# 机器学习时代三大神器

# GBDT、XGBoost、LightGBM

本文主要简要的比较了常用的boosting算法的一些区别，从AdaBoost到LightGBM，包括AdaBoost，GBDT，XGBoost，LightGBM四个模型的简单介绍，一步一步从原理到优化对比。

## AdaBoost原理

原始的AdaBoost算法是在算法开始的时候，为每一个样本赋上一个权重值，初始的时候，大家都是一样重要的。在每一步训练中得到的模型，会使得数据点的估计有对有错，我们就在每一步结束后，增加分错的点的权重，减少分对的点的权重，这样使得某些点如果老是被分错，那么就会被“重点关注”，也就被赋上一个很高的权重。然后等进行了N次迭代（由用户指定），将会得到N个简单的分类器（basic learner），然后我们将它们组合起来（比如说可以对它们进行加权、或者让它们进行投票等），得到一个最终的模型。

关键字：样本的加权分布，各基分类器的加权投票

## GBDT概述

GBDT（梯度提升树，Gradient Boosting Decison Tree）中的树都是回归树，GBDT用来做回归预测，调整后也可以用于分类（设定阈值，大于阈值为正例，反之为负例），可以发现多种有区分性的特征以及特征组合。GBDT是把所有树的结论累加起来做最终结论的，GBDT的**核心**就在于，每一棵树学的是之前所有树结论和的残差(负梯度)，这个残差就是一个加预测值后能得真实值的累加量。比如A的真实年龄是18岁，但第一棵树的预测年龄是12岁，差了6岁，即残差为6岁。那么在第二棵树里我们把A的年龄设为6岁去学习，如果第二棵树真的能把A分到6岁的叶子节点，那累加两棵树的结论就是A的真实年龄；如果第二棵树的结论是5岁，则A仍然存在1岁的残差，第三棵树里A的年龄就变成1岁，继续学。

Boosting的最大好处在于，每一步的残差计算其实变相地增大了分错instance的权重，而已经分对的instance则都趋向于0。这样后面的树就能越来越专注那些前面被分错的instance。

### Gradient Boost与AdaBoost的区别

* Gradient Boost每一次的计算是为了减少上一次的残差(residual)，而为了消除残差，我们可以在残差减少的梯度(Gradient)方向上建立一个新的模型。所以说，在Gradient Boost中，每个新的模型的建立是为了使得之前模型的残差往梯度方向减少。Shrinkage（缩减）的思想认为，每次走一小步逐渐逼近结果的效果，要比每次迈一大步很快逼近结果的方式更容易避免过拟合。即它不完全信任每一个棵残差树，它认为每棵树只学到了真理的一小部分，累加的时候只累加一小部分，通过多学几棵树弥补不足。本质上，Shrinkage为每棵树设置了一个weight，累加时要乘以这个weight，但和Gradient并没有关系。
* Adaboost是另一种boost方法，它按分类对错，分配不同的weight，计算cost function时使用这些weight，从而让“错分的样本权重越来越大，使它们更被重视”

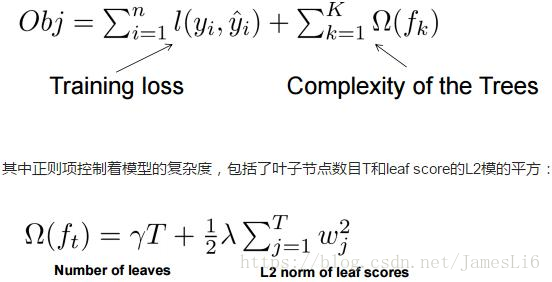
### Gradient Boost优缺点

* 优点：它的非线性变换比较多，表达能力强，而且不需要做复杂的特征工程和特征变换。
* 缺点：Boost是一个串行过程，不好并行化，而且计算复杂度高，同时不太适合高维稀疏特征。

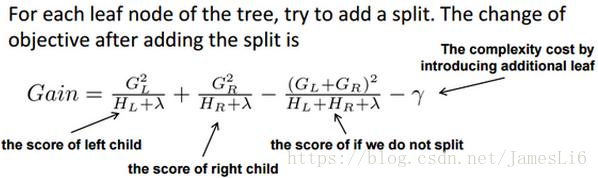
## XGBoost

XGBoost能自动利用cpu的多线程，而且适当改进了gradient boosting，加了**剪枝**，控制了模型的复杂程度

* 传统GBDT以CART作为基分类器，特指梯度提升决策树算法，而XGBoost还支持线性分类器(gblinear)，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）。
* 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工具支持自定义代价函数，只要函数可一阶和二阶求导。
* xgboost在代价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、每个叶子节点上输出的score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。



* xgboost中树节点分裂时所采用的公式：
* Shrinkage（缩减），相当于学习速率（xgboost中的eta）。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。（传统GBDT的实现也有学习速率）
* 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。
* 对缺失值的处理。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。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次迭代的代价函数里包含了前面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。(特征粒度上的并行，block结构，预排序)



* 这个公式形式上跟ID3算法、CART算法是一致的，都是用分裂后的某种值减去分裂前的某种值，从而得到增益。为了限制树的生长，我们可以加入阈值，当增益大于阈值时才让节点分裂，上式中的gamma即阈值，它是正则项里叶子节点数T的系数，所以xgboost在优化目标函数的同时相当于做了预剪枝。另外，上式中还有一个系数lambda，是正则项里leaf score的L2模平方的系数，对leaf score做了平滑，也起到了防止过拟合的作用，这个是传统GBDT里不具备的特性。
* XGBoost实现层面
* 内置交叉验证方法
* 能够输出特征重要性文件辅助特征筛选

XGBoost优势小结：

* 显式地将树模型的复杂度作为正则项加在优化目标
* 公式推导里用到了二阶导数信息，而普通的GBDT只用到一阶
* 允许使用列抽样(column(feature) sampling)来防止过拟合，借鉴了Random Forest的思想，sklearn里的gbm好像也有类似实现。
* 实现了一种分裂节点寻找的近似算法，用于加速和减小内存消耗。
* 节点分裂算法能自动利用特征的稀疏性。
* 样本数据事先排好序并以block的形式存储，利于并行计算
* penalty function Omega主要是对树的叶子数和叶子分数做惩罚，这点确保了树的简单性。
* 支持分布式计算可以运行在MPI，YARN上，得益于底层支持容错的分布式通信框架rabit。

## LightGBM

lightGBM：基于决策树算法的分布式梯度提升框架。

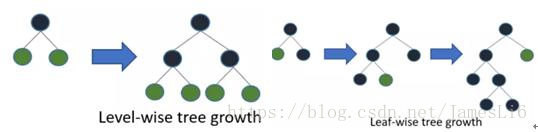
### LightGBM与XGBoost的区别：

* 切分算法(切分点的选取)
* 占用的内存更低，只保存特征离散化后的值，而这个值一般用8位整型存储就足够了，内存消耗可以降低为原来的1/8。
* 降低了计算的代价：预排序算法每遍历一个特征值就需要计算一次分裂的增益，而直方图算法只需要计算k次（k可以认为是常数），时间复杂度从O(#data#feature)优化到O(k#features)。(相当于LightGBM牺牲了一部分切分的精确性来提高切分的效率，实际应用中效果还不错)
* 空间消耗大，需要保存数据的特征值以及特征排序的结果(比如排序后的索引，为了后续快速计算分割点)，需要消耗两倍于训练数据的内存
* 时间上也有较大开销，遍历每个分割点时都需要进行分裂增益的计算，消耗代价大
* 对cache优化不友好，在预排序后，特征对梯度的访问是一种随机访问，并且不同的特征访问的顺序不一样，无法对cache进行优化。同时，在每一层长树的时候，需要随机访问一个行索引到叶子索引的数组，并且不同特征访问的顺序也不一样，也会造成较大的cache miss。
* XGBoost使用的是pre-sorted算法（对所有特征都按照特征的数值进行预排序，基本思想是对所有特征都按照特征的数值进行预排序；然后在遍历分割点的时候用O(#data)的代价找到一个特征上的最好分割点最后，找到一个特征的分割点后，将数据分裂成左右子节点。优点是能够更精确的找到数据分隔点；但这种做法有以下缺点
* LightGBM使用的是histogram算法，基本思想是先把连续的浮点特征值离散化成k个整数，同时构造一个宽度为k的直方图。在遍历数据的时候，根据离散化后的值作为索引在直方图中累积统计量，当遍历一次数据后，直方图累积了需要的统计量，然后根据直方图的离散值，遍历寻找最优的分割点；

### 优点在于

决策树生长策略上：

* XGBoost采用的是带深度限制的level-wise生长策略，Level-wise过一次数据可以能够同时分裂同一层的叶子，容易进行多线程优化，不容易过拟合；但不加区分的对待同一层的叶子，带来了很多没必要的开销（因为实际上很多叶子的分裂增益较低，没必要进行搜索和分裂）
* LightGBM采用leaf-wise生长策略，每次从当前所有叶子中找到分裂增益最大（一般也是数据量最大）的一个叶子，然后分裂，如此循环；但会生长出比较深的决策树，产生过拟合（因此 LightGBM 在leaf-wise之上增加了一个最大深度的限制，在保证高效率的同时防止过拟合）。



* Histogram 做差加速。一个容易观察到的现象：一个叶子的直方图可以由它的父亲节点的直方图与它兄弟的直方图做差得到。通常构造直方图，需要遍历该叶子上的所有数据，但直方图做差仅需遍历直方图的k个桶。利用这个方法，lightgbm可以在构造一个叶子的直方图后，可以用非常微小的代价得到它兄弟叶子的直方图，在速度上可以提升一倍。
* 直接支持类别特征：lightgbm优化了对类别特征的支持，可以直接输入类别特征，不需要额外的0/1展开。并在决策树算法上增加了类别特征的决策规则。

分布式训练方法上(并行优化)：

* 在特征并行算法中，通过在本地保存全部数据避免对数据切分结果的通信；
* 在数据并行中使用分散规约(Reduce scatter)把直方图合并的任务分摊到不同的机器，降低通信和计算，并利用直方图做差，进一步减少了一半的通信量。基于投票的数据并行(Parallel Voting)则进一步优化数据并行中的通信代价，使通信代价变成常数级别。
* 特征并行的主要思想是在不同机器在不同的特征集合上分别寻找最优的分割点，然后在机器间同步最优的分割点。
* 数据并行则是让不同的机器先在本地构造直方图，然后进行全局的合并，最后在合并的直方图上面寻找最优分割点。
* 原始
* LightGBM针对这两种并行方法都做了优化，
* Cache命中率优化
* 基于直方图的稀疏特征优化
* DART(Dropout + GBDT)
* GOSS(Gradient-based One-Side Sampling):一种新的Bagging(row subsample)方法,前若干轮(1.0f / gbdtconfig->learning\_rate)不Bagging;之后Bagging时, 采样一定比例g(梯度)大的样本

LightGBM优点小结(相较于XGBoost)：

* 速度更快
* 内存消耗更低