Lightgbm算法

# 发展过程----why Lightgbm

 CART模型往往过于简单无法有效地进行预测，因此一个更加强力的模型叫做tree ensemble。

## AdaBoost算法

AdaBoost是一种提升树的方法，和三个臭皮匠，赛过诸葛亮的道理一样(类似于专家打分)。

AdaBoost两个问题：

1. 如何改变训练数据的权重或概率分布

提高前一轮被弱分类器错误分类的样本的权重，降低前一轮被分对的权重

1. 如何将弱分类器组合成一个强分类器，亦即，每个分类器，前面的权重如何设置

采取”多数表决”的方法.加大分类错误率小的弱分类器的权重，使其作用较大，而减小分类错误率大的弱分类器的权重，使其在表决中起较小的作用。

## 2.GBDT算法以及优缺点

GBDT和AdaBosst很类似，但是又有所不同。

GBDT和其它Boosting算法一样，通过将表现一般的数个模型（通常是深度固定的决策树）组合在一起来集成一个表现较好的模型。AdaBoost是通过提升错分数据点的权重来定位模型的不足， Gradient Boosting通过负梯度来识别问题，通过计算负梯度来改进模型，即通过反复地选择一个指向负梯度方向的函数，该算法可被看做在函数空间里对目标函数进行优化。

因此可以说。

缺点：

GBDT ->预排序方法(pre-sorted)

(1).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（例如排序后的索引，为了后续快速的计算分割点），这里需要消耗训练数据两倍的内存。

(2).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。

(3).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。

## 3.Why？

常用的机器学习算法，例如神经网络等算法，都可以以mini-batch的方式训练，训练数据的大小不会受到内存限制。而GBDT在每一次迭代的时候，都需要遍历整个训练数据多次。如果把整个训练数据装进内存则会限制训练数据的大小；如果不装进内存，反复地读写训练数据又会消耗非常大的时间。尤其面对工业级海量的数据，普通的GBDT算法是不能满足其需求的。

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。

# 二. 基本介绍

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages:

LightGBM 是一个基于树学习的梯度提升框架，支持高效率的并行训练，它有以下优势：

Faster training speed and higher efficiency

- 更快的训练效率

Lower memory usage

- 低内存使用

Better accuracy

- 更好的准确率

Parallel and GPU learning supported

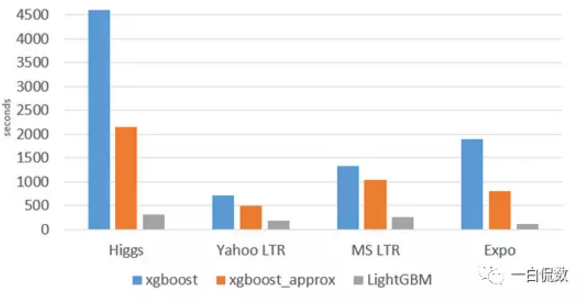
- 支持并行和GPU

Capable of handling large-scale data

- 可处理大规模数据

参考：https://github.com/Microsoft/LightGBM

LightGBM主打的高效并行训练让其性能超越现有其他boosting工具。一个实验表明，在Higgs数据集上LightGBM比XGBoost快将近10倍，内存占用率大约为XGBoost的1/6。



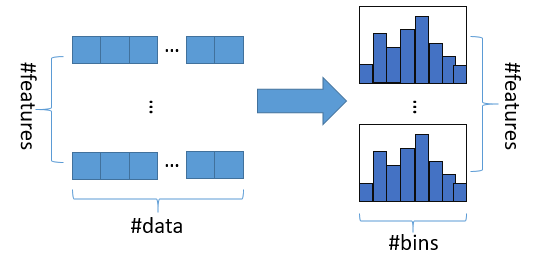
# 三. lighgbm原理

1. 基于Histogram的决策树算法
2. 带深度限制的Leaf-wise的叶子生长策略
3. 直方图做差加速
4. 直接支持类别特征(Categorical Feature)
5. Cache命中率优化
6. 基于直方图的稀疏特征优化
7. 多线程优化

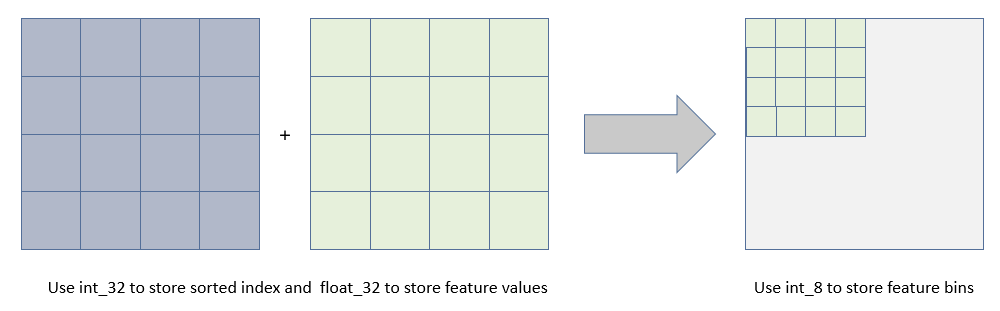
下面主要介绍Histogram（直方图）做差加速、带深度限制的Leaf-wise的叶子生长策略。

## 1.直方图算法

直方图算法的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。



使用直方图算法有很多优点。首先，最明显就是内存消耗的降低，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。

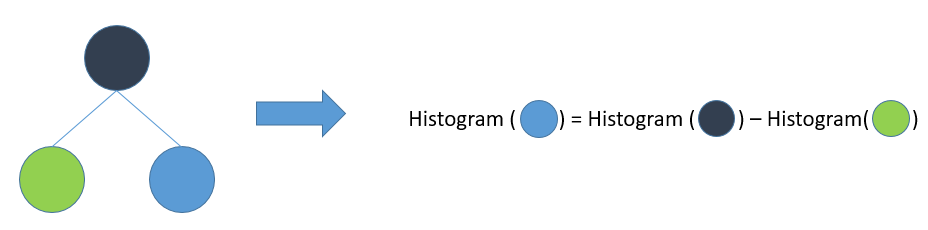


然后在计算上的代价也大幅降低，预排序算法每遍历一个特征值就需要计算一次分裂的增益，而直方图算法只需要计算k次（k可以认为是常数），时间复杂度从O(#data\*#feature)优化到O(k\*#features)。

当然，Histogram算法并不是完美的。由于特征被离散化后，找到的并不是很精确的分割点，所以会对结果产生影响。但在不同的数据集上的结果表明，离散化的分割点对最终的精度影响并不是很大，甚至有时候会更好一点。原因是决策树本来就是弱模型，分割点是不是精确并不是太重要；较粗的分割点也有正则化的效果，可以有效地防止过拟合；即使单棵树的训练误差比精确分割的算法稍大，但在梯度提升（Gradient Boosting）的框架下没有太大的影响。

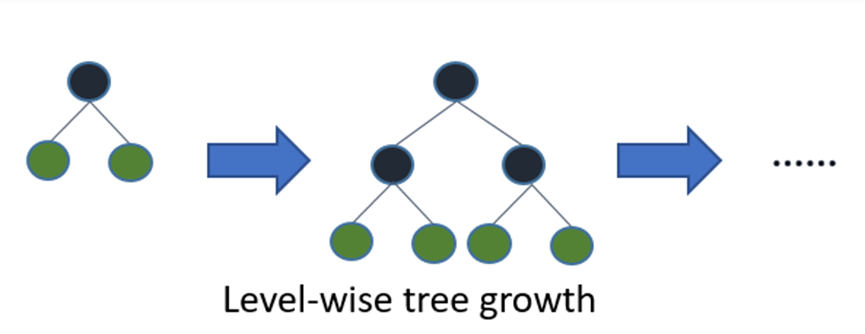
## 2.Lightgbm 的Histogram（直方图）做差加速

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。

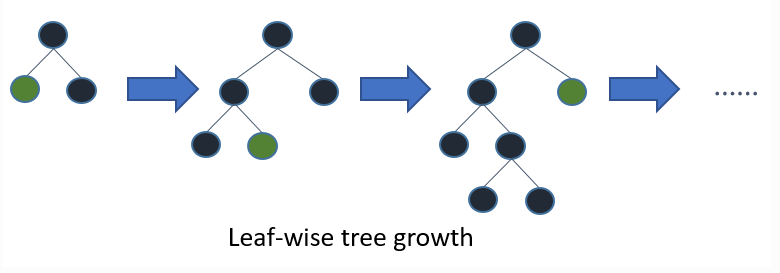


## 3.带深度限制的Leaf-wise的叶子生长策略

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，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，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。



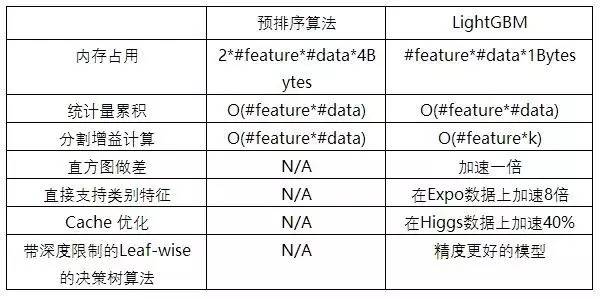
Leaf-wise则是一种更为高效的策略，每次从当前所有叶子中，找到分裂增益最大的一个叶子，然后分裂，如此循环。因此同Level-wise相比，在分裂次数相同的情况下，Leaf-wise可以降低更多的误差，得到更好的精度。Leaf-wise的缺点是可能会长出比较深的决策树，产生过拟合。因此LightGBM在Leaf-wise之上增加了一个最大深度的限制，在保证高效率的同时防止过拟合。



## 4.直接支持类别特征

实际上大多数机器学习工具都无法直接支持类别特征，一般需要把类别特征，转化到多维的0/1特征，降低了空间和时间的效率。而类别特征的使用是在实践中很常用的。基于这个考虑，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。在Expo数据集上的实验，相比0/1展开的方法，训练速度可以加速8倍，并且精度一致。据我们所知，LightGBM是第一个直接支持类别特征的GBDT工具。

LightGBM的单机版本还有很多其他细节上的优化，比如cache访问优化，多线程优化，稀疏特征优化等等，更多的细节可以查阅Github Wiki(https://github.com/Microsoft/LightGBM/wiki)上的文档说明



## 5.直接支持高效并行

LightGBM还具有支持高效并行的优点。LightGBM原生支持并行学习，目前支持特征并行和数据并行的两种。

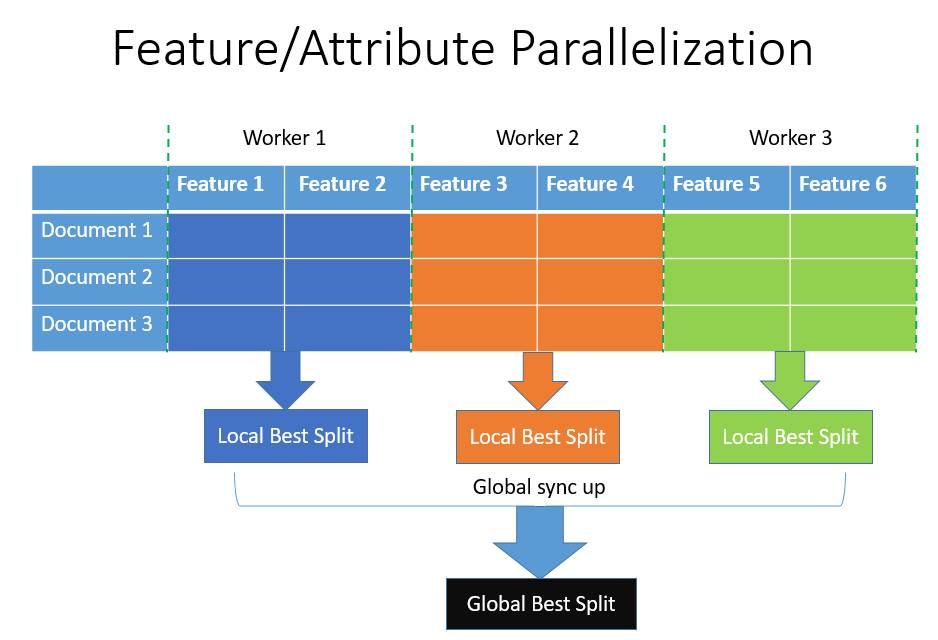
特征并行的主要思想是在不同机器在不同的特征集合上分别寻找最优的分割点，然后在机器间同步最优的分割点。

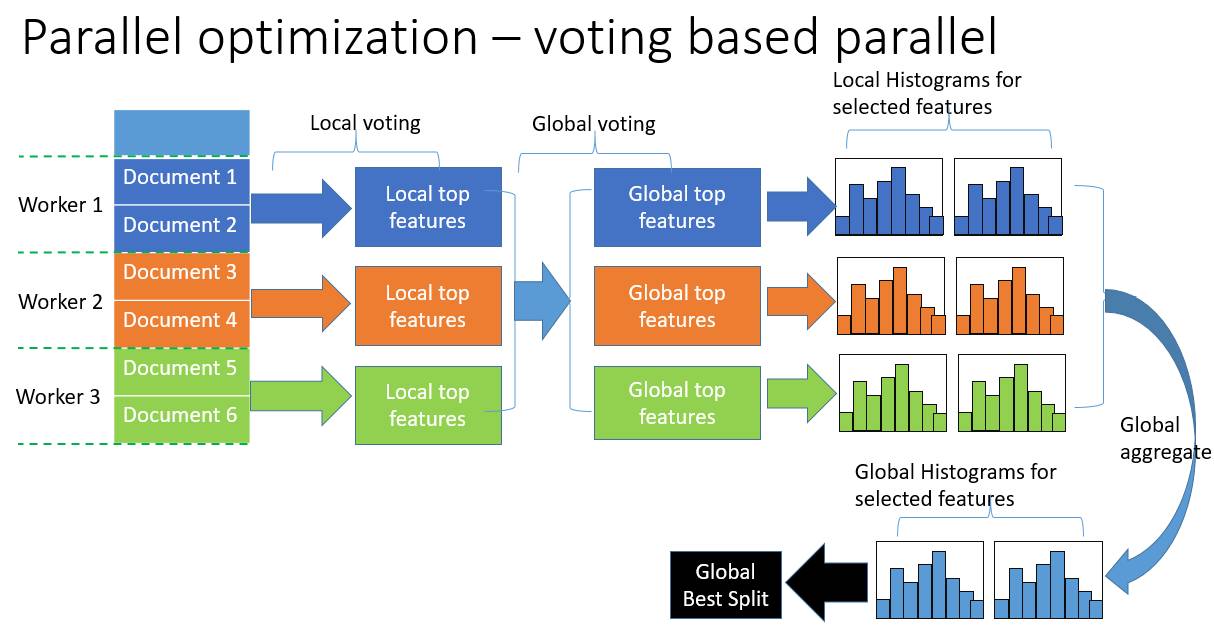
数据并行则是让不同的机器先在本地构造直方图，然后进行全局的合并，最后在合并的直方图上面寻找最优分割点。

LightGBM针对这两种并行方法都做了优化，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。在数据量很大的时候，使用投票并行可以得到非常好的加速效果。更具体的内容可以看NIPS2016的文章[A Communication-Efficient Parallel Algorithm for Decision Tree]。







# 四. LightGBM的参数

LightGBM uses the leaf-wise tree growth algorithm, while many other popular tools use depth-wise tree growth. Compared with depth-wise growth, the leaf-wise algorithm can convenge much faster. However, the leaf-wise growth may be over-fitting if not used with the appropriate parameters.

## 1参数微调

**To get good results using a leaf-wise tree, these are some important parameters:**

### (1).num\_leaves.

This is the main parameter to control the complexity of the tree model. Theoretically, we can set num\_leaves = 2^(max\_depth) to convert from depth-wise tree. However, this simple conversion is not good in practice. The reason is, when number of leaves are the same, the leaf-wise tree is much deeper than depth-wise tree. As a result, it may be over-fitting. Thus, when trying to tune the num\_leaves, we should let it be smaller than 2^(max\_depth).

For example, when the max\_depth=6 the depth-wise tree can get good accuracy, but setting num\_leaves to 127 may cause over-fitting, and setting it to 70 or 80 may get better accuracy than depth-wise. Actually, the concept depth can be forgotten in leaf-wise tree, since it doesn’t have a correct mapping from leaves to depth.

使用num\_leaves

因为LightGBM使用的是leaf-wise的算法，因此在调节树的复杂程度时，使用的是num\_leaves而不是max\_depth，大致换算关系：num\_leaves = 2^(max\_depth)

### (2).min\_data\_in\_leaf

This is a very important parameter to deal with over-fitting in leaf-wise tree. Its value depends on the number of training data and num\_leaves. Setting it to a large value can avoid growing too deep a tree, but may cause under-fitting. In practice, setting it to hundreds or thousands is enough for a large dataset.

### (3).max\_depth

You also can use max\_depth to limit the tree depth explicitly.

### (4). 对于非平衡数据集

可以param['is\_unbalance']='true’

### (5).Bagging参数

bagging\_fraction+bagging\_freq（必须同时设置）、feature\_fraction

## 2.For Faster Speed

Use bagging by setting bagging\_fraction and bagging\_freq

Use feature sub-sampling by setting feature\_fraction

Use small max\_bin

Use save\_binary to speed up data loading in future learning

Use parallel learning, refer to Parallel Learning Guide

## 3.For Better Accuracy

Use large max\_bin (may be slower)

Use small learning\_rate with large num\_iterations

Use large num\_leaves (may cause over-fitting)

Use bigger training data

Try dart

## 4.Deal with Over-fitting

Use small max\_bin

Use small num\_leaves

Use min\_data\_in\_leaf and min\_sum\_hessian\_in\_leaf

Use bagging by set bagging\_fraction and bagging\_freq

Use feature sub-sampling by set feature\_fraction

Use bigger training data

Try lambda\_l1, lambda\_l2 and min\_gain\_to\_split for regularization

Try max\_depth to avoid growing deep tree建模过程（python）

参考文献

[1]<https://github.com/Microsoft/LightGBM/blob/master/docs/Features.rst#features>

[2]<https://lightgbm.readthedocs.io/en/latest/Features.html>

[3] http://www.sohu.com/a/123480446\_133098