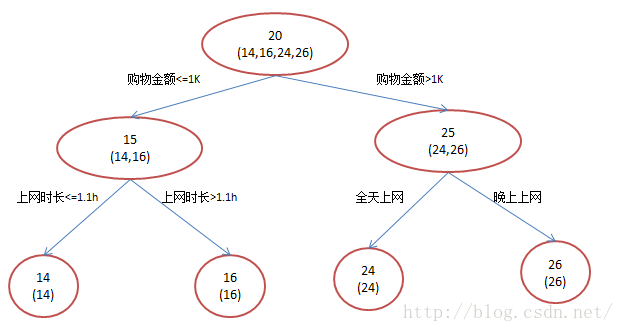
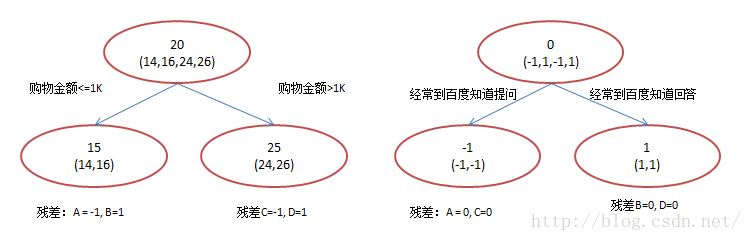
**1. GBDT执行示例**

训练集只有4个人，A,B,C,D，他们的年龄分别是14,16,24,26。其中A、B分别是高一和高三学生；C,D分别是应届毕业生和工作两年的员工。如果是用一棵传统的回归决策树来训练，会得到如下图所示结果：



使用GBDT来做这件事，由于数据太少，我们限定叶子节点做多有两个，即每棵树都只有一个分枝，并且限定只学两棵树。我们会得到如下图所示结果：



在第一棵树分枝和图1一样，由于A,B年龄较为相近，C,D年龄较为相近，他们被分为两拨，每拨用平均年龄作为预测值。此时计算残差（残差的意思就是： A的预测值 + A的残差 = A的实际值），所以A的残差就是16-15=1（注意，A的预测值是指前面所有树累加的和，这里前面只有一棵树所以直接是15，如果还有树则需要都累加起来作为A的预测值）。进而得到A,B,C,D的残差分别为-1,1，-1,1。然后我们拿残差替代A,B,C,D的原值，到第二棵树去学习，如果我们的预测值和它们的残差相等，则只需把第二棵树的结论累加到第一棵树上就能得到真实年龄了。这里的数据显然是我可以做的，第二棵树只有两个值1和-1，直接分成两个节点。此时所有人的残差都是0，即每个人都得到了真实的预测值。

换句话说，现在A,B,C,D的预测值都和真实年龄一致了：

A: 14岁高一学生，购物较少，经常问学长问题；预测年龄A = 15 – 1 = 14

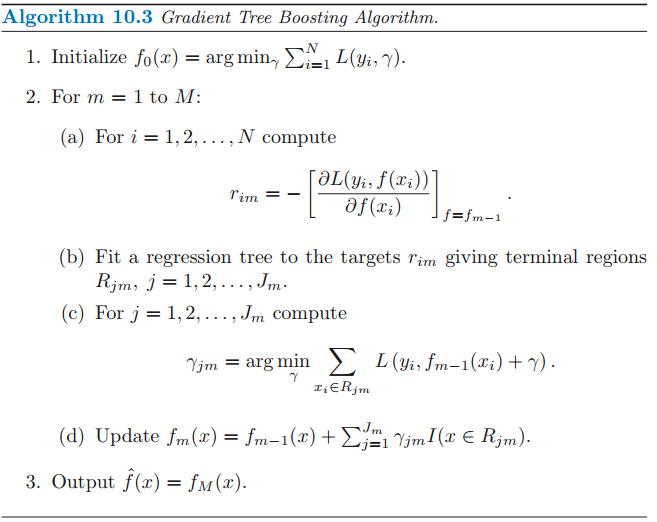
B: 16岁高三学生；购物较少，经常被学弟问问题；预测年龄B = 15 + 1 = 16

C: 24岁应届毕业生；购物较多，经常问师兄问题；预测年龄C = 25 – 1 = 24

D: 26岁工作两年员工；购物较多，经常被师弟问问题；预测年龄D = 25 + 1 = 26

那么哪里体现了Gradient呢？其实回到第一棵树结束时想一想，无论此时的cost function是什么，是均方差还是均差，只要它以误差作为衡量标准，残差向量(-1, 1, -1, 1)都是它的全局最优方向，这就是Gradient。

**2. GBDT代码示例**



算法步骤解释：

1. 初始化，估计使损失函数极小化的常数值，它是只有一个根节点的树，即ganma是一个常数值。

（a）计算损失函数的负梯度在当前模型的值，将它作为残差的估计

（b）估计回归树叶节点区域，以拟合残差的近似值

（c）利用线性搜索估计叶节点区域的值，使损失函数极小化

（d）更新回归树

1. 得到输出的最终模型 f(x)

**3. GBDT和随机森林**

根据个体学习器的生成方式，目前的集成学习方法大致可以分为两大类，即个体学习器间存在强依赖关系，必须串行生成的序列化方法，以及个体学习器间不存在强依赖关系、可同时生成的并行化方法；前者的代表是Boosting，后者的代表是Bagging和“随机森林”。

梯度提升：Boosting的改进，GBDT应用于分类问题的建模更为tricky一些。

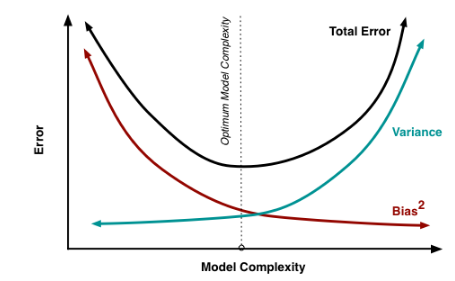
**为什么GBDT树深度很小的时候就能得到较高的精度，而随机森林需要更大的树深？**

Boosting主要关注降低偏差，因此Boosting能基于泛化性能相当弱的学习器构建出很强的集成； Bagging主要关注降低方差，因此它在不剪枝的决策树、神经网络等学习器上效用更为明显。

就机器学习算法来说，其泛化误差可以分解为两部分，偏差（bias)和方差(variance)。这个可由下图的式子导出（这里用到了概率论公式D(X)=E(X^2)-[E(X)]^2）。偏差指的是算法的期望预测与真实预测之间的偏差程度，反应了模型本身的拟合能力；方差度量了同等大小的训练集的变动导致学习性能的变化，刻画了数据扰动所导致的影响。这个有点儿绕，不过你一定知道过拟合。

  如下图所示，当模型越复杂时，拟合的程度就越高，模型的训练偏差就越小。但此时如果换一组数据可能模型的变化就会很大，即模型的方差很大。所以模型过于复杂的时候会导致过拟合。

  当模型越简单时，即使我们再换一组数据，最后得出的学习器和之前的学习器的差别就不那么大，模型的方差很小。还是因为模型简单，所以偏差会很大。



模型复杂度与偏差方差的关系图

  也就是说，当我们训练一个模型时，偏差和方差都得照顾到，漏掉一个都不行。

  对于Bagging算法来说，由于我们会并行地训练很多不同的分类器的目的就是降低这个方差(variance) ,因为采用了相互独立的基分类器多了以后，h的值自然就会靠近.所以对于每个基分类器来说，目标就是如何降低这个偏差（bias),所以我们会采用深度很深甚至不剪枝的决策树。

  对于Boosting来说，每一步我们都会在上一轮的基础上更加拟合原数据，所以可以保证偏差（bias）,所以对于每个基分类器来说，问题就在于如何选择variance更小的分类器，即更简单的分类器，所以我们选择了深度很浅的决策树。

**4. GBDT在Spark MLlib中的实现**

**算法主要步骤：**

1. 先训练第一棵树：

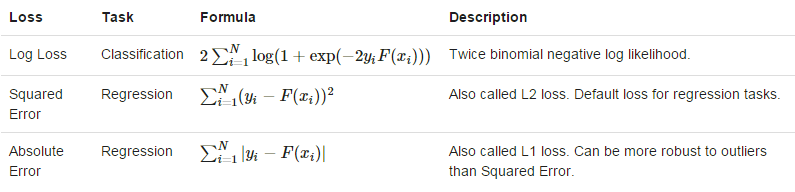
RandomForest.run()创建一颗决策树。

2. computeInitialPredictionAndError：

Compute the initial predictions and errors for a dataset for the first

iteration of gradient boosting. 计算初始的预测和误差

损失计算有以下3种，Regression的默认Loss为Squared Error:



3. 更新预测和误差：

预测：

prediction + tree.rootNode.predictImpl(features).prediction \* weight

误差：

val err = label - prediction

err \* err

梯度：

gradient: - 2.0 \* (label - prediction)

新的训练数据：LabeledPoint(-loss.gradient(pred, point.label), point.features)

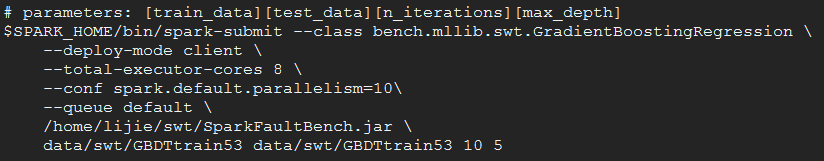
循环直到：

// 1. Reduction in error is less than the validationTol or

// 2. If the error increases, that is if the model is overfit.

测试：

100,000条数据样本，1000个特征，正负样例比为1:1，数据稀疏度为0的数据进行测试。



第一棵树的构建时间：

init: 2.010038704

total: 6.626371424

findSplits: 1.859002398

findBestSplits: 4.610553325

chooseSplits: 4.609807934

构建梯度提升树：

building tree 4: 7.188305229

init: 0.006802764

total: 71.4665078

building tree 3: 6.670765754

building tree 6: 7.036429536

building tree 9: 6.628537742

building tree 0: 9.481328615

building tree 8: 6.876688026

building tree 2: 6.564188997

building tree 5: 6.647070516

building tree 7: 7.205187681

building tree 1: 7.05390194

**TODO数据流图**

**5. GBDT的优化**

**XGBoost**

对比Spark MLlib中的GBDT的对比通信时间（End-to-end Time）、计算时间（Per iteration cost）

**LightGBM**

评价标准：训练速度、内存消耗、准确率

一般：预排序，找分裂点。

LightGBM的改进：

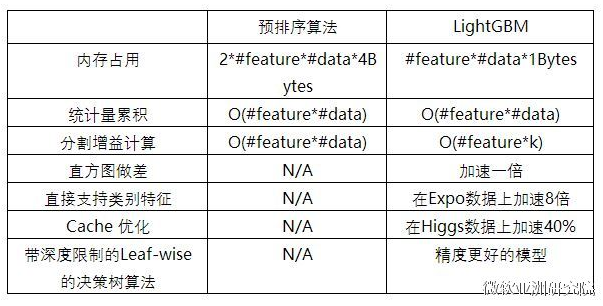
1. 直方图离散化，降低内存消耗和时间复杂度。

2. 带有深度限制的按叶子生长 (leaf-wise) 算法，Level-wise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。同Level-wise相比，在分裂次数相同的情况下，Leaf-wise可以降低更多的误差，得到更好的精度。Leaf-wise的缺点是可能会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。

3. 直方图差加速，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。

4. 直接支持类别特征

其他：cache访问优化，多线程优化，稀疏特征优化等。



支持特征并行和数据并行。

**参考：**

[1] 机器学习，周志华

[2] The Elements of Statistical Learning

[3] [GBDT（MART） 迭代决策树入门教程 | 简介](http://blog.csdn.net/w28971023/article/details/8240756)

[4] <https://www.zhihu.com/question/45487317>

[5] <http://weibo.com/ttarticle/p/show?id=2309404060623678912231>

[6] [Tianqi Chen. XGBoost: A Scalable Tree Boosting System. KDD, 2016](https://arxiv.org/pdf/1603.02754v3.pdf)