**XGBoost算法梳理**

**1. 算法原理**

XGBoost是GBDT的一种高效实现，主要从以下三个方面做了优化：

* 一是算法本身的优化：   
  在算法的弱学习器模型选择上，对比GBDT只支持决策树，还可以直接很多其他的弱学习器。在算法的损失函数上，除了本身的损失，还加上了正则化部分。在算法的优化方式上，GBDT的损失函数只对误差部分做负梯度（一阶泰勒）展开，而XGBoost损失函数对误差部分做二阶泰勒展开，更加准确。
* 二是算法运行效率的优化：   
  对每个弱学习器，比如决策树建立的过程做并行选择，找到合适的子树分裂特征和特征值。在并行选择之前，先对所有的特征的值进行排序分组，方便前面说的并行选择。对分组的特征，选择合适的分组大小，使用CPU缓存进行读取加速。将各个分组保存到多个硬盘以提高IO速度。
* 三是算法健壮性的优化：   
  对于缺失值的特征，通过枚举所有缺失值在当前节点是进入左子树还是右子树来决定缺失值的处理方式。算法本身加入了L1和L2正则化项，可以防止过拟合，泛化能力更强。

**2. 损失函数**

在GBDT损失函数$L(y, f\_{t-1}(x)+ h\_t(x))$的基础上，我们加入正则化项如下：   
$\begin{align} \Omega(h\_t) = \gamma J + \frac{\lambda}{2}\sum\limits\_{j=1}^Jw\_{tj}^2 \end{align}$   
这里的$J$是叶子节点的个数，而$w\_{tj}$是第$j$个叶子节点的最优值。   
最终XGBoost的损失函数可以表达为：   
$\begin{align} L\_t=\sum\limits\_{i=1}^mL(y\_i, f\_{t-1}(x\_i)+ h\_t(x\_i)) + \gamma J + \frac{\lambda}{2}\sum\limits\_{j=1}^Jw\_{tj}^2 \end{align}$   
最终我们要极小化上面这个损失函数，得到第$t$个决策树最优的所有$J$个叶子节点区域和每个叶子节点区域的最优解$w\_{tj}$。XGBoost没有和GBDT一样去拟合泰勒展开式的一阶导数，而是期望直接基于损失函数的二阶泰勒展开式来求解。现在我们来看看这个损失函数的二阶泰勒展开式：   
$\begin{align} L\_t & = \sum\limits\_{i=1}^mL(y\_i, f\_{t-1}(x\_i)+ h\_t(x\_i)) + \gamma J + \frac{\lambda}{2}\sum\limits\_{j=1}^Jw\_{tj}^2 \\ & \approx \sum\limits\_{i=1}^m( L(y\_i, f\_{t-1}(x\_i)) + \frac{\partial L(y\_i, f\_{t-1}(x\_i) }{\partial f\_{t-1}(x\_i)}h\_t(x\_i) + \frac{1}{2}\frac{\partial^2 L(y\_i, f\_{t-1}(x\_i) }{\partial f\_{t-1}^2(x\_i)} h\_t^2(x\_i)) + \gamma J + \frac{\lambda}{2}\sum\limits\_{j=1}^Jw\_{tj}^2 \end{align}$   
为了方便，我们把第i个样本在第t个弱学习器的一阶和二阶导数分别记为：   
$\begin{align} g\_{ti} = \frac{\partial L(y\_i, f\_{t-1}(x\_i) }{\partial f\_{t-1}(x\_i)}, \; h\_{ti} = \frac{\partial^2 L(y\_i, f\_{t-1}(x\_i) }{\partial f\_{t-1}^2(x\_i)} \end{align}$   
则我们的损失函数现在可以表达为：   
$\begin{align} L\_t \approx \sum\limits\_{i=1}^m( L(y\_i, f\_{t-1}(x\_i)) + g\_{ti}h\_t(x\_i) + \frac{1}{2} h\_{ti} h\_t^2(x\_i)) + \gamma J + \frac{\lambda}{2}\sum\limits\_{j=1}^Jw\_{tj}^2 \end{align}$   
损失函数里面$L(y\_i, f\_{t-1}(x\_i))$是常数，对最小化无影响，可以去掉，同时由于每个决策树的第$j$个叶子节点的取值最终会是同一个值$w\_{tj}$,因此我们的损失函数可以继续化简。   
$\begin{align} L\_t & \approx \sum\limits\_{i=1}^m g\_{ti}h\_t(x\_i) + \frac{1}{2} h\_{ti} h\_t^2(x\_i)) + \gamma J + \frac{\lambda}{2}\sum\limits\_{j=1}^Jw\_{tj}^2 \\ & = \sum\limits\_{j=1}^J (\sum\limits\_{x\_i \in R\_{tj}}g\_{ti}w\_{tj} + \frac{1}{2} \sum\limits\_{x\_i \in R\_{tj}}h\_{ti} w\_{tj}^2) + \gamma J + \frac{\lambda}{2}\sum\limits\_{j=1}^Jw\_{tj}^2 \\ & = \sum\limits\_{j=1}^J [(\sum\limits\_{x\_i \in R\_{tj}}g\_{ti})w\_{tj} + \frac{1}{2}( \sum\limits\_{x\_i \in R\_{tj}}h\_{ti}+ \lambda) w\_{tj}^2] + \gamma J \end{align}$   
我们把每个叶子节点区域样本的一阶和二阶导数的和单独表示如下：   
$\begin{align} G\_{tj} = \sum\limits\_{x\_i \in R\_{tj}}g\_{ti},\; H\_{tj} = \sum\limits\_{x\_i \in R\_{tj}}h\_{ti} \end{align}$   
最终损失函数的形式可以表示为：   
$\begin{align} L\_t = \sum\limits\_{j=1}^J [G\_{tj}w\_{tj} + \frac{1}{2}(H\_{tj}+\lambda)w\_{tj}^2] + \gamma J \end{align}$

**3. 分裂结点算法**

* 贪婪算法：
* 从树的深度为0开始，美意街店都遍历所有特征。对于某个特征，先按照该特征里的值进行排序，然后线性扫描该特征来决定最好的分割点，最后在所有特征里选择分割后Gain最高的那个特征。   
  这时，就有两种后续。一种是当最好的分割的情况下，Gain为负时就停止树的生长，这样的话效率会比较高也简单，但是这样就放弃了未来可能会有更好的情况。另一种就是一直分别到最大深度，然后进行修剪，递归的把划分叶子得到的Gain为负的收回。一般来说，后一种要好一些，于是我们采用后一种。

近似算法

因为精确贪心算法需要遍历所有特征和取值，当数据量非常大的时候，无法将所有数据同时加载进内存时，精确贪心算法会非常耗时，XGBoost的作者引进了近似算法。近似算法对特征值进行了近似处理，即根据每个特征$k$的特征值分布，确定出候选切分点$S\_k={s\_{k1}, s\_{k2},...,s\_{kl}}$，即按特征分布将连续的特征值划分到$l$个候选点对应的桶（buckets）中，并且对每个桶中每个样本的$G\_i, H\_i$进行累加。

## 4. 正则化

XGBoost的损失函数添加了正则化项，并且同时使用了L1、L2正则用以控制模型的复杂度，正则项里包含了树的叶子节点个数、每个叶子节点权重（叶结点的socre值）的平方和。（详见2）

## 5. 对缺失值处理

XGBoost处理缺失值的方法和其他树模型不同。根据作者陈天奇在论文中3.4介绍，XGBoost把缺失值当做稀疏矩阵来对待，本身的在节点分裂时不考虑的缺失值的数值。缺失值数据会被分到左子树和右子树分别计算损失，选择较优的那一个。如果训练中没有数据缺失，预测时出现了数据缺失，那么默认被分类到右子树。