1. LightGBM?还得从XGBoost说起

我们在上一篇文章中提到过， xgboost是属于boosting家族，是GBDT算法的一个工程实现， 在模型的训练过程中是聚焦残差，在目标函数中使用了二阶泰勒展开并加入了正则，在决策树的生成过程中采用了精确贪心的思路，寻找最佳分裂点的时候，使用了预排序算法， 对所有特征都按照特征的数值进行预排序， 然后遍历**所有特征上的所有分裂点位**，计算按照这些候选分裂点位分裂后的**全部样本**的目标函数增益，找到最大的那个增益对应的特征和候选分裂点位，从而进行分裂。这样**一层一层**的完成建树过程， xgboost训练的时候，是通过加法的方式进行训练，也就是每一次通过聚焦残差训练一棵树出来， 最后的预测结果是所有树的加和表示。

上面简单的把xgboost的一些知识给梳理了一下，我们主要是看看xgboost在树生成的过程中，是否存在某些策略上的问题啊！机智的你可能会说：**xgboost在进行最优分裂点的选择上是先进行预排序，然后对所有特征的所有分裂点计算按照这些分裂点位分裂后的全部样本的目标函数增益**，这样会不会太费时间和空间了啊！ 哈哈， 果真是一语中的， 还真会带来这样的问题， 首先就是空间消耗很大，因为预排序的话既需要保存数据的特征值， 还得保存特征排序后的索引，毕竟这样后续计算分割点的时候快一些，但是这样就需要消耗训练数据两倍的内存。其次， 时间上也有很大的开销，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。

这时候你又可能说了xgboost不是有个近似分割的算法吗？这个不就对分裂点进行了分桶了，不就可以少遍历一些分裂点了？ 嗯嗯， **这个其实就是下面要讲的lightgbm里面的直方图的思路， 所以直方图这个思路在xgboost里面也体现过，不算是lightgbm的亮点了**， 这个是会有一些效果，可以减少点计算，但是比较微妙，lightgbm直方图算法进行了更好的优化(具体的下面说)， 比xgboost的这个还要快很多，并且XGB虽然每次只需要遍历几个可能的分裂节点，然后比较每个分裂节点的信息增益，选择最大的那个进行分割，但比较时需要考虑**所有样本**带来的信息增益，这样还是比较费劲。所以基于xgboost寻找最优分裂点的复杂度，我总结了下面三点：

$$xgboost\ 寻找最优分裂点的复杂度 = 特征数量 \times 分裂点的数量 \times 样本的数量$$

所以如果想在xgboost上面做出一些优化的话，我们是不是就可以从上面的三个角度下手，比如想个办法减少点特征数量啊， 分裂点的数量啊， 样本的数量啊等等。 元芳，你怎么看？

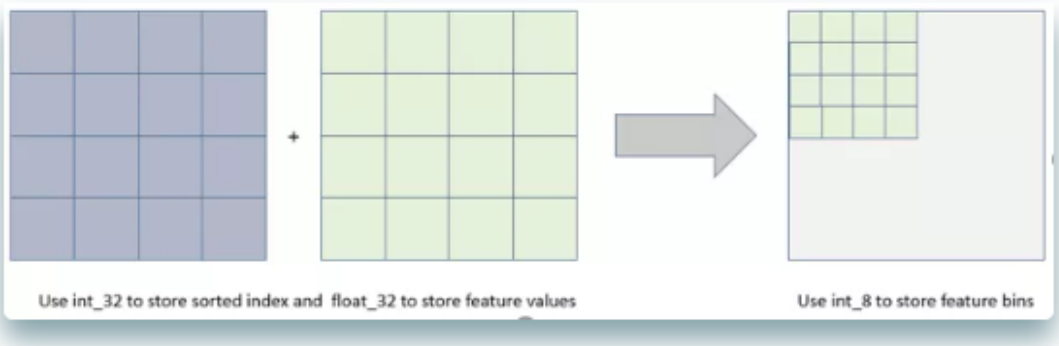
哈哈， 微软里面提出lightgbm的那些大佬还真就是这样做的， **Lightgbm里面的直方图算法就是为了减少分裂点的数量， Lightgbm里面的单边梯度抽样算法就是为了减少样本的数量， 而Lightgbm里面的互斥特征捆绑算法就是为了减少特征的数量。并且后面两个是Lightgbm的亮点所在**。

2. LightGBM的直方图算法（Histogram）

直方图算法说白了就是把连续的浮点特征离散化为k个整数（也就是分桶bins的思想）， 比如[0, 0.1) ->0, [0.1, 0.3)->1。并根据特征所在的bin对其进行**梯度累加和个数统计**，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。这么说起来，可能还是一脸懵逼， 那么就再来形象的画个图吧（有图就有真相了，哈哈，我们就拿出某一个连续特征来看看如何分桶的）：

这里注意一下，XGBoost 在进行预排序时只考虑**非零值进行加速**，而 LightGBM 也采用类似策略：只用**非零特征**构建直方图。这种离散化分桶思路其实有很多优点的， 首先最明显就是内存消耗的降低，xgboost需要用32位的浮点数去存储特征值， 并用32位的整型去存

储索引，而Lightgbm的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。

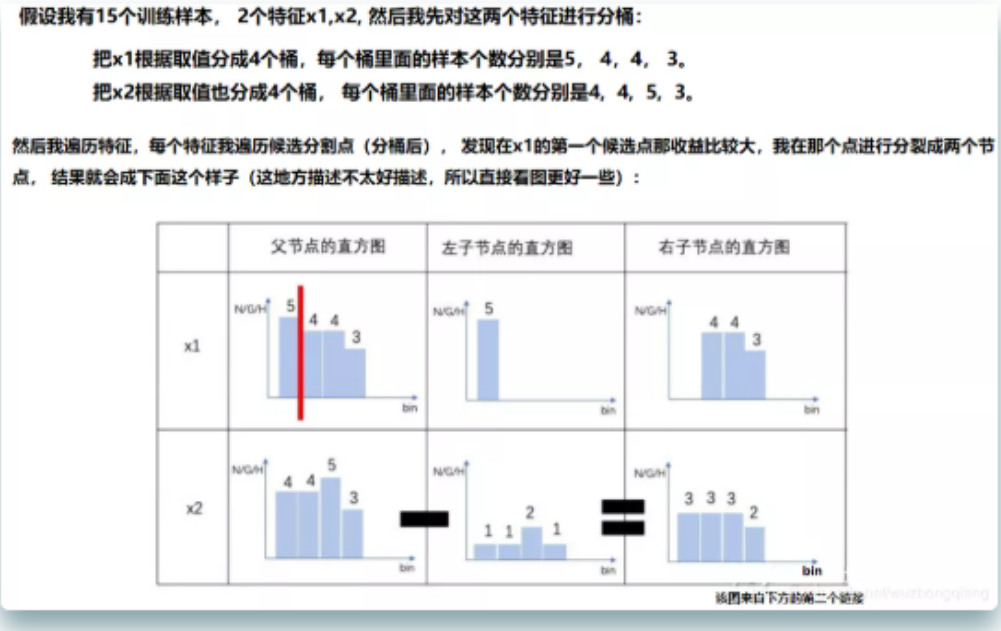


然后在计算上的代价也大幅降低，预排序算法每遍历一个特征值就需要计算一次分裂的增益，而Lightgbm直方图算法只需要计算k次（k可以认为是常数），时间复杂度从 $O(\#featureValue\_nums - 1 \times \#features\_num)$ 优化到 $O(bin\_nums - 1 \times \#features\_num)$。而我们知道 $featureValue\_nums >> bin\_nums$

直方图作差加速

当节点分裂成两个时，右边的子节点的直方图其实等于其父节点的直方图减去左边子节点的直方图：

这是为啥啊？看完之后，又一脸懵逼呢？ 其实在说这么个意思， 举个例子就明白了，



通过这种直方图加速的方式，又可以使得Lightgbm的速度快进一步啦。

关于直方图算法基本上就是这些了，当然还有很多细节简单的描述一下，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。但在实际的数据集上表明，离散化的分裂点对最终的精度影响并不大，甚至会好一些。原因在于decision tree本身就是一个弱学习器，分割点是不是精确并不是太重要，采用Histogram算法会起到正则化的效果，有效地防止模型的过拟合（bin数量决定了正则化的程度，bin越少惩罚越严重，欠拟合风险越高）。 直方图算法可以起到的作用就是可以减小分割点的数量， 加快计算。 bin越少分得越粗糙，越不精确，模型越简单

3. LightGBM的两大先进技术（GOSS&EFB）

到了这里，才是Lightgbm的亮点所在， 下面的这两大技术是Lightgbm相对于xgboost独有的， 分别是**单边梯度抽样算法(GOSS)和互斥特征捆绑算法(EFB)**, 我们上面说到，GOSS可以减少样本的数量，而EFB可以减少特征的数量，这样就能降低

模型分裂过程中的复杂度。减少样本和减少特征究竟是怎么做到的？我们下面一一来看看吧。

3.1 单边梯度抽样算法

单边梯度抽样算法(Gradient-based One-Side Sampling)是从减少样本的角度出发， 排除大部分权重小的样本**（可以理解为权重小的样本对模型的继续优化帮助不大）**，仅用剩下的样本计算信息增益，它是一种在**减少数据和保证精度上平衡**的算法。

看到这里你可能一下子跳出来进行反驳了，众所周知，GBDT中没有原始样本的权重，既然Lightgbm是GBDT的变种，应该也没有原始样本的权重，你这里怎么排除大部分权重小的样本？我读的书少，你可别蒙我。哈哈，你还别说， 你这样想还真是有点道理的， 我们知道在AdaBoost中，会给每个样本一个权重，然后每一轮之后调大错误样本的权重，让后面的模型更加关注前面错误区分的样本，这时候样本权重是数据重要性的标志（你还记得AdaBoost的这个过程吗？），到了GBDT中， 确实没有一个像Adaboost里面这样的样本权重，理论上说是不能应用权重进行采样的， But, 我们发现啊， **GBDT中每个数据都会有不同的梯度值， 这个对采样是十分有用的，即梯度小的样本，训练误差也比较小，说明数据已经被模型学习的很好了，因为GBDT不是聚焦残差吗？在训练新模型的过程中，梯度比较小的样本对于降低残差的作用效果不是太大，所以我们可以关注梯度高的样本，这样不就减少计算量了吗？**
当然这里你可能没有明白为啥梯度小的样本对降低残差效果不大， 那咱可以看看GBDT的这个残差到底是个什么东西。 我把我xgboost里面的一个图截过来：

这里我们以平方损失函数为例：

$$\sum_{i=1}^{n}\left(y_i-\left(\hat{y}_i^{t-1}+f_t\left(x_i\right)\right)\right)^2$$

则对于每一个样本：

$$g_i=\frac{\partial\left(\hat{y}^{t-1}-y_i\right)^2}{\partial\hat{y}^{t-1}}=2\left(\hat{y}^{t-1}-y_i\right)$$

$$h_i=\frac{\partial^2\left(\hat{y}^{t-1}-y_i\right)^2}{\hat{y}^{t-1}}=2$$

当然GBDT没有用到二阶导，这个不用管，我们就看上面的一阶导部分，是不是可以发现这个参数其实就是每个样本梯度的一个相反数啊？ 也就是

残差=负数*梯度

$$(y_i-y_{hat}^{t-1})=-C\times g_i$$

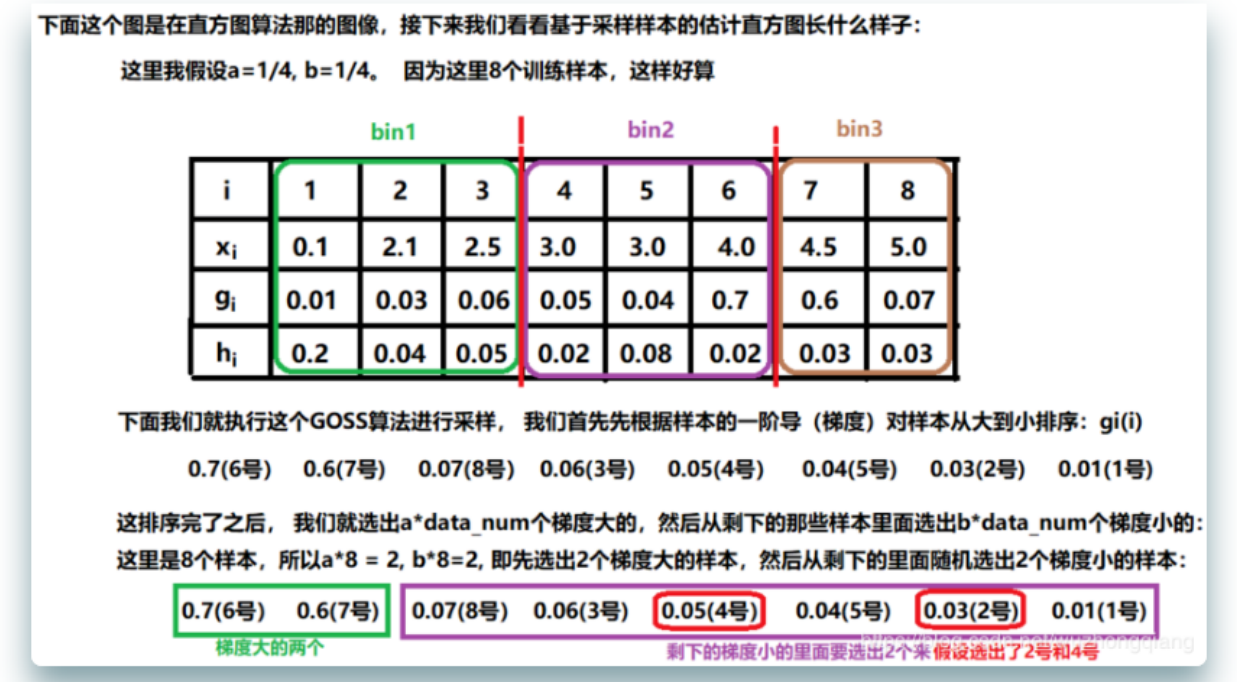
这个常数不用管， 这样也就是说如果我**新的模型想降低残差的效果好（等式左边尽可能大），那么样本的梯度应该越大越好，所以这就是为啥梯度小的样本对于降低残差的效果不大。**也是为啥样本的梯度大小可以反映样本权重的原因，这样说清楚了吧。

但是要是盲目的直接去掉这些梯度小的数据，这样就会改变数据的分布了啊，所Lightgbm才提出了单边梯度抽样算法，根据样本的权重信息对样本进行抽样，减少了大量梯度小的样本，但是还能不会过多的改变数据集的分布，这就比较牛了。怎么做的呢？

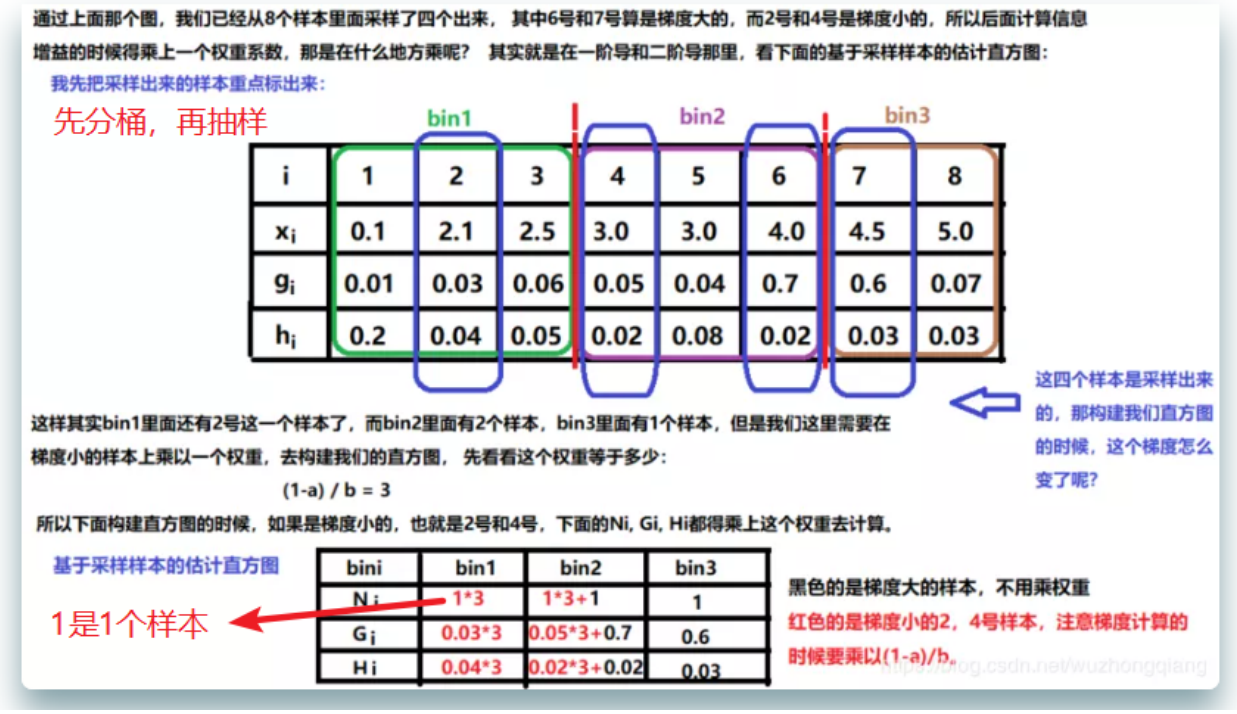
GOSS 算法保留了梯度大的样本，并对梯度小的样本进行随机抽样，为了不改变样本的数据分布，**在计算增益时为梯度小的样本引入一个常数进行平衡**。首先将要进行分裂的特征的所有取值按照绝对值大小降序排序（xgboost也进行了排序，但是LightGBM不用保存排序后的结果），然后拿到前$a\%$ 的梯度大的样本，和剩下样本的$b\%$，在计算增益时，后面的这$b\%$通过乘上$\frac{1-a}{b}$来放大梯度小的样本的权重。一方面算法将更多的注意力放在训练不足的样本上，另一方面通过乘上权重来防止采样对原始数据分布造成太大的影响。

是按照梯度进行排序

这个地方要注意一下，我看到很多资料描述的并不是那么清晰，如果去看论文的话，这个前$a\%$的样本和剩下样本的$b\%$其实是这样算的， 前$a\%$就是选出$a\times\#samples$个样本作为梯度大的，$b\%$就是在剩下的样本中选出$b\times\#samples$个样本作为梯度小但是保留下来的样本，这就是原文的：



通过上面，我们就通过采样的方式，选出了我们的样本，两个梯度大的6号和7号，然后又从剩下的样本里面随机选了2个梯度小的，4号和2号，这时候我们重点看看基于采样样本的估计直方图长什么样子，毕竟我们从8个里面选出了四个，如果直接把另外四个给删掉的话，这时候会改变数据的分布，但应该怎么做呢？也就是乘上$\frac{1-a}{b}$来放大梯度小的样本的权重到底是怎么算的？看下图：



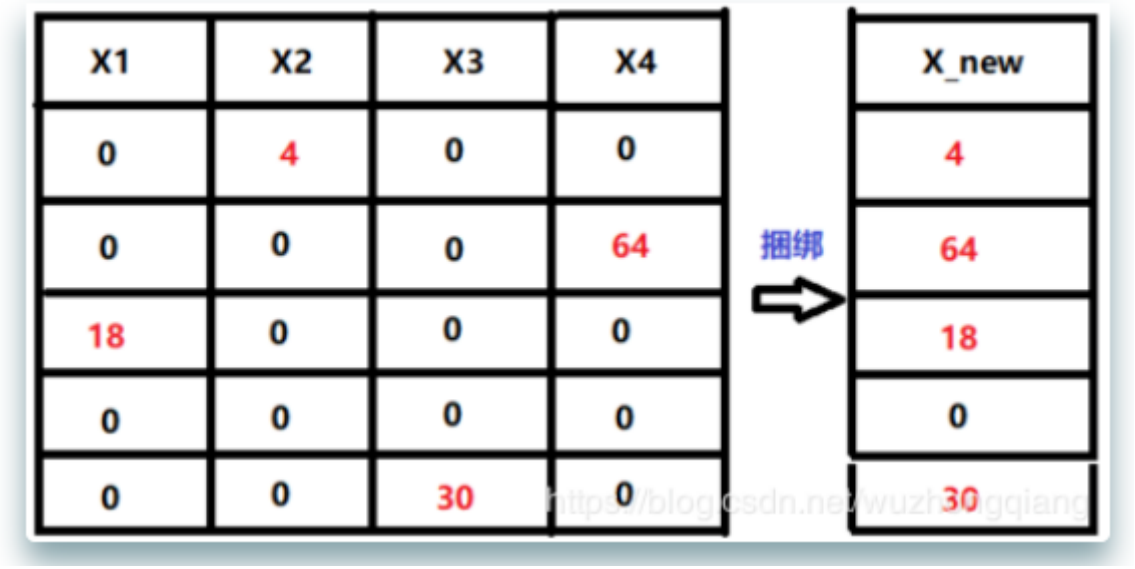
梯度小的样本乘上相应的权重之后，我们在基于采样样本的估计直方图中可以发现Ni的总个数依然是8个， 虽然6个梯度小的样本中去掉了4个，留下了两个。但是这2个

样本在**梯度上和个数**上都进行了3倍的放大，所以可以防止采样对原数数据分布造成太大的影响， 这也就是论文里面说的将更多的注意力放在训练不足的样本上的原因。

3.2 互斥捆绑特征

高维度的数据往往是稀疏的，这种稀疏性启发我们设计一种无损的方法来减少特征的维度。通常被捆绑的特征都是**互斥**的（即特征不会同时为非零值，**即一个特征为0，另一个特征不为0或两个都为0**，像one-hot就是互斥的），这样两个特征捆绑起来才不会丢失信息（**如果都不为0合并的话就会丢失信息**）。如果两个特征并不是完全互斥（部分情况下两个特征都是非零值），可以用一个指标对特征不互斥程度进行衡量，**称之为冲突比率，当这个值较小时，我们可以选择把不完全互斥的两个特征捆绑，而不影响最后的精度。**

到这又一脸懵逼，这又是说的什么鬼？什么稀疏，互斥，冲突的？如果上面的听不懂，我可以举个比较极端的例子来看一下特征捆绑到底是在干嘛：



看到上面的这些特征够稀疏了吧（大部分都是0），而每一个特征都只有一个训练样本是非0且都不是同一个训练样本，这样的话特征之间也没有冲突了。这样的情况就可以把这四个特征捆绑成一个，这样是不是维度就减少了啊。（有没有感觉这种矩阵很眼熟，从右往左的话有没有种one-hot的味道)

所以互斥特征捆绑算法（Exclusive Feature Bundling）是从减少特征的角度去帮助Lightgbm更快， 它指出如果将一些特征进行融合绑定，则可以降低特征数量。这样在构建直方图的时候时间复杂度从$O(\#data \times \#feature)$变成$O(\#data \times \#bundle)$，这里的$\#bundle$指的特征融合后特征包的个数，且$\#bundle << \#feature$。这样又可以使得速度加快了，哈哈。但是针对这个特征捆绑融合，有两个问题需要解决， 毕竟像我上面举得那种极端的例子除了OneHot之后的编码，其实很少见。

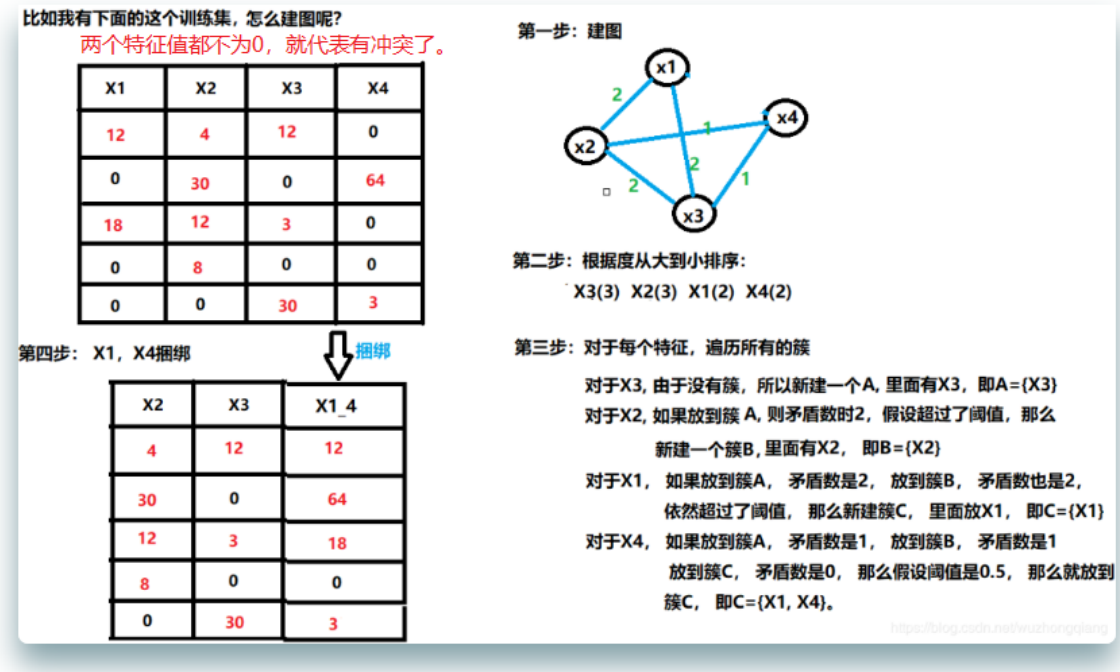
- 怎么判定哪些特征应该绑在一起?
- 特征绑在一起之后，特征值应该如何确定呢?

对于问题一：EFB 算法利用特征和特征间的关系构造一个加权无向图，并将其转换为图着色的问题来求解，求解过程中采用的贪心策略。感觉这里如果说成图着色问题的话反而有点难理解了，毕竟这里是加权无向图，而图着色问题可以去百度一下到底是怎么回事，反正觉得还不如直接说过程好理解，所以直接看过程反而简单一些。

其实说白了，捆绑特征就是在干这样的一件事：

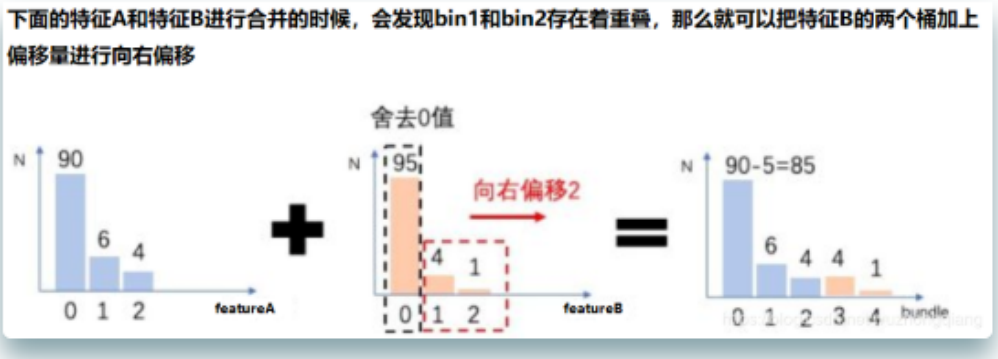
- 首先将所有的特征看成图的各个顶点，将不相互独立的特征用一条边连起来，边的权重就是两个相连接的特征的总冲突值~~（也就是这两个特征上不同时为0的样本个数）~~。
  也就是两个特征上同时不为0的样本个数
- 然后按照节点的度对特征降序排序， 度越大，说明与其他特征的冲突越大
- 对于每一个特征， 遍历已有的特征簇，如果发现该特征加入到特征簇中的矛盾数不超过某一个阈值，则将该特征加入到该簇中。如果该特征不能加入任何一个已有的特征簇，则新建一个簇，将该特征加入到新建的簇中。

什么？没明白？ 比如上面画的那个图的例子，会发现这些特征都是相互独立的点，度为0，这样排序之后也发现与其他特征的冲突为0，这样的直接放到一个簇里面就没问题，所以这四个特征直接可以合并。当然一般没有这么巧的事， 所以我把上面的随便改几个数看看这个过程是什么样子的：



上面这个过程的时间复杂度其实是$O(\#features^2)$的，因为要遍历特征，每个特征还要遍历所有的簇， 在特征不多的情况下还行，但是如果特征维度很大，就不好使了。所以为了改善效率，可以不建立图，而是将特征按照非零值个数进行排序，因为更多的非零值的特征会导致更多的冲突，所以跳过了上面的第一步，直接排序然后第三步分簇。

这样哪些特征捆绑的问题就解决了，下面就是第二个， **捆绑完了之后特征应该如何取值呢?** 这里面的一个关键就是**原始特征能从合并的特征中分离出来**， 这是什么意思? 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里面进行识别，考虑到直方图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簇中的不同bins里面去，这可以通过在特征值中加入一个偏置常量来解决。比如，我们把特征A和B绑定到了同一个bundle里面， A特征的原始取值区间[0,10), B特征原始取值区间[0,20), 这样如果直接绑定，那么会发现我从bundle里面取出一个值5， 就分不出这个5到底是来自特征A还是特征B了。所以我们可以再B特征的取值上加一个常量10转换为[10, 30)，这样绑定好的特征取值就是[0,30)，我如果再从bundle里面取出5， 就一下子知道这个是来自特征A的。这样就可以放心的融合特征A和特征B了。看下图:



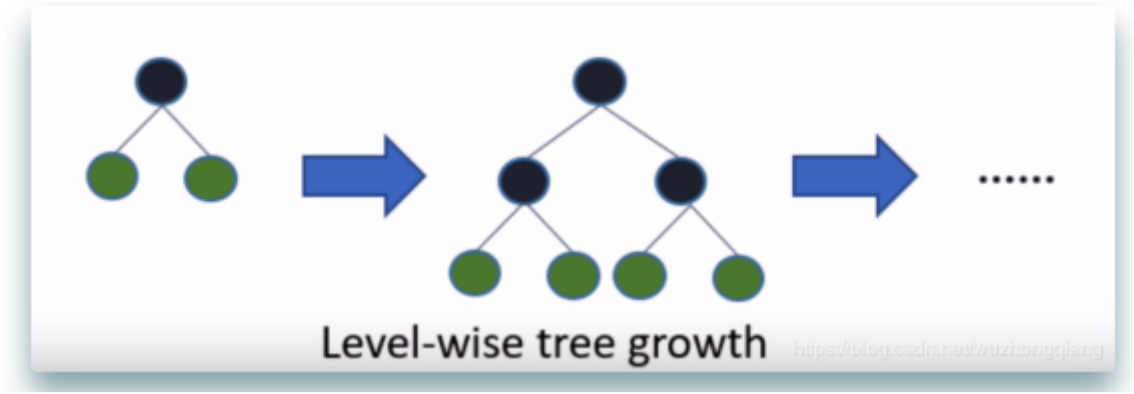
特征捆绑算法到这里也就基本上差不多了， 通过EFB，许多排他的特征就被捆绑成了更少的密集特征，**这个大大减少的特征的数量，对训练速度又带来很大的提高**。利用这种思路，可以通过对某些特征的取值重新编码，将多个这样互斥的特征捆绑成为一个新的特征。有趣的是，对于类别特征，如果转换成onehot编码，则这些onehot编码后的多个特征相互之间是互斥的，从而可以被捆绑成为一个特征。因此，对于指定为类别型的特征，LightGBM可以直接将每个类别取值和一个bin关联，从而自动地处理它们，而无需预处理成onehot编码多此一举。

4. LightGBM的生长策略（Leaf-wise）

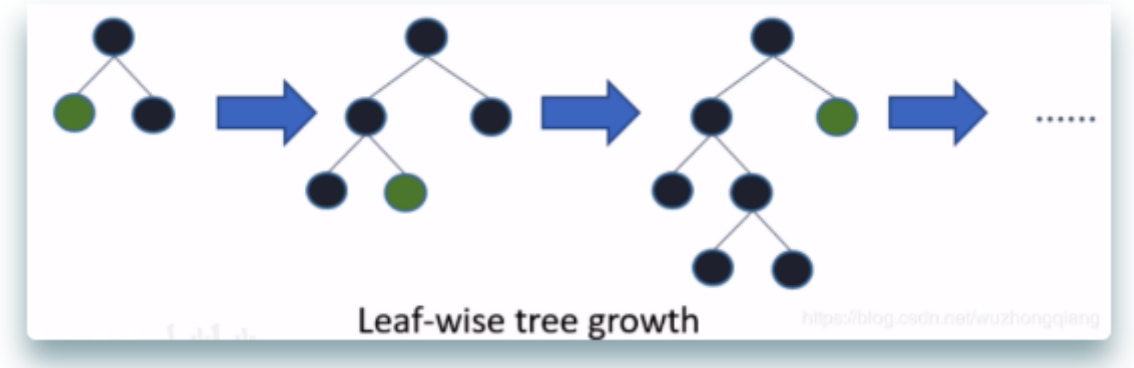
上面我们已经整理完了LightGBM是如何在寻找最优分裂点的过程中降低时间复杂度的， 可以简单的回忆一下，我们说xgboost在寻找最优分裂点的时间复杂度其实可以归到三个角度：特征的数量，分裂点的数量和样本的数量。而LightGBM也提出了三种策略分别从这三个角度进行优化，直方图算法就是为了减少分裂点的数量， GOSS算法为了减少样本的数量，而EFB算法是为了减少特征的数量。

那么lightgbm除了在寻找最优分裂点过程中进行了优化，其实在树的生成过程中也进行了优化， 它抛弃了xgboost里面的按层生长(level-wise)， 而是使用了带有深度限制的按叶子生长(leaf-wise)。这个有什么好处吗?

好不好处先不谈，首先看看这两种生长方式是怎么回事， XGBoost 在树的生成过程中采用 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(一层一层的走，不管它效果到底好不好)



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。（最大信息增益的优先， 我才不管层不层呢）



所以看到这里应该知道Leaf-wise的优势了吧， Level-wise的做法会产生一些低信息增益的节点，浪费运算资源，但是这个对于防止过拟合挺有用。而Leaf-wise能够追求更好的精度，降低误差，但是会带来过拟合问题。那你可能问，那为啥还要用Leaf-wise呢？过拟合这个问题挺严重鸭！ 但是人家能提高精度啊，哈哈，哪有那么十全十美的东西， 并且作者也使用了max_depth来控制树的高度。其实敢用Leaf-wise还有一个原因就是Lightgbm在做数据合并，直方图和GOSS等各个操作的时候，其实都有天然正则化的作用，所以作者感觉在这里使用Leaf-wise追求高精度是一个不错的选择。

5. LightGBM的工程优化

工程优化这部分主要涉及到了三个点：

1. 类别特征的支持（这个不算是工程）
2. 高效并行

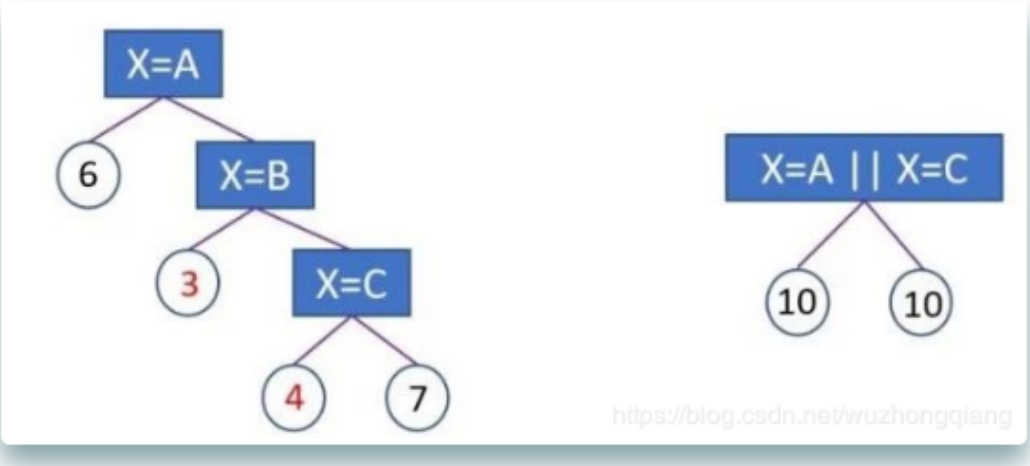
3. Cache命中率优化

5.1 支持类别特征

首先从第一个点开始，**LightGBM是第一个直接支持类别特征的GBDT工具**。我们知道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，通过one-hot 编码，转化到多维的0/1特征，降低了空间和时间的效率。但对于决策树来说，其实并不推荐使用独热编码，尤其是特征中类别很多，会存在以下问题：

- 会产生样本切分不平衡问题，切分增益会非常小。如，国籍切分后，会产生是否中国，是否美国等一系列特征，这一系列特征上只有少量样本为 1，大量样本为 0。这种划分的增益非常小：较小的那个拆分样本集，它占总样本的比例太小。无论增益多大，乘以该比例之后几乎可以忽略；较大的那个拆分样本集，它几乎就是原始的样本集，增益几乎为零；
- 影响决策树学习：决策树依赖的是数据的统计信息，而独热码编码会把数据切分到零散的小空间上。在这些零散的小空间上统计信息不准确的，学习效果变差。本质是因为独热码编码之后的特征的表达能力较差的，特征的预测能力被人为的拆分成多份，每一份与其他特征竞争最优划分点都失败，最终该特征得到的重要性会比实际值低。

LightGBM 原生支持类别特征，采用 many-vs-many 的切分方式将类别特征分为两个子集，实现类别特征的最优切分。假设有某维特征有 k 个类别，则有 $2^{(k-1)}-1$ 中可能，时间复杂度为 $O(2^k)$，LightGBM 基于 Fisher 大佬的《On Grouping For Maximum Homogeneity》实现了 $O(klog_2k)$ 的时间复杂度。

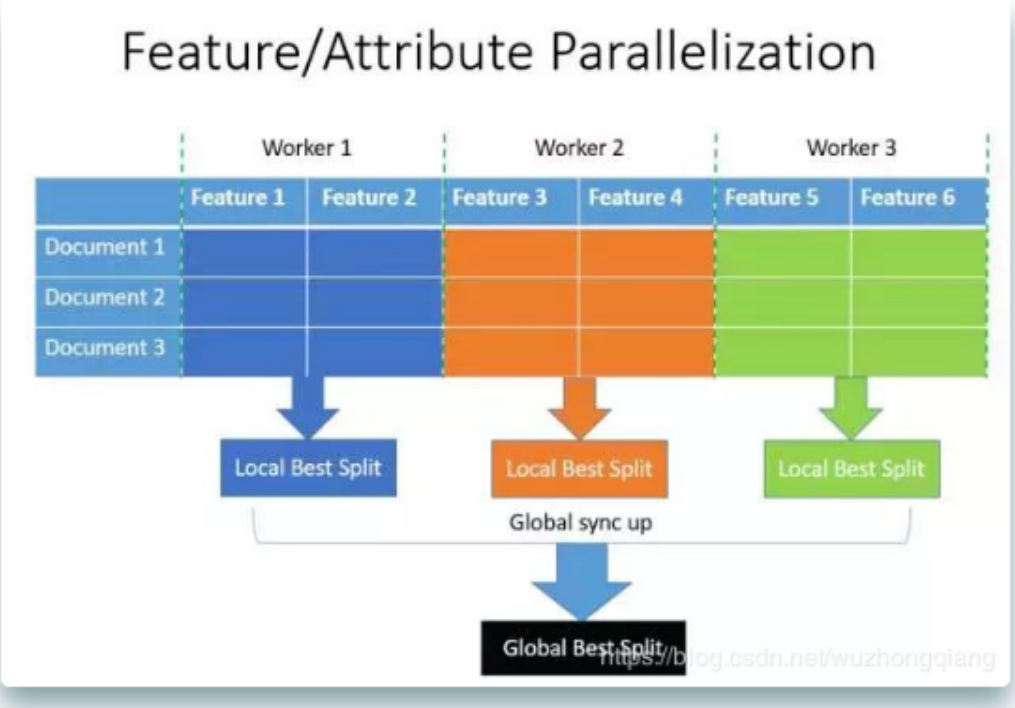


上图为左边为基于 one-hot 编码进行分裂，后图为 LightGBM 基于 many-vs-many 进行分裂，右边叶子节点的含义是 $X=A$ 或者 $X=C$ 放到左孩子，其余放到右孩子，右边的切分方法，数据会被切分到两个比较大的空间，进一步的学习也会更好。

5.2 支持高效并行

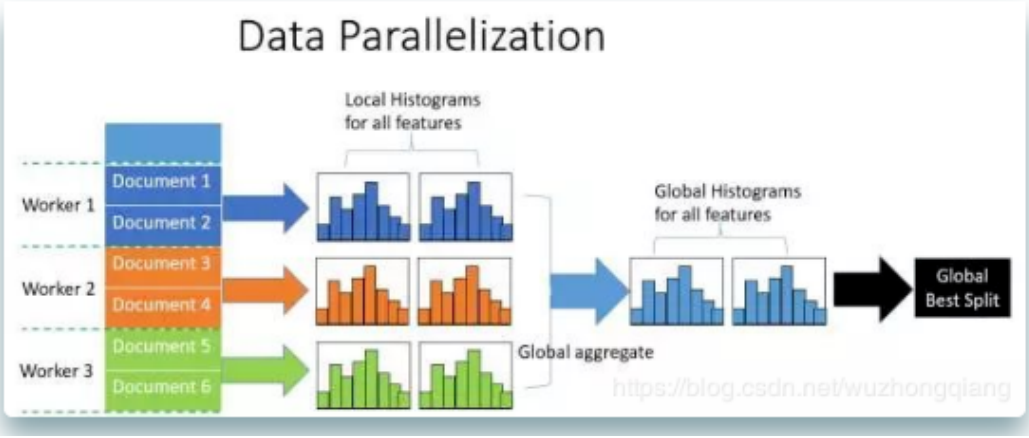
1. 特征并行 特征并行的主要思想是**不同机器在不同的特征集合上分别寻找最优的分割点，然后在机器间同步最优的分割点**。XGBoost使用的就是这种特征并行方法。这种特征并行方法有个很大的缺点：就是对数据进行垂直划分，每台机器所含数据不同，然后使用不同机器找到不同特征的最优分裂点，划分结果需要通过通信告知每台机器，增加了额外的复杂度。

   LightGBM 则不进行数据垂直划分，而是**在每台机器上保存全部训练数据，在得到最佳划分方案后可在本地执行划分而减少了不必要的通信**。具体过程如下图所示。



2. 数据并行 传统的数据并行策略主要为水平划分数据，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。这种数据划分有一个很大的缺点：通讯开销过大。如果使用点对点通信，一台机器的通讯开销大约为 $O(\#machine \times \#feature \times \#bin)$；如果使用集成的通信，则通讯开销为 $O(2 \times \#feature \times \#bin)$。

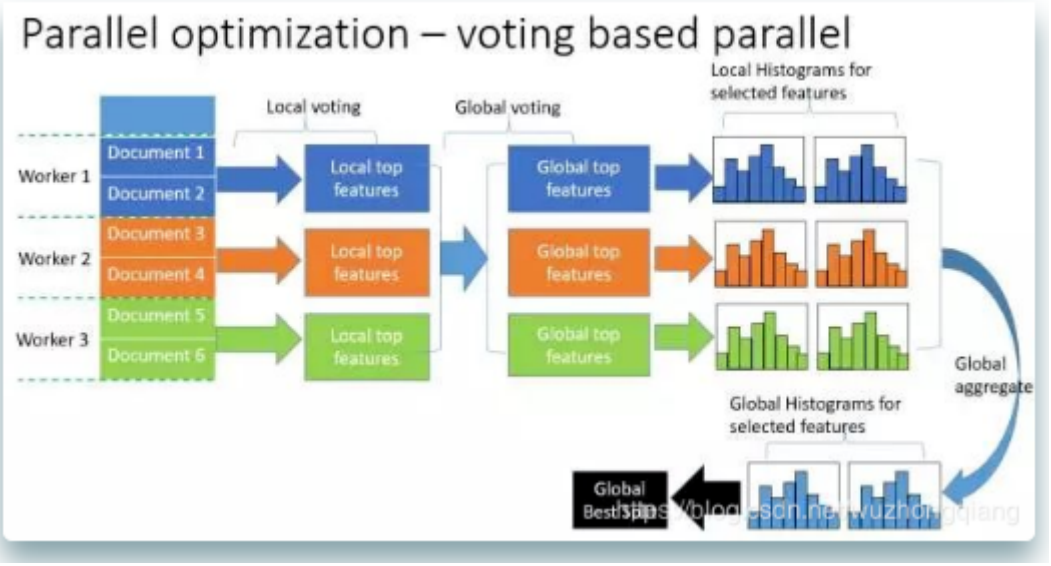
LightGBM在数据并行中使用分散规约 (Reduce scatter) 把直方图合并的任务分摊到不同的机器，降低通信和计算，并利用直方图做差，进一步减少了一半的通信量。具体过程如下图所示。



投票并行

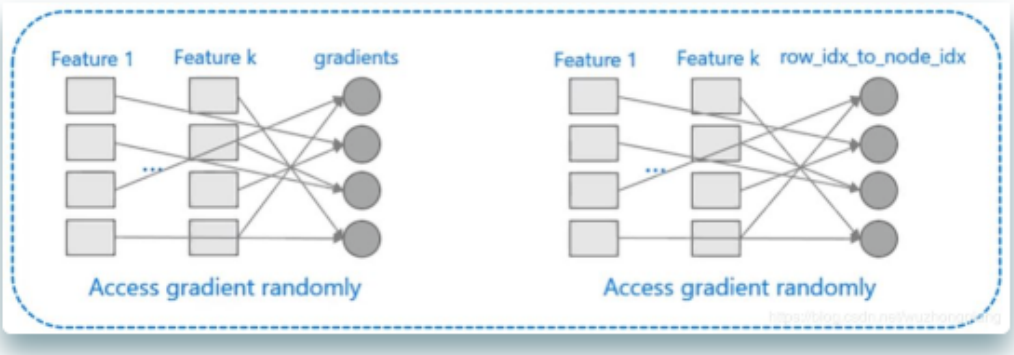
基于投票的数据并行则进一步优化数据并行中的通信代价，使通信代价变成常数级别。在数据量很大的时候，使用投票并行的方式只合并部分特征的直方图从而达到降低通信量的目的，可以得到非常好的加速效果。具体过程如下图所示。大致步骤为两步：

- 本地找出 Top K 特征，并基于投票筛选出可能是最优分割点的特征；
- 合并时只合并每个机器选出来的特征



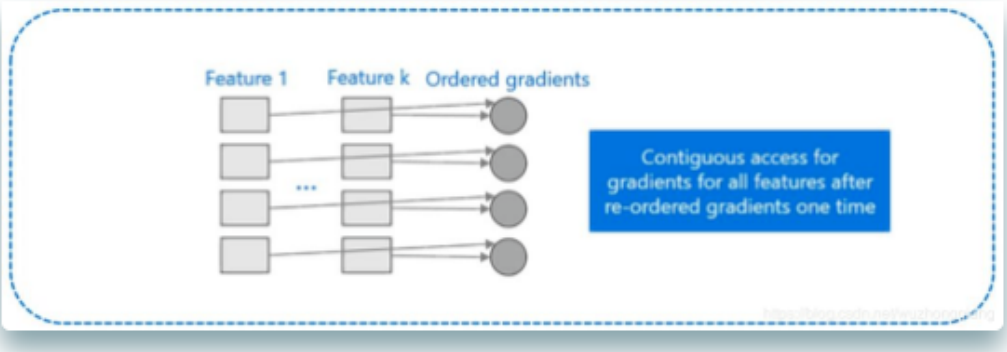
5.3 Cache命中率优化

XGBoost对cache优化不友好，如下图所示。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。为了解决缓存命中率低的问题，XGBoost 提出了缓存访问算法进行改进。



而 LightGBM 所使用直方图算法对 Cache 天生友好：

- 首先，所有的特征都采用相同的方式获得梯度（区别于XGBoost的不同特征通过不同的索引获得梯度），只需要对梯度进行排序并可实现连续访问，大大提高了缓存命中率；
- 其次，因为不需要存储行索引到叶子索引的数组，降低了存储消耗，而且也不存在 Cache Miss 的问题。



6. LightGBM实战

Lightgbm的参数非常多，有核心参数，学习控制参数，IO参数，目标函数参数，度量参数等很多，但是我们调参的时候不需要关注这么多，只需要记住常用的关键的一些参数即可，下面从四个问题的维度整理一些调参的指导：

- 针对leaf-wise树的参数优化
  - num_leaves:控制了叶节点的数目。它是控制树模型复杂度的主要参数。如果是level-wise，则该参数为2^{depth}，其中depth为树的深度。但是当叶子数量相同时，leaf-wise的树要远远深过level-wise树，非常容易导致过拟合。因此应该让num_leaves小于2^{depth}。在leaf-wise树中，并不存在depth的概念。因为不存在一个从leaves到depth的合理映射。
  - min_data_in_leaf: 每个叶节点的最少样本数量。它是处理leaf-wise树的过拟合的重要参数。将它设为较大的值，可以避免生成一个过深的树。但是也可能导致欠拟合。
  - max_depth：控制了树的最大深度。该参数可以显式的限制树的深度。
- 针对更快的训练速度
  - 通过设置 bagging_fraction 和 bagging_freq 参数来使用 bagging 方法
  - 通过设置 feature_fraction 参数来使用特征的子抽样
  - 使用较小的 max_bin
  - 使用 save_binary 在未来的学习过程对数据加载进行加速
- 获得更好的准确率
  - 使用较大的 max_bin （学习速度可能变慢）
  - 使用较小的 learning_rate 和较大的 num_iterations
  - 使用较大的 num_leaves （可能导致过拟合）
  - 使用更大的训练数据
  - 尝试DART
- 缓解过拟合
  - 使用较小的 max_bin， 分桶粗一些
  - 使用较小的 num_leaves 不要在单棵树分的太细
  - 使用 lambda_l1, lambda_l2 和 min_gain_to_split 来使用正则
  - 尝试 max_depth 来避免生成过深的树
  - 使用 min_data_in_leaf 和 min_sum_hessian_in_leaf， 确保叶子节点有足够多的数据

"

下面就与xgboost对比一下，总结一下lightgbm的优点作为收尾， 从内存和速度两方面总结：

1. 内存更小

- XGBoost 使用预排序后需要记录特征值及其对应样本的统计值的索引，而 LightGBM 使用了直方图算法将特征值转变为 bin 值，且不需要记录特征到样本的索引，将空间复杂度从 O(2*#data) 降低为 O(#bin)，极大的减少了内存消耗；
- LightGBM 采用了直方图算法将存储特征值转变为存储 bin 值，降低了内存消耗；
- LightGBM 在训练过程中采用互斥特征捆绑算法减少了特征数量，降低了内存消耗。

2. 速度更快

- LightGBM 采用了直方图算法将**遍历样本转变为遍历直方图**，极大的降低了时间复杂度；
- LightGBM 在训练过程中**采用单边梯度算法过滤掉梯度小的样本**，减少了大量的计算；
- LightGBM 采用了基于 Leaf-wise 算法的增长策略构建树，减少了很多不必要的计算量；
- LightGBM 采用优化后的特征并行、数据并行方法加速计算，当数据量非常大的时候还可以采用投票并行的策略；
- LightGBM 对缓存也进行了优化，增加了 Cache hit 的命中率。