# XGBoost: A Scalable Tree Boosting System

摘要

提升树是一种非常有效且被广泛使用的机器学习方法。在本文中，我们描述了一个名为 XGBoost 的有延展性的端到端的树提升系统，数据科学家们广泛使用该系统来实现许多机器学习挑战的最新成果。我们提出了一种新颖的稀疏数据感知算法用于稀疏数据，一种带权值的分位数算法(weighted quantile sketch)来近似实现树的学习。更重要的是，我们提供有关缓存访问模式，数据压缩和分片的见解，以构建有延展性的提升树系统。通过结合这些见解，XGBoost 可用比现系统少得多的资源来处理数十亿规模的数据。

关键词

大规模机器学习

## 1、介绍

机器学习和数据驱动方法在许多领域都非常重要。智能垃圾邮件分类器通过学习大量垃圾邮件数据和用户反馈来保护我们的电子邮件;广告系统学会将正确的广告与正确的背景相匹配;欺诈检测系统保护银行免受恶意攻击者的侵害异常事件检测系统帮助实验物理学家找到导致新物理学的事件。推动这些成功应用的因素有两个：使用捕获复杂数据依赖性的有效（统计）模型和可从大型数据集中学习感兴趣模型的可扩展学习系统。

在实践中使用的机器学习方法中，梯度提升树是一种在许多应用中闪耀的技术。提升树已被证明可以在许多标准分类基准上给出最优的结果。LambdaMART [5]是一种用于排名的提升树变体，它可以为排名问题获得最先进的结果。除了独立用作预测器之外，它还被整合到实际生产流水线中，用于广告点击率预测。最后，它是集成方法的常用选择，并用于诸如 Netflix 奖等挑战。

在本文中，我们描述了 XGBoost，一种用于树提升的可扩展机器学习系统。该系统可作为开源软件包使用。该系统的影响已在许多机器学习和数据挖掘挑战中得到广泛认可。以机器学习竞赛网站 Kaggle 中的挑战为例，在 2015 年 Kaggle 博客上发布的 29 个挑战获胜解决方案中，有 17 个解决方案使用了 XGBoost。在这些解决方案中，8 个仅使用 XGBoost 来训练模型，而大多数其他解决方案将 XGBoost 与神经网络结合在一起。为了比较，第二种最常用的方法

是深度神经网络，用于 11 种解决方案。该系统的成功也在 KDDCup 2015 中见证，其中 XGBoost 被前 10 名中的每个获胜团队使用。此外，获奖团队报告说，整体方法的表现仅优于配置良好的 XGBoost。

这些结果表明，我们的系统在广泛的问题上提供了最优的结果。这些获胜解决方案中存在的问题包括：商店销售预测; 高能物理事件分类;网络文本分类;顾客行为预测;运动检测;广告点击率预测;恶意软件分类;产品分类;危险风险预测;大规模的在线课程辍学率预测。虽然域依赖数据分析和特征工程在这些解决方案中发挥着重要作用，但 XGBoost 是学习者的共识选择这一事实表明了我们的系统和树提升的影响和重要性。

XGBoost 成功背后最重要的因素是它在所有场景中的可扩展性。该系统在单台机器上运行速度比现有流行解决方案快十倍以上，并且可以在分布式或内存限制设置中扩展到数十亿个示例。 XGBoost 的可扩展性归功于几个重要的系统和算法优化。这些创新包括：一种新颖的树学习算法，用于处理稀疏数据;理论上合理的加权分位数草图过程使得能够在近似树学习中处理实例权重。并行和分布式计算使学习更快，从而加快了模型的开发速度。更重要的是，XGBoost 利用核外计算，使数据科学家能够在桌面上处理数亿个示例。最后，结合这些技术使端到端系统以最少的集群资源扩展到更大的数据更令人兴奋。本文的主要贡献如下：

- 我们设计并构建了一个高度可扩展的端到端树推进系统。
- 我们提出了一个理论上合理的加权分位数算法，用于高效的计算。
- 我们介绍了一种新颖的稀疏感知算法，用于并行树学习。
- 我们提出了一种有效的缓存感知块结构，用于核外树学习。

虽然有一些关于提升树并行工作的研究，但尚未探索诸如核外计算，缓存感知和稀疏感知学习等方向。更重要的是，结合所有这些方面的端到端系统为现实世界的用例提供了一种新颖的解决方案。这使得数据科学家和研究人员能够构建提升树算法的强大变体。除了这些主要贡献之外，我们还在提出正规化学习目标方面做出了进一步的改进，我们将包括完整性。

在本文的其余部分安排如下。 我们将首先回顾树木提升并在第 2 节中引入正则化目标。然后，我们描述了第 3 节中的分裂发现方法以及第 4 节中的系统设计，包括相关的实验结果，为我们描述的每个优化提供定量支持。相关工作在第 5 节中讨论。详细的端到端评估包含在第 6 节中。最后，我们在第 7 节中

总结了论文。

# 2、提升树介绍

我们在本节中回顾了梯度提升树算法。推导遵循现有文献中梯度提升的相同思想。特别地，二阶方法源自 Friedman 等人。我们对正则化目标做了微小改进，这在实践中很有帮助。

## 2.1 正则化的学习目标

对于具有 n 个示例和 m 个特征的给定数据集，$D = \{(x_i, y_i)\}(|D|=n, x_i \in R_m, y_i \in R)$，树集合模型（如图 1 所示）使用 K 附加功能来预测输出。

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

其中 $F = \{f(x)= w \cdot q(x)\}(q:R_m \rightarrow T, w \in R_T)$是回归树的空间(也称为 CART)。这里 q 表示将示例映射到相应叶索引的每棵树的结构。T 是树中叶子的数量。每个 $f_k$ 对应于独立树结构 q 和叶权 w。与决策树不同，每个回归树在每个叶子上包含连续分数，我们使用 $w_i$ 来表示第 i 个叶子上的分数。

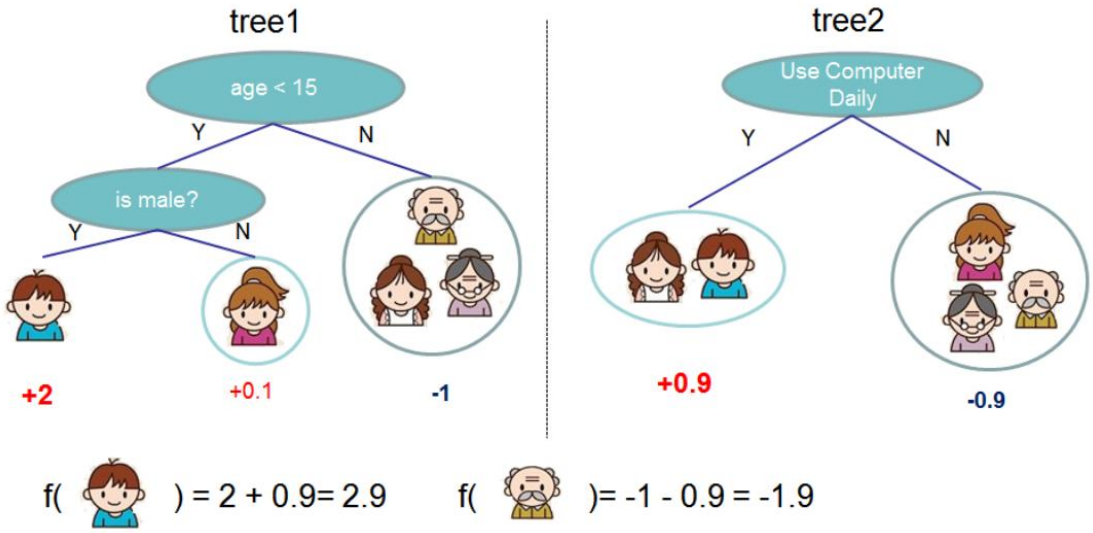



图 1：树集合模型（给定示例的最终预测是来自每棵树的预测的总和）

对于给定的示例，我们将使用树中的决策规则（由 q 给出）将其分类为叶子并通过对相应叶子中的分数求和（由 w 给出）来计算最终预测。要了解模型

中使用的函数集，我们最小化以下正则化目标。

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$
$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2 \tag{2}$$

这里 l(y^,y)是可微分凸损函数，其测量预测 $y_i$ 和目标 $y_i$ 之间的差异。第二项 Ω 惩罚模型的复杂性（即回归树函数）。额外的正则化项使得最终学习的权重更加平滑，以避免过拟合现象。很明显正则化目标将倾向于选择采用简单的预测函数模型。类似的正则化技术已被用于正则化贪心森林（RGF）模型。我们的目标和相应的学习算法比 RGF 更简单，更易于并行化。当正则化参数设置为零时，目标将回退到传统的梯度树提升。

## 2.2 梯度提升树

方程（2）中的树集合模型中包含函数这种参数，这不能使用欧几里德空间中的传统优化方法进行优化。相反，模型以附加方式进行训练。形式上，让 y（t）成为第 t 次迭代的第 i 个实例的预测，我们需要添加 $f_t$ 以最小化以下目标：

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

这意味着我们根据方程（2）贪婪地添加最能改进我们模型的 $f_t$。二阶近似可用于在一般设置中快速优化目标。

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

其中 gi =∂y(t-1)|(yi,y(t-1))和 $h_i$ =∂2y(t-1)l(yi,y(t-1))是一阶和二阶梯度统计损失功能。我们可以删除常数项以在步骤 t 获得以下简化目标。

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t) \tag{3}$$

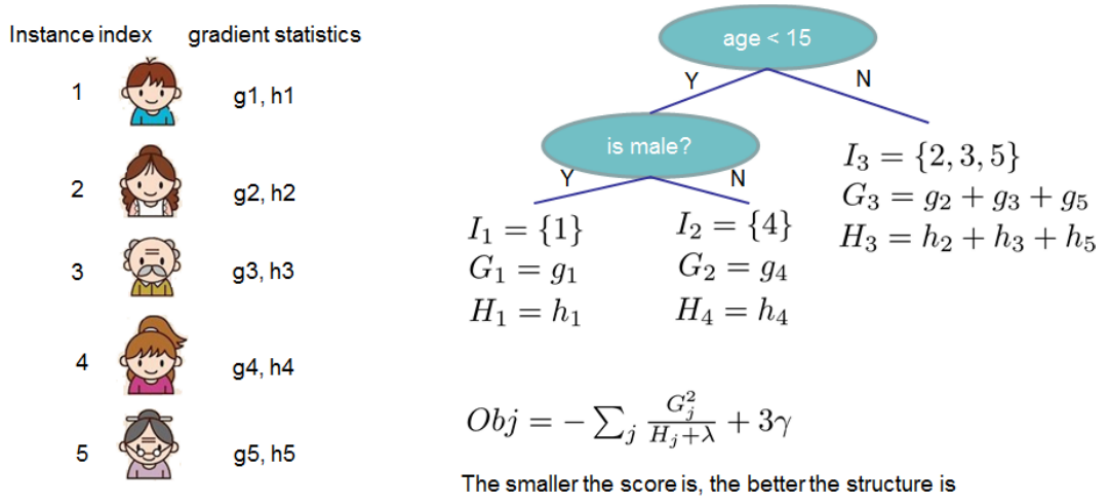



图 2：结构分数计算(我们只需要总结每个叶子上的梯度和二阶梯度统计量，然后应用评分公式来获得质量分数)

将 Ij = {i | q（$x_i$）= j}定义为叶 j 的实例集。我们可以通过如下扩展 Ω 来重写 Eq(3)

$$\begin{aligned} \tilde{\mathcal{L}}^{(t)} &= \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \end{aligned} \tag{4}$$

对于固定结构 q(x)，我们可以通过下式计算叶片 j 的最佳权重 w * j：

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}, \tag{5}$$

并通过下式计算相应的最优值：

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T. \tag{6}$$

方程（6）可以用作评分函数来测量树结构 q 的质量。该评分类似于评估决策树的杂质评分，除了它是针对更广泛的目标函数得出的。图 2 说明了如何计算得分。

通常，不可能枚举所有可能的树结构 q。使用从单个叶子开始并迭代地将分支添加到树的贪婪算法。 假设 IL 和 IR 是拆分后左右节点的实例集。 Lettting I = IL IR，然后分割后的损失减少

$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{7}$$

该公式通常用于评估分割候选人。

## 2.3 收缩和列子采样

除了第二节中提到的正则化目标。 2.1，使用另外两种技术来进一步防止过度拟合。 第一种技术是 Friedman [11]引入的收缩。 在树木提升的每个步骤之后，收缩比例新增加了因子 η 的权重。 与随机优化中的学习速率类似，收缩减少了每棵树的影响，并为将来的树木留出了改进模型的空间。 第二种技术是列（特征）子采样。 这种技术用于 RandomForest [4,13]，它是在商业软件 TreeNet 中实现的。

用于梯度提升，但未在现有的开源软件包中实现。 根据用户反馈，使用列子采样可以防止过度拟合，甚至比传统的行子采样（也支持）更加过度拟合。列子样本的使用也加速了后面描述的并行算法的计算。

**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**
  $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$
  $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$
**end**
Follow same step as in previous section to find max score only among proposed splits.

# 3. 分裂查找算法

## 3.1 基本精确贪心算法

树学习中的关键问题之一是找到方程（7）所示的最佳分裂。 为此，分裂查找算法枚举所有特征上的所有可能分割。 我们称之为精确的贪婪算法。大多数现有的单机树提升实现，例如 scikit-learn [20]，R 的 gbm [21]以及 XGBoost 的单机版本都支持精确的贪心算法。 确切的贪婪算法如 Alg 所示。 1.枚举连续特

征的所有可能分裂在计算上要求很高。 为了有效地执行此操作，算法必须首先根据特征值对数据进行排序，并按排序顺序访问数据，以累积方程（7）中结构分数的梯度统计。

## 3.2 近似算法

确切的贪婪算法非常强大，因为它贪婪地枚举了所有可能的分裂点。 但是，当数据不完全适合存储器时，不可能有效地这样做。 在分布式设置中也出现同样的问题。 为了在这两种设置中支持有效的梯度树增强，需要近似算法。

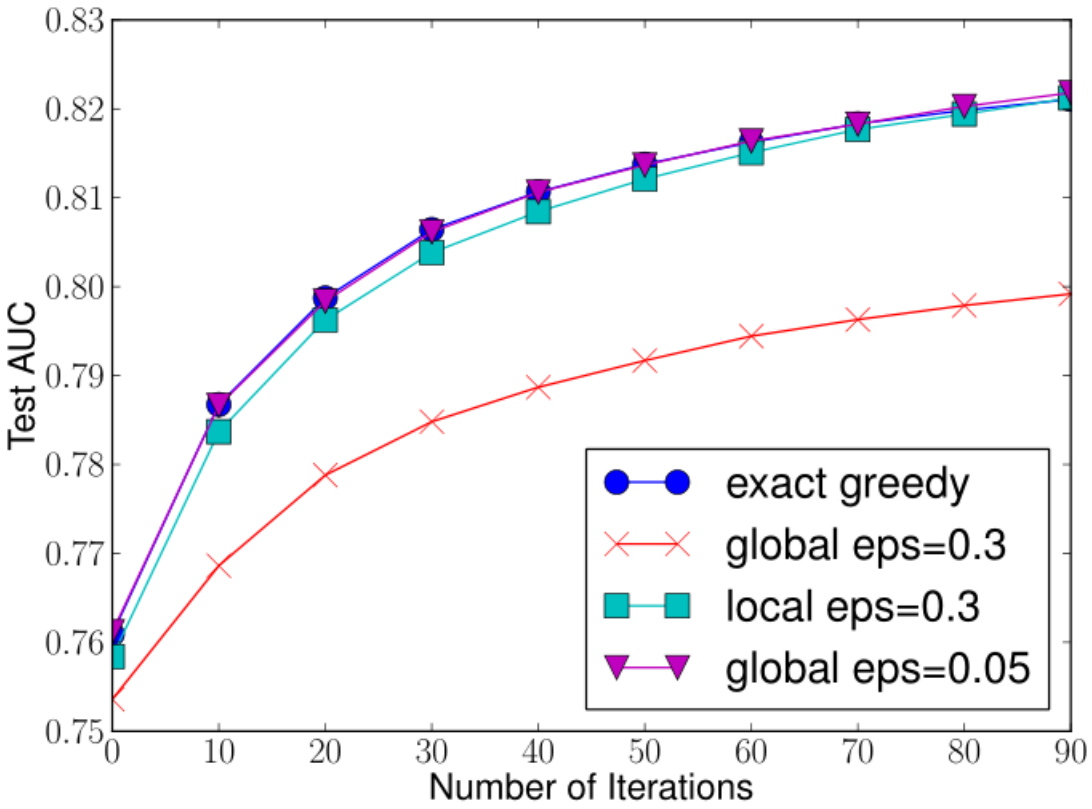



图 3：Higgs 10M 数据集的测试 AUC 收敛的比较。 eps 参数对应于近似草图的精度。 这大致转换为提案中的 1 / eps 桶。 我们发现本地提案需要更少的存储桶，因为它可以优化分割候选。

我们总结了一个近似的框架，类似于过去的文献[17,2,22]中提出的观点，在 Alg 中。 2.总之，该算法首先根据特征分布的百分位数提出候选分裂点（具体标准将在 3.3 节中给出）。 然后，算法将连续特征映射到由这些候选点分割的区域，汇总统计信息并根据聚合统计信息在提案中找到最佳解决方案。

该算法有两种变体，具体取决于提议的时间。全局变体在树构建的初始阶段提出所有候选分裂，并在所有级别使用相同的分裂查找提议。每次拆分后，

本地变体会重新提出。全局方法比本地方法需要更少的提议步骤。但是，全球提案通常需要更多的候选点，因为候选人在每次拆分后都没有得到完善。当地提案在拆分后对候选人进行了细化，并且可能更适合更深层的树木。图 3 给出了希格斯玻色子数据集上不同算法的比较。我们发现本地提案确实需要较少的候选者。全球提案可以与给予足够候选人的当地提案一样准确。

用于分布式树学习的大多数现有近似算法也遵循该框架。值得注意的是，也可以直接构建梯度统计的近似直方图[22]。也可以使用其他变化的分类策略而不是分位数[17]。分位数策略受益于可分发和可重构，我们将在下一小节中详细介绍。从图 3 中，我们还发现，在给定合理的近似水平的情况下，分位数策略可以获得与精确贪婪相同的精度。

我们的系统有效地支持单机设置的精确贪婪，以及所有设置的本地和全局提议方法的近似算法。用户可以根据自己的需要自由选择方法。

## 3.3 加权分位数素描

近似算法中的一个重要步骤是提出候选分裂点。 通常，特征的百分位数用于使候选者均匀地分布在数据上。 形式上，让多组 D k = {（x 1k，h 1），（x 2k，h 2）...（x nk，hn）}表示每个训练的第 k 个特征值和二阶梯度统计量实例。 我们可以定义秩函数 r k：R→[0，+∞）作为：

$$r_k(z) = \frac{1}{\sum_{(x,h)\in\mathcal{D}_k} h} \sum_{(x,h)\in\mathcal{D}_k, x<z} h, \tag{8}$$

表示特征值 k 小于 z 的实例的比例。 目标是找到候选分裂点{s k1，s k2，... s kl}，这样

$$\left|r_k(s_{k,j}) - r_k(s_{k,j+1})\right| < \epsilon, \quad s_{k1} = \min_i \mathbf{x}_{ik}, s_{kl} = \max_i \mathbf{x}_{ik}. \tag{9}$$

这里 e 是一个近似因子。 直观地说，这意味着大约有 1 /e 候选人点数。这里每个数据点由 h i 加权。 为了了解为什么 h i 代表重量，我们可以将 Eq（3）重写为

$$\sum_{i=1}^{n} \frac{1}{2} h_i (f_t(\mathbf{x}_i) - g_i/h_i)^2 + \Omega(f_t) + constant,$$

这是标签 gi / hi 和权重 hi 的加权平方损失。 对于大型数据集，找到满足条件的候选分割是非常重要的。 当每个实例具有相同的权重时，称为分位数草图的现有算法[14,24]解决了该问题。 但是，加权数据集不存在现有的分位数草图。 因此，大多数现有的近似算法要么对有可能失败的数据的随机子集进行排序，要么使用没有理论保证的启发式算法。 为了解决这个问题，我们引入了一种新颖的分布式加权分位数草图算法，该算法可以处理加权数据并具有可证明的理论保证 一般的想法是提出一种支持合并和修剪操作的数据结构，每个操作都被证明可以保持一定的准确度。 附录中给出了算法的详细描述以及证明。

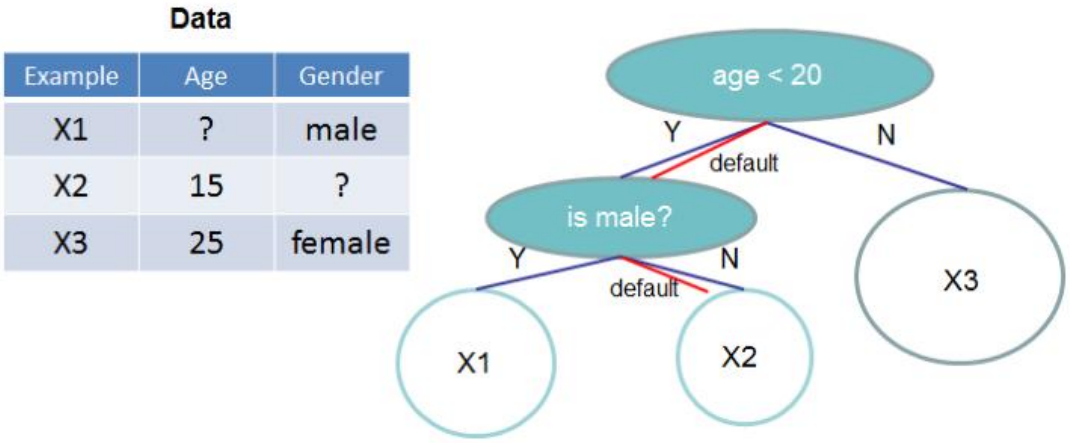



图 4：具有默认方向的树结构。 当缺少拆分所需的功能时，示例将被分类为默认方向。

## 3.4 稀疏性分裂发现稀疏性分裂发现

在许多现实问题中，输入 x 稀疏是很常见的。稀疏性有多种可能的原因：1）数据中存在缺失值; 2）统计中频繁的零项; 3）特征工程的工件，例如单热编码。使算法了解数据中的稀疏模式非常重要。为了做到这一点，我们建议在每个树节点中添加一个默认方向，如图 4 所示。当稀疏矩阵 x 中缺少一个值时，实例被分类为默认方向。每个分支中有两种默认方向选择。从数据中学习最佳默认方向。算法显示在 Alg 中。 3.关键的改进是只访问非缺失的条目 Ik。所提出的算法将非存在视为缺失值并且学习处理缺失值的最佳方向。当非存在对应

于用户指定值时，也可以通过将枚举限制为一致的解决方案来应用相同的算法。

据我们所知，大多数现有的树木学习算法要么仅针对密集数据进行优化，要么需要特定的过程来处理有限的情况，例如分类编码。 XGBoost 以统一的方式处理所有稀疏模式。更重要的是，我们的方法利用稀疏性使计算复杂度与输入中的非缺失条目的数量成线性关系。图 5 显示了对 Allstate-10K 数据集的稀疏性和初始实现的比较（第 6 节中给出的数据集的描述）。我们发现稀疏感知算法比天真版本快 50 倍。这证实了稀疏感知算法的重要性。

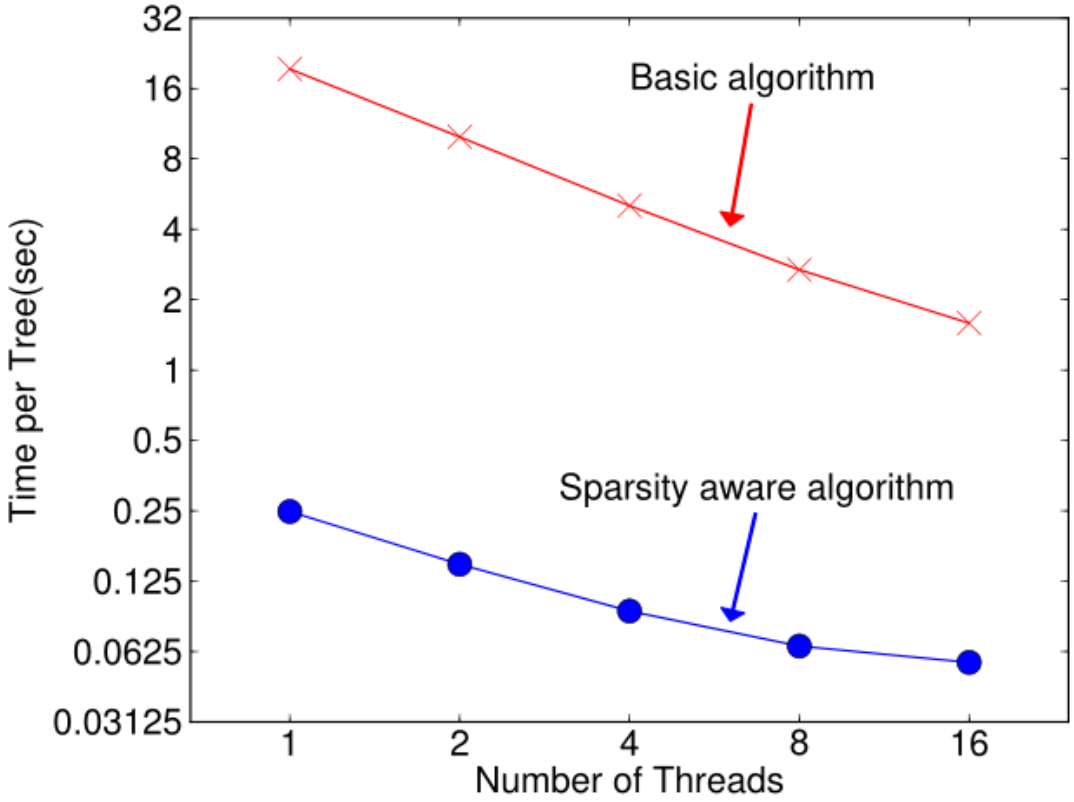



图 5：稀疏感知算法对 Allstate-10K 的影响。 数据集稀疏主要是由于单热编码。 稀疏感知算法比不考虑稀疏性的幼稚版本快 50 倍以上。

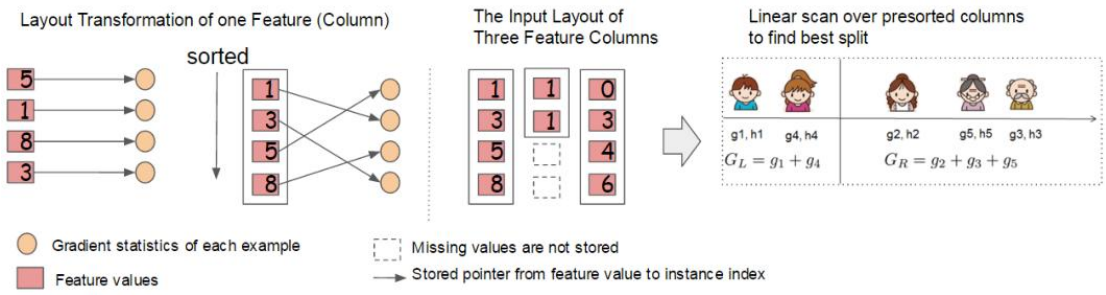



图 6：并行学习的块结构。 块中的每列按相应的特征值排序。 块中一列的线性扫描足以枚举所有分割点。

**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

# 4. 系统设计

## 4.1 用于并行学习的列块

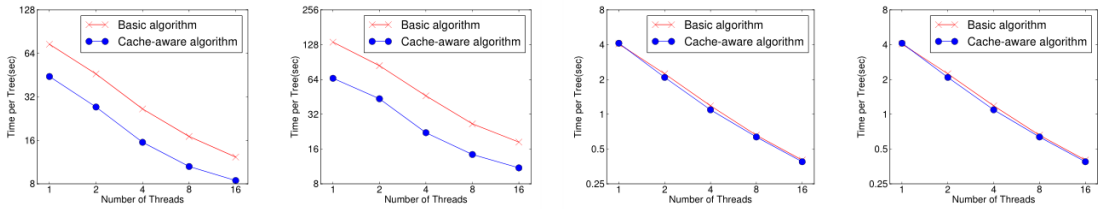
树学习中最耗时的部分是将数据按顺序排列。为了降低排序成本，我们建议将数据存储在内存单元中，我们称之为块。每个块中的数据以压缩列（CSC）格式存储，每列按相应的特征值排序。此输入数据布局仅需要在训练之前计算一次，并且可以在以后的迭代中重复使用。

在精确的贪心算法中，我们将整个数据集存储在一个块中，并通过对预先排序的条目进行临时扫描来运行拆分搜索算法。我们集体对所有叶子进行拆分

查找，因此对块进行一次扫描将收集所有叶子分支中的分割候选者的统计数据。图 6 显示了我们如何将数据集转换为格式并使用块结构找到最佳分割。在使用近似算法时，块结构也有帮助。在这种情况下可以使用多个块，

每个块对应于数据集中的行子集。不同的块可以跨机器分布，也可以在核外设置中存储在磁盘上。使用排序结构，分位数查找步骤将成为已排序列的线性扫描。这对于本地提议算法尤其有用，在每个分支中频繁生成候选者。直方图聚合中的二分搜索也变为线性时间合并样式算法。

收集每列的统计数据可以并行化，为我们提供了一种用于拆分查找的并行算法。重要的是，列块结构还支持列子采样，因为很容易在块中选择列的子集。




(a) Allstate 10M (b) Higgs 10M (c) Allstate 1M (d) Higgs 1M

图 7：精确贪婪算法中缓存感知预取的影响。 我们发现缓存未命中效应会影响大型数据集（1000 万个实例）的性能。 当数据集很大时，使用缓存感知预取可将性能提高两倍。

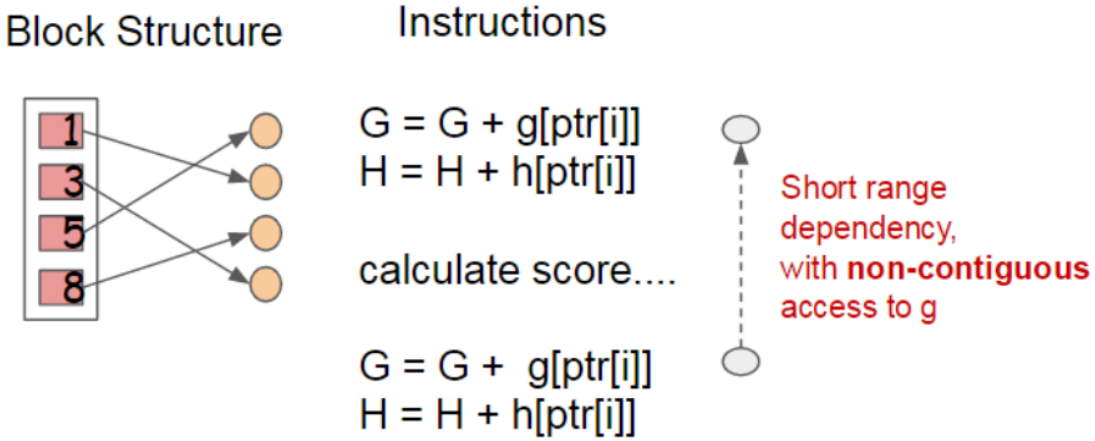



图 8：短距离数据依赖模式，可能由于缓存未命中而导致停顿。

时间复杂度分析设 d 为树的最大深度，K 为树的总数。对于精确的贪婪算法，原始 spase 感知算法的时间复杂度为 O（Kdkxk 0 logn）。这里我们使用 kxk 0 来表示训练数据中的非缺失条目的数量。

另一方面，块结构上的树提升仅花费 O（Kdkxk 0 + kxk 0 logn）。这里 O（kxk 0 logn）是可以摊销的一次性预处理成本。该分析表明块结构有助于节省额外的 logn 因子，这在 n 很大时很重要。对于近似算法，二元搜索的原算法的

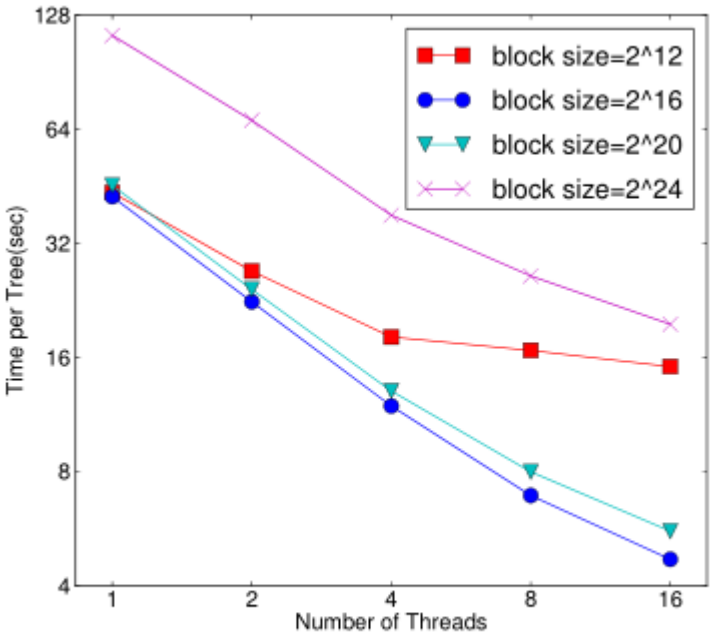
时间复杂度为 O（Kdkxk 0 logq）。这里 q 是数据集中候选提案的数量。虽然 q 通常介于 32 和 100 之间，但对数因子仍会引入开销。使用块结构，我们可以将时间减少到 O（Kdkxk 0 + kxk 0 logB），其中 B 是每个块中的最大行数。我们可以再次将额外的 logq 因子保存在计算中。

## 4.2 缓存感知访问

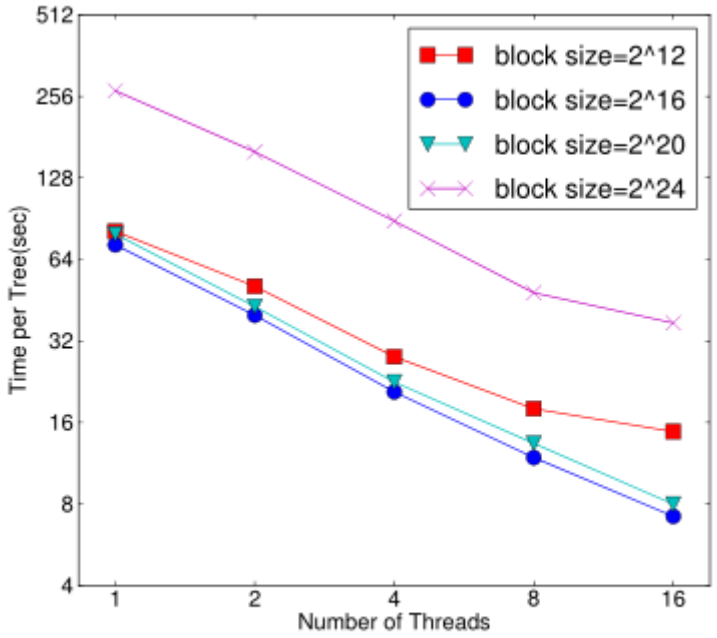
虽然所提出的块结构有助于优化分裂查找的计算复杂度，但是新算法需要通过行索引间接提取梯度统计，因为这些值是按特征的顺序访问的。 这是一种非连续的内存访问。 分裂枚举的简单实现在累积和非连续存储器获取操作之间引入了立即读/写依赖性（参见图 8）。 当梯度统计信息不适合 CPU 缓存并发生缓存未命中时，这会减慢拆分查找速度。

对于精确的贪婪算法，我们可以通过缓存感知预取算法来缓解这个问题。具体来说，我们在每个线程中分配一个内部缓冲区，获取梯度统计信息，然后以小批量方式执行累积。 此预取将 directread / write 依赖项更改为更长的依赖项，并有助于在行中的行数较大时减少运行时开销。 图 7 给出了 Higgs 和 All-state 数据集上缓存感知与非缓存感知算法的比较。 我们发现，当数据集很大时，精确贪婪算法的缓存感知实现的运行速度是天真版本的两倍。

对于近似算法，我们通过选择正确的块大小来解决问题。 我们将块大小定义为块中包含的最大示例数，因为这反映了梯度统计的高速缓存存储成本。 选择过小的块大小会导致每个线程的工作量很小，并导致低效的并行化。 另一方面，过大的块会导致高速缓存未命中，因为梯度统计信息不适合 CPU 高速缓存。 块大小的良好选择平衡了这两个因素。 我们在两个数据集上比较了块大小的各种选择。 结果如图 9 所示。该结果验证了我们的讨论，并表明每个块选择 216 个示例可以平衡缓存属性和并行化。




(a) Allstate 10M




(b) Higgs 10M

图 9：块大小在近似算法中的影响。 我们发现过小的块导致低效的并行化，而过大的块也会因缓存未命中而减慢训练速度。

Table 1: Comparison of major tree boosting systems.

| System | exact greedy | approximate global | approximate local | out-of-core | sparsity aware | parallel |
|---|---|---|---|---|---|---|
| **XGBoost** | yes | yes | yes | yes | yes | yes |
| pGBRT | no | no | yes | no | no | yes |
| Spark MLLib | no | yes | no | no | partially | yes |
| H2O | no | yes | no | no | partially | yes |
| scikit-learn | yes | no | no | no | no | no |
| R GBM | yes | no | no | no | partially | no |

### 4.3 用于核外计算的块

我们系统的一个目标是充分利用机器的资源来实现可扩展的学习。除处理器和内存外，利用磁盘空间处理不适合主内存的数据也很重要。为了实现核外计算，我们将数据分成多个块并将每个块存储在磁盘上。在计算过程中，使用独立的线程将块预取到主存储器缓冲区是很重要的，因此计算可以在磁盘读取的情况下进行。但是，这并不能完全解决问题，因为磁盘读取占用了大部分计算时间。减少开销并增加磁盘 IO 的吞吐量非常重要。我们主要使用两种技术来改进核外计算。

块压缩我们使用的第一种技术是块压缩。该块由列压缩，并在加载到主存储器时由独立线程动态解压缩。这有助于将解压缩中的一些计算与磁盘读取成本进行交换。我们使用通用压缩算法来压缩特征值。对于行索引，我们通过块的开始索引来减去行索引，并使用 16 位整数来存储每个偏移量。这需要每个块 216 个例子，这被证实是一个好的设置。在我们测试的大多数数据集中，我们实现了大约 26％到 29％的压缩比。

块分片第二种技术是以另一种方式将数据分片到多个磁盘上。将预取程器线程分配给每个磁盘并将数据提取到内存缓冲区中。然后，训练线程交替地从每个缓冲区读取数据。当有多个磁盘可用时，这有助于提高磁盘读取的吞吐量。

## 5. 相关作品

我们的系统实现了梯度增强[10]，它在功能空间中进行了加法优化。 梯度树增强已成功用于分类[12]，学习排名[5]，结构化预测[8]以及其他领域。 XGBoost 采用正则化模型来防止过度拟合。 这类似于以前关于正则化贪婪森林的工作[25]，但简化了并行化的目标和算法。 列采样是一种从 RandomForest

[4]借来的简单但有效的技术。 虽然稀疏感知学习在其他类型的模型中是必不可少的，例如线性模型[9]，但很少有关于树学习的工作以原则的方式考虑这个主题。 本文提出的算法是第一种处理各种稀疏模式的统一方法。

有几个关于并行树学习的现有工作[22,19]。大多数这些算法都属于本文所述的近似框架。值得注意的是，也可以按列[23]对数据进行分区，并应用精确的贪婪算法。我们的框架也支持这一点，并且可以使用诸如缓存感知预知之类的技术来使这种类型的算法受益。虽然大多数现有工作都集中在并行化的算法方面，但我们的工作在两个未开发的系统方向上进行了改进：核外计算和缓存感知学习。这为我们提供了有关如何联合优化系统和算法的见解，并提供了一个端到端系统，可以在非常有限的计算资源下处理大规模问题。我们还总结了表 1 中系统与现有开源实现之间的比较。

分位数摘要（无权重）是数据库社区中的经典问题[14,24]。然而，近似树提升算法揭示了一个更普遍的问题 - 在加权数据上找到分位数。据我们所知，本文提出的加权分位数草图是解决该问题的第一种方法。加权分位数摘要也不是特定于树学习的，并且可以在将来有益于数据科学和机器学习中的其他应用。

# 6 端到端的评估

## 6.1 系统实施

我们将 XGBoost 实现为开源包 5。 该包装是便携式和可重复使用的。 它支持各种加权分类和秩目标函数，以及用户定义的目标函数。 它以流行的语言提供，例如 python，R，Julia，并且与语言本机数据科学管道（如 scikit-learn）自然集成。 分布式版本构建在 rabit 库 6 之上，用于 allreduce。 XGBoost 的可移植性使其可用于许多生态系统，而不仅仅是绑定到特定平台。 分布式 XGBoost 在 Hadoop，MPI Sun Grid 引擎上本机运行。 最近，我们还在 jvm bigdata 堆栈（如 Flink 和 Spark）上启用了分布式 XGBoost。 分布式版本也已集成到阿里巴巴的云平台天池 7 中。 我们相信未来会有更多的整合。

## 6.2 数据集和设置

我们在实验中使用了四个数据集。 表 2 中给出了这些数据集的摘要。在一些实验中，由于基线较慢，我们使用随机选择的数据子集，或者演示具有不同数据集大小的算法的性能。 在这些情况下，我们使用后缀来表示大小。 例如，Allstate-10K 表示具有 10K 实例的 Allstate 数据集的子集。 我们使用的第一个数据集是 Allstate 保险索赔数据集 8。 任务是根据不同的风险因素预测保险索赔的可能性和成本。 在实验中，我们将任务简化为仅预测保险索赔的可能性。 此数据集用于评估稀疏性感知算法在 Sec 中的影响。3.4。 此数据中的大多数稀疏功能都来自单热编码。 我们随机选择 10M 实例作为训练集，并将其余部分用作评估集。

第二个数据集是来自高能物理学的 Higgs 玻色子数据集 9。数据是使用物理事件的蒙特卡罗模拟生成的。它包含 21 个运动学特性，由加速器中的粒子探测器测量。它还包含七个额外的粒子派生物理量。任务是分类事件是否与希格斯玻色子相对应。我们随机选择 10M 实例作为训练集，并将其余部分用作评估集。第三个数据集是 Yahoo!学习排名挑战数据集[6]，这是学习排名算法最常用的基准之一。数据集包含 20K Web 搜索查询，每个查询对应于大约 22 个文档的列表。任务是根据查询的相关性对文档进行排名。我们在实验中使用官方列车测试分组。

最后一个数据集是 criteo terabyte click log dataset10。我们使用此数据集来评估系统在核外和分布式设置中的扩展属性。该数据包含 13 个整数功能和 26 个用户，项目和广告商信息的 ID 功能。由于基于树的模型更好地处理连续特征，我们通过计算前十天的平均 CTR 和 ID 特征的统计数据来预处理数据，在接下来的十天内用相应的计数统计数据替换 ID 特征为了训练。预处理后的训练集包含 17 个具有 67 个特征的实例（13 个整数，26 个平均点击率统计和 26 个计数）。整个数据集的 LibSVM 格式超过 1TB。

我们将前三个数据集用于单机平行设置，并将最后一个数据集用于分布式和核外设置。所有单机实验均在戴尔 PowerEdge R420 上进行，配备两个八核 Intel Xeon（E5-2470）（2.3GHz）和 64GB 内存。如果未指定，则使用机器中的所有可用核心运行所有实验。分布式和核外实验的机器设置将在相应的部分中描述。在所有实验中，我们使用最大深度等于 8 的共同设置来提升树，收缩等于

0.1 并且除非明确指定，否则不进行列子采样。当我们使用其他最大深度设置时，我们可以找到类似的结果。

## 6.3 分类

在本节中，我们使用 Higgs-1M 数据上的精确贪婪算法，通过将其与其他两种常用的精确贪婪树提升实现进行比较，评估 XGBoost 在单台机器上的性能。由于 scikit-learn 只处理非稀疏输入，我们选择密集的 Higgs 数据集进行公平比较。我们使用 1M 子集在合理的时间内运行 scikit-learn。在比较的方法中，R 的 GBM 使用贪婪的方法，只扩展树的一个分支，这使得它更快但可能导致更低的准确性，而 scikit-learn 和 XGBoost 都学习完整的树。结果显示在表 3 中.XGBoost 和 scikit-learn 都比 R 的 GBM 提供更好的性能，而 XGBoost 的运行速度比 scikit-learn 快 10 倍。在此实验中，我们还发现列子样本的性能略差于使用所有功能。这可能是因为此数据集中的重要特征很少，我们可以从所有功能中贪婪地选择。

## 6.4 学习排名

我们接下来评估 XGBoost 在学习排名问题上的表现。 我们比较 pGBRT [22]，这是此任务中最好的先前发布的系统。 XGBoost 运行精确的贪心算法，而 pGBRT 仅支持近似算法。 结果显示在表 4 和图 10 中。我们发现 XGBoost 运行得更快。 有趣的是，二次采样列不仅可以缩短运行时间，而且可以为这个问题提供更高的性能。 这可能是由于子采样有助于防止过度拟合，这是许多用户所观察到的。

## 6.5 核心外实验

我们还在 criteo 数据的 out-of-core 设置中评估我们的系统。 我们在一台 AWS c3.8xlarge 机器上进行了实验（32 个 vcores，两个 320 GB SSD，60 GB RAM）。 结果显示在图 11 中。我们可以发现压缩有助于将计算速度提高三倍，并且分成两个磁盘进一步提供 2 倍的加速。 对于此类实验，使用非常大的数据集来排空系统文件缓存以实现真正的核外设置非常重要。 这确实是我们的设

置。 当系统用完文件缓存时，我们可以观察到转换点。 请注意，最终方法的转换不那么引人注目。 这是由于更大的磁盘吞吐量和更好的计算资源利用率。我们的最终方法能够在一台机器上处理 17 亿个示例。

### 6.6 分布式实验

最后，我们在分布式设置中评估系统。我们在 EC2 上使用 m3.2xlarge 机器建立了一个 YARN 集群，这是集群的一个非常常见的选择。每台机器包含 8 个虚拟内核，30GB 内存和两个 80GB SSD 本地磁盘。数据集存储在 AWS S3 而不是 HDFS 上，以避免购买持久存储。我们首先将我们的系统与两个生产级分布式系统进行比较：Spark MLLib [18]和 H2O 11.我们使用 32 m3.2xlarge 机器测试性能具有各种输入大小的系统。两个基线系统都是内存分析框架，需要将数据存储在 RAM 中，而 XGBoost 可以在内存不足时切换到核外设置。结果如图 12 所示。我们可以发现 XGBoost 的运行速度比基线系统快。更重要的是，它能够利用核外计算的优势，并在给定有限的计算资源的情况下平稳扩展到所有 17 亿个示例。基线系统仅能够使用给定资源处理数据的子集。该实验显示了将所有系统改进结合在一起并解决实际规模问题的优势。我们还通过改变机器数量来评估 XGBoost 的缩放属性。结果显示在图 13 中。随着我们添加更多机器，我们可以发现 XGBoost 的性能随线性变化。重要的是，XGBoost 只需要四台机器即可处理整个 17 亿个数据。这表明系统有可能处理更大的数据。

## 7. 总结

在本文中，我们描述了我们在构建 XGBoost 时学到的经验教训，XGBoost 是一个可扩展的树推进系统，被数据科学家广泛使用，并提供了许多问题的最新结果。我们提出了一种用于处理稀疏数据的新型稀疏感知算法和用于近似学习的理论上合理的加权分位数草图。我们的经验表明，缓存访问模式，数据压缩和分片是构建可扩展的端到端系统以实现树提升的基本要素。这些课程也可以应用于其他机器学习系统。通过结合这些见解，XGBoost 能够使用最少量的资源解决实际规模问题。

# 8. 致谢

我们要感谢 Tyler B. Johnson，Marco Tulio Ribeiro，Sameer Singh，Arvind Krishnamurthy 提供的宝贵意见。我们也衷心感谢 Tong He，Bing Xu，Michael Benesty，Yuan Tang，刘洪亮，Qiang Kou，Nan Zhu 以及 XGBoost 社区的所有其他贡献者。这项工作部分由 ONR（PECASE）N000141010672，NSF IIS 1258741 支持

以及由 MARCO 和 DARPA 赞助的 TerraSwarm 研究中心。