深入理解GBDT、XGBoost、LightGBM系列(二)

导语：深入理解了GBDT的原理再来看XGBoost会省力不少，大框架是一致的，但具体实现中又有很多不同，比如树结构分值的计算方法。跟着论文看，你会发现XGBoost文档中的gamma, eta等参数和公式中是一一对应的。

XGBoost有多火就不说了，即使是2021年下半年的一个比赛分享中，大部分同学还是用树模型(XGBoost, LightGBM)解决赛题的。毕竟门槛低、效果好，相信还能流行很长时间。

有了上一篇从加性模型方向对boosting算法的理解，这一篇结合XGBoost的经典论文(参考资料[2])，具体看看XGBoost的实现，相信能有更多的启发。

本文还是按照模型、策略、算法的机器学习经典思维模型展开，以论文为主线，结合参考资料[1]，一起来领略XGBoost框架中的智慧。

模型

先对在使用回归树模型作为基分类器的梯度提升算法进行整体定义。

假定，数据集有n个样本，每个样本有m个特征，$D=\{(x_i,y_i)\}\,(|D|=n,x_i\in\mathfrak{R}^m,y_i\in\mathfrak{R})$ 。

加性模型的预测输出表示为：

$$\hat{y}_i=\phi(x_i)=\sum_{k=1}^{K}f_k(x_i),f_k\in F \quad (1)$$

其中 $F=\{f(x)=w_{q(x)}\}\,(q:\mathfrak{R}^m\to T,w\in\mathfrak{R}^T)$ 是回归树（CART）的空间。q 代表每一棵树的结构，它会把一个样本映射到对应的叶子节点。T 是一棵树叶子节点的数量。 f_k 对应这一棵独立的树结构q和叶子权值w。这个集成模型可以用图表示如下：

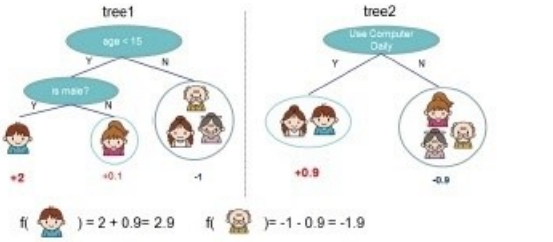


Figure 1: Tree Ensemble Model. The final prediction for a given example is the sum of predictions from each tree.

回归树的每一个叶子节点都有一个连续型的分值，用 w_i 表示第 i 个叶子的分值。

给定一条数据，每一个基分类器也就是每一棵树都会把它划分到一个叶子节点，最终的预测值是把这些叶子的分值加起来。为了学习整个集成模型，最小化的损失函数定义如下，损失函数中包含了正则项。

$$L(\phi)=\sum_i l(\hat{y}_i,y_i)+\sum_k \Omega(f_k) \quad (2)$$

其中 $\Omega(f)=\Upsilon T+\frac{1}{2}\lambda\|w\|^2$

可微分的凸函数 l ,用来衡量预测值 $\hat{y}_i$ 和目标值 y_i 之间的差异。第二项 Ω 用来惩罚模型的复杂性避免过拟合。当惩罚项设为0，目标函数就和传统的GBDT一样了。其中，在惩罚项中Υ、λ表示惩罚系数，T表示给定一颗树的叶节点数，$\|w\|^2$表示每颗树叶节点上的输出分数的平方(相当于L2正则)。

至此整个提升树模型已经定义完毕，接下解决如何优化这个问题，也就是在XGBoost中如何具体应用前向分步法。

用 $\hat{y}_i^{(t)}$ 表示第 i 个样本在第 t 次迭代的预测值，

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i) \quad (3)$$

公式（3）表示样本i在t次迭代后的预测值=样本i在前t-1次迭代后的预测值+当前第t颗树预测值。

损失函数此时表示为：

$$L^{(t)} = \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{i}^{t} \Omega(f_i)$$

$$= \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + constant \quad （4）$$

公式（3）（4）摘自参考资料[1]，这个比原论文的更有利于理解。注意公式(4)上下标的变化，前半部分是对n个样本，后半部分是对 t 次迭代。

公式（4）表示贪心地添加使得模型提升最大的 f_t ，其中constant表示常数项,这个常数项是指前t-1次迭代的惩罚项为一个常数，即公式中 $\sum_{i}^{t} \Omega(f_i)$ 部分，在第t次迭代时，前t-1次迭产生的t-1颗树已经完全确定，则t-1颗树的叶节点以及权重都已经确定，所以变成了常数。

在式（4）中如果考虑平方损失函数，则式（4）可以表示为：

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

$$= \sum_{i=1}^{n} \left(y_i - y_i^{(t-1)} - f_t(x_i)\right)^2 + \Omega(f_t) + constant \quad （5）$$

式（5）中 $y_i - y_i^{(t-1)}$ 表示残差，即经过前t-1颗树的预测之后与真实值之间的差距，也就是在GBDT中所使用的残差概念。

为了能够应用更加广泛的损失函数，在GBDT中引入了梯度的方法，用来近似表示两次迭代之间的损失函数变化。

在XGBoost中使用二阶泰勒展开式近似表示式（5），依据参考资料[3]这样能更快的优化目标函数，但要求损失函数必须二阶可导。

泰勒展开式的二阶形式为：

$$f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2} f''(x)\Delta x^2 \quad (6)$$

定义 $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ 则根据式(6)可以将式(4)表示为：

$$L^{(t)} \simeq \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + constant \quad (7)$$

注意式(7)中 $l(y_i, \hat{y}_i^{(t-1)})$ 部分表示前t-1次迭代的损失函数，在当前第t次迭代来说已经是一个确定的常数，省略常数项则得到下面的式子：

$$L^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \quad （8）$$

则目标函数变成了公式(8)。可以看出目标函数只依赖于数据点的一阶和二阶导数。

其中 $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$

至此完成了梯度提升(gradient boosting)部分的定义，接下来细化当基分类器为回归树时损失函数的演进。

定义集合 $I_j = \{i|q(x_i) = j\}$ 为树的第j个叶节点上的所有样本点的集合，即给定一颗树，所有按照决策规则被划分到第j个叶节点的样本集合。则根据式(2)中对模型复杂度惩罚项的定义，将其带入式(8)有：

$$L^{(t)} = \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \Omega(f_t)$$

$$= \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T}w_j^2$$

$$= \sum_{j=1}^{T}[(\sum_{i\in I_j}g_i)w_j + \frac{1}{2}(\sum_{i\in I_j}h_i + \lambda)w_j^2] + \gamma T \quad (9)$$

对式(9)进行求导：

$$令 \frac{\partial L^{(t)}}{\partial w_j} = 0$$

$$\Rightarrow (\sum_{i\in I_j}g_i) + (\sum_{i\in I_j}h_i + \lambda)w_j = 0$$

$$\Rightarrow (\sum_{i\in I_j}h_i + \lambda)w_j = -\sum_{i\in I_j}g_i$$

$$\Rightarrow w_j^* = -\frac{\sum_{i\in I_j}g_i}{\sum_{i\in I_j}h_i + \lambda} \quad (10)$$

对于凸函数，导数为0的点，即为极小值点。

将式(10)带入式(9)中得到式(11)：

$$\tilde{L}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j}g_i)^2}{\sum_{i\in I_j}h_i + \lambda} + \gamma T \quad (11)$$

令 $G_i = \sum_{i\in I_j}g_i, H_i = \sum_{i\in I_j}h_i$ 则，式(11)可以简化为式(12)

$$\tilde{L}(q)^{(t)} = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T \quad 式(12)$$

在XGBoost中，式(12)被用作评价树结构 q 质量的分值，其计算方式如下图。

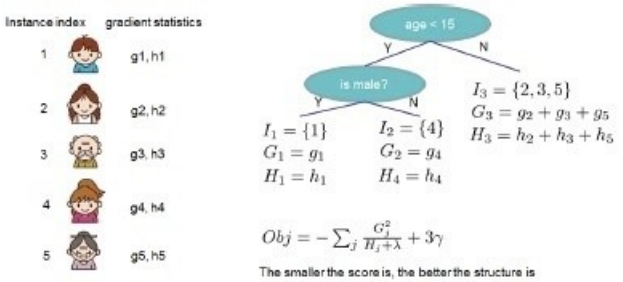


Figure 2: Structure Score Calculation. We only need to sum up the gradient and second order gradient statistics on each leaf, then apply the scoring formula to get the quality score.

这个分值和评估决策树的不纯度分值是相似的，但它是从更为通用的目标函数推导出来的。在XGBoost中用式(12)的分值来决定是否进行分裂，这里和CART中采用基尼系数作为分值是不同的。

XGBoost定义特征选择和切分点选择的指标如下，

$$Gain = \frac{1}{2}[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}] - \gamma \quad (13)$$

XGBoost中使用过（式13）判断切分增益，Gain值越大，说明分裂后能使目标函数减少越多，就越好。其中 $\frac{G_L^2}{H_L + \lambda}$ 表示在某个节点按条件切分后左节点的得分， $\frac{G_R^2}{H_R + \lambda}$ 表示在某个节点按条件切分后右节点的

$\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}$ 得分，γ 表示切分后模型复杂度的增加量。现在有了判断增益的方法，就需要使用该方法去查找最优特征以及最优切分点。

注意这里的γ有没有似曾相识的感觉，在XGBoost的参数中也有一个gamma，如下图，从参数说明来看和式(13)完全吻合，这也就能理解为什么会有这样一个参数。

- gamma [default=0]
  - minimum loss reduction required to make a further
    algorithm will be.
  - range: [0,∞]

根据论文顺序，在这里提供了两个防止过拟合的方案，分别是剪枝和列采样。剪枝的方式是加入了一个类似梯度下降中步长的参数 η，对应XGBoost中的参数eta，如下图。

- eta [default=0.3]
  - step size shrinkage used in update to prevents overf
    and eta actually shrinks the feature weights to make
  - range: [0,1]

剪枝降低了每一棵树和叶子空间对后续树学习的影响。

列采样，也就是随机去掉一些特征，和随机森林采用的方案一样。根据用户反馈，列采样对于抑制过拟合的效果要优于行采样，同时提高了并行计算的速度。XGBoost同样支持行采样。

算法

在XGBoost中树是如何分裂的，具体的分裂算法在论文中给出了2种，一种是Exact Greedy Algorithm for Split Finding，即精确的贪婪分裂算法；另一种是Approximate Algorithm for Split Finding，即近似的分裂算法。两个算法的具体流程下面的两图。

两者的差别主要体现在对连续型特征的处理上，精确的算法会按顺序遍历连续型特征的每一个点，而近似的算法会首先对连续型特征按照百分位数(percentile)提取出建议的分裂点，然后再遍历这些建议的分裂点。XGBoost两者是都支持的。

Algorithm 1: Exact Greedy Algorithm for Split Finding

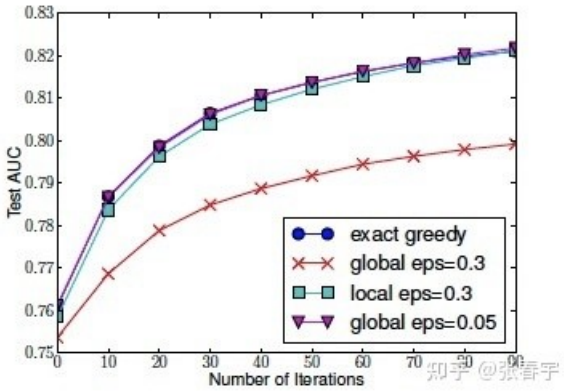
Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i\in I} g_i$, $H \leftarrow \sum_{i\in I} h_i$
for $k = 1$ *to* m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted*(I, *by* $\mathbf{x}_{jk}$) **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ *to* m do
  Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature k.
  Proposal can be done per tree (global), or per split(local).
end
for $k = 1$ *to* m do
  $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} g_j$
  $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} h_j$
end
Follow same step as in previous section to find max score only among proposed splits.

引入近似算法是因为精确算法无法支持两种场景，一是数据无法全部读入内存，二是分布式训练。

近似算法又有两种变形，全局(global)方案和局部(local)方案。全局方案建议分裂点发生在树构建的开始阶段，而局部方案在每一次树分裂之后都会重新建议分裂点。全局方案需要更少的建议步骤，但需要更多的分裂点，因为后续不再修正分裂点。局部方案每次分裂后都修正分裂点，更适合比较深的树。当给出足够的候选分裂点时，全局方案和局部方案一样准确，如下图所示。



在近似算法中最重要的一步就是如何提供候选的分裂点，一个特征的百分位数使得分裂点在数据上分布均匀。正式的表示是，对数据集 $D_k = \{(x_{1k}, h_1), (x_{2k}, h_2), \ldots, (x_{nk}, h_n)\}$ 表示每个样本的第k个特征值(x_{nk})和二阶导数(h_{nk})的集合。定义排名函数 r_k ：

$$r_k(z) = \frac{1}{\sum_{(x,h) \in D_k} h} \sum_{(x,h) \in D_k, x < z} h \quad (14)$$

式(14)表示数据集中第k个特征值小于z的样本所占比例，就是特征值小于z的样本的权重和，占所有样本权重总和的百分比。目标是找到一个候选切分点 $\{s_{k1}, s_{k2}, \ldots, s_{kl}\}$ ，可以根据下式进行侯选点的选取

$$|r_k(s_k, j) - r_k(s_k, j+1)| < \epsilon \quad (15)$$

式(15)表示落在两个相邻的候选切分点之间样本占比小于某个值 ϵ （很小的常数），也就是有 $1/\epsilon$ 个候选切分点。

由式(14)看的样本是以二阶导数作为加权考虑占比的，那么问题来了，为什么使用二阶导数作为加权呢？以下内容直接摘自参考资料[1]。

对目标函数式(8)进行改写，过程如下：

$$L^{(t)} = \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \Omega(f_t)$$

$$= \sum_{i=1}^{n}\frac{1}{2}h_i[\frac{2g_i f_t(x_i)}{h_i} + f_t^2(x_i)] + \Omega(f_t)$$

$$= \sum_{i=1}^{n}\frac{1}{2}h_i[\frac{2g_i f_t(x_i)}{h_i} + f_t^2(x_i) + \left(\frac{g_i}{h_i}\right)^2 - \left(\frac{g_i}{h_i}\right)^2] + \Omega(f_t)$$

$$= \sum_{i=1}^{n}\frac{1}{2}h_i\left[f_t(x_i) - \left(-\frac{g_i}{h_i}\right)\right]^2 + \Omega(f_t) - \sum_{i=1}^{n}\frac{1}{2}\frac{g_i^2}{h_i}$$

$$= \sum_{i=1}^{n}\frac{1}{2}h_i\left[f_t(x_i) - \left(-\frac{g_i}{h_i}\right)\right]^2 + \Omega(f_t) - constant \quad (16)$$

式(16)可以看成权重 为h_i 的label为 $\left(-\frac{g_i}{h_i}\right)$（ps：这个值比作者论文中多出一个负号，有文章说作者论文里面少写了负号。个人觉得这个正负号不是关注重点，它不是一种定量的计算，而是表示一种以二阶导数作为加权的合理性说明。）的平方损失函数，其权重 h_i 则为二阶导数。由此表明将二阶导数作为样本权重的考虑是合理的。

上述的分裂算法是针对连续特征，那么对于离散特征，XGBoost中是如何处理的呢，接下来看看工具中对于稀疏值的处理方案。

在实际数据中，稀疏特征是非常普遍的，比如存在缺失值，在统计中未出现的值，one-hot编码等，所以对稀疏值的处理非常必要。论文中采用为每一个树节点增加一个默认方向的方式处理稀疏值，默认方向有两个选择，通过学习的方式决定选择哪个方向。具体的算法如下图，

Algorithm 3: Sparsity-aware Split Finding

Input: I, instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: d, feature dimension
Also applies to the approximate setting, only collect statistics of non-missing entries into buckets
$gain \leftarrow 0$
$G \leftarrow \sum_{i\in I} g_i, H \leftarrow \sum_{i\in I} h_i$
for $k = 1$ *to* m **do**
  // enumerate missing value goto right
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for j *in sorted*$(I_k,$ *ascent order by* $\mathbf{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
  // enumerate missing value goto left
  $G_R \leftarrow 0,\ H_R \leftarrow 0$
  for j *in sorted*$(I_k,$ *descent order by* $\mathbf{x}_{jk})$ **do**
    $G_R \leftarrow G_R + g_j,\ H_R \leftarrow H_R + h_j$
    $G_L \leftarrow G - G_R,\ H_L \leftarrow H - H_R$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split and default directions with max gain

由算法可见，在最初还是采用近似的候选分裂点选择算法，但此时仅考虑非缺失值。然后在每次切分是，让缺失值分别被切分到左节点以及右节点，计算得分值比较两种切分方法哪一个更优，这样每个特征的缺失值都会学习到一个最优的默认切分方向。

并且这样处理后，还优化了迭代过程中的计算量，因为算法迭代时只考虑了非缺失值数据的遍历，缺失值数据直接被分配到左右节点，所需要遍历的样本量变少了。具体的性能数据如下图。

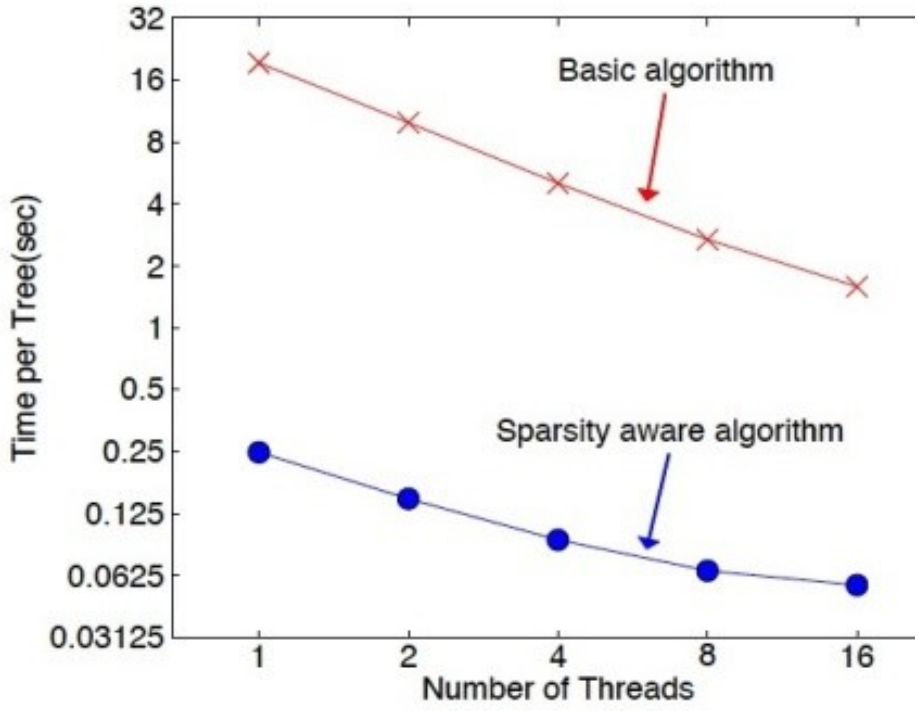


Figure 5: Impact of the sparsity aware
on Allstate-10K. The dataset is sparse n
to one-hot encoding. The sparsity aware
is more than 50 times faster than the nai
that does not take sparsity into considera

至此，XGBoost中的算法部分就完成了，实现了树之间的迭代，以及每一棵树如何分类。

工程优化

在论文中专门有一个部分介绍系统设计，用于解释XGBoost为什么执行效率那么高，在这里做一个简单的介绍。

首先是用于并行学习的列块(column block)，在树的学习中最耗时的是在各个特征中寻找分裂点，而寻找分裂点中最耗时的是对特征值进行排序。为了减少排序的消耗，XGBoost会把排好顺序的数据保存到内存块中，称为”block"。这些内存块仅需在训练前计算一次，后续迭代中即可重复使用。

对于精确的贪婪算法，所有的数据存储到一个block中，在做分裂点查找时所有叶子节点一起做，所以对block的一次遍历即可获取到所有的分裂候选节点的统计数据，具体做法如下图。

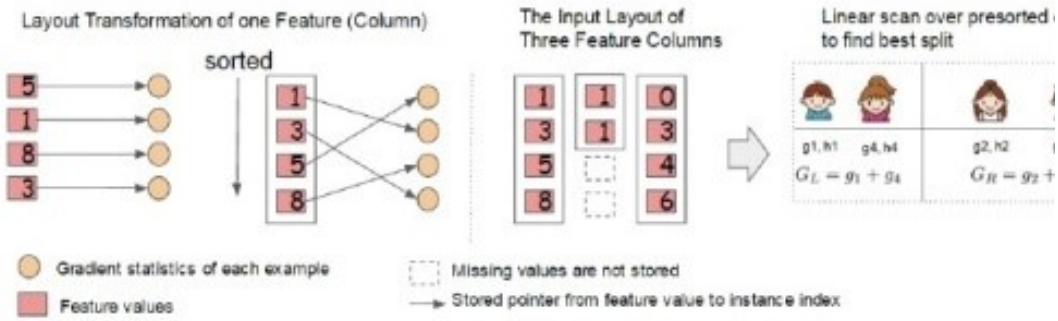


Figure 6: Block structure for parallel learning. Each column in a block is sorted by the
value. A linear scan over one column in the block is sufficient to enumerate all the sp

由图中可见，在block块中会按特征值排好序，然后通过引用指向对应的已经计算好的梯度。

对于近似的算法也非常有帮助，这时可以使用多个block，每一个block中存储对应的样本数据。不同的block可以存储在不同的机器上或者存储到硬盘中。有了这些排序好的block，查找建议候选分裂点时，仅需要线性的扫描那些排好序的列。这对于local模式尤其重要，因为需要频繁的变换候选分裂点。

有了这些block，每一列的分裂点查找可以并行。同时，它还支持列采样，因为在一个块中选择列的子集很容易。

通过block结构的使用优化了分裂点查找的计算复杂度，但在取梯度时是通过引用，这些没有存储在连续的内存中，在梯度数据不符合CPU缓存的要求或者没有存储在缓存中时会降低运算效率。

对于精确的算法，会在每一个线程中申请一块缓存空间，然后不断的读取梯度数据到这个缓存中，通过mini-batch的方式进行加速。

对于近似算法，通过选择合适的block大小来进行加速，太小的block每一个线程的负载过小，降低了并行效率。过大的block导致数据不能进入CPU的缓存。根据这些因素选择一个合适大小的块，能够提高运算效率。经过实验 2^{16} 条数据是最高效的选择。

在对CPU和内存进行优化之后，为了充分的利用机器资源，XGBoost对硬盘的应用也做了优化，以应对内存无法完全加载数据的情况。首先把数据分成好很多块，然后存储到硬盘中。在计算时会用一个单独的取数据线程，提前把数据块加载到内存中，所以计算和数据读取会同时进行。但这还不足以完全解决硬盘读取占用过多时间的问题，所以降低读取开销和增加硬盘IO的吞吐量很重要，XGBoost采用了两种办法。

一是块压缩，每个块按列压缩，然后用一个独立的线程在读入内存时进行解压缩，这样就用解压的计算交换了硬盘的读取消耗。对于行的索引仅记录每个块的第一个，然后用一个16位的整型记录后续行索引的offset。

二是块分片(block sharding)，把不同的片保存到不同的硬盘中，给每一个盘分配一个预取线程，把数据读到在内存设置的缓存中，然后训练线程再从这些缓存中读取数据。这样做的目的是增加硬盘的吞吐量。

至此论文中的重点内容都介绍完了，真心佩服大神的理论功底和对性能的极致追求。

在写这篇文章中大量借鉴了参考文献[1]，几个关键节点，比如如何由损失函数转化为树的分值，建议候选切分点的公式变换，都直接照抄了。我一度怀疑自己写这篇文章干嘛，纸上得来终觉浅，自己写一遍，和只看大神的文章比，理解程度有很大的不同，否则就提不出这些问题。希望朋友们读完这篇文章对理解原论文有帮助。

参考文献

[1] 金贵涛：对xgboost的理解

[2] 《XGBoost: A Scalable Tree Boosting System》https://arxiv.org/pdf/1603.02754.pdf

[3] 《Additive logistic Regression: a statistical view of boosting》 https://web.stanford.edu/~hastie/Papers/AdditiveLogisticRegression/alr.pdf