1、Boosting 方法简介

Boosting 是一族可将弱学习器提升为强学习器的算法。关于 Boosting 的两个核心问题：

1、在每一轮如何改变训练数据的权值或概率分布？

通过提高那些在前一轮被弱分类器分错样例的权值，减小前一轮分对样本的权值，而误分的样本在后续受到更多的关注.

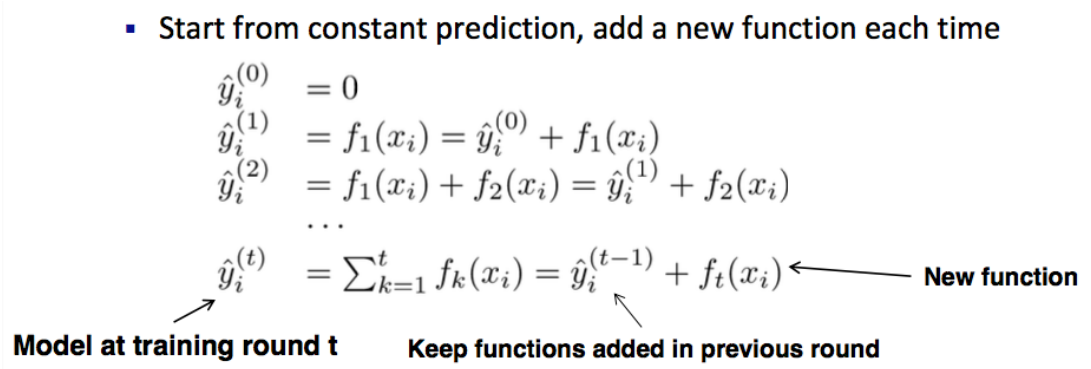
2、通过什么方式来组合弱分类器？

通过加法模型将弱分类器进行线性组合，比如 AdaBoost 通过加权多数表决的方式，即增大错误率小的分类器的权值，同时减小错误率较大的分类器的权值。而提升树通过拟合残差的方式逐步减小残差，将每一步生成的模型叠加得到最终模型。

我们常见的 Boosting 算法有 **AdaBoost，梯度提升决策树 GBDT，XgBoost 以及 LightGBM**。

2、前向分步算法

前向分步算法是大多 Boosting 算法的一个基础，其基本思想是：从前向后，每一步只学习一个基函数，逐步逼近优化目标函数式：



3、AdaBoost 算法

3.1 核心思想

AdaBoost 算法的核心思想有两个：

1、在每轮的迭代中，**提高那些前一轮弱分类器分类错误样本的权值，降低那些被正确分类样本的权值**，这样一来，那些没有得到正确分类的数据，由于其权值加大后受到后一轮若分类器的更大关注，于是，分类问题被一系列弱分类器**“分而治之”**。

2、采用**加权多数表决**的方法来组合各个弱分类器，具体地说，加大分类误差率大的弱分类器的权值，使其在表决中起到较大的作用，减小分类误差率大的弱分类器的权值，使其在表决中起到较小的作用。

3.2 基本流程

下面的流程选自李航博士的《统计学习方法》，我们重点关注的是从前向分步算法角度 AdaBoost 的推导，因此这里一步带过，感兴趣的可以看原书的相关章节。

输入：训练数据集$T=\{(x_1,y_1),(x_2,y_2),\ldots\ldots,(x_N,y_N)\}$,其中$x_i \in X \subseteq R^n, y_i \in \gamma=\{-1,+1\}$

输出：最终分类器$G(X)$.

(1)初始化训练数据的权值分布

$$D_1=(w_{11},\ldots\ldots w_{1i},\ldots\ldots\ldots w_{1N}), \qquad w_{1i}=\frac{1}{N}, i=1,2,\ldots.N$$

(2) 对 m=1，2，… M

（a）使用具有权值分布D_m的训练数据集学习，得到基本分类器

$$G_m(x): X \to \{-1,+1\}$$

（b）计算$G_m(x)$在训练数据集上的分类误差率

$$e_m=P(G_m(x_i)\neq y_i)=\sum_{i=1}^{N} w_{mi} I(G_m(x_i)\neq y_i)$$

（c）计算$G_m(x)$的系数

$$a_m=\frac{1}{2}\log\frac{1-e_m}{e_m}$$

（d）更新训练数据的权值分布

$$D_{m+1}=\left(w_{m+1,1},\ldots\ldots w_{m+1,i},\ldots\ldots\ldots w_{m+1,N}\right)$$

$$w_{m+1,i}=\frac{w_{mi}}{Z_m}\exp\left(-a_m y_i G_m(x_i)\right)$$

这里，Z_m是规范化因子

$$Z_m=\sum_{i=1}^{N} w_{mi}\exp\left(-a_m y_i G_m(x_i)\right)$$

（3）构建基本分类器的线性组合

$$f(x)=\sum_{i=1}^{M} a_m G_m(x)$$

得到最终分类器

$$G(x)=sign(f(x)=sign(\sum_{i=1}^{M} a_m G_m(x))$$

3.3 从前向分步算法来看 AdaBoost

AdaBoost 算法是前向分步加法算法的特例，这时，模型是由**基本分类器组成的加法模型，损失函数是指数函数**。

前向分步算法会逐一学习每个基函数$G(x)$，这一过程与 AdaBoost 算法逐一学习基本分类器的过程一致。AdaBoost 算法的损失函数为指数函数，其形式如下：

$$L\big(y, f(x)\big) = \exp\left[-yf(x)\right]$$

假设我们经过了 m-1 轮的迭代，得到：

$$f_{m-1}(x) = a_1 G_1(x) + a_2 G_2(x) + \cdots + a_{m-1} G_{m-1}(x)$$

在 m 轮继续迭代得到：

$$f_m(x) = f_{m-1}(x) + a_m G_m(x)$$

第 m 轮的迭代目标是使前向分步算法得到的系数和基本类器在训练数据集 T 上的指数损失最小，即：

$$\big(a_m, G_m(x)\big) = argmin \sum_{i=1}^{N} \overline{w_{mi}} \exp\left(-a_m y_i G_m(x_i)\right)$$

其中，$\overline{w_{mi}} = \exp\left[-y_i f_{m-1}(x_i)\right]$, 咦，是不是感觉跟之前见过的不太一样？哈哈，对的，是有一点不一样，没有进行规范化处理，不过其他是一样的，因为：

$$\begin{aligned}\overline{w_{mi}} &= \exp[-y_i f_{m-1}(x_i)] \\ &= \exp[-a_{m-1} y_i G_{m-1}(x_i)] * \exp[-a_{m-2} y_i G_{m-2}(x_i)] * \ldots \\ &* \exp[-a_1 y_i G_1(x_i)]\end{aligned}$$

对于上面的式子，我们首先来求解$G^*(x)$, 对任意的$a > 0$, $G_m^*(x)$由下式得到：

$$G_m^*(x) = \operatorname{argmin} \sum_{i=1}^{N} \overline{w_{mi}} I(G(x_i) \neq y_i)$$

此分类器$G_m^*(x)$就是 AdaBoost 算法的基本分类器$G_m(x)$，因为它是使第 m 轮加权训练数据分类误差率最小的基本分类器。

之后求解a^*,

$$\begin{aligned}\sum_{i=1}^{N} \overline{w_{mi}} \exp\big(-a_m y_i G_m(x_i)\big) &= \sum_{i=G_m(x_i)} \overline{w_{mi}}\, e^{-a} + \sum_{y_i \neq G_m(x_i)} \overline{w_{mi}}\, e^{a} \\ &= (e^{a} - e^{-a}) \sum_{i=1}^{N} \overline{w_{mi}} I(G(x_i) \neq y_i) + e^{-a} \sum_{i=1}^{N} \overline{w_{mi}} \\ &= (e^{a} - e^{-a}) e_m + e^{-a}\end{aligned}$$

对上式求导之后可以得到：

$$a^* = \frac{1}{2} \log \frac{1 - e_m}{e_m}$$

这跟我们刚才的计算过程完全一致，因此我们说当使用指数损失函数时，AdaBoost 就是前向加法模型的一种特例。

4、梯度提升决策树(GBDT)

4.1 提升树-boosting tree

以决策树为基函数的提升方法称为提升树，其决策树可以是分类树或者回归树。提升树模型可以表示为决策树的加法模型。

$$f_M(x) = \sum_{m=1}^{M} T(x;\Theta_m)$$

其中，$T(x;\theta_m)$表示决策树，θ_m表示树的参数，M 为树的个数。
针对不同的问题的提升树算法主要的区别就在于损失函数的不同，对于回归问题来说，我们使用的是平方损失函数，对于分类问题来说，我们使用的是指数损失函数。对二分类问题来说，提升树算法只需将 AdaBoost 的基分类器设置为二分类树即可，可以说此时的提升树算法时 AdaBoost 算法的一个特例。所以我们不再描述，我们主要关注回归问题的提升树算法。

对于回归问题的提升树算法来说，我们每一步主要拟合的是前一步的残差，为什么是残差，看下面的公式推导：

$$\begin{aligned} L\big(y, f_m(x)\big) &= L\big(y, f_{m-1}(x) + T(x;\theta_m)\big) \\ &= [y - f_{m-1}(x) - T(x;\theta_m)]^2 \\ &= [r - T(x;\theta_m)]^2 \end{aligned}$$

其中，r 代表的就是残差。
因此，我们得到回归问题中的提升树算法如下：

输入：训练数据集$T = \{(x_1, y_1), (x_2, y_2), \ldots\ldots, (x_N, y_N)\}$,其中$x_i \in X \subseteq R^n, y_i \in \gamma = \{-1, +1\}$
输出：提升树$f_M(x)$.
(1)初始化$f_0(x) = 0$
(2)对 m=1,2…M
(a) 计算每个数据的残差：

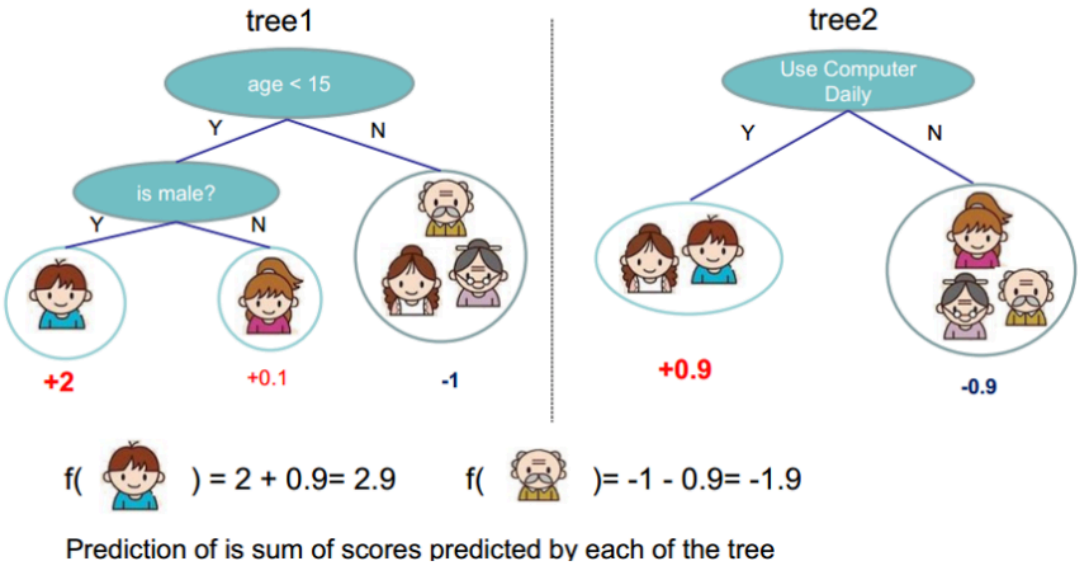
$$r_{mi} = y_i - f_{m-1}(x_i), i = 1,2, \ldots.N$$

(b) 拟合残差学习一颗回归树，得到$T(x;\theta_m)$
(c) 更新$f_m(x) = f_{m-1}(x) + T(x;\theta_m)$

(3) 得到回归问题提升树

$$f_M(x) = \sum_{m=1}^{M} T(x;\theta_m)$$

得到一颗提升树之后，我们可以对输入数据进行预测，假设得到了两棵树，下图给出了预测过程：



4.2 梯度提升-Gradient Boosting

梯度提升的思想借鉴于梯度下降法，我们先来回顾一下梯度下降法。对于优化问题：min $f(w)$

使用梯度下降法求解的基本步骤为：

(1)随机选择一个初始点w_0

(2)重复以下过程：

a. 决定下降方向：$d_i = -\frac{\partial}{\partial w} f(w)|_{w_i}$

b. 选择步长α

c. 更新：$w_{i+1} = w_i + \alpha * d_i$

(3)直到满足终止条件

由以上的过程，我们可以看出，对于最终的最优解w^*,是由初始值w_0经过 M 代迭代之后得到的，在这里设$w_0 = d_0$,则w^*为：

$$w^* = \sum_{i=0}^{M} \alpha_i * d_i$$

在函数空间中，我们也可以借鉴梯度下降的思想，进行最优函数的搜索。对于模型的损失函数$L\big(y, F(X)\big)$，为了能够求解出最优的函数$F^*(X)$,首先，设置初始值为：

$$F_0(X) = f_0(x)$$

以函数 F(X)作为一个整体，与梯度下降法的更新过程一致，假设经过 M 代，得到最优的函数$F^*(X)$为：

$$F^*(X) = \sum_{i=0}^{M} f_i(x)$$

其中，$f_i(x)$为：

$$f_i(x) = -\alpha_i g_i(X) = -\alpha_i * [\frac{\partial L\big(y, F(X)\big)}{\partial F(X)}]_{F(X)=F_{m-1}(X)}$$

可以看到，这里的梯度变量是一个函数，是在函数空间上求解，而以往算法梯度下降是在 N 维的参数空间的负梯度方向，变量是参数。这里的变量是函数，更新函数通过当前函数的负梯度方向来修正模型，使它更优，最后累加的模型近似最优函数。

4.3 梯度提升决策树(GBDT)

了解了 GBDT 的两个部分，提升树和梯度提升之后，我们可以顺利得到 GBDT 的实现思路（这里仍然以回归树为例，基模型是 CART 回归树）：

输入：训练数据集$T = \{(x_1, y_1), (x_2, y_2), \ldots\ldots, (x_N, y_N)\}$,其中$x_i \in X \subseteq R^n, y_i \in \gamma = \{-1, +1\}$

输出：提升树$f_M(x)$.

(1)初始化$f_0(x) = argmin \sum_{i=1}^{N} L(y_i, c)$

(2)对 m=1,2…M

(a) 计算每个数据：$r_{mi} = [\frac{\partial L(y, f(x_i))}{\partial f(x_i)}]_{f(x)=f_{m-1}(x)}$

(b) 拟合r_{mi}学习一颗回归树，得到$T(x; \theta_m)$。更详细一点，得到得到第 m 棵树的叶结点区域$R_{mj}, j = 1,2 \ldots J$,即一颗由 J 个叶子结点组成的树。

(c) 计算每一个区域的最优输出：

$$c_{mj} = argmin(\sum_{x_i \epsilon R_{mj}} L(y_i, f_{m-1}(x_i) + c)$$

上面的（b）和（c）两步相当于 CART 回归树算法中寻找最优的切分变量 j 和最优分割点 s，然后在每个节点区域寻找最优的输出 c。

（d）更新$f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J} c_{mj} I(x_i \epsilon R_m$ ）

(3) 得到回归问题梯度提升树

$$f_M(x) = \sum_{m=1}^{M} f_m(x) = \sum_{m=1}^{M} \sum_{j=1}^{J} c_{mj} I(x_i \epsilon R_{mj})$$

算法 1 步获得使得损失函数最小的常数估计值，是一个只有根节点的树。在 2.a 步计算**损失函数的负梯度在当前模型的值**，将它作为残差估计。在 2.b 步估计回归树的叶结点区域，来拟合残差的近似值。在 2.c 步利用线性搜索估计回归树叶结点区域的值，使损失函数最小化。2.d 更新回归树。第 3 步获得输出的最终模型。

5、XGBoost 模型

5.1 XGBoost 模型基本原理

5.1.1 基本推导

XGBoost 模型的损失函数包含两部分，模型的经验损失以及正则化项：

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

对目标函数进行一个改写：

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i{}^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

这里**主要用的思想是**二阶泰勒展开：

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

这里我们定义每个数据点的在误差函数上的一阶导数和二阶导数：

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

此时，我们的目标函数可以写作：

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n}[l(y_i,\hat{y}^{(t-1)})+g_i f_t(\mathbf{x}_i)+\frac{1}{2}h_i f_t^2(\mathbf{x}_i)]+\Omega(f_t)$$

如果我们的损失定义为平方损失的话：

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

其实这里也能看到我们**为什么只用到了二阶泰勒展开**，因为在平方损失的时候，三阶展开已经是 0 了。

进一步，对 XGboost 来说，每一个数据点x_i最终会落到一个叶子结点上，而对于落在同一个叶子结点上的数据点来说，其输出都是一样的，假设我们共有 J 个叶子结点，每个叶子结点对应的输出为w_j(w_j也是我们想要求解最优权重)，则目标函数可以进一步改写（第一项为常数项，可以忽略）：

$$\begin{aligned}\tilde{\mathcal{L}}^{(t)} &= \sum_{i=1}^{n}[g_i f_t(\mathbf{x}_i)+\frac{1}{2}h_i f_t^2(\mathbf{x}_i)]+\gamma T+\frac{1}{2}\lambda\sum_{j=1}^{T}w_j^2\\ &= \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)w_j+\frac{1}{2}(\sum_{i\in I_j} h_i+\lambda)w_j^2]+\gamma T\end{aligned}$$

此时，我们就可以进行求解了，最优权重为：

$$w_j^*=-\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i+\lambda},$$

将权重带回我们的目标函数，目标函数变为：

$$\tilde{\mathcal{L}}^{(t)}(q)=-\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i+\lambda}+\gamma T.$$

有了目标函数，我们可以得到一个简单的决定是否可以进行分裂的判断标准，即分裂之后两边的目标函数之和是否能够大于不分裂的目标函数值：

$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{7}$$

5.1.2 Shrinkage and 采样

除了以上提到了正则项以外，我们还有 shrinkage **与采样技术来避免过拟合**的出现。所谓 shrinkage 就是在每个迭代中树中，对叶子结点乘以一个缩减权重 eta。该操作的作用就是减少每颗树的影响力，留更多的空间给后来的树提升。

另一个技术则是采样的技术，有两种，**一种是列采样和一种是行采样**。其中列采样的实现一般有两种方式，一种是按层随机（一般来讲这种效果会好一点），另一种是建树前就随机选择特征。

按层随机的意思就是，上文提到每次分裂一个结点的时候，我们都要遍历所有的特征和分割点，从而确定最优的分割点，那么如果加入了列采样，我们会在对同一层内每个结点分裂之前，先随机选择一部分特征，于是我们只需要遍历这部分的特征，来确定最优的分割点。建树前随机选择特征就表示我们在建树前就随机选择一部分特征，然后之后所有叶子结点的分裂都只使用这部分特征。

而行采样则是 bagging 的思想，每次只抽取部分的样本进行训练，而不使用全部的样本，从而增加树的多样性。

5.2 分裂算法

5.2.1 Basic Exact Greedy Algorithm

最基本的算法及遍历每个特征及分割点，选择目标函数增加最多的特征和分割点的组合：

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

5.2.2 Approximate Algorithm

精确通过列举特征所有可能的划分，耗时，当数据量大的时候，几乎不可能将数据全部加载进内存，精确划分在分布式中也会有问题。因此文章中提出了近似的策略。近似方法通过特征的分布，按照百分比确定一组候选分裂点，通过遍历所有的候选分裂点来找到最佳分裂点，方法过程如下图所示：

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.

近似方式有两种策略：一种是在建立第 k 棵树的时候利用样本的二阶梯度对样本进行离散化，每一维的特征都建立 buckets。在建树的过程中，我们就重复利用这些 buckets 去做。另一种是每次进行分支时，我们都重新计算每个样本

的二阶梯度并重新构建 buckets，再进行分支判断。前者我们称之为全局选择，后者称为局部选择。显然局部选择的编码复杂度更高，但是实验当中效果极其的好，甚至与 Exact Greedy Algorithm 一样。

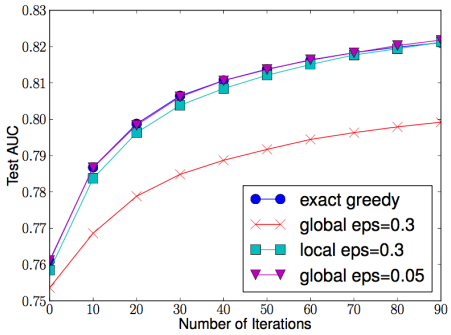


Figure 3: Comparison of test AUC convergence on Higgs 10M dataset. The eps parameter corresponds to the accuracy of the approximate sketch. This roughly translates to 1 / eps buckets in the proposal. We find that local proposals require fewer buckets, because it refine split candidates.

5.2.3 Weighted Quantile Sketch

关于上面的式子，我们还有一点没有讲到，我们怎么来选择$S_k = \{s_{k1}, s_{k2}..s_{kl}\}$。即我们怎么限定近似算法的边界？如果我们用 k 分位点来选取的话，这样就是 Quantile Sketch 的思想。但是实际中，我们要均分的是 loss，而不是样本的数量，而每个样本对 loss 的贡献可能是不一样的，按样本均分会导致 loss 分布不均匀，取到的分位点会有偏差。怎么衡量每个样本对 loss 的贡献呢，文章中选取的是每个样本点的二阶梯度 h，我们定义一个 rank 函数：

$$r_k(z) = \frac{1}{\sum_{(x,h)\in\mathcal{D}_k} h} \sum_{(x,h)\in\mathcal{D}_k, x<z} h,$$

rank 的计算是对某一个特征上，样本特征值小于 z 的二阶梯度除以所有的二阶梯度总和。其实就是对样本的二阶梯度进行累加求和，随后，我们选择的分割点要满足下面的式子：

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon, \quad s_{k1} = \min_i \mathbf{x}_{ik}, s_{kl} = \max_i \mathbf{x}_{ik}.$$

上面的ϵ用于设定最大的 rank 值，其中 eps 在(0, 1)的区间上。我们可以大致的认为有 $1/\epsilon$个 buckets。而我们对这么多个 buckets 进行分支判断。显然,比起对 m 个样本找分裂节点，对 $1/\epsilon$个 buckets 找分裂节点更快捷，而且ϵ越大 buckets 数量越少，粒度越粗。

还有一点提到的就是，这里我们在进行最优分割点选择的时候，会首先对特征的每一个值进行 Pre-sort。

5.2.4 Sparsity-aware Split Finding

在实际应用中，稀疏数据是不可避免的，造成稀疏数据的原因主要有：数据缺失、统计上的 0 以及特征表示中的 one-hot 形式，以往经验表明当出现稀疏、缺失值时时，算法需要很好的稀疏自适应。因此本文针对稀疏数据，提出了相应的分割算法。**其基本思想是将缺失值捆绑起来处理，要么都放在左节点，要么都放在右节点。随后看放在左边还是放在右边的收益更高。**

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

5.3 XGBoost 与 GBDT 的异同

这一部分来自于知乎，参考文献[6]，很全面的总结了 GBDT 和 XGBoost 算法的异同。

- 传统 GBDT 以 CART 作为基分类器，xgboost 还支持线性分类器，这个时候 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）。
- 传统 GBDT 在优化时只用到一阶导数信息，xgboost 则对代价函数进行了二阶泰勒展开，同时用到了一阶和二阶导数。顺便提一下，xgboost 工具支持自定义代价函数，只要函数可一阶和二阶求导。
- xgboost 在代价函数里加入了正则项，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、每个叶子节点上输出的 score 的 L2 模的平方和。从 Bias-variance tradeoff 角度来讲，正则项降低了模型的 variance，使学习出来的模型更加简单，防止过拟合，这也是 xgboost 优于传统 GBDT 的一个特性。
- Shrinkage（缩减），相当于学习速率（xgboost 中的 eta）。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。实际应用中，一般把 eta 设置得小一点，然后迭代次数设置得大一点。（补充：传统 GBDT 的实现也有学习速率）
- 列抽样（column subsampling）。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，这也是 xgboost 异于传统 gbdt 的一个特性。

- 对缺失值的处理。对于特征的值有缺失的样本，xgboost 可以自动学习出它的分裂方向。
- xgboost 工具支持并行。boosting 不是一种串行的结构吗?怎么并行的?注意 xgboost 的并行不是 tree 粒度的并行，xgboost 也是一次迭代完才能进行下一次迭代的（第 t 次迭代的代价函数里包含了前面 t-1 次迭代的预测值）。xgboost 的并行是在特征粒度上的。我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），xgboost 在训练之前，预先对数据进行了排序，然后保存为 block 结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个

block 结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行。

- 可并行的近似直方图算法。树节点在进行分裂时，我们需要计算每个特征的每个分割点对应的增益，即用贪心法枚举所有可能的分割点。当数据无法一次载入内存或者在分布式情况下，贪心算法效率就会变得很低，所以 xgboost 还提出了一种可并行的近似直方图算法，用于高效地生成候选的分割点。

6、LightGBM

6.1 LightGBM 基本原理

LightGBM 是去年微软开源的一套框架，算是对 GBDT 的一种改进吧。对 GBDT 来说，**对于每一个特征的每一个分裂点，都需要遍历全部数据来计算信息增益。**因此，其计算复杂度将受到特征数量和数据量双重影响，造成处理大数据时十分耗时。解决这个问题的直接方法就是减少特征量和数据量而且不影响精确度，有部分工作根据数据权重采样来加速 boosting 的过程，但由于 GBDT 没有样本权重不能应用。LightGBM 文章中提出两种新方法实现此目标。

6.1.1 Gradient-based One-Side Sampling (GOSS)

AdaBoost 中，样本权重是数据实例重要性的指标。然而在 GBDT 中没有原始样本权重，不能应用权重采样。但是，GBDT 中每个数据都有不同的梯度值，对采样十分有用，即实例的梯度小，实例训练误差也就较小，已经被学习得很好了，直接想法就是丢掉这部分梯度小的数据。然而这样做会改变数据的分布，将会影响训练的模型的精确度，为了避免此问题，提出了 GOSS。

GOSS 保留所有的梯度较大的实例，在梯度小的实例上使用随机采样。为了抵消对数据分布的影响，计算信息增益的时候，GOSS 对小梯度的数据引入常量乘数。GOSS 首先根据数据的梯度绝对值排序，选取 top a 个实例。然后在剩余的数据中随机采样 b 个实例。接着计算信息增益时为采样出的小梯度数据乘以(1-a)/b，这样算法就会更关注训练不足的实例，而不会过多改变原数据集的分布。

6.1.2 Exclusive Feature Bundling (EFB)

这块其实不是很懂。。

高维度的数据通常是非常稀疏的，这样的特性为特征的相互之间结合提供了便利。通过将一些特征进行融合绑定在一起，可以使特征数量大大减少，这样在进行histogram building时时间复杂度从O(#data * #feature)变为O(#data * #bundle)，这里#bundle远小于#feature。

那么将哪些特征绑定在一起呢？又要怎么绑定呢？

对于第一个问题，将相互独立的特征进行bunding是一个NP难问题，lightGBM将这个问题转化为图着色的为题来求解，将所有特征视为图G的一个顶点，将不是相互独立的特征用一条边连接起来，边的权重就是两个相连接的特征的总冲突值，这样需要绑定的特征就是在图着色问题中要涂上同一种颜色的那些点（特征）。

对于第二个问题，绑定几个特征在同一个bundle里需要保证绑定前的原始特征的值可以在bundle中识别，考虑到 histogram-based算法将连续的值保存为离散的bins，我们可以使得不同特征的值分到bundle中的不同bins中，这可以通过在特征值中加一个偏置常量来解决，比如，我们在bundle中绑定了两个特征A和B，A特征的原始取值为区间[0,10)，B特征的原始取值为区间[0,20），我们可以在B特征的取值上加一个偏置常量10，将其取值范围变为[10,30），这样就可以放心的融合特征A和B了，因为在树模型中对于每一个特征都会计算分裂节点的，也就是通过将他们的取值范围限定在不同的bins中，在分裂时可以将不同特征很好的分裂到树的不同分支中去。

Algorithm 3: Greedy Bundling

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

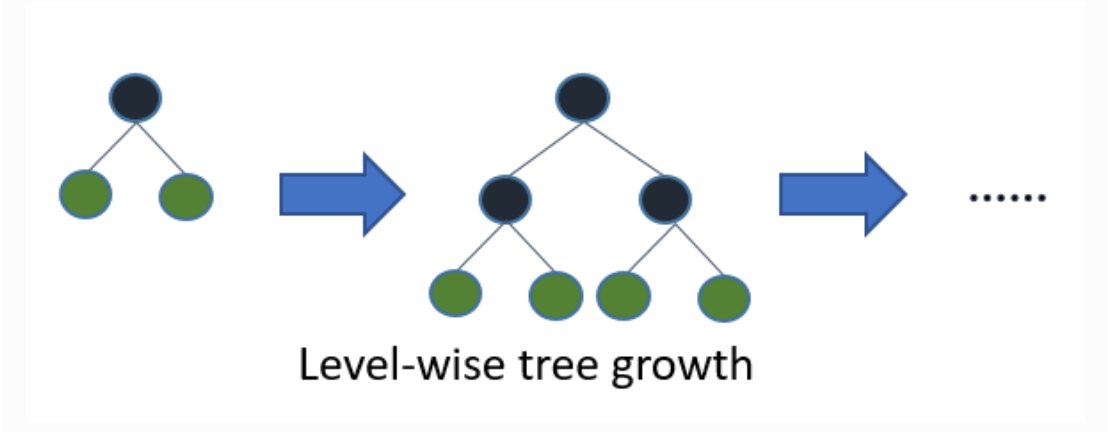
Algorithm 4: Merge Exclusive Features

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

6.2 LightGBM 的其他特性

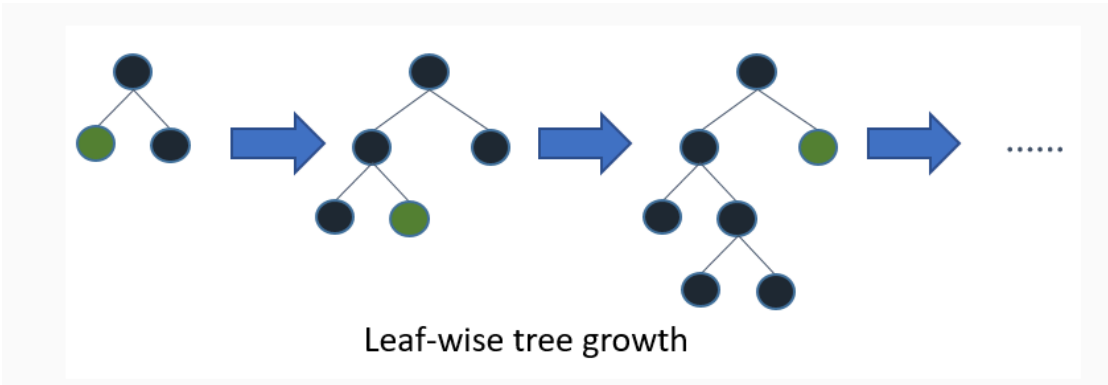
6.2.1 Level-wise VS Leaf-wise

大部分决策树（包括 XGBoost）的学习算法通过 level(depth)-wise 策略生长树，如下图一样：



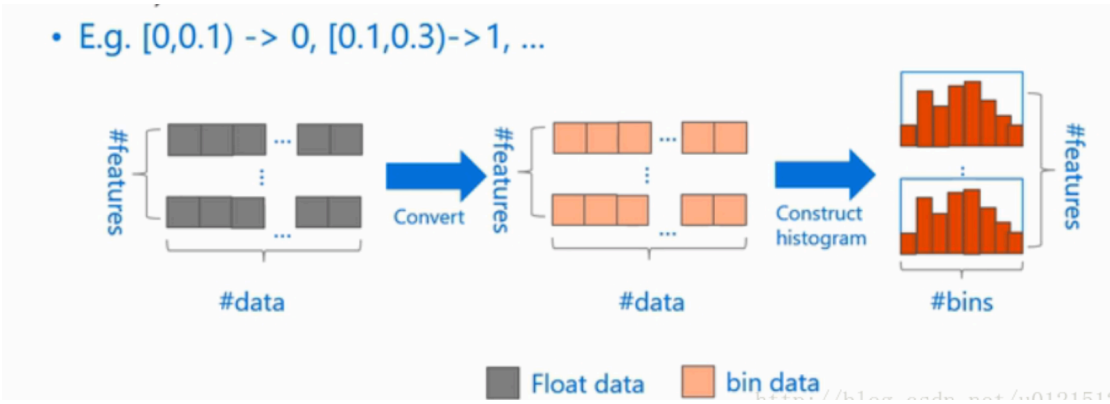
LightGBM 通过 leaf-wise (best-first)策略来生长树。它将选取具有最大 delta loss 的叶节点来生长。 当生长相同的叶子结点时，leaf-wise 算法可以比 level-wise 算法减少更多的损失。

当 data 较小的时候，leaf-wise 可能会造成过拟合。 所以，LightGBM 可以利用限制树的深度并避免过拟合（树的生长仍然通过 leaf-wise 策略）。



6.2.2 直方图优化

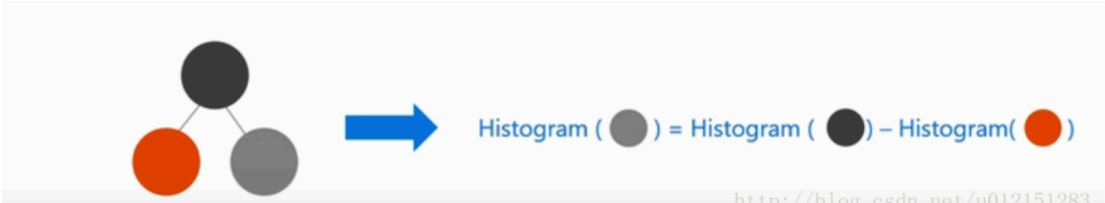
LightGBM 采用了直方图算法寻找最优的分割点，将连续值特征离散化，转换到不同的 bin 中，过程如下图所示：



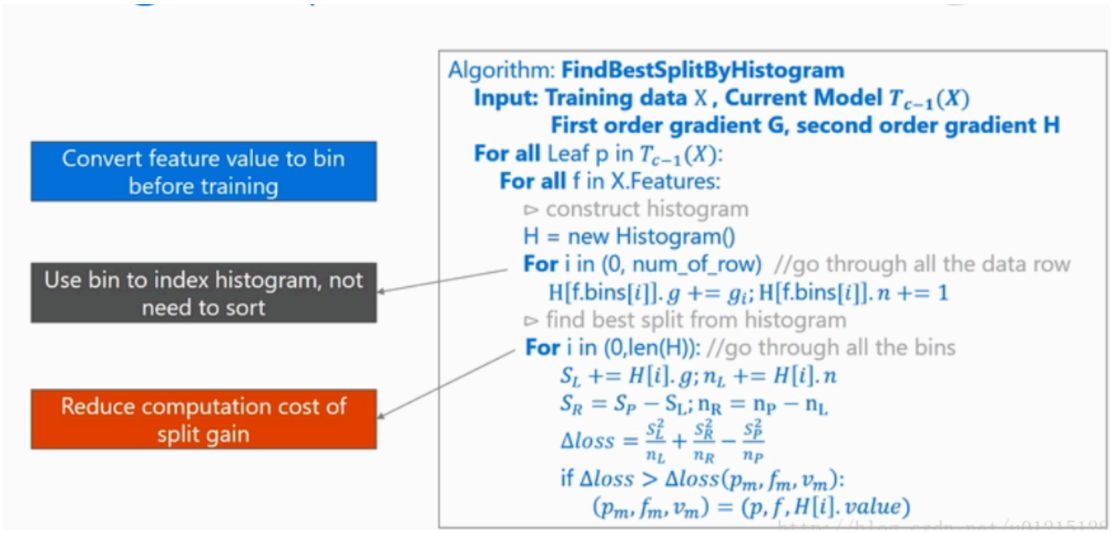
在转换到 bin 之后，还是根据和 XGBoost 一样的式子寻找最优分裂点，再回顾一下：

$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{7}$$

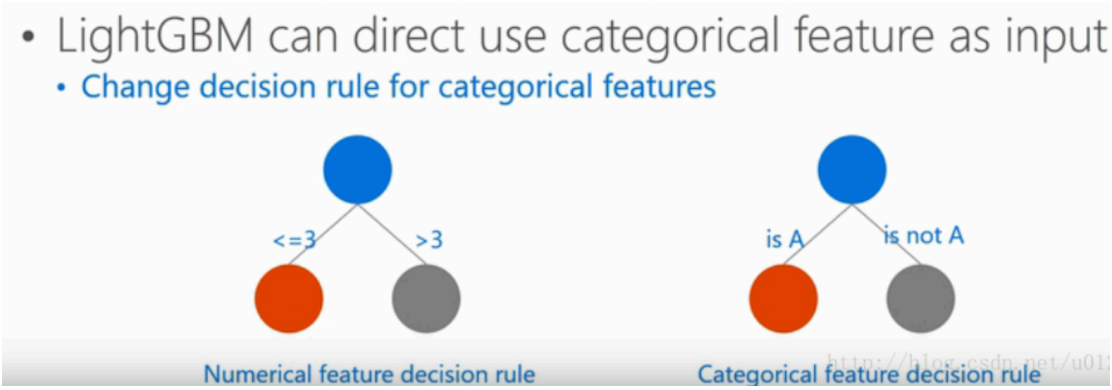
不过在分裂过程中，LightGBM 用了一个小 trick，叫做**直方图做差**。父节点的直方图等于其左右节点的直方图的累加值，因此在实际中，统计好父节点的直方图后，我们只需要在统计数据量较少的一个子节点的直方图，数据量较大的节点的直方图可以通过做差得到，这样又将效率提升了一倍：



综上所述，LightGBM 中用到的最优分裂点算法过程如下：



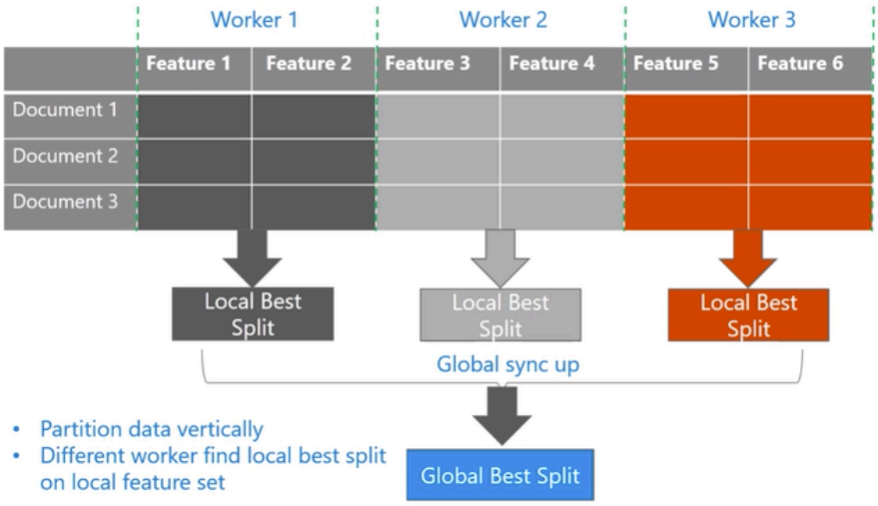
同时，对于离散特征，LightGBM 也可以很好的处理：



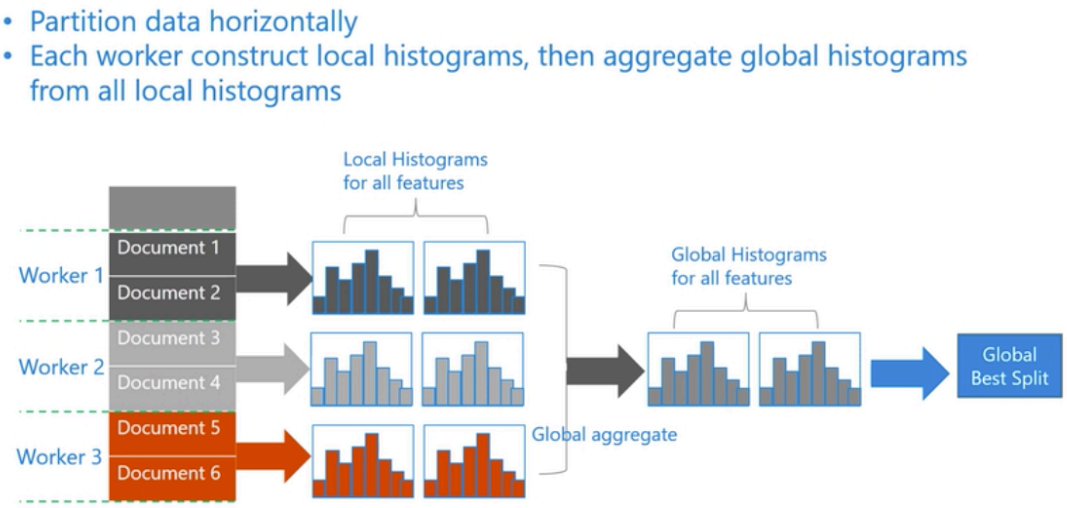
6.2.3 并行策略

XGBoost 主要是特征粒度的并行，而 LightGBM 的并行体现在三个方面，分别是**特征并行，数据并行，投票并行**。但这三种并不是同时存在的，针对不同的数据集会选择不同的并行方式。

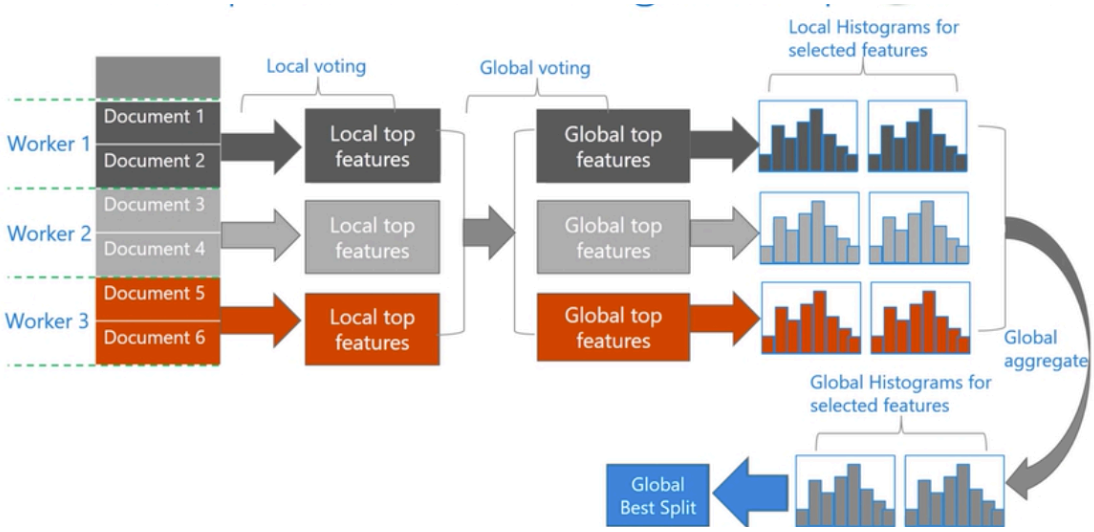
对于特征并行来说，适用于数据少，但是特征比较多的情况。LightGBM 的每个 worker 保存所有的数据集，并在其特征子集上寻找最优切分点。随后，worker 之间互相通信，找到全局最佳切分点。最后每个 worker 都在全局最佳切分点进行节点分裂。这样做的好处主要是减小网络通信量，但是当数据量比较大的时候，由于每个 worker 都存储的是全量数据，因此数据存储代价高。



数据并行适用于数据量大，特征比较少的情况。数据并行对数据进行水平切分，每个 worker 上的数据先建立起局部的直方图，然后合并成全局的直方图，子后找到最优的分割点。



投票并行适用于数据量大，特征也比较大的情况。投票并行主要是对数据并行的一个优化。数据并行中合并直方图的时候通信代价比较大。给予投票的方式，我们只会合并部分的特征的直方图。具体来说，每个 worker 首先会找到它们数据中最优的一些特征，然后进行全局的投票，根据投票的结果，选择 top 的特征的直方图进行合并，在寻求全局的最优分割点。



6.3 LightGBM 与 XGBoost 的不同点

参考文献[10],总结下面几**点不同**：

1、由于在决策树在每一次选择节点特征的过程中，要遍历所有的属性的所有取值并选择一个较好的。XGBoost 使用的是近似算法算法，先对特征值进行排序 Pre-sort，然后根据二阶梯度进行分桶，能够更精确的找到数据分隔点；但是复杂度较高。LightGBM 使用的是 histogram 算法，这种只需要将数据分割成不

同的段即可，不需要进行预先的排序。占用的内存更低，数据分隔的复杂度更低。

2、决策树生长策略，我们刚才介绍过了，XGBoost 采用的是 Level-wise 的树生长策略，LightGBM 采用的是 leaf-wise 的生长策略。

3、并行策略对比，XGBoost 的并行主要集中在特征并行上，而 LightGBM 的并行策略分特征并行，数据并行以及投票并行。

参考文献

1、李航《统计学习方法》

2、『机器学习笔记 』GBDT 原理-Gradient Boosting Decision Tree：https://blog.csdn.net/shine19930820/article/details/65633436

3、简单易学的机器学习算法——梯度提升决策树 GBDT：https://blog.csdn.net/google19890102/article/details/51746402/

4、xgboost 中的数学原理：https://blog.csdn.net/a358463121/article/details/68617389

5、通俗、有逻辑的写一篇说下 Xgboost 的原理，供讨论参考：https://blog.csdn.net/github_38414650/article/details/76061893

6、机器学习算法中 GBDT 和 XGBOOST 的区别有哪些？https://www.zhihu.com/question/41354392

7、LightGBM 视频：https://v.qq.com/x/page/k0362z6lqix.html?

8、快的不要不要的 lightGBM：https://zhuanlan.zhihu.com/p/31986189

9、XGBoost all in one：http://www.pengfoo.com/post/machine-learning/2017-03-03

10、 机器学习实践-XGboost 与 LightGBM 对比:https://jiayi797.github.io/2018/03/06/机器学习实践-XGboost 与 LightGBM 对比

11、 XGBoost 论文：http://www.kdd.org/kdd2016/papers/files/rfp0697-chenAemb.pdf

12、 LightGBM 论文：https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree