## [机器学习中的算法(1)-决策树模型组合之随机森林与GBDT](http://www.cnblogs.com/LeftNotEasy/archive/2011/03/07/random-forest-and-gbdt.html)

**版权声明：**

    本文由LeftNotEasy发布于[http://leftnoteasy.cnblogs.com](http://leftnoteasy.cnblogs.com/), 本文可以被全部的转载或者部分使用，但请注明出处，如果有问题，请联系[wheeleast@gmail.com](mailto:wheeleast@gmail.com)

**前言：**

    决策树这种算法有着很多良好的特性，比如说训练时间复杂度较低，预测的过程比较快速，模型容易展示（容易将得到的决策树做成图片展示出来）等。但是同时，单决策树又有一些不好的地方，比如说容易over-fitting，虽然有一些方法，如剪枝可以减少这种情况，但是还是不够的。

    模型组合（比如说有Boosting，Bagging等）与决策树相关的算法比较多，这些算法最终的结果是生成N(可能会有几百棵以上）棵树，这样可以大大的减少单决策树带来的毛病，有点类似于三个臭皮匠等于一个诸葛亮的做法，虽然这几百棵决策树中的每一棵都很简单（相对于C4.5这种单决策树来说），但是他们组合起来确是很强大。

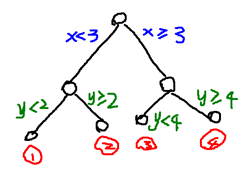
    在最近几年的paper上，如iccv这种重量级的会议，[iccv 09](http://www.cvpapers.com/iccv2009.html)年的里面有不少的文章都是与Boosting与随机森林相关的。模型组合+决策树相关的算法有两种比较基本的形式 - 随机森林与GBDT((Gradient Boost Decision Tree)，其他的比较新的模型组合+决策树的算法都是来自这两种算法的延伸。本文主要侧重于GBDT，对于随机森林只是大概提提，因为它相对比较简单。

    在看本文之前，建议先看看[机器学习与数学(3)](http://www.cnblogs.com/LeftNotEasy/archive/2011/01/02/machine-learning-boosting-and-gradient-boosting.html)与其中引用的论文，本文中的GBDT主要基于此，而随机森林相对比较独立。

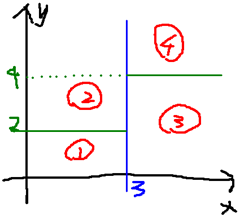
**基础内容：**

    这里只是准备简单谈谈基础的内容，主要参考一下别人的文章，对于随机森林与GBDT，有两个地方比较重要，首先是information gain，其次是决策树。这里特别推荐Andrew Moore大牛的[Decision Trees Tutorial](http://www.autonlab.org/tutorials/dtree.html)，与[Information Gain Tutorial](http://www.autonlab.org/tutorials/infogain.html)。Moore的[Data Mining Tutorial系列](http://www.autonlab.org/tutorials/)非常赞，看懂了上面说的两个内容之后的文章才能继续读下去。

    决策树实际上是将空间用超平面进行划分的一种方法，每次分割的时候，都将**当前的**空间一分为二，比如说下面的决策树：

[](http://images.cnblogs.com/cnblogs_com/LeftNotEasy/201103/201103072353099568.png)

    就是将空间划分成下面的样子：

[](http://images.cnblogs.com/cnblogs_com/LeftNotEasy/201103/201103072353092567.png)

    这样使得**每一个叶子节点都是在空间中的一个不相交的区域**，在进行决策的时候，会根据输入样本每一维feature的值，一步一步往下，最后使得样本落入N个区域中的一个（假设有N个叶子节点）

**随机森林(Random Forest):**

    随机森林是一个最近比较火的算法，它有很多的优点：

* 在数据集上表现良好
* 在当前的很多数据集上，相对其他算法有着很大的优势
* 它能够处理很高维度（feature很多）的数据，并且不用做特征选择
* 在训练完后，它能够给出哪些feature比较重要
* 在创建随机森林的时候，对generlization error使用的是无偏估计
* 训练速度快
* 在训练过程中，能够检测到feature间的互相影响
* 容易做成并行化方法
* 实现比较简单

    随机森林顾名思义，是用随机的方式建立一个森林，森林里面有很多的决策树组成，随机森林的每一棵决策树之间是没有关联的。在得到森林之后，当有一个新的输入样本进入的时候，就让森林中的每一棵决策树分别进行一下判断，看看这个样本应该属于哪一类（对于分类算法），然后看看哪一类被选择最多，就预测这个样本为那一类。

在建立每一棵决策树的过程中，有两点需要注意 - 采样与完全分裂。首先是两个随机采样的过程，random forest对输入的数据要进行行、列的采样。对于行采样，采用有放回的方式，也就是在采样得到的样本集合中，可能有重复的样本。假设输入样本为N个，那么采样的样本也为N个。这样使得在训练的时候，每一棵树的输入样本都不是全部的样本，使得相对不容易出现over-fitting。

然后进行列采样，从M个feature中，选择m个(m << M)。之后就是对采样之后的数据使用完全分裂的方式建立出决策树，这样决策树的某一个叶子节点要么是无法继续分裂的，要么里面的所有样本的都是指向的同一个分类。一般很多的决策树算法都一个重要的步骤 - 剪枝，但是这里不这样干，由于之前的两个随机采样的过程保证了随机性，所以就算不剪枝，也不会出现over-fitting。

    按这种算法得到的随机森林中的每一棵都是很弱的，但是大家组合起来就很厉害了。我觉得可以这样比喻随机森林算法：每一棵决策树就是一个精通于某一个窄领域的专家（因为我们从M个feature中选择m让每一棵决策树进行学习），这样在随机森林中就有了很多个精通不同领域的专家，对一个新的问题（新的输入数据），可以用不同的角度去看待它，最终由各个专家，投票得到结果。

    随机森林的过程请参考[Mahout的random forest](https://cwiki.apache.org/MAHOUT/random-forests.html) 。这个页面上写的比较清楚了，其中可能不明白的就是Information Gain，可以看看之前推荐过的Moore的页面。

**Gradient Boost Decision Tree:**

   GBDT是一个应用很广泛的算法，可以用来做分类、回归。在很多的数据上都有不错的效果。GBDT这个算法还有一些其他的名字，比如说MART(Multiple Additive Regression Tree)，GBRT(Gradient Boost Regression Tree)，Tree Net等，其实它们都是一个东西（参考自[wikipedia – Gradient Boosting](http://en.wikipedia.org/wiki/Gradient_boosting))，发明者是Friedman

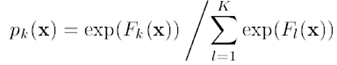
   Gradient Boost其实是一个框架，里面可以套入很多不同的算法，可以参考一下机器学习与数学(3)中的讲解。Boost是"提升"的意思，一般Boosting算法都是一个迭代的过程，每一次新的训练都是为了改进上一次的结果。

   原始的Boost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“严重关注”，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。

   而Gradient Boost与传统的Boost的区别是，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，每个新的模型的简历是为了使得之前模型的残差往梯度方向减少，与传统Boost对正确、错误的样本进行加权有着很大的区别。

   在分类问题中，有一个很重要的内容叫做Multi-Class Logistic，也就是多分类的Logistic问题，它适用于那些类别数>2的问题，并且在分类结果中，样本x不是一定只属于某一个类可以得到样本x分别属于多个类的概率（也可以说样本x的估计y符合某一个几何分布），这实际上是属于Generalized Linear Model中讨论的内容，这里就先不谈了，以后有机会再用一个专门的章节去做吧。这里就用一个结论：**如果一个分类问题符合几何分布，那么就可以用Logistic变换来进行之后的运算**。

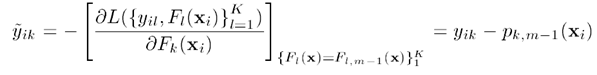
   假设对于一个样本x，它可能属于K个分类，其估计值分别为F1(x)…FK(x)，**Logistic变换如下**，logistic变换是一个平滑且将数据规范化（使得向量的长度为1）的过程，结果为属于类别k的概率pk(x)，

[](http://images.cnblogs.com/cnblogs_com/LeftNotEasy/201103/201103072353094170.png)

   对于Logistic变换后的结果，损失函数为：

[mage](http://images.cnblogs.com/cnblogs_com/LeftNotEasy/201103/20110307235310549.png)    其中，yk为输入的样本数据的估计值，当一个样本x属于类别k时，yk = 1，否则yk = 0。

    将Logistic变换的式子带入损失函数，并且对其求导，可以得到**损失函数的梯度**：

[](http://images.cnblogs.com/cnblogs_com/LeftNotEasy/201103/201103072353106928.png)    上面说的比较抽象，下面举个例子：

    假设输入数据x可能属于5个分类（分别为1,2,3,4,5），训练数据中，x属于类别3，则y = (0, 0, 1, 0, 0)，假设模型估计得到的F(x) = (0, 0.3, 0.6, 0, 0)，则经过Logistic变换后的数据p(x) = (0.16,0.21,0.29,0.16,0.16)，y - p得到梯度g：(-0.16, -0.21, 0.71, -0.16, -0.16)。观察这里可以得到一个比较有意思的结论：

    假设gk为样本当某一维（某一个分类）上的梯度:

    gk>0时，越大表示其在这一维上的概率p(x)越应该提高，比如说上面的第三维的概率为0.29，就应该提高，属于**应该往“正确的方向”前进**

                  越小表示这个估计越“准确”

    gk<0时，越小，负得越多表示在这一维上的概率应该降低，比如说第二维0.21就应该得到降低。属于**应该朝着“错误的反方向”前进**

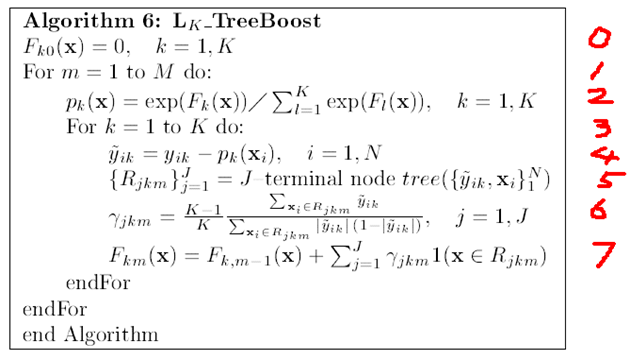
                  越大，负得越少表示这个估计越“不错误 ”

    总的来说，**对于一个样本，最理想的梯度是越接近0的梯度**。所以，我们要能够让函数的估计值能够使得梯度往反方向移动（>0的维度上，往负方向移动，<0的维度上，往正方向移动）最终使得梯度尽量=0），并且**该算法在会严重关注那些梯度比较大的样本，跟Boost的意思类似**。

    得到梯度之后，就是如何让梯度减少了。这里是用的一个**迭代+决策树**的方法，当初始化的时候，随便给出一个估计函数F(x)（可以让F(x)是一个随机的值，也可以让F(x)=0），然后之后每迭代一步就根据当前每一个样本的梯度的情况，建立一棵决策树。就让函数往梯度的反方向前进，最终使得迭代N步后，梯度越小。

    这里建立的决策树和普通的决策树不太一样，首先，这个决策树是一个叶子节点数J固定的，当生成了J个节点后，就不再生成新的节点了。

    算法的流程如下:（参考自treeBoost论文）

[](http://images.cnblogs.com/cnblogs_com/LeftNotEasy/201103/201103072353119893.png)

     0. 表示给定一个初始值

     1. 表示建立M棵决策树（迭代M次，M棵树不相互独立）

     2. 表示对函数估计值F(x)进行Logistic变换

     3. 表示对于K个分类进行下面的操作（其实这个for循环也可以理解为向量的操作，每一个样本点xi都对应了K种可能的分类yi，所以yi, F(xi), p(xi)都是一个K维的向量，这样或许容易理解一点）

     4. 表示求得残差减少的梯度方向

     5. 表示根据每一个样本点x，与其残差减少的梯度方向，得到一棵由J个叶子节点组成的决策树

     6. **为当决策树建立完成后，通过这个公式，可以得到每一个叶子节点的增益（这个增益在预测的时候用的）**

       每个增益的组成其实也是一个K维的向量，表示如果在决策树预测的过程中，如果某一个样本点掉入了这个叶子节点，则其对应的K个分类的值是多少。比如说，GBDT得到了三棵决策树，一个样本点在预测的时候，也会掉入3个叶子节点上，其增益分别为（假设为3分类的问题）：

       (0.5, 0.8, 0.1),  (0.2, 0.6, 0.3),  (0.4, 0.3, 0.3)，那么这样最终得到的分类为第二个，因为选择分类2的决策树是最多的。

     7. 的意思为，将当前得到的决策树与之前的那些决策树合并起来，作为新的一个模型(跟6中所举的例子差不多)

**GBDT的算法大概就讲到这里了，希望能够弥补一下上一篇文章中没有说清楚的部分：）**

**实现：**

     看明白了算法，就需要去实现一下，或者看看别人实现的代码，这里推荐一下wikipedia中的gradient boosting页面，下面就有一些开源软件中的一些实现，比如说下面这个：<http://elf-project.sourceforge.net/>

**参考资料：**

     除了文章中的引用的内容（已经给出了链接）外，主要还是参考Friedman大牛的文章：Greedy function approximation : A Gradient Boosting Machine