

目录

决策树的理论和应用..... 3

1.1 机器学习中的决策树 3

1.2 XGBoost 的基石 – 回归树 5

1.3 集成树模型-随机森林 6

1.4 分裂条件选取..... 8

 I. Gini Impurity.....8

 II. Information Gain.....10

 III. Tree Ensemble Gain.....11

GBDT (Gradient Boosted Decision Trees)..... 15

2.1 GBDT 概念 15

2.2 GBDT 分析 17

2.3 运用 GBDT 进行回归或二分类..... 18

XGBoost..... 19

3.1 XGBoost 概念 19

 降低预测精确性+ 缩短训练时间19

 保持预测精确性+ 缩短训练时间22

 提高模型应用性.....23

 提升预测效果24

3.2	XGBoost 调参	25
	模型通用的调参	25
	XGBoost 调参	33
3.3	XGBoost 的局限性	34
3.4	XGBoost 与其他模型的对比	35
	随机森林	35
	Light GBM	35
	Catboost	36
	<i>附录</i>	<i>38</i>
4.1	逻辑回归	38
4.2	回归树和分类树的区别	39
4.3	XGBoost 超参数列表	43
	<i>参考资料</i>	<i>43</i>

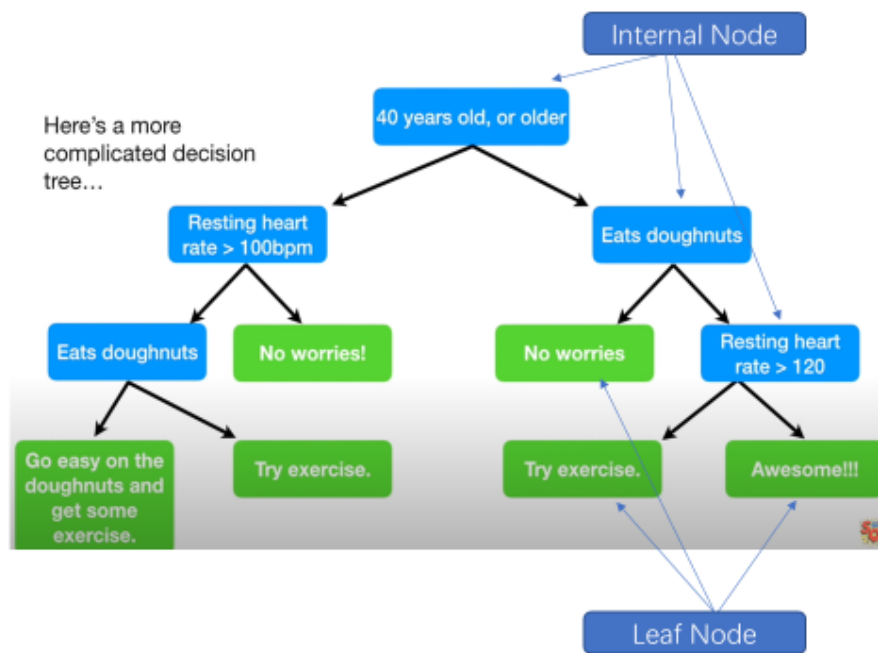
前言

说到机器学习，人们通常会想到聚类，回归问题，以及常用的解决方法，如线性回归，神经网络等。而相比较下，决策树可能没有那么被大众所熟知，更不会想到决策树模型是某些问题的最佳选择。今天我们要讲解的基于决策树的 **XGBoost 是目前业界广泛运用，效果最佳最稳定的结构化有监督学习算法之一，并常年霸占 Kaggle 比赛冠军席位。它有着训练迅速，AUC 高，支持多种语言 (C++, Java, Python, R, Julia, Perl, and Scala) 和系统 (Linux, Windows, and macOS)，且模型易于理解的特性。** 本文会从数据结构和算法方面来深入的了解一下如此强大的 XGBoost 是如何实现的，并且尽量做到没有基础的同学能看懂，而了解机器学习的同学能对 XGBoost 背后的算法有更深入的了解~

决策树的理论和应用

相信大家生活中或多或少都见过一些决策树。在对产品，生活和规划做决策时，决策树能很好的总结归纳我们的逻辑并输出一个相对粗略但实用的结果。决策树的这种特性也被机器学习领域发掘，形成了规范化的决策树。

1.1 机器学习中的决策树



决策树范例

构造：

决策树的节点分成两种：带有分裂条件的分裂节点 (Internal Node) 和存储预测

结果的叶子节点(leaf node)。内部节点的分裂条件由两部分组成:feature 和 limit,

例如 $\text{age} \geq 40$, $\text{color} = \text{red}$ 。而叶子节点中的预测值可能是数值或类别。

构建方法：

通常建造决策树时我们会尝试不同的分裂条件去分裂训练集中的数据，并选取其中

效果最优的作为分裂节点的条件，再对分裂后生成的两个孩子节点重复同样的步骤。

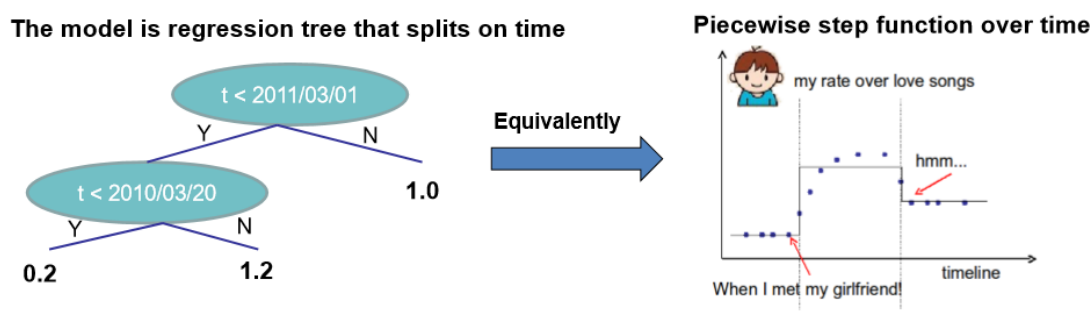
结果预测：

对于预测集的每条数据，从根节点的条件开始判断，如果符合条件，则进入左子树

继续预测，如不符合，进入右子树，重复此步骤直到进入叶子节点得到预测结果。

决策树本质：

由于每个分裂节点把节点中的数据一分为二，决策树本质上就是一个 piecewise step function。下图为单一特征 ($R \rightarrow R$) 的范例。当我们对 K 个特征进行学习时，决策树可以转化为 $R^K \rightarrow R$ 的 step function, 且每个 piece 的定义域都是一个 K -cell.



通过上图可以发现,我们的目的是输出一个相对简单(低 variance)且准确(低 bias, 贴合输入数据)的决策树 / step function。然而对于单个决策树模型, 由于分裂条件的选取高度依赖于训练集的数据分布, 很容易使模型过拟合, 从而使得预测效果差。为了解决这个问题, 人类发明了集成树 (Tree Ensemble), 会在 1.3 讲解。

1.2 XGBoost 的基石 – 回归树

XGBoost 算法用到了一种特殊的决策树 – 回归树 (Regression Tree)。它隶属于 CART (Classification and Regression Tree) (通常两者分别用于分类和回归问题)。而 XGBoost 使用回归树同时应对回归和分类问题, 提升了结果的准确性, 并通过 logistic function (二分类) + softmax (多分类)把预测值转化为 $[0,1]$ 之间的数来达到分类的效果并且把回归树的优势传递到了分类问题上。

下面我们来讲解一下回归树的构造方法：

1. 对于每个 feature 选取所有候选分割点进行分裂效果的计算 (计算方式详见

1.4), 选取效果最好的作为分裂条件, 选取候选分割点的方法如下:

- 数值特征 $x_i \rightarrow x_i$ 里所有相邻数值的均值

- (例: $x_i = \{1, 4, 2, 6, 4\} \Rightarrow$

$$\text{候选值} = \{(1+2)/2, (2+4)/2, (4+6)/2\} = \{1.5, 3, 5\}$$

- 类别特征 $x_j \rightarrow \text{one-hot encoding}$ 把特征分成 $[x_1, x_2, \dots, x_n]$ 并依次取

$x_k > 0.5$ 作为分割点 (较高效, 有信息损失) 或者取特征种类的所有排列组

合 (低效)

- (例: $x_i = \{\text{red}, \text{green}, \text{red}, \text{blue}\} \Rightarrow$

$$x_1 = \text{red}, x_2 = \text{green}, x_3 = \text{blue}, (\text{range } 0-1)$$

$$\text{候选值} = \{x_1 > 0.5, x_2 > 0.5, x_3 > 0.5\}$$

2. 对于分割出的两个组重复 step 1, 直到判定不应该再细分 (判定条件参见 1.4)

我们可以发现回归树的分裂条件永远是对每个组最优, 而不是全局最优。 简单地

说, 如果说预测效果最好的模型的树集合为 S , 回归树并不考虑当前最优的分裂条件

是否属于对应 S 里树的条件。这种贪心算法大大提升了 XGBoost 模型的训练速度。

1.3 集成树模型-随机森林

集成树模型是指通过多个决策树进行训练/预测的模型, 而随机森林是集成树模型最

简单的一种。因为 XGBoost 和随机森林一样属于集成树模型, 我想通过随机森林介

绍集成树模型的优势并引出 XGBoost 运用多个树的原因。

随机森林的建造方法运用到了 bagging:

在建造每个树时, 随机建立 bootstrapped 数据集 (大小和训练集相同, 但每条数据都是均匀随机地从训练集中选取)。 并在分割每一个节点时, 从特征中随机选取 m 个特征作为分裂条件的选取范围。在判定不应该再细分节点时停止分割。(判定方法详见 1.4)

随机森林通过随机限制分割条件的范围, 使得每个分割节点的条件有可能并不是当前最优, 而通过 bootstrapped 数据集把一些数据排除在外, 改变了一些数据的比例, 使得数据集分布区别于训练集。这两者都降低了模型受训练集数据分布的影响。

很好, 那我们为什么要建立多个决策树呢? 我们发现, 如果随机森林算法只用一个树作为模型, 则会让模型过于拟合唯一的一个 bootstrapped 数据集, 并且因为随机选取的特征造成结果偏差 (bias)。所以通过多个这样的决策树可以使得数据和特征的随机性和造成的 variance 被中和。

随机森林预测:

对于分类树: 对于一条数据 x_i , 用每颗树对其进行预测得到 K 条结果, 返回结果数最多的种类。

对于回归树: 对于一条数据 x_i , 用每颗树对其进行预测得到 K 条结果, 返回结果平均数。如果把 n 条结果标记成 $[g_1(x_i), g_2(x_i), \dots, g_K(x_i)]$, 可以得到:

预测值 y_i

$$= (g_1(x_i) + g_2(x_i) \dots + g_K(x_i)) / K$$

$$= g_1(x_i) / K + g_2(x_i) / K \dots + g_K(x_i) / K$$

定义 $f_k(x_i) = g_k(x_i) / K$

能得出预测值的公式

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

除了上面所说的，随机森林还可以补充缺失值。使用的方法包括 proximity matrix, 中位数（数值特征），众数（类别特征）等，这里就不一一介绍了，可以看这个视频了解。

https://www.youtube.com/watch?v=nyxTdL_4Q-Q

1.4 分裂条件选取

在构造决策树时，我们需要选取分裂节点的分裂条件，常用的有以下几个方案

I. Gini Impurity

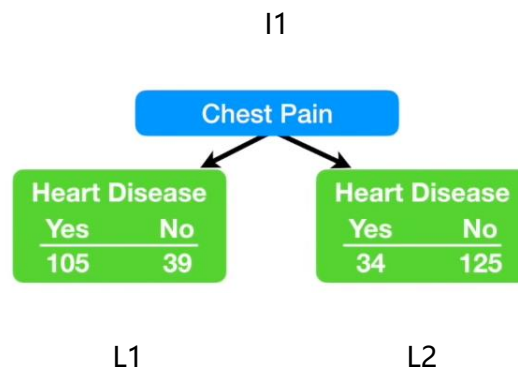
当我们通过决策树对训练集里的每条数据进行归类后，如果叶子节点里包含了结果值不同的数据，我们称叶子节点中的数据是不纯的。Gini impurity 是一种衡量单个节点内数据不纯度的方式，代表了一条随机的样本被错误预测的概率。因此，Gini impurity 越低，预测效果越好。

单个节点 G 的 Gini Impurity 的计算公式， p_i 表示节点内第 i 种类别的比例：

$$I_G(p) = \sum p_i * (1 - p_i) = 1 - \sum p_i^2$$

对于 Gini Impurity 的通俗解释：一个随机样例属于第 i 种类型的概率是 p_i ，而预测正确的概率也是 p_i ，所以样本真实值使 p_i 且预测正确的概率为 p_i^2 ，所以每个随机样本被预测正确的概率是 $\sum p_i^2$ ，错误的概率是 $1 - \sum p_i^2$

范例：



叶子节点的 Gini Impurity:

$$I_G(L1) = 1 - \left(\frac{105}{144}\right)^2 - \left(\frac{39}{144}\right)^2 = 0.395$$

$$I_G(L2) = 1 - \left(\frac{125}{159}\right)^2 - \left(\frac{34}{159}\right)^2 = 0.336$$

分割前分裂节点的 Gini Impurity:

$$I_G(I1) = 1 - \left(\frac{139}{303}\right)^2 - \left(\frac{164}{303}\right)^2 = 0.497$$

分割后分裂节点的 Gini Impurity:

$$I_G(I1 | (L1, L2)) = I_G(L1) * Pr(L1) + I_G(L2) * Pr(L2) = 0.364$$

解释：一个随机样本有 $p(L1)$ 概率进入左分支 (impurity 为 $LG(L1)$)，而有 $p(L2)$ 概率进入右分支 (impurity 为 $LG(L2)$)

Gini Impurity 在分裂后有所降低

在选取分裂条件时，对于每一个候选分割点如上计算 Gini Impurity，**选取使得 Gini impurity 最低的分裂条件**。如果此节点在分裂后总的 Gini impurity 反而增长，则不进行分裂。

II. Information Gain

Information Gain 运用到了 Entropy (熵) 的概念，而在机器学习中 **Entropy** 跟 Gini Impurity 类似，也**是一种测量数据不纯度的方法**。而 **Information Gain 就是节点分裂后 Entropy (可理解为不纯度) 的下降**。所以 $\text{Information Gain} > 0$ 则代表分裂后预测效果更好，且 Information Gain 越高越好。

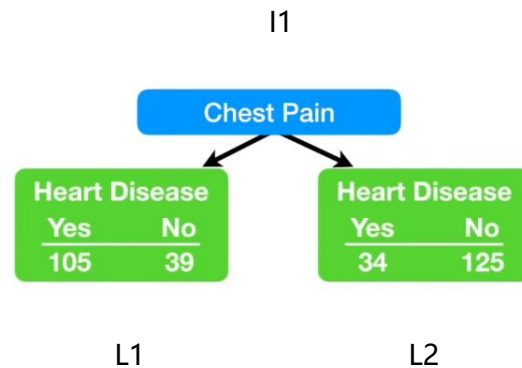
Entropy 的定义：

$$H(T) = I_E(p_1, p_2, \dots, p_J) = - \sum_{i=1}^J p_i \log_2 p_i$$

Information Gain 的定义：

$$\begin{aligned} \overbrace{IG(T, a)}^{\text{Information Gain}} &= \overbrace{H(T)}^{\text{Entropy (parent)}} - \overbrace{H(T|a)}^{\text{Sum of Entropy (Children)}} \\ &= - \sum_{i=1}^J p_i \log_2 p_i - \sum_{i=1}^J - \text{Pr}(i|a) \log_2 \text{Pr}(i|a) \end{aligned}$$

范例：



叶子节点的 Entropy:

$$H(L1) = -\left(\frac{105}{144} \log\left(\frac{105}{144}\right) + \frac{39}{144} \log\left(\frac{39}{144}\right)\right) = 0.254$$

$$H(L2) = -\left(\frac{34}{159} \log\left(\frac{34}{159}\right) + \frac{125}{159} \log\left(\frac{125}{159}\right)\right) = 0.225$$

分割前分裂节点的 Entropy:

$$H(I1) = -\left(\frac{139}{303} \log\left(\frac{139}{303}\right) + \frac{164}{303} \log\left(\frac{164}{303}\right)\right) = 0.300$$

分割后分裂节点的 Entropy:

$$H(I1|(L1, L2)) = H(L1) * Pr(L1) + H(L2) * Pr(L2) = 0.238$$

Information Gain:

$$I_G(I1, L1, L2) = H(I1) - H(I1|(L1, L2)) = 0.062$$

此处 Information Gain > 0, 判定应该进行分裂。

III. Tree Ensemble Gain

Tree Ensemble Gain 是 XGBoost 中运用的一种分裂条件选取方案，是

information gain 的一种变式。在 XGBoost 中加入了 λ 正则项后 Tree Ensemble

Gain 可推导出下列公式

y_i 预测值:

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

损失函数对于建立 $t-1$ 棵树后 y_i 预测值的 hessian:

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

hessian 之和:

$$H_j = \sum_{i \in I_j} h_i$$

节点预测效果 w_j^* 的评估公式:

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

此处 w_j^* 跟 Gini Impurity 和 Entropy 同样代表不纯度。

分母中的正则项 $\lambda > 0$, 在 GDBT(详见 2.1)中没有, XGBoost 中才被加入。 λ 使得

w_j^* 变小, 更易被剪枝 (详见下方公式)

Tree Ensemble Gain = 0.5 * (子节点效果和-父节点效果) - γ 。

γ 用于剪枝, γ 越大, gain 越小, 更容易剪枝, 模型越保守。

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

推导详见 <https://xgboost.readthedocs.io/en/latest/tutorials/model.html>

虽然公式看起来很复杂，但如果我们进行回归时选取的 loss function 是 MSE,

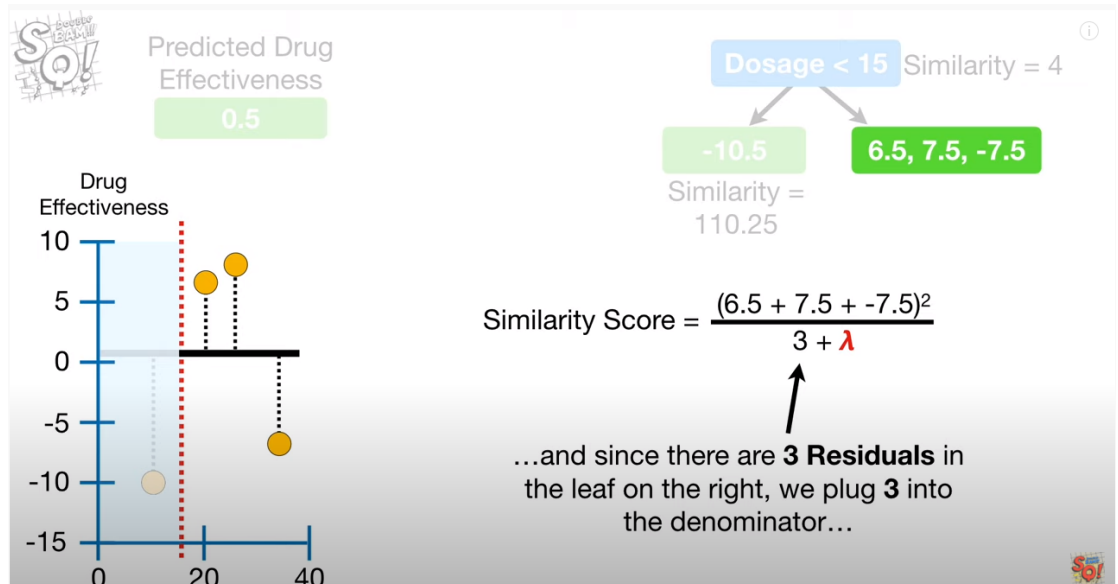
因 hessian hi 是 MSE 的二次导数，hi 永远等于 2。且因 Gain 是一个相对值

(选取分裂条件时我们只关心每个条件算出的 gain 的相对大小以及 gain 是

否>0)，我们可以把 hi 定义成 1。H 就是经过此节点的数据数量。

而选取其他损失函数会使计算稍微复杂，我提供了一个用逻辑损失函数计算 Gain

的例子。



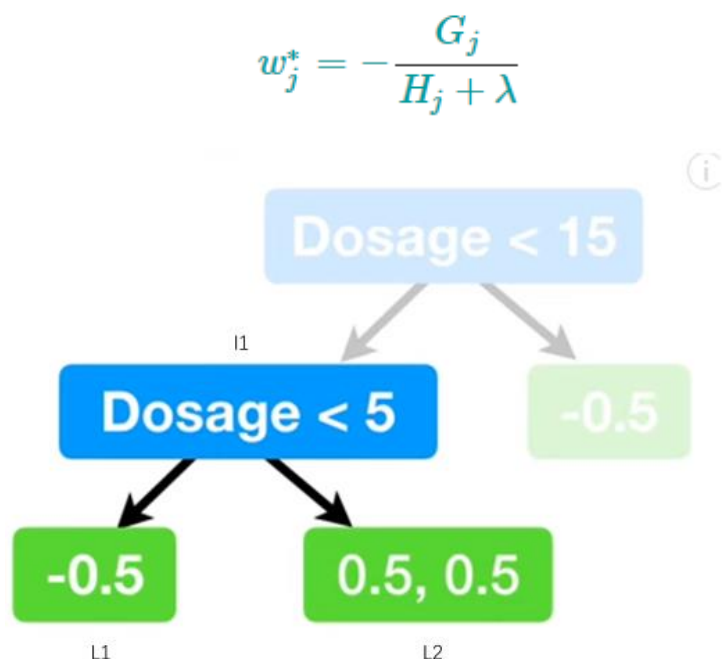
逻辑损失函数:

$$\sum y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i)$$

(pi 代表当前树之前的集成树预测的 yi 值，pi 在[0,1]区间内)

可推导出 $H \approx \sum p_i * (1 - p_i) = \text{Gini Impurity}$

Surprise! 逻辑损失函数的二次导数近似居然是 Gini Impurity。节点分裂后，叶子节点的分母的 Gini Impurity 越小，Gain 值越高，可以发现两者的定义完美地被满足。接下来就要计算 Gain 的具体值了



在此二分类问题的决策树范例中，叶子节点的每一条数值代表了一条训练集数据的残差 (真实值-先前预测值)，由于真实值等于 0 或 1，可以推断出之前每条数据的先前预测值都是 0.5。假设正则项 $\lambda = 1$ ， $\gamma = 0.2$ 。

叶子节点的 Gain:

$$w(L1) = \frac{(-0.5)^2}{(1 - 0.5) * 0.5 + 1} = 0.2$$

$$w(L2) = \frac{(0.5 + 0.5)^2}{(1 - 0.5) * 0.5 + (1 - 0.5) * 0.5 + 1} = 0.333$$

分割前分裂节点的 Gain:

$$w(I2) = \frac{(0.5 + 0.5 - 0.5)^2}{(1 - 0.5) * 0.5 + (1 - 0.5) * 0.5 + (1 - 0.5) * 0.5 + 1} = 0.143$$

分割后分裂节点的 Gain:

$$w(I2|(L1, L2)) = w(L1) + w(L2) = 0.533$$

Tree Ensemble Gain:

$$I_G(I1, L1, L2) = w(I2|(L1, L2)) - w(I2) - 0.2 = 0.19 > 0$$

此处 Tree Ensemble Gain 判定应该进行分裂。(Gain > 0)

GBDT (Gradient Boosted Decision Trees)

了解过这些 XGBoost 依赖的基础模块后，我们可以开始理解 XGBoost 是如何合理运用这些模块达到良好的预测效果的。

XGBoost 的核心算法是对在每次建立新的决策树时,运用已经完成的决策树进行 Gradient Boost--每个树都对在它之前建立的树们进行 Gradient Descent。接下来我会详细分析 Gradient Boosted Decision Trees 的算法和优势。

2.1 GBDT 概念

简单来说，GBDT 的本质就是在建立每颗决策树时，尝试减少前面的树遗留下来的误差。

具体算法如下分为解释 + 公式的形式：

输入：选取训练集和用于计算残差 (Residual) 的 Loss Function L

目标：降低 L 的值

Input: Data $\{(x_i, y_i)\}_{i=1}^n$, and a differentiable **Loss Function** $L(y_i, F(x))$

1. 建造叶子节点包含相同值的树(值 = 最小化 loss function 的预测值 γ , 如二分类问题

$\gamma = 0.5$, 回归问题 $\gamma = \text{label 平均值}$

Step 1: Initialize model with a constant value: $F_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^n L(y_i, \gamma)$

2. 一共建造 M 个树

对于每个树:

A. 计算每个预测值的残差

B. 根据残差值建造 regression tree, 将叶子节点标注为 R_{jm} , (并且把数据按照叶子节点归类)

C. 对每个叶子节点, 计算使得 loss(叶子节点内数据) 最小的 γ , 并将 γ 作为叶子节点的输出值

D. 把这个树乘上 Shrinkage(学习率, ν (常叫做 eta)), 加入预测树集

Step 2: for $m = 1$ to M :

(A) Compute $r_{im} = - \left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)}$ for $i = 1, \dots, n$

(B) Fit a regression tree to the r_{im} values and create terminal regions R_{jm} , for $j = 1 \dots J_m$

(C) For $j = 1 \dots J_m$ compute $\gamma_{jm} = \underset{\gamma}{\operatorname{argmin}} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$

(D) Update $F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$

3. 预测结果 = $F_0(x)$ (常数值的树) + v *第一个树的预测 + v *第二个树的预测...

Step 3: Output $F_M(x)$

2.2 GBDT 分析

接下来分析一下 GBDT 里 gradient descent 的特性和时间复杂度。

1. Gradient Boosting 和线性回归中 Gradient Descent 的对比

- 线性回归: 重复计算损失函数的导数并乘以 Learning Rate 用于更改预测值, 直到损失函数接近最小值 (θ 趋近于 argmin (损失函数))
- GBDT: 重复计算 argmin (损失函数) 并乘以 Learning Rate 用于更改预测值, 直到模型预测值不再接近真实值 (bias 和 variance 的平衡)

可以发现, 两者都运用到了 Gradient Descent, 但因目标不同, 方案也有差别

2. 时间复杂度 -> $O(M*s*m*n)$

- M = 树的数量
- s = 分裂节点数
- m = 特征数
- n = 训练集大小

解释: 通过 memoization, 计算每个分裂节点的 gain 只需要 $O(1)$ 时间, 所以时间复

杂度 = gain 的计算次数 = M *每棵树的次数 = $M*s$ *每个节点的计算次数 =

$M*s*m$ 每个特征在每个节点里最大候选分割点个数 = $M*s*m*n$.

2.3 运用 GBDT 进行回归或二分类

回归:

- 常用损失函数: $L(y, \gamma) = \frac{1}{2} \sum (y_i - \gamma_i)^2$
- 预测值 = $F_0(x) + v \sum \gamma_i$

二分类:

- 常用损失函数:

$$L(y_i, p_i)$$

$$= \sum y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i)$$

$$= \sum y_i * \log\left(\frac{p_i}{1-p_i}\right) + \log(1 + e^{\log\left(\frac{p_i}{1-p_i}\right)})$$

- $L(y_i, p_i)$ 对于 $\log\left(\frac{p_i}{1-p_i}\right)$ 的导数 r_i

$$= y_i + \frac{1}{1 + e^{-\log\left(\frac{p_i}{1-p_i}\right)}} = y_i + p_i$$

(步骤 2A)

- $\gamma_j = \frac{r_i}{p_i * (1-p_i)}$ (用于 Gain 的计算)

(步骤 2C)

- 集成树输出值(log odds) = 预测的 $\log\left(\frac{p_i}{1-p_i}\right)$ 值 = $F_0(x) + v \sum \gamma_i$

- 输出值 $p = \frac{1}{1 + e^{-\log\left(\frac{p_i}{1-p_i}\right)}} \in [0, 1]$

(步骤 3)

XGBoost

3.1 XGBoost 概念

XGBoost 对 GBDT 的理论进行了多语言 implementation 和预测效果优化, 并且在数据量较大时, 通过多种方法节省运行时间, 使得模型能在可控的时间内进行训练 (5-60 min)。

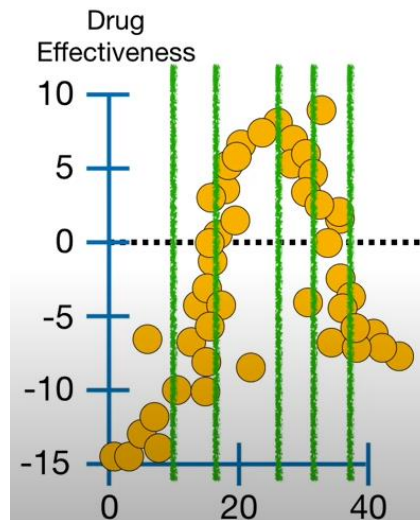
XGBoost 对于 GBDT 的优化大致可分为这几类:

降低预测精确性+缩短训练时间的优化

I. Approximate Greedy Algorithm + Weighted Quantile

a. Approximate Greedy Algorithm

在 1.2 章节介绍 Regression Tree 的时候, 我们了解到 Regression Tree 分割节点时只要求分裂条件是当前最优, 不考虑是否对于整个数据集的分割最优, 因此它是 Greedy 算法。然而当数据量很大时, 在每次分裂时都计算所有相邻数据的平均值的 Gain 仍然会花费过多的时间。因此, XGBoost 在 GBDT 上更进一步, 把每个特征的数据分成大小相同 quantiles (分位数) 并用 quantile 的分割值作为候选分裂条件 (默认每个 feature 使用 33 个 quantile)。



虽然这会降低一定的精确性，但 approximate greedy algorithm 通过限制候选分裂条件的数量大大加快了模型的建立。

b. Weighted Quantile

我们在把数据分割成大小相同的 quantile 时，不可避免地对 Regression

Tree 的二分类模型引入了 bias。在进行节点分割时，我们需要通过以下公式

计算 gain 最大的分割点

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

而每个 entry 的 h 在分类问题中等于 $p_i * (1 - p_i)$

假设我们对有两个 Quantile, Q1 和 Q2, Quantile Q1 里数据的 H1 大于 Q2 的

H2。我们可以推断 Q1 里的数据值离真实值比 Q2 里的更近。因此，在 Q2 中进行

分割对于模型的提升效果比在 Q1 中好。Gain 最高的分裂条件更有可能落在 Q2

中。因此，我们希望在 Quantile 分割后仍然有更多候选分割值在 Q2 中，使得

Q2 中的某些分割值离整体最优解更近，使得模型效果和训练速度更佳。

因此，我们对每个 entry 计算 $h: p_i * (1 - p_i)$ ，并在分割 quantile 的时候使得所有 quantile 里 h 的和基本相等。这便是 weighted quantile。H 高的区域会分到更多的 quantile，Gain 最高的分裂条件更有可能落在其中。

对于回归问题，则直接用原始方法进行分割（每个 quantile 数据数量相等），因为对于 h_i 恒等于 1，不存在上述 bias。

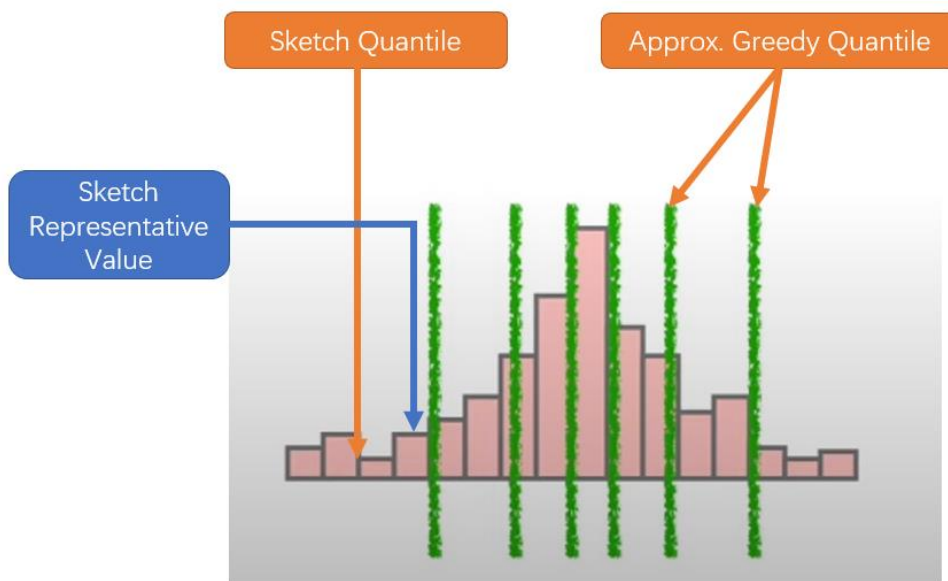
II. Weighted Quantile Sketch

Weight Quantile Sketch = Weighted Quantile + Quantile Sketch

Quantile sketch 会从每个 quantile 中选取一个代表值 (representative value) 作为训练的数据值。通过这个流程我们会得到数量与 quantile 数相同的代表值。在选取分裂条件时，对于这些代表值计算 gain/impurity，而不是数据集里的所有值。

由于 XGBoost 要建立很多树，并且每次分割节点都要计算 gain/impurity，

weighted quantile sketch 通过损失一些精确性的方式大大加快了模型建立的速度。

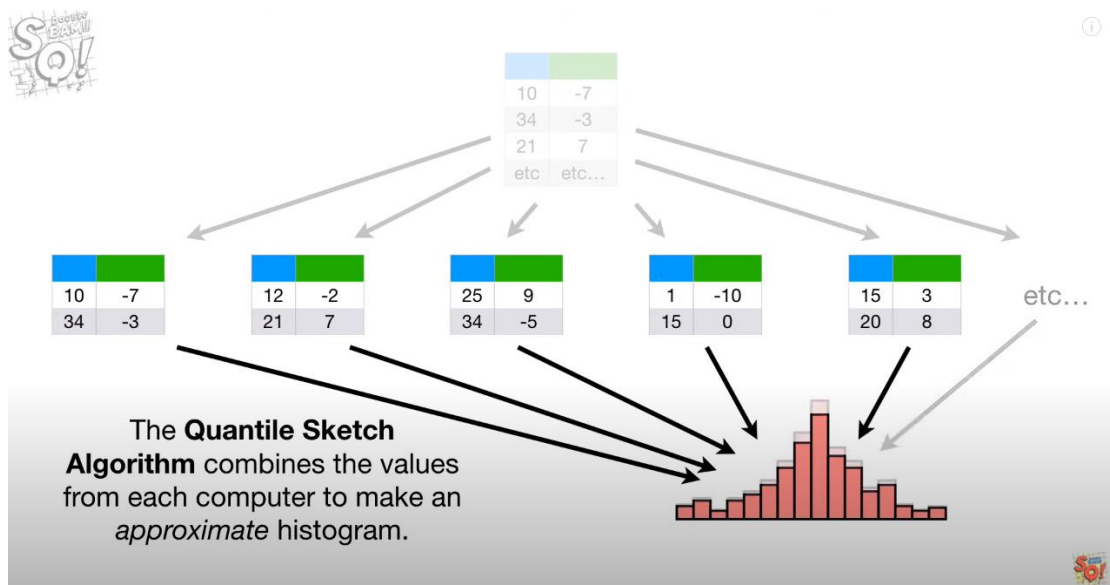


保持预测精确性+缩短训练时间的优化

III. 并行计算

XGBoost 通过切分数据，对例如 Weighted Quantile Sketch 的算法进行并行和结

果叠加，且在多个机器中运算时有更多能利用的 Cache/内存，大幅加快了运行速度



IV. Cache 优化

XGBoost 在 Cache 中运算计算量较大的 Hessian，避免了短板效应。

V. Out-of-core Computation

在数据量大的情况下，计算机必须把一些数据存储进硬盘。因此提升运用硬盘读写速度便特别重要。**XGBoost 会首先尝试压缩数据**，因为压缩解压缩的速度一般来说要高于内存读写，**如果问题没有得到解决，则把数据分成 blocks 用 sharding 算法存进硬盘，并在有需求时并行读写。**

提高模型应用性的优化

VI. Sparsity Aware Split Finding

我们的训练集数据有时会有一些缺失的数据，而 XGBoost 在这种情况下仍然能训练模型。具体补缺失值的方法如下：

在选取分裂条件时，对于每一个特征，XGBoost 把**此特征数据完整和有缺失的数据分到两个 table：T1 和 T2**。把 T1 分成 quantile，对于每一个 quantile 的分界值 s_i ，把 T2 中**所有数据**归类到 $x_i \leq s_i$ 计算 Gain，再把 T2 中所有数据归类到 $x_i > s_i$ 计算，最终取 Gain 最佳的 s_i 作为分割值，**并且把 T2 数据按照最佳方案全部归类到左节点或右节点**。我们无法知道缺失部分的真实值，但仍然可以把它们归类到使模型效果较优的一边。**之所以 XGBoost 能如此简便地归类有缺失值的数据是因为 Gain 的计算不依赖缺失数据的数值。**

VII. 支持多种预测模式

XGBoost 除了 regression 和 binary classification 外，还提供了 multi:softmax 来解决多 category 的分类问题。

提升预测效果的优化

VIII. XGBoost Pruning

与 GBDT 不同，**XGBoost 允许手动设置树的高度限制**。另外，树在达到高度之前可以自由分割，即使 $gain \leq 0$ 。**所以在剪枝前的 XGBoost 树是一个满二叉树**。树达到指定高度后再从底部朝上对 $gain \leq 0$ 的节点进行 pruning。**这使得一些有负 gain 值的节点被保留下来**（pruning 后的它们的子节点均有 $gain > 0$ ）这样我们就**不会 prune 掉暂时降低 gain 但之后产出提升 gain 的节点的分割节点**。

IX. XGBoost 正则项

XGBoost 在 GBDT 的基础上加入了正则项，正则越大 Gain 越小，模型更 conservative。

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

正则参数：

alpha: L1 正则, default = 0

lambda: L2 正则, default = 1

3.2 XGBoost 调参

调整超参数 (hyperparameter tuning) 能让模型的预测效果更好。超参数指决定模型训练方式的数据, 如 learning rate, 正则化系数, 决策树深度 (高度) 的限制。超参数不会被模型训练到, 所以必须在最终 XGBoost 模型训练前被确定。

为什么这里我们要强调超参数必须在最终的 XGBoost 模型训练前被确定呢?

因为我们可以用算法通过训练多个不会用于预测结果的 XGBoost 模型自动化训练超参数。

我们自然的会想到一个简单粗暴的算法:

定义一个超参数的组合为 $(p_1=a_1, p_2=a_2 \dots p_n=a_n, n \text{ 为参数个数}, a_i \text{ 为有理数/种类})$ 。

对于所有超参数的组合训练一个 XGBoost 模型, 并选取效果最佳的组合。

然而, 由于多个超参数的排列组合数量巨大, 且单个模型训练时间也不可忽视。尝试所有的情况并不现实。幸运的是我们可以运用一些更好的算法趋近超参数最佳的组合。

模型通用的调参

I. Grid Search

Grid Search 是一个相对入门且容易编写的自动化调参算法。

Step 1: 对于每个超参数 x_i , 选定几个候选值

Step 2: 对所有超参数的所有候选值的组合进行模型训练, 选取效果最佳的模型和组合。

优点:

- 编写方便，便于理解

缺点：

- 运行缓慢（排列组合数量多）
- 有较大概率结果远离最佳值

Python 范例（二分类问题 scoring 建议使用 roc_auc）：

注：这些超参数的排列组合共有 $2*4*3*2*2*3 = 288$ 种组合，所以我们要训练 288 个模型，可以看出运行即使每个超参数只有 2-4 个候选值，Grid Search 也会让计算量爆炸。

```
def hyperParameterTuning(X_train, y_train):
    param_tuning = {
        'learning_rate': [0.01, 0.1],
        'max_depth': [3, 5, 7, 10],
        'min_child_weight': [1, 3, 5],
        'subsample': [0.5, 0.7],
        'colsample_bytree': [0.5, 0.7],
        'n_estimators': [100, 200, 500],
        'objective': ['reg:squarederror']
    }

    xgb_model = XGBRegressor()

    gsearch = GridSearchCV(estimator = xgb_model,
                           param_grid = param_tuning,
                           #scoring = 'neg_mean_absolute_error', #MAE
                           #scoring = 'neg_mean_squared_error', #MSE
                           cv = 5,
                           n_jobs = -1,
                           verbose = 1)

    gsearch.fit(X_train, y_train)
```

II. Random Search

Random Search 通过随机选取超参数组合实现训练相近数量的模型后得到比 Grid Search 更优的结果。Random Search 作为一个同样容易理解和编写的算法被广泛应用于实战。

Random Search 算法:

for 限定的运行次数:

Step 1: 对于每个超参数 x_i , 我们选定它的合理取值区间 $[a_i, b_i]$ (上网搜索常用范围即可)。

Step 2: 随机选取多个 v (超参数组合), 使得 v_i 在 $[a_i, b_i]$ 之间, 并对每个 v 进行模型训练, 选取效果最佳的模型和组合。

优点:

- 编写方便, 便于理解
- 总体优于 Grid Search

缺点:

- 有一定概率结果远离最佳值

Python 范例 (注: 二分类问题 scoring 建议使用 roc_auc)

```

import numpy as np
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy

start_time=time.time()

#### Create X and Y training data here.....

# grid search
model = XGBRegressor()

param_grid = {
    'max_depth': [3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
    'min_child_weight': np.arange(0.0001, 0.5, 0.001),
    'gamma': np.arange(0.0, 40.0, 0.005),
    'learning_rate': np.arange(0.0005, 0.3, 0.0005),
    'subsample': np.arange(0.01, 1.0, 0.01),
    'colsample_bylevel': np.round(np.arange(0.1, 1.0, 0.01),
    'colsample_bytree': np.arange(0.1, 1.0, 0.01),

{

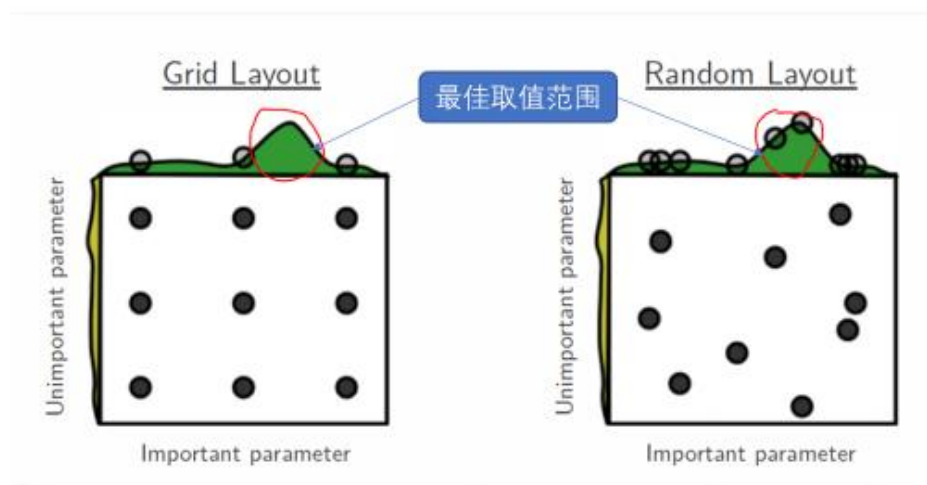
kfold = KFold(n_splits=10, shuffle=True, random_state=10)
grid_search = RandomizedSearchCV(model, param_grid, scoring="accuracy", n_iter = 500, cv=kfold)
grid_result = grid_search.fit(X,Y)

# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_[ 'mean_test_score' ]
stds = grid_result.cv_results_[ 'std_test_score' ]
params = grid_result.cv_results_[ 'params' ]

print(time.time()-start_time)

```

与 Grid Search 的对比:



上图很好的说明了为什么 random search 总体优于 grid search。

对于每个模型，一些超参数对预测结果的影响比其他的要大。如图所示由于 **grid search** 运用排列离散排列组合的特性，它在重要的超参数上取值的数量会受到限制，并且可能跳过最佳的取值范围。不仅如此，超参数调参的过程属于黑盒，因此用户很难在选取 grid search 候选值的时候知道它们有没有跳过最佳取值范围。

而 **random search** 基于随机性，在取点个数相同的情况下更有可能有点落在最佳取值

范围内，得到比 grid search 更准确的效果。

然而我们要注意的是当重要性大的超参数最佳取值范围较大时，random search 仍然有一定概率错过最佳取值范围。所以更优秀的算法被开发出来。

III. Bayesian 优化

Bayesian 优化通过运用统计原理实现了在一定步骤后得到比 random search 更优的结果，并因此成为了目前超参数调参的首选之一。

Surrogate: objective function 的近似 (objective function 运算耗时，运用近似运算更快)，输入参数组，输出此参数组 objective function 的概率分布。

Acquisition Function: 将 surrogate 的概率分布转化为概率值。

Bayesian 优化的算法：

建立初始 surrogate

for 限定的运行次数：

S = acquisition function 预测值最高的参数组

在 S 上训练模型 M

用 M 的结果修正 surrogate (下一个 iteration acquisition 的值也会随着 surrogate 的变化而变化)

优点：

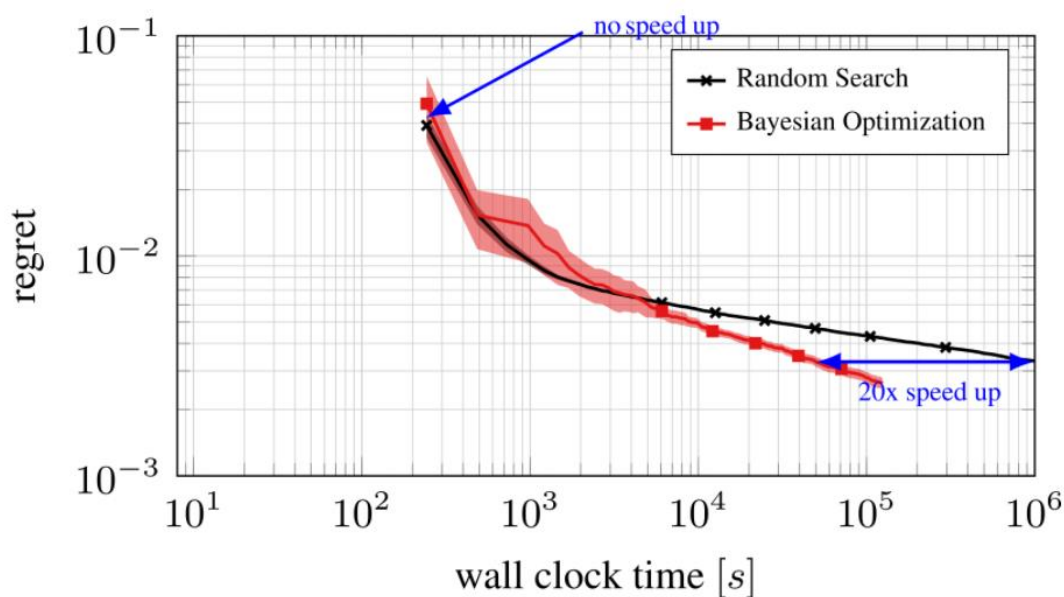
- 结果更近似最佳值

缺点:

- 非并行 (每次 iteration 依赖之前的数据)

Python 范例:

<https://betatim.github.io/posts/bayesian-hyperparameter-search/>



前期效果类似 random search, 后期更优

IV. Bayesian with Hyperband

由于 Bayesian 需要非并行地训练模型, 它的高效会有所中和。所以当统计学家意识到这个问题, 想着如果一个处理器只能同时运行一个 Bayesian 优化, 那何不同时在多个小区间内运行的多个 Bayesian 优化呢, 这样的话不就只消耗了区间 Bayesian 优化的时间吗? 因此 Bayesian with Hyperband 诞生了。

公式:

随机选取 2^k 个超参数组合并行 Bayesian。(2^k 个候选 model)

while (model 个数 > 1)

 每个 Bayesian 运行 m 次。

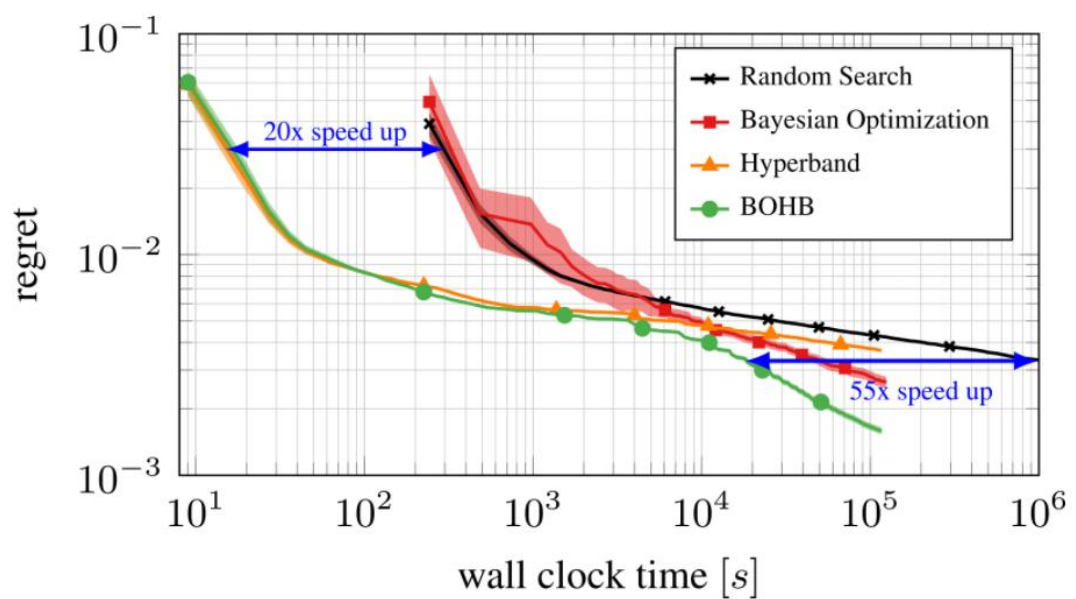
 丢弃预测效果较差的一半 model

优点:

- 可并行运行多个 Bayesian 优化

缺点:

- k/m trade-off: 为控制运行时间, 需手动选取 k 和 m , 或用算法近似最佳的 k 和 m (两层超参数, 极为耗时)
- 编写较为复杂



由于 Hyperband 初始选取参数组数量多，因此某个参数组的效果跟单个 bayesian 比更有可能接近最优，所以初期 loss function 下降快。而当 Hyperband 只剩一个参数组进行 bayesian 优化，在后期效果会优于 random search (Bayesian 优化的特性)。

V. PBT

PBT 是 Genetic Algorithm 的变形，具体算法如下：

Step 1: 选取一些超参数组合进行神经网络训练

Step 2: 在一定 iteration 后把效果较差的组合替换成效果最好的一组超参数 v

并计算周边 y 值 (使其与 v 不同)。重复 step 2.

优点：

- 可并行
- 效果较好

缺点：

- 涉及神经网络数据的替换，编写较为复杂

一些调参的工具：

<https://medium.com/neptune-ai/hyperparameter-tuning-in-python-a-complete-guide-2020-cfd8886c784b>

XGBoost 调参

XGBoost 的调参可以运用上面提到的或专门针对 XGBoost 的方法。

通常我们需要 tuning 的超参数无外乎以下几个：

- eta: 学习速率
 - 一般范围：0.01-0.3

- max_depth: 决策树的高度，depth 约大 bias 越小 variance 越大，底部分裂条件可能被 noise 大幅影响
 - 一般范围：3-10

- min_child_weight: 叶子节点最小允许的 cover 值，用于 pruning，cover 越大 variance 越小。
 - 一般范围：1-7

- gamma: 计算 Gain 的 gamma，取值越大 variance 越小。
 - 一般范围：0.0-0.5

- colsample_bytree: 如取值 m 在 $[0,1]$ 之间，则在建立每个树时，随机选取 $m \times (\text{feature 总数})$ 个 feature 进行树的训练，减少 variance。如 $m > 1$ ，则对于每个树随机选取 m 个 feature 进行树的训练。
 - 一般范围：0.5-1

- subsample: 对于每个树，随机选取 m^* (训练集大小) 的数据进行训练。

- 一般范围: 0.5-1

特别针对 XGBoost 的调参法:

http://km.oa.com/articles/show/313234?kmref=search&from_page=1&no=1

一些网上 XGBoost 的模型运用了如上把关联较大参数配对进行 grid search 的方法。

在这里给出一些提升调参效果的方法建议:

- 可运用 BOHB。

注: 如需调整 L1 和 L2 Regularization 可调整参数 alpha 和 lambda (scala) / reg_alpha 和 reg_lambda (python)。

注: 建议设置参数 numEarlyStoppingRounds (scala) / early_stopping_rounds (python) 为 10 左右或更少, 即如果连续的 10 个树没有提升模型效果, 则提前终止模型训练。而因此也可将 numRound (scala) / n_estimators (python) 设置得更大。

3.3 XGBoost 的局限性

虽然 XGBoost 有监督地训练的模型结构性数据 (tables) 上的效果是目前最稳定最好的之一, 但不能处理像 neural network 那样很好的处理非结构性数据 (文字, 图片, 视频), 并且不能像 KNN 那样实现无监督聚类。对于股票价格这种 noise 很高而且随

机性很高的数据集的也很容易产生 overfitting。

3.4 XGBoost 与其他模型的对比

在最后我想对比一下 XGBoost 和其他广泛应用或效果较好的有监督结构化数据的模型，以帮助大家在最合适的场景下运用 XGBoost。

随机森林

- XGBoost 把随机森林对于特征和数据的随机选取作为 parameter 结合进了模型中 (colsample_bynode (only in python) / bylevel / bytree, scala 中没有 bynode 应该是因为对于每个 node 随机选取 feature 比较耗费时间，且 bynode 和 bylevel 训练出的模型 AUC 差别不是很大)
- XGBoost 里树的建立方式依赖于之前的树，而随机森林的树是独立的。XGBoost 的树尝试对之前的模型有所提升，并对随机树造成的 bias 进行中和，而随机森林只有后者。
- XGBoost 整合了更多的 functionality，如 3.3 里提到的参数，进一步提升模型准确性，且运行时间加快很多

Light GBM

Light GBM 在 XGBoost 的基础上在基本不减少准确性的情况下在建模时间上做了重

点优化，适合训练时间过长的 XGBoost 模型。

特点：

- 运用和 XGBoost 不同的 boosting algorithm (**按照节点而非 level 建造树**)，控制了 variance
- 加入了一些超参数（节点数限制，etc）,调参后效果与 XGBoost 相似。
- 已被整合进较新版本的 XGBoost

效果展示：

	Xgboost	LightGBM
AUC	0.8496	0.8464
Training time	6.576s	1.964s
Prediction time	266ms	400ms

Catboost

Catboost 是俄罗斯 Yandex 推出的一款**预测准确率高于 XGBoost** 的 GBDT implementation.










特点：

- **对类别特征的处理进行了统计和算法上的优化。**
- 运用和 XGBoost 不同的树结构 (**对称树**)，减少了 variance，并**加快了训练速度**
- 运用和 XGBoost 不同的 boosting algorithm (**ordered boosting**)，进一步控制

variance

- 简化了调参，并且使 default parameter 输出的模型效果更接近调参后模型。
- 缩短预测时间

效果展示：

	CatBoost		LightGBM		XGBoost		H2O	
	Tuned	Default	Tuned	Default	Tuned	Default	Tuned	Default
 Adult	0.26974	0.27298 +1.21%	0.27602 +2.33%	0.28716 +6.46%	0.27542 +2.11%	0.28009 +3.84%	0.27510 +1.99%	0.27607 +2.35%
 Amazon	0.13772	0.13811 +0.29%	0.16360 +18.80%	0.16716 +21.38%	0.16327 +18.56%	0.16536 +20.07%	0.16264 +18.10%	0.16950 +23.08%
 Click prediction	0.39090	0.39112 +0.06%	0.39633 +1.39%	0.39749 +1.69%	0.39624 +1.37%	0.39764 +1.73%	0.39759 +1.72%	0.39785 +1.78%
 KDD appetency	0.07151	0.07138 -0.19%	0.07179 +0.40%	0.07482 +4.63%	0.07176 +0.35%	0.07466 +4.41%	0.07246 +1.33%	0.07355 +2.86%
 KDD churn	0.23129	0.23193 +0.28%	0.23205 +0.33%	0.23565 +1.89%	0.23312 +0.80%	0.23369 +1.04%	0.23275 +0.64%	0.23287 +0.69%
 KDD internet	0.20875	0.22021 +5.49%	0.22315 +6.90%	0.23627 +13.19%	0.22532 +7.94%	0.23468 +12.43%	0.22209 +6.40%	0.24023 +15.09%
 KDD upselling	0.16613	0.16674 +0.37%	0.16682 +0.42%	0.17107 +2.98%	0.16632 +0.12%	0.16873 +1.57%	0.16824 +1.28%	0.16981 +2.22%
 KDD 98	0.19467	0.19479 +0.07%	0.19576 +0.56%	0.19837 +1.91%	0.19568 +0.52%	0.19795 +1.69%	0.19539 +0.37%	0.19607 +0.72%
 Kick prediction	0.28479	0.28491 +0.05%	0.29566 +3.82%	0.29877 +4.91%	0.29465 +3.47%	0.29816 +4.70%	0.29481 +3.52%	0.29635 +4.06%

更多详情请参见：

<https://arxiv.org/abs/1706.09516>

<https://www.youtube.com/watch?v=8o0e-r0B5xQ>

附录

4.1 逻辑回归

逻辑回归概念：

通过 $p_i = \frac{1}{1+e^{-x}}$ ，把 $x \in [-\infty, +\infty]$ （原始预测值）转换成 $p_i \in [0,1]$ ， x 通过推断等于 $\log\left(\frac{p_i}{1-p_i}\right)$ 。

逻辑回归预测方法：

输入特征数据 \rightarrow 预测 $x = \log\left(\frac{p_i}{1-p_i}\right) \rightarrow$ 预测 $p_i = \frac{1}{1+e^{-\log\left(\frac{p_i}{1-p_i}\right)}}$

逻辑回归损失函数和其梯度的推断：

$$h_{\theta}(x; \theta) = p(y = 1|x; \theta) = \phi(w^T x + b) = \phi(z) = \frac{1}{1 + e^{-\theta^T x}}$$

$$p(y = 0|x; w) = 1 - \phi(z)$$

将上面两式写为一般形式：

$$p(y|x; \theta) = h_{\theta}(x; \theta)^y (1 - h_{\theta}(x; \theta))^{(1-y)}$$

接下来使用极大似然估计来根据给定的训练集估计出参数 w ：

$$L(\theta) = \prod_{i=1}^n p(y^i|x^i; \theta) = \prod_{i=1}^n h_{\theta}(x^i; \theta)^{y^i} (1 - h_{\theta}(x^i; \theta))^{(1-y^i)}$$

为了简化运算，我们对上述等式两边取一个对数：

$$l(\theta) = \ln L(\theta) = \sum_{i=1}^n y^i \ln(h_{\theta}(x^i; \theta)) + (1 - y^i) \ln(1 - h_{\theta}(x^i; \theta))$$

现在要求使得 $l(w)$ 最大的 w ，在 $l(w)$ 前面加一个负号就变为最小化负对数似然函数：

$$J(\theta) = -l(\theta) = - \left(\sum_{i=1}^n y^i \ln(h_{\theta}(x^i; \theta)) + (1 - y^i) \ln(1 - h_{\theta}(x^i; \theta)) \right)$$

统计学习方法都是由模型、策略和算法构成的，以logistic回归为例，模型自然是logistic，策略最常用的方式是用一个损失函数来度量预测错误程度，算法则是求解过程。

logistic的函数表达式为：

$$g(z) = \frac{1}{1 + e^{-z}}$$

而它的梯度为：

$$\begin{aligned} g'(z) &= \frac{d}{dz} \cdot \frac{1}{1 + e^{-z}} \\ &= \frac{1}{(1 + e^{-z})^2} \cdot (e^{-z}) \\ &= \frac{1}{1 + e^{-z}} \cdot \left(1 - \frac{1}{1 + e^{-z}}\right) \\ &= g(z) \cdot (1 - g(z)) \end{aligned}$$

4.2 回归树和分类树的区别

两者大致相似，区别如下：

区别	回归树	分类树
输入数据类型	数值	类别
输出数据类型	数值	类别
常用分割条件选取方案	MSE / （二分类：Gini / IG / Tree Ensemble Gain）	Gini / IG / Tree Ensemble Gain
叶子节点数据类型	数值	类别
叶子节点值	节点内数据均值	节点内数据众数

4.3 回归树详细算法

Queue of groups $q = \emptyset$, X = dataframe that contain feature values, Y = dataframe that contain label values, limit = lower bound for # of entries in

leaf node (optional)

split_finding_method \in {Gini Impusurity, Information Gain, Tree Ensemble Gain, Loss Reduction}

Add (X, Y) (a group) to q

While(q \neq \emptyset)

Group g = q.pop (g := (X' , Y') := (X1' ,X2'Xm' ,Y'))

If(size of g < limit) continue;

minCost = $+\infty$

maxGain = 0

splitThreshold = None

For feature j in J:

Li = \emptyset

Add all entries of X' to Li;

Sort(Li) based on the jth feature (Xj' value)

For adjacent pair (v1, v2) in Li:

Set threshold T = avg(v1, v2)

if(

{

Gini Impurity

$$P = \sum \frac{tag(e \in g)}{size(e \in g)};$$

$$original_cost = 2 * P * (1 - P);$$

$$let\ gv1 = \{e : e \in g\ and\ e_j \leq v1\};\ gv2 = \{e : e \in g\ and\ e_j \geq v2\}$$

$$P1 = \sum \frac{tag(e \in gv1)}{size(e \in gv1)}; P2 = \sum \frac{tag(e \in gv2)}{size(e \in gv2)};$$

$$I_G(L1) = 2 * P1 * (1 - P1); \quad I_G(L2) = 2 * P2 * (1 - P2);$$

$$\text{Cost(Gini Impurity)} = I_G(L1) * p(L1) + I_G(L2) * p(L2)$$

```
if(Cost < min(minCost, original_cost)){
```

```
    minCost = Cost
```

```
    splitThreshold = (j, T)
```

```
}
```

```
}
```

```
{
```

Information Gain

$$P = \sum \frac{tag(e \in g)}{size(e \in g)};$$

$$H(\text{Parent}) = -P \log(P) - (1 - P) \log(1 - P);$$

$$\text{let } gv1 = \{e : e \in g \text{ and } e_j \leq v1\}; \quad gv2 = \{e : e \in g \text{ and } e_j \geq v2\}$$

$$P1 = \sum \frac{tag(e \in gv1)}{size(e \in gv1)}; P2 = \sum \frac{tag(e \in gv2)}{size(e \in gv2)};$$

$$H(\text{Leaf1}) = -P1 \log(P1) - (1 - P1) \log(1 - P1);$$

$$H(\text{Leaf2}) = -P2 \log(P2) - (1 - P2) \log(1 - P2);$$

$$H(\text{Parent} \mid (\text{Leaf1}, \text{Leaf2})) = H(\text{Leaf1}) * P1 + H(\text{Leaf2}) * P2$$

```
if(H(Parent | (Leaf1, Leaf2)) < min(minCost, H(Parent))){
```

```
    minCost = H(Parent | (Leaf1, Leaf2))
```

```
    splitThreshold = (j, T)
```

```
}
```

```
}
```

```
{
```

Tree Ensemble Gain

$$H(\text{Parent}) = \frac{(\sum_{y_i \in g(y)} (\hat{y} - \bar{y}_i))^2}{(\sum_{y_i \in g(y)} \text{loss}''(y_i, \hat{y})) + \lambda};$$

$$\text{let } gv1 = \{e : e \in g \text{ and } e_j \leq v1\}; \text{ } gv2 = \{e : e \in g \text{ and } e_j \geq v2\}$$

$$P1 = \sum \frac{\text{tag}(e \in gv1)}{\text{size}(e \in gv1)}, P2 = \sum \frac{\text{tag}(e \in gv2)}{\text{size}(e \in gv2)},$$

$$H(\text{Leaf1}) = \frac{(\sum_{y_i \in gv1(y)} (\hat{y} - \bar{y}_i))^2}{(\sum_{y_i \in gv1(y)} \text{loss}''(y_i, \hat{y})) + \lambda};$$

$$H(\text{Leaf2}) = \frac{(\sum_{y_i \in gv2(y)} (\hat{y} - \bar{y}_i))^2}{(\sum_{y_i \in gv2(y)} \text{loss}''(y_i, \hat{y})) + \lambda};$$

$$\text{Gain} = 0.5 * (H(\text{Leaf1}) + H(\text{leaf2}) - H(\text{Parent})) - \text{gamma}$$

```
if(Gain > maxGain){
```

```
    maxGain = Gain
```

```
    splitThreshold = (j, T)
```

```
}
```

```
}
```

```
{
```

Loss Reduction

$$M1 = \text{mean}(v: v \leq T); M2 = \text{mean}(v: v > T)$$

$$\text{Cost} = \text{Loss}(v: v \leq T, M1) + \text{Loss}(v: v > T, M2) \text{ Loss}(\text{observe}, \\ \text{predict})$$

```
If(Cost < minCost){
```

```
    minCost = Cost
```

```
    splitThreshold = (j, T)
```

```
}
```

}

Split g based into g_1, g_2 with `splitThreshold`

Add g_1, g_2 to q

Theoretical Complexity: $O(sm n)$:

s = size of tree

m = number of features

n = number of data entries.

4.3 XGBoost 超参数列表

Python

<https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/>

Scala

https://xgboost.readthedocs.io/en/release_1.1.0/jvm/scaladocs/xgboost4j-spark/ml/dmlc/xgboost4j/scala/spark/XGBoostRegressor.html

参考资料

StatQuest Decision Tree

<https://www.youtube.com/watch?v=7VeUPuFGJHk>

StatQuest Random Forest

https://www.youtube.com/watch?v=J4Wdy0Wc_xQ

StatQuest Regression Tree

<https://www.youtube.com/watch?v=g9c66TUyIZ4>

Decision Tree: CART

<https://www.youtube.com/watch?v=DCZ3tsQIoGU>

Gini Impurity

<https://www.youtube.com/watch?v=u4IxOk2ijSs>

逻辑回归模型解释与 loss function 推导

<https://blog.csdn.net/yinyu19950811/article/details/81321944>

XGBoost Gain

<https://xgboost.readthedocs.io/en/latest/tutorials/model.html>

XGBoost and GBDT

<http://theprofessionalspoint.blogspot.com/2019/02/difference-between-gbm-gradient.html>

StatQuest Series: XGBoost

<https://www.youtube.com/watch?v=OtD8wVaFm6E>

<https://www.youtube.com/watch?v=ZVFeW798-2I>

Hyperparameter Tuning

<https://medium.com/neptune-ai/hyperparameter-tuning-in-python-a-complete-guide-2020-cfd8886c784b>

<https://www.jeremyjordan.me/hyperparameter-tuning/>

<https://www.kaggle.com/felipefiorini/xgboost-hyper-parameter-tuning>

<https://kevinvecmanis.io/machine%20learning/hyperparameter%20tuning/dataviz/python/2019/05/11/XGBoost-Tuning-Visual-Guide.html>

<https://medium.com/criteo-engineering/hyper-parameter-optimization-algorithms-2fe447525903>

<https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/>

Machine Learning Method Comparison

<https://bobjin.com/blog/view/c689cc0297e92fe784a5db123a5b12da.html#xgboost>

<https://www.youtube.com/watch?v=5CWwwtEM2TA>

<https://www.youtube.com/watch?v=8o0e-r0B5xQ>