机器学习算法系列（36）：GBDT算法原理深入解析

梯度提升（Gradient boosting）是一种用于回归、分类和排序任务的机器学习技术[1]，属于Boosting算法族的一部分。Boosting是一族可将弱学习器提升为强学习器的算法，属于集成学习（ensemble learning）的范畴。Boosting方法基于这样一种思想：对于一个复杂任务来说，将多个专家的判断进行适当的综合所得出的判断，要比其中任何一个专家单独的判断要好。通俗地说，就是“三个臭皮匠顶个诸葛亮”的道理。梯度提升同其他boosting方法一样，通过集成（ensemble）多个弱学习器，通常是决策树，来构建最终的预测模型。

Boosting、bagging和stacking是集成学习的三种主要方法。不同于bagging方法，boosting方法通过分步迭代（stage-wise）的方式来构建模型，在迭代的每一步构建的弱学习器都是为了弥补已有模型的不足。Boosting族算法的著名代表是AdaBoost，AdaBoost算法通过给已有模型预测错误的样本更高的权重，使得先前的学习器做错的训练样本在后续受到更多的关注的方式来弥补已有模型的不足。与AdaBoost算法不同，梯度提升方法在迭代的每一步构建一个能够沿着梯度最陡的方向降低损失（steepest-descent）的学习器来弥补已有模型的不足。经典的AdaBoost算法只能处理采用指数损失函数的二分类学习任务[2]，而梯度提升方法通过设置不同的可微损失函数可以处理各类学习任务（多分类、回归、Ranking等），应用范围大大扩展。另一方面，AdaBoost算法对异常点（outlier）比较敏感，而梯度提升算法通过引入bagging思想、加入正则项等方法能够有效地抵御训练数据中的噪音，具有更好的健壮性。这也是为什么梯度提升算法（尤其是采用决策树作为弱学习器的GBDT算法）如此流行的原因，有种观点认为GBDT是性能最好的机器学习算法，这当然有点过于激进又固步自封的味道，但通常各类机器学习算法比赛的赢家们都非常青睐GBDT算法，由此可见该算法的实力不可小觑。

基于梯度提升算法的学习器叫做GBM(Gradient Boosting Machine)。理论上，GBM可以选择各种不同的学习算法作为基学习器。现实中，用得最多的基学习器是决策树。为什么梯度提升方法倾向于选择决策树（通常是CART树）作为基学习器呢？这与决策树算法自身的优点有很大的关系。决策树可以认为是if-then规则的集合，易于理解，可解释性强，预测速度快。同时，决策树算法相比于其他的算法需要更少的特征工程，比如可以不用做特征标准化，可以很好的处理字段缺失的数据，也可以不用关心特征间是否相互依赖等。决策树能够自动组合多个特征，它可以毫无压力地处理特征间的交互关系并且是非参数化的，因此你不必担心异常值或者数据是否线性可分（举个例子，决策树能轻松处理好类别A在某个特征维度x的末端，类别B在中间，然后类别A又出现在特征维度x前端的情况）不过，单独使用决策树算法时，有容易过拟合缺点。所幸的是，通过各种方法，抑制决策树的复杂性，降低单颗决策树的拟合能力，再通过梯度提升的方法集成多个决策树，最终能够很好的解决过拟合的问题。由此可见，梯度提升方法和决策树学习算法可以互相取长补短，是一对完美的搭档。至于抑制单颗决策树的复杂度的方法有很多，比如限制树的最大深度、限制叶子节点的最少样本数量、限制节点分裂时的最少样本数量、吸收

bagging的思想对训练样本采样（subsample），在学习单颗决策树时只使用一部分训练样本、借鉴随机森林的思路在学习单颗决策树时只采样一部分特征、在目标函数中添加正则项惩罚复杂的树结构等。现在主流的GBDT算法实现中这些方法基本上都有实现，因此GBDT算法的超参数还是比较多的，应用过程中需要精心调参，并用交叉验证的方法选择最佳参数。

本文对GBDT算法原理进行介绍，从机器学习的关键元素出发，一步一步推导出GBDT算法背后的理论基础，读者可以从这个过程中了解到GBDT算法的来龙去脉。对于该算法的工程实现，本文也有较好的指导意义，实际上对机器学习关键概念元素的区分对应了软件工程中的“开放封闭原则”的思想，基于此思想的实现将会具有很好的模块独立性和扩展性。

一、机器学习的关键元素

先复习下监督学习的关键概念：模型（model）、参数（parameters）、目标函数（objective function）

模型就是所要学习的条件概率分布或者决策函数，它决定了在给定特征向量x时如何预测出目标y。定义$x_i \in R^d$为训练集中的第i个训练样本，则线性模型（linear model）可以表示为：$\hat{y} = \sum_j w_j x_{ij}$。模型预测的分数$\hat{y_i}$在不同的任务中有不同的解释。例如在逻辑回归任务中，$1/(1+exp(-\hat{y}_i))$表示模型预测为正例的概率；而在排序学习任务中，$\hat{y_i}$表示排序分。

参数就是我们要从数据中学习得到的内容。模型通常是由一个参数向量决定的函数。例如，线性模型的参数可以表示为：$\Theta = \{w_j | j = 1, \cdots, d\}$

目标函数通常定义为如下形式：

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

其中，$L(\Theta)$是损失函数，用来衡量模型拟合训练数据的好坏程度；$\Omega(\Theta)$称之为正则项，用来衡量学习到的模型的复杂度。训练集上的损失（Loss）定义为：$L = \sum_{i=1}^{n} l(y_i, \hat{y}_i)$。常用的损失函数有平方损失（square loss）：$l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$；Logistic损失：$l(y_i, \hat{y}_i) = y_i ln(1 + e^{y_i}) + (1 - y_i) ln(1 + e^{\hat{y}_i})$。常用的正则项有L1范数$\Omega(w) = \lambda \| w \|_1$和L2范数$\Omega(w) = \lambda \| w \|_2$。Ridge regression就是指使用平方损失和L2范数正则项的线性回归模型；Lasso regression就是指使用平方损失和L1范数正则项的线性回归模型；逻辑回归（Logistic regression）指使用logistic损失和L2范数或L1范数正则项的线性模型。

目标函数之所以定义为损失函数和正则项两部分，是为了尽可能平衡模型的偏差和方差（Bias Variance Trade-off）。最小化目标函数意味着同时最小化损失函数和正则项，损失函数最小化表明模型能够较好的拟合训练数据，一般也预示着模型能够较好地拟合真实数据（groud true）；另一方面，对正则项的优化鼓励算法学习到较简单的模型，简单模型一般在测试样本上的预测结

果比较稳定、方差较小（奥坎姆剃刀原则）。也就是说，优化损失函数尽量使模型走出欠拟合的状态，优化正则项尽量使模型避免过拟合。

从概念上区分模型、参数和目标函数给学习算法的工程实现带来了益处，使得机器学习的各个组成部分之间耦合尽量松散。

二、加法模型

GBDT算法可以看成是由K棵树组成的加法模型：

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

其中F为所有树组成的函数空间，以回归任务为例，回归树可以看作为一个把特征向量映射为某个score的函数。该模型的参数为：$\Theta = \{f_1, f_2, \cdots, f_K\}$。于一般的机器学习算法不同的是，加法模型不是学习d维空间中的权重，而是直接学习函数（决策树）集合。

上述加法模型的目标函数定义为：$Obj = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$，其中表示决策树的复杂度，那么该如何定义树的复杂度呢？比如，可以考虑树的节点数量、树的深度或者叶子节点所对应的分数的L2范数等等。

如何来学习加法模型呢?

解这一优化问题，可以用前向分布算法（forward stagewise algorithm）。因为学习的是加法模型，如果能够从前往后，每一步只学习一个基函数及其系数（结构），逐步逼近优化目标函数，那么就可以简化复杂度。这一学习过程称之为Boosting。具体地，我们从一个常量预测开始，每次学习一个新的函数，过程如下：

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

那么，在每一步如何决定哪一个函数f被加入呢？指导原则还是最小化目标函数。
在第t步，模型对x_i的预测为：$\hat{y}_i^t = \hat{y}_i^{t-1} + f_t(x_i)$，其中$f_t(x_i)$为这一轮我们要学习的函数（决策树）。这个时候目标函数可以写为：

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

举例说明，假设损失函数为平方损失（square loss），则目标函数为：

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

其中$(\hat{y}_i^{t-1} - y_i)$，称之为残差（residual）。因此，使用平方损失函数时，GBDT算法的每一步在生成决策树时只需要拟合前面的模型的残差。

> 泰勒公式：设n是一个正整数，如果定义在一个包含a的区间上的函数f在a点处$n+1$次可导，那么对于这个区间上的任意x都有：
>
> $$f(x) = \sum_{n=0}^{N} \frac{f^{(n)}(a)}{n!}(x-a)^n + R_n(x)$$
>
> ，其中的多项式称为函数在a处的泰勒展开式，$R_n(x)$是泰勒公式的余项且是$(x-a)^n$的高阶无穷小。----维基百科

根据泰勒公式把函数$f(x+\Delta x)$在点处二阶展开，可得到如下等式：

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

由等式(1)可知，目标函数是关于变量$\hat{y}_i^{t-1} + f_t(x_i)$若把变量$\hat{y}_i^{t-1}$看成是等式(3)中的$x$，把变量$f_t(x_i)$看成是等式(3)中的$\Delta x$，则等式(1)可转化为：

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

其中，g_i定义为损失函数的一阶导数，即$g_i = \partial_{\hat{y}^{t-1}} l(y_i, \hat{y}^{t-1})$；$h_i$定义为损失函数的二阶导数，即$h_i = \partial^2_{\hat{y}^{t-1}} l(y_i, \hat{y}^{t-1})$。 假设损失函数为平方损失函数，则$g_i = \partial_{\hat{y}^{t-1}}(\hat{y}^{t-1} - y_i)^2 = 2(\hat{y}^{t-1} - y_i)$，$h_i = \partial^2_{\hat{y}^{t-1}}(\hat{y}^{t-1} - y_i)^2 = 2$，把$g_i$和$h_i$代入等式(4)即得等式(2)。由于函数中的常量在函数最小化的过程中不起作用，因此我们可以从等式(4)中移除掉常量项，得：

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

由于要学习的函数仅仅依赖于目标函数，从等式(5)可以看出只需为学习任务定义好损失函数，并为每个训练样本计算出损失函数的一阶导数和二阶导数，通过在训练样本集上最小化等式(5)即可求得每步要学习的函数$f(x)$，从而根据加法模型等式(0)可得最终要学习的模型。

二、GBDT算法

一颗生成好的决策树，假设其叶子节点个数为T，该决策树是由所有叶子节点对应的值组成的向量$w \in R^T$，以及一个把特征向量映射到叶子节点索引（Index）的函数$q: R^d \to \{1, 2, \cdots, T\}$组成的。因此，决策树可以定义为$f_t(x) = w_{q(x)}$。

决策树的复杂度可以由正则项$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2$来定义，即决策树模型的复杂度由生成的树的叶子节点数量和叶子节点对应的值向量的L2范数决定。

定义集合$I_j = \{i \mid q(x_i) = j\}$为所有被划分到叶子节点的训练样本的集合。等式(5)可以根据树的叶子节点重新组织为T个独立的二次函数的和：

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

定义$G_j = \sum_{i\in I_j} g_i$，$H_j = \sum_{i\in I_j} h_i$，则等式(6)可写为：

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

假设树的结构是固定的，即函数$q(x)$确定，令函数$Obj^{(t)}$的一阶导数等于0，即可求得叶子节点对应的值为：$w_j^* = -\frac{G_j}{H_j+\lambda}$此时，目标函数的值为

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

综上，为了便于理解，单颗决策树的学习过程可以大致描述为：

1. 枚举所有可能的树结构q
2. 用等式(8)为每个q计算其对应的分数Obj，分数越小说明对应的树结构越好。
3. 根据上一步的结果，找到最佳的树结构，用等式(7)为树的每个叶子节点计算预测值

然而，可能的树结构数量是无穷的，所以实际上我们不可能枚举所有可能的树结构。通常情况下，我们采用贪心策略来生成决策树的每个节点。

1. 从深度为0的树开始，对每个叶节点枚举所有的可用特征
2. 针对每个特征，把属于该节点的训练样本根据该特征值升序排列，通过线性扫描的方式来决定该特征的最佳分裂点，并记录该特征的最大收益（采用最佳分裂点时的收益）
3. 选择收益最大的特征作为分裂特征，用该特征的最佳分裂点作为分裂位置，把该节点生长出左右两个新的叶节点，并为每个新节点关联对应的样本集
4. 回到第1步，递归执行到满足特定条件为止

在上述算法的第二步，样本排序的时间复杂度为$O(nlogn)$，假设共用K个特征，那么生成一颗深度为K的树的时间复杂度为$O(dKnlogn)$。具体实现可以进一步优化计算复杂度，比如可以缓存每个特征的排序结果等。

如何计算每次分裂的收益呢？假设当前节点记为,分裂之后左孩子节点记为，右孩子节点记为，则该分裂获得的收益定义为当前节点的目标函数值减去左右两个孩子节点的目标函数值之和：$Gain = Obj_C - Obj_L - Obj_R$，具体地，根据等式(8)可得：

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

其中，$-\gamma$项表示因为增加了树的复杂性（该分裂增加了一个叶子节点）带来的惩罚。等式(9)还可以用来计算输入特征的相对重要程度，具体见下一节

最后，总结一下GBDT的学习算法：

1. 算法每次迭代生成一颗新的决策树
2. 在每次迭代开始之前，计算损失函数在每个训练样本点的一阶导数g_i和二阶导数h_i
3. 通过贪心策略生成新的决策树，通过等式(7)计算每个叶节点对应的预测值
4. 把新生成的决策树$f_t(x)$添加到模型中：$\hat{y}_i^t = \hat{y}_i^{t-1} + f_t(x_i)$

通常在第四步，我们把模型更新公式替换为：$\hat{y}_i^t = \hat{y}_i^{t-1} + \epsilon f_t(x_i)$，其中$\epsilon$称之为步长或者学习率。增加因子的目的是为了避免模型过拟合。

三、特征重要度

集成学习因具有预测精度高的优势而受到广泛关注，尤其是使用决策树作为基学习器的集成学习算法。树的集成算法的著名代码有随机森林和GBDT。随机森林具有很好的抵抗过拟合的特性，并且参数（决策树的个数）对预测性能的影响较小，调参比较容易，一般设置一个比较大的数。GBDT具有很优美的理论基础，一般而言性能更有优势。

基于树的集成算法还有一个很好的特性，就是模型训练结束后可以输出模型所使用的特征的相对重要度，便于我们选择特征，理解哪些因素是对预测有关键影响，这在某些领域（如生物信息学、神经系统科学等）特别重要。本文主要介绍基于树的集成算法如何计算各特征的相对重要度。

3.1 优势

- 使用不同类型的数据时，不需要做特征标准化/归一化
- 可以很容易平衡运行时效率和精度；比如，使用boosted tree作为在线预测的模型可以在机器资源紧张的时候截断参与预测的树的数量从而提高预测效率
- 学习模型可以输出特征的相对重要程度，可以作为一种特征选择的方法
- 模型可解释性好
- 对数据字段缺失不敏感
- 能够自动做多组特征间的interaction，具有很好的非性线性

3.2 特征重要度的计算

Friedman在GBM的论文中提出的方法：

特征j的全局重要度通过特征j在单颗树中的重要度的平均值来衡量：

$$\hat{J_j^2} = \frac{1}{M}\sum_{m=1}^{M}\hat{J_j^2}(T_m)$$

其中，M是树的数量。特征j在单颗树中的重要度的如下：

$$\hat{J_j^2}(T) = \sum_{t=1}^{L-1}\hat{i_t^2}1(v_t = j)$$

其中，L为树的叶子节点数量，$L-1$即为树的非叶子节点数量（构建的树都是具有左右孩子的二叉树），是和节点t相关联的特征，$\hat{i_t^2}$是节点分裂之后平方损失的减少值。

3.3 实现代码

为了更好的理解特征重要度的计算方法，下面给出scikit-learn工具包中的实现，代码移除了一些不相关的部分。

下面的代码来自于GradientBoostingClassifier对象的feature_importances属性的计算方法：

```
def feature_importances_(self):
    total_sum = np.zeros((self.n_features, ), dtype=np.float64)
    for tree in self.estimators_:
        total_sum += tree.feature_importances_
    importances = total_sum / len(self.estimators_)
    return importances
```

其中，self.estimators_是算法构建出的决策树的数量，tree.feature_importances_ 是单棵树的特征重要度向量，其计算方法如下：

```
cpdef compute_feature_importances(self, normalize=True):
    """Computes the importance of each feature (aka variable)."""
    while node != end_node:
        if node.left_child != _TREE_LEAF:
            # ... and node.right_child != _TREE_LEAF:
            left = &nodes[node.left_child]
            right = &nodes[node.right_child]
            importance_data[node.feature] += (
                node.weighted_n_node_samples * node.impurity -
                left.weighted_n_node_samples * left.impurity -
                right.weighted_n_node_samples * right.impurity)
        node += 1
    importances /= nodes[0].weighted_n_node_samples
    return importances
```

上面的代码关键点是两个：

第一点：weighted_n_node_samples : array of int, shape [node_count]
weighted_n_node_samples[i] holds the weighted number of training samples reaching node i.

第二点：impurity : array of double, shape [node_count]
impurity[i] holds the impurity (i.e., the value of the splitting criterion) at node i.

当然上面的代码经过了简化，保留了核心思想。计算所有的非叶子节点在分裂时加权不纯度的减少，减少得越多说明特征越重要

不纯度的减少实际上就是该节点此次分裂的收益，因此我们也可以这样理解，节点分裂时收益越大，该节点对应的特征的重要度越高。关于收益的定义就是上一节中等式(9)的定义。

参考资料

[1] Gradient Boosting 的更多内容
[2] XGBoost是一个优秀的GBDT开源软件库，有多种语言接口
[3] Pyramid是一个基于Java语言的机器学习库，里面也有GBDT算法的介绍和实现
[4] Friedman的论文《Greedy function approximation: a gradient boosting machine》是比较早的GBDT算法文献，但是比较晦涩难懂，不适合初学者，高阶选手可以进一步学习
[5] "A Gentle Introduction to Gradient Boosting"是关于Gradient Boosting的一个通俗易懂的解释，比较适合初学者或者是已经对GBDT算法原理印象不深的从业者
[6] 关于GBDT算法调参的经验和技巧可以参考这两篇博文：《GBM调参指南》、《XGBoost调参指南》，作者使用的算法实现工具来自于著名的Python机器学习工具scikit-learn
[7] GBDT算法在搜索引擎排序中的应用可以查看这篇论文《Web-Search Ranking with Initialized Gradient Boosted Regression Trees 》，这篇论文提出了一个非常有意思的方法，用一个已经训练好的随机森林模型作为GBDT算法的初始化，再用GBDT算法优化最终的模型，取得了很好的效果

[1] Feature Selection for Ranking using Boosted Trees
[2] Gradient Boosted Feature Selection
[3] Feature Selection with Ensembles, Artificial Variables, and Redundancy Elimination