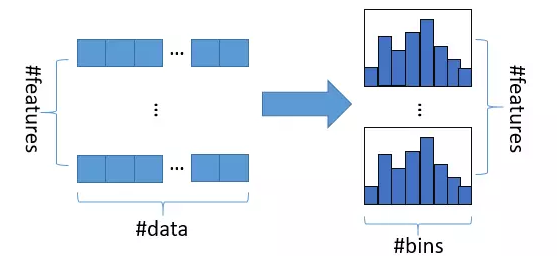
## lightgbm的优势

### 优化速度和内存使用

xgboost使用的是pre-sorted（exact）算法；lightgbm使用histogram算法，将连续的feature values转化为离散的bin。



#### 减少内存

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

#### 数据分割

在计算上的优势则主要体现在“数据分割”。决策树算法有两个主要操作组成，一个是“寻找分割点”，另一个是“数据分割”。从算法时间复杂度来看，Histogram 算法和 pre-sorted 算法在“寻找分割点”的代价是一样的，都是O(#feature\*#data)。而在“数据分割”时，pre-sorted 算法需要O(#feature\*#data)，而 histogram 算法是O(#data)。因为 pre-sorted 算法的每一列特征的顺序都不一样，分割的时候需要对每个特征单独进行一次分割。Histogram算法不需要排序，所有特征共享同一个索引表，分割的时候仅需对这个索引表操作一次就可以。（更新: 这一点不完全正确，pre-sorted 与 level-wise 结合的时候，其实可以共用一个索引表(row\_idx\_to\_tree\_node\_idx)。然后在寻找分割点的时候，同时操作同一层的节点，省去分割的步骤。但这样做的问题是会有非常多随机访问，有很大的chche miss，速度依然很慢。）。

#### 减少了计算分割点增益的次数

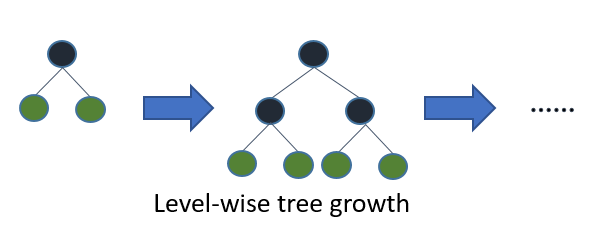
大幅减少了计算分割点增益的次数。对于一个特征，pre-sorted 需要对每一个不同特征值都计算一次分割增益，而 histogram 只需要计算 #bin (histogram 的横轴的数量) 次。

#### 差加速

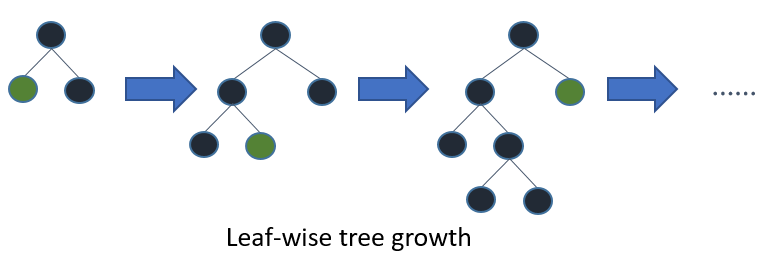
histogram 做差加速。一个叶子的直方图可以由它的父亲节点的直方图与它兄弟的直方图做差得到。利用这个方法，LightGBM 可以在构造一个叶子的直方图后，可以用非常微小的代价得到它兄弟叶子的直方图，在速度上可以提升一倍。

### accuracy优化

大部分决策树都是按照level(depth)-wise



LightGBM按照leaf-wise (best-first)



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

### 通信

在数据并行的时候，用 histgoram 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。

### 并行

feature parallel

data parallel

vote parallel

## xgboost

* 传统GBDT以CART作为基分类器，xgboost还支持线性分类器，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。
* 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。（补充：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。boosting不是一种串行的结构吗?怎么并行的？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。
* 可并行的近似直方图算法。树节点在进行分裂时，我们需要计算每个特征的每个分割点对应的增益，即用贪心法枚举所有可能的分割点。当数据无法一次载入内存或者在分布式情况下，贪心算法效率就会变得很低，所以xgboost还提出了一种可并行的近似直方图算法，用于高效地生成候选的分割点。