

集成模型之GBDT算法

本文从泰勒公式为起点，推导出梯度下降法和牛顿法，进而将梯度下降法和牛顿法推广到函数空间中，得到传统的GBDT算法和XGBoost算法。本文着重讲解了GBDT，XGBoost，LightGBM的算法原理，参数解析，调参的大致流程以及三者之间的异同点。希望读者读完本篇文章后，能够对GBDT算法系列有更深刻的了解和认识。

1. 引言

1.1 泰勒公式

- 定义：泰勒公式一个用函数在某点的信息描述其附近取值，具有局部有效性。
- 迭代形式：

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} (x - x_0)^n$$

一阶泰勒公式展开： $f(x) = f(x_0) + f'(x_0)(x - x_0)$

二阶泰勒公式展开： $f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2$

- 迭代形式：假设 $x^t = x^{t-1} + \Delta x$ ，将 $f(x^t)$ 在 x^{t-1} 处进行泰勒公式展开：

$$f(x^t) = f(x^{t-1} + \Delta x) \approx f(x^{t-1}) + f'(x^{t-1})\Delta x + \frac{f''(x^{t-1})\Delta x^2}{2}$$

1.2 最优化方法

梯度下降法 (Gradient Descend Method)

在机器学习任务中，需要最小化损失函数 $L(\theta)$ ，其中 θ 是要求解的模型参数。梯度下降法常用来求解这种无约束最优化问题，它是一种迭代方法：选取初值 θ^0 ，不断迭代，更新 θ 的值，进行损失函数的极小化。

- 迭代公式： $\theta^t = \theta^{t-1} + \Delta\theta$
- 将 $L(\theta^t)$ 在 θ^{t-1} 处进行一阶泰勒展开：

$$\begin{aligned} L(\theta^t) &= L(\theta^{t-1} + \Delta\theta) \\ &\approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta \end{aligned}$$

要使得 $L(\theta^t) < L(\theta^{t-1})$ ，可取： $\Delta\theta = -\alpha L'(\theta^{t-1})$

- 则： $\theta^t = \theta^{t-1} - \alpha L'(\theta^{t-1})$

这里 α 是步长，一般直接赋一个小的数。

牛顿法 (Newton's Method)

- 将 $L(\theta^t)$ 在 θ^{t-1} 处进行二阶泰勒展开：

$$L(\theta^t) \approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta + L''(\theta^{t-1})\frac{\Delta\theta^2}{2}$$

为了简化分析过程，假设参数是标量（即 只有一维），则可将一阶和二阶导数分别记为 g 和 h ：

$$L(\theta^t) \approx L(\theta^{t-1}) + g\Delta\theta + h\frac{\Delta\theta^2}{2}$$

- 要使得 $L(\theta^t)$ 极小，即让 $g\Delta\theta + h\frac{\Delta\theta^2}{2}$ 极小，可令： $\frac{\partial g\Delta\theta + h\frac{\Delta\theta^2}{2}}{\partial \Delta\theta} = 0$
求得： $\Delta\theta = -\frac{g}{h}$ ，故 $\theta^t = \theta^{t-1} + \Delta\theta = \theta^{t-1} - \frac{g}{h}$
- 参数 θ 推广到向量形式，迭代公式： $\theta^t = \theta^{t-1} - H^{-1}g$ ，这里 H 是海森矩阵。

1.3 从参数空间到函数空间

- GBDT 在函数空间中利用梯度下降法进行优化
- XGBoost 在函数空间中用牛顿法进行优化

注：实际上GBDT泛指所有梯度提升树算法，包括XGBoost，它也是GBDT的一种变种，这里为了区分它们，GBDT特指“Greedy unction Approximation: A Gradient Boosting Machine”里提出的算法，它只用了一阶导数信息。

以下展示了从Gradient Descend 到 Gradient Boosting：

参数空间	函数空间
$\theta^t = \theta^{t-1} + \theta_t$	$f^t(x) = f^{t-1}(x) + f_t(x)$
$\theta_t = -\alpha_t g_t$	$f_t(x) = -\alpha_t g_t(x)$
$\theta = \sum_{t=0}^T \theta_t$	$F(x) = \sum_{t=0}^T f_t(x)$
最终参数等于每次迭代的增量的累加和， θ_0 为初值	最终函数等于每次迭代的增量的累加和， $f_0(x)$ 为初值，通常为常数

以下展示了从Newton's Method 到 Newton Boosting的过程：

参数空间	函数空间
$\theta^t = \theta^{t-1} + \theta_t$	$f^t(x) = f^{t-1}(x) + f_t(x)$
$\theta_t = -H^{-1}g_t$	$f_t(x) = -\frac{g_t(x)}{h_t(x)}$
$\theta = \sum_{t=0}^T \theta_t$	$F(x) = \sum_{t=0}^T f_t(x)$
最终参数等于每次迭代的增量的累加和， θ_0 为初值	最终函数等于每次迭代的增量的累加和， $f_0(x)$ 为初值，通常为常数

表中 θ^t 表示第t次迭代后的参数， θ_t 表示第t次迭代的参数增量。

2. GBDT

GBDT有很多简称，有GBT（Gradient Boosting Tree），GTB（Gradient Tree Boosting），GBRT（Gradient Boosting Regression Tree），MART(Multiple Additive Regression Tree)，本文统一简称GBDT。

回顾下 Adaboost，是利用前一轮迭代弱学习器的误差率更新训练集的权重来训练下一个弱学习器，这样一轮轮的迭代下去。GBDT 也是迭代，并且使用了前向分布算法，同时迭代思路和 Adaboost 也有所不同。

GBDT 通过多轮迭代，每轮迭代产生一个弱分类器，每个分类器在上一轮分类器的残差基础上进行训练。对弱分类器的要求一般是足够简单，并且是低方差和高偏差的。因为训练的过程是通过降低偏差来不断提高最终分类器的精度。

由以上我们知道GBDT也是一个前向加法模型，即：

$$f^t(x) = f^{t-1}(x) + f_t(x)$$

那么：

$$f^t(x) - f^{t-1}(x) = f_t(x)$$

其中

举一个通俗的例子解释，假如有个人30岁，我们首先用20岁去拟合，发现损失有10岁，这时我们用6岁去拟合剩下的损失，发现差距还有4岁，第三轮我们用3岁拟合剩下的差距，差距就只有一岁了。如果我们的迭代轮数还没有完，可以继续迭代下面，每一轮迭代，拟合的岁数误差都会减小，这样我们拟合的最终结果便是：

$$20 + 6 + 3 + \dots$$

总结一下，GBDT和XGBoost最主要的区别在于两者如何识别模型的问题。AdaBoost用错分数据点来识别问题，通过调整错分数据点的权重来改进模型。Gradient Boosting通过残差来识别问题，通过计算负梯度来改进模型。

2.1 GBDT的算法原理

输入是训练集样本 $T = \{(x, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$ ，最大迭代次数 T ，损失函数 L 。

输出是强学习器 $f(x)$ 。

1. 初始化基学习器：

$$f_0(x) = \underbrace{\arg \min}_c \sum_{i=1}^m L(y_i, c)$$

2. 对迭代轮数 $t = 1, 2, \dots, T$ 有：

◦ 对样本 $i = 1, 2, \dots, m$ 计算负梯度：

$$r_{ti} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x)=f_{t-1}(x)}$$

◦ 将上步得到的残差作为样本新的真实值，利用 (x_i, r_{ti}) ($i = 1, 2, \dots, m$) 作为下棵树的训练数据，拟合一颗CART回归树，得到第 t 颗回归树，其对应的叶子节点区域为 $R_{tj}, j = 1, 2, \dots, J$ ，其中 J 为回归树 t 的叶子节点的个数。

◦ 对叶子区域 $j = 1, 2, \dots, J$ ，计算最佳拟合值

$$c_{tj} = \underbrace{\arg \min}_c \sum_{x_i \in R_{tj}} L(y_i, f_{t-1}(x_i) + c)$$

o 更新强学习器:

$$f_t(x) = f_{t-1}(x) + \sum_{j=1}^J c_{tj} I(x \in R_{tj})$$

3. 得到强学习器 $f(x)$ 的表达式:

$$f(x) = f_T(x) = f_0(x) + \sum_{t=1}^T \sum_{j=1}^J c_{tj} I(x \in R_{tj})$$

2.2 GBDT常用损失函数

对于分类算法，其损失函数一般有对数损失函数和指数损失函数两种:

- 如果是指数损失函数，则损失函数表达式为

$$L(y, f(x)) = e^{-yf(x)}$$

对应的梯度为:

$$-ye^{-yf(x)}$$

- 如果是对数损失函数，则损失函数表达式为

$$L(y, f(x)) = \log(1 + e^{-yf(x)})$$

对应负梯度误差为:

$$y_i / (1 + e^{y_i f(x_i)})$$

对于回归算法，常用损失函数有如下4种:

- 均方差:

$$L(y, f(x)) = (y - f(x))^2$$

- 绝对损失

$$L(y, f(x)) = |y - f(x)|$$

对应负梯度误差为:

$$\text{sign}(y_i - f(x_i))$$

- Huber损失，它是均方差和绝对损失的折衷产物，对于远离中心的异常点，采用绝对损失，而中心附近的点采用均方差。这个界限一般用分位数点度量。损失函数如下:

$$L(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & |y - f(x)| \leq \delta \\ \delta(|y - f(x)| - \frac{\delta}{2}) & |y - f(x)| > \delta \end{cases}$$

对应的负梯度误差为:

$$\begin{cases} y_i - f(x_i) & |y_i - f(x_i)| \leq \delta \\ \delta \operatorname{sign}(y_i - f(x_i)) & |y_i - f(x_i)| > \delta \end{cases}$$

- 分位数损失。它对应的是分位数回归的损失函数，表达式为

$$L(y, f(x)) = \sum_{y \geq f(x)} \theta |y - f(x)| + \sum_{y < f(x)} (1 - \theta) |y - f(x)|$$

其中 θ 为分位数，我们需要在回归前指定。对应的负梯度误差为：

$$\begin{cases} \theta & y_i \geq f(x_i) \\ \theta - 1 & y_i < f(x_i) \end{cases}$$

对于Huber损失和分位数损失，主要用于健壮回归，也就是减少异常点对损失函数的影响。

2.3 GBDT的正则化

GBDT的正则化主要有三种方式：

1. 和Adaboost类似添加步长 (learning rate) ν ，对于前面的弱学习器的迭代

$$f_k(x) = f_{k-1}(x) + h_k(x)$$

加上了正则化项之后，则有

$$f_k(x) = f_{k-1}(x) + \nu h_k(x)$$

ν 的取值范围为 $0 < \nu \leq 1$ 。对于同样的训练集学习效果，较小的 ν 意味着我们需要更多的弱学习器的迭代次数。通常我们用步长和迭代最大次数一起来决定算法的拟合效果。

2. 子采样 (subsample)。注意这里的子采样和随机森林不一样，随机森林使用的是放回抽样，而这里是不放回抽样。通过子采样可以减少方差，即防止过拟合，但是会增加样本拟合的偏差，因此取值不能太低。推荐子采样比例在 $[0.5, 0.8]$ 之间。
3. 对于弱学习器即CART回归树进行正则化剪枝。

2.4 GBDT参数解析

```
from sklearn.ensemble import GradientBoostingClassifier
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_decrease=0.0, min_impurity_split=None, init=None, random_state=None,
max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, presort='auto')
```

总的来说GBM的参数可以被归为三类：

1. 树参数：调节模型中每个决定树的性质

1. min_samples_split：定义了树中一个节点所需要用来分裂的最少样本数

- 可以避免过度拟合(over-fitting)。如果用于分类的样本数太小，模型可能只适用于用来训练的样本的分类，而用较多的样本数则可以避免这个问题。
- 但是如果设定的值过大，就可能出现欠拟合现象(under-fitting)。因此我们可以用CV值（离散系数）考量调节效果。

2. min_samples_leaf: 定义了树中终点节点所需要的最少的样本数

- 同样，它也可以用来防止过度拟合。
- 在不均等分类问题中(imbalanced class problems)，一般这个参数需要被设定为较小的值，因为大部分少数类别 (minority class) 含有的样本都比较小。

3. min_weight_fraction_leaf: 终点节点所需的样本数占总样本数的比值

- 和min_samples_leaf很像，不同的是这里需要的是一个比例而不是绝对数值：
- 2 和 3 只需要定义一个就行了

4. max_depth: 定义了树的最大深度

- 它也可以控制过度拟合，因为分类树越深就越可能过度拟合。
- 当然也应该用 CV 值检验。

5. max_leaf_nodes: 定义了决定树里最多能有多少个终点节点

- 这个属性有可能在上面 max_depth 里就被定义了。比如深度为n的二叉树就有最多 2^n 个终点节点。
- 如果我们定义了max_leaf_nodes，GBM就会忽略前面的max_depth。

6. max_features: 用于分类的最大特征数

- 根据经验一般选择总特征数的平方根就可以工作得很好了，但还是应该用不同的值尝试，最多可以尝试总特征数的30%-40%。
- 过多的分类特征可能也会导致过度拟合。

2. Boosting参数：调节模型中boosting的操作

1. learning_rate

- 这个参数决定着每一个决定树对于最终结果（步骤2.4）的影响。GBM设定了初始的权重值之后，每一次树分类都会更新这个值，而learning_rate控制着每次更新的幅度。
- 一般来说这个值不应该设的比较大，因为较小的learning_rate使得模型对不同的树更加稳健，就能更好地综合它们的结果。

2. n_estimators

- 定义了需要使用到的决定树的数量（步骤2）
- 虽然GBM即使在有较多决定树时仍然能保持稳健，但还是可能发生过度拟合。所以也需要针对learning_rate用CV值检验。

3. subsample

- 训练每个决定树所用到的子样本占总样本的比例，而对于子样本的选择是随机的。
- 用稍小于1的值能够使模型更稳健，因为这样减少了方差。
- 一把来说用0.8就行了，更好的结果可以用调参获得。

3. 其他模型参数：调节模型总体的各项运作

1. loss

- 指的是每一次节点分裂所要最小化的损失函数(loss function)
- 对于分类和回归模型可以有不同的值。一般来说不用更改，用默认值就可以了，除非你对它及它对模型的影响很清楚。

2. init

- 它影响了输出参数的起始化过程
- 如果我们有一个模型，它的输出结果会用来作为GBM模型的起始估计，这个时候就可以用init

3. random_state

- 作为每次产生随机数的随机种子
- 使用随机种子对于调参过程是很重要的，因为如果我们每次都使用不同的随机种子，即使参数值没变每次出来的结果也会不同，这样不利于比较不同模型的结果。
- 任一个随即样本都有可能产生过度拟合，可以用不同的随机样本建模来减少过度拟合的可能，但这样计算上也会昂贵很多，因而我们很少这样用

4. verbose：决定建模完成后对输出的打印方式：

- 0：不输出任何结果（默认）
- 1：打印特定区域的树的输出结果
- >1：打印所有结果

5. warm_start

- 这个参数的效果很有趣，有效地使用它可以省很多事
- 使用它我们就可以用一个建好的模型来训练额外的决定树，能节省大量的时间，对于高阶应用我们应该多多探索这个选项。

6. presort

- 决定是否对数据进行预排序，可以使得树分裂地更快。
- 默认情况下是自动选择的，当然你可以对其更改

简单总结一下GBDT调参方法：

1. 选择一个相对来说稍微高一点的learning rate。一般默认的值是0.1，不过针对不同的问题，0.05到0.2之间都可以。
2. 决定当前learning rate下最优的决策树数量 n_estimators。它的值应该在40-70之间 (根据电脑性能而定)。如果n_estimators过大，可以适当调高learning rate，再调整n_estimators；如果n_estimators过小，可以适当降低learning rate，再调整n_estimators。
3. 接着调节树参数：
 - 调节max_depth和 min_samples_split
 - 调节min_samples_leaf
 - 调节max_features
4. 降低learning rate，同时会增加n_estimators使得模型更加稳健

参考文献：

1. T. Hastie / R. Tibshirani / J. H. Friedman 《The Elements of Statistical Learning》
2. [梯度提升树\(GBDT\)原理小结](#)
3. [Complete Guide to Parameter Tuning in Gradient Boosting \(GBM\) in Python](#)
4. [机器学习系列\(11\) Python中Gradient Boosting Machine\(GBM\) 调参方法详解](#)

3. XGBoost

XGBoost 是 “Extreme Gradient Boosting” 的缩写

3.1 假设函数形式

给定数据集 $\mathcal{D} = \{(x_i, y_i)\}$ ，XGBoost进行加法训练，学习K棵树，采用以下函数对样本进行预测：

$$\hat{y}_i = \sum_{k=1}^K f_k(x_i), \quad f_k \in \mathcal{F}$$

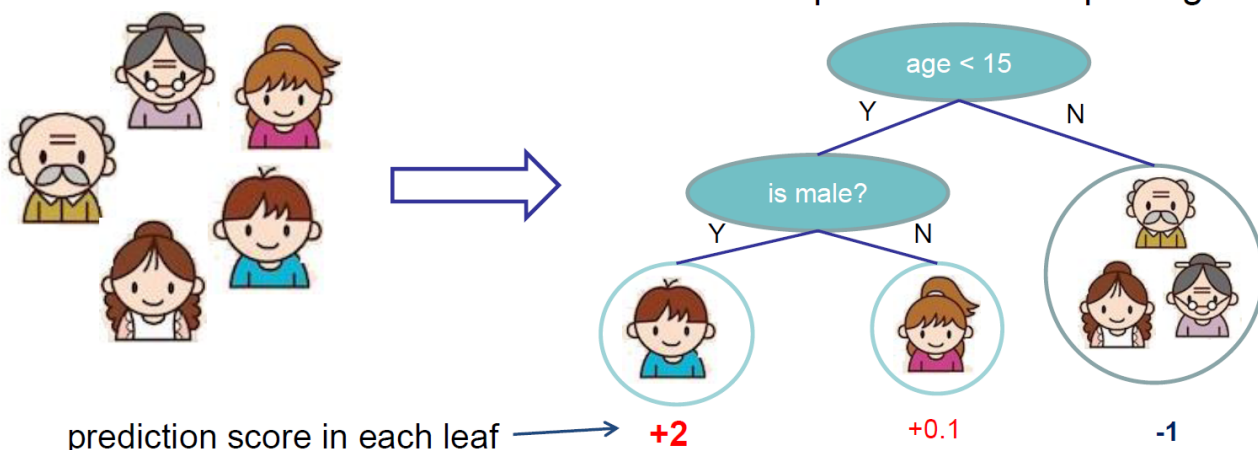
这里的 \mathcal{F} 是包含所有的回归树的函数空间, $f_k(x)$ 是回归树(CART)。

那么什么是回归树呢? 下面用一个小例子来解释。

例1: CART 回归树

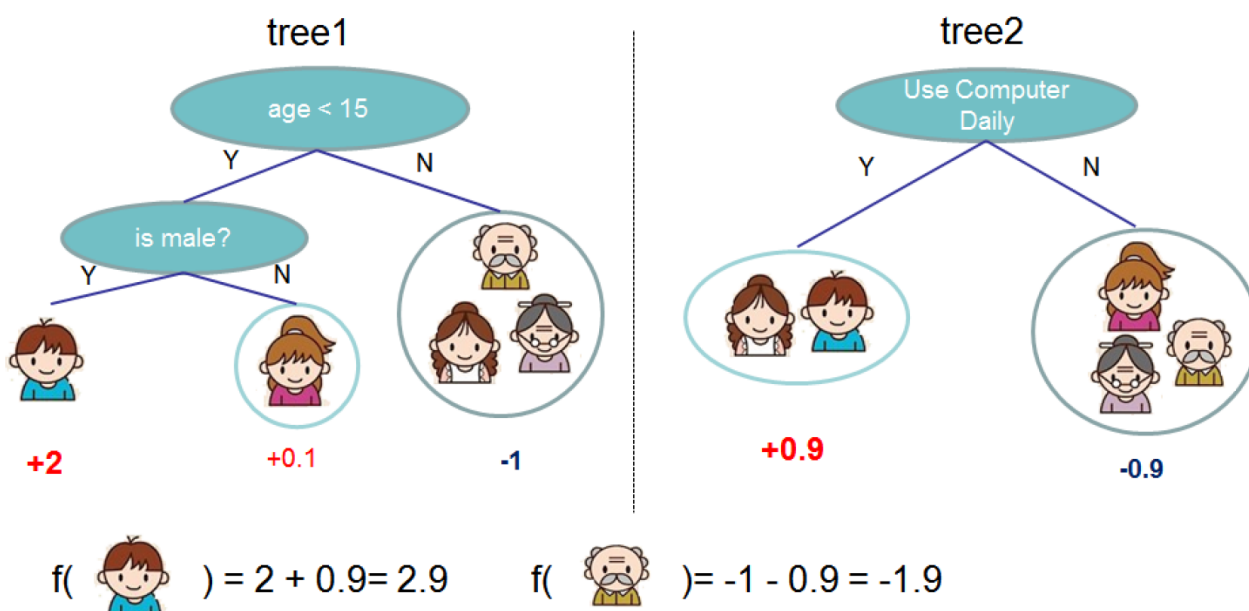
Input: age, gender, occupation, ...

Does the person like computer games



CART会把样本根据输入的属性分配到各个叶子节点, 而每个叶子节点上面都会对应一个实数分数。上面的例子可以把叶子的分数理解为有多可能这个人喜欢电脑游戏。我们可以简单地把CART树理解为decision tree的一个扩展。从简单的类标到分数之后, 我们可以做很多事情, 如回归、分类, 排序。对于回归问题, 可以直接作为目标值, 对于分类问题, 需要映射成概率, 二分类一般用sigmoid函数, 多分类一般用softmax函数。

和决策树类似, 一个CART往往过于简单无法有效地预测, 因此一个更加强力的模型叫做tree ensemble。而XGBoost便是一个tree ensembles。



在上面的例子中, 我们用两棵树来进行预测。我们对于每个样本的预测结果就是每棵树预测分数的和。

到这里，模型的一般形式就介绍完毕了。下面有两个小问题：

Q1：之前讲解的随机森林，boosted tree和tree ensemble有什么关系呢？

RF和boosted tree的模型都是tree ensemble，只是构造（学习）模型参数的方法不同。

Q2：在XGBoost中的“参数”是什么？

这里的参数对应了树的结构，以及每个叶子节点上面的预测分数。

3.2 目标函数

3.2.1 目标函数的一般形式

那么我们怎样学习XGBoost的参数呢？通常我们会先定义一个目标函数，然后利用优化算法进行优化。目标函数的一般形式如下所示：

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

$F(\Theta)$ ：误差项，表示模型拟合数据的程度，这有助于避免欠拟合。常见的误差项有square loss：

$l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$ ，logloss： $l(y_i, \hat{y}_i) = y_i \ln(1 + e^{-\hat{y}_i}) + (1 - y_i) \ln(1 + e^{\hat{y}_i})$ 等。

$\Omega(\Theta)$ ：正则化项，用于控制模型的复杂性，这有助于避免过拟合。常见的正则化项包括L1正则，L2正则等。

下面我们分别讲解一下误差项和正则项。

3.2.2 误差项

我们知道 *XGBoost* 是一个加法模型，可以的带第 t 次迭代后，模型的预测等于前 $t-1$ 次的模型预测加上第 t 棵树的预测：

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

此时目标函数可写作：

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

公式中 $y_i, \hat{y}_i^{(t-1)}$ 都已知，模型要学习的只有第 t 棵树 $f_t(x_i)$ 。

我们使用泰勒公式：

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

将误差函数在 $\hat{y}_i^{(t-1)}$ 处进行展开：

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

公式中， $g_i = \frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$ ， $h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial^2 \hat{y}_i^{(t-1)}}$

将公式中的常数项去掉，得到：

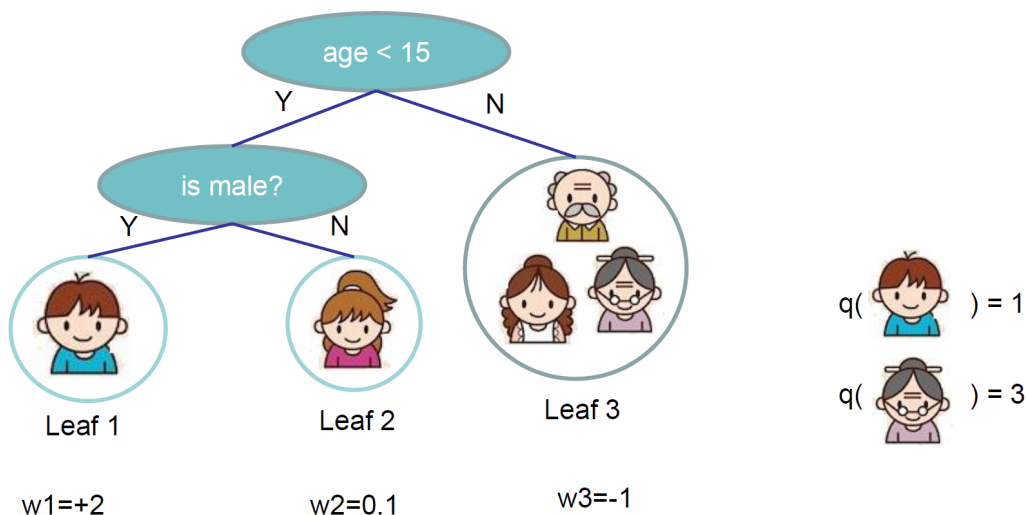
$$\tilde{L}^{(t)} \approx \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

由于这里基学习器 f_t 是一个回归树：

$$f_t(x) = w_{q(x)} \quad (q: R^m \rightarrow T, w \in R^T)$$

回归树实质是一个能够将特征映射到分数上的函数。 $q(x)$ 表示将样本 x 分到了某个叶子节点上， w 是叶子节点的分数 (leaf score)， 所以 $w_{q(x)}$ 表示回归树对样本 x 的分数。

如下所示： 男孩表示 x ， $q(x)$ 表示 Leaf 1， $w_{q(x)}$ 表示 $w_1 = +2$ 。



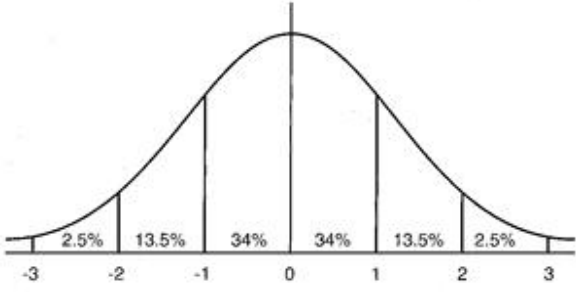
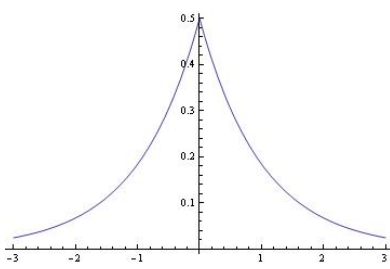
写成树结构的形式，得到：

$$\tilde{L}^{(t)} = \sum_{i=1}^n [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \Omega(f_t)$$

式中 $\sum_{i=1}^n$ 表示对样本进行累加

3.2.3 正则项

正则项的原理我们在[直观理解正则化](#)已经介绍过，这里我们从Bayes角度来分析，正则化相当于对模型参数引入先验分布：

L2正则化	L1正则化
	
模型参数服从高斯分布，对参数加了分布约束，大部分绝对值很小	模型参数服从拉普拉斯分布，对参数加了分布约束，大部分取值为0

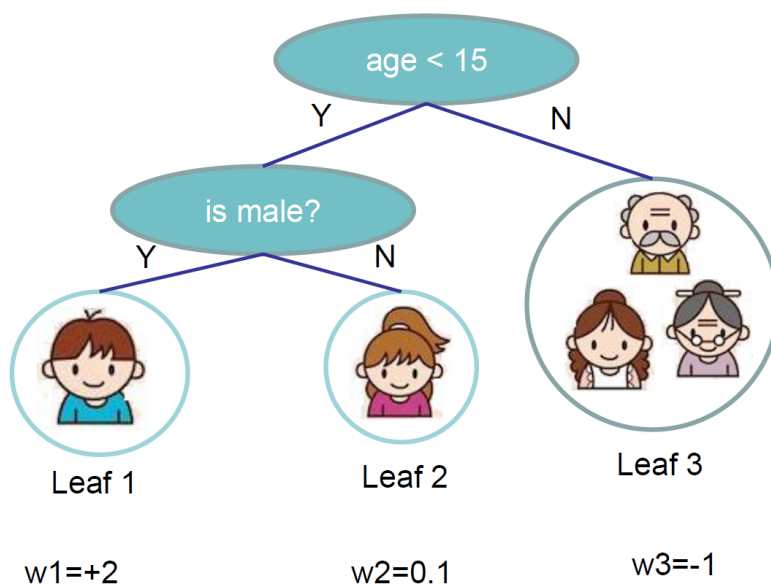
那么有哪些指标可以衡量树的复杂度？

- 树的深度，
- 内部节点个数，
- **叶子节点个数(T)**,
- **叶节点分数(w)**...

XGBoost采用的叶子节点个数和叶子节点分数来定义惩罚项：

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

式中 $\sum_{j=1}^T$ 表示对叶结点进行累加。对叶子节点个数进行惩罚，相当于在训练过程中做了剪枝。



$$\Omega(f) = \gamma 3 + \frac{1}{2} \lambda (2^2 + 0.1^2 + (-1)^2)$$

3.2.4 统一目标函数

将误差项和正则项分别化简，得到：

$$\tilde{L}^{(t)} = \sum_{i=1}^n [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

式中 $\sum_{i=1}^n$ 表示对样本进行累加， $\sum_{j=1}^T$ 表示对叶结点进行累加。

以上我们可以发现误差项是样本的累加，而正则化项是叶结点的累加，那么我们该怎样把这两种形式统一呢？

定义映射函数 $q(x_i) = j$ ，表示样本 i 的得分 $q(x_i)$ 与叶子节点 j 的得分一致时 ($q(x_i) = j$)，可以将样本 i 映射到叶子节点 j 上。

那么每个叶节点 j 上的样本集合可以表示为 $I_j = \{i | q(x_i) = j\}$

则目标函数可以写成按叶节点累加的形式：

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

令 $G_j = \sum_{i \in I_j} g_i$ ， $H_j = \sum_{i \in I_j} h_i$ ，那么：

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

如果确定了树的结构（即 $q(x)$ 确定），为了使目标函数最小，可以令其导数为 0，解得每个叶节点的最优预测分数为：






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

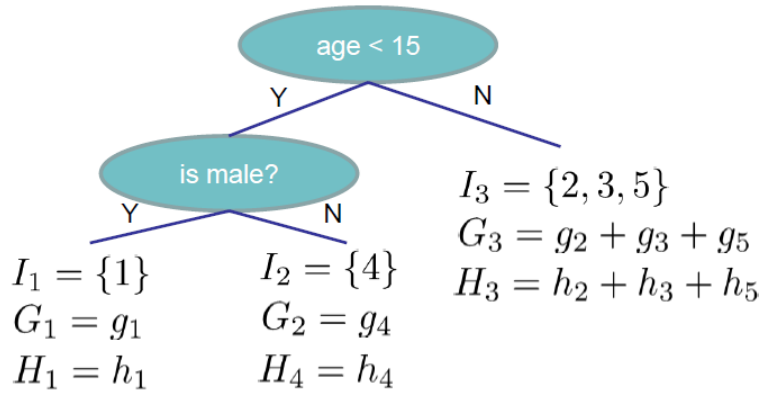
代入目标函数，得到最小损失为：

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

下图展示了目标函数的计算过程。对于一个给定的树结构，我们把统计 g_i 和 h_i push 到它们所属的叶子上，统计每个叶子数据加和得到 G_i, H_i ，然后带入目标函数的公式即可。

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

3.3 回归树的学习策略

当回归树的结构确定时，我们前面已经推导出其最优的叶节点分数以及对应的最小损失值，问题是怎么确定树的结构？

- 暴力枚举法：从函数空间里所有的决策树中找出最优的决策树
- 启发式方法：每次尝试分裂一个叶节点，计算分裂前后的增益，选取增益最大的特征的进行分割。

分裂前后的增益怎么计算？

- ID3算法采用entropy计算信息增益
- C4.5算法采用信息增益比
- CART采用gini指数计算增益

XGBoost呢？XGBoost采用目标函数来计算增益。XGBoost得目标函数：

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

因此，对一个叶子节点进行分裂，分裂前后的增益差定义为：

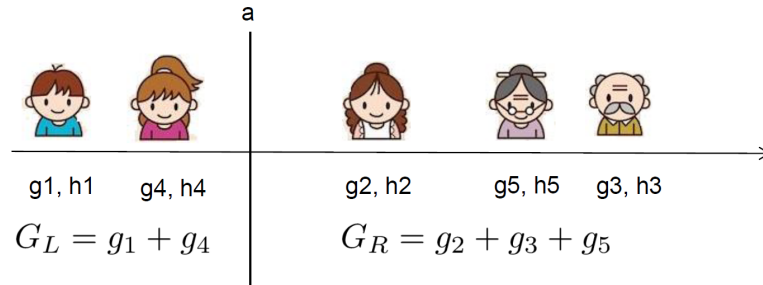
$$Gain = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_R + G_L)^2}{H_R + H_L + \lambda} - \gamma$$

$\frac{G_L^2}{H_L + \lambda}$ 为分裂后左孩子节点的得分 (the score of left child), $\frac{G_R^2}{H_R + \lambda}$ 为分裂后右孩子节点的得分 (the score of right child), $\frac{(G_R + G_L)^2}{H_R + H_L + \lambda}$ 表示分裂前的叶子节点的得分, γ 表示由于引入新的叶子节点产生的复杂度代价(the complexity cost by introducing additional leaf).

- 这个公式形式上跟ID3算法（采用entropy计算增益）、CART算法（采用gini指数计算增益）是一致的，都是用分裂后的某种值减去分裂前的某种值，从而得到增益
- 引入分割不一定会使情况变好，因为最后有一个新叶子的惩罚项。即当引入的分割所带来的增益大于阈值 γ 时才让节点分裂， γ 是正则项里叶子节点数T的系数，所以xgboost在优化目标函数的同时相当于做了预剪枝。

- 上式中还有一个系数 λ ，是正则项里leaf score的平方的系数，对leaf score做了平滑，也起到了防止过拟合的作用，这个是传统GBDT里不具备的特性

以之前的案例为例，假设我们需要分裂的特征为年龄，把所有的实例按照排序顺序排列，只要做一遍从左到右的扫描就可以枚举出年龄字段所有可能得分割点，计算增益，选取增益最大得那个点。假设分割点为a时，分割之后叶节点为以下表格所示：



将 G_L, G_R 和 H_L, H_R 带入Gain公式即可求得增益。

3.4 查找分裂节点 (Split Finding)

3.4.1 精确算法

遍历所有特征的所有可能的分割点，计算gain值，选取值最大的特征和分割点去分割

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** $sorted(I, \text{by } \mathbf{x}_{jk})$ **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

3.4.2 近似算法

如果数据不能一次读入内存，使用贪心算法效率较低，这时候就需要用到近似算法，近似算法时对于每个特征，只考察分位点，减少计算复杂度。

- Global: 学习每棵树前，提出候选切分点
- Local: 每次分裂前，重新提出候选切分点

Algorithm 2: Approximate Algorithm for Split Finding

```

for  $k = 1$  to  $m$  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  $m$  do
  |  $G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} g_j$ 
  |  $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} h_j$ 
end
  Follow same step as in previous section to find max
  score only among proposed splits.
  
```

近似算法举例：三分位数

			1/3 percentile			2/3 percentile			
features	1	1	3	4	5	12	45	50	99
labels	1	0	0	1	1	0	0	0	0
g_i	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9
h_i	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9
	G_1, H_1			G_2, H_2			G_3, H_3		

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

3.4.3 加权的近似算法

实际上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 作为权重(Weighted Quantile Sketch)，比如：

features	1	1	3	4	5	12	45	50	99
h_i	0.1	0.1	0.1	0.1	0.1	0.1	0.4	0.2	0.6
							1/3 percentile	2/3 percentile	

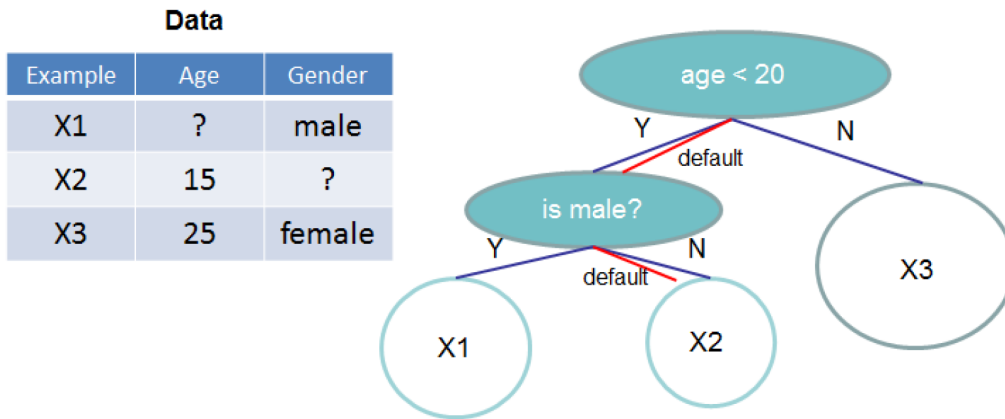
为什么用 h_i 加权？把目标函数整理成以下形式，可以看出 h_i 有对loss加权的作用

$$\sum_{i=1}^n \frac{1}{2} h_i (f_t(x_i) - g_i/h_i)^2 + \Omega(f_t) + constant$$

这是一个关于标签为 $\frac{g_i}{h_i}$ 和权重为 h_i 的平方误差形式。

3.4.4 稀疏值处理

寻找分裂点的过程中，如何克服数据稀疏。稀疏数据可能来自于missing value、大量的0值、或者特征工程例如采用one-hot表示带来的。为了解决这个问题，设定一个默认指向，当发生特征缺失的时候，将样本分类到默认分支，如下图：



默认方向由训练集中non-missing value学习而得，把不存在的值也当成missing value进行学习和处理，如下：

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 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 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

以上出处[Boosted Trees 介绍](#)

3.5 XGBoost的特性

- xgboost与传统的GBDT相比，对代价函数进行了二阶泰勒展开，同时用到了一阶与二阶导数，而GBDT在优化时只用到一阶导数的信息，类似牛顿法与梯度下降的区别。
- xgboost在代价函数里加入了正则项 (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)，用于控制模型的复杂度，防止过拟合。
- 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。
- 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。
- 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。注意xgboost的并行不是tree层面的并行，xgboost也是一次迭代完才能进行下一次迭代的。xgboost的并行是在特征层面的。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。
- 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，xgboost采用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。

3.6 XGBoost参数解析

```
from xgboost.sklearn import XGBClassifier

XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
objective='binary:logistic', booster='gbtree', n_jobs=1, nthread=None, 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)
```

XGBoost的参数可以分为三类：

1. 通用参数：用来控制XGBoost的宏观功能

1. booster：默认gbtree

- 选择每次迭代的模型，有两种选择：
gbtree：基于树的模型
gblinear：线性模型

2. silent：默认0

- 当这个参数值为1时，静默模式开启，不会输出任何信息。
- 一般这个参数就保持默认的0，因为这样能帮我们更好地理解模型。

3. nthread (默认None，使用最大可能的线程) 或 n_jobs (默认值为1，只使用一个线程)：

- 这个参数用来进行多线程控制，应当输入系统的核数。
- 通常用n_jobs
- 如果你希望使用CPU全部的核，n_jobs = -1。

还有两个参数，XGBoost会自动设置。

2. booster参数：控制每一步的booster(tree/regression)

尽管有两种booster可供选择，我这里只介绍**tree booster**，因为它的表现远远胜过**linear booster**，所以linear booster很少用到。

1. learning_rate: 默认0.3

- 和GBDT中的 learning rate 参数类似。
- 通过减少每一步的权重，可以提高模型的鲁棒性。
- 典型值为0.01-0.2。

2. min_child_weight: 默认1

- 决定最小叶子节点样本权重和。
- 和GBDT的 min_child_leaf 参数类似，但不完全一样。XGBoost的这个参数是最小样本权重的和，而GBDT参数是最小样本总数。
- 这个参数用于避免过拟合。当它的值较大时，可以避免模型学习到局部的特殊样本。
- 但是如果这个值过高，会导致欠拟合。这个参数需要使用CV来调整。

3. max_depth: 默认6

- 和GBDT中的参数相同，这个值为树的最大深度。
- 这个值也是用来避免过拟合的。max_depth越大，模型会学到更具体更局部的样本。
- 需要使用CV函数来进行调优。
- 典型值: 3-10

4. max_leaf_nodes

- 树上最大的节点或叶子的数量。
- 可以替代max_depth的作用。因为如果生成的是二叉树，一个深度为n的树最多生成 2^n 个叶子。
- 如果定义了这个参数，GBDT会忽略max_depth参数。

5. gamma[默认0]

- 在节点分裂时，只有分裂后损失函数的值下降了，才会分裂这个节点。Gamma指定了节点分裂所需的最小损失函数下降值。
- 这个参数的值越大，算法越保守。这个参数的值和损失函数息息相关，所以是需要调整的。

6. max_delta_step[默认0]

- 这参数限制每棵树权重改变的最大步长。如果这个参数的值为0，那就意味着没有约束。如果它被赋予了某个正值，那么它会让这个算法更加保守。
- 通常，这个参数不需要设置。但是当各类别的样本十分不平衡时，它对逻辑回归是很有帮助的。
- 这个参数一般用不到，但是你可以挖掘出来它更多的用处。

7. subsample: 默认1

- 和GBDT中的subsample参数一模一样。这个参数控制对于每棵树，随机采样的比例。
- 减小这个参数的值，算法会更加保守，避免过拟合。但是，如果这个值设置得过小，它可能会导致欠拟合。
- 典型值: 0.5-1

8. colsample_bytree: 默认1

- 和GBDT里面的max_features参数类似。用来控制每棵树随机采样的列数的占比(每一列是一个特征)。
- 典型值: 0.5-1

9. colsample_bylevel: 默认1

- 用来控制树的每一级的每一次分裂，对列数的采样的占比。
- 我个人一般不太用这个参数，因为subsample参数和colsample_bytree参数可以起到相同的作用。但是如果感兴趣，可以挖掘这个参数更多的用处。

10. reg_lambda: 默认1

- 权重的L2正则化项。(和Ridge regression类似)。
- 这个参数是用来控制XGBoost的正则化部分的。虽然大部分数据科学家很少用到这个参数，但是这个参数在减少过拟合上还是可以挖掘出更多用处的。

11. reg_alpha: 默认1

- 权重的L1正则化项。(和Lasso regression类似)。
- 可以应用在很高维度的情况下，使得算法的速度更快。

12. scale_pos_weight: 默认1

- 在各类别样本十分不平衡时，把这个参数设定negative_samples/positive_samples，可以使算法更快收敛。
- 比如当正负样本比例为1:10时，可以调节scale_pos_weight=10。

13. n_estimators: 默认100

- 定义了需要使用到的决策树的数量
- 虽然GBM即使在有较多决定树时仍然能保持稳健，但还是可能发生过度拟合。所以也需要针对learning rate用CV值检验。

3. 学习目标参数：控制训练目标的表现

1. objective: 默认binary:logistic

定义需要被最小化的损失函数。最常用的值有：

- binary: logistic 二分类的逻辑回归，返回预测的概率(不是类别)。
- multi: softmax 使用softmax的多分类器，返回预测的类别(不是概率)。
 - 在这种情况下，你还需要多设一个参数：num_class(类别数目)。
- multi: softmax 和multi:softmax参数一样，但是返回的是每个数据属于各个类别的概率。

2. seed (默认None) 和random_state (默认0) :

- 随机数的种子，通常用random_state
- 设置它可以复现随机数据的结果，也可以用于调整参数

下面简单介绍一下XGBoost参数调优的一般方法：

1. 选择较高的学习速率(learning rate)。一般默认的值是0.1，不过针对不同的问题，0.05到0.3之间都可以。选择对应于此学习速率的理想决策树数量。
2. 决定当前learning rate下最优的决定树数量 n_estimators。它的值应该在40-70之间 (根据电脑性能而定)。如果n_estimators过大，可以适当调高learning rate，再调整n_estimators；如果n_estimators过小，可以适当降低learning rate，再调整n_estimators。
3. 对于给定的学习速率和决策树数量，进行决策树特定参数调优。
 - max_depth, min_child_weight
 - gamma
 - subsample, colsample_bytree

4. xgboost的正则化参数的调优: `reg_lambda`, `reg_alpha`。这些参数可以降低模型的复杂度, 从而提高模型的表现。
5. 降低learning rate, 同时会增加 `n_estimators` 使得模型更加稳健

参考文献:

1. Tianqi Chen: Introduction to Boosted Trees
2. wepon: GBDT算法原理与系统设计简介
3. [XGBoost深入浅出](#)
4. [XGBoost 与 Boosted Tree](#)
5. [Complete Guide to Parameter Tuning in XGBoost \(with codes in Python\)](#)
6. [机器学习系列\(12\) XGBoost参数调优完全指南 \(附Python代码\)](#)

4. LightGBM

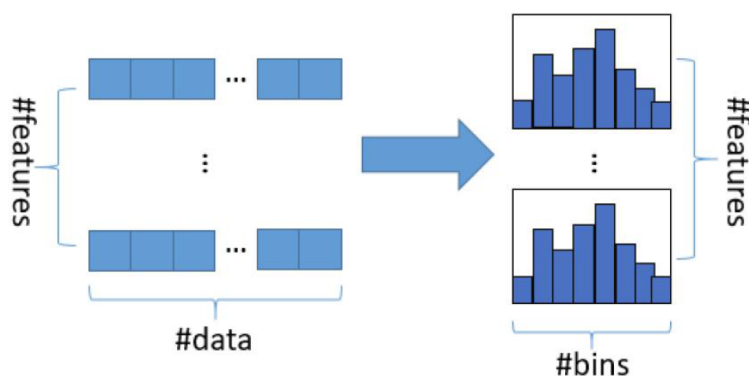
- 速度更快
- 内存占用更低
- 准确率更高 (优势不明显, 与XGBoost相当)

4.1 LightGBM的改进

4.1.1 速度和内存使用的优化

1. 直方图算法

通过将连续特征 (属性) 值分段为 discrete bins 来加快训练的速度并减少内存的使用。



基于 histogram 算法的优点:

- 减少分割增益的计算量: 从 $O(\#data)$ 降到 $O(\#bins)$
- 减少内存的使用
- 减少并行学习的通信代价
- 对于稀疏的特征仅仅需要 $O(2 * \#non_zero_data)$ 来建立直方图

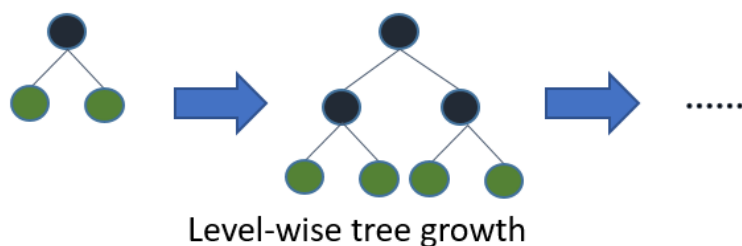
2. 直方图差加速

在二叉树中可以通过利用叶节点的父节点和相邻节点的直方图的相减来获得该叶节点的直方图, 速度可以提升一倍。



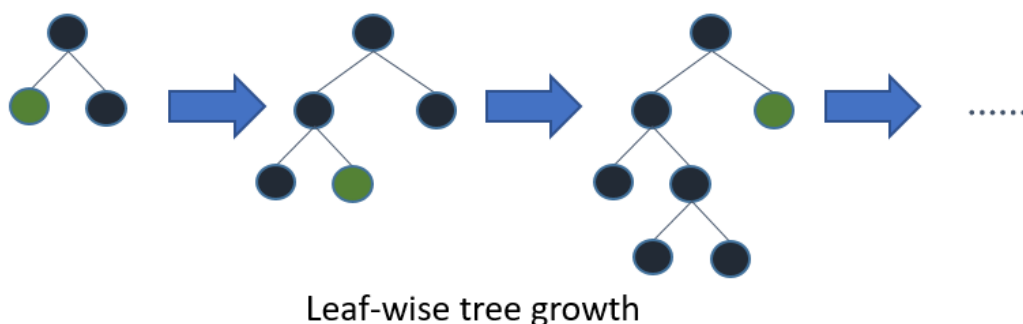
4.1.2 准确率的优化

Leaf-wise (Best-first) 的决策树生长策略



LightGBM 通过 leaf-wise (best-first)策略来生长树。当叶子节点数相同时，leaf-wise 算法可以比 level-wise 算法减少更多的损失。

当样本量较小的时候，leaf-wise 可能会造成过拟合。此时，LightGBM 可以利用额外的参数 `max_depth` 来限制树的深度并避免过拟合。



4.1.3 类别特征值的最优分割

对类别特征最常见的方式就是转化为 one-hot coding。但这样有很大的缺陷，比如容易生成非常不平衡的树，再比如可能一棵树需要很深才能完整地表达某些特征。事实上，最好的解决方案是将类别特征划分为两个子集。基本的思想是根据训练目标的相关性对类别进行重排序。更具体的说，根据 $(\text{sum_gradient} / \text{sum_hessian})$ 重新对直方图进行排序，然后在排好序的直方图中寻找最好的分割点。

4.1.4 网络通信的优化

4.1.5 并行学习的优化：特征并行，数据并行

4.1.6 投票并行

4.2 LightGBM参数解析

```
import lightgbm as lgb

lgb.LGBMClassifier(boosting_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1,
n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None,
min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0,
subsample_freq=1, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None,
n_jobs=-1, silent=True, **kwargs)
```

重要的参数有：

- num_leaves：用来控制树复杂度的一个主要参数
理论上，设置 $num_leaves = 2^{max_depth}$ ，就可以完成 depth-wise tree growth 和 leaf-wise growth 的转换，但这样容易过拟合，因为当这两个参数相等时，leaf-wise tree 的深度要远超 depth-wise tree。因此在调参时，往往会把 num_leaves 的值设置得小于 2^{max_depth} 。事实上，当我们用 leaf-wise tree 时，我们可以忽略 depth 这个概念，毕竟 leaves 跟 depth 之间没有一个确切的关系。
- min_data_in_leaf：这个参数是控制模型过拟合的重要参数
该值受到训练集数量和 num_leaves 的影响。把该参数设的更大能够避免生长出过深的树，但也要避免欠拟合。在分析大型数据集时，该值区间在数百到数千之间较为合适。
- max_depth：这个参数一般不考虑，但是设置它也可以一定程度上降低树的复杂度。

下面简单介绍一下lightgbm参数调优的一般方法：(个人经验)

1. 选择较高的学习速率(learning rate)。一般默认的值是0.1，不过针对不同的问题，0.05到0.3之间都可以。选择对应于此学习速率的理想决策树数量。
2. 决定当前learning rate下最优的决定树数量 n_estimators。它的值应该在40-70之间(根据电脑性能而定)。如果n_estimators过大，可以适当调高learning rate，再调整n_estimators；如果n_estimators过小，可以适当降低learning rate，再调整n_estimators。
3. 对于给定的学习速率和决策树数量，进行决策树特定参数调优。
 - num_leaves, min_data_in_leaf
 - max_depth
 - subsample, colsample_bytree
4. xgboost的正则化参数的调优：reg_alpha, reg_lambda。这些参数可以降低模型的复杂度，从而提高模型的表现。
5. 降低learning rate，同时会增加n_estimators使得模型更加稳健

参考文献：

<http://lightgbm.apachecn.org/cn/latest/Features.html>