（Adaboost）GBM->GBDT->XGBoost->LGBM

Boosting类算法的发展历史大致如下：

* 1997年，Yoav Freund 和 Robert Schapire 提出了第一个成功且实用的提升算法 Adaboost 的概念。
* 1998年，Leo Breiman 将 Adaboost 归纳为建立在特殊损失函数的梯度下降方向上的算法。
* 1999-2001年，Jerome H. Friedman 将梯度下降的思想引入 Boosting 算法，提出了Gradient Boosting 的概念，以便处理不同的损失函数。
* 1999-2000年，Llew Mason，Jonathan Baxter，Peter Bartlett 和 Marcus Frean 也提出了适用于更普遍情况的一般功能性梯度提升 （General Functional Gradient Boosting），通过迭代地选择指向负梯度方向的函数来优化功能空间上的成本函数。
* 2015年，陈天奇 提出了Xgboost的概念并开源使其作为一个研究项目，并用于深度机器学习社区 (DMLC) 。

机器学习中常用的GBDT、XGBoost和LightGBM算法（或工具）都是基于梯度提升机（Gradient Boosting Machine，GBM）的算法思想，即为GBM的衍生算法。本文简要介绍了GBM的核心思想，旨在帮助大家快速理解，需要详细了解的朋友请参看Friedman的论文（Friedman J H. Greedy function approximation: a gradient boosting machine[J]. Annals of statistics, 2001: 1189-1232）。AdaBoost 本质上也可以从广义的 Gradient Boosting 推导得到（损失函数使用指数损失）

**Background**：一个弱学习器（以下简称model），其预测效果有限，一种直观的提升方法就是训练第二个model去学习特征到残差（即第一个model的输出与真实标签的差距）的映射，最终把两个model的预测结果加起来得到最终的预测结果。当然，两个model通常情况下也无法做到完美，因此上述过程可以迭代下去。

**Problem**：当损失函数是平方损失和指数损失时，每一步优化是很简单的，但对一般损失函数而言不太容易。

**Solution**：类似梯度下降法（减少损失），使用梯度来进行提升（提升模型的拟合效果，等价于减少损失），即将损失函数的负梯度在当前模型的值作为残差的近似值[2]，这样便可以通用地对采用其他损失函数的情况进行操作。

文本, 信件

描述已自动生成

boosting，该方法通常考虑的是**同质**弱学习器。它以一种高度自适应的方法顺序地学习这些弱学习器（每个基础模型都依赖于前面的模型），并按照某种确定性的策略将它们组合起来。**GBM**是一个改善弱学习器效果的**计算框架**。**GBM**梯度提升模块可以根据需要任意插入各种各样的弱分类器。若弱学习器限定了只能使用**CART中的回归树**模型，则变成了**GBDT**梯度提升决策树。

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。我们知道，单独使用决策树算法时，容易出现**过拟合**问题。假如通过方法来抑制决策树的复杂性，降低单颗决策树的拟合能力，再通过梯度提升方法集成多个决策树，最终能很好地解决过拟合问题。由此可见，梯度提升方法和决策树算法可以**取长补短**，是一对完美搭档。

AdaBoost采用的是增加上一轮学习错误样本的权重的策略，而在Gradient Boosting中则将负梯度作为上一轮基学习器犯错的衡量指标，在下一轮学习中通过拟合负梯度来纠正上一轮犯的错误。

# GBM

Gradient Boosting 还可以将其理解为**函数空间上的梯度下降**。我们比较熟悉的梯度下降通常是值在参数空间上的梯度下降（如训练神经网络，每轮迭代中计算当前损失关于参数的梯度，对参数进行更新）。

而在 Gradient Boosting 中，每轮迭代生成一个弱学习器，这个弱学习器拟合损失函数关于之前累积模型的梯度，然后将这个弱学习器加入累积模型中，逐渐降低累积模型的损失。即**参数空间的梯度下降**利用梯度信息调整参数，从而降低损失，而**函数空间的梯度下降**利用梯度，拟合一个新的函数，从而降低损失。

# Adaboost（Adaptive Boosting）

AdaBoost具有适应性，即它能适应弱分类器各自的训练误差率。这也是它的名称（适应的提升）的由来，Ada即Adaptive。

AdaBoost是利用前一轮迭代的误差率来更新训练集的权重，校正前一轮迭代被错误分类的样本，通俗一点的理解就是将重心放在分错的样本上。提升树也是boosting家族的成员，意味着提升树也采用加法模型（基学习器线性组合）和前向分步算法。

Teams

描述已自动生成

加法模型是指强分类器由一系列弱分类器线性相加而成。一般组合形式如下：

其中，为弱分类器，是弱分类器学习到的最优参数，为弱分类器在强分类器中所占比重，P是所有和的组合。这些弱分类器相加组成强分类器。

前向分步是指在训练过程中，下一轮迭代产生的分类器是在上一轮的基础上训练得来的。可以表示为如下形式：

由于采用的损失函数不同，Boosting算法也因此有了不同的类型，AdaBoost就是损失函数为指数损失的Boosting算法。AdaBoost使用的是指数损失，这个损失函数的缺点是对于异常点非常敏感，（关于各种损失函数可见之前的文章： [常见回归和分类损失函数比较](https://www.cnblogs.com/massquantity/p/8964029.html)），因而通常在噪音比较多的数据集上表现不佳。

AdaBoost改变了训练数据的权值，也就是样本的概率分布，其思想是将关注点放在被错误分类的样本上，减小上一轮被正确分类的样本权值，提高那些被错误分类的样本权值。然后，再根据所采用的一些基本机器学习算法进行学习，比如逻辑回归，SVM，决策树等等。在弱分类器的确定上，AdaBoost采用加权多数表决的方法，加大分类错误率小的弱分类器的权重。

理论上任何学习器都可以用于Adaboost.但一般来说，使用最广泛的Adaboost弱学习器是决策树和神经网络。对于决策树，Adaboost分类用了CART分类树，而Adaboost回归用了CART回归树。

**Adaboost的主要优点有：**  
1. Adaboost作为分类器时，分类精度很高  
2. 在Adaboost的框架下，可以使用各种回归分类模型来构建弱学习器，非常灵活。  
3. 作为简单的二元分类器时，构造简单，结果可理解。  
4. 不容易发生过拟合

**Adaboost的主要缺点有**：对异常样本敏感，异常样本在迭代中可能会获得较高的权重，影响最终的强学习器的预测准确性（损失函数为指数函数，在噪音比较多的数据集上表现不佳）。



图示

描述已自动生成

原理请戳：<https://zhuanlan.zhihu.com/p/39972832/>

# GBDT（Gradient Boosting Decision Tree）

GBDT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**的学习器来弥补已有模型的不足，在下一轮学习中通过拟合负梯度来纠正上一轮犯的错误。GBDT在函数空间中利用**梯度下降法**进行**优化**。

* GBDT损失函数的数值优化可以看成是在函数空间，而不是在参数空间。
* 损失函数包含平方损失,绝对值损失用于回归问题，负二项对数似然用于分类。
* 函数空间的梯度下降：

这里首先回顾一下梯度下降 (Gradient Descend)。机器学习的一大主要步骤是通过优化方法最小化损失函数𝐿(𝜃)，进而求出对应的参数𝜃。梯度下降是经典的数值优化方法，其参数更新公式：

Gradient Boosting 采用和AdaBoost同样的加法模型，在第m次迭代中，前m-1个基学习器都是固定的，即

𝑓𝑚(𝑥)=𝑓𝑚−1(𝑥)+𝜌𝑚ℎ𝑚(𝑥)

因而在第m步我们的目标是最小化损失函数 ，进而求得相应的基学习器。若将𝑓(𝑥)当成参数，则同样可以使用梯度下降法：

可以发现若将，即用基学习器拟合前一轮模型损失函数的负梯度，就是通过梯度下降法最小化𝐿(𝑓)。由于𝑓(𝑥)实际为函数，所以该方法被认为是函数空间的梯度下降。

算法流程如下：

文本, 信件

描述已自动生成

GBDT算法优点：

1. 灵活处理各种类型的数据，包括连续值和离散值；
2. 在相对少的调参时间情况下，预测的准确率也可以比较高；
3. 抵御训练数据中的噪音，具有更好的健壮性；
4. 使用一些健壮的损失函数，对异常值的鲁棒性非常强；
5. 性能最好的机器学习算法之一；
6. 适用面非常广。

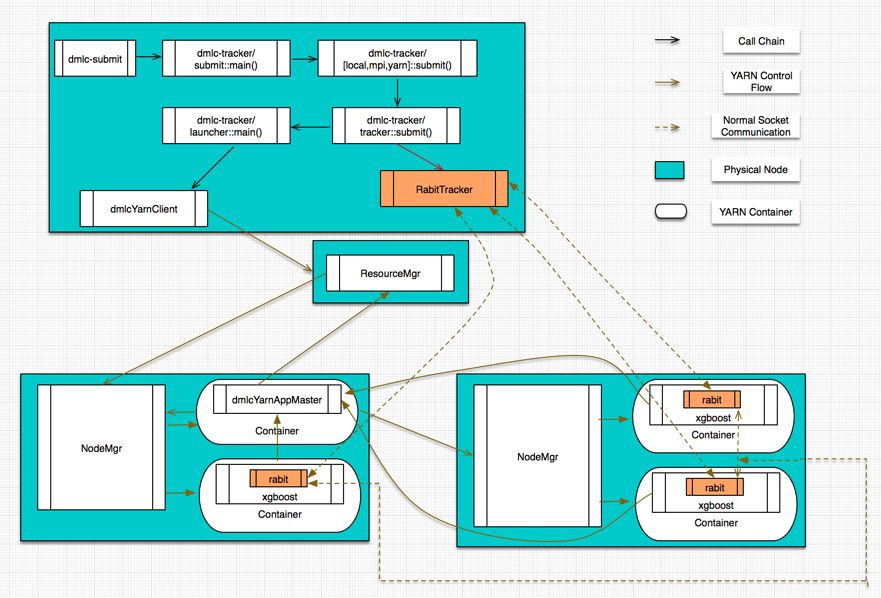
GBDT算法缺点：

1. 由于弱学习器之间存在依赖关系，难以并行训练数据；
2. 缺乏平滑性；
3. 在高维稀疏特征上表现一般。

# XGBoost （eXtreme Gradient Boosting）

图示

描述已自动生成



原理暂戳：<https://zhuanlan.zhihu.com/p/90520307>

# LGBM

GBDT (Gradient Boosting Decision Tree) 是机器学习中一个长盛不衰的模型，其主要思想是利用弱分类器（决策树）迭代训练以得到最优模型，该模型具有训练效果好、不易过拟合等优点。GBDT不仅在工业界应用广泛，通常被用于多分类、点击率预测、搜索排序等任务；在各种数据挖掘竞赛中也是致命武器，据统计Kaggle上的比赛有一半以上的冠军方案都是基于GBDT。而LightGBM（Light Gradient Boosting Machine）是一个实现GBDT算法的框架，支持高效率的并行训练，并且具有更快的训练速度、更低的内存消耗、更好的准确率、支持分布式可以快速处理海量数据等优点。

LightGBM提出的主要原因就是为了解决GBDT在海量数据遇到的问题，让GBDT可以更好更快地用于工业实践。

1. XGBoost的不足

在LightGBM提出之前，最有名的GBDT工具就是XGBoost了，它是基于预排序方法的决策树算法。这种构建决策树的算法基本思想是：首先，对所有特征都按照特征的数值进行预排序。其次，在遍历分割点的时候用的代价找到一个特征上的最好分割点。最后，在找到一个特征的最好分割点后，将数据分裂成左右子节点。

这样的预排序算法的优点是能精确地找到分割点。但是缺点也很明显：首先，空间消耗大。这样的算法需要保存数据的特征值，还保存了特征排序的结果（例如，为了后续快速的计算分割点，保存了排序后的索引），这就需要消耗训练数据两倍的内存。其次，时间上也有较大的开销，在遍历每一个分割点的时候，都需要进行分裂增益的计算，消耗的代价大。最后，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。

1. LightGBM的优化

为了避免上述XGBoost的缺陷，并且能够在不损害准确率的条件下加快GBDT模型的训练速度，lightGBM在传统的GBDT算法上进行了如下优化：

* **基于Histogram的决策树算法**
  + **内存占用更小：** 直方图算法不仅不需要额外存储预排序的结果，而且可以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。也就是说XGBoost需要用 32位的浮点数去存储特征值，并用 32位的整形去存储索引，而 LightGBM只需要用 8位去存储直方图，内存相当于减少为1/8；
  + **计算代价更小：**预排序算法XGBoost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LightGBM只需要计算k次（k可以认为是常数），直接将时间复杂度从O(#data\*#feature)降低到O(k\*#feature)，而我们知道#data>>k。
  + **Histogram直方图做差加速：**在实际构建树的过程中，LightGBM还可以先计算直方图小的叶子节点，然后利用直方图做差来获得直方图大的叶子节点，这样就可以用非常微小的代价得到它兄弟叶子的直方图。

图表, 气泡图

描述已自动生成

* **单边梯度采样 Gradient-based One-Side Sampling(GOSS)**：使用GOSS可以减少大量只具有小梯度的数据实例，这样在计算信息增益的时候只利用剩下的具有高梯度的数据就可以了，相比XGBoost遍历所有特征值节省了不少时间和空间上的开销。

文本

描述已自动生成

* **互斥特征捆绑 Exclusive Feature Bundling(EFB)**：

使用EFB可以将许多互斥的特征绑定为一个特征，这样达到了降维的目的。LightGBM的EFB算法将这个问题转化为图着色的问题来求解，将所有的特征视为图的各个顶点，将不是相互独立的特征用一条边连接起来，边的权重就是两个相连接的特征的总冲突值，这样需要绑定的特征就是在图着色问题中要涂上同一种颜色的那些点（特征）。此外，我们注意到通常有很多特征，尽管不是100％相互排斥，但也很少同时取非零值。如果我们的算法可以允许一小部分的冲突，我们可以得到更少的特征包，进一步提高计算效率。经过简单的计算，随机污染小部分特征值将影响精度最多，是每个绑定中的最大冲突比率，当其相对较小时，能够完成精度和效率之间的平衡。具体步骤如下：

* + 1. 构造一个加权无向图，顶点是特征，边有权重，其权重与两个特征间的冲突相关；
    2. 根据节点的度进行降序排序，度越大，与其它特征的冲突越大；
    3. 遍历每个特征，将它分配给现有特征包，或者新建一个特征包，使得总体冲突最小。

特征合并算法，其关键在于原始特征能从合并的特征中分离出来。绑定几个特征在同一个bundle里需要保证绑定前的原始特征的值可以在bundle中识别，考虑到histogram-based算法将连续的值保存为离散的bins，我们可以使得不同特征的值分到bundle中的不同bin（箱子）中，这可以通过在特征值中加一个偏置常量来解决。比如，我们在bundle中绑定了两个特征A和B，A特征的原始取值为区间，B特征的原始取值为区间，我们可以在B特征的取值上加一个偏置常量10，将其取值范围变为，绑定后的特征取值范围为，这样就可以放心的融合特征A和B了。

图形用户界面, 文本

描述已自动生成

* **带深度限制的Leaf-wise的叶子生长策略**：大多数GBDT工具使用低效的按层生长 (level-wise) 的决策树生长策略，因为它不加区分的对待同一层的叶子，带来了很多没必要的开销。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使用了带有深度限制的按叶子生长 (leaf-wise) 算法。

XGBoost 采用 Level-wise 的增长策略，该策略遍历一次数据可以同时分裂同一层的叶子，容易进行多线程优化，也好控制模型复杂度，不容易过拟合。但实际上Level-wise是一种低效的算法，因为它不加区分的对待同一层的叶子，实际上很多叶子的分裂增益较低，没必要进行搜索和分裂，因此带来了很多没必要的计算开销。卡通人物

描述已自动生成

图：按层生长的决策树

LightGBM采用Leaf-wise的增长策略，该策略每次从当前所有叶子中，找到分裂增益最大的一个叶子，然后分裂，如此循环。因此同Level-wise相比，Leaf-wise的优点是：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会在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。卡通画

描述已自动生成

图：按叶子生长的决策树

* **直接支持类别特征(Categorical Feature)**

对于类别型的数据，我们通常将类别特征转化为one-hot/哑变量编码。 然而，对于学习树来说这不是个好的解决方案。 原因是，对于一个基数较大的类别特征，学习树会生长的非常不平衡，切分增益可能会非常低，并且需要非常深的深度才能来达到较好的准确率。

每次都采用二分的方法，无法拟合理想的切分效果，**会把数据切分到很多零碎的小空间上，**影响下一层节点的分裂，如下图左边所示。而决策树学习时利用的是统计信息，在这些数据量小的空间上，统计信息不准确，学习会变差。

LightGBM采用many-vs-many的切分方式将类别特征分为两个子集，数据会被切分到两个比较大的空间，进一步的学习也会更好，实现类别特征的最优切分。假设某维特征有k个类别，则有种可能，时间复杂度为，LightGBM 基于 Fisher的《On Grouping For Maximum Homogeneity》论文实现了的时间复杂度。

左图为基于one-hot编码进行分裂，右图为LightGBM基于many-vs-many进行分裂

手机屏幕截图

中度可信度描述已自动生成

一阶导数是Y, 二阶导数是1

文本

描述已自动生成

具体方案：<https://blog.csdn.net/anshuai_aw1/article/details/83275299>

<https://blog.csdn.net/anshuai_aw1/article/details/83040541>

* **支持高效并行**
  + 特征并行

特征并行算法目的是在决策树生成过程中的每次迭代，高效地找到最优特征分裂点。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。

* 传统的特征并行算法：

1. 根据不同的特征子集，将数据集进行垂直切分。（不同机器worker有不同的特征子集）
2. 每个worker寻找局部的最优分裂特征以及分裂点。
3. 不同worker之间进行网络传输，交换最优分裂信息，最终得到最优的分裂信息。
4. 具有最优分裂特征的worker，局部进行分裂，并将分裂结果广播到其他worker。
5. 其他worker根据接收到的数据进行切分数据。

该方法不能有效地加速特征选择的效率，当数据量#data很大时，该并行方法不能加快效率。并且，最优的分裂结果需要在worker之间进行传输，需要消耗很多的传输资源以及传输时间。

* LightGBM的特征并行算法：

LightGBM并没有垂直的切分数据集，而是每个worker都有全量的训练数据，因此最优的特征分裂结果不需要传输到其他worker中，只需要将最优特征以及分裂点告诉其他worker，worker随后本地自己进行处理。处理过程如下：

1. 每个worker在基于局部的特征集合找到最优分裂特征。
2. workder间传输最优分裂信息，并得到全局最优分裂信息。
3. 每个worker基于全局最优分裂信息，在本地进行数据分裂，生成决策树。

然而，当数据量很大时，特征并行算法还是受限于特征分裂效率。因此，当数据量大时，推荐使用数据并行算法。

图示

描述已自动生成

* + 数据并行
* 传统的数据并行算法

1. 水平切分数据集。
2. 每个worker基于数据集构建局部特征直方图（Histogram）。
3. 归并所有局部的特征直方图，得到全局直方图。
4. 找到最优分裂信息，进行数据分裂。

缺点：网络传输代价比较大，如果使用point-to-point的传输算法，每个worker的传输代价为O(#machine \* #feature \* #bin). 如果使用All Reduce并行算子，传输代价为O(2\* #feature \* #bin).

* LightGBM的数据并行算法

1. LightGBM算法使用Reduce Scatter并行算子归并来自不同worker的不同特征子集的直方图，然后在局部归并的直方图中找到最优局部分裂信息，最终同步找到最优的分裂信息。
2. 除此之外，LightGBM使用直方图减法加快训练速度。我们只需要对其中一个子节点进行数据传输，另一个子节点可以通过histogram subtraction得到。
3. LightGBM可以将传输代价降低为O(0.5 \* #feature \* #bin)。

图表

描述已自动生成

* + 投票并行

基于投票机制的并行算法，是在每个worker中选出top k个分裂特征，然后将每个worker选出的k个特征进行汇总，并选出全局分裂特征，进行数据分裂。有理论证明，这种voting parallel以很大的概率选出实际最优的特征，因此不用担心top k的问题。

图示

描述已自动生成

图形用户界面, 文本, 应用程序, 信件, 电子邮件

描述已自动生成

* **Cache命中率优化**

由两种框架在生成树的策略导致的。

LGBM和XGBoost：

GBDT 虽然是个强力的模型，但却有着一个致命的缺陷，不能用类似 mini batch 的方式来训练，需要对数据进行无数次的遍历。如果想要速度，就需要把数据都预加载在内存中，但这样数据就会受限于内存的大小；如果想要训练更多的数据，就要使用外存版本的决策树算法。虽然外存算法也有较多优化，SSD 也在普及，但在频繁的 IO 下，速度还是比较慢的。

为了能让 GBDT 高效地用上更多的数据，我们把思路转向了分布式 GBDT， 然后就有了 LightGBM。设计的思路主要是两点，1. 单个机器在不牺牲速度的情况下，尽可能多地用上更多的数据；2.  
多机并行的时候，通信的代价尽可能地低，并且在计算上可以做到线性加速。

基于这两个需求，LightGBM 选择了基于 histogram 的决策树算法。相比于另一个主流的算法 pre-sorted（如 xgboost 中的 exact 算法），histogram 在内存消耗和计算代价上都有不少优势。

* Pre-sorted 算法需要的内存约是训练数据的两倍(2 \* #data \* #features  
  \* 4Bytes)，它需要用32位浮点来保存 feature value，并且对每一列特征，都需要一个额外的排好序的索引，这也需要32位的存储空间。对于 histogram 算法，则只需要(#data  
  \* #features \* 1Bytes)的内存消耗，仅为 pre-sorted算法的1/8。因为 histogram 算法仅需要存储 feature bin value (离散化后的数值)，不需要原始的 feature value，也不用排序，而 bin value 用 uint8\_t (256 bins) 的类型一般也就足够了。
* Pre-sorted 算法虽然理论上可以过一次数据，长满一层叶子。但由于每个特征对梯度的访问顺序都是随机的，且访问顺序不一样。这么多的随机访问会有很大的cache miss，导致实际运行效率低很多。而 histogram 算法里，不同特征访问梯度的顺序是一样的，可以提前把梯度存在连续的数组中，让不同特征访问的时候都是连续的，不会产生cache miss的问题
* 另一个计算上的优势则是大幅减少了计算[分割点](https://www.zhihu.com/search?q=%E5%88%86%E5%89%B2%E7%82%B9&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A130946285%7D" \t "_blank)增益的次数。对于一个特征，pre-sorted 需要对每一个不同特征值都计算一次分割增益，而 [histogram](https://www.zhihu.com/search?q=histogram&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A130946285%7D) 只需要计算 #bin (histogram 的横轴的数量) 次。
* 最后，在数据并行的时候，用 [histgoram](https://www.zhihu.com/search?q=histgoram&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A130946285%7D) 可以大幅降低通信代价。用 pre-sorted 算法的话，通信代价是非常大的（几乎是没办法用的）。所以 xgoobst 在并行的时候也使用 histogram 进行通信。

当然， histogram 算法也有缺点，它不能找到很精确的分割点，训练误差没有 pre-sorted 好。但从实验结果来看， histogram 算法在测试集的误差和 pre-sorted 算法差异并不是很大，甚至有时候效果更好。实际上可能决策树对于分割点的精确程度并不太敏感，而且较“粗”的分割点也自带正则化的效果。

在 histogram 算法之上， LightGBM 进行进一步的优化。首先它抛弃了大多数 GBDT 工具使用的按层生长  
(level-wise) 的决策树生长策略，而使用了带有深度限制的按叶子生长 (leaf-wise) 算法。 level-wise 过一次数据可以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。但实际上level-wise是一种低效的算法，因为它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。leaf-wise则是一种更为高效的策略，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。因此同 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。

另一个比较巧妙的优化是 histogram 做差加速。一个容易观察到的现象：一个叶子的[直方图](https://www.zhihu.com/search?q=%E7%9B%B4%E6%96%B9%E5%9B%BE&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A130946285%7D" \t "_blank)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。

# Bagging和Boosting的区别：

* **样本选择上：**

Bagging：训练集是在原始集中有放回选取的，从原始集中选出的各轮训练集之间是独立的。

Boosting：每一轮的训练集不变，只是训练集中每个样例在分类器中的权重发生变化。而权值是根据上一轮的分类结果进行调整。

* **样例权重：**

Bagging：使用均匀取样，每个样例的权重相等

Boosting：根据错误率不断调整样例的权值，错误率越大则权重越大。

* **预测函数：**

Bagging：所有预测函数的权重相等。

Boosting：每个弱分类器都有相应的权重，对于分类误差小的分类器会有更大的权重。

* **并行计算：**

Bagging：各个预测函数可以并行生成

Boosting：各个预测函数只能顺序生成，因为后一个模型参数需要前一轮模型的结果。

* **bagging是减少variance，而boosting是减少bias**

# XGBoost防止过拟合的方法

详细链接请戳：<https://blog.csdn.net/Ray_awakepure/article/details/119643040>

思路一：控制模型的复杂度：（限制树的最大深度（max\_depth）、调节正则项系数（lambda、alpha、gamma）、限制叶子节点样本数量(min\_child\_weight)）

思路二：增加随机性使得模型对噪声鲁棒：（控制随机采样比例（subsample）、调节学习率（减少lr））

思路三：通过监控loss防止过拟合发生（EarlyStopping(early\_stopping\_rounds)）

思路四：缓解样本不均衡问题（SMOTE上采样（适用于分类问题））