秋雨淅淅l
本身是一个搬砖工!

1. 前言

> 本系列其它三篇：
> 决策树（一）| 基础决策树 ID3、C4.5、CART 核心概要
> 决策树（二）| 集成学习 | AdaBoost、GBDT、Random Forest原理解析
> 决策树（三）| XGBoost深度解析

本篇文章是我们决策树系列的第四篇：LightGBM原理解析。

LightGBM是2016年微软发布的一个集成学习模型，与我们上篇学习的XGBoost相比，LightGBM主打的优势是：**在准确率相近的前提下，可以数倍的减少耗时和内存。**

凭借着这两大优点，LightGBM在很短的时间内风靡着算法竞赛和工业应用领域。

接下来，我们通用从算法和工程两个角度，来详细地解读LightGBM。

2. LightGBM的核心创新点

LightGBM作者发现，限于集成学习的算法架构，GBDT类的模型每次训练都需要**加载所有的样本数据**，同时**需要频繁的为样本排序**，而这个缺陷在大数面前 就容易出现**训练时长高**和**内存占用多**的问题。在当时，面对这一问题，尽管是当时市面上最为流行的XGBoost、pGBDT也无法很好的解决。

据此，LightGBM从算法和工程两个角度同时发力，对GBDT做出了优化。

在算法上，其最主要的创新之处在于提出了：

- **单边梯度采样法**（Gradient-based One-Side Sampling, GOSS）；
  - **样本采样；**只采样梯度大的样本和少部分小梯度样本。
- **互斥特征捆绑**（Exclusive Feature Bundling, EFB）；
  - **特征降维；**合并不同时为空的特征，在尽可能小的特征损失下，降低要用的特征数量。
- **采用了直方图算法**（Histogram algorithm）和**直方图作差**的计算方法；
  - 将占内存空间大的连续特征，以分桶的形式，转为离散特征；
  - 直方图的叶子节点，有一个可以通过作差的方式得出。

上述三点，尤其是前两点创新 是LightGBM变快、变轻的主要原因。

此外，除上述优化外，LightGBM还有其它两个层面的算法创新：

- **决策树生长策略由Level-wise变为Leaf-wise；**
  - 减少许多无效的节点切分。
- **支持直接处理类别；**
  - 调包侠福音。

而在工程上，LightGBM的优化主要由如下几点：

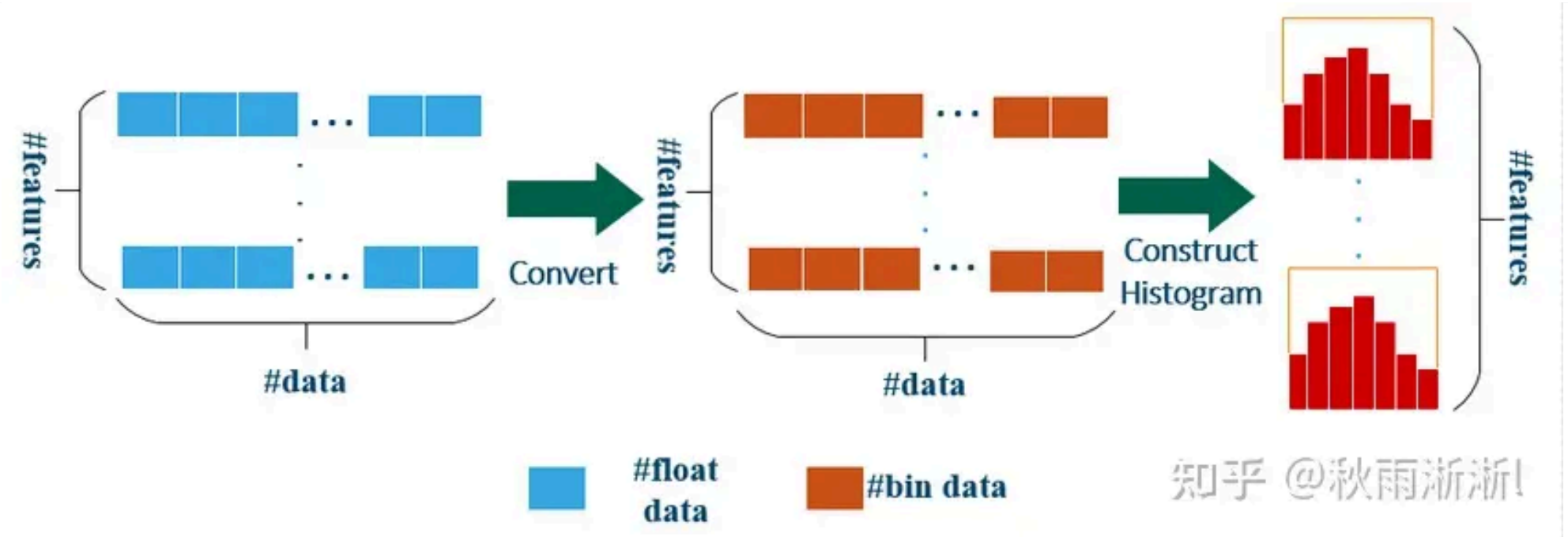
- **特征并行**
- **数据并行**
- **投票并行**

上述工程优化助力的大数据计算，极大的减少了并行计算的通讯成本。

3. LightGBM的算法优化

3.1 直方图和直方图的优点

直方图算法（Histogram algorithm）和直方图加速是后文算法理解的一个先决条件，这里我们先剖析一下LightGBM使用的直方图算法是什么。



如上图所示，直方图算法其实就是将 **连续的浮点型特征，以分桶的形式，转为离散特征。**

不过，可能这么一句简单的话，我们可能不能完全get到LightGBM采用直方图算法带来的好处，特别是我们已经知道XGBoost也有着类似的分桶操作。

但实际上，采用直方图算法至少带来了如下优点：

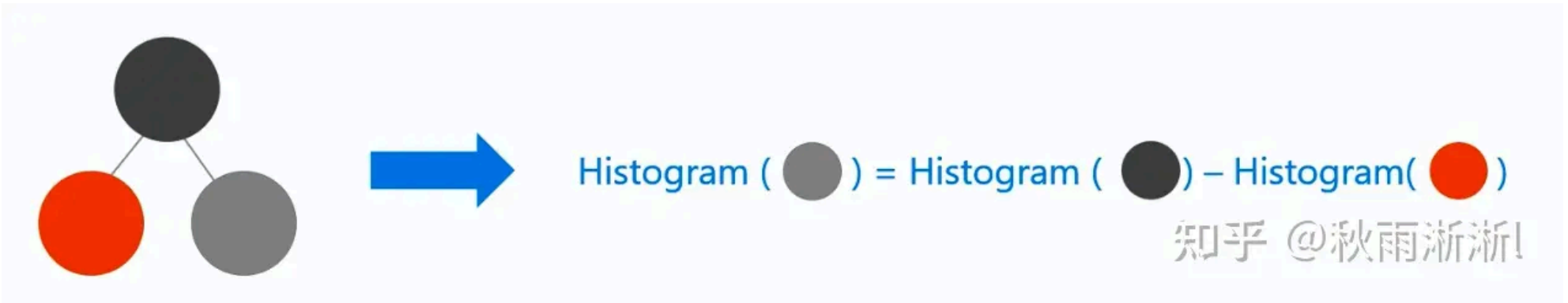
- **内存占用降低了7倍。**
  - 对比类似于XGBoost的Pre-sorted 算法，由于其是基于32位浮点型来存储原始特征，而且每一个特征还需要相同内存大小的索引空间，因此需要(2 * #data * #features * 4Bytes)的空间。但直方图算法只需要(#data * #features * 1Bytes)的内存消耗，**仅为 pre-sorted算法的1/8。**
  - 考虑直方图算法仅需要存储 feature bin value (离散化后的数值)，不需要原始的 feature value，也不用排序，而 bin value 用 uint8(256 bins)的类型一般也就足够了
- **避免了XGBoost中的cache miss问题。**
  - 在XGBoost中，由于特征的梯度值在存储上并不连续，因此节点分裂取数据时，经常会发生cache miss的问题，尽管XGBoost后续也提出了优化方法。但在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。
- **大幅度减少了决策树的节点分裂次数。**
  - 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。
- **数据并处理时，可以大幅度减少通信开销。**
  - 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）。

不过直方图算法并非没有缺点，由于histgoram会把原本连续的特征离散化，这就在一定程度上损失了特征的信息，使得节点分裂时找到的分割点不是很精确，因此也会对结果产生一定影响（训练误差没有 pre-sorted 好）。但好在LightGBM本身就是**弱学习器**，这部分的信息损失在梯度提升（Gradient Boosting）的框架下并其没有带来太大的影响，而且从某种层度上还带来了正则化的效果，降低了模型整体的过拟合风险。

3.3 直方图加速原理

上面提及的直方图算法，其实并不是LightGBM首次提出的，但基于直方图做加速计算，就是LightGBM实打实的Trick了。

不过说起直方图做加速计算，其实也很容易理解的。



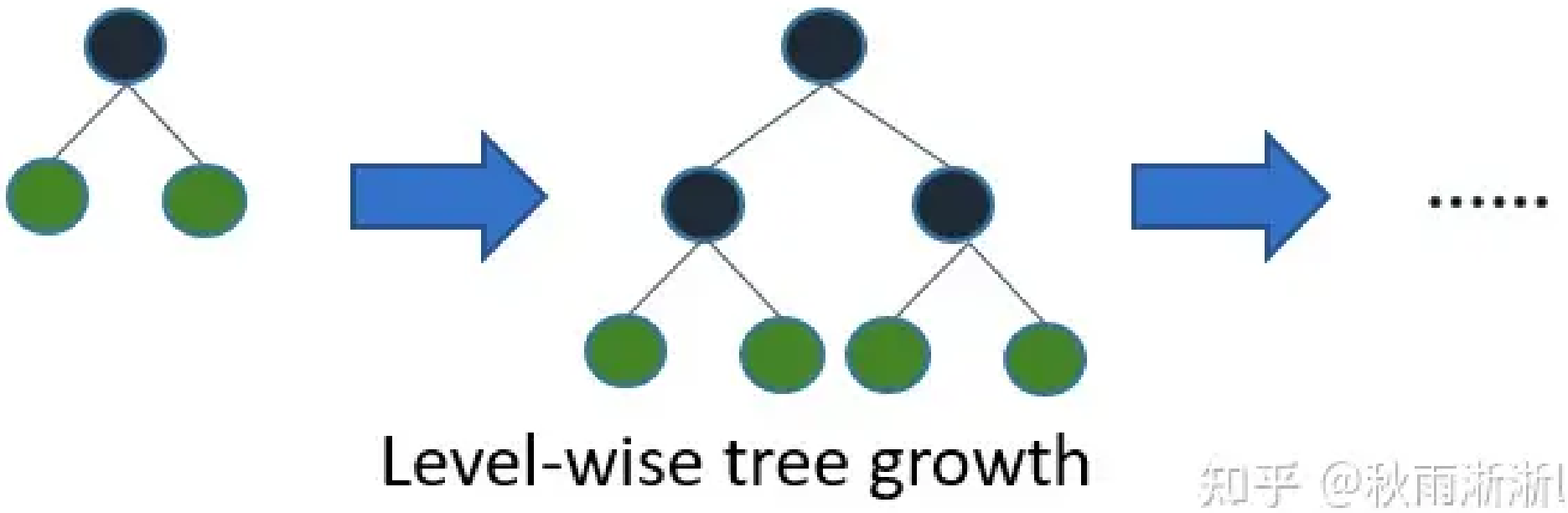
如上图所示，一个叶子节点的直方图可以由它的父节点与其兄弟节点作差得到，这一优化LightGBM称之为直方图作差加速。从效果上来看，单这一点优化，就可以使得LightGBM比常规的直方图算法在速度上快一倍。

3.4 Leaf-wise生长策略

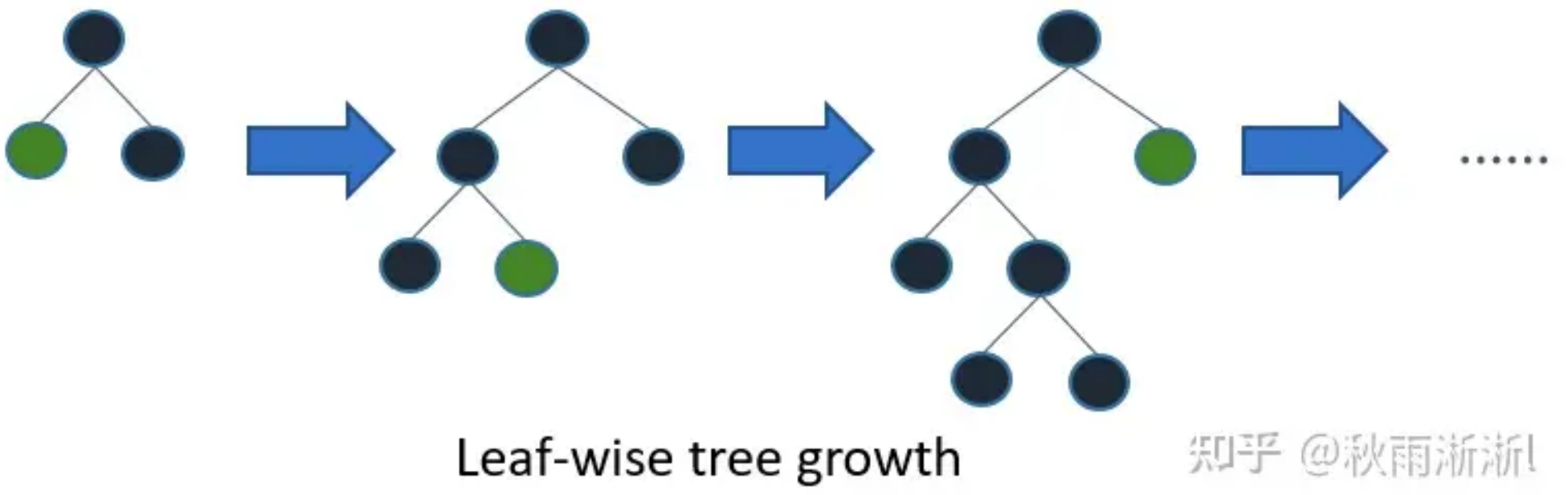
还有一点需要先提及的是，LightGBM独特的Leaf-wise决策树 生长策略。

传统的决策树，包括XGBoost在内，采用的都是Level-wise模式的生长形式。如下图，Level-wise的生长形式就是层层水平的划分形式。

虽然咋一看，level-wise 扫描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。而这么说的主要原因就是很多叶子的分裂增益较低，没必要进行搜索和分裂。



而Leaf-wise则不同，它是一种更为高效的策略，如下图所示，Leaf-wise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，然后纵向分裂。



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。

不过leaf-wise 也就一个明显的缺点，就是可能会长出比较深的决策树，从而带来过拟合。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。

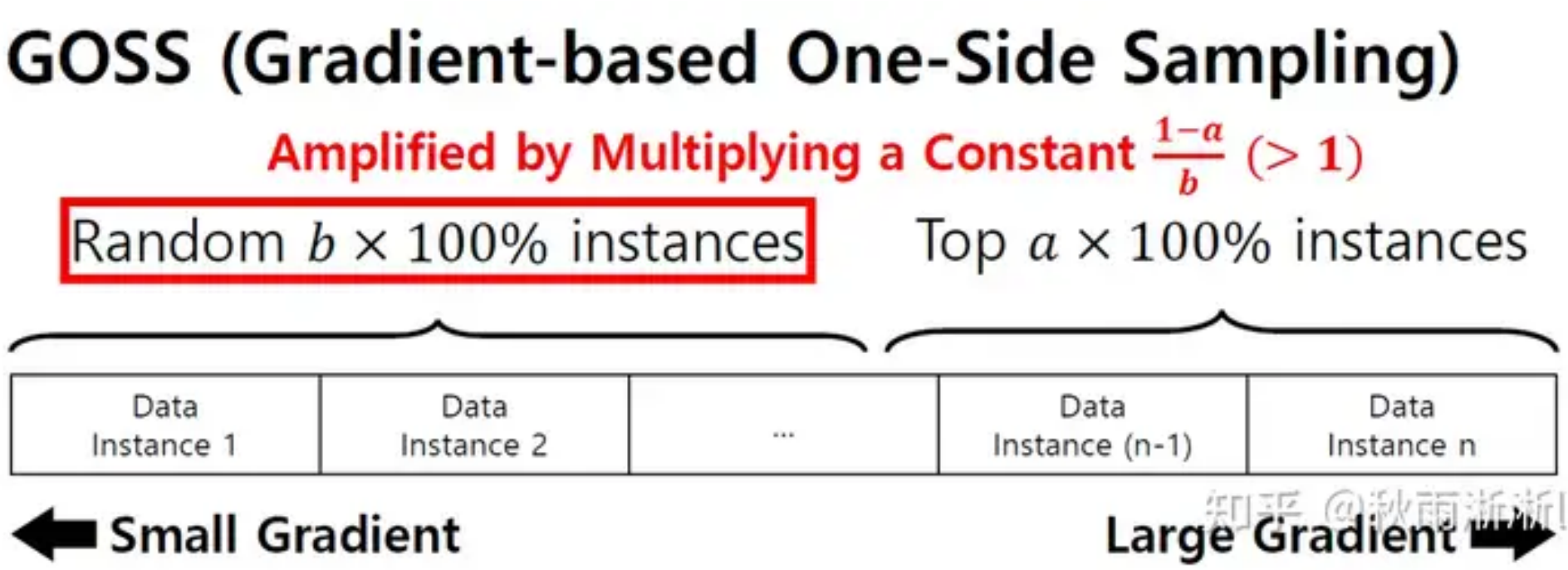
3.4 单边梯度采样

在介绍完LightGBM采用的直方图算法、直方图作差加速以及Leaf-wise生长策略后，我们就可以着重的来介绍LightGBM最为重要的一个加速优化算法--单边梯度采样（Gradient-based One-Side Sampling, GOSS）算法。

前文提到过，单边梯度采样是对样本进行欠采样，而GOSS在做欠采样时，则主要围绕着如下问题开展：

- 如何采取到更有”信息量“的样本；
- 如何保障采样前后的样本分布一致；

针对第一个问题，LightGBM的答案是**优先采取梯度值在Top a% 的样本。**而这么做的原因是因为在Gradient Boosting的框架下，梯度越大的样本在分裂时的增益越大，小梯度的样本说已经被训练的比较好了。



而针对第二个问题，LightGBM首先是**在剩下的小梯度样本中随机采样b%比率的样本。**其次在权重计算时，LightGBM对这b%的样本，统一加权了 $\frac{1-a}{b}$ 倍（a和b都是小数值），从而进一步的避免样本的分布不一致问题。

单边梯度采样的算法描述如下：

Algorithm 2: Gradient-based One-Side Sampling

Input: I: training data, d: iterations
Input: a: sampling ratio of large gradient data
Input: b: sampling ratio of small gradient data
Input: $loss$: loss function, L: weak learner
models $\leftarrow \{\}$, fact $\leftarrow \frac{1-a}{b}$
topN $\leftarrow$ a $\times$ len(I) , randN $\leftarrow$ b $\times$ len(I)
for $i = 1$ ***to*** d **do**
preds $\leftarrow$ models.predict(I)
g $\leftarrow$ $loss$(I, preds), w $\leftarrow$ {1,1,...}
sorted $\leftarrow$ GetSortedIndices(abs(g))
topSet $\leftarrow$ sorted[1:topN]
randSet $\leftarrow$ RandomPick(sorted[topN:len(I)], randN)
usedSet $\leftarrow$ topSet + randSet
w[randSet] $\times =$ fact $\triangleright$ Assign weight $fact$ to the small gradient data.
newModel $\leftarrow$ L(I[usedSet], $-$ g[usedSet], w[usedSet])
models.append(newModel)

小结：可以看到的是，LightGBM仅仅只需要原样本（a+b）%数据量，就达到了全量数据类似准确度的训练结果，而且也没有过多改变样本分布。

3.5 互斥特征捆绑

互斥特征捆绑（Exclusive Feature Bundling, EFB）是LightGBM另一大算法优化，它的出发点是**降低特征的维度。**（降维的思路很容易理解的，因为无论是竞赛还是工业界面临的数据都是高维稀疏的。）

而说起EFB的降维思路，其实可以用一句概述，即：**将特征不同时为空的两个特征捆绑在一起；**

这怎么理解？

首先，这种不同时为空的特征被LightGBM称为**互斥特征（**Exclusive Feature， 不过老实说这种比喻并不直观**），**而它们可以被”**捆**“在一起的的逻辑也很简单，因为两个不同时为空的特征合在一起才不会丢失信息。

据此，在理想情况下，LightGBM很容易就基于EFB完成了特征降维的工作。但遗憾的是，在实际应用中， 同时不为空的特征比率还是比较少的。为此，LightGBM提出了一种基于图着色的特征捆绑策略；

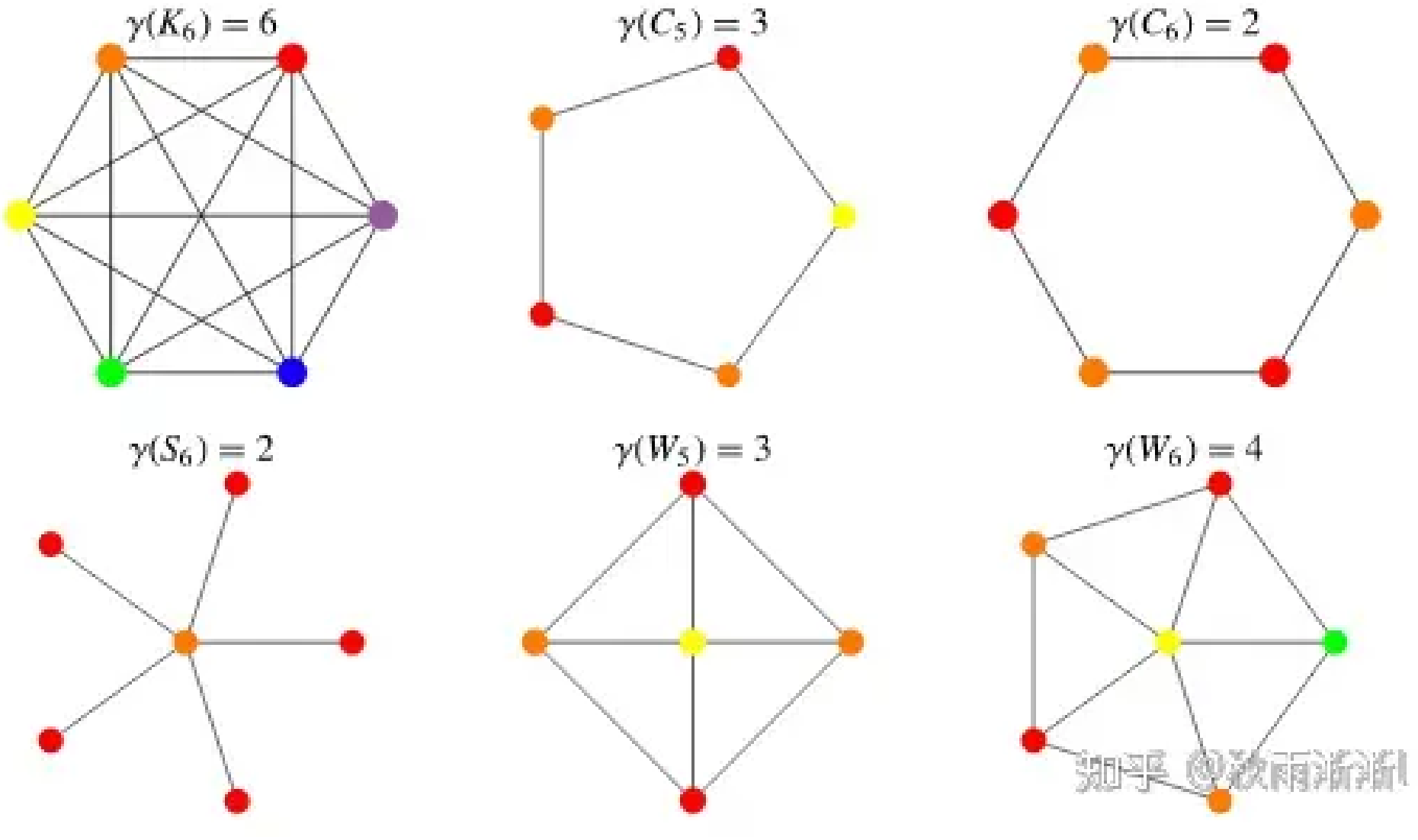
3.5.1 特征捆绑策略

首先要抛出的一个事实是：如何将众多相互独立的特征进行捆绑是一个NP-Hard问题；

> NP-Hard（non-deterministic polynomial）问题，无法在多项式时间里得到精确的求解结果；

在这一事实下，LightGBM的EFB算法选择将上述问题转化为一个**图着色问题（**Graph coloring - Wikipedia**）**，并采取贪心的近似方法来求解。

具体的来说，EFB算法将特征视为图中的各个顶点，各个顶点（**特征**）之间边权重，就是两个特征之间的**冲突值，**



这样，样绑定的特征就是在图着色问题中要涂上同一种颜色的特征点。

此外，EFB算法注意到，其实有很多特征尽管不是100%互斥的，但是也很少同时取到非零值。据此，EFB算法提出，如果算法允许一小部分的冲突，就可以进一步的增加可捆绑的特征对，从而也能进一步的降低特征的维度。而通过计算，EFB这部分冲突带来的数据污染，最多带来 $O([(1-\gamma)n]^{-2/3})$ 的损失（其中 γ 是设定的最大冲突比，当前较小时可以比较好的平衡精度和效率）。

EFB的算法语言如下：

Algorithm 3: Greedy Bundling

Input: F: features, K: max conflict count
Construct graph G
searchOrder $\leftarrow$ G.sortByDegree()
bundles $\leftarrow$ { }, bundlesConflict $\leftarrow$ { }
for i ***in*** *searchOrder* **do**
    needNew $\leftarrow$ True
    for $j = 1$ ***to*** *len(bundles)* **do**
        cnt $\leftarrow$ ConflictCnt(bundles[j],F[i])
        if *cnt* + *bundlesConflict[i]* $\leq K$ **then**
            bundles[j].add(F[i]), needNew $\leftarrow$ False
            break
    if *needNew* **then**
        Add $F[i]$ as a new bundle to *bundles*
Output: *bundles*

具体步骤可总结为：

1. 构造一个加权无向图，其顶点为特征，边带权重，权重为两个特征间冲突值；
2. 根据节点的度进行降序排序，度越大，则代表特征之间的冲突越大；
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。

3.5.2 特征合并策略

在解决了特征捆绑问题后，有一个新的问题出现了：捆绑在一起的特征怎么合并在一起使用?

关于这个问题的解决方法比较朴素。

由于LightGBM是基于直方图计算的，那么当多个特征合并到一起时，EFB算法只需要要给不同的特征分配不同的取值空间（bins）即可。

举例来说，假设特征A的取值空间为[0, 10)，那么预期捆绑在一起的特征B（假设其bin的取值空间为[0, 20]），只需要在B原本的bins上都加10就好了。这样处理后的特征B的取值空间就从原本的[0,20]，变为了 [10, 30]。通过这种方式，EFB算法使得捆绑在一起的特征在取值上不重叠。

feature1	feature2	feature_bundle
0	2	+ 4 = 6
0	1	+ 4 = 5
0	2	+ 4 = 6
1	0	1
2	0	2
3	0	3
4	0	4

算法步骤如下：

Algorithm 4: Merge Exclusive Features

Input: $numData$: number of data
Input: F: One bundle of exclusive features
binRanges $\leftarrow \{0\}$, totalBin $\leftarrow 0$
for f ***in*** F **do**
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin $\leftarrow$ new Bin(numData)
for $i = 1$ ***to*** $numData$ **do**
    newBin[i] $\leftarrow 0$
    for $j = 1$ ***to*** $len(F)$ **do**
        if $F[j].bin[i] \neq 0$ **then**
            newBin[i] $\leftarrow$ F[j].bin[i] + binRanges[j]
Output: $newBin$, $binRanges$

小结：LightGBM通过互斥特征捆绑的方式，降低了特征的数量，使得直方图的构建时间复杂度从$O(\#data * \#feature)$变为了$O(\#data * \#bundle)$。

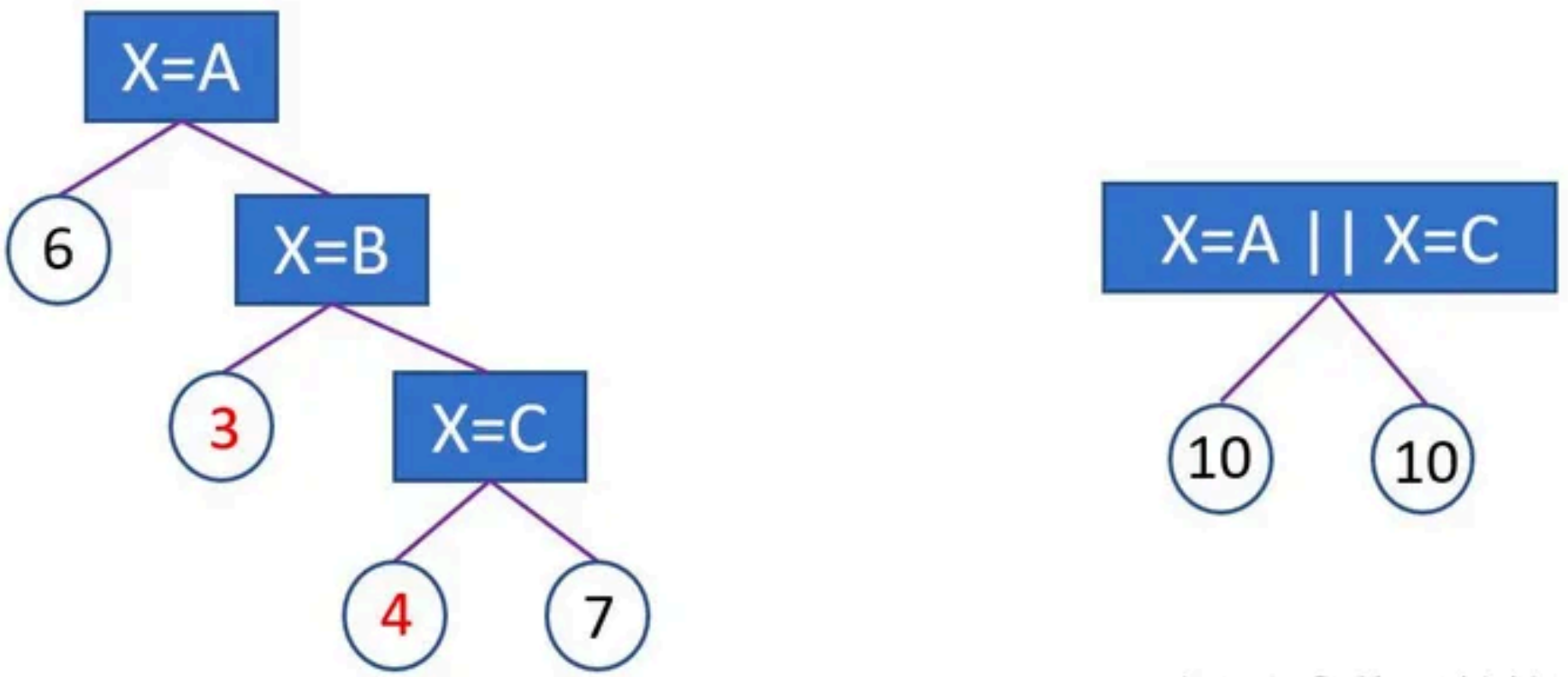
3.6 类别特征处理优化

LightGBM是深度学习模型之前，为数不多的可以直接处理类别特征的算法包。考虑到类别型特征经常出现在日常的模型训练中，且one-hot这种通用的处理方式其实很不适用于决策树家族，因此LightGBM的这一优化可谓是我等调包侠工程师的福音。

首先我们先来解释一下为什么通用的one-hot coding不适用与决策树家族?

原因如下：

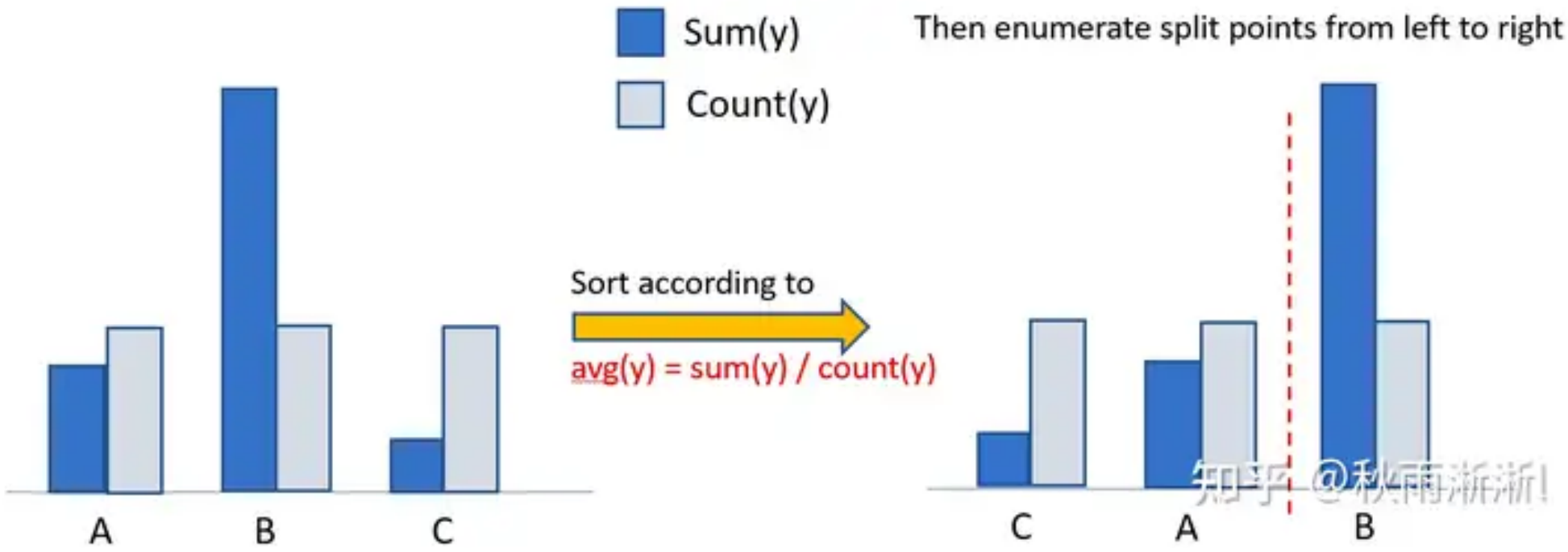
- **切分收益低；**
  - 使用了one-hot 编码后，当节点划分时，只能做one-vs-rest（如是不是猫，是不是狗，等等）。而当特征的维度比较高时（极端情况下把身份证ID用于训练），这时每个类别上的数量都很少，而这就会带来切分不平衡问题，从而导致节点划分的切分增益非常的小（极端情况下，不平衡的切分和不切分几乎没有区别）。
- **影响学习效果；**
  - 如果one-hot后，可以在某个类别特征切分，但由于切分后的零散小空间比较多。正如我们前几篇文章提到过的，决策树家族是利用统计信息来计算节点划分收益，当数据量较小时，统计信息就不准确，因此学到的信息就会变差，从而就会影响模型最终的效果（**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，导致该特征最终得到的重要性会比实际值低。**）



LightGBM的处理方法

LightGBM的优化是基于**many-vs-many**的切分方式（如上右图）。对比左图的one-vs-rest，many-vs-many的切分方式，可以使得数据被切分到两个比较大的空间，这样进一步的学习效果会更好。

具体计算时，LightGBM会基于直方图计算每个类别对应的label均值（即 $avg(y) = \frac{sum(y)}{count(y)}$ ），然后按lable均值从大到下的排序，而后在这个基础依次枚举最优切分点。



假设有 k 个类别，基于朴素枚举算法可知存在 $2^{k-1}-1$ 种组合，时间复杂度为 $O(2^k)$ ，不过LightGBM基于 Fisher 大佬的 《On Grouping For Maximum Homogeneity》实现了$O(klogk)$的时间复杂度。

不过基于上述方式得到的决策树容易过拟合，针对这一操作带来的问题，LightGBM也为此增加了很多约束和正则化。

最后，LightGBM作者表示，通过这种方法在500棵255深度的树时，LightGBM的这单点优化带来了1.5点的auc提升，而在耗时上，只增加了20%。

Comparison (500 tress, each with 255 leaves):

Data		One-hot	Optimal
Expo	AUC	0.78368	0.79847
	Time	138 s	165 s

4. LightGBM的工程优化

相对于XGBoost**分块并行**、**缓存访问** 、**"核外"块计算** 几大针对内存工程优化，LightGBM的三个工程优化则主要是针对于降低通讯成本（不需要过多的考虑内存问题，主要得益于LightGBM一系列的算法优化，较大的较少了内存和计算开销）。

具体的来说，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。

4.1 特征并行

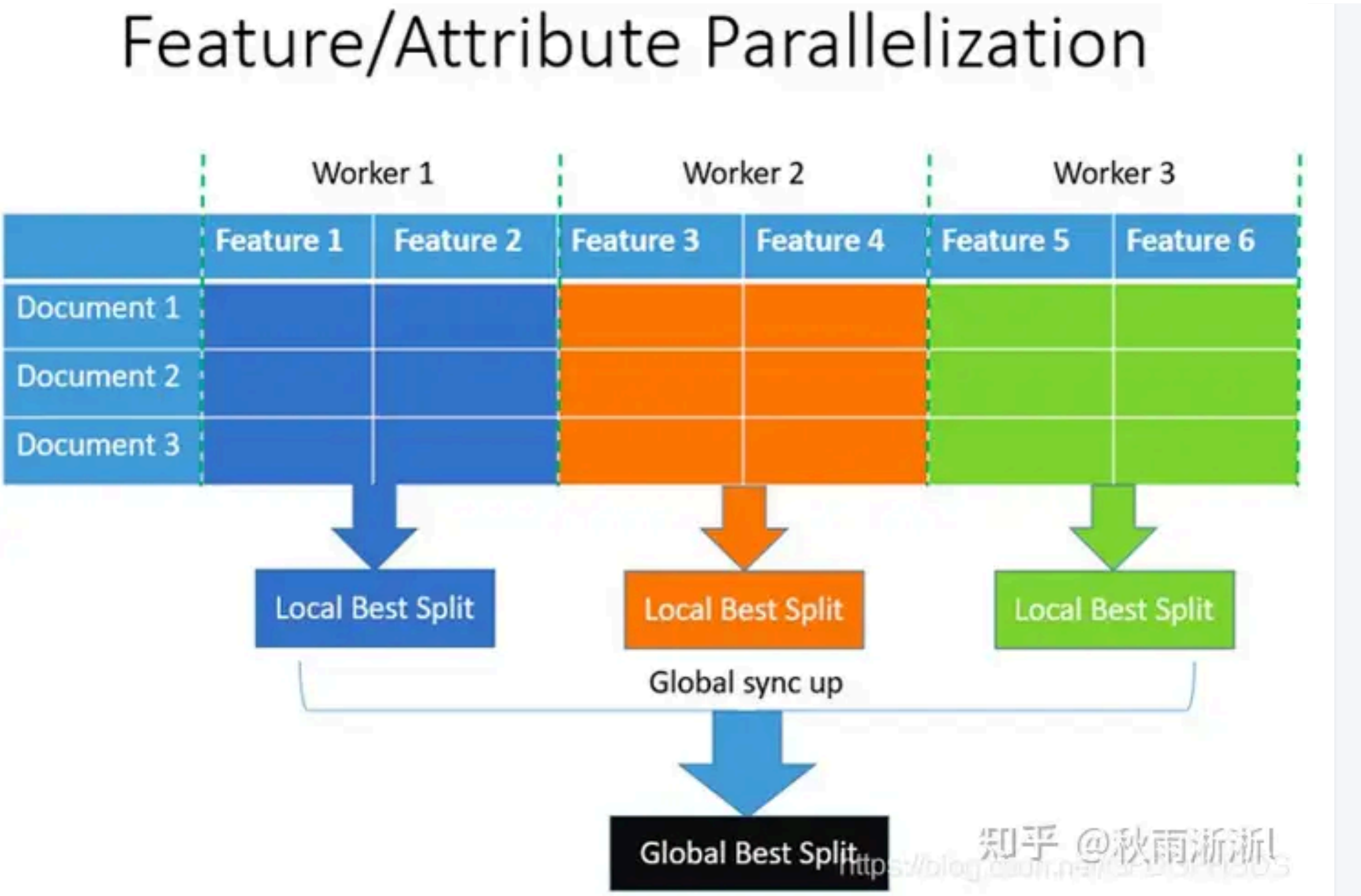
特征并行的思路是：

- 将不同的特征下发到不同的机器，然后各个机器在各自特征集中寻找最优切分点，最后再把最优切分点在集群中共享，从而完成最终的训练。

XGBoost的并行采用的就是这样方式。但我们容易发现这种并行方式有一个比较大的缺点是：

- 节点划分结果需要在集群中不停地广播，而这就有**巨大的通讯成本。**

而LightGBM则不同于XGBoost，LightGBM会在每台机器上保存全部的训练数据，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：



4.2 数据并行

数据并行思路是：

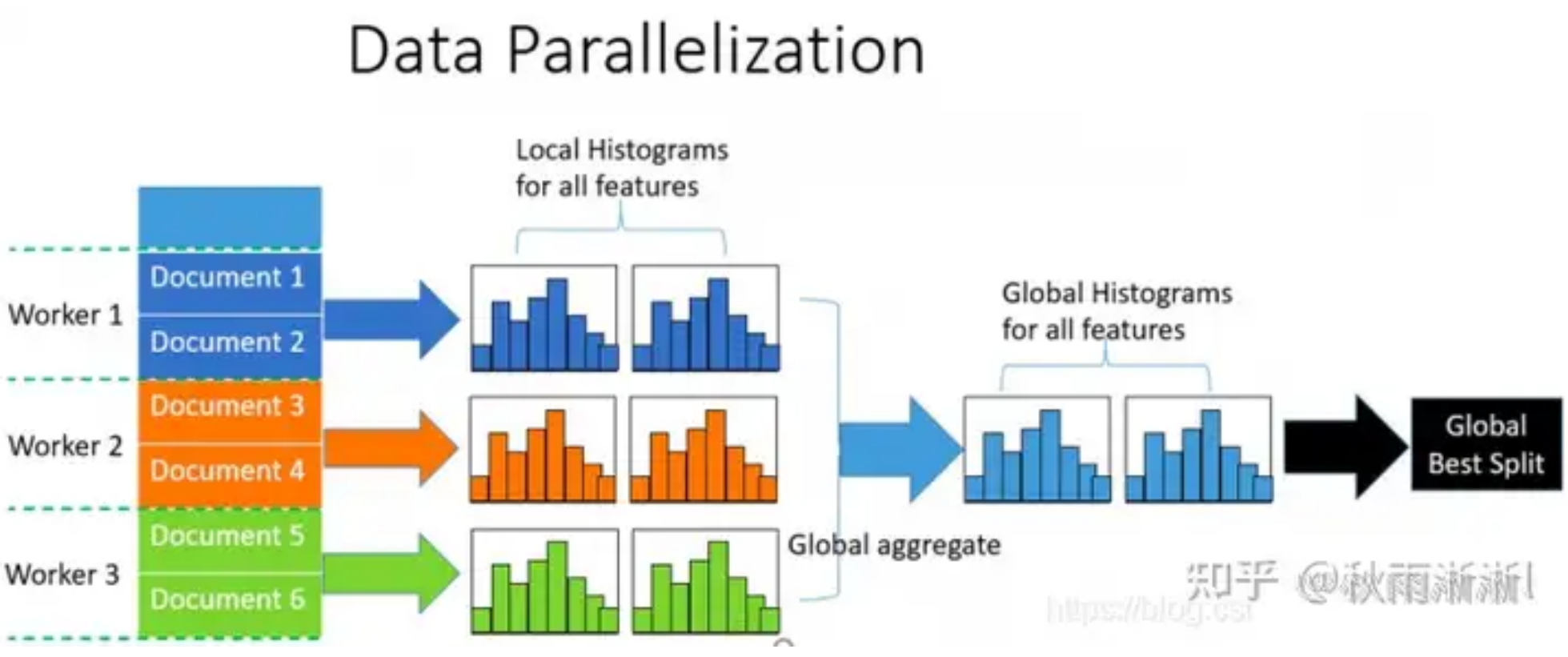
- **让不同的机器先在本地构造直方图，然后进行全局的合并，最后在合并的直方图上面寻找最优分割点。**

但传统数据并行有一个很大的缺点：**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 $O(\#machine * \#feature * \#bin)$；
- 如果使用集成的通信，通讯开销为 $O(2 * \#feature * \#bin)$ 。

而LightGBM通过数据并行中使用**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，且通过利用直方图做差，进一步减少了一半的通信量。

具体过程如下图：



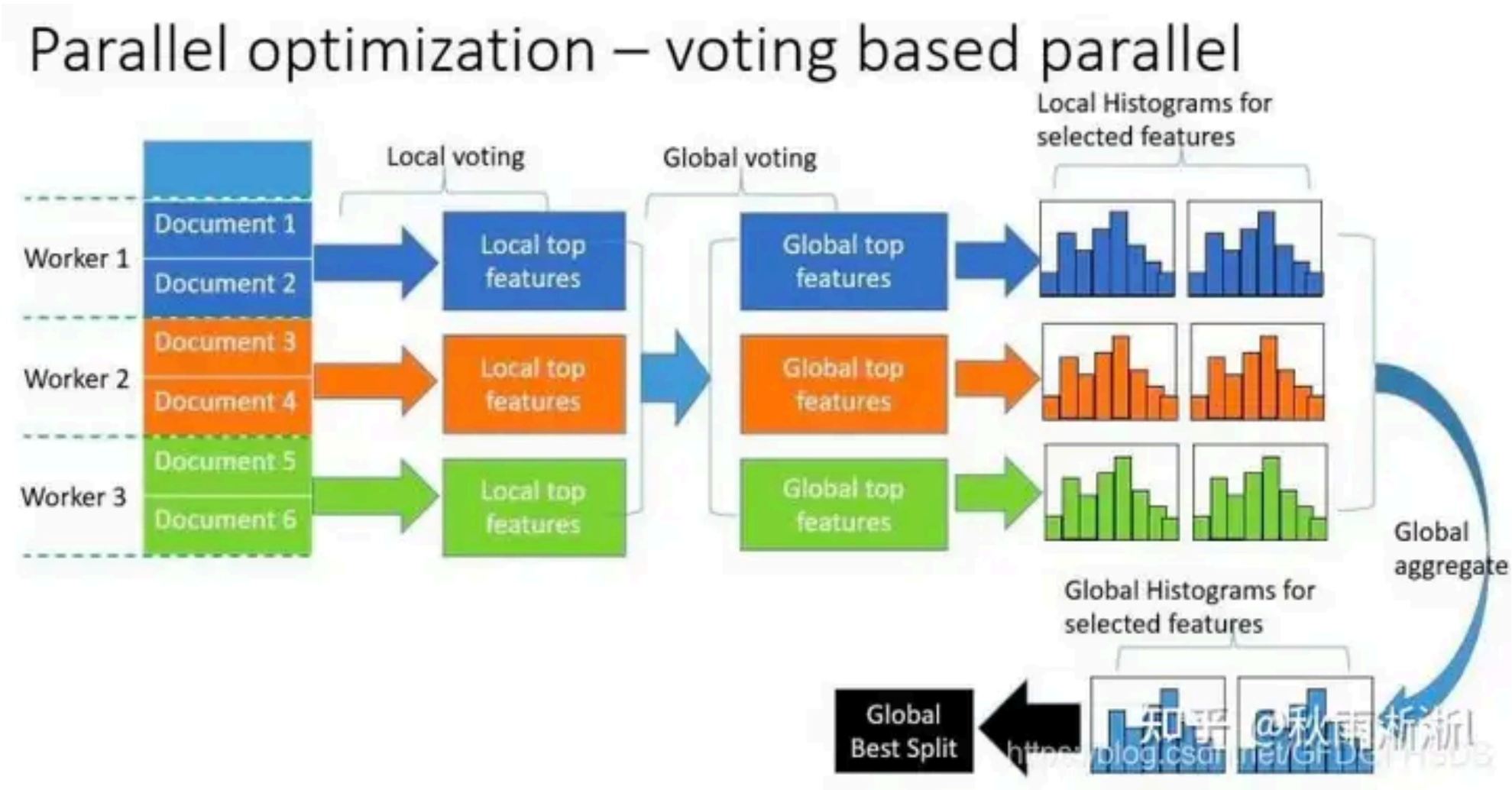
4.3 投票并行

投票并行的目的是：

- 在数据并行的基础上，进一步降低通讯代价，使得通讯代价变成常数级。
  - 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。

具体过程如下图所示，大致步骤为两步：

1. 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；
2. 合并时只合并每个机器选出来的特征。



5.结束语

本文从LightGBM原文从发，从算法和工程两个角度详细的剖析了LightGBM的技术细节和创新之处。

可以看到的是，LightGBM从Gradient Boosting的局限性出发，采用了直方图算法、直方图加速优化、Leaf-wise生成策略以及**单边梯度采样、互斥特征捆绑**、类别特征处理等多重算法优化，使得LightGBM有在没有降低预测精度的同时，更快、更轻，较为完美的符合着互联网人对“快、准、稳、轻”的四大极致追求和向往。

而在面对大数据集时，LightGBM也提出了特征并行、数据并行、投票并行三大工程优化，使得LightGBM具备并行化处理能力。

参考文献

- 如何看待微软新开源的LightGBM? | XGBoost&LightGBM两大 原文大佬精彩评论
- 关于sklearn中的决策树是否应该用one-hot编码？ | LightGBM原文大佬精彩评论
- NIPS 2017 有什么值得关注的亮点？ | LightGBM原文大佬精彩评论
- Microstrong：深入理解LightGBM

发布于 2022-11-03 21:30 · IP 属地广东