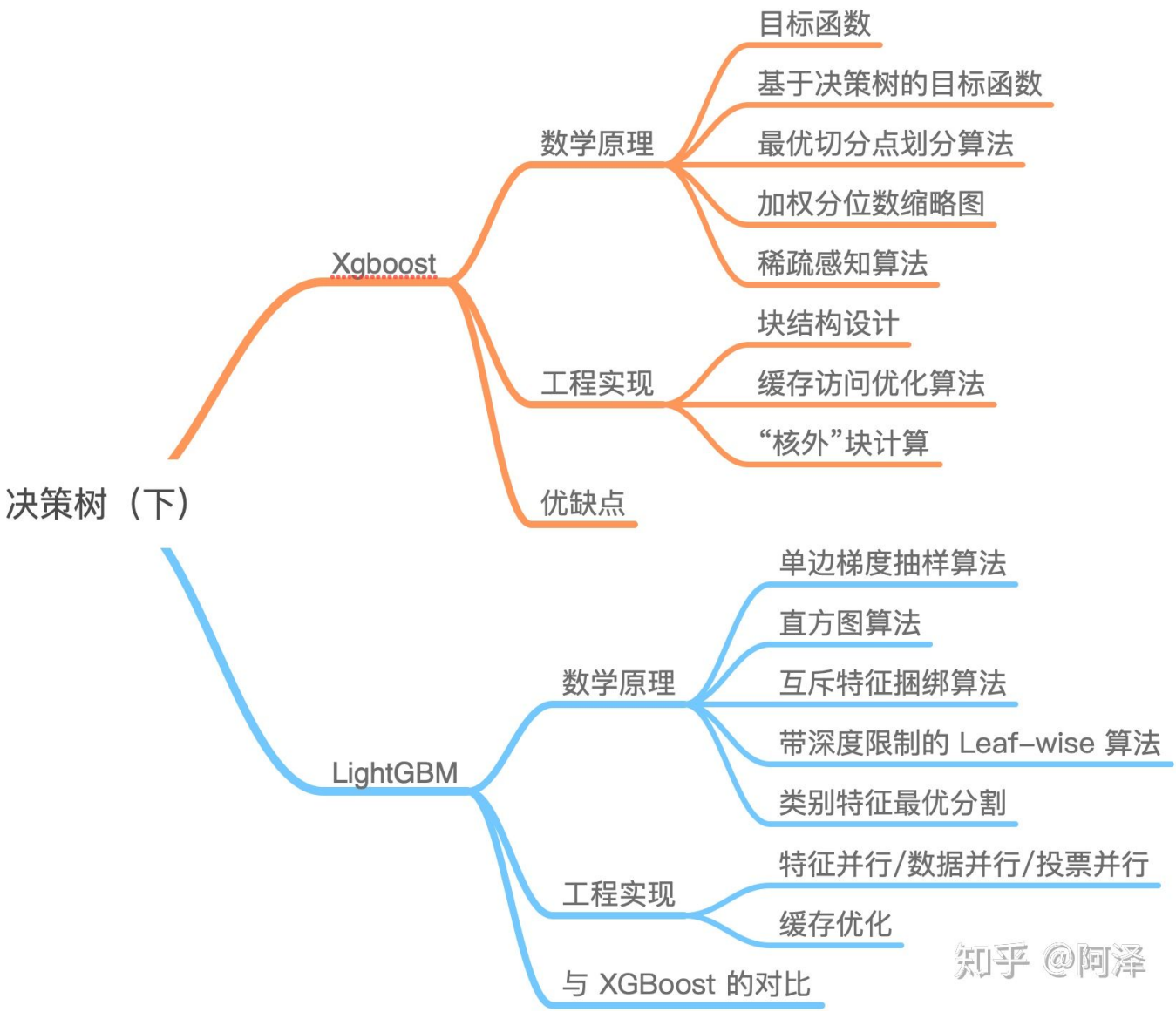
阿泽
复旦大学 计算机技术硕士

已关注

创作声明：内容包含虚构创作

本文是决策树的第三篇，主要介绍基于 Boosting 框架的主流集成算法，包括 XGBoost 和 LightGBM。

不知道为什么知乎文章封面的照片会显示不全，在这里补上完整的思维导图：



1. XGBoost

XGBoost 是大规模并行 boosting tree 的工具，它是目前最快最好的开源 boosting tree 工具包，比常见的工具包快 10 倍以上。Xgboost 和 GBDT 两者都是 boosting 方法，除了工程实

现、解决问题上的一些差异外，最大的不同就是目标函数的定义。故本文将从数学原理和工程实现上进行介绍，并在最后介绍下 Xgboost 的优点。

1.1 数学原理

1.1.1 目标函数

我们知道 XGBoost 是由 k 个基模型组成的一个加法运算式：

$$\hat{y}_i = \sum_{t=1}^{k} f_t(x_i)$$

其中 f_k 为第 k 个基模型， $\hat{y}_i$ 为第 i 个样本的预测值。

损失函数可由预测值 $\hat{y}_i$ 与真实值 y_i 进行表示：

$$L = \sum_{i=1}^{n} l(y_i, \hat{y}_i)$$

其中 n 为样本数量。

我们知道模型的预测精度由模型的偏差和方差共同决定，损失函数代表了模型的偏差，想要方差小则需要简单的模型，所以目标函数由模型的损失函数 L 与抑制模型复杂度的正则项 Ω 组成，所以我们有：

$$Obj = \sum_{i=1}^{n} l(\hat{y}_i, y_i) + \sum_{t=1}^{k} \Omega(f_t)$$

Ω 为模型的正则项，由于 XGBoost 支持决策树也支持线性模型，所以这里再不展开描述。

我们知道 boosting 模型是前向加法，以第 t 步的模型为例，模型对第 i 个样本 x_i 的预测为：

$$\hat{y}_i^t = \hat{y}_i^{t-1} + f_t(x_i)$$

其中 $\hat{y}_i^{t-1}$ 由第 $t-1$ 步的模型给出的预测值，是已知常数， $f_t(x_i)$ 是我们这次需要加入的新模型的预测值，此时，目标函数就可以写成：

$$
\begin{aligned}
Obj^{(t)} &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^t) + \sum_{i=1}^{t} \Omega(f_i) \\
&= \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{t-1} + f_t(x_i)\right) + \sum_{i=1}^{t} \Omega(f_i)
\end{aligned}
$$

求此时最优化目标函数，就相当于求解 $f_t(x_i)$ 。

> 泰勒公式是将一个在 $x = x_0$ 处具有 n 阶导数的函数 $f(x)$ 利用关于 $x - x_0$ 的 n 次多项式来逼近函数的方法，若函数 $f(x)$ 在包含 x_0 的某个闭区间 $[a, b]$ 上具有 n 阶导数，且在开区间 (a, b) 上具有 $n+1$ 阶导数，则对闭区间 $[a, b]$ 上任意一点 x 有 $f(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i + R_n(x)$ ，其中的多项式称为函数在 x_0 处的泰勒展开式， $R_n(x)$ 是泰勒公式的余项且是 $(x - x_0)^n$ 的高阶无穷小。

根据泰勒公式我们把函数 $f(x + \Delta x)$ 在点 x 处进行泰勒的二阶展开，可得到如下等式：

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

我们把 $\hat{y}_i^{t-1}$ 视为 x ， $f_t(x_i)$ 视为 Δx ，故可以将目标函数写为：

$$
Obj^{(t)} = \sum_{i=1}^{n} \left[l(y_i, \hat{y}_i^{t-1}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \sum_{i=1}^{t} \Omega(f_i)
$$

其中 g_i 为损失函数的一阶导， h_i 为损失函数的二阶导，**注意这里的导是对 $\hat{y}_i^{t-1}$ 求导。**

我们以平方损失函数为例：

$$
\sum_{i=1}^{n} \left(y_i - (\hat{y}_i^{t-1} + f_t(x_i))\right)^2
$$

则：

$$
\begin{aligned}
g_i &= \frac{\partial (\hat{y}^{t-1} - y_i)^2}{\partial \hat{y}^{t-1}} = 2(\hat{y}^{t-1} - y_i) \\
h_i &= \frac{\partial^2 (\hat{y}^{t-1} - y_i)^2}{\hat{y}^{t-1}} = 2
\end{aligned}
$$

由于在第 t 步时 $\hat{y}_i^{t-1}$ 其实是一个已知的值，所以 $l(y_i, \hat{y}_i^{t-1})$ 是一个常数，其对函数的优化不会产生影响，因此目标函数可以写成：

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

所以我们只需要求出每一步损失函数的一阶导和二阶导的值（由于前一步的 $\hat{y}^{t-1}$ 是已知的，所以这两个值就是常数），然后最优化目标函数，就可以得到每一步的 $f(x)$ ，最后根据加法模型得到一个整体模型。

1.1.2 基于决策树的目标函数

我们知道 Xgboost 的基模型**不仅支持决策树，还支持线性模型**，这里我们主要介绍基于决策树的目标函数。

我们可以将决策树定义为 $f_t(x) = w_{q(x)}$ ， x 为某一样本，这里的 $q(x)$ 代表了该样本在哪个叶子结点上，而 w_q 则代表了叶子结点取值 w ，所以 $w_{q(x)}$ 就代表了每个样本的取值 w （即预测值）。

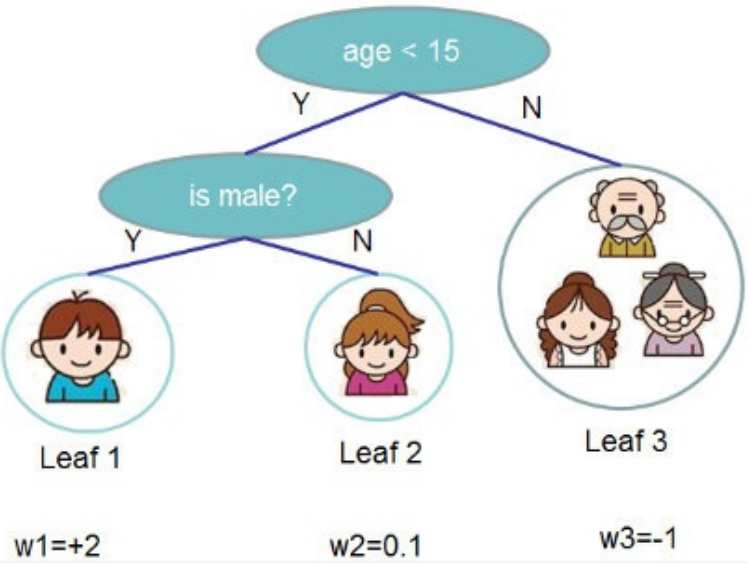
决策树的复杂度可由叶子数 T 组成，叶子节点越少模型越简单，此外叶子节点也不应该含有过高的权重 w （类比 LR 的每个变量的权重），所以目标函数的正则项可以定义为：

$$\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$$

即决策树模型的复杂度由生成的所有决策树的叶子节点数量，和所有节点权重所组成的向量的 L_2 范式共同决定。

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

叶子的个数　　w的L2模平方



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

这张图给出了基于决策树的 XGBoost 的正则项的求解方式。

我们设 $I_j = \{i|q(x_i) = j\}$ 为第 j 个叶子节点的样本集合，故我们的目标函数可以写成：

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

第二步到第三步可能看的不是特别明白，这边做些解释：第二步是遍历所有的样本后求每个样本的损失函数，但样本最终会落在叶子节点上，所以我们也可以遍历叶子节点，然后获取叶子节点上的样本集合，最后在求损失函数。即我们之前样本的集合，现在都改写成叶子结点的集合，由于一个叶子结点有多个样本存在，因此才有了 $\sum_{i\in I_j} g_i$ 和 $\sum_{i\in I_j} h_i$ 这两项， w_j 为第 j 个叶子节点取值。

为简化表达式，我们定义 $G_j = \sum_{i\in I_j} g_i$ ， $H_j = \sum_{i\in I_j} h_i$ ，则目标函数为：

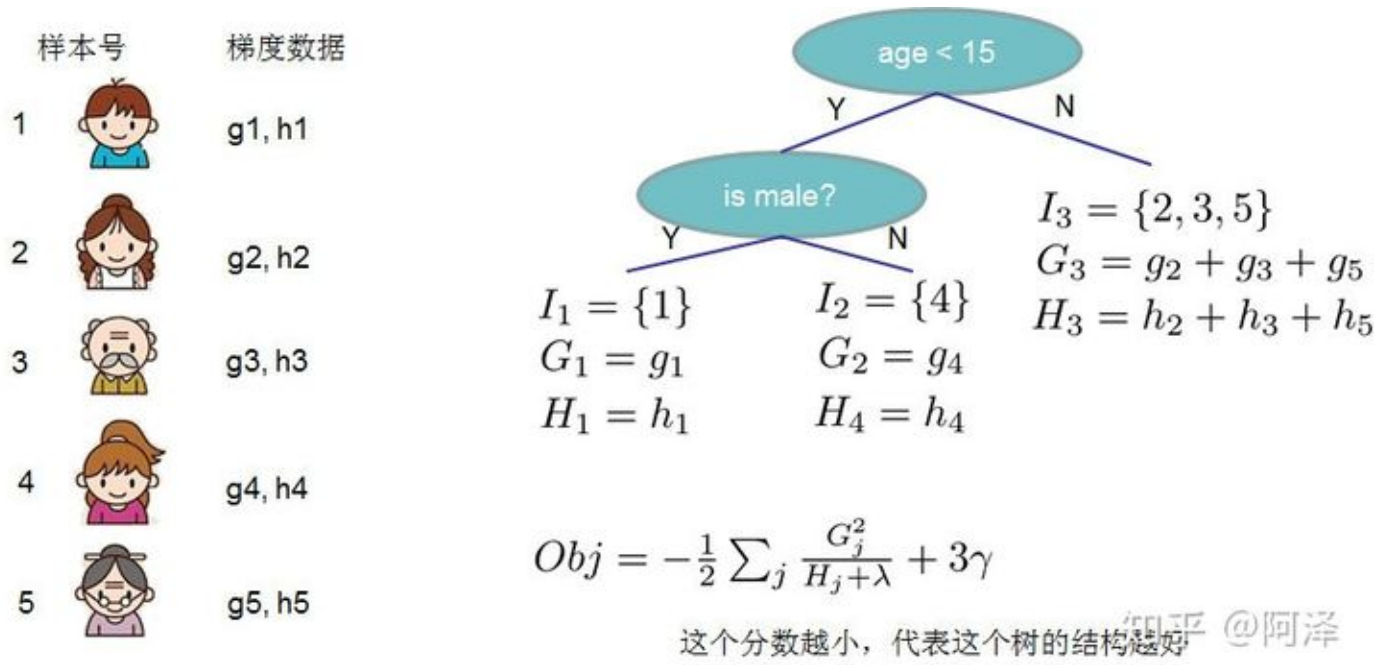
$$Obj^{(t)} = \sum_{j=1}^{T}\left[G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2\right] + \gamma T$$

这里我们要注意 G_j 和 H_j 是前 $t-1$ 步得到的结果，其值已知可视为常数，只有最后一棵树的叶子节点 w_j 不确定，那么将目标函数对 w_j 求一阶导，并令其等于 0 ，则可以求得叶子结点 j 对应的权值：

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

所以目标函数可以化简为：

$$Obj = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T$$



上图给出目标函数计算的例子，求每个节点每个样本的一阶导数 g_i 和二阶导数 h_i ，然后针对每个节点对所含样本求和得到的 G_j 和 H_j ，最后遍历决策树的节点即可得到目标函数。

1.1.3 最优切分点划分算法

在决策树的生长过程中，一个非常关键的问题是如何找到叶子的节点的最优切分点，Xgboost 支持两种分裂节点的方法——贪心算法和近似算法。

1）贪心算法

1. 从深度为 0 的树开始，对每个叶节点枚举所有的可用特征；
2. 针对每个特征，把属于该节点的训练样本根据该特征值进行升序排列，通过线性扫描的方式来决定该特征的最佳分裂点，并记录该特征的分裂收益；
3. 选择收益最大的特征作为分裂特征，用该特征的最佳分裂点作为分裂位置，在该节点上分裂出左右两个新的叶节点，并为每个新节点关联对应的样本集

4. 回到第 1 步，递归执行到满足特定条件为止

那么如何计算每个特征的分裂收益呢?

假设我们在某一节点完成特征分裂，则分列前的目标函数可以写为:

$$Obj_1 = -\frac{1}{2}[\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}]+\gamma$$

分裂后的目标函数为:

$$Obj_2 = -\frac{1}{2}[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}]+2\gamma$$

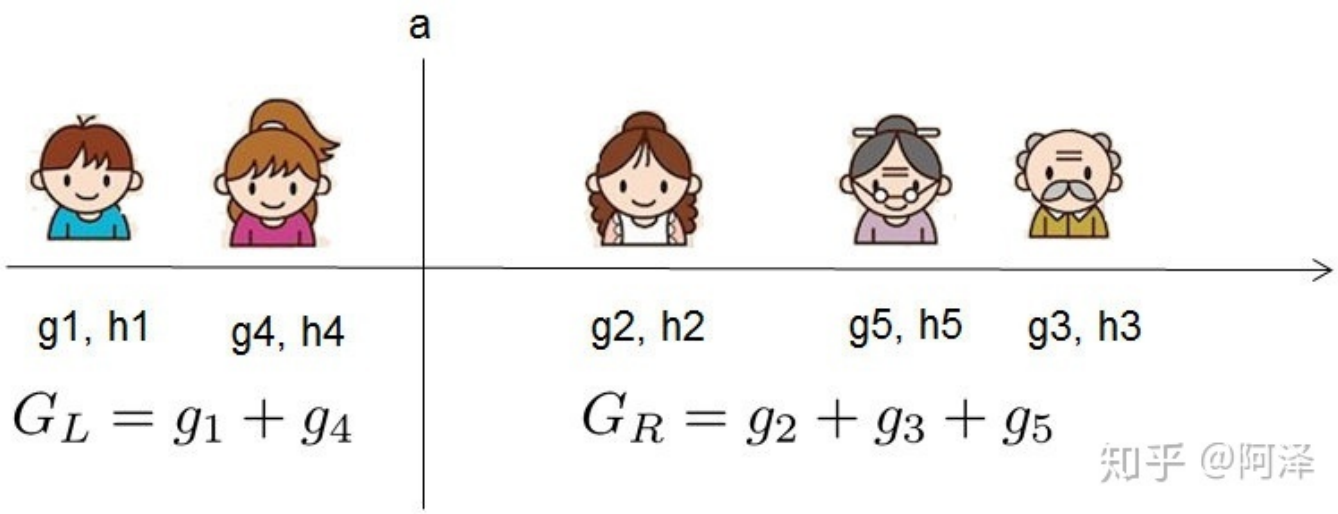
则对于目标函数来说，分裂后的收益为:

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right]-\gamma$$

注意该特征收益也可作为特征重要性输出的重要依据。

对于每次分裂，我们都需要枚举所有特征可能的分割方案，如何高效地枚举所有的分割呢?

我假设我们要枚举所有 $x < a$ 这样的条件，对于某个特定的分割点 a 我们要计算 a 左边和右边的导数和。



我们可以发现对于所有的分裂点 a ，我们只要做一遍从左到右的扫描就可以枚举出所有分割的梯度和 G_L 和 G_R 。然后用上面的公式计算每个分割方案的分数就可以了。

观察分裂后的收益，我们会发现节点划分不一定会使得结果变好，因为我们有一个引入新叶子的惩罚项，也就是说引入的分割带来的增益如果小于一个阀值的时候，我们可以剪掉这个分割。

2）近似算法

贪婪算法可以的到最优解，但当数据量太大时则无法读入内存进行计算，近似算法主要针对贪婪算法这一缺点给出了近似最优解。

对于每个特征，只考察分位点可以减少计算复杂度。

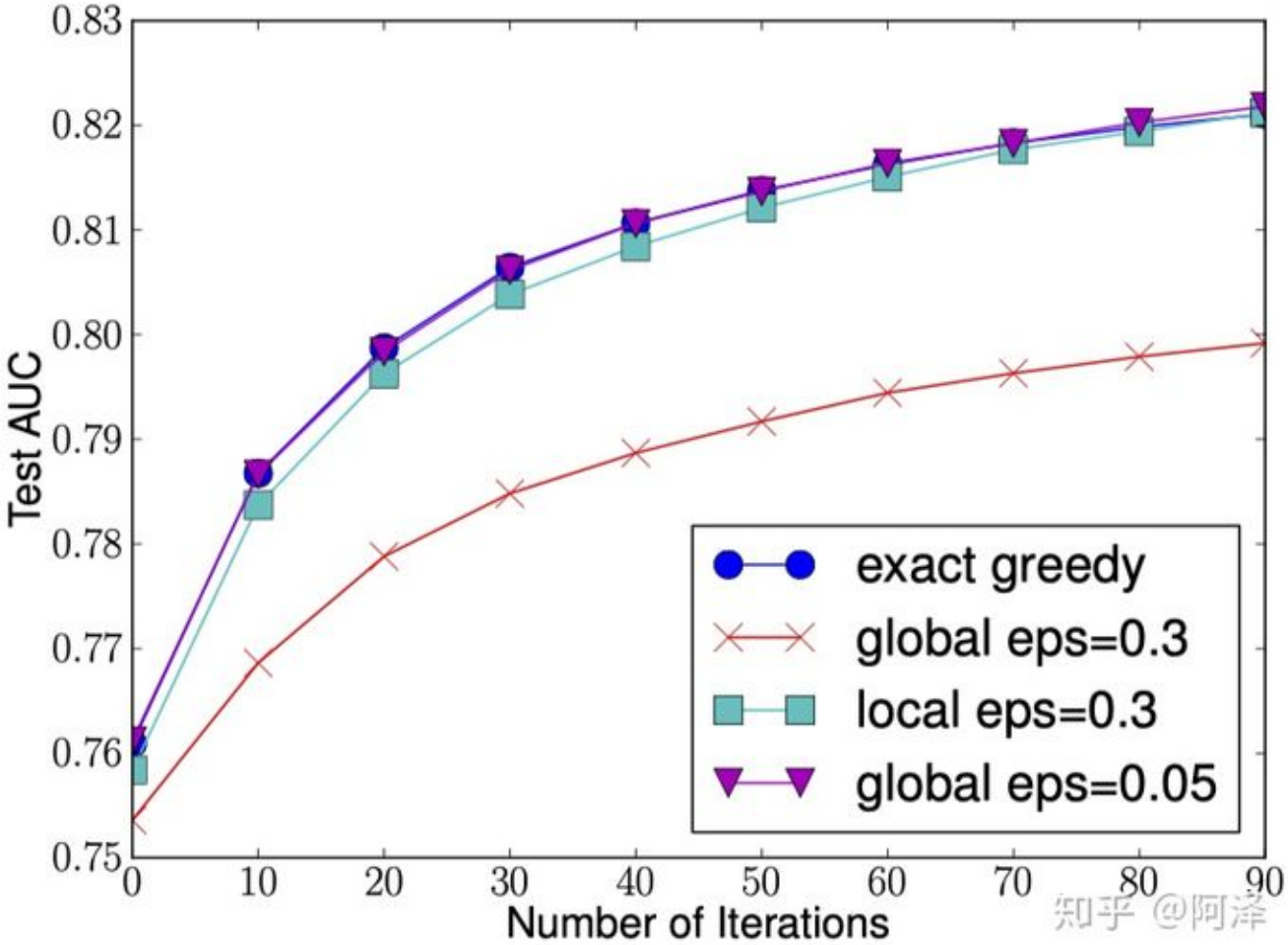
该算法会首先根据特征分布的分位数提出候选划分点，然后将连续型特征映射到由这些候选点划分的桶中，然后聚合统计信息找到所有区间的最佳分裂点。

在提出候选切分点时有两种策略：

• Global：学习每棵树前就提出候选切分点，并在每次分裂时都采用这种分割；
• Local：每次分裂前将重新提出候选切分点。

直观上来看，Local 策略需要更多的计算步骤，而 Global 策略因为节点没有划分所以需要更多的候选点。

下图给出不同种分裂策略的 AUC 变换曲线，横坐标为迭代次数，纵坐标为测试集 AUC，eps 为近似算法的精度，其倒数为桶的数量。



我们可以看到 Global 策略在候选点数多时（eps 小）可以和 Local 策略在候选点少时（eps 大）具有相似的精度。此外我们还发现，在 eps 取值合理的情况下，分位数策略可以获得与贪婪算法相同的精度。

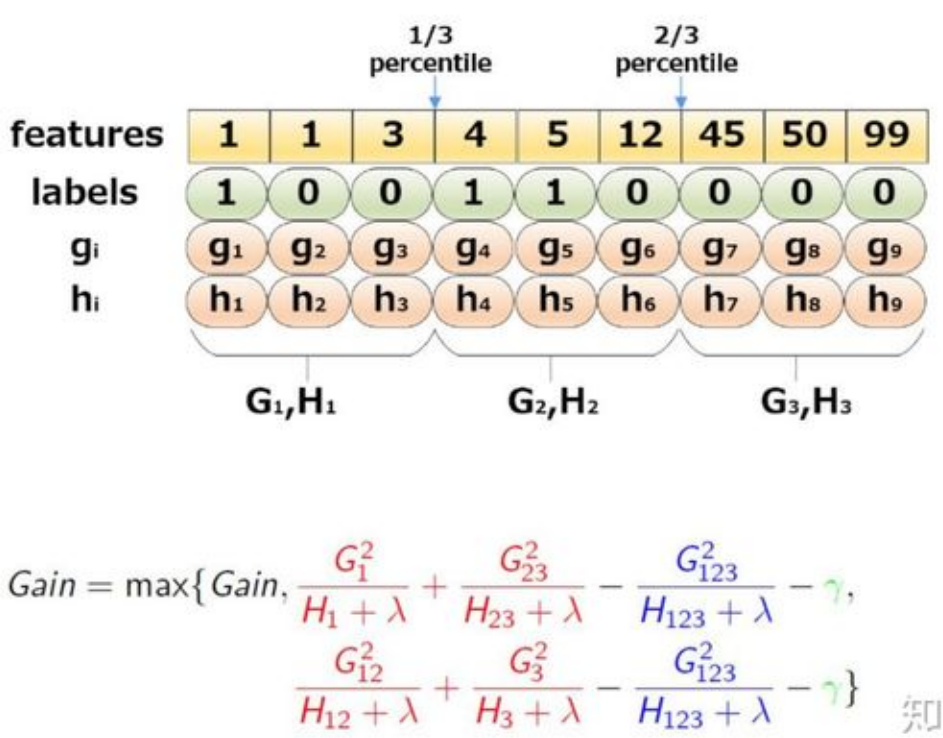
Algorithm 2: Approximate Algorithm for Split Finding

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.

- **第一个 for 循环**：对特征 k 根据该特征分布的分位数找到切割点的候选集合 $S_k = \{s_{k1}, s_{k2}, \ldots, s_{kl}\}$ 。XGBoost 支持 Global 策略和 Local 策略。

- **第二个 for 循环**：针对每个特征的候选集合，将样本映射到由该特征对应的候选点集构成的分桶区间中，即 $s_{k,v} \geq x_{jk} > s_{k,v-1}$ ，对每个桶统计 G, H 值，最后在这些统计量上寻找最佳分裂点。

下图给出近似算法的具体例子，以三分位为例：

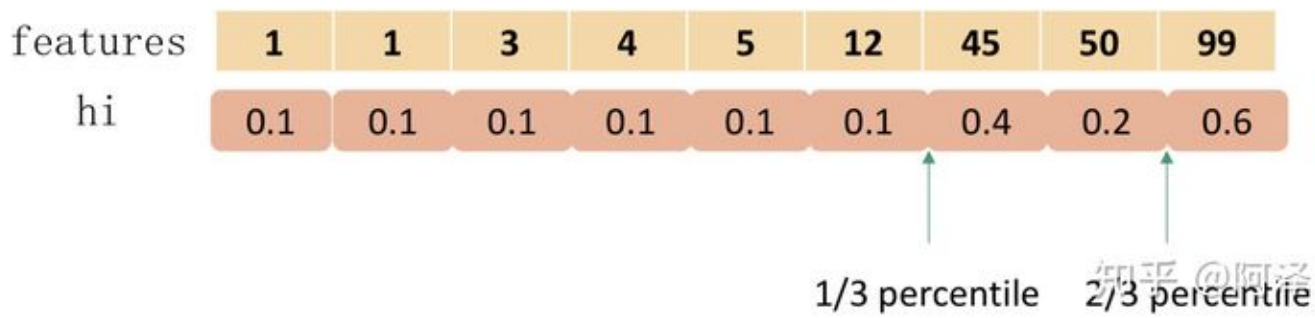


$$Gain = \max\{Gain, \frac{G_1^2}{H_1+\lambda} + \frac{G_{23}^2}{H_{23}+\lambda} - \frac{G_{123}^2}{H_{123}+\lambda} - \gamma, \frac{G_{12}^2}{H_{12}+\lambda} + \frac{G_3^2}{H_3+\lambda} - \frac{G_{123}^2}{H_{123}+\lambda} - \gamma\}$$

根据样本特征进行排序，然后基于分位数进行划分，并统计三个桶内的 G, H 值，最终求解节点划分的增益。

1.1.4 加权分位数缩略图

事实上， XGBoost 不是简单地按照样本个数进行分位，而是以二阶导数值 h_i 作为样本的权重进行划分，如下：



那么问题来了：为什么要用 h_i 进行样本加权？

我们知道模型的目标函数为：

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

我们稍作整理，便可以看出 h_i 有对 loss 加权的作用。

$$\begin{aligned} Obj^{(t)} &\approx \sum_{i=1}^{n}\left[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\right] + \sum_{i=1}^{t}\Omega(f_i) \\ &= \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i) + \frac{1}{2}\frac{g_i^2}{h_i}] + \Omega(f_t) + C \\ &= \sum_{i=1}^{n}\frac{1}{2}h_i\left[f_t(x_i) - \left(-\frac{g_i}{h_i}\right)\right]^2 + \Omega(f_t) + C \end{aligned}$$

其中 $\frac{1}{2}\frac{g_i^2}{h_i}$ 与 C 皆为常数。我们可以看到 h_i 就是平方损失函数中样本的权重。

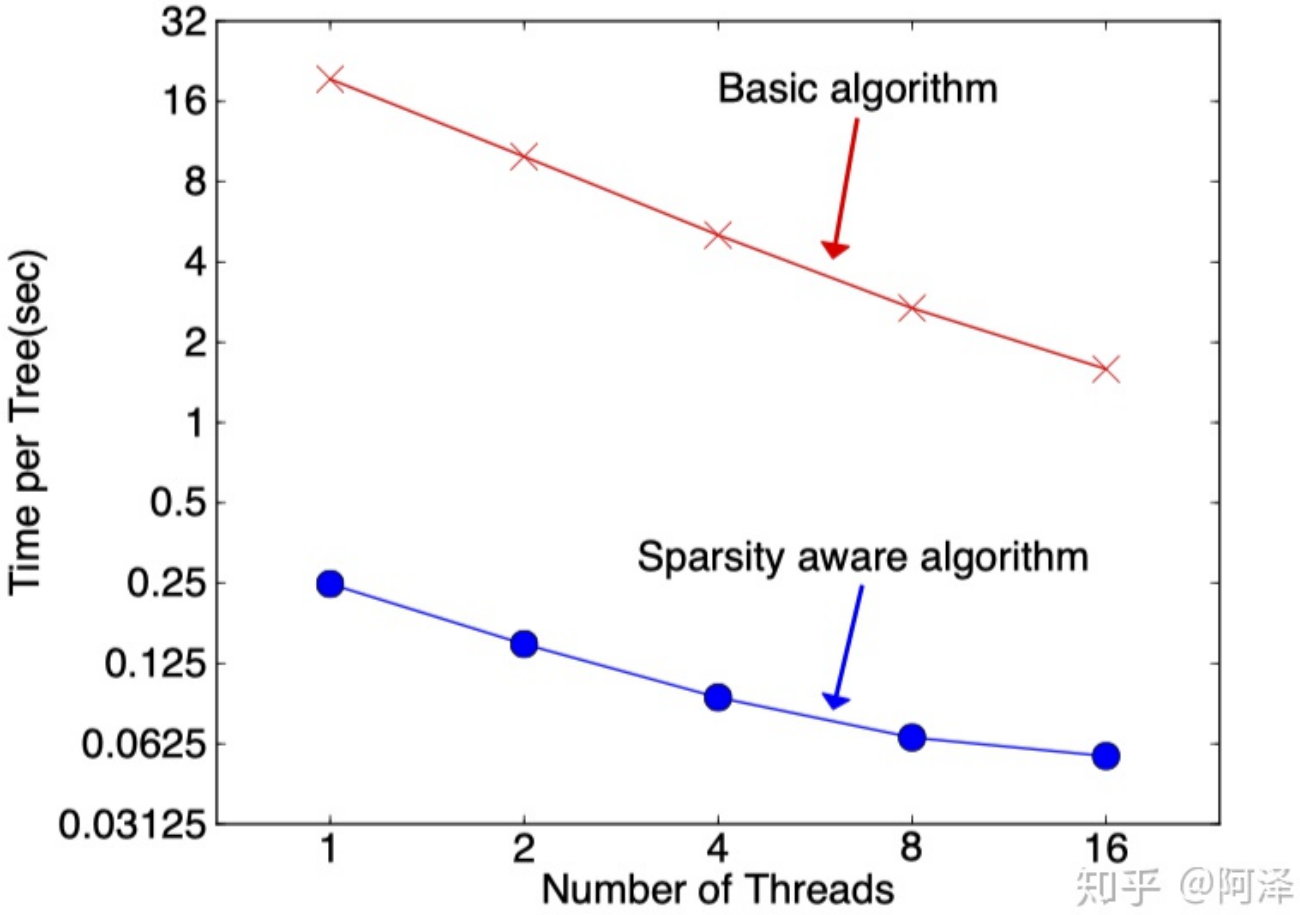
对于样本权值相同的数据集来说，找到候选分位点已经有了解决方案（GK 算法），但是当样本权值不一样时，该如何找到候选分位点呢？（作者给出了一个 Weighted Quantile Sketch 算法，这里将不做介绍。）

1.1.5 稀疏感知算法

在决策树的第一篇文章中我们介绍 CART 树在应对数据缺失时的分裂策略，XGBoost 也给出了其解决方案。

XGBoost 在构建树的节点过程中只考虑非缺失值的数据遍历，而为每个节点增加了一个缺省方向，当样本相应的特征值缺失时，可以被归类到缺省方向上，最优的缺省方向可以从数据中学到。至于如何学到缺省值的分支，其实很简单，分别枚举特征缺省的样本归为左右分支后的增益，选择增益最大的枚举项即为最优缺省方向。

在构建树的过程中需要枚举特征缺失的样本，乍一看该算法的计算量增加了一倍，但其实该算法在构建树的过程中只考虑了特征未缺失的样本遍历，而特征值缺失的样本无需遍历只需直接分配到左右节点，故算法所需遍历的样本量减少，下图可以看到稀疏感知算法比 basic 算法速度块了超过 50 倍。

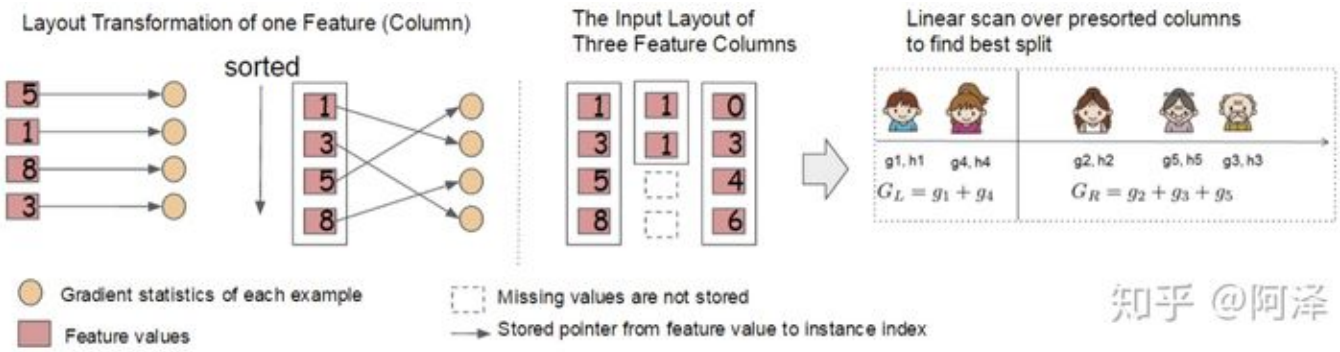


1.2 工程实现

1.2.1 块结构设计

我们知道，决策树的学习最耗时的一个步骤就是在每次寻找最佳分裂点是都需要对特征的值进行排序。而 XGBoost 在训练之前对根据特征对数据进行了排序，然后保存到块结构中，并在每个块结构中都采用了稀疏矩阵存储格式（Compressed Sparse Columns Format，CSC）进行存储，后面的训练过程中会重复地使用块结构，可以大大减小计算量。

- 每一个块结构包括一个或多个已经排序好的特征；
- 缺失特征值将不进行排序；
- 每个特征会存储指向样本梯度统计值的索引，方便计算一阶导和二阶导数值；



这种块结构存储的特征之间相互独立，方便计算机进行并行计算。在对节点进行分裂时需要选择增益最大的特征作为分裂，这时各个特征的增益计算可以同时进行，这也是 Xgboost 能够实现分布式或者多线程计算的原因。

1.2.2 缓存访问优化算法

块结构的设计可以减少节点分裂时的计算量，但特征值通过索引访问样本梯度统计值的设计会导致访问操作的内存空间不连续，这样会造成缓存命中率低，从而影响到算法的效率。

为了解决缓存命中率低的问题，XGBoost 提出了缓存访问优化算法：为每个线程分配一个连续的缓存区，将需要的梯度信息存放在缓冲区中，这样就是实现了非连续空间到连续空间的转换，提高了算法效率。

此外适当调整块大小，也可以有助于缓存优化。

1.2.3 “核外”块计算

当数据量过大时无法将数据全部加载到内存中，只能先将无法加载到内存中的数据暂存到硬盘中，直到需要时再进行加载计算，而这种操作必然涉及到因内存与硬盘速度不同而造成的资源浪费和性能瓶颈。为了解决这个问题，XGBoost 独立一个线程专门用于从硬盘读入数据，以实现处理数据和读入数据同时进行。

此外，XGBoost 还用了两种方法来降低硬盘读写的开销：

- **块压缩：**对 Block 进行按列压缩，并在读取时进行解压；
- **块拆分：**将每个块存储到不同的磁盘中，从多个磁盘读取可以增加吞吐量。

1.3 优缺点

1.3.1 优点

1. **精度更高：**GBDT 只用到一阶泰勒展开，而 XGBoost 对损失函数进行了二阶泰勒展开。XGBoost 引入二阶导一方面是为了增加精度，另一方面也是为了能够自定义损失函数，二阶泰勒展开可以近似大量损失函数；
2. **灵活性更强：**GBDT 以 CART 作为基分类器，XGBoost 不仅支持 CART 还支持线性分类器，（使用线性分类器的 XGBoost 相当于带 L1 和 L2 正则化项的逻辑斯蒂回归（分类问题）或者线性回归（回归问题））。此外，XGBoost 工具支持自定义损失函数，只需函数支持一阶和二阶求导；
3. **正则化：**XGBoost 在目标函数中加入了正则项，用于控制模型的复杂度。正则项里包含了树的叶子节点个数、叶子节点权重的 L2 范式。正则项降低了模型的方差，使学习出来的模型更加简单，有助于防止过拟合；
4. **Shrinkage（缩减）：**相当于学习速率。XGBoost 在进行完一次迭代后，会将叶子节点的权重乘上该系数，主要是为了削弱每棵树的影响，让后面有更大的学习空间；

5. **列抽样：**XGBoost 借鉴了随机森林的做法，支持列抽样，不仅能降低过拟合，还能减少计算；
6. **缺失值处理：**XGBoost 采用的稀疏感知算法极大的加快了节点分裂的速度；
7. **可以并行化操作：**块结构可以很好的支持并行计算。

1.3.2 缺点

1. 虽然利用预排序和近似算法可以降低寻找最佳分裂点的计算量，但在节点分裂过程中仍需要遍历数据集；
2. 预排序过程的空间复杂度过高，不仅需要存储特征值，还需要存储特征对应样本的梯度统计值的索引，相当于消耗了两倍的内存。

2. LightGBM

LightGBM 由微软提出，主要用于解决 GDBT 在海量数据中遇到的问题，以便其可以更好更快地用于工业实践中。

从 LightGBM 名字我们可以看出其是轻量级（Light）的梯度提升机（GBM），其相对 XGBoost 具有训练速度快、内存占用低的特点。下图分别显示了 XGBoost、XGBoost_hist（利用梯度直方图的 XGBoost） 和 LightGBM 三者之间针对不同数据集情况下的内存和训练时间的对比：

Data	xgboost	xgboost_hist	LightGBM
Higgs	4.853GB	3.784GB	**0.868GB**
Yahoo LTR	1.907GB	1.468GB	**0.831GB**
MS LTR	5.469GB	3.654GB	**0.886GB**
Expo	1.553GB	1.393GB	**0.543GB**
Allstate	6.237GB	4.990GB	**1.027GB**

Data	xgboost	xgboost_hist	LightGBM
Higgs	3794.34 s	551.898 s	**238.505513 s**
Yahoo LTR	674.322 s	265.302 s	**150.18644 s**
MS LTR	1251.27 s	385.201 s	**215.320316 s**
Expo	1607.35 s	588.253 s	**138.504179 s**
Allstate	2867.22 s	1355.71 s	**348.084475 s**

那么 LightGBM 到底如何做到更快的训练速度和更低的内存使用的呢？

我们刚刚分析了 XGBoost 的缺点，LightGBM 为了解决这些问题提出了以下几点解决方案：

1. 单边梯度抽样算法；
2. 直方图算法；
3. 互斥特征捆绑算法；
4. 基于最大深度的 Leaf-wise 的垂直生长算法；
5. 类别特征最优分割；
6. 特征并行和数据并行；
7. 缓存优化。

本节将继续从数学原理和工程实现两个角度介绍 LightGBM。

2.1 数学原理

2.1.1 单边梯度抽样算法

GBDT 算法的梯度大小可以反应样本的权重，梯度越小说明模型拟合的越好，单边梯度抽样算法（Gradient-based One-Side Sampling, GOSS）利用这一信息对样本进行抽样，减少了大量梯度小的样本，在接下来的计算锅中只需关注梯度高的样本，极大的减少了计算量。

GOSS 算法保留了梯度大的样本，并对梯度小的样本进行随机抽样，为了不改变样本的数据分布，在计算增益时为梯度小的样本引入一个常数进行平衡。具体算法如下所示：

Algorithm 2: Gradient-based One-Side Sampling

Input: I: training data, d: iterations
Input: a: sampling ratio of large gradient data
Input: b: sampling ratio of small gradient data
Input: $loss$: loss function, L: weak learner
models $\leftarrow \{\}$, fact $\leftarrow \frac{1-a}{b}$
topN $\leftarrow$ a $\times$ len(I) , randN $\leftarrow$ b $\times$ len(I)
for $i = 1$ ***to*** d **do**
    preds $\leftarrow$ models.predict(I)
    g $\leftarrow$ $loss$(I, preds), w $\leftarrow$ {1,1,...}
    sorted $\leftarrow$ GetSortedIndices(abs(g))
    topSet $\leftarrow$ sorted[1:topN]
    randSet $\leftarrow$ RandomPick(sorted[topN:len(I)], randN)
    usedSet $\leftarrow$ topSet + randSet
    w[randSet] $\times =$ fact $\triangleright$ Assign weight $fact$ to the small gradient data.
    newModel $\leftarrow$ L(I[usedSet], $-$ g[usedSet], w[usedSet])
    models.append(newModel)

我们可以看到 GOSS 事先基于梯度的绝对值对样本进行排序（**无需保存排序后结果**），然后拿到前 a% 的梯度大的样本，和总体样本的 b%，在计算增益时，通过乘上 $\frac{1-a}{b}$ 来放大梯度小的样本的权重。**一方面算法将更多的注意力放在训练不足的样本上，另一方面通过乘上权重来防止采样对原始数据分布造成太大的影响。**

2.1.2 直方图算法

1) 直方图算法

直方图算法的基本思想是将连续的特征离散化为 k 个离散特征，同时构造一个宽度为 k 的直方图用于统计信息（含有 k 个 bin）。利用直方图算法我们无需遍历数据，只需要遍历 k 个 bin 即可找到最佳分裂点。

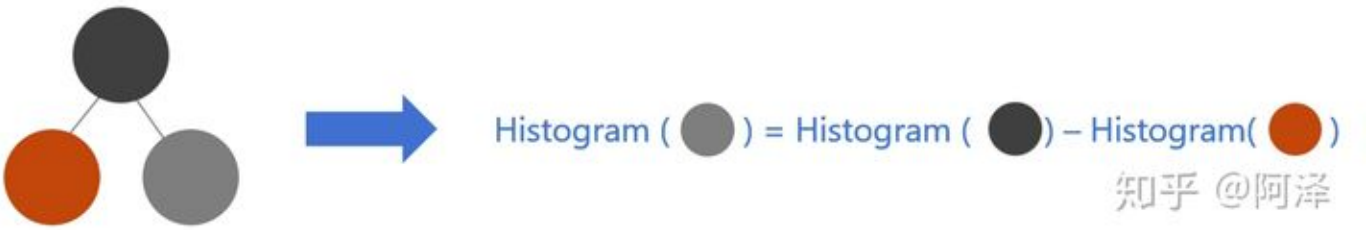
我们知道特征离散化的具有很多优点，如存储方便、运算更快、鲁棒性强、模型更加稳定等等。对于直方图算法来说最直接的有以下两个优点（以 k=256 为例）：

- **内存占用更小：**XGBoost 需要用 32 位的浮点数去存储特征值，并用 32 位的整形去存储索引，而 LightGBM 只需要用 8 位去存储直方图，相当于减少了 1/8；
- **计算代价更小：**计算特征分裂增益时，XGBoost 需要遍历一次数据找到最佳分裂点，而 LightGBM 只需要遍历一次 k 次，直接将时间复杂度从 $O(\#data * \#feature)$ 降低到 $O(k * \#feature)$，而我们知道 $\#data >> k$ 。

虽然将特征离散化后无法找到精确的分割点，可能会对模型的精度产生一定的影响，但较粗的分割也起到了正则化的效果，一定程度上降低了模型的方差。

2) 直方图加速

在构建叶节点的直方图时，我们还可以通过父节点的直方图与相邻叶节点的直方图相减的方式构建，从而减少了一半的计算量。在实际操作过程中，我们还可以先计算直方图小的叶子节点，然后利用直方图作差来获得直方图大的叶子节点。



3) 稀疏特征优化

XGBoost 在进行预排序时只考虑非零值进行加速，而 LightGBM 也采用类似策略：只用非零特征构建直方图。

2.1.3 互斥特征捆绑算法

高维特征往往是稀疏的，而且特征间可能是相互排斥的（如两个特征不同时取非零值），如果两个特征并不完全互斥（如只有一部分情况下是不同时取非零值），可以用互斥率表示互斥程度。互斥特征捆绑算法（Exclusive Feature Bundling, EFB）指出如果将一些特征进行融合绑定，则可以降低特征数量。

针对这种想法，我们会遇到两个问题：

1. 哪些特征可以一起绑定？
2. 特征绑定后，特征值如何确定？

对于问题一：EFB 算法利用特征和特征间的关系构造一个加权无向图，并将其转换为图着色算法。我们知道图着色是个 NP-Hard 问题，故采用贪婪算法得到近似解，具体步骤如下：

1. 构造一个加权无向图，顶点是特征，边是两个特征间互斥程度；
2. 根据节点的度进行降序排序，度越大，与其他特征的冲突越大；
3. 遍历每个特征，将它分配给现有特征包，或者新建一个特征包，是的总体冲突最小。

算法允许两两特征并不完全互斥来增加特征捆绑的数量，通过设置最大互斥率 γ 来平衡算法的精度和效率。EFB 算法的伪代码如下所示：

Algorithm 3: Greedy Bundling

Input: F: features, K: max conflict count
Construct graph G
searchOrder $\leftarrow$ G.sortByDegree()
bundles $\leftarrow$ {}, bundlesConflict $\leftarrow$ {}
for i ***in*** *searchOrder* **do**
    needNew $\leftarrow$ True
    for $j = 1$ ***to*** *len(bundles)* **do**
        cnt $\leftarrow$ ConflictCnt(bundles[j],F[i])
        if *cnt + bundlesConflict[i]* $\leq K$ **then**
            bundles[j].add(F[i]), needNew $\leftarrow$ False
            break
    if *needNew* **then**
        Add $F[i]$ as a new bundle to *bundles*
Output: *bundles*

我们看到时间复杂度为 $O(\#feature^2)$ ，在特征不多的情况下可以应付，但如果特征维度达到百万级别，计算量则会非常大，为了改善效率，我们提出了一个更快的解决方案：将 EFB 算法中通过构建图，根据节点度来排序的策略改成了根据非零值的技术排序，因为非零值越多，互斥的概率会越大。

对于问题二：论文给出特征合并算法，其关键在于原始特征能从合并的特征中分离出来。假设 Bundle 中有两个特征值，A 取值为 [0, 10]、B 取值为 [0, 20]，为了保证特征 A、B 的互斥性，我们可以给特征 B 添加一个偏移量转换为 [10, 30]，Bundle 后的特征其取值为 [0, 30]，这样便实现了特征合并。具体算法如下所示：

Algorithm 4: Merge Exclusive Features

Input: $numData$: number of data
Input: F: One bundle of exclusive features
binRanges $\leftarrow \{0\}$, totalBin $\leftarrow 0$
for f ***in*** F **do**
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin $\leftarrow$ new Bin(numData)
for $i = 1$ ***to*** $numData$ **do**
    newBin[i] $\leftarrow 0$
    for $j = 1$ ***to*** $len(F)$ **do**
        if $F[j].bin[i] \neq 0$ **then**
            newBin[i] $\leftarrow$ F[j].bin[i] + binRanges[j]
Output: $newBin, binRanges$

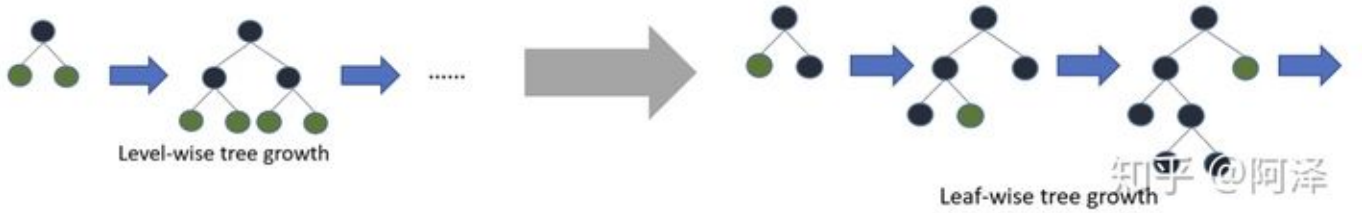
2.1.4 带深度限制的 Leaf-wise 算法

在建树的过程中有两种策略：

- Level-wise：基于层进行生长，直到达到停止条件；
- Leaf-wise：每次分裂增益最大的叶子节点，直到达到停止条件。

XGBoost 采用 Level-wise 的增长策略，方便并行计算每一层的分裂节点，提高了训练速度，但同时也因为节点增益过小增加了很多不必要的分裂，降低了计算量；LightGBM 采用 Leaf-wise 的

增长策略减少了计算量，配合最大深度的限制防止过拟合，由于每次都需要计算增益最大的节点，所以无法并行分裂。

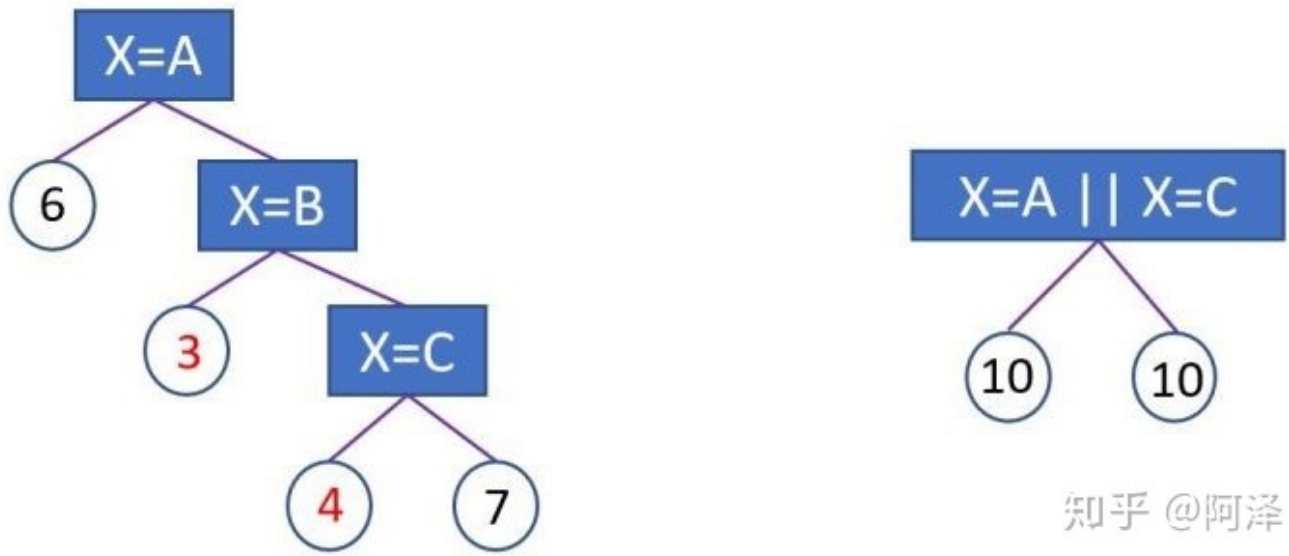


2.1.5 类别特征最优分割

大部分的机器学习算法都不能直接支持类别特征，一般都会对类别特征进行编码，然后再输入到模型中。常见的处理类别特征的方法为 one-hot 编码，但我们知道对于决策树来说并不推荐使用 one-hot 编码：

1. 会产生样本切分不平衡问题，切分增益会非常小。如，国籍切分后，会产生是否中国，是否美国等一系列特征，这一系列特征上只有少量样本为 1，大量样本为 0。这种划分的增益非常小：较小的那个拆分样本集，它占总样本的比例太小。无论增益多大，乘以该比例之后几乎可以忽略；较大的那个拆分样本集，它几乎就是原始的样本集，增益几乎为零；
2. 影响决策树学习：决策树依赖的是数据的统计信息，而独热码编码会把数据切分到零散的小空间上。在这些零散的小空间上统计信息不准确的，学习效果变差。本质是因为独热码编码之后的特征的表达能力较差的，特征的预测能力被人为的拆分成多份，每一份与其他特征竞争最优划分点都失败，最终该特征得到的重要性会比实际值低。

LightGBM 原生支持类别特征，采用 many-vs-many 的切分方式将类别特征分为两个子集，实现类别特征的最优切分。假设有某维特征有 k 个类别，则有 $2^{(k-1)}-1$ 中可能，时间复杂度为 $O(2^k)$，LightGBM 基于 Fisher 大佬的《On Grouping For Maximum Homogeneity》实现了 $O(klogk)$ 的时间复杂度。



上图为左边为基于 one-hot 编码进行分裂，后图为 LightGBM 基于 many-vs-many 进行分裂，在给定深度情况下，后者能学出更好的模型。

其基本思想在于每次分组时都会根据训练目标对类别特征进行分类，根据其累积值 $\frac{\sum gradient}{\sum hessian}$ 对直方图进行排序，然后在排序的直方图上找到最佳分割。此外，LightGBM 还加了约束条件正则化，防止过拟合。

Comparison (500 tress, each with 255 leaves):

Data		One-hot	Optimal
Expo	AUC	0.78368	0.79847
	Time	138 s	165 s

我们可以看到这种处理类别特征的方式使得 AUC 提高了 1.5 个点，且时间仅仅多了 20%。

2.2 工程实现

2.2.1 特征并行

传统的特征并行算法在于对数据进行垂直划分，然后使用不同机器找到不同特征的最优分裂点，基于通信整合得到最佳划分点，然后基于通信告知其他机器划分结果。

传统的特征并行方法有个很大的缺点：需要告知每台机器最终划分结果，增加了额外的复杂度（因为对数据进行垂直划分，每台机器所含数据不同，划分结果需要通过通信告知）。

LightGBM 则不进行数据垂直划分，每台机器都有训练集完整数据，在得到最佳划分方案后可在本地执行划分而减少了不必要的通信。

2.2.2 数据并行

传统的数据并行策略主要为水平划分数据，然后本地构建直方图并整合成全局直方图，最后在全局直方图中找出最佳划分点。

这种数据划分有一个很大的缺点：通讯开销过大。如果使用点对点通信，一台机器的通讯开销大约为 $O(\#machine * \#feature * \#bin)$；如果使用集成的通信，则通讯开销为 $O(2 * \#feature * \#bin)$，

LightGBM 采用分散规约（Reduce scatter）的方式将直方图整合的任务分摊到不同机器上，从而降低通信代价，并通过直方图做差进一步降低不同机器间的通信。

2.2.3 投票并行

针对数据量特别大特征也特别多的情况下，可以采用投票并行。投票并行主要针对数据并行时数据合并的通信代价比较大的瓶颈进行优化，其通过投票的方式只合并部分特征的直方图从而达到降低通信量的目的。

大致步骤为两步：

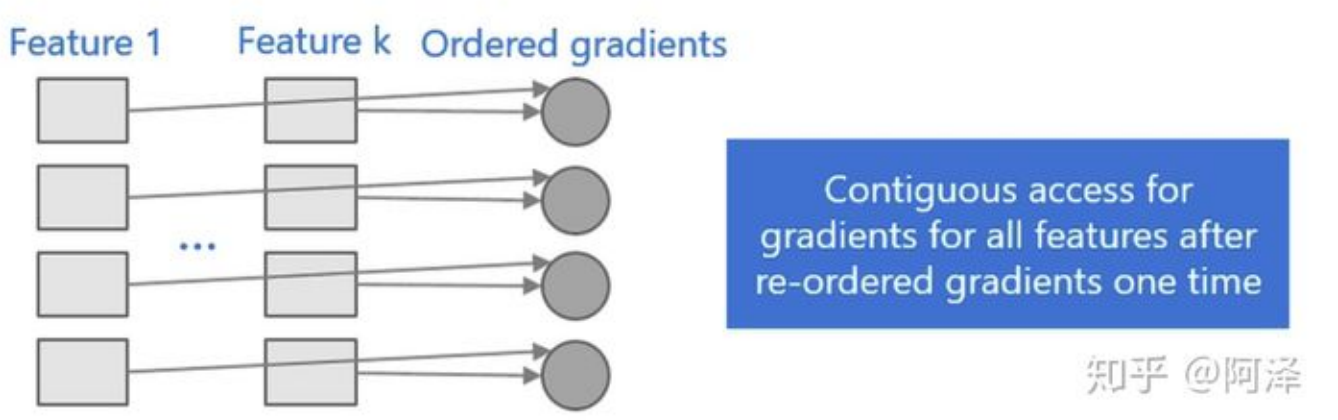
1. 本地找出 Top K 特征，并基于投票筛选出可能是最优分割点的特征；
2. 合并时只合并每个机器选出来的特征。

2.2.4 缓存优化

上边说到 XGBoost 的预排序后的特征是通过索引给出的样本梯度的统计值，因其索引访问的结果并不连续，XGBoost 提出缓存访问优化算法进行改进。

而 LightGBM 所使用直方图算法对 Cache 天生友好：

1. 首先，所有的特征都采用相同的方法获得梯度（区别于不同特征通过不同的索引获得梯度），只需要对梯度进行排序并可实现连续访问，大大提高了缓存命中；
2. 其次，因为不需要存储特征到样本的索引，降低了存储消耗，而且也不存在 Cache Miss的问题。



2.3 与 XGBoost 的对比

本节主要总结下 LightGBM 相对于 XGBoost 的优点，从内存和速度两方面进行介绍。

2.3.1 内存更小

1. XGBoost 使用预排序后需要记录特征值及其对应样本的统计值的索引，而 LightGBM 使用了直方图算法将特征值转变为 bin 值，且不需要记录特征到样本的索引，将空间复杂度从 $O(2*\#data)$ 降低为 $O(\#bin)$ ，极大的减少了内存消耗；
2. LightGBM 采用了直方图算法将存储特征值转变为存储 bin 值，降低了内存消耗；

3. LightGBM 在训练过程中采用互斥特征捆绑算法减少了特征数量，降低了内存消耗。

2.3.2 速度更快

1. LightGBM 采用了直方图算法将遍历样本转变为遍历直方图，极大的降低了时间复杂度；
2. LightGBM 在训练过程中采用单边梯度算法过滤掉梯度小的样本，减少了大量的计算；
3. LightGBM 采用了基于 Leaf-wise 算法的增长策略构建树，减少了很多不必要的计算量；
4. LightGBM 采用优化后的特征并行、数据并行方法加速计算，当数据量非常大的时候还可以采用投票并行的策略；
5. LightGBM 对缓存也进行了优化，增加了 Cache hit 的命中率。

参考文献

1. 《XGBoost: A Scalable Tree Boosting System》
2. 陈天奇论文演讲 PPT
3. 机器学习算法中 GBDT 和 XGBOOST 的区别有哪些？ - wepon的回答 - 知乎
4. LightGBM: A Highly Efficient Gradient Boosting Decision Tree
5. LightGBM 文档
6. 论文阅读——LightGBM 原理
7. 机器学习算法之 LightGBM
8. 关于sklearn中的决策树是否应该用one-hot编码？ - 柯国霖的回答 - 知乎
9. 如何玩转LightGBM
10. A Communication-Efficient Parallel Algorithm for Decision Tree.