1.Xgboost? 这个故事还得先从AdaBoost和GBDT说起

AdaBoost，是英文"Adaptive Boosting"（自适应增强），**它的自适应在于：前一个基本分类器分错的样本会得到加强，加权后的全体样本再次被用来训练下一个基本分类器。同时，在每一轮中加入一个新的弱分类器，直到达到某个预定的足够小的错误率或达到预先指定的最大迭代次数**。白话的讲，就是它在训练弱分类器之前，会给每个样本一个权重，训练完了一个分类器，就会调整样本的权重，前一个分类器分错的样本权重会加大，这样后面再训练分类器的时候，就会更加注重前面分错的样本， 然后一步一步的训练出很多个弱分类器，最后，根据弱分类器的表现给它们加上权重，组合成一个强大的分类器，就足可以应付整个数据集了。 这就是AdaBoost， 它强调自适应，不断修改样本权重， 不断加入弱分类器进行boosting。

那么，boosting还有没有别的方式呢？GBDT(Gradient Boost Decision Tree)就是另一种boosting的方式， 上面说到**AdaBoost训练弱分类器关注的是那些被分错的样本，AdaBoost每一次训练都是为了减少错误分类的样本。而GBDT训练弱分类器关注的是残差，也就是上一个弱分类器的表现与完美答案之间的差距，GBDT每一次训练分类器，都是为了减少这个差距**，GBDT每一次的计算是都为了减少上一次的残差，进而在残差减少（负梯度）的方向上建立一个新的模型。
事实上，如果不考虑工程实现、解决问题上的一些差异，xgboost与gbdt比较大的不同就是目标函数的定义，但这俩在策略上是类似的，都是聚焦残差（更准确的说，xgboost其实是gbdt算法在工程上的一种实现方式），GBDT旨在通过不断加入新的树最快速度降低残差，而XGBoost则可以人为定义损失函数（可以是最小平方差、logistic loss function、hinge loss function或者人为定义的loss function），**只需要知道该loss function对参数的一阶、二阶导数便可以进行boosting**，其进一步增大了模型的泛化能力，其贪婪法寻找添加树的结构以及loss function中的损失函数与正则项等一系列策略也使得XGBoost预测更准确。

AdaBoost、GBDT、XGBoost这三个同是属于集成学习的boosting流派，AdaBoost叫做自适应提升，和GBDT，Xgboost提升时采用的策略不同，**前者聚焦错误样本，后者聚焦与标准答案的残差**。而GBDT和Xgboost叫做boosting集成学习，提升时策略类似，都是聚焦残差，但是降低残差的方式又各有不同。

2.XGBoost的基本原理

如果boosting算法每一步的弱分类器生成都是依据损失函数的梯度方向，则称之为梯度提升(Gradient boosting)，XGBoost算法是采用分步前向加性模型，只不过在每次迭代中生成弱学习器后不再需要计算一个系数，XGBoost 是由 k 个基模型组成的一个加法运算式：

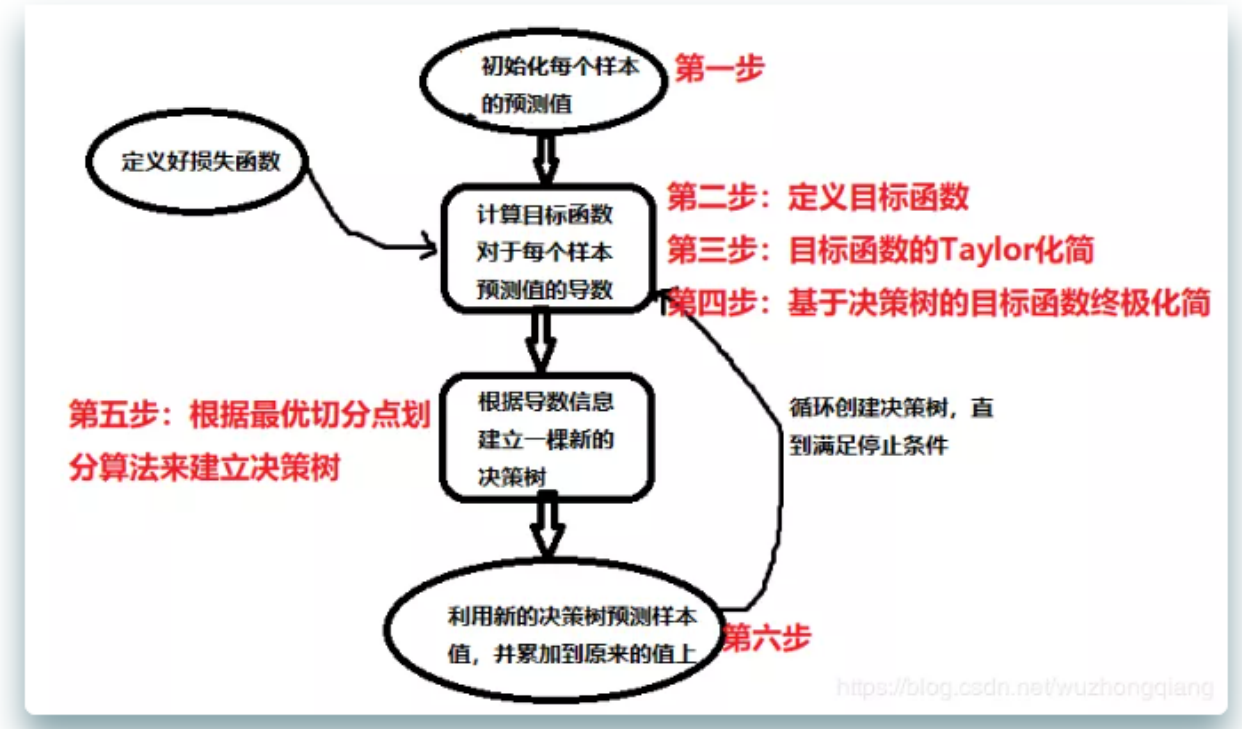
$$\hat{y}_i = \sum_{t=1}^{k} f_t(x_i)$$

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

XGBoost的基本流程



XGB的目标函数

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

这个公式应该不需要过多的解释了吧， 其中$\Omega(f_t)$是正则化项

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

前面的T_t为叶子节点数， w_j表示j叶子上的节点权重， γ， λ是预先给定的超参数。引入了正则化之后，算法会选择简单而性能优良的模型， 正则化项只是用来在每次迭代中抑制弱分类器$f_i(x)$过拟合，不参与最终模型的集成。（这个正则化项可以先不用管它， 有个印象即可，后面树那个地方会统一解释）

我们知道， boosting模型是前向加法， 以第t步模型为例， 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，$\hat{y}_i^{t-1}$是第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，所以把这个$\hat{y}_i^t$代入上面，就可以进一步化简：

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

这个就是xgboost的目标函数了，最优化这个目标函数，其实就是相当于求解当前的$f_t(x_i)$。**Xgboost系统的每次迭代都会构建一颗新的决策树，决策树通过与真实值之间残差来构建**。什么，没有看到残差的身影？别急，后面会看到这个残差长什么样子。

通过泰勒展开式化简目标函数

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

我们看看这里的$l\left(y_i, \hat{y}_i^{t-1} + f_t(x_i)\right)$，这个部分，如果结合伟大的Taylor的话，会发生什么情况，你还记得泰勒吗？

> 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。
>
> 在这里插入图片描述

根据Taylor公式，我们把函数$f(x + \Delta(x))$在点x处二阶展开，可得到：

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

那么我们把$l\left(y_i, \hat{y}_i^{t-1} + f_t(x_i)\right)$中的$\hat{y}_i^{t-1}$视为$x$，$f_t(x_i)$视为$\Delta(x)$，那么目标函数就可以写成：

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

其中g_i是损失函数l的一阶导数，h_i是损失函数l的二阶导，注意这里的求导是对$\hat{y}_i^{t-1}$求导。

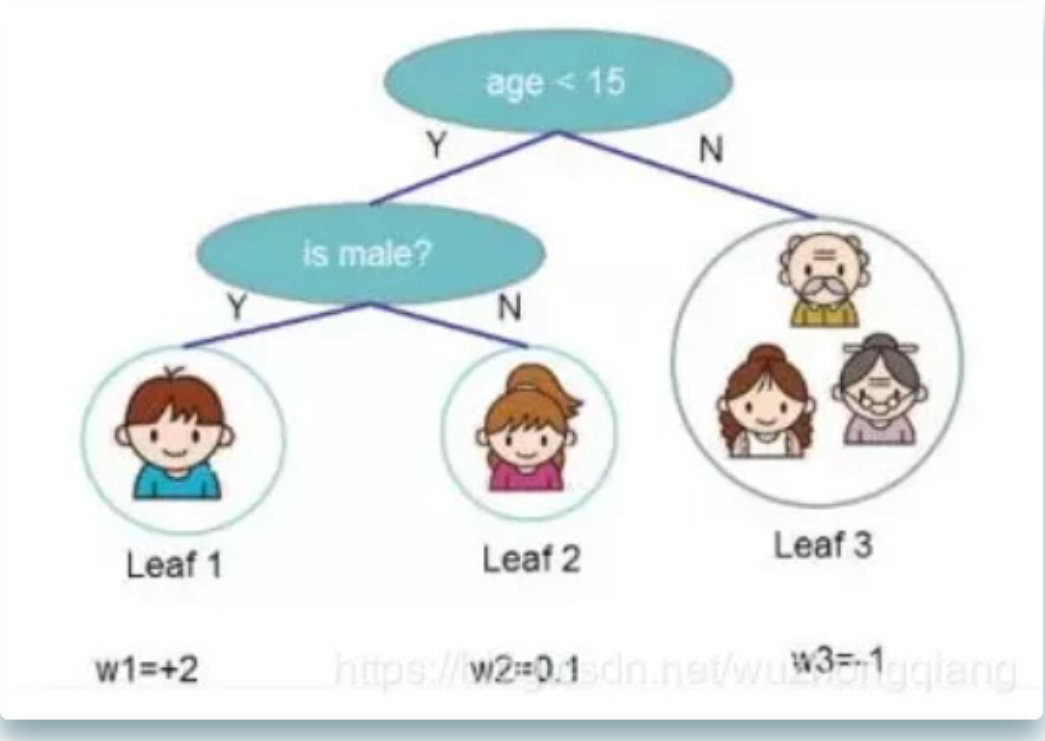
由于在第t步时$\hat{y}_i^{t-1}$是一个已知的值，所以$l\left(y_i, \hat{y}_i^{t-1}\right)$是一个常数，其对函数的优化不会产生影响，因此目标函数可以进一步写成：

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

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$\hat{y}_i^{t-1}$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。
但是还有个问题，就是我们如果是建立决策树的话，根据上面的可是无法建立出一棵树来。因为这里的$f_t(x_i)$是什么鬼？ 咱不知道啊！所以啊，还得进行一步映射， 将样本x映射到一个相对应的叶子节点才可以，看看是怎么做的？

基于决策树的目标函数的终极化简

我们这里先解决一下这个$f_t(x_i)$的问题，这个究竟怎么在决策树里面表示呢？那解决这个问题之前，我们看看这个$f_t(x_i)$表示的含义是什么， f_t就是我有一个决策树模型， x_i是每一个训练样本，那么这个整体$f_t(x_i)$就是我某一个样本x_i经过决策树模型f_t得到的一个预测值，对吧？那么，我如果是在决策树上，可以这么想， 我的决策树就是这里的f_t， 然后对于每一个样本x_i，**我要在决策树上遍历获得预测值，其实就是在遍历决策树的叶子节点，因为每个样本最终通过决策树都到了叶子上去**， 不信？看下图（样本都在叶子上， 只不过这里要注意一个叶子上不一定只有一个样本）：



所以，**通过决策树遍历样本，其实就是在遍历叶子节点**。这样我们就可以把问题就行转换，把决策树模型定义成$f_t(x) = w_{q(x)}$，其中$q(x)$代表了该样本在哪个叶子节点上， w表示该叶子节点上的权重（上面分数预测里面+90， +60就是叶子节点的权重）。所以$w_{q(x)}$就代表了每个样本的取值（预测值）。 那么这个样本的遍历，就可以这样化简：

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

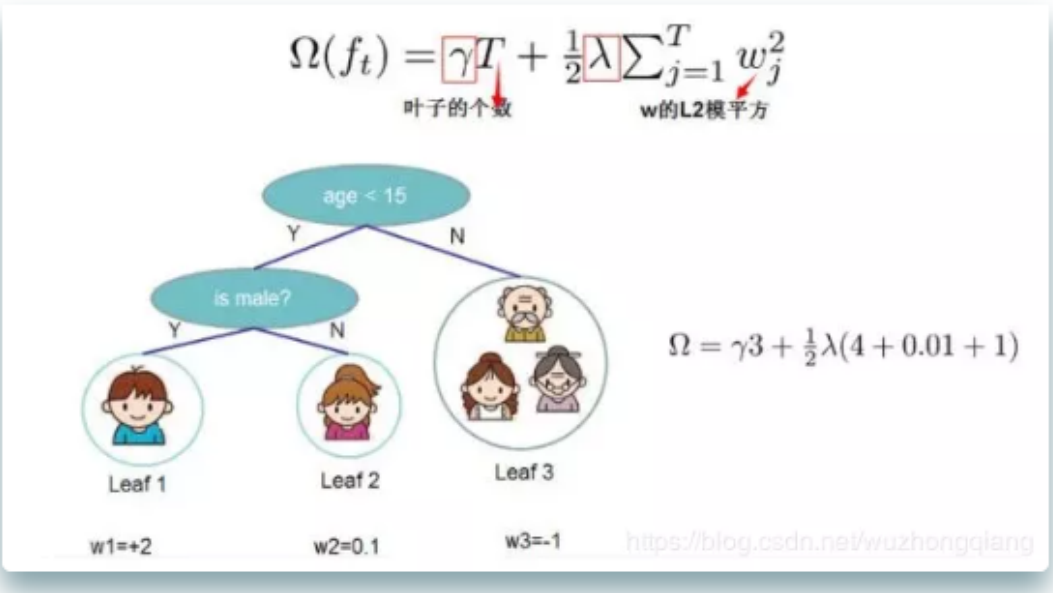
$$\sum_{i=1}^{n}\left[g_i w_{q(x_i)}+\frac{1}{2}h_i w_{q(x_i)}^2\right]=\sum_{j=1}^{T}\left[\left(\sum_{i\in I_j}g_i\right)w_j+\frac{1}{2}\left(\sum_{i\in I_j}h_i\right)w_j^2\right]$$

这个再解释一遍就是：**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（注意第二个等式和第三个等式求和符号的上下标， T代表叶子总个数）。**
由于一个叶子节点有多个样本存在，所以后面有了$\sum_{i\in I_j}g_i$和$\sum_{i\in I_j}h_i$这两项，这里的I_j它代表一个集合，集合中每个值代表一个训练样本的序号，整个集合就是某棵树第j个叶子节点上的训练样本 ，w_j为第j个叶子节点的取值。只要明白了这一步，后面的公式就很容易理解了。

我们再解决一下后面那部分，在决策树中，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(γ权衡)，和所有节点权重(λ权衡)所组成的向量的 范式共同决定。



这张图给出了基于决策树的 XGBoost 的正则项的求解方式。

这样，目标函数的前后两部分都进行了解决，那么目标函数就可以化成最后这个样子，看看能懂吗？

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

这里的$I_j = \{i|q(x_i) = j\}$为第j个叶子节点的样本集合。为了简化表达式，我们再定义：$G_j = \sum_{i \in I_j} g_i \quad H_j = \sum_{i \in I_j} h_i$，那么决策树版本xgboost的目标函数：

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

这里要注意G_j和H_j是前t-1步得到的结果，其值已知，只有最后一棵树的叶子节点w_j的值不确定，那么将目标函数对w_j求一阶导，并令其等于0，$\frac{\partial J(f_t)}{\partial w_j} = G_j + (H_j + \lambda) w_j = 0$，则可以求得叶子节点j对应的权值：

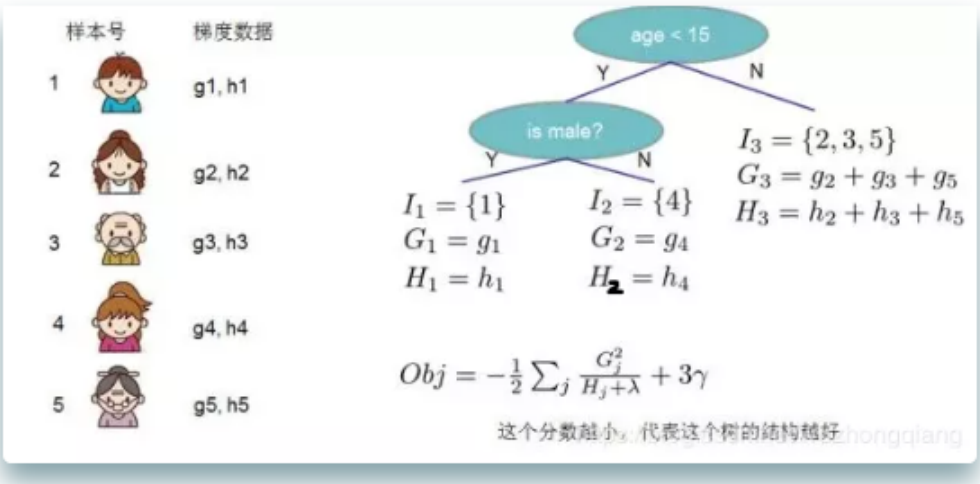
$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

那么这个目标函数又可以进行化简：

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

这个就是基于决策树的xgboost模型的目标函数最终版本了，这里的G和H的求法，就需要明确的给出损失函数来，然后求一阶导和二阶导，然后代入样本值即得出。

这个obj代表了当我们指定一个树的结构的时候，我们在目标上最多能够减少多少，我们之前不是说建立一个树就是让残差尽可能的小吗？到底小多少呢？这个obj就是衡量这个的，可以叫做结构分数。就类似于基尼系数那样对树结构打分的一个函数。那么这个分数怎么算呢？看下面的例子：



还是上面的那个预测玩游戏的意愿，我们假设建了右边的那棵树，那么每个样本都对应到了叶子节点上去，每一个样本都会对应一个g和h， 那么我们遍历叶子节点，就会得到G和H，然后累加就可以得到这棵树的结构分数obj（这里有个小细节就是假设有N个训练样本， 那么就会有N次计算各自的gi 和 hi，但是由于每个样本的gi 和 hi没有啥关系，所以可以并行计算，这样就可以加速训练了，而且，gi 和 hi是不依赖于损失函数的形式的，只要这个损失函数二次可微就可以了，emmm...powerful）。

有了这个，我们就知道这棵树建的好不好了。

最优切分点划分算法及优化策略

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：

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

分裂后的目标函数：

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

则对于目标函数来说，分裂后的收益为（Obj1-Obj2）：

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

注意该特征收益也可作为特征重要性输出的重要依据。

那么我们就可以来梳理一下最优切分点的划分算法了：

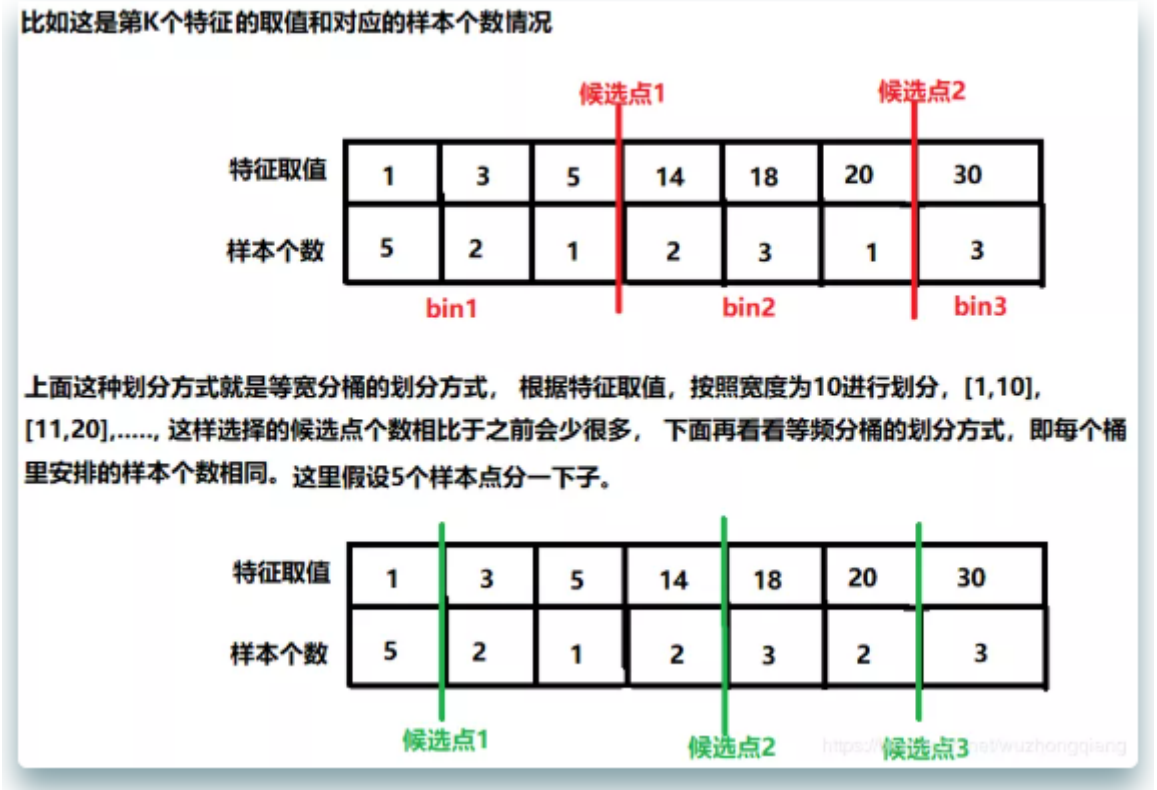
对每个特征进行排序，再寻找该特征的最佳分裂点

- 从深度为 0 的树开始，对每个叶节点枚举所有的可用特征；
- 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；（这个过程每个特征的收益计算是可以并行计算的，xgboost之所以快，其中一个原因就是因为它支持并行计算，而这里的并行正是指的特征之间的并行计算，千万不要理解成各个模型之间的并行）
- 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（这里稍微提一下，xgboost是可以处理空值的，也就是假如某个样本在这个最优分裂点上值为空的时候， 那么xgboost先把它放到左子树上计算一下收益，再放到右子树上计算收益，哪个大就把它放到哪棵树上。）
- 回到第 1 步，递归执行到满足特定条件为止

遍历完所有特征后，我们就可以确定应该在哪个特征的哪个点进行切分。对切分出来的两个节点，递归地调用这个过程，我们就能获得一个相对较好的树结构，有了树结构就比较容易找最优的叶子节点，这样就能对上面的样本进行预测了。当然，特征与特征之间的收益计算是互不影响的，所以这个遍历特征的过程其实可以并行运行。

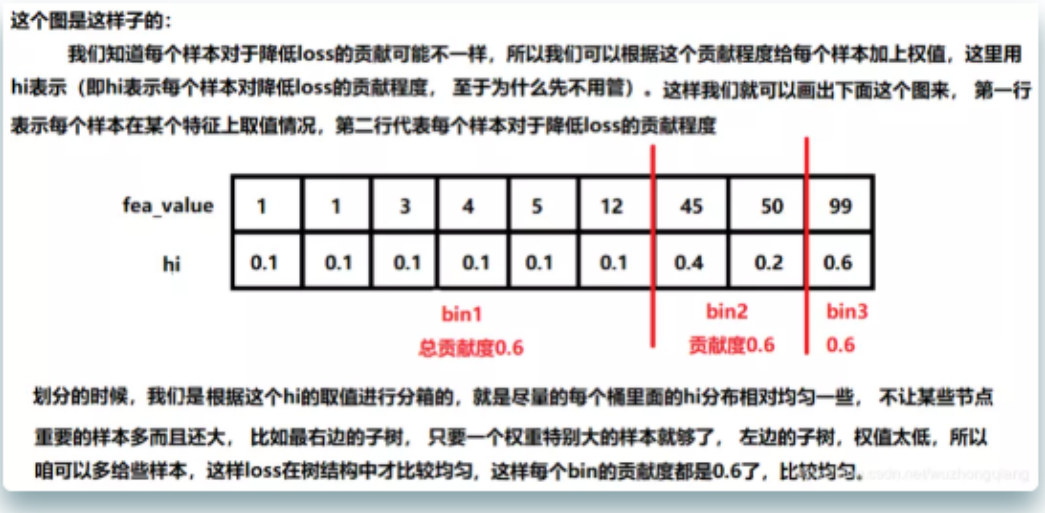
在这个过程中你是否注意到了一个问题， 就是xgboost的切分操作和普通的决策树切分过程是不一样的。普通的决策树在切分的时候并不考虑树的复杂度，所以才有了后续的剪枝操作。而xgboost在切分的时候就已经考虑了树的复杂度（obj里面看到那个γ了吗）。所以，它不需要进行单独的剪枝操作。

这就是xgboost贪心建树的一个思路了，即**遍历所有特征以及所有分割点，每次选最好的那个**。GBDT也是采用的这种方式， 这算法的确不错，但是有个问题你发现了没？就是计算代价太大了，尤其是数据量很大，分割点很多的时候，计算起来非常复杂并且也无法读入内存进行计算。所以作者想到了一种近似分割的方式（可以理解为分割点分桶的思路），选出一些候选的分裂点，然后再遍历这些较少的分裂点来找到最佳分裂点。那么怎么进行分桶选候选分裂点才比较合理呢？我们一般的思路可能是根据特征值的大小直接进行等宽或者等频分桶，像下面这样（这个地方理解起来有点难，得画画了，图可能不太好看，能说明问题就行，哈哈）：



上面就是等频和等宽分桶的思路了（这个不用较真，我这里只是为了和作者的想法产生更清晰的对比才这样举得例子），这样选择出的候选点是不是比就少了好多了？但是这样划分其实是有问题的，因为这样划分没有啥依据啊， 比如我上面画的等频分桶，我是5个训练样本放一个桶，但是你说你还想10个一组来，没有个标准啥的啊。即上面那两种常规的划分方式缺乏可解释性，**所以重点来了，作者这里采用了一种对loss的影响权重的等值percentiles（百分比分位数）划分算法（Weight Quantile Sketch）**， 我上面的这些铺垫也正是为了引出这个方式，下面就来看看作者是怎么做的，这个地方其实不太好理解，所以慢一些

作者进行候选点选取的时候，考虑的是想让loss在左右子树上分布的均匀一些，而不是样本数量的均匀，因为每个样本对降低loss的贡献可能不一样，按样本均分会导致分开之后左子树和右子树loss分布不均匀，取到的分位点会有偏差。这是啥意思呢？再来一个图（这个图得看明白了）：



这其实就是作者提出的那种找候选节点的方式（分桶的思路），明白了这个图之后，下面就是解释一下上面这个图的细节：第一个就是h_i是啥？它为啥就能代表样本对降低loss的贡献程度？ 第二个问题就是这个bin是怎么分的，为啥是0.6一个箱？

这其实就是作者提出的那种找候选节点的方式（分桶的思路），明白了这个图之后，下面就是解释一下上面这个图的细节：第一个就是h_i是啥？它为啥就能代表样本对降低loss的贡献程度？ 第二个问题就是这个bin是怎么分的，为啥是0.6一个箱？

下面从第一个问题开始，揭开h_i的神秘面纱， 其实h_i上面已经说过了，损失函数在样本i处的二阶导数啊！还记得开始的损失函数吗？

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

就是这个h_i， 那么你可能要问了， 为啥它就能代表第i个样本的权值啊？这里再拓展一下吧，我们在引出xgboost的时候说过，GBDT这个系列都是聚焦在残差上面，但是我们单看这个目标函数的话并没有看到什么残差的东西对不对？其实这里这个损失函数还可以进一步化简的（和上面的化简不一样，上面的化简是把遍历样本转到了遍历叶子上得到基于决策树的目标函数，这里是从目标函数本身出发进行化简）：

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

(hi/2)*(gi/hi)²

这样化简够简洁明了了吧， 你看到残差的身影了吗？后面的每一个分类器都是在拟合每个样本的一个残差$-\frac{g_i}{h_i}$，其实把上面化简的平方损失函数拿过来就一目了然了。而前面的h_i可以看做计算残差时某个样本的重要性,即每个样本对降低loss的贡献程度。第一个问题说的听清楚了吧 ;)

PS：这里加点题外话，Xgboost引入了二阶导之后，相当于在模型降低残差的时候给各个样本根据贡献度不同加入了一个权重，这样就能更好的加速拟合和收敛，GBDT只用到了一阶导数，这样只知道梯度大的样本降低残差效果好，梯度小的样本降低残差不好（这个原因我会放到Lightgbm的GOSS那里说到），但是好与不好的这个程度，在GBDT中无法展现。而xgboost这里就通过二阶导可以展示出来，这样模型训练的时候就有数了。

下面再解释第二个问题，这个分箱是怎么分的？比如我们定义一个数据集$D_k=\{(x_{1k},h_1),(x_{2k},h_2),\ldots,(x_{nk},h_n)\}$代表每个训练样本的第$k$个特征的取值和二阶梯度值，那么我们可以有一个排名函数$r_k(z)$:

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

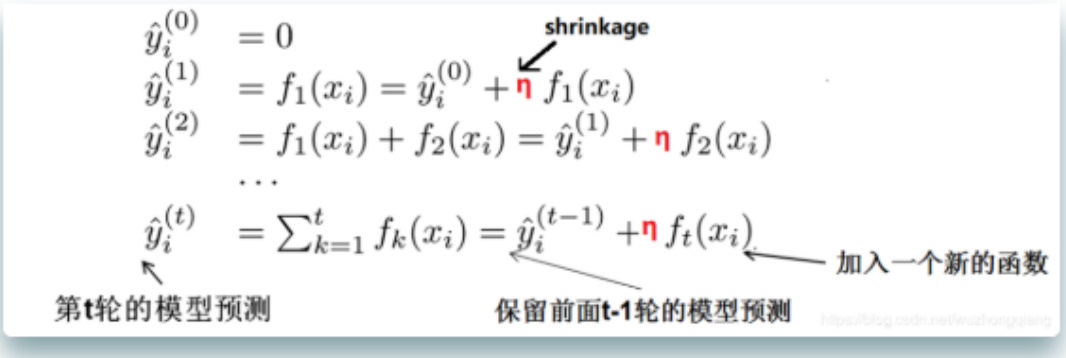
这里的z代表特征值小于z的那些样本。这个排名函数表示特征值小于z的样本的贡献度比例。假设上面图中，z是第一个候选点，那么$r_k(z)=\frac{1}{3}$，这个东西的目的就是去找相对准确的候选点$\{s_{k1},s_{k2},\ldots s_{kl}\}$，这里的$s_{k1}=\min_i \mathbf{x}_{ik}, s_{kl}=\max_i \mathbf{x}_{ik}$， 而相邻两个桶之间样本贡献度的差距应满足下面这个函数：

$$|r_k(s_{k,j})-r_k(s_{k,j+1})|<\epsilon$$

这个ϵ控制每个桶中样本贡献度比例的大小，其实就是贡献度的分位点。我们自己设定。比如在上面图中我们设置了$\epsilon=1/3$，这意味着每个桶样本贡献度的比例是1/3(贡献度的1/3分位点)，而所有的样本贡献度总和是1.8， 那么每个箱贡献度是0.6($1.8*\epsilon$)，分为3（$\frac{1}{\epsilon}$）个箱， 上面这些公式看起来挺复杂，可以计算起来很简单，就是计算一下总的贡献度， 然后指定ϵ，两者相乘得到每个桶的贡献度进行分桶即可。这样我们就可以确定合理的候选切分点，然后进行分箱了。

利用新的决策树预测样本值，并累加到原来的值上若干个决策树是通过加法训练的，所谓加法训练，本质上是一个元算法，适用于所有的加法模型，它是一种启发式算

法。运用加法训练，我们的目标不再是直接优化整个目标函数，而是分步骤优化目标函数，**首先优化第一棵树，完了之后再优化第二棵树，直至优化完K棵树**。整个过程如下图所示：



在这里插入图片描述

上图中会发现每一次迭代得到的新模型前面有个η（这个是让树的叶子节点权重乘以这个系数），这个叫做收缩率，这个东西加入的目的是削弱每棵树的作用，让后面有更大的学习空间，有助于防止过拟合。也就是，我不完全信任每一个残差树，每棵树只学到了模型的一部分，希望通过更多棵树的累加来来弥补，这样让这个让学习过程更平滑，而不会出现陡变。这个和正则化防止过拟合的原理不一样，这里是削弱模型的作用，而前面正则化是控制模型本身的复杂度。

好了，到这里为止，xgboost的数学原理部分就描述完了，希望我描述清楚了吧。简单的回顾一下上面的过程吧：xgboost是好多弱分类器的集成，训练弱分类器的策略就是尽量的减小残差，使得答案越来越接近正确答案。xgboost的精髓部分是目标函数的Taylor化简，这样就引入了损失函数的一阶和二阶导数。然后又把样本的遍历转成了对叶子节点的遍历，得到了最终的目标函数。这个函数就是衡量一棵树好坏的标准。在建树过程中，xgboost采用了贪心策略，并且对寻找分割点也进行了优化。基于这个，才有了后面的最优点切分建立一棵树的过程。xgboost训练的时候，是通过加法进行训练，也就是每一次只训练一棵树出来，最后的预测结果是所有树的加和表示。

3.实战

```
xgb1 = XGBClassifier( learning_rate =0.1,
n_estimators=1000, max_depth=5, min_child_weight=1,
gamma=0, subsample=0.8, colsample_bytree=0.8, objective=
'binary:logistic', nthread=4, scale_pos_weight=1, seed=27)
```

- 'booster':'gbtree'，这个指定基分类器
- 'objective': 'multi:softmax', 多分类的问题， 这个是优化目标，必须得有，因为xgboost里面有求一阶导数和二阶导数，其实就是这个。
- 'num_class':10, 类别数，与 multisoftmax 并用
- 'gamma':损失下降多少才进行分裂， 控制叶子节点的个数
- 'max_depth':12, 构建树的深度，越大越容易过拟合
- 'lambda':2, 控制模型复杂度的权重值的L2正则化项参数，参数越大，模型越不容易过拟合。
- 'subsample':0.7, 随机采样训练样本
- 'colsample_bytree':0.7, 生成树时进行的列采样
- 'min_child_weight':3, 孩子节点中最小的样本权重和。如果一个叶子节点的样本权重和小于min_child_weight则拆分过程结束
- 'silent':0 ,设置成1则没有运行信息输出，最好是设置为0.
- 'eta': 0.007, 如同学习率
- 'seed':1000,
- 'nthread':7, cpu 线程数

"

4.总结

下面看看xgboost相比于GBDT有哪些优点（面试的时候可能会涉及）：

- 精度更高：GBDT只用到一阶泰勒， 而xgboost对损失函数进行了二阶泰勒展开， 一方面为了增加精度， 另一方面也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
- 灵活性更强：GBDT以CART作为基分类器，而Xgboost不仅支持CART，还支持线性分类器，另外，Xgboost支持自定义损失函数，只要损失函数有一二阶导数。
- 正则化：xgboost在目标函数中加入了正则，用于控制模型的复杂度。有助于降低模型方差，防止过拟合。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。

- Shrinkage（缩减）：相当于学习速率。这个主要是为了削弱每棵树的影响，让后面有更大的学习空间，学习过程更加的平缓
- 列抽样：这个就是在建树的时候，不用遍历所有的特征了，可以进行抽样，一方面简化了计算，另一方面也有助于降低过拟合
- 缺失值处理：这个是xgboost的稀疏感知算法，加快了节点分裂的速度
- 并行化操作：块结构可以很好的支持并行计算

1. 二阶导数使结果更精确，加快收敛
2. 二阶导数可以作为样本对降低残差的贡献度。XGB中使用Weight quantile sketch方法来寻找树构建过程中近似的最佳分裂点