XGBoost 原理解析

作者：Drxan

邮箱：yuwei8905@126.com

2017 年 04 月 15 日

目录

摘要：本文主要参照陈天奇同学的论文[1]对 XGBoost 算法的数学原理进行了详细讲解，对一些重要的算法步骤做了解释。同时本文对 Gradient Boosting 算法也作了简要介绍，用于对比学习。

关键字：GBDT,XGBoost

前言

XGBoost 算法自从被提出来后就因其出众的效率和较高的准确度受到了广泛关注，在各大比赛中大放异彩。下面这幅图的主角我觉得是 XGBoost 的完美代言人。

工欲善其事，必先利其器。尽管 XGBoost 如此优秀，网上也有一大把教程教我们如何使用它，我的内心也是极其渴望尽快能够使用 XGBoost，但我的原则就是不搞清楚原理前基本上是不会主动去使用（但从工作效率来讲，不建议大家这么做，大家最好是先硬着头皮直接跟着样例教程去直观地去体验一下 XGBoost 的使用方法，对你后面的原理学习有一定指导意义）。今天特地抽空拜读了陈天奇同学的论文[1]，从头至尾读下来那种感觉就像是我第一次看《变形金刚》，炫酷的场景总能给你带来视觉和心灵的震撼。

XGBoost 全称 eXtreme Gradient Boosting。它是 Gradient Boosting Machine 的一个 c++实现，作者为正在华盛顿大学研究机器学习的大牛陈天奇。他在研究中深感自己受制于现有库的计算速度和精度，因此着手搭建 xgboost 项目。xgboost 最大的特点在于，它能够自动利用 CPU 的多线程进行并行，同时在算法上加以改进提高了精度[2]。在陈天奇的论文中，提到了 XGBoost 的几个主要特点：

（1） 基于树的能够自动处理稀疏数据的提升学习算法；
（2） 采用加权分位数法来搜索近似最优分裂点；
（3） 并行和分布式计算；
（4） 基于分块技术的大量数据高效快速处理；

基于以上这些特性，XGBoost 使得我们能够在有限的资源条件下高效快速的实现大规模数据的任务处理。除了这些特性外，XGBoost 还有一个有别于 GBDT、AdaBoost 等传统算法的特点：以加入了正则化项的结构化损失函数为优化目标函数。这也进一步减小了其生成模型过拟合的风险。

要理解 XGBoost 算法，大家应该对“分步前向加性模型（Forward Stagewise Additive Modeling）”和“函数空间”有一个大致的理解，下面这几篇论文建议大家参考阅读 [3,4,5,6,7]

尤其是第[3]篇，作者详细讲述了 Gradient Boosting 算法的框架，并给出了几种常用损失函数条件下该算法的具体实现形式。我们的 XGBoost 正是基于 Gradient Boosting 算法进行改进的。

1 基本概念解释

1.1 函数空间中的优化问题

Boosting 算法家族中首先引入“在函数空间中做优化”这一概念的是 Gradient Boosting 算法[3]。引入了函数空间的概念后，就可以方便的使用损失函数的导数等概念并借助常规的优化算法来学习弱学习器。机器学习的监督学习问题中，我们的目标是在提出的假设空间H中找到一个最优的假设$F^*(x)\in \mathrm{H}$使得它具有最小的泛化误差。

$$\begin{aligned} F^*(X) &= \arg\min_{F(X)} E_{y,X}\psi(y,F(x)) \\ &= \arg\min_{F(X)} E_X[E_y\psi(y,F(x))\,|\,x] \end{aligned} \qquad \text{（式 1.1）}$$

$\Psi(y,F(x))$为我们定义的某种损失函数。

假设我们的训练数据 D 包含 N 个样例

$$D=\{(x_1,y_1),(x_2,y_2),...,(x_N,y_N)\},x_i\in R^n,y\in R$$

我们从假设空间中任选一个假设$F(x)$，在训练集上对每一个样本进行映射就可以得到一个 N 维点

$$P=F(X)=(F(x_1),F(x_2),...,F(x_N))$$

此时我们的损失可表示为为

$$\Psi(P)=\Psi(F(X))=\frac{1}{N}\sum_{i=1}^{N}\psi(y,F(x_i)) \qquad \text{（式 1.2）}$$

由于联合分布$P(X,Y)$未知，所以我们只能用训练数据的平均损失作为期望损失的无偏估计[8]。当我们选取不同的假设时，就会得到不一样的 P，进而得到不同的损失值。那么 P 就相当于是一个 N 维空间中的变量，而损失就是变量 P 的函数值。此时的问题就变成了在一个 N 维空间中的优化问题：

$$\min\ \Psi(P)=\Psi(y,(F(x_1),F(x_2),...,F(x_N))) \qquad \text{（式 1.3）}$$

如式 1.1 所示，P 通常是一个无限维度的变量（X 通常有无限个取值），并且我们的优化应该是针对y在x上的边缘分布下损失函数值的期望最小化进行，但

由于现实中我们只能拿到有限个数的训练数据，并且联合分布 $P(X,Y)$ 通常是未知的，所以只能基于训练数据把每一个函数变量 $F(x_i)$ 当做一个维度，用 $P=F(X)=(F(x_1),F(x_2),...,F(x_N))$ 来近似代替真实的函数点[3]。

上面的分析可以简要概括如下：

$$D \xrightarrow{H} \mathrm{P}=\{P_1,P_2,...,P_i,...\}$$

P_i 表示用假设空间中的函数 $F_i(X)\in \mathrm{H}$ 在训练集 D 上针对每一个样本进行映射而得到的一个 N 维点。P 就是优化问题 1.3 的可行域，它通常是一个无限 N 维点集，即它是凸集。那么寻找最优假设的问题可以描述如下：

$$\begin{aligned} &\min \quad \Psi(P)=\Psi(y,(F(x_1),F(x_2),...,F(x_N))) \\ &subject\,to\ P\in \mathrm{P} \end{aligned} \qquad \text{（式 1.4）}$$

如果我们的损失函数是一阶连续可到导甚至高阶连续可导的话，就可以借用导数相关的优化算法（比如线性搜索法，这是 Gradient Boosting 算法采用的搜索方法）在空间 P 中找到一个使得 $\Psi(P)$ 取极小值的点 P^*，此时对应的假设就是我们期望训练到的最优假设 F^*。

1.2 分步加性模型的理解

分步加性模型是大多数 Boosting 算法的核心。集成学习的主要手段就是反复训练多个模型，并将这些模型通过一定方式组合在一起，形成一个高性能的强大的集成模型。在 Boosting 算法体系中一般采用迭代串行的形式生成一系列模型，然后将这些模型进行线性加权相加，得到最终集成学习器。假设已经迭代到 m-1 次，得到的集成模型为

$$F_{m-1}(x)=\sum_{k=1}^{m-1}\alpha_k f_k(x) \qquad \text{（式 1.5）}$$

在下一次迭代中，我们要训练 $f_m(x)$，它应该是让新生成的集成模型在训练集上损失最小的模型

$$\begin{aligned} &(\alpha_m, f_m(x))=\arg\min_{\alpha,f(x)} E_{y,X}\psi(y,F_{m-1}(x)+\alpha f(x)) \\ &F_m(x)=F_{m-1}(x)+\alpha_m f_m(x) \end{aligned} \qquad \text{（式 1.6）}$$

回想 1.1 节，对于每一次迭代后更新得到的新集成模型 F_m（$m=1,2,...,T$），将它作用于训练集 D 中的每一个样本，就可以得到一个 N 维向量 $P_m=(F_m(x_1),F_m(x_2),...,F_m(x_N))\in \mathrm{P}\subset R^N$。经过 T 次迭代后就得到一系列的集成模

型 $F_m \in \{F_1, F_2, ..., F_T\}$，分别将他们作用于训练集，就得到一系列 N 维的点 $P_m = F_m(X) \in \{F_1(X), F_2(X), ..., F_T(X)\} = \{P_1, P_2, ..., P_T\}$。随着 m 的增大我们的模型 F_m 的训练误差 $\Psi(Y, F_m(X))$ 会越来越小，如果无限迭代下去，理想情况下训练误差就会收敛到一个极小值，相应的序列 P_m 会收敛到一个极小值点 P^*。这是不是有种似曾相似的感觉，想一想在凸优化里面梯度下降法，是不是很像？我们就把 $F_m(X)$ 看成是在 N 维空间中的一个一个的点，而损失函数就是这个 N 维空间中的一个函数，我们要用某种逐步逼近的算法来求解损失函数的极小值（最小值）。把我们的思维切换到 N 维空间 P 中，当我们已经迭代了前 m-1 次后，得到的集成模型如（式 1.5 所示），此时我们在 N 维空间 P 中处于点 $P_{m-1} = (F_{m-1}(x_1), F_{m-1}(x_2), ..., F_{m-1}(x_N))$，那么要搜索的下一个点就是 P_m，这个点该怎么搜索呢？这个问题是 boosting 算法的核心，也正是不同版本的 Boosting 算法的重点却别之处。下一节我们会回顾 GradientBoosting 算法。

2 Gradient Boosting 算法原理

之所以要在这里回顾 Gradient Boosting 算法，是因为 XGBoost 算法与该算法很相似，理解了 Gradient Boosting 算法对理解 XGBoost 有很大的帮助。你也可以跳过这一小节，而直接看下一节对 XGBoost 的讲解。

第 1 节我们提到的对于搜索下一个点 P_m 的方法，优化理论里有一种简单的算法就是线性搜索。假设我们的损失函数至少是一阶连续可导的，那么我们可以在点 P_{m-1} 处对损失函数求导，得到损失函数在该点的负梯度 ρ_{m-1}

$$\begin{aligned} -\nabla\Psi(P_{m-1}) &= -\frac{\partial\Psi(P_{m-1})}{\partial P_{m-1}} \\ &= -\frac{\partial\Psi(F_{m-1}(X))}{\partial F_{m-1}(X)} \\ &= \left(-\frac{\partial\psi(F_{m-1}(x_1))}{\partial F_{m-1}(x_1)}, -\frac{\partial\psi(F_{m-1}(x_2))}{\partial F_{m-1}(x_2)}, ..., -\frac{\partial\psi(F_{m-1}(x_N))}{\partial F_{m-1}(x_N)}\right)^T \end{aligned} \quad \text{（式 2.1）}$$

然后在点 P_{m-1} 处沿着负梯度方向即在射线 $P_{m-1} - \lambda\nabla\Psi(P_{m-1}), \lambda \geq 0$ 上搜索使得损失最小的点 P_m

$$
\begin{aligned}
P_m &= \arg\min_{\lambda} \Psi(P_{m-1} - \lambda \nabla \Psi(P_{m-1})) \\
&= \arg\min_{\lambda} \Psi(F_{m-1}(X) - \lambda \frac{\partial \Psi(F_{m-1}(X))}{\partial F_{m-1}(X)}) \\
&= F_{m-1}(X) - \lambda_m \frac{\partial \Psi(F_{m-1}(X))}{\partial F_{m-1}(X)}
\end{aligned}
\qquad （式 2.2）
$$

而这种方法正是 Gradient Boosting 算法采用的搜索方法。下面是 Gradient Boosting 算法的通用流程：

Algorithm 1: Gradient_Boost

1. $F_0(\mathbf{x}) = \arg\min_{\rho} \sum_{i=1}^{N} \Psi(y_i, \rho)$
2. For $m = 1$ to M do:
3. $\quad \tilde{y}_i = -\left[\frac{\partial \Psi(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}, \; i = 1, N$
4. $\quad \mathbf{a}_m = \arg\min_{\mathbf{a},\beta} \sum_{i=1}^{N} [\tilde{y}_i - \beta h(\mathbf{x}_i; \mathbf{a})]^2$
5. $\quad \rho_m = \arg\min_{\rho} \sum_{i=1}^{N} \Psi(y_i, F_{m-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i; \mathbf{a}_m))$
6. $\quad F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \rho_m h(\mathbf{x}; \mathbf{a}_m)$
7. endFor

end Algorithm

图 2.1 Gradient Boosting

从图 2.1 看到，Gradient Boosting 算法首先利用训练数据训练一个常数值函数

$$F_0(x) = \arg\min_{\rho} \sum_{i=1}^{N} \Psi(y_i, \rho)$$

相当于我们在搜索最优解时选定一个初始点 $P_0 = F_0(X)$，如果我们的损失函数是平方损失的话，$F_0(X) = \sum_{i=1}^{N} \omega_i y_i$，即为加权平均值。其实，这个初始点是可以随便选取的，如果损失函数是凸函数，则一定可以搜索到最小值点，如果不是凸函数，我们选取的不同的初始点有可得到不同的极小值点。即可能得到不同的集成模型。

在选定了初始点（生成了第一个弱学习器）后，下一个弱学习器该怎么生成呢？Gradient Boosting 的思路是：既然我们有了第一个初始点 $P_0 = F_0(X)$，那么我们就可以求出损失函数 $\Psi(y, F(X))$ 在点 $F_0(X)$ 处的负梯度

$$-\nabla_0 \Psi(F_0(X)) = (-\frac{\partial \psi(F_0(x_1))}{\partial F_0(x_1)}, -\frac{\partial \psi(F_0(x_2))}{\partial F_0(x_2)}, ..., -\frac{\partial \psi(F_0(x_N))}{\partial F_0(x_N)})^T$$

每一个样本都对应该负梯度的一个分量

$$(x_i, y_i) \leftrightarrow y_i = -\frac{\partial \psi(F_0(x_i))}{\partial F_0(x_i)}, i = 1,2,...,N$$

我们现在想要生成下一个弱学习器 f_1，让它在训练集 D 上的输出 $p_1 = (f_1(x_1), f_1(x_2),..., f_1(x_N))$ 与 $-\nabla_0 \Psi(F_0(X))$ 尽量同向，即利用 $f_1(X)$ 得到负梯度的近似值。我们用 y_i 代替原始训练集中的目标变量（类标签或实数值）y_i，得到一个新的样本 (x_i, y_i)，每一个样本都这样做暂时性的替换后得到新的训练集 $D = \{(x_1, y_1),(x_2, y_2),...,(x_N, y_N)\}$。因为 $y_i \in R$，训练 $f_1(X)$ 就变成了一个回归问题。

通常我们会采用 CART 回归树来训练弱学习器（此时我们称 Gradient Boosting 为 GBDT，或者 GBRT 等，有很多不同的叫法，本质都一样），训练 f_1 就跟我们平常使用 CART 回归树训练模型是一样的过程了。在得到 f_1 后，就相当于模拟出了负梯度方向，那么又如何确定步长 λ_1 呢？在 CART 回归树中，每一个叶子节点的输出的值都是一样的，Gradient Boosting 算是将负梯度按照叶子节点为单位划分为 L（L 为叶子节点数）个区间，然后在各个叶子节点中对处于该节点上的负梯度分量所确定的方向上寻找最佳步长 $\lambda_{1l}, l \in \{1,2,...,L\}$，这样就会大大简化最佳步长的确定问题，尤其是当我们的损失函数采取绝对值损失时，每个叶子节点的输出值直接为该节点上残差的中位数。具体的大家可以参考论文[3]，里面有详细的介绍。

按照同样的方法，我们逐步迭代，直到达到预先设置的迭代次数 T 后，我们就得到了最终的模型

$$F_T(X) = \sum_{t=0}^{T} \alpha_t f_t(X)$$

即在给定一个初始点 $P_0 = F_0(X)$，经过 T 次迭代后我们搜索到了点 $P_T = F_T(X)$（注意，时刻想着 $F(X)$ 是一个 N 维的点），想要提高精度（降低误差）就增大搜索次数 T。

这就是 Gradient Boosting 算法的主要思想，它是通过优化经验损失函数在通过迭代反复拟合损失函数的负梯度并利用线性搜索法来生成最优的弱学习器 f_m 和系数 α_m。而 XGBoost 算法是通过优化结构化损失函数（加入了正则项的损失函数，可以起到降低过拟合的风险）[8]来实现弱学习器的生成，并且 XGBoost 算法没有采用上述搜索方法，而是直接利用了损失函数的一阶导数和二阶导数值，并通过预排序、加权分位数、稀疏矩阵识别以及缓存识别等技术来大大提高了算

法的性能。

3 XGBoost 算法原理

3.1 XGBoost 的损失函数

XGBoost 算法是基于树的 boosting 算法，并且其优化的目标函数引入了正则化项

$$\begin{aligned}\Psi(y,F(X)) &= \sum_{i=1}^{N}\psi(y_i,F(x_i)) + \sum_{m=0}^{T}\Omega(f_m) \\ &= \sum_{i=1}^{N}\psi(y_i,F(x_i)) + \sum_{m=0}^{T}(\gamma L_m + \frac{1}{2}\lambda \| \omega_m \|^2)\end{aligned} \quad \text{（式 3.1）}$$

上式中 L_m 表示第 m 次迭代中生成的树模型 f_m 的叶子节点数，$\omega_m = (\omega_{m1},\omega_{m2},...,\omega_{tL_m})$ 表示 f_m 各个叶子节点的输出值。γ 和 λ 是正则化系数，从直观上我们能看出这两个值能对模型的复杂度和输出值起到很强的控制，当 γ 和 λ 都为零时，只剩下经验风险部分，即生成的树的规模和输出值不受限制。在引入了正则化项后，算法会选择简单而性能优良的模型，上式中右端的正则化项 $\sum_{m=0}^{T}\Omega(f_m)$ 只是用来在每次迭代中抑制弱学习器 $f_m(X)$ 过拟合的，并不参与最终模型的集成。另外 XGBoost 要求 ψ 至少是二阶连续可导的凸函数。

XGBoost 算法也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，模型形式如下：

$$F_T(X) = \sum_{m=0}^{T} f_m(X) \quad \text{（式 3.2）}$$

这跟采用绝对值损失函数的 GBDT 的形式是一样的。

接着第 2 节的内容我们继续。假设在点 $P_{m-1} = F_{m-1}(X)$ 处，Gradient Boosting 算法是学习一个弱学习器 f_m 来近似损失函数在 $P_{m-1} = F_{m-1}(X)$ 处的负梯度。而 XGBoost 则是直接先求损失函数在该点处的泰勒近似值，然后通过最小化该近似损失函数值来训练弱学习器 f_m。在点 $P_{m-1} = F_{m-1}(X)$ 处对损失函数做一阶近似（省去二阶及以上的部分）：

$$\begin{aligned}\Psi_m &= \sum_{i=1}^{N}\psi(y_i, F_{m-1}(x_i)+f_m(x_i))+\Omega(f_m)\\ &\approx \sum_{i=1}^{N}[\psi(y_i, F_{m-1}(x_i))+g_i f_m(x_i)+\frac{1}{2}h_i f_m^2(x_i)]+\Omega(f_m) \qquad \text{（式 3.3）}\\ &= \Psi_m\end{aligned}$$

上式中 $g_i=\frac{\partial\psi(y_i,F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}$ 为损失函数在点 $P_{m-1}(X)$ 处对第 i 个分量 $F_{m-1}(x_i)$ 的一阶偏导数，$h_i=\frac{\partial^2\psi(y_i,F_{m-1}(x_i))}{\partial F_{m-1}(x_i)^2}$ 为损失函数在点 $P_{m-1}(X)$ 处对第 i 个分量 $F_{m-1}(x_i)$ 的二阶偏导数。这样我们就可以以 Ψ_m 作为我们的近似优化目标函数。这里解释下为什么二阶展开式是这样的形式，请时刻记住：$P_m=F_m(X)$ 是一个 N 维的点。那么 $f_m(X)$ 也是一个 N 维的向量，因为

$$\begin{aligned}F_m(X) &= \sum_{m=0}^{T} f_{m-1}(X)+f_m(X)\\ &= F_{m-1}(X)+f_m(X)\\ &= P_{m-1}(X)+f_m(X)\\ &= P_m(X)\end{aligned}$$

所以，我们可以把 $f_m(X)=(f_m(x_1), f_m(x_2),..., f_m(x_N))$ 看成是在 N 维空间中的点 $P_m(X)$ 相对于点 $P_{m-1}(X)$ 的一个增量。那么我们现在的目标就是要找到一个最优的增量使得损失函数在下一个点的损失最小

$$f_m(X)=\arg\min_{f(X)}\sum_{i=1}^{N}\psi(y_i,F_{m-1}(x_i)+f(x_i))$$

我们对（式 3.3）做个变形，

$$\begin{aligned}\Psi_m &= \sum_{i=1}^{N}\psi(y_i,F_{m-1}(x_i))+\sum_{i=1}^{N}[g_i f_m(x_i)+\frac{1}{2}h_i f_m^2(x_i)]+\Omega(f_m)\\ &= A+\sum_{i=1}^{N}[g_i f_m(x_i)+\frac{1}{2}h_i f_m^2(x_i)]+\Omega(f_m)\end{aligned}$$

上式右端第一项对于第 m 次迭代来说是常数，将其去掉不会影响我们优化的结果，因此，优化函数变成如下形式：

$$\overline{\Psi}_m=\sum_{i=1}^{N}[g_i f_m(x_i)+\frac{1}{2}h_i f_m^2(x_i)]+\Omega(f_m) \qquad \text{（式 3.4）}$$

3.2 确定各叶子节点的最优输出值

假设我们在第 m 轮迭代中已经生成了一颗树，它有 L 个叶子节点 $\{l_1,l_2,...,l_L\}$，设 $I_j=\{i\,|\,q(x_i)=j\}$ 表示落在第 j 个叶子节点的样本的索引号，q 表示第 m 轮生成的树的结果，它将一个样本 x 映射到一个相对应的叶子节点。那么（式 3.4）可以写成如下形式

$$\begin{aligned}\overline{\Psi}_m &= \sum_{i=1}^{N}[g_i f_m(x_i)+\frac{1}{2}h_i f_m^2(x_i)]+\Omega(f_m)\\ &= \sum_{j=1}^{L}[(\sum_{i\in I_j} g_i)\omega_j+\frac{1}{2}(\lambda+\sum_{i\in I_j} h_i)\omega_j^2]+\gamma L \qquad \text{（式 3.5）}\\ &= \sum_{j=1}^{L} f(\omega_j)+\gamma L\end{aligned}$$

原论文中 λ 写在后面，这里为了清晰直观，我把它挪到前面了。在第 m 轮迭代中由于 γL 是常数，所以最小化 $\overline{\Psi}_m$ 就可以通过分别在每个叶子节点中极小化损失值 $f(\omega_j)$ 来实现。因为前面我们要求损失函数 $\psi(y,F(x))$ 是凸函数，因此有 $\sum_{i\in I_j} h_i \geq 0$，另外正则化系数 $\lambda \geq 0$，所以

$$f(\omega_j)=(\sum_{i\in I_j} g_i)\omega_j+\frac{1}{2}(\lambda+\sum_{i\in I_j} h_i)\omega_j^2 \qquad \text{（式 3.6）}$$

是凸函数，那么它存在极小值，又由于 $\omega_j \in R$，所以这是一个无约束的凸优化问题。直接对式 3.6 求导数并令其等于零，得到第 j 个叶子节点的最优输出值

$$\omega_j^* = -\frac{\sum_{i\in I_j} g_j}{\lambda+\sum_{i\in I_j} h_j} \qquad \text{（式 3.7）}$$

这样就可以计算出 $\overline{\Psi}_m$ 的最小值为

$$\overline{\Psi}_m(q) = -\frac{1}{2}\sum_{j=1}^{L}\frac{(\sum_{i\in I_j} g_j)^2}{\lambda+\sum_{i\in I_j} h_j}+\gamma L \qquad \text{（式 3.8）}$$

其中 q 表示第 m 轮迭代中生成的树的结构，上式的值代表了第 m 次迭代中生

成的模型带来的损失减小值，明显 $\overline{\Psi}_m(q) \le 0$（因为 $g \ge 0, h \ge 0, L > 0, \gamma \ge 0, \lambda \ge 0$），$\overline{\Psi}_m(q)$ 的值越小，则第 m 个模型加入后得到的新集成模型 $F_m(X)$ 相对于 $F_{m-1}(X)$ 减小的损失函数值就越大。

从（式 3.7）和（式 3.8）我们可以看到，使得第 m 轮生成的模型 $F_m(X) = F_{m-1}(X) + f_m(X)$ 的损失最小的树 q，其每个叶子节点的最优输出值确定只由损失函数 ψ 在点 $F_{m-1}(X)$ 的一阶导数 $g = (g_1, g_2, ..., g_N)$ 和二阶导数 $h = (h_1, h_2, ..., h_N)$ 决定。所以在第 m 轮迭代时，只要我们预先计算出 g 和 h，在训练生成一颗树模型后就可以直接算出每个叶子节点的最有输出值。那么现在的关键是根据什么条件来生成一颗合适的树呢？这个问题的解决，不得不佩服陈天奇同学的智商，他提出在生成树的过程中，在每个叶子节点的分裂判定时利用“最大损失减小值”的原则来选择最优的分裂属性和分列点。

3.3 分裂条件

传统的 CART 回归树的分裂条件是均方差（或绝对偏差等等）减小最大的点作为分裂点。但 XGBoost 没有这么干，它在树的生成过程中依然只借助了 3.1 小节提到的损失函数的一阶和二阶导数值，这些值都是在第 m 轮中提前计算好的。

假设我们从一个初始的节点（根节点）开始，I 表示该节点上所有样本的索引集，现在我们要对该节点进行分裂，设 I_L, I_R 分别为分裂后左右子节点中样本的索引集，并且有 $I = I_L \cup I_R$。那么根据（式 3.8）我们可以计算出分裂前后第 m 个树模型的损失函数值减小量为

$$\Delta\Psi = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\lambda + \sum_{i\in I_L} h_i} + \frac{(\sum_{i\in I_R} g_i)^2}{\lambda + \sum_{i\in I_R} h_i} - \frac{(\sum_{i\in I} g_i)^2}{\lambda + \sum_{i\in I} h_i}\right] - \gamma \qquad \text{（式 3.9）}$$

这个公式跟我们之前遇到的信心增益或基尼值增量的公式是一个道理。XGBoost 就是利用这个公式计算出的值作为分裂条件，在每一个节点的分裂中寻找最优的分裂和属性和分裂点。这样我们就能顺利地得到我们在第 m 轮迭代中所需要的最优的模型 $f_m(X)$。

3.4 弱学习器的集成

通过上述算法经过 T 此迭代我们得到 T+1 个弱学习器

$\{f_0(X), f_1(X), ..., f_T(X)\}$，那么通什么样的形式将他们集成起来呢？答案是直接将 T+1 个模型相加，只不过为了防止过拟合，XGBoost 也采用了 shrinkage 方法来降低过拟合的风险，其模型集成形式如下

$$F_m(X) = F_{m-1}(X) + \eta f_m(X), 0 < \eta \le 1$$

Shrinkage 技术最早由 Friedman 于 1999 年在论文《Greedy Function Approximation: A Gradient Bo osting Machine》中提出。通常η取较小的值时会使得模型的泛化能力较强，该参数与迭代次数 T 高度负相关（较大的η值对应着相对较小的迭代次数 T）。为什么要乘上一个η值呢？大致有以下三个原因：

（1） 我们的训练数据由于样本量有限，所以是存在一定的信息丢失的；
（2） 训练数据中存在一些噪声信息；
（3） 如（式 1.1）所示，我们应该是在由每一个不同的x对应一个维度所构成的 N（N 可能无穷大）维空间中去拟合模型$F^*(X)$，该模型对于一个确定的x所给出的预测的损失的期望$E_y\psi(y, F^*(x))$应该最小。但是，我们的算法是把训练数据中的每一个样本都当做了一个不同的维度，这样必然会导致$P_m = (P_{m1}, P_{m2}, ..., P_{mN})$与真实的$P_m$存在偏差。

上述三个因素的存在，如果我们依然在每一次都取最优的$f_m(X)$，就极有可能导致各个弱学习器过拟合。因此通过对$f(X)$给出的最优值乘以一个小于 1 的系数，让该模型尽量只学习到有用的信息，以抑制它过拟合。

关于η和迭代次数 T 的取值，可以通过交叉验证得到合适的值，通常针对不同问题，其具体值是不同的。一般来说，当条条件允许时（如对模型训练时间没有要求等）可以设置一个较大的迭代次数 T，然后针对该 T 值利用交叉验证来确定一个合适的η值。但η的取值也不能太小，否则模型达不到较好的效果[6]。

除了利用 Shrinkage 技术来抑制过拟合外，XGBoost 在构造树模型过程中也借用了随机森林中随机选择一定量的特征子集来确定最优分裂点的做法，以达到抑制过拟合的目的。特征子集越大则每个弱学习器的偏差就越小，但方差就越大。这个具体的比例需要我们利用交叉验证来权衡确定。

看到这里，基本上你就把 XGBoost 的原理理解清楚了。在陈天奇同学的论文中还对 XGBoost 的几处性能优化技术进行了讨论。下面我们简单介绍下。

4 XGBoost 的优化

这一节主要是介绍 XGBoost 算法对计算性能和预测结果的优化方法，但也会跟着原版论文的内容介绍下 XGBoost 算法实现的一些技术。

4.1 分裂点的搜索算法

在分裂结点时，跟大多数基于树的算法一样，XGBoost 也实现了一种完全搜索式的算法（Exact Greedy Algorithm）。这种搜索算法会遍历一个特征上所有可能的分裂点，分别计算其损失减小量，然后选择最优的分裂点。这种算法在 scikit-learn、R 语言中的 gbm 包等常见库中都实现了。其优点是能够搜索到最优的分裂点，但缺点就是当数据量很大时比较浪费计算资源。当我们的训练数据量很大，以致内存无法一次性加载完或者是在分布式计算环境下，该算法就无能为力。

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

图 4.1 Exact Greedy Algorithm for Split Finding

为此 XGBoost 同时实现了一种基于特征值分位数的近似搜索算法。该算法首先是根据训练数据中某个特征 K 的取值的分布情况，选取 l 个分位数 $S_k = \{s_{k1}, s_{kl}, ..., s_{kl}\}$，利用这 l 个分位数将特征 K 的取值区间截断为 $l+1$ 个子区间。然后分别在每个子区间中将该区间中各个样本在当前迭代中的一阶导数和二阶导数求和，得到该区间的一阶和二阶导数的汇总统计量。后面在寻找最优分裂点时就只搜索这 l 个分位数点，并从这 l 各分位数点中找到最好的分裂点作为该特征上最优分裂点的近似。

XGBoost 算法在计算特征 k 的分位数点时并没有按照均匀分为的方式来将样本等分到各个区间，而是采用了加权求分位数的方式。那这个权重是怎么来的呢？

我们回头看（式 3.4），将它稍微做个变形得到如下形式

$$\begin{aligned}\overline{\Psi}_m &= \sum_{i=1}^{N}[g_i f_m(x_i)+\frac{1}{2}h_i f_m^2(x_i)]+\Omega(f_m)\\ &= \sum_{i=1}^{N}\frac{1}{2}h_i[2\frac{g_i}{h_i}f_m(x_i)+f_m^2(x_i)]+\Omega(f_m)\\ &= \sum_{i=1}^{N}\frac{1}{2}h_i(f_m(x_i)-\frac{g_i}{h_i})^2+\sum_{i=1}^{N}(\frac{g_i^2}{2h_i}-2g_i f_m(x_i))+\Omega(f_m)\end{aligned}$$

上式中的第二项在原文中用 constant 来表示，它可以被看成是某种正则化项，那么第一项就是一个标准的平方损失的表达式，只不过此时样本 x_i 对应的输出的值变成了 $\frac{g_i}{h_i}$，而其权重恰好就是 h_i。所以，在 XGBoost 算法里，在第 m 轮迭代中计算特征值的分位数时是把损失函数在各个样本点的二阶导数值作为其样本权重。样本权重代表的就是概率，概率越大表示该点出现的次数或是该点附近的值出现的次数就越多。XGBoost 算法依据该权重分步来选取分位数。其具体分位数计算如下：

（1） 首先将训练集样本根据特征 k 的值从小到大的顺序进行排列；

（2） 选定一个合适的权重累积阈值 ε，将总的权重分成 $\frac{1}{\varepsilon}$ 等份（等分数即为区间数+2）；

（3） 对于 $i=0,1,2,...,[\frac{1}{\varepsilon}]$ 循环执行如下操作

设定 $h_{iv}=0$，然后从第一个样本开始遍历，并累加各样本的权重 $h_{iv}+=h_i$，当 $h_{iv}>\varepsilon$ 时，停止累加，并将已经遍历过样本归入第 i 个区间，得到第 i+2 个分位数。

对于第（3）步求分位数中，第 1 个和最后一个分位数是例外，它们默认为特征 k 的最小值和最大值。我这里描述得有点复杂，如果大家不明白建议直接看论文原文，会比较清楚。

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.

图 4.2 Approximate Algorithm for Split Finding

对分位数的计算 XGBoost 也实现了两种形式，一种是在每次迭代中只生成一次全局的分位数，然后在树的各层构造中均使用相同的分位数值。另一种是每一次分裂后都对每一个新结点上所有样本按照同样方式重新计算加权分位数。两种方法各有优劣，大家根据具体情况进行选择。

4.2 稀疏数据的自动识别

在现实生活中，我们的数据会包含大量缺失值，零值或是由于我们在做哑变量时引入的零值，这种情况下我们就成训练数据是稀疏的，对应位置上的值就是缺失的。

XGBoost 算法实现了一种自动处理稀缺失据的算法。即通过默认值形式或是我们认为指定某个值为缺失值的形式，在构造树模型过程中，选择最优分裂点时只利用那些非缺失值，而缺失值一概跳过。

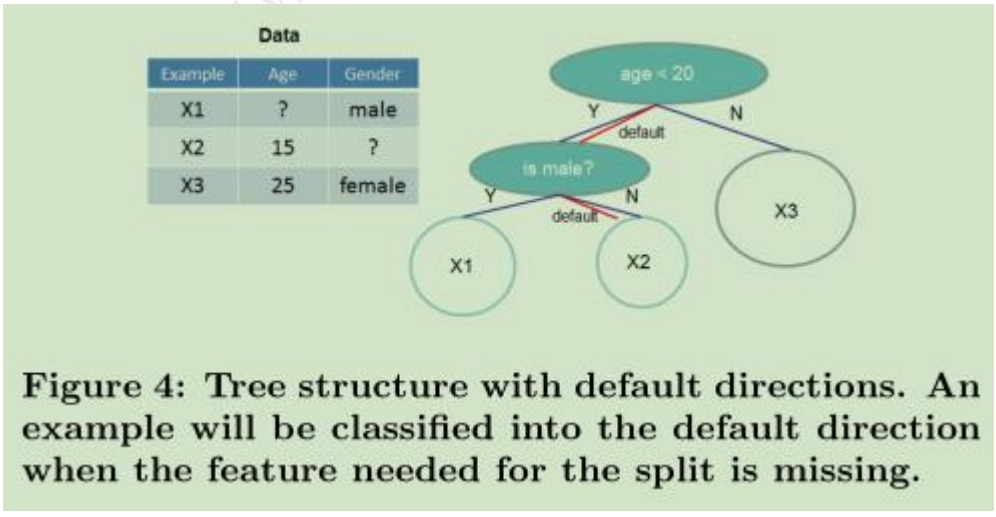

Figure 4: Tree structure with default directions. An example will be classified into the default direction when the feature needed for the split is missing.

图 4.1 对含有缺失值的样本的分裂方向的处理

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

图 4.2 Sparsity-aware Split Finding

如果某个样本在特征 k 上的值缺失，分别考虑将其分到左子节点和右子节点（把在该特征上含缺失值的所有样本要么全部放进左子节点要么全部放进右子节点），根据不同的选择方向可以得到两个不同的模型，最后以两个模型中最优的模型对含有缺失值的样本分裂方向为最优方向。所以这个含有缺失值样本的分裂方向是根据训练数据自动学习到的。算法在原论文写得很详细，大家可以直接看原论文。

4.3 其他计算性能优化

这里简要说一下 XGBoost 实现中基于数据块的预排序。为了减小 CART 树构造过程中的计算开销，XGBoost 在构造 CART 树之前先构造一个经过压缩处理的数据块，该数据块中存储的是按照各特征值排序后对样本索引的指针。每次寻找最优分裂点时只需要在这个数据块中依次遍历，然后根据其指针来提取各样本的一阶导数和二阶导数即可。这样就不必反复地进行排序了，大大提高了计算的速度。

由于在寻找最优分裂点时 XGBoost 使用的是数据块中各个特征值对应的指针

来间接获取对应样本的导数值，虽然数据块中的数据是连续的，但其指针对应的样本的导数值不是连续存储的，所以就需要频繁的访问内存中不连续的区域，而缓存中通常存放的是我们最近一次访问地址的附近区域的数据，所以对于我们要访问不连续的数据是极有可能不同时存在于缓存中的。如果要获取的样本点的导数值恰好不在当前缓存中，CPU 就必须访问内存，这就延长了处理时间。为了解决这个问题，XGBoost 采用了 Cache-aware Access 技术来降低处理时间。

另外，当我们需要处理的训练数据非常大，以至于内存无法一次加载完时，传统的方法就难以应付。XGBoost 提出了 Block Compression 技术，它将整个训练数据集切割成多个块（block），并将其存储在外存中。在训练模型时 XGBoost 通过一个单独的线程来将需要处理的某个数据块预加载到主存缓冲区，以实现处理操作在数据上的无缝衔接。XGBoost 不是简单的将数据分块存放在外存中，而是采用一种按列压缩的方法对数据进行压缩，并且在读入时利用一个单独的线程来实时解压。另外，XGBoost 还通过将数据分散到多个可用的磁盘上，并为每一个磁盘设置一个数据加载线程，这样就能同时从多个磁盘向内存缓冲区加载数据，模型训练线程就能自由地从每个缓冲区读取数据，这大大增加了数据吞吐量。这些技术的实现一是为了提告模型训练的效率，二是为了提升 XGBoost 处理大量数据的能力，充分利用计算机有限的资源。

5 总结

到这里总算把 XGBoost 的基本原理讲解完了，有些地方为了讲解清楚写的有点啰嗦。读完陈天奇同学的这篇论文，我不得不叹服他的创新能力，文中利用二阶泰勒展开式近似损失函数、用二阶导数表示样本权重、用导数统计量来作为树构造的分裂条件以及其他的的性能优化技术都让你感到兴奋。读完后你是否感觉这些技术似乎自己都懂，但确实自己想不到这个点上来。陈天奇同学能够想到这些，离不开他无数个日日夜夜的努力及其对自身天赋的合理利用。我们不应该只有仰望，更应该有坚韧的步伐，坚持自己的梦想，坚定自己的信念。只要我们深入透彻地熟悉理解了自己所在领域后，才会有创新。因为平时自己比较忙，很多学习笔记都是写在自己的 world 笔记本上，没有多少时间发布到网上。之所以今天写下这篇文章，是希望能进自己最大努力简单易懂的将 XGBoost 算法原理讲清楚，能够对想要学习 XGBoost 的朋友有所帮助。本人水平有限，难免会有些错误之处，希望大家能不吝指正。如果有疑问或建议，欢迎发邮件到我邮箱。

参考文献

[1] 《XGBoost: A Scalable Tree Boosting System》(Tianqi Chen and Carlos,2016)

[2] 何通.https://cos.name/2015/03/xgboost/?utm_source=tuicool&utm_medium=referral

[3] 《Greedy Function Approximation: A Gradient Bo osting Machine》(Jerome H. Friedman,1999)

[4] 《Additive Logistic Regression: a Statistical View of Boosting》（Jerome Friedman，Trevor Hastie，Robert Tibshirani..1998）
[5] 《Improved Boosting Algorithms Using Confidence-rated Predictions》（Schapire，Singer.1999）
[6] 《A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting》（Yoav Freund ,Robert E. Schapire.1996）
[7] 《An Empirical Comparison of Supervised Learning Algorithms》ICML2006
[8] 《统计学习方法》（李航，2012）