

# 机器学习算法系列（8）：XgBoost

---

## 一、XGBoost简介

---

在数据建模中，经常采用Boosting方法通过将成百上千个分类准确率较低的树模型组合起来，成为一个准确率很高的预测模型。这个模型会不断地迭代，每次迭代就生成一颗新的树。但在数据集较复杂的时候，可能需要几千次迭代运算，这将造成巨大的计算瓶颈。

针对这个问题。华盛顿大学的陈天奇博士开发的XGBoost（eXtreme Gradient Boosting）基于C++通过多线程实现了回归树的并行构建，并在原有Gradient Boosting算法基础上加以改进，从而极大地提升了模型训练速度和预测精度。

在Kaggle的希格斯子信号识别竞赛，XGBoost因为出众的效率与较高的预测准确度在比赛论坛中引起了参赛选手的广泛关注，在1700多支队伍的激烈竞争中占有一席之地。随着它在Kaggle社区知名度的提高，最近也有队伍借助XGBoost在比赛中夺得第一。其次，因为它的效果好，计算复杂度不高，也在工业界中有大量的应用。

## 二、监督学习的三要素

---

因为Boosting Tree本身是一种有监督学习算法，要讲Boosting Tree，先从监督学习讲起。在监督学习中几个逻辑上的重要组成部件，粗略地可以分为：模型、参数、目标函数和优化算法。

### 2.1 模型

模型指的是给定输入 $x_i$ 如何去预测输出 $y_i$ 。我们比较常见的模型如线性模型（包括线性回归和Logistic Regression）采用线性加和的方式进行预测

$$\hat{y}_i = \sum_j w_j x_{ij}$$

这里的预测值 $y$ 可以由不同的解释，比如我们可以把它作为回归目标的输出，或者进行sigmoid变换得到概率（即用 $\frac{1}{1+e^{-\hat{y}_i}}$ 来预测正例的概率），或者作为排序的指标等。而一个线性模型根据 $y$ 的解释不通（以及设计对应的目标函数）用到回归、分类或者排序等场景。

### 2.2 参数

参数就是我们根据模型要从数据里头学习的东西，比如线性模型中的线性系数：

$$\Theta = \{w_j | j = 1, 2, \dots, d\}$$

## 2.3 目标函数：误差函数+正则化项

模型和参数本身指定了给定输入我们如何预测，但是没有告诉我们如何去寻找一个比较好的参数，这个时候就需要目标函数函数登场了。一般地目标函数包含两项：一项是损失函数，它说明了我们的模型有多拟合数据；另一项是正则化项，它惩罚了复杂模型。

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

**Training Loss** measures how well model fit on training data

**Regularization**, measures complexity of model

- 1)  $L(\Theta)$ : 损失函数  $L = \sum_{i=1}^n l(y_i, \hat{y}_i)$ ，常见的损失函数有：
  - 平方损失:  $l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$
  - Logistic损失:  $l(y_i, \hat{y}_i) = y_i \ln(1 + e^{-y_i}) + (1 - y_i) \ln(1 + e^{y_i})$
- 2)  $\Omega(\Theta)$ : 正则化项，之所以要引入它是因为我们的目标是希望生成的模型能准确地预测新的样本（即应用于测试数据集），而不是简单地拟合训练集的结果（这样会导致过拟合）。所以需要在保证模型“简单”的基础上最小化训练误差，这样得到的参数才具有好的泛化性能。而正则化项就是用于惩罚复杂模型，避免模型过分拟合训练数据。常用的正则项有L1正则与L2正则
  - L1正则 (lasso) :  $\Omega(w) = \lambda ||w||_1$
  - L2正则:  $\Omega(w) = \lambda ||w||^2$

这样目标函数的设计来自于统计学习里面的一个重要概念叫做Bias-variance tradeoff（偏差-方差权衡），比较感性的理解，*Bias*可以理解为假设我们有无限多数据的时候，可以训练出最好的模型所拿到的误差。而*Variance*是因为我们只有有限数据，其中随机性带来的误差。目标中误差函数鼓励我们的模型尽量去拟合训练数据，这样相对来说最后的模型会有比较少的*Bias*。而正则化项则鼓励更加简单的模型。因为当模型简单之后，有限数据拟合出来结果的随机性比较小，不容易过拟合，使得最后模型的预测更加稳定。

## 2.4 优化算法

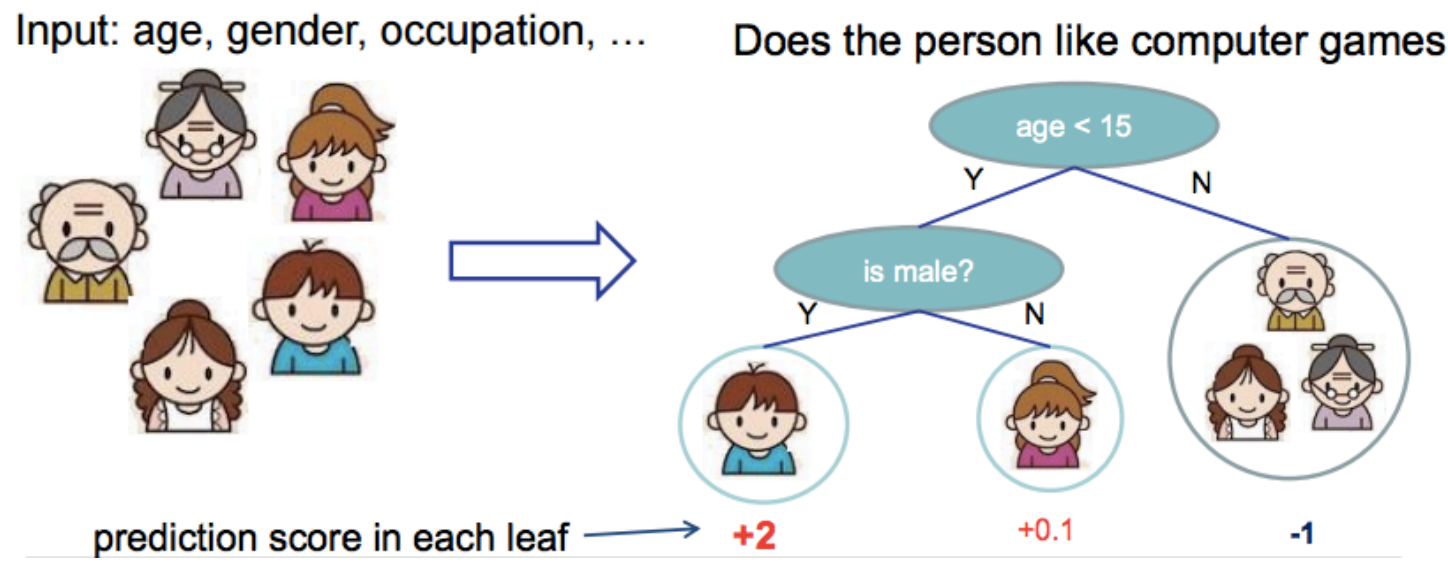
上面三个部分包含了机器学习的主要成分，也是机器学习工具划分模型比较有效的办法。其实这几部分之外，还有一个优化算法，就是给定目标函数之后怎么学的问题。有时候我们往往只知道“优化算法”，而没有仔细考虑目标函数的设计问题，比如常见的例子如决策树的学习算法的每

一步去优化基尼系数，然后剪枝，但是没有考虑到后面的目标是什么。而这些启发式优化方法背后往往隐含了一个目标函数，理解了目标函数本身也有利于我们设计相应的学习算法。

## 三、回归树与树集成

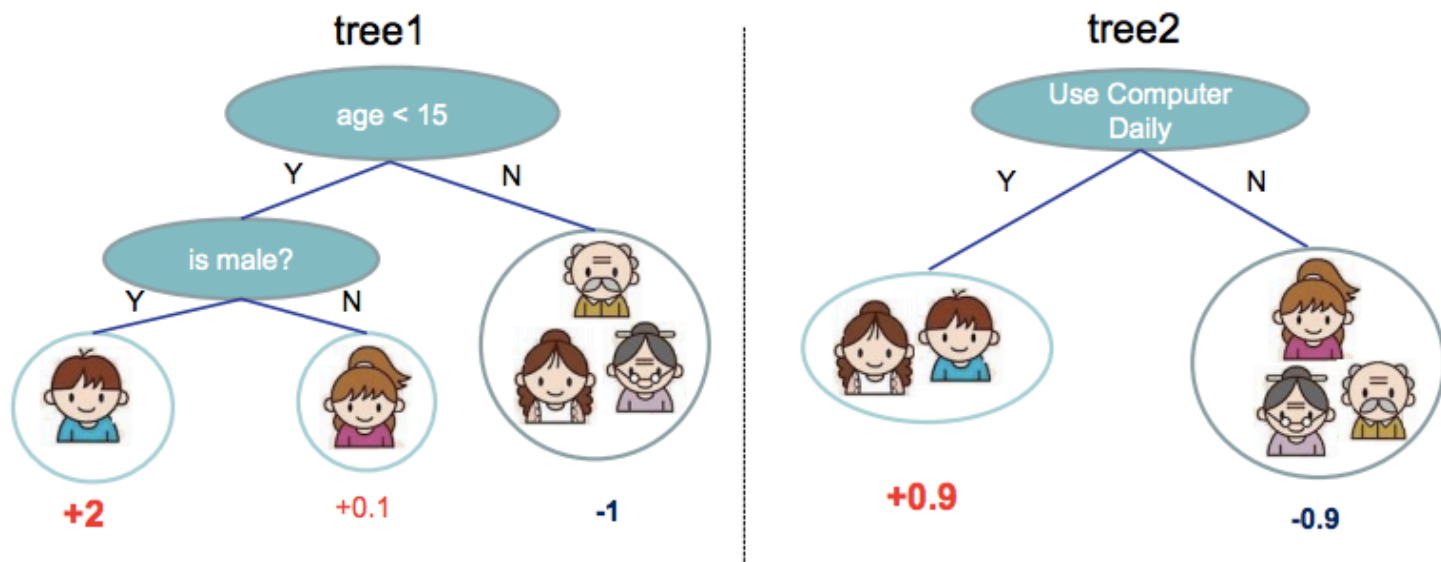
### 3.1 回归树

在介绍*XGBoost*之前，首先得了解一下回归树和树集成的概念，其实在*AdaBoost*算法中已经详细讲述过这一部分了。Boosting Tree最基本的组成部分叫做回归树（regression tree），下面就是一个回归树的例子。它把输入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有一个实数分数。具体地，下图给出了一个判断用户是否会喜欢电脑游戏的回归树模型，每个树叶的得分对应了该用户有多可能喜欢电脑游戏（分值越大可能性越大）。



### 3.2 树集成

上图中的回归树只用到了用户年龄和性别两个信息，过于简单，预测的准确性自然有限。一个回归树往往过于简单无法有效地预测，因此一个更加强有力的模型叫做tree ensemble。



在上图中使用两个回归树对用户是否喜欢电脑游戏进行了预测，并将两个回归树的预测结果加和得到单个用户的预测结果。在实际的预测模型建立过程中，我们通过不断地增加新的回归树，并给每个回归树赋予合适的权重，在此基础上综合不同的回归树得分获得更为准确的预测结果，这也就是树集成的基本思路。在预测算法中，随机森林和提升树都采用了树集成的方法，但是在具体地模型构造和参数调整的方法有所差别。

在这个树集成模型中，我们可以认为参数对应了树的结构，以及每个叶子节点上面的预测分数。

那么我们如何来学习这些参数。在这一部分，答案可能千奇百怪，但是最标准的答案始终是一个：定义合理的目标函数，然后去尝试优化这个目标函数。决策树学习往往充满了启发式算法，如先优化基尼系数，然后再剪枝，限制最大深度等等。其实这些启发式算法背后往往隐含了一个目标函数，而理解目标函数本身也有利于我们设计学习算法。

## 四、XGBoost的推导过程

### 4.1 XGBoost的目标函数与泰勒展开

对于tree ensemble，我们可以把某一个迭代后集成的模型写成为：

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

其中每个 $f$ 是一个在函数空间( $F$ )里面的函数，而 $F$ 对应了所有regression tree的集合。我们设计的目标函数也需要遵循前面的主要原则，包含两部分

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

其中第一部分是训练损失，如上面所述的平方损失或者Logistic Loss等，第二部分是每棵树的复杂度的和。因为现在我们的参数可以认为是在一个函数空间里面，我们不能采用传统的如SGD之类的算法来学习我们的模型，因此我们会采用一种叫做additive training的方式。即每次迭代生成一棵新的回归树，从而使预测值不断逼近真实值（即进一步最小化目标函数）。每一次保留原来的模型不变，加入一个新的函数到模型里面：

$$\begin{aligned}\hat{y}_i^{(0)} &= 0 \\ \hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\ \hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\ &\dots \\ \hat{y}_i^{(t)} &= \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i) \leftarrow \text{New function}\end{aligned}$$

**Model at training round t**      **Keep functions added in previous round**

其中 $\hat{y}_i^{(t-1)}$ 就是前 $t-1$ 轮的模型预测， $f_t(x_i)$ 为新 $t$ 轮加入的预测函数。

这里自然就涉及一个问题：如何选择在每一轮中加入的 $f(x_i)$ 呢？答案很直接，选取的 $f(x_i)$ 必须使得我们的目标函数尽量最大地降低（这里应用到了Boosting的基本思想，即当前的基学习器重点关注以前所有学习器犯错误的那些数据样本，以此来达到提升的效果）。先对目标函数进行改写，表示如下：

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

**Goal: find  $f_t$  to minimize this**

如果我们考虑平方误差作为损失函数，公式可改写为：

$$\begin{aligned}Obj^{(t)} &= \sum_{i=1}^n \left( y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)) \right)^2 + \Omega(f_t) + const \\ &= \sum_{i=1}^n \left[ 2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2 \right] + \Omega(f_t) + const\end{aligned}$$

**This is usually called residual from previous round**

更加一般的，对于不是平方误差的情况，我们可以采用如下的泰勒展开近似来定义一个近似的目标函数，方便我们进行下一步的计算。

泰勒展开一般表达式为：

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

用泰勒展开来近似我们原来的目标：首先定义

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

得到

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

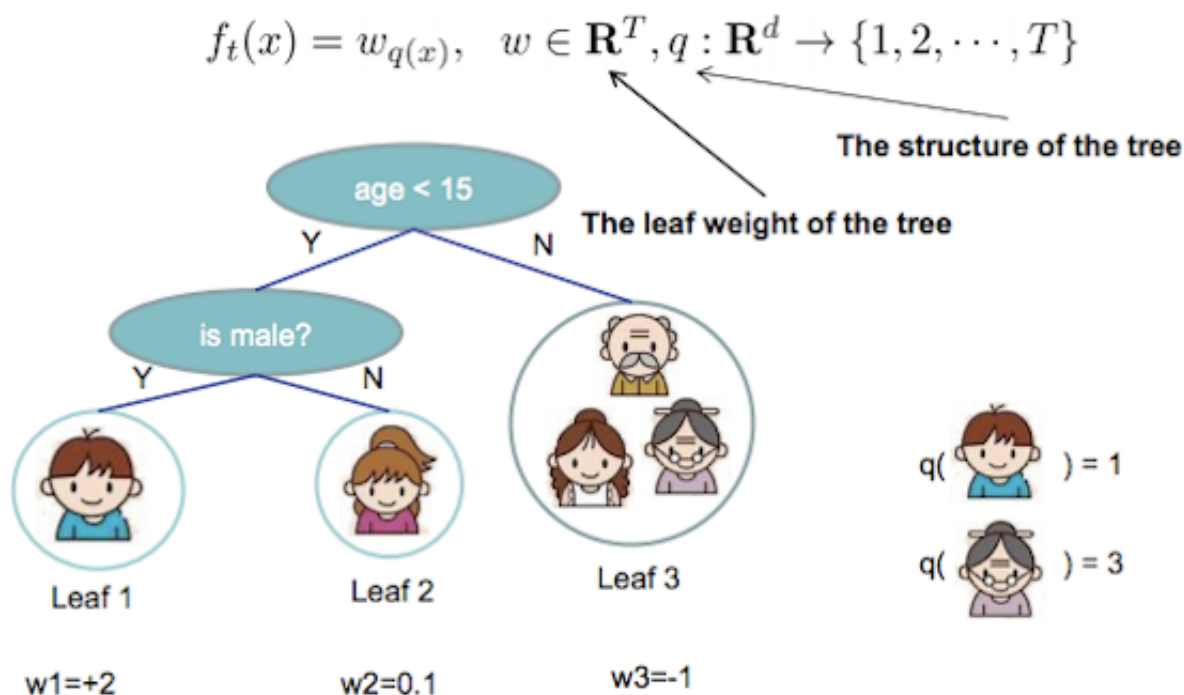
如果移除掉常数项，我们会发现这个目标函数有一个非常明显的特点，它只依赖于每个数据点的在误差函数上的一阶导数和二阶导数。可能有人会问，这个方式似乎比我们之前学过的决策树学习难懂。为什么要花这么多力气来做推导呢？

这是因为，这样做首先有理论上的好处，它会使得我们可以很清楚地理解整个目标是什么，并且一步一步推导出如何进行树的学习。然后这一个抽象的形式对于工程商实现机器学习工具也是非常有帮助的。因为它包含所有可以求到的目标函数，也就是说有了这个形式，我们写出来的代码可以用来求解包括回归、分类和排序的各种问题，正式的推导可以使得机器学习的工具更加一般化。

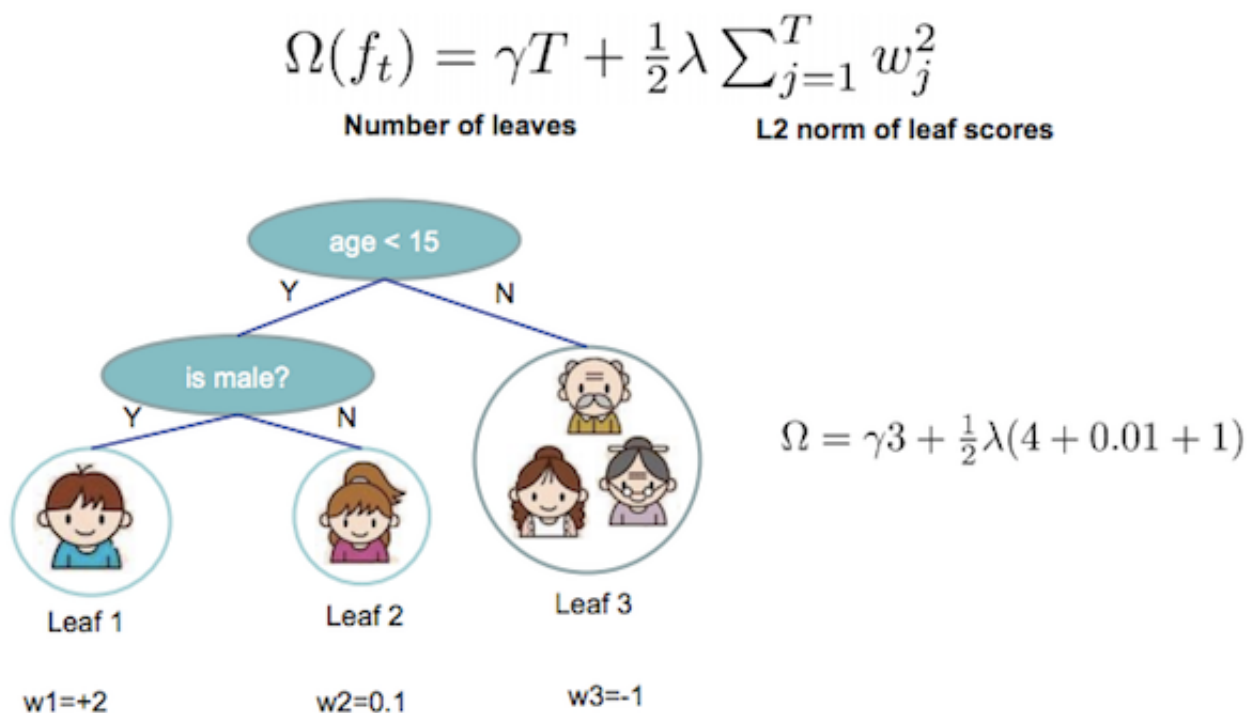
## 4.2 决策树的复杂度

到目前为止我们讨论了目标函数中训练误差的部分。接下来我们讨论如何定义树的复杂度。我们先对于 $f$ 的定义做一下细化，把树拆分成结构部分 $q$ 和叶子权重部分 $w$ 。其中结构部分 $q$ 把输入映射到叶子的索引号上面去，而 $w$ 给定了每个索引号对应的叶子分数是什么。





当我们给定了如上定义之后，我们可以定义一棵树的复杂度如下。这个复杂度包含了一棵树里面节点的个数，以及每个树叶子节点上面输出分数的 $L_2$ 范数平方。当然这不是唯一的一种定义方式，不过这一定义方式学习出的树效果一般都比较不错。下图给出了复杂度计算的一个例子。



## 4.3 目标函数的最小化

接下来是最关键的一步，在这种新的顶一下，我们可以把目标函数进行如下改写，其中 $I$ 被定义为每个叶子上面样本集合 $I_j = \{i \mid q(x_i) = j\}$

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

这一目标包含了 $T$ 个互相独立的单变量二次函数

$$\operatorname{argmin}_x Gx + \frac{1}{2} Hx^2 = -\frac{G}{H}, \quad H > 0 \quad \min_x Gx + \frac{1}{2} Hx^2 = -\frac{1}{2} \frac{G^2}{H}$$

我们可以定义


$$G_j = \sum_{i \in I_j} g_i \quad H_j = \sum_{i \in I_j} h_i$$

那么这个目标函数可以进一步改写成如下的形式，假设我们已经知道树的结构 $q$ ，我们可以通过这个目标函数来求解出最好的 $w$ ，以及最好的 $w$ 对应的目标函数最大的增益

$$\begin{aligned}
Obj^{(t)} &= \sum_{j=1}^T \left[ \left( \sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left( \sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T \\
&= \sum_{j=1}^T \left[ G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma T
\end{aligned}$$

可以观察到上式是由 $T$ 个相互独立的单变量二次函数再加上 $L1$ 范数构成。这样的特性意味着单个树叶的权重计算与其他树叶的权重无关，所以我们可以非常方便计算第 $j$ 个树叶的权重，以及目标函数。由此，我们将目标函数转换为一个一元二次方程求最小值的问题（在此式中，变量为 $w_j$ ，函数本质上是关于 $w_j$ 的二次函数），略去求解步骤，最终结果如下所示：

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

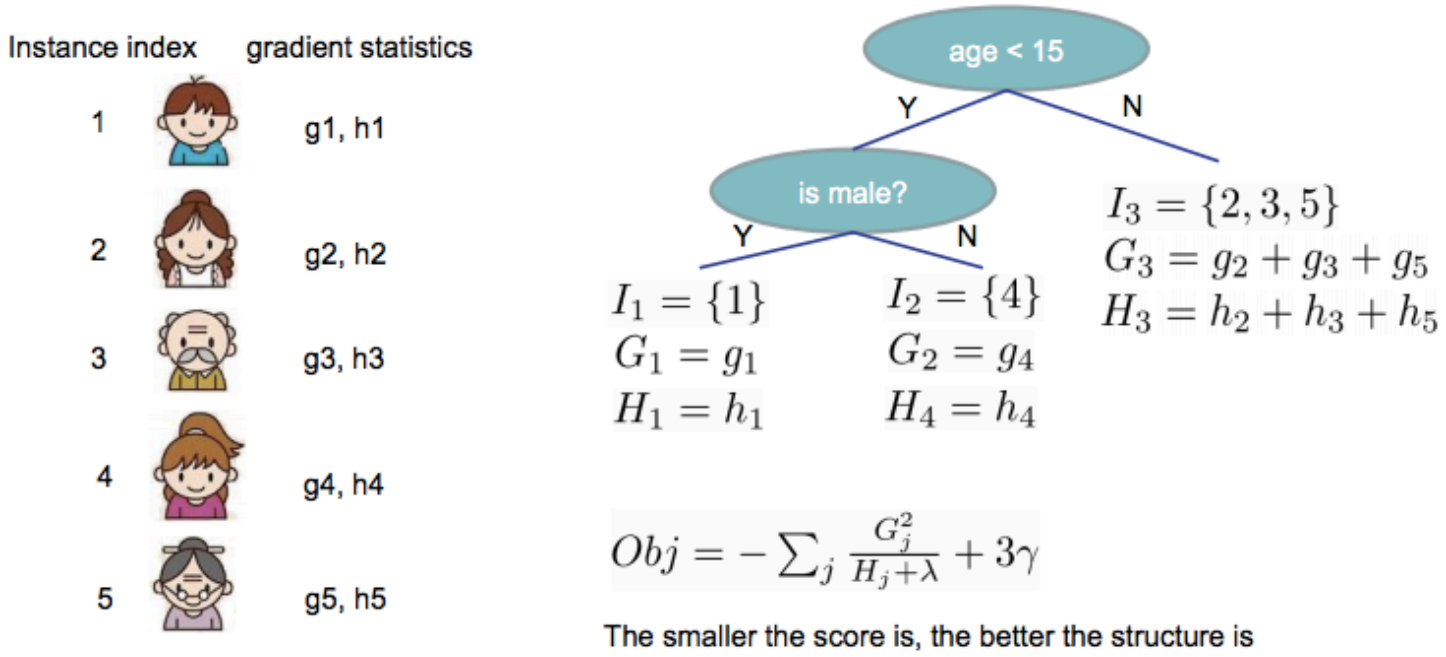

**This measures how good a tree structure is!**

乍一看目标函数的计算与回归树的结构 $q$ 函数没有什么关系，但是如果我们仔细回看目标函数的构成，就会发现其中 $G_j$ 和 $H_j$ 的取值是由第 $j$ 个树叶上数据样本所决定的。而第 $j$ 个树上所具有的数据样本则是由树结构 $q$ 函数决定的。也就是说，一旦回归树的结构 $q$ 确定，那么相应的目标函数就能够根据上式计算出来。那么回归树的生成问题也就转换为找到一个最优的树结构 $q$ ，使得它具有最小的目标函数。

计算求得的 $Obj$ 代表了当指定一个树的结构的时候，目标函数上面最多减少多少。我们可以把它叫做结构分数（structure score）。可以把它认为是类似于基尼系数一样更加一般的对于树结构

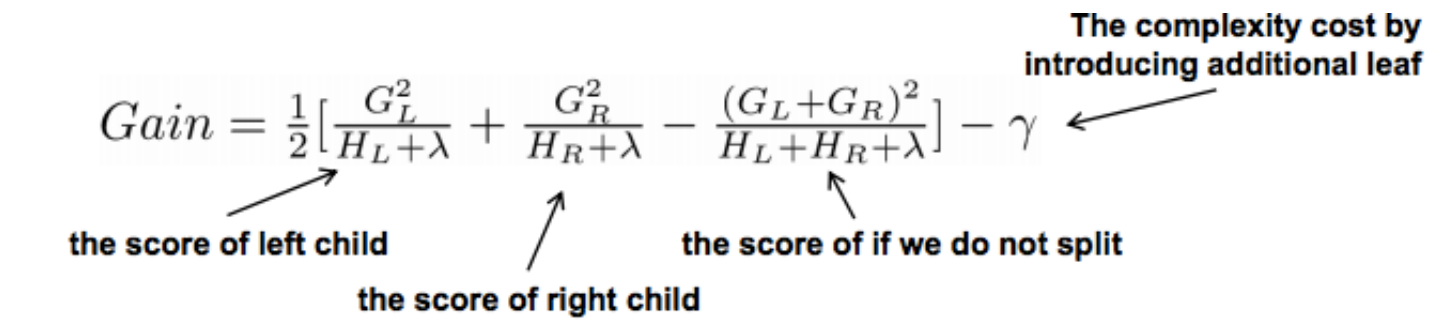


进行打分的函数。下面是一个具体的打分函数计算的例子，它根据决策树的预测结果得到各样本的梯度数据，然后计算出实际的结构分数。这个分数越小，代表这个树的结构越好：

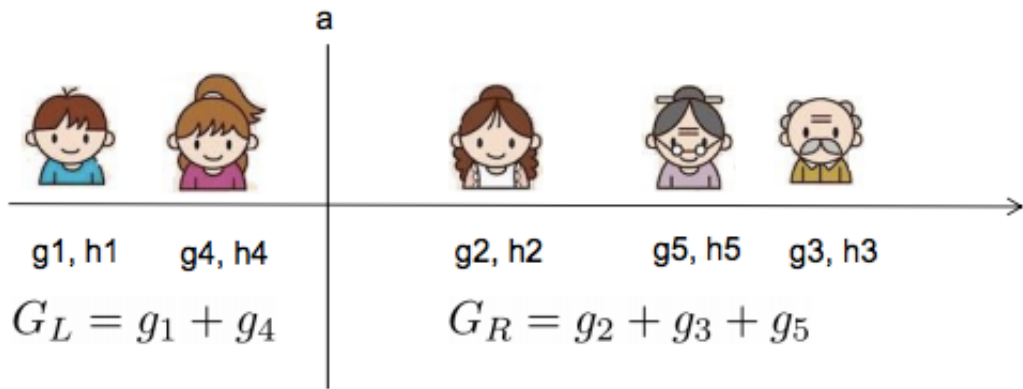


## 4.4 枚举树的结果——贪心法

在前面分析的基础上，当寻找到最优的树结构时，我们可以不断地枚举不同树的结构，利用这个打分函数来寻找一个最优结构的树，加入到我们的模型中，然后再重复这样的操作。不过枚举所有树结构这个操作不太可行，在这里XGBoost采用了常用的贪心法，即每一次尝试区队已有的叶子加入一个分割。对于一个剧透的分割方案，我们可以获得的增益可以由如下公式计算得到：



需要枚举所有 $x < a$ 这样的条件，那么对于某个特定的分割 $a$ 我们要计算 $a$ 左边和右边的导数和，在实际应用中如下图所示：



我们可以发现对于所有的 $a$ ，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与 $G_L$ 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。

但需要注意的是：引入的分割不一定会使得情况变好，因为在引入分割的同时也引入新叶子的惩罚项。所以通常需要设定一个阈值，如果引入的分割带来的增益小于一个阈值的时候，我们可以剪掉这个分割。此外在XGBoost的具体实践中，通常会设置树的深度来控制树的复杂度，避免单个树过于复杂带来的过拟合问题。

以上介绍了如何通过目标函数优化的方法比较严格地推导出boosted tree的学习的整个过程。因为有这样一般的推导，得到的算法可以直接应用到回归，分类排序各个应用场景中去。

## 五、QA

### 5.1 机器学习算法中GBDT和XGBOOST的区别有哪些？

- **基分类器的选择**：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）。
- **二阶泰勒展开**：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工具支持自定义损失函数，只要函数可一阶和二阶求导。
- **方差-方差权衡**：XGBoost在目标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 $T$ 、每个叶子节点上输出分数的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 $\epsilon$ )。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）
  - **Add  $f_t(x)$  to the model**  $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$ 
    - Usually, instead we do  $y^{(t)} = y^{(t-1)} + \epsilon f_t(x_i)$
    - $\epsilon$  is called step-size or shrinkage, usually set around 0.1
    - This means we do not do full optimization in each step and reserve chance for future rounds, it helps prevent overfitting
- **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。
- **缺失值处理**：XGBoost考虑了训练数据为稀疏值的情况，可以为缺失值或者指定的值指定分支的默认方向，这能大大提升算法的效率，paper提到50倍。即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 $t$ 次迭代的损失函数里包含了前面 $t-1$ 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(块)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。
- **线程缓冲区存储**：按照特征列方式存储能优化寻找最佳的分割点，但是当以行计算梯度数据时会导致内存的不连续访问，严重时会导致cache miss，降低算法效率。paper中提到，可先将数据收集到线程内部的buffer（缓冲区），主要是结合多线程、数据压缩、分片的方法，然后再计算，提高算法的效率。
- **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。大致的思想是根据百分位法列举几个可能成为分割点的候选者，然后从候选者中根据上面求分割点的公式计算找出最佳的分割点。

## 5.2 为什么在实际的 kaggle 比赛中 gbd 和 random forest 效果非常好？

转载自[知乎](#)

这是一个非常好，也非常值得思考的问题。换一个方式来问这个问题：为什么基于 tree-ensemble 的机器学习方法，在实际的 kaggle 比赛中效果非常好？

通常，解释一个机器学习模型的表现是一件很复杂事情，而这篇文章尽可能用最直观的方式来解释这一问题。

我主要从三个方面来回答楼主这个问题。

- 1. 理论模型（站在 vc-dimension 的角度）
- 2. 实际数据
- 3. 系统的实现（主要基于 xgboost）

通常决定一个机器学习模型能不能取得好的效果，以上三个方面的因素缺一不可。

### 5.2.1 站在理论模型的角度

统计机器学习里经典的 vc-dimension 理论告诉我们：一个机器学习模型想要取得好的效果，这个模型需要满足以下两个条件：

1. 模型在我们的训练数据上的表现要不错，也就是 training error 要足够小。
2. 模型的 vc-dimension 要低。换句话说，就是模型的自由度不能太大，以防 overfit.

当然，这是我用大白话描述出来的，真正的 vc-dimension 理论需要经过复杂的数学推导，推出 vc-bound. vc-dimension 理论其实是从另一个角度刻画了一个我们所熟知的概念，那就是 bias variance trade-off.

好，现在开始让我们想象一个机器学习任务。对于这个任务，一定会有一个“上帝函数”可以完美的拟合所有数据（包括训练数据，以及未知的测试数据）。很可惜，这个函数我们肯定是不知道的（不然就不需要机器学习了）。我们只可能选择一个“假想函数”来逼近这个“上帝函数”，我们通常把这个“假想函数”叫做 hypothesis.

在这些 hypothesis 里，我们可以选择 svm, 也可以选择 logistic regression. 可以选择单棵决策树，也可以选择 tree-ensemble (gbd, random forest). 现在的问题就是，为什么 tree-ensemble 在实际中的效果很好呢？

区别就在于“模型的可控性”。先说结论，tree-ensemble 这样的模型的可控性是好的，而像 LR 这样的模型的可控性是不够好的（或者说，可控性是没有 tree-ensemble 好的）。为什么会这样？别急，听我慢慢道来。

我们之前说，当我们选择一个 hypothesis 后，就需要在训练数据上进行训练，从而逼近我们的“上帝函数”。我们都知道，对于 LR 这样的模型。如果 underfit，我们可以通过加 feature，或者通过高次的特征转换来使得我们的模型在训练数据上取得足够高的正确率。而对于 tree-ensemble 来说，我们解决这一问题的方法是通过训练更多的“弱弱”的 tree。所以，这两类模型都可以把 training error 做的足够低，也就是说模型的表达能力都是足够的。但是这样就完事了吗？没有，我们还需要让我们的模型的 vc-dimension 低一些。而这里，重点来了。在 tree-ensemble 模型中，通过加 tree 的方式，对于模型的 vc-dimension 的改变是比较小的。而在 LR 中，初始的维数设定，或者说特征的高次转换对于 vc-dimension 的影响都是更大的。换句话说，tree-ensemble 总是用一些“弱弱”的树联合起来去逼近“上帝函数”，一次一小步，总能拟合的比较好。而对于 LR 这样的模型，我们很难去猜到这个“上帝函数”到底长什么样子（到底是2次函数还是3次函数？上帝函数如果是介于2次和3次之间怎么办呢？）。所以，一不小心我们设定的多项式维数高了，模型就“刹不住车了”。俗话说的好，步子大了，总会扯着蛋。这也就是我们之前说的，tree-ensemble 模型的可控性更好，也即更不容易 overfitting。

## 5.2.2 站在数据的角度

除了理论模型之外，实际的数据也对我们的算法最终能取得好的效果息息相关。kaggle 比赛选择的都是真实世界中的问题。所以数据多多少少都是有噪音的。而基于树的算法通常抗噪能力更强。比如在树模型中，我们很容易对缺失值进行处理。除此之外，基于树的模型对于 categorical feature 也更加友好。

除了数据噪音之外，feature 的多样性也是 tree-ensemble 模型能够取得更好效果的原因之一。通常在一个kaggle任务中，我们可能有年龄特征，收入特征，性别特征等等从不同 channel 获得的特征。而特征的多样性也正是为什么工业界很少去使用 svm 的一个重要原因之一，因为 svm 本质上是属于一个几何模型，这个模型需要去定义 instance 之间的 kernel 或者 similarity（对于 linear svm 来说，这个similarity 就是内积）。这其实和我们在之前说过的问题是相似的，我们无法预先设定一个很好的similarity。这样的数学模型使得 svm 更适合去处理“同性质”的特征，例如图像特征提取中的 lbp。而从不同 channel 中来的 feature 则更适合 tree-based model, 这些模型对数据的 distribution 通常并不敏感。

## 5.2.3 站在系统实现的角度

除了有合适的模型和数据，一个良好的机器学习系统实现往往也是算法最终能否取得好的效果的关键。一个好的机器学习系统实现应该具备以下特征：

- 1. 正确高效的实现某种模型。我真的见过有些机器学习的库实现某种算法是错误的。而高效的实现意味着可以快速验证不同的模型和参数。
- 2. 系统具有灵活、深度的定制功能。
- 3. 系统简单易用。
- 4. 系统具有可扩展性, 可以从容处理更大的数据。

到目前为止，xgboost 是我发现的唯一一个能够很好的满足上述所有要求的 machine learning package. 在此感谢青年才俊 陈天奇。

在效率方面，xgboost 高效的 c++ 实现能够通常能够比其它机器学习库更快的完成训练任务。在灵活性方面，xgboost 可以深度定制每一个子分类器，并且可以灵活的选择 loss function (logistic, linear, softmax 等等)。除此之外，xgboost还提供了一系列在机器学习比赛中十分有用的功能，例如 early-stop, cv 等等

在易用性方面，xgboost 提供了各种语言的封装，使得不同语言的用户都可以使用这个优秀的系统。

最后，在可扩展性方面，xgboost 提供了分布式训练（底层采用 rabbit 接口），并且其分布式版本可以跑在各种平台之上，例如 mpi, yarn, spark 等等。

有了这么多优秀的特性，自然这个系统会吸引更多的人去使用它来参加 kaggle 比赛。

综上所述，理论模型，实际的数据，良好的系统实现，都是使得 tree-ensemble 在实际的 kaggle 比赛中“屡战屡胜”的原因