Raha 无配置错误检测系统

09/17/2024-09/22/2024

摘要

检测错误值是数据清理的关键一步。错误检测算法通常要求用户以规则或统计参数的形式提供输入配置。然而，为每个新数据集提供一套完整而正确的配置并不容易，因为用户必须预先了解数据集和错误检测算法。在本文中，我们介绍了一种新型免配置错误检测系统 Raha。通过为涵盖各种类型数据错误的错误检测算法生成数量有限的配置，我们可以为每个元组值生成富有表现力的特征向量。利用这些特征向量，我们提出了一种新颖的采样和分类方案，可以有效地选择最具代表性的值进行训练。此外，我们的系统还能利用历史数据过滤掉不相关的错误检测算法和配置。在我们的实验中，Raha 在每个数据集上的标注图元不超过 20 个的情况下，表现优于最先进的错误检测技术。

1 引言

数据科学家认为数据清理是最耗时的任务之一[20]。数据清洗包括连续的错误检测和数据修复任务 [14]。错误检测识别偏离（通常不可用的）基本真实值的数据值。错误检测策略可分为定量方法（如检测异常值）[36] 和定性方法（如检测违反完整性约束的情况）[18]。数据修复则是为检测到的错误数据确定正确的值[17, 18, 39]。在实践中，错误检测和数据修复通常是迭代进行的，而不是按顺序进行的。因此，它们在效果上会相互影响 [39]。这项工作的重点是错误检测。只运行一种错误检测算法往往会导致召回率较低[1]，因此建议运行多种方法来涵盖各种类型的错误。不过，这需要用户进行一系列耗时且不繁琐的步骤：

(1) 算法选择。面对各种类型的错误，用户需要选择一种或多种算法来运行。用户通常根据以往的经验选择算法，如果没有可用的算法，则随机选择。尝试所有这些算法非常麻烦，而且它们的联合聚合往往会导致精度不高[1]。另一种方法是根据每个检测器的精度依次运行[1]，这需要用户参与计算精度。

(2) 算法配置。用户还需要根据给定数据集的特点配置所选的错误检测算法。根据错误检测算法的不同，配置可能包括统计参数，如预期平均值和标准偏差；模式，如 “dd.mm.yyyy”；规则说明；或提供参考数据集的链接，如 DBpedia [7]。必须选择适当的算法及其适当的配

置，这就为确定最佳错误检测策略提供了巨大的搜索空间。

(3) 结果验证。即使选择了算法并手动配置了相应的参数，用户有时仍需要参与结果验证，以便对算法进行微调并提高其准确性。

要完成这些任务，用户必须了解数据集领域、其质量问题以及错误检测算法本身。

有人尝试使用基于机器学习的方法，例如 ActiveClean [32] 和元数据驱动方法 [45]。这些方法试图通过学习用户标注为脏或干净的数据点来克服其中的一些挑战。然而，它们普遍存在两个问题。首先，设计的特征向量要么表现力不够，无法捕捉各种错误类型，要么必须手动调整。ActiveClean 使用 TF-IDF 特征[32]，它只能揭示单词层面的错误。元数据驱动方法利用配置检测工具的输出[45]，需要用户配置工具。其次，所需的训练数据相当大，而且随着数据集的大小而增加。例如，元数据驱动方法需要多达 1%的数据集作为训练数据[45]。这对于大数据来说可能相当大。

在本文中，我们提出了一种新的半监督错误检测方法，它不需要用户提供配置。简而言之，我们的方法为每个数据单元分配一个特征向量。向量的每个分量代表特定错误检测算法的特定配置的二进制输出。然后，我们根据特征向量对每列数据的单元格进行聚类，并要求用户每次只标记一个聚类。

特征表示。为了描述数据错误的特征，特征向量对传统错误检测技术的四大系列，即离群点检测、模式违规检测、规则违规检测和知识库违规检测算法的输出进行了编码[1]。对于每个算法系列，我们都会自动生成一套原则性的、有限的可能配置，例如离群点检测算法的不同阈值，以及相应的违规检测算法的不同模式、规则和参考数据集。此外，我们还限制了模式和依赖关系的指数数量，并根据理论边界和最佳实践启发式方法对统计参数的连续空间进行离散化处理。我们使用每个配置的结果来生成每个数据单元的特征向量。由于该向量的每个分量并不代表特定数据集的有效配置，因此通过投票或其他集合方法汇总结果可能会产生误导。相反，我们使用这些信息丰富的特征向量来计算数据单元之间的相似性，从而对它们进行聚类。

用户参与。为了限制用户参与，我们设计了一种聚类和学习方案，以实现有效的标记和标签传播。由于每个聚类包含具有相似特征向量的值，我们要求用户只为每个聚类的一个实例贴标签。因此，生成簇的数量就决定了所需的用户交互次数。我们认为，每个被归类为“脏 ”

的簇都代表了一种隐含的数据误差类型。然后，最先进的分类器可以使用用户标签和噪声标签将所有数据单元划分为 “脏 ”或 “干净”。表 1 显示了 Raha 与现有方法相比在用户参与方面的优势。其他方法需要用户参与多个步骤，而 Raha 只需要少量用户标签。

过滤特征。虽然我们的方法只需要极少量的用户参与，而且无需任何配置，但由于特征数量庞大，因此在特征生成过程中会消耗更多的运行时间。尽管如此，如果可以使用过去的净化数据集，我们生成特征的运行时间还是可以大大缩短的。我们提出的方法可以过滤掉无关的错误检测算法和配置，这些算法和配置在过去类似的脏数据集上对整体性能没有贡献。

贡献。我们的贡献如下：

- 我们提出了无配置错误检测系统，命名为 Raha（第 3 节），它不需要任何用户提供的数据约束或参数。

- 我们为错误检测任务提出了一种新颖的特征向量（第 4 节），其中包含了各种错误检测策略的信号。

- 我们提出了一种基于聚类的采样方法（第 4 节），大大减少了用户参与标签制作的程度。利用基于聚类的标签传播方法，我们获得了额外的噪声标签，从而提高了的 Raha 的分类性能。

- 我们提出了一种策略过滤方法（第 5 节），可根据过去的净化数据集对算法配置进行修剪。这种方法将运行时间缩短了一个数量级以上，但效率略有降低。

- 我们进行了大量实验（第 6 节），对 Raha 的有效性、效率和用户参与度进行了评估。我们将 Raha 与 4 种独立错误检测算法和 3 种错误检测聚合器进行了比较。在仅使用 20 个标记图元的情况下，Raha 的性能优于其他任何现有解决方案。

2 基础

我们首先定义数据错误并对其类型进行分类。 然后，我们回顾文献中用于 Raha 的一般错误检测算法系列。最后，我们正式定义无配置错误检测问题。

2.1 数据错误

数据误差是指数据集中与实际地面实况不符的数据值。更正式地说，让 $d = \{t_1,t_2,...,t_{|d|}\}$ 是一个大小为 $|d|$ 的关系数据集，其中每个 t_i 表示一个元组。设 $A =\{a_1,a_2,...,a_{|A|}\}$ 是数据集 d 的模式，属性为 $|A|$。让 $d[i, j]$ 表示数据集 d 的 t_i 元组中的数据单元和模式 A 的 a_j 属性。每一个

数据单元 d[i, j] 如果与地面实况 $d^*[i, j]$ 中的相应单元不同，就会被视为数据错误。例如，在表 2 中的脏数据集 d 中，d[3,2] = ∅、d[4,2] = ∅、d[5,2] = 123 和 d[6, 2] = Shire 这几个数据单元格根据基础数据 d^*是错误的。

数据错误可分为语法错误和语义错误。语法错误是指那些不符合正确值的结构或领域的值。语义错误是指虽然语法正确，但出现在错误上下文中的值。例如，数字数据值 d[5, 2] = 123 就是一个语法错误，因为它不符合任何王国名称的语法。数据值 d[6, 2] = Shire 是一个语义错误，因为虽然它是《指环王》中一个语法正确的王国，但该数据单元格的正确值是另一个王国，即 Rohan。

2.2 误差检测算法

原则上，我们的系统可以利用任何以连续数值参数或离散标称参数为输入的误差检测算法。任何其他不需要任何此类参数的错误检测算法，即黑盒子，只能使用单一配置。现有的错误检测算法[17, 18, 29, 36]可分为四类：离群点检测、模式违规检测、规则违规检测和知识库违规检测[1]。 前两种主要针对语法数据错误，后两种针对语义错误。请注意，Raha 并不局限于这些类别。

离群点检测算法[36]根据数据值是否符合列内数值的一般分布来评估数据值的正确性。直方图建模和高斯建模[36]是两种基本的离群点检测算法，它们分别利用了数据值的出现率和大小。直方图建模策略根据特定数据列中数据值的频率建立直方图分布。该策略将来自稀有箱的数据单元标记为错误，即归一化项频率小于阈值 $\theta_{tf} \in (0,1)$的数据单元。高斯建模策略根据特定列中数值的大小建立高斯分布。该策略将那些与均值的归一化距离大于阈值 $\theta_{dist} \in (0, \infty)$ 的数值单元标记为错误。

例如，对于表 2 中的数据集 d，通过设置 $\theta_{tf} = 2/6$，两个基于直方图的离群点检测算法在属性 Kingdom 上的输出结果为 s_{o1}= {d[1,2],d[2,2],d[5,2],d[6,2]} ；通过设置 $\theta_{tf} = 3/6$，输出结果为 s_{o2}= {d[1,2],d[2,2],d[3,2],d[4,2],d[5,2],d[6,2]}。

模式违规检测算法[29]根据与预先确定的数据模式的兼容性来评估数据值的正确性。模式违规检测算法会标记出与特定模式不匹配的数据值。

例如，对于表 2 中的数据集 d，通过将预定义模式设置为字母值，两种针对属性 Kingdom 的模式违规检测算法的输出结果为 s_{p1} = {d[3,2],d[4,2],d[5,2]} ；通过将预定义模式设置为非

空值，两种针对属性 Kingdom 的模式违规检测算法的输出结果为 s_{p2} = {d[3, 2], d[4, 2]} 。

违规检测算法 [18] 根据数据值是否符合完整性约束来评估数据值的正确性。由于离群值和模式违规检测算法已隐含了值范围和长度等单列规则，因此我们在此只包含检查列间依赖关系的规则违规检测策略。特别是，我们将重点放在功能依赖（FD）形式的规则上。其他类型的规则，如条件 FD，也可以用同样的方法来考虑。

例如，在表 2 中的数据集 d 上，通过检查 FD Lord → Kingdom，两个针对属性 Kingdom 的规则违反检测算法的输出结果将是 s_{r1} = {}，而通过检查 FD Kingdom → Lord，输出结果将是 s_{r2} = {d[3,2],d[4,2]}。（译注：此时检测的是数据依赖：例如 s_{r1},必须先有 Lord 再有 Kingdom）

知识库违规检测算法（如 Katara [17]）通过与知识库（如 DBpedia [7]）中的数据进行交叉检查，来评估数据值的正确性。知识库中的数据通常以实体关系的形式存储，如 City isCapital of Country。在这里，城市和国家是实体类型，而 isCapitalOf 是一种关系。算法会尝试将关系的每一方与数据集中的不同数据列进行匹配。如果有两个数据列与关系的两边都匹配（如城市和国家），算法就会在匹配的数据列中标记出与知识库中实体关系相冲突的数据值。例如，该算法会将 isCapitalOf 关系匹配到两列，并标记违规元组中的数据单元格，如那些错误地将柏林作为法国首都的单元格。因此，知识库违规检测算法也能识别违反列间依赖关系的数据错误。

对于表 2 中的数据集 d，通过将实体关系设置为 Lord isKingOf Kingdom，两个针对王国属性的知识库违规检测器的输出结果为 s_{k1} = {d[3,2],d[4,2],d[5,2],d[6,2]}；通过将实体关系设置为 City isCapitalOf Country，输出结果为 s_{k2} = {}。

2.3 问题陈述

给定一个脏数据集 d 作为输入,一组需要配置的可用错误检测算法 B = {b1,b2, ...,$b_{|B|}$ },以及一个具有标签预算 θ_{labels} 的用户来注释元组,我们希望识别 d 中的所有错误值。具体来说,问题是自动配置错误检测算法 B 并将其输出汇总为一组标记数据单元 ,即 O = {(d[i, j], l) | 1⩽i⩽|d|, 1⩽j⩽|A|, l∈{dirty, clean}},其中,|d|是数据集 d 的行数,|A|是数据集 d 的列数。

为了将错误检测问题转化为分类任务,我们必须解决三个子问题。首先,我们需要能够表示句法和语义数据错误的特征向量。其次,我们需要一种能够处理正确值和错误值之间的类不平衡比

的采样方法。第三,在运行所有策略之前,最好先过滤掉不相关的错误检测算法和配置,以减少运行时间。为了简化本文的其余部分,我们将算法和配置的每种组合视为一种不同的错误检测策略。

定义 1 (错误检测策略)。设 B = {b1, b2, ...,$b_{|B|}$ }为错误检测算法集合。设 G_b ={g_1,g_2, ...,$g_{|G_b|}$}是错误检测算法 b 的不同配置的有限/无限空间集合。我们将错误检测策略定义为错误检测算法 b 和配置 g∈G_b 的组合。因此,错误检测策略集合是所有可能错误检测策略的子集 $S \subseteq B \times G_b = \{(b,g) \mid b \in B, g \in G_b\}$。

3 Raha 概述

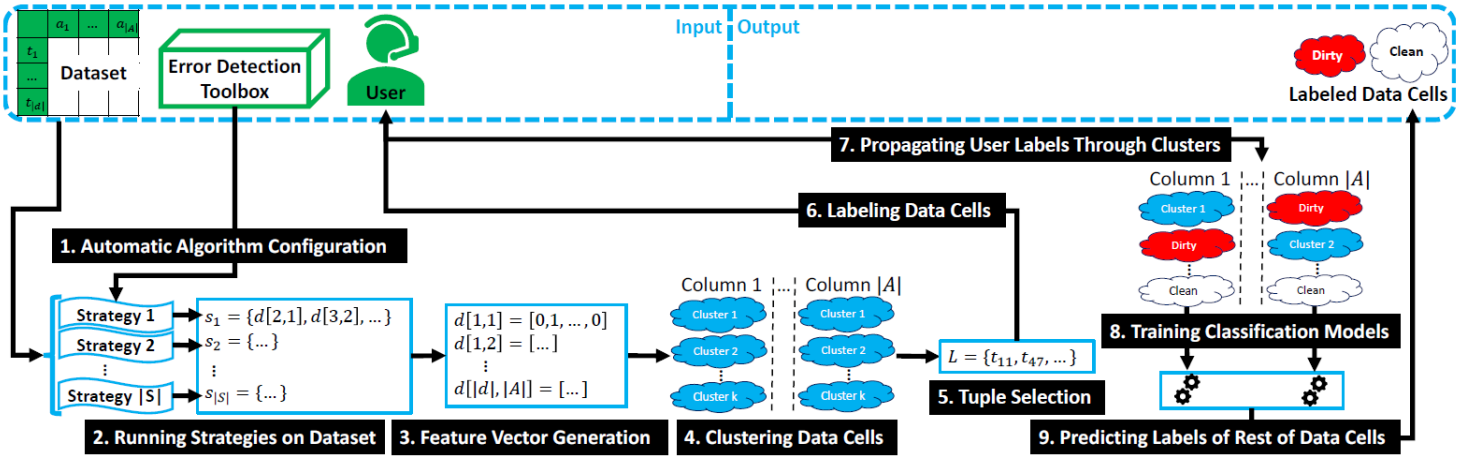


Figure 1: The workflow of Raha.

Table 3: Clustering data cells of column *Kingdom*.

ID	Kingdom	s_{o1}	s_{o2}	s_{p1}	s_{p2}	s_{r1}	s_{r2}	s_{k1}	s_{k2}
1	Minas Tirith	1	1	0	0	0	0	0	0
2	Mordor	1	1	0	0	0	0	0	0
3	∅	0	1	1	1	0	1	1	0
4	∅	0	1	1	1	0	1	1	0
5	123	1	1	1	0	0	0	1	0
6	Shire	1	1	0	0	0	0	1	0

我们在图 1 中使用第 2 节中的运行示例解释了 Raha 工作流程的每个步骤。给定脏数据集、错误检测工具箱和用户反馈作为输入，Raha 将每个数据值分类为干净或脏。

第 1 步：自动算法配置。Raha 会对每种现有算法进行系统配置，生成一组错误检测策略。在第 2 节的示例中，我们为每一类错误检测算法引入了两种配置。这将产生 |S| = |B×G_b | =

4×2 = 8 种策略： $S = \{s_{o1}, s_{o2}, s_{p1}, s_{p2}, s_{r1}, s_{r2}, s_{k1}, s_{k2}\}$。我们将在第 4.1 节详细介绍这一步骤。

第 2 步：运行错误检测策略。每个策略将一组数据单元标记为数据错误。如第 2 节所述，离群点检测策略 s_{o1} 在属性 Kingdom 上标记数据单元 s_{o1} = {d[1,2],d[2,2],d[5,2],d[6,2]}。我们将在第 4.2 节详细介绍这一步骤。

第 3 步：生成特征向量。Raha 通过收集所有错误检测策略的输出，为每个数据单元生成一个特征向量。数据单元特征向量中的每个元素都是二进制值，表示特定策略是否将该数据单元标记为数据错误。如表 3 所示，在我们的运行示例中，数据单元 d[1, 2] = Minas Tirith 的特征向量为 [1, 1, 0, 0, 0, 0, 0]。我们将在第 4.2 节详细介绍这一步骤。

第 4 步：对数据单元格进行聚类。Raha 根据特征向量的相似性，将每列的单元格分成不同的簇。如表 3 所示，Kingdom 列中的数据单元格被分为 3 组：黑色组、蓝色组和紫色组。我们将在第 4.3 节详细介绍这一步骤。

第 5 步：采样图元。Raha 会抽取一组图元，由用户进行标注。由于每列数据都有一组簇，理想情况下，采样的图元应尽可能多地覆盖所有数据列中未标记的簇。在上述例子中，假设 Raha 采样了图元 t_1 和 t_3。这两个图元覆盖了 Kingdom 数据列上的黑色和蓝色集群。理想情况下，这两个图元应覆盖其他数据列（即 Lord 列）中的两个不同群集。我们将在第 4.3 节详细介绍这一步骤。

第 6 步：标注数据单元格。拉哈要求用户将采样图元的数据单元格标记为脏单元格或干净单元格。在示例中，用户将数据单元格 d[1,1]、d[1,2] 和 d[3,1] 标为干净，将 d[3, 2] 标为脏。我们将在附录 C 中详细介绍这一步骤。

第 7 步：通过聚类传播用户标签。数据单元格 d[2, 2]将被标记为干净，因为它与数据单元格 d[1, 2]在同一个聚类中。同样，数据单元格 d[4, 2]将被标记为脏。这些标签是有噪声的，因为它们未经用户验证。我们将在第 4.4 节详细介绍这一步骤。

第 8 步：训练分类模型。Raha 根据数据单元的特征向量和传播的标签(即用户标签和含噪标签)对每个数据列训练分类模型。我们将在第 4.4 节详细介绍这一步骤。

第 9 步：预测其余数据单元格的标签。训练后的分类模型用于预测其他未标记集群中其余数据单元格的标签。在上述示例中,训练后的数据列 Kingdom 的分类模型预测未标记的紫色集群

中数据单元格 d[5, 2]和 d[6, 2]的标签为脏。我们将在第 4.4 节中详细介绍此步骤。

4 错误检测引擎

Algorithm 1: Raha(d, B, θ_{labels}).

Input: dataset d, set of error detection algorithms B, labeling budget θ_{labels}.
Output: set of labeled data cells O.

1 $S \leftarrow$ generate strategies by automatically configuring algorithms $b \in B$;
2 $V \leftarrow$ generate features by running strategies $s \in S$ on dataset d;
3 $k \leftarrow 2$; // number of clusters per data column
4 $L \leftarrow \{\}$; // set of user labeled tuples
5 **while** $|L| < \theta_{labels}$ **do**
6 **for** *each data column* $j \in [1, |A|]$ **do**
7 $\phi_j \leftarrow$ cluster data cells of column j into k clusters;
8 $t^* \leftarrow$ draw a tuple with probability proportional to $P(t)$;
9 ask the user to label tuple t^*;
10 $L \leftarrow L \bigcup \{t^*\}$;
11 $k \leftarrow k + 1$;
12 $O \leftarrow \{\}$; // set of labeled data cells
13 **for** *each data column* $j \in [1, |A|]$ **do**
14 $L'_j \leftarrow$ propagate the user labels through the clusters ϕ_j;
15 $m_j \leftarrow$ train a classification model with feature vectors V_j and labels L'_j;
16 $O \leftarrow O \bigcup$ apply the classification model m_j on rest of feature vectors V_j;

算法 1 显示了 Raha 的主要步骤。它首先配置错误检测算法（第 1 行），然后生成特征向量（第 2 行）。接下来，Raha 对数据单元进行聚类，并抽取元组进行用户标签（第 3-11 行）。最后，它通过聚类传播用户标签，并训练一组分类器来预测所有单元格的标签（第 12-16 行）。

4.1 自动算法配置

如第 2 节所述，我们可以识别具有数值参数和名义参数的算法。对于具有数值参数的算法，如离群点检测算法，我们对参数的连续范围进行量化。对于具有无限名义参数的算法，如模式、规则和知识库违规检测算法，我们会确定启发式方法来有效限制参数空间。当然，生成的错误检测策略中只有一个子集可以有效地标记给定数据集上的数据错误，而且其中大部分可能并不精确。尽管如此，只要每个策略使用相同的逻辑标记数据单元，拉哈就可以使用策略的输出作为比较数据单元的相似性概念。

离群点检测策略。直方图建模离群点检测策略 $s_{\theta tf}$ 要求最小项频阈值 $\theta_{tf} \in (0, 1)$。Raha 通过设置阈值 $\theta_{tf} \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$，生成了 9 个直方图建模离群点检测策略。需要注意的是，在实践中，我们发现更细的粒度值（如 $\theta_{tf} = 0.15$）与更粗粒度的阈值选择相比，并不能创建更独特的离群点检测策略。

$$s_{\theta_{\text{tf}}}(d[i,j]) = \begin{cases} 1, & \text{iff} \quad \frac{TF(d[i,j])}{\sum_{i'=1}^{|d|} TF(d[i',j])} < \theta_{\text{tf}}; \\ 0, & \text{otherwise.} \end{cases}$$

高斯建模离群点检测策略 $s_{\theta dist}$ 要求距离阈值 $\theta_{dist} \in (0, \infty)$。Raha 根据 68-95-99.7 规则[44]，通过自动设置阈值 $\theta_{dist} \in \{1，1.3，1.5，1.7，2，2.3，2.5，2.7，3\}$，生成了 9 种高斯建模离群点检测策略。

$$s_{\theta_{\text{dist}}}(d[i,j]) = \begin{cases} 1, & \text{iff} \quad \frac{|d[i,j]-\mu_j|}{\sigma_j} > \theta_{\text{dist}}; \\ 0, & \text{otherwise.} \end{cases}$$

模式违规检测策略。理论上，我们可以使用日期和 URL 等特定域数据模式来检测模式违规。然而，由于所有可能数据模式的集合是无限的，我们需要一种合理的方法来限制模式违规检测策略的集合，而无需对数据域做任何特定假设。为此，我们采用了字符袋表示法[41]，这是一种用于编码所有可能数据模式（即字符组合）的通用表示法。这种表示法还能对数据值的长度和类型进行编码，因为它还能显示数据值中出现了哪些和多少个不同的字符。

具体来说，我们生成了一组字符检查策略 s_{ch}，用于检查数据单元中是否存在每个字符 ch。对于数据列 j 中所有字符集合中的每个字符 ch，如果数据单元格 d[i, j]中包含字符 ch，则 s_{ch} 会将该单元格标记为数据错误。例如，假设我们有一个日期列，其模式为 “dd.mm.yyyy”。字符袋表示法会自动生成策略 s-，并暴露出错误分隔符的出现，例如 “16-11-1990 ”中的字符 “-”。

总的来说，我们的系统中有 $\bigcup_{j=1}^{|A|} |\Sigma j|$ 种模式违规检测策略，其中 |A| 是数据列的数量，Σ j 是数据列 j 中出现的所有不同字符的集合。 规则违规检测策略。为了合理地限制所有可能的 FD 的指数空间，我们将 FD 的范围限制在其左侧只有一个属性的范围内。这样，我们还能避免因考虑大量数据列集而产生的潜在意外依赖性。正如文献[34]所述，大多数有趣的 FD

只涉及少数几个属性。由于我们事先并不知道哪些 FD 是有用的，因此我们将所有列对都视为潜在的 FD。通过这种方法，我们还涵盖了用户未知的部分 FD 关系。对于每一对属性 $\forall a, a' \in A$, 策略 $s_{a \to a'}$ 标记所有违反 FD $a \to a'$ 的数据单元格 d[i,j]，这与之前的工作[1, 15]是一致的。

知识库冲突检测策略。Raha 在数据集和知识库(如 DBpedia[7])之间建立联系。对于知识库中的每个关系 r,策略 s_r 标记所有违反(矛盾)该关系的数据单元 d[i, j]。

总体而言,我们的知识库冲突检测策略的数量与 DBpedia 知识库[7]中的关系数量相同,即 2064 个策略。

4.2 生成特征向量

我们将每个数据单元映射为一个特征向量，该向量由错误检测策略的输出组成。有了一组错误检测策略 $S = \{s_1, s_2, ..., s_{|S|}\}$，我们在数据集 d 上运行每个策略 $s \in S$。正式来说,我们将这些信息存储为 $s(d[i, j]) = 1$, 如果 s 将 d[i, j]标记为数据错误;否则为 0。

数据单元 d[i, j]的特征向量是错误检测策略 $s \in S$ 对该数据单元的所有输出的向量。

形式上, $\mathbf{v(d[i, j]) = [s(d[i, j]) \mid \forall s \in S]. (1)}$

因此,特定数据列 j 内数据单元的特征向量集 $\mathbf{v(d[i, j])}$**为** $\mathbf{V_j = \{v(d[i, j]) \mid 1 \leqslant i \leqslant |d|\}}$**。(2)**

我们对每个数据列 Vj 的特征向量进行后处理,以去除该数据列所有数据单元中均恒定的非信息特征。

4.3 基于聚类的采样

使用描述的特征向量, Raha 学习如何借助用户标签将数据单元分类为干净或脏。我们为每一列训练一个分类器,因为我们想要识别单元级别的错误,而不是元组级别的错误。因此,我们需要每一列的标签。在本节中,我们将描述我们的解决方案,通过从每一列中采样包含有希望单元值的元组来减少标记工作量。

数据单元的聚类。利用我们表达力强的特征向量,我们可以根据聚类假设[13]来增加每个数据列的标记数据单元的数量,该假设指出,如果两个数据点属于同一个聚类,则它们很可能具有相同的类标签[13]。因此,如果我们对数据单元进行聚类,则每个聚类中的单元很可能具有相同的脏标签或干净标签。

出于每列使用一个分类器的相同原因,我们也会对每列数据应用单独的聚类方法。通过为每列数据创建一个聚类模型,我们可以更准确地测量数据单元的相似性,因为数据值仅在各自领域内具有可比性。由于标记将在聚类级别进行,我们需要一种方法来控制每列的聚类数量 k。较小的 k 值会产生较大的簇,这些簇更有可能包含脏数据和干净数据单元的混合。相反,较大的 k 值会产生更多的簇,需要用户提供更多的、可能不必要的标签。有一些聚类算法可以自动选择簇的数量,例如 DBSCAN [22]。然而,这些聚类方法的缺点是用户无法控制簇的数量,因此也无法控制所需的标签数量。

我们应用了分层聚类[5],这种方法不需要指定聚类数量(译注: Hierarchical Clustering (HC)，层次聚类)。这种方法从每列只有两个聚类开始,然后在每次迭代中增加聚类数量。它的迭代过程允许用户根据标签预算随意终止基于聚类的采样过程。由于相似度度量和连接方法的选择不会影响 Raha 的性能,我们使用了默认的余弦相似度度和平均连接方法。

元组选择。到目前为止,用户需要为同一元组的列值和其他值分别标注。然而,要标注一个数据单元,用户通常需要检查其整个元组。因此,与逐列标注单个单元格相比,对用户标注而言,对整个元组进行采样更为直观。

选择最佳元组并非易事,因为每列中可能包含完全不同的元组错误。在每次聚类迭代中,每个数据列被分为 k 个数据元组簇。其中一些簇是未标记的,即它们的数据元组都没有被标记。理想情况下,采样元组应覆盖每列中所有未标记的簇。这样,对采样元组进行标记即可对所有未标记的数据元组进行标记。然而,找到覆盖所有未标记簇的最小元组集是经典的集合覆盖问题[30]。为了解决这个 NP 完全问题,我们设计了一种具有两个属性的近似方法。首先,我们通过在每次迭代中只选择一个元组来缓解找到最小元组集的挑战。这样,我们避免了找到不覆盖同一簇集的元组的问题。其次,我们不是确定性地选择覆盖最多未标记簇的元组,而是概率性地选择该元组。这样,我们避免了陷入局部最优,因为事实证明,与确定性贪婪启发式算法相比,该问题的概率性解决方案对局部最优更具适应性[25]。因此,在每次迭代中,我们根据 softmax 概率函数绘制元组 t^*

$$P(t) = \frac{\exp(\sum_{c \in t} \exp(-N_c))}{\sum_{t' \in d} \exp(\sum_{c \in t'} \exp(-N_c))}, \qquad (3)$$

其中,Nc 是当前数据单元 c 集群中带有标签的数据单元的数量,exp 是以 e 为底数的指数函数。该评分公式有利于那些数据单元大多属于标签较少集群的元组。

所提出的基于聚类的抽样方案有两个特点。首先,由于采样方法会迭代对标记不足的集群进行聚类和标记,因此我们期望在某个时刻将每列分别聚类为同质干净或脏集群。换句话说,如果一个数据列中存在某种类型的错误,分层聚类方法最终会识别出其对应的集群。此外,采样方法还可以解决自然类不平衡的问题,因为罕见的脏标签将通过相应的集群传播。

在每次迭代结束时,我们得到一组用户标记的元组 $L \subset d$。在接下来的迭代中,只要用户的标记预算没有耗尽,即 $|L| < \theta_{labels}$,就会重复这一迭代过程,并使用更大的集群数量 $k \in \{2, 3, ...\}$。在采样过程结束时,每个数据列有 $k = \theta_{labels} + 1$ 个簇,且 $|L| = \theta_{labels}$ 个已标注元组。

4.4 标签传播和分类

在像我们这样的半监督设置中,拥有更多的用户标签可以加快模型的收敛速度[43]。因此,根据聚类假设[13],我们利用聚类来增加标签的数量。由于我们根据表达性特征向量对数据单元进行聚类,因此一个特定聚类中的单元很可能具有相同的脏/干净标签。因此,我们可以通过聚类传播用户标签。

Raha 将每个数据单元的用户标签传播到其集群中的所有数据单元。设 $L' = \{d[i, j] \mid i \in L, 1 \leqslant j \leqslant |A|\}$ 为带标签数据单元的集合,即带标签元组 L 的所有数据单元。所有未标记的数据单元 $d[i', j'] \notin L'$ 位于标记数据单元 $d[i, j] \in L'$ 的同一簇内,因此它们与数据单元 $d[i, j]$ 具有相同的脏/净标签。这样,训练标签 L' 的数量显著增加。

由于一个簇可能包含多个用户标记的数据单元,且这些单元的脏/干净标签相互矛盾,我们需要一种方法来解决这种矛盾。我们研究了两种冲突解决函数,以在簇中传播用户标签;一种是基于同质性的函数,另一种是基于多数的函数。基于同质性的函数仅在不包含两个具有矛盾标签的单元的簇中传播用户标签。这种方法基于用户标签完美的假设,在这种情况下,一个集群中存在矛盾的用户标签表明该集群的标签传播不够均匀。如果一个标签类别占多数,基于多数的函数也会在混合标签的集群中传播用户标签。直觉告诉我们,一个均匀的集群也可能存在矛盾的用户标签,这是由于用户标签错误造成的。在我们的数据集中,一列中的 k 个集群中的 k-1 个通常都是只有干净或脏数据单元的小集群,而最后一个 k'个集群是一个混合了脏数据和干净数据单元的大集群。

然后,Raha 为每个数据列 j 训练一个分类器 m_j,用于预测同一列中未标记数据单元的标签。(译注:此时未必所有的聚类模块都存在用户标记过的 t^*,所以需要预测那些没有用户标记的单元

的情况) 在训练阶段,每个分类模型 m_j 获取数据列中数据单元的特征向量(即 V_j)以及已标记数据单元(即 L')。在预测阶段,每个分类模型 m_j 预测每个数据列中其余数据单元的标签。请注意,虽然我们为每个数据列训练一个分类模型,但分类器仍然能够检测由于规则和知识库违规检测功能导致的列间依赖性违规错误。

4.5 系统分析

Raha 的主要目标是优化效率和用户参与度。Raha 的无配置特性是通过利用每个错误检测算法的大量预定义配置来实现的。虽然利用丰富的特征向量和基于聚类的采样方案可以在少量标签的情况下实现高精度,但 Raha 可能会提高效率问题,因为运行这些大量的错误检测策略可能会很耗时。虽然并行化可以缩短总体运行时间,但停止运行不相关的策略会更理想。为此,必须事先确定最有前景的策略,即特征。使用现有的特征选择技术无法做到这一点,因为它们需要策略对训练数据的输出。这就是为什么 Raha 支持基于历史数据的策略过滤方法,如下所述。

5 利用历史数据进行运行优化

如果有历史数据，即以前清理过的数据集，Raha 可以过滤不相关的错误检测策略。其原理是，在相似的数据域上，相似的错误检测策略会有相似的表现。例如，在域为 City 的数据列上，我们只需要运行那些在某些历史数据集的数据列 Capital 上表现良好的错误检测策略。这样，我们就能大大改善系统的运行时间。为此，我们需要解决两个难题。首先，我们需要一个相似性概念来捕捉错误检测策略与两列的相关性。其次，我们需要一种算法，能够根据错误检测策略在其他类似数据列中的有效性，为新数据列系统地调整和选择有前途的错误检测策略。这样，系统就能选择排名靠前的适应策略，并过滤掉其他策略。

5.1 数据列相似性

衡量数据列的相似性是模式匹配[37]和数据剖析[2]中一个经过深入研究的问题。在此研究基础上，我们用列轮廓来表示每个数据列。列轮廓表示数据列的语法和语义相似性，以暴露语法和语义数据错误。语法相似性可以根据字符分布的相似性来捕捉。例如，数据列 "年龄"和 "年份"在语法上相似，因为它们具有相似的字符分布，即数字字符。这就是为什么用非数字字符标记数据值的模式违规检测策略在这两列数据中都能很好地发挥作用。可以根据实际值的重叠来捕捉列的语义相似性。例如，数据列 "城市"和 "首都"在语义上是相似的，因为它们具有相似的值分布，即城市名称。这就是为什么基于 ZIP → City 的规则违规检测策略也

能适用于 ZIP → Capital。

列轮廓描述了数据列的内容及其字符和值分布。字符分布是一个概率分布函数，用于计算在数据列 d[:, j] 中观察到包含字符 ch 的数据单元格的概率。

$$p(ch|d[:,j]) = \frac{|\{d[i,j] \mid 1 \le i \le |d|, d[i,j] \text{ contains } ch\}|}{|d|}. \quad (4)$$

值分布是一个概率分布函数，用于计算在数据列 d[:, j] 中观察到数据单元格等于值 v 的概率。

$$p(v|d[:,j]) = \frac{|\{d[i,j] \mid 1 \le i \le |d|, d[i,j] \text{ equals to } v\}|}{|d|}. \quad (5)$$

总之，我们可以通过对两列的轮廓（即字符和值分布）应用任何相似性度量来测量它们的相似性。实际上，我们计算的是余弦相似度。

5.2 策略调整和选择

获得每一列与之前清理过的列的相似度后，我们就可以针对手头的数据集调整策略，根据策略的相关性对其进行排序，并从中选择最有前途的子集。

算法 2 展示了 Raha 如何利用一列 dnew[:, j] 和另一列 d[:, j′]之间的相似性，为新列 dnew[:, j]选择有前途的错误检测策略。该算法包括四个主要步骤。

首先，根据上一节的讨论计算新数据集 dnew 的每一列 dnew[:, j]与任意数据集 d 的每一列 d[:, j′]之间的相似性（第 7 行）。

第二，Raha 检索存储的策略 s 在列 d[:, j′]上的 F1 分数，即 F(s,d[:, j′])（第 9 行）。

第三，Raha 可能需要修改策略 s，使其能够在新列 dnew[:, j]上运行（第 10 行）。 对于离群点检测、模式违规检测或知识库违规检测策略来说，这种修改是不必要的。Raha 只需在新数据列上运行相同的策略即可。但是，对于规则违反检测策略，Raha 需要根据新数据集的模式修改规则。假设我们有一个功能依赖检查器 $s_{j'1 \to j'2}$,它检测到历史数据集 d 的数据列 $d[:, j'_1]$ 和 $d[:, j'_2]$ 存在数据错误。假设 Raha 识别出历史数据列 $d[:, j'_1]$ 与新数据列 dnew[:, j] 相似，因此,Raha 将策略 $s_{j'1 \to j'2}$ 转换为新数据集 dnew 的策略 $s_{j \to q}$。为了适应策略 $s_{j'1 \to j'2}$,Raha 将数据列 $d[:, j'_1]$替换为其对应的数据列 dnew[:, j],并将数据列 $d[:, j'_2]$替换为与其最相似的数据列

dnew[:,q],这些列都在新数据集中。

第四，Raha 会给每个更新的错误检测策略 s′ 打分（第 11 行）。该分数是数据列 dnew[:, j] 和 d[:, j′]的相似度与策略 s 在数据列 d[:, j′]上的 F1 分数的乘积。形式上，

$$\text{score}(s') = \text{sim}(d_{new}[:, j], d[:, j']) \times F(s, d[:, j']). \quad (6)$$

如果数据列 dnew[:, j] 和 d[:, j′]相似，且策略 s 在数据列 d[:, j′]上的 F1 得分高，则得分就高，这表明更新后的策略 s′ 对数据列 dnew[:, j] 有帮助。

Algorithm 2: StrategyFilterer(d_{new}, S, D, D^*).

Input: new dataset d_{new}, set of error detection strategies S, set of historical datasets D, set of historical ground truths D^*.
Output: set of promising error detection strategies S^*.

1 **for** *each data column* $j \in [1, |A_{new}|]$ *of dataset* d_{new} **do**
2 S'_j gain$(S^*_j) = \sum_{s \in S^*_j} score(s) - \frac{1}{2} \sum_{s \in S^*_j} \sum_{s' \neq s \in S^*_j} |score(s) - score(s')|$. for column j
3 p_d
4 fo (7)
5 **for** *each data column* $j' \in [1, |A|]$ *of dataset* d **do**
6 $p_{d[:,j']} \leftarrow$ generate profile for data column $d[:, j']$;
7 $\text{sim}(d_{\text{new}}[:, j], d[:, j']) \leftarrow$ similarity of $p_{d_{\text{new}}[:,j]}$ and $p_{d[:,j']}$;
8 **for** *each strategy* $s \in S$ **do**
9 $F(s, d[:, j']) \leftarrow F_1$ score of s on $d[:, j']$;
10 $s' \leftarrow$ adapt s for dataset d_{new};
11 score$(s') \leftarrow$ assign the score of s';
12 $S'_j \leftarrow S'_j \bigcup \{s'\}$;
13 $S^*_j \leftarrow \{\}$; // set of promising adapted strategies for column j
14 **do**
15 $s^* \leftarrow \arg\max_{s \in S'_j} \text{score}(s)$;
16 $S^*_j \leftarrow S^*_j \bigcup \{s^*\}$;
17 $S'_j \leftarrow S'_j - \{s^*\}$;
18 **while** *adding* s^* *to* S^*_j *does not decrease the gain of* S^*_j ;
19 $S^* \leftarrow \bigcup_{j=1}^{|A_{\text{new}}|} S^*_j$;

我们可以对每列数据 dnew[:, j] 的得分策略 S'_j 进行排序，从而只选出每列数据中得分最高

的策略。选择最有希望的策略子集的无阈值方法是应用文献 [3] 中使用的增益函数。其思路是将得分最高的策略 $s^* \in S'_j$ 迭代添加到最有希望的策略集 S^*_j 中（第 13 行至第 18 行），直到添加下一个最佳策略使下面的增益函数[3]减小，增益达到局部最大值：

$$\text{gain}(S_j^*) = \sum_{s \in S_j^*} score(s) - \frac{1}{2} \sum_{s \in S_j^*} \sum_{s' \neq s \in S_j^*} |score(s) - score(s')|. \tag{7}$$

最后，有希望的错误检测策略集 S^*是所有数据列中有希望的策略的集合（第 19 行）。 在算法 2 中，Raha 会遍历所有历史数据集（第 4 行）及其数据列（第 5 行）。相反，可以创建数据列索引，如 MinHash [12]，为新数据列快速查找相关的历史数据集和数据列。不过，由于历史清理数据集的数量通常有限（例如，在我们的实验设置中为 8 个），因此我们忽略了这一优化。

6 实验

我们的实验旨在回答以下问题。(1) 我们的系统与现有的错误检测方法相比如何? (2) 每个特征组的影响是什么? (3) 采样如何影响系统的收敛? (4) 我们的策略过滤方法与其他技术相比如何? (5) 用户标记错误如何影响系统性能? (6) 系统性能如何随着行数和列数的增加而变化? (7) 分类模型的选择如何影响系统性能?由于篇幅有限,最后两项实验在附录 B 中介绍。附录 C 还提供了关于用户标注过程的案例研究。

Table 4: Dataset characteristics. The error types are missing value (MV), typo (T), formatting issue (FI), and violated attribute dependency (VAD) [38].

Name	Size	Error Rate	Error Types
Hospital	1000 × 20	0.03	T, VAD
Flights	2376 × 7	0.30	MV, FI, VAD
Address	94306 × 12	0.14	MV, FI, VAD
Beers	2410 × 11	0.16	MV, FI, VAD
Rayyan	1000 × 11	0.09	MV, T, FI, VAD
Movies	7390 × 17	0.06	MV, FI
IT	2262 × 61	0.20	MV, FI
Tax	200000 × 15	0.04	T, FI, VAD

6.1 实验设置

我们在表 4 中描述的 8 个数据集上评估了我们的系统。*Hospital* 和 *Flights* 是两个真实数据集,我们是从以前的研究项目中获得的,并附带了它们的真实数据[39]。*Address* 是一个带有真实数据的专有数据集。*Beers* 是一个真实数据集,是通过抓取网络收集的,并经过人工清理[28]。*Rayyan* [33]和 *IT* [1]也是真实数据集,由数据集所有者自己清理。*Movies* 是 Magellan 存储库中可用的数据集[19]。我们使用重复元组的现有标签为该数据集提供地面真实数据。*Tax* 是 BART 存储库中一个大型合成数据集[6]。我们使用该数据集来评估 Raha 的可扩展性。

我们利用不同的评估指标来评估我们的系统。我们报告精确度、召回率和 F1 评分来评估有效性。我们还报告运行时间(以秒为单位)来评估效率。我们报告标记元组的数量来评估人工参与度。由于我们的元组采样方法是概率性的,因此我们将评估指标报告为 10 次独立运行的平均值。为了便于阅读,我们在图中省略了±1%,因为标准偏差总是小于 1%。

作为默认参数设置,我们将所有特征组都纳入了我们的系统。我们使用梯度提升[26]作为分类模型。我们将用户的标签预算设置为 $\theta_{labels} = 20$。我们在一台 Ubuntu 16.04 LTS 机器上运行所有实验,该机器有 28 个 2.60 GHz 内核和 264 GB 内存。我们的系统可在线使用 1。

6.2 与基准的比较

独立错误检测工具。我们将 Raha 与表 5 中的三个独立错误检测工具进行了比较。*dBoost* [36]是一个异常检测工具。*NADEEF* [18]是一个基于规则的数据清理系统。*KATARA* [17]使用知识库来检测错误。*ActiveClean* [32]是一种基于机器学习的的数据清理方法。我们在附录 A 中详细介绍了每种基准方法及其用法。表 5 显示,在 F1 评分方面,我们的方法在所有数据集上均优于所有独立错误检测工具,至少高出 12%,最高高出 42%。我们的无配置错误检测系统只需要有限数量的标记元组,即 $\theta_{labels} = 20$。

Table 5: Comparison with the stand-alone error detection tools.

Approach	Hospital			Flights			Address			Beers			Rayyan			Movies			IT		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
dBoost	0.54	0.45	0.49	0.78	0.57	0.66	0.23	0.50	0.31	0.54	0.56	0.55	0.12	0.26	0.16	0.18	0.72	0.29	0.00	0.00	0.00
NADEEF	0.05	0.37	0.09	0.30	0.06	0.09	0.51	0.73	0.60	0.13	0.06	0.08	0.74	0.55	0.63	0.13	0.43	0.20	0.99	0.78	0.87
KATARA	0.06	0.37	0.10	0.07	0.09	0.08	0.25	0.99	0.39	0.08	0.26	0.12	0.02	0.10	0.03	0.01	0.17	0.02	0.11	0.17	0.14
ActiveClean	0.02	0.14	0.03	0.28	**0.94**	0.44	0.14	**1.00**	0.25	0.16	**1.00**	0.28	0.09	**1.00**	0.16	0.02	0.01	0.01	0.20	**1.00**	0.33
Raha	**0.94**	**0.59**	**0.72**	**0.82**	0.81	**0.81**	**0.91**	0.80	**0.85**	**0.99**	0.99	**0.99**	**0.81**	0.78	**0.79**	**0.85**	**0.88**	**0.86**	**0.99**	0.99	**0.99**

dBoost、*NADEEF* 和 *KATARA* 的召回率较低,因为它们都针对特定类型的错误:*dBoost* 将统计异

常值标记为错误,*NADEEF* 主要针对用户定义的完整性规则检测到的错误,而 *KATARA* 将不符合知识库中实体关系的数据值标记为错误。换句话说,这些工具无法捕获 Raha 检测到的许多错误。*dBoost* 的精度低是由于所应用的启发式算法将合法值识别为错误。*NADEEF* 的精度也较低,因为与其他基于完整性规则的方法类似,它以粗粒度形式报告错误,即以由多个数据单元组成的违规形式报告错误。*KATARA* 的精度低是由于概念的模糊性导致手头数据与知识库中的数据概念不匹配。就用户参与而言,*dBoost*、*NADEEF* 和 *KATARA* 需要用户仔细提供的输入配置。

ActiveClean 的性能不佳是由于它的假设。首先,*ActiveClean* 以元组为单位而不是以单元为单位工作,即它输出脏元组而不是单元。这种设置导致精度不佳,因为输出元组中的所有数据单元实际上并不脏。其次,*ActiveClean* 假设特征是由用户给出的,而 *TF-IDF* 特征化只是一个后备计划。*TF-IDF* 不能有效反映每个数据单元的数据质量问题。第三,*ActiveClean* 假定存在支持元组采样策略的机器学习任务,而不是为通用错误检测而设计。*ActiveClean* 还利用了与 Raha 相同数量的标记元组,即 $\theta_{labels} = 20$。下面,我们还将提供元组级的比较。

元组级错误检测方法。由于 *Active-Clean* 是为检测错误元组而设计的,因此我们还将 Raha 与 *ActiveClean* 在输出错误元组正确性和完整性方面的表现进行了比较。

Table 6: Comparison in terms of detecting erroneous tuples.

Approach	Hospital			Flights			Address			Beers			Rayyan			Movies			IT		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
ActiveClean	0.33	0.12	0.18	0.80	**1.00**	0.89	0.73	**0.99**	0.84	1.00	1.00	1.00	0.77	**1.00**	0.87	0.77	**1.00**	0.87	1.00	1.00	1.00
Raha	**0.96**	**0.67**	**0.79**	**0.85**	0.93	**0.89**	**0.94**	0.94	**0.94**	**1.00**	**1.00**	**1.00**	**0.88**	0.88	**0.88**	**0.90**	0.97	**0.93**	**1.00**	**1.00**	**1.00**

如表 6 所示,在所有数据集上,Raha 在元组 F1 得分方面也优于 ActiveClean。这两个系统在 *Beers* 和 *IT* 数据集上都能给出完美的结果,因为这些数据集包含完全错误的数据列。在这些数据集上,元组错误检测并不重要,因为所有元组都是错误的。为了进一步研究,我们从数据集中删除了这些列。现在,Raha 明显优于 *ActiveClean*;在 *Beers* 和 *IT* 数据集上,它的 F1 得分分别为 0.95 和 1.0,而 *ActiveClean* 的得分分别为 0.64 和 0.90。

错误检测聚合器。在图 2 中,我们还比较了我们的系统与三个错误检测聚合器的性能。这些聚合器方法与我们的系统一样,在内部结合了多种错误检测策略。我们在附录 A 中详细介绍了每种基准方法及其用法。

- Min-k [1]输出由超过 k%错误检测策略标记的数据单元。

• 基于最大熵的次序选择[1] 输出误差检测策略的输出结果,对评估的数据样本进行高精度处理。

• 元数据驱动方法[45]是一种基于机器学习的聚合器,结合了手动配置的独立错误检测工具。

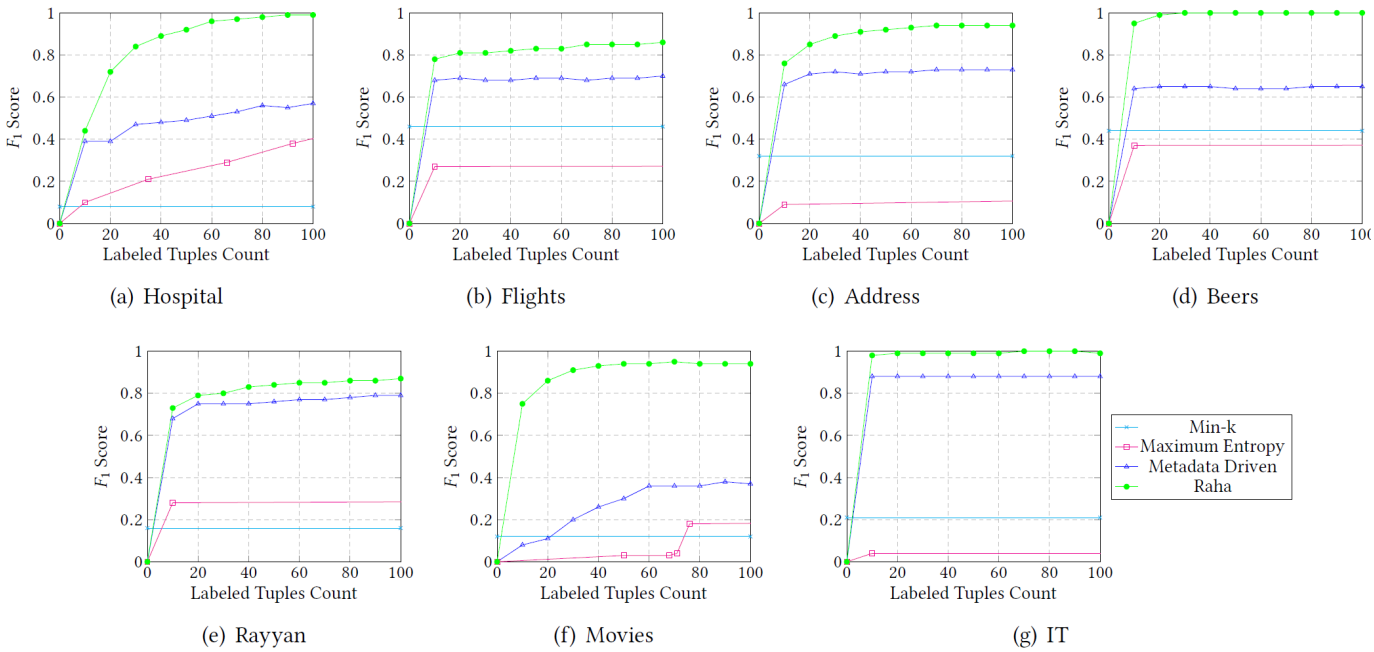


Figure 2: Comparison with the error detection aggregators.

如图 2 所示,在所有数据集上,Raha 在 F1 评分和用户标签方面都优于所有聚合方法。它收敛速度更快,需要的标记元组更少。Raha 相对于 min-k 的显著优势表明 Raha 涵盖了各种数据错误类型。我们假设用户知道最佳 k,并绘制最佳 k 的结果。

6.3 特征影响分析

我们在表 7 中分析了不同特征组对系统性能的影响。我们运行了包含所有特征组的系统(行 *All*)。然后,我们逐个排除每个特征组,以分析其影响。例如, *All - OD* 表示 Raha 利用了所有特征组,但排除了异常检测特征组。这里,我们还报告了 Raha 在使用 *TF-IDF* 特征时的有效性,这是 *ActiveClean* 的特征化方法[32]。

Table 7: System effectiveness with different feature groups: outlier detection (OD), pattern violation detection (PVD), rule violation detection (RVD), knowledge base violation detection (KBVD), and all together (All).

Feature Group	Hospital			Flights			Address			Beers			Rayyan			Movies			IT		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
TF-IDF	0.98	0.10	0.18	0.63	0.88	0.73	0.84	0.57	0.68	0.73	0.80	0.76	0.91	0.58	0.70	0.19	0.04	0.07	0.92	0.97	0.95
All - OD	0.93	0.53	0.68	0.80	0.85	0.82	0.89	0.84	0.86	0.95	0.95	0.95	0.78	0.72	0.75	0.66	0.82	0.73	0.99	0.98	0.98
All - PVD	0.95	0.62	0.75	0.83	0.84	0.83	0.87	0.89	0.88	0.92	0.94	0.93	0.74	0.74	0.74	0.77	0.84	0.80	0.98	0.97	0.97
All - RVD	0.75	0.37	0.50	0.80	0.78	0.79	0.86	0.87	0.86	0.97	0.98	0.97	0.82	0.78	0.80	0.92	0.90	0.91	0.99	0.98	0.98
All - KBVD	0.95	0.60	0.74	0.85	0.78	0.81	0.85	0.76	0.80	0.98	0.98	0.98	0.83	0.76	0.79	0.80	0.88	0.84	0.99	0.97	0.98
All	0.94	0.59	0.72	0.82	0.81	0.81	0.91	0.80	0.85	0.99	0.99	0.99	0.81	0.78	0.79	0.85	0.88	0.86	0.99	0.99	0.99

如表 7 所示,Raha 对删除特征组具有鲁棒性,因为当特征组被排除时,其性能不会崩溃。然而,根据错误率和普遍的数据错误类型,删除特征组可能会显著降低性能。例如,在包含数百个 FD 的医院数据集中,删除规则违规检测功能会严重降低性能。在错误率较低的电影数据集中,数据错误主要是异常值,删除异常值检测功能会严重降低性能。

有趣的是,使用 Raha 的采样方法,*ActiveClean* 的 *TF-IDF* 特征化可以带来更高的 F1 评分。然而,词级 *TF-IDF* 特征的整体性能总是比 Raha 的全特征集差。

6.4 采样影响分析

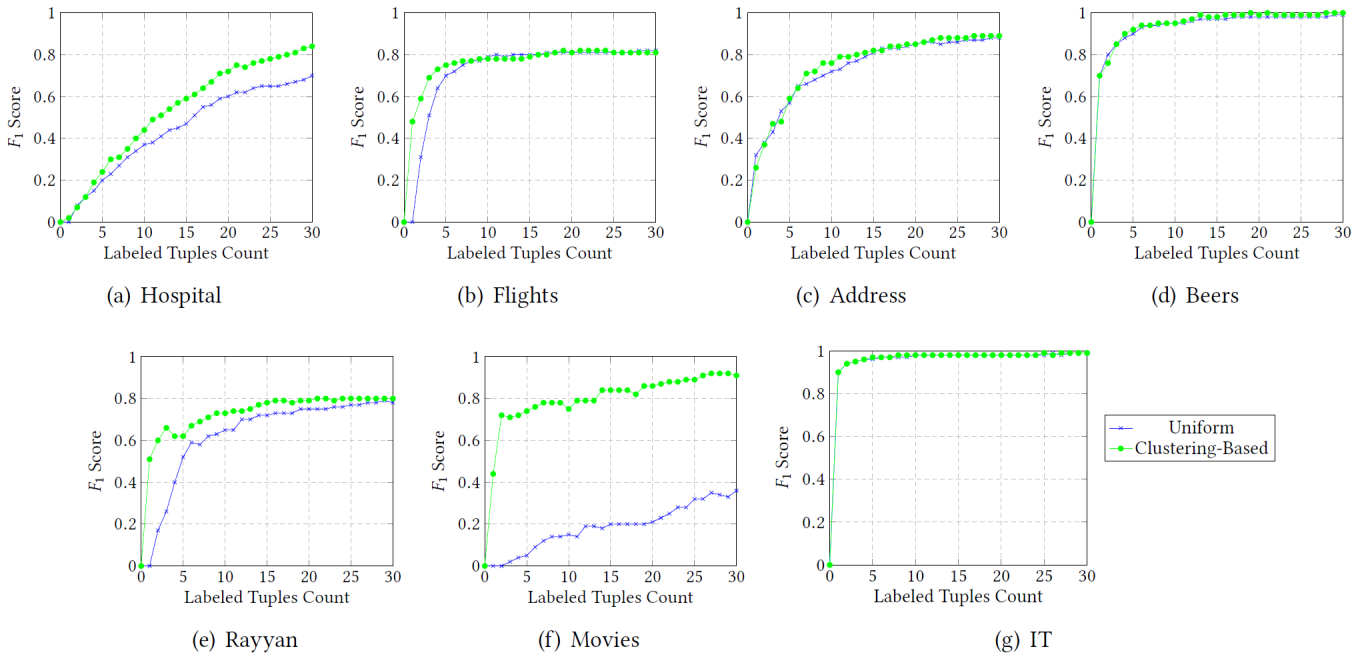


Figure 3: System effectiveness with different sampling approaches.

我们分析了采样方法对系统性能的影响,如图 3 所示。具体而言,我们比较了两种不同采样方法的两种系统版本。均匀采样方法根据均匀概率分布[45]为用户标注选择元组。另一方面,我们提出的基于聚类的采样方法首先根据现有的数据单元集群选择元组,然后通过集群传播标签。

如图 3 所示,我们基于聚类的采样方法加快了系统的收敛速度。这种更高的收敛速度在错误率较低的数据集(如 *Hospital* 和 *Movies*)上更为明显。在错误率较低的数据集上,很难找到足够的脏数据单元来训练分类器。然而, 我们基于聚类的采样方法通过使用每个集群中的额外噪声标签解决了这个问题。

6.5 策略过滤影响分析

我们通过历史数据分析了策略过滤对 Raha 性能的影响。我们使用手头的数据集,按照以下方

式进行了实验。在每次运行中,我们根据众所周知的“留一法”[40],将一个数据集作为新的脏数据集 dnew,其余数据集作为历史数据集 D。然后,我们根据第 5 节中描述的方法只选择有前途的错误检测策略。这样,我们就可以分析有限计算资源对特征提取的影响。请注意,我们考虑了策略过滤(第 5 节)和特征提取(第 4.2 节)的运行时间。特征提取显然占用了我们系统的大部分运行时间,因为它需要对数据集运行多个错误检测策略。

理想情况下,历史数据应该包含与新数据集来自相同领域中的数据集。然而,我们强烈认为用户的历史数据中总是有类似的数据集。因此,我们仅假设用户在历史数据中拥有一些经过清理的数据集,这些数据集可能与新数据集具有一些相似的数据列(例如,城市和邮政编码)。即使在我们来自不同领域的各种数据集中,我们仍然可以找到相似的列并过滤掉不相关的策略,从而提高运行时间。

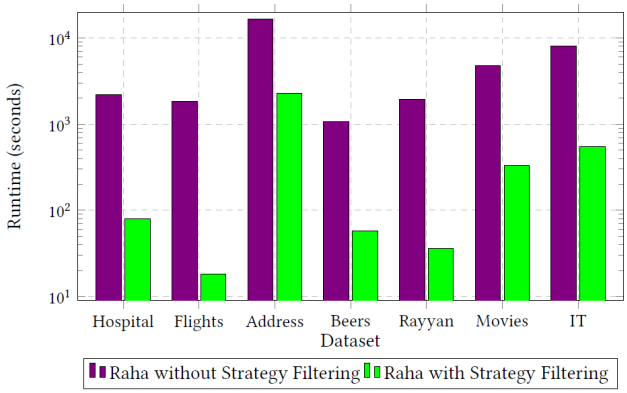


Figure 4: System efficiency with/without strategy filtering via historical data.

图 4 显示了 Raha 在有策略过滤和无策略过滤情况下的运行时间。由于系统只需要运行所有可能错误检测策略中的一小部分,因此运行时间显著提高了不止一个数量级。

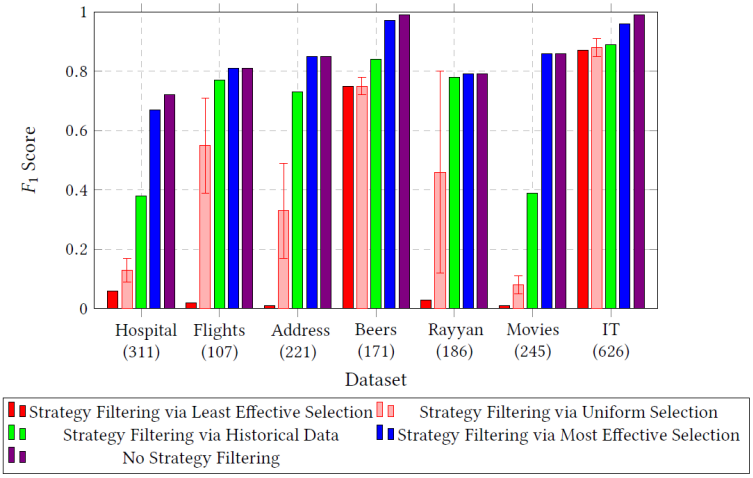


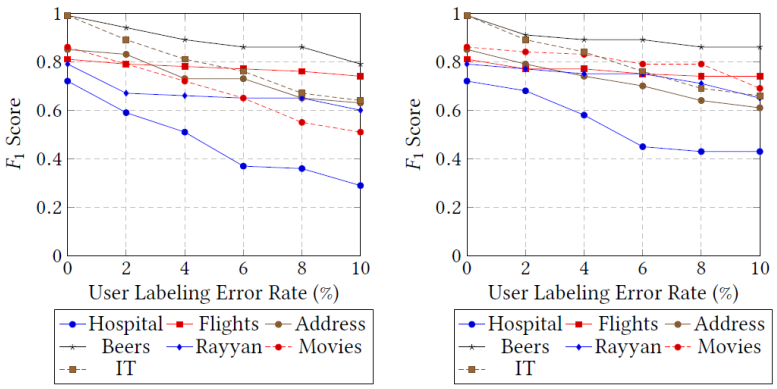
Figure 5: System effectiveness with different strategy filtering approaches. The numbers of selected strategies are denoted inside the brackets.

我们还通过历史数据将我们的策略过滤方法与其他策略过滤方法进行了比较,以更好地评估其有效性。图 5 显示了五种不同方法的 F_1 评分。我们利用数据集的真实数据来评估所有错误检测策略的 F_1 评分,以便对其进行相应排序。因此, 第一种和第四种方法是以最不有效和最有效的策略为特征的 Raha。这两种极端方法可以作为任何其他策略过滤方法有效性的下限和上限。第二种方法是在 Raha 中采用统一策略过滤, 将错误检测策略作为特征。由于这种方法是概率性的,我们重复了 5 次,并报告了平均值和标准差。第三和第五种方法是 Raha 中分别通过历史数据过滤策略以及不进行任何策略过滤。请注意, 所有策略过滤方法选择的策略数量相同, 由 Raha 来计算。

如图 5 所示,通过历史数据筛选策略的方法比效果最差的策略选择和统一策略选择方法的效果更好。Raha 通过历史数据筛选策略的有效性略低于最有效的策略选择方法。我们的策略筛选方法几乎可以达到相同的效果,而无需在新数据集上运行或评估任何策略。请注意, Raha 在没有策略筛选的情况下,其有效性高于最有效的策略选择方法。这表明,即使无效的错误检测策略仍然可以为特征向量添加一些信息。

通常, 基于历史数据的分析结果能够显示列特征如何有效地捕捉数据集的数据质量问题。例如, 针对 *Hospital* 数据集,所选的有效策略主要是直方图建模策略,这些策略对数据集非常有效。另一方面,针对 *Address* 数据集,所选的有效策略主要是功能依赖性检查器,这些策略涉及属性 ZIP、City 和 State。我们的列特征能够将这些数据列有效地映射到 *Hospital* 和 *Beers* 中的类似数据域。

6.6 用户标注错误影响分析



(a) Homogeneity Resolution (b) Majority Resolution

Figure 6: System effectiveness in the presence of user labeling errors with (a) homogeneity-based and (b) majority-based conflict resolution functions.

在图 6 中,我们比较了第 4.4 节中标签传播的冲突解决函数在存在用户标注错误情况下的表现。错误的用户标签随机分布在数据单元格中。

如图 6 所示,Raha 的有效性随着用户标注错误的增加而略有下降。在 Hospital 数据集上,这种下降更为严重,因为该数据集包含许多类似的数据错误,即随机插入字符“x”导致的拼写错误。对于这种类似的数据错误,错误的用户标签会使分类器感到困惑。在存在用户标签错误的情况下,基于多数的冲突解决函数比基于同质的冲突解决函数表现更好,因为如果用户的标签相互矛盾,后者不会进行任何标签传播。在这些情况下,基于多数的冲突解决功能更稳健。在用户标签完美的假设下,两种方法的表现几乎相同。由于这两种解决方案不能在所有数据集上都被视为稳健的,我们认为处理错误的用户标签是未来研究的一个有趣的方向。

7 相关工作

预配置错误检测。这一系列工作假设错误检测工具在使用前已经配置好。对于异常值检测,用户必须手动设置参数[9]。对于规则违反检测,用户必须提供功能依赖(FD)[4]、条件功能依赖(CFD)[23]及其变体[11, 31]、拒绝约束(DC)[16]或用户定义函数(UDF)[8, 18]。正确配置所有工具是一项繁琐的工作,需要用户同时了解数据集和错误检测工具。相比之下, Raha 免除了用户配置工具的繁琐工作,并且仍然优于预配置方法。

聚合错误检测。另一条研究路线是通过投票、基于精度的排序[1]或集成方法[45]来聚合各种错误检测技术的结果。这些方法的假设前提是用户已经配置了每个错误检测工具。Raha 没有做出这样的假设,因为我们自动生成了许多配置。正如我们的实验所显示的那样, Raha 明显优于这些技术。

交互式错误检测。除了使用预先配置的错误检测工具,另一种思路是让人工参与进来。为了清理数据, 后端数据清理引擎通过数据转换(例如 Trifacta [21]、BlinkFill [42]和 Falcon [27])或机器学习模型(例如 GDR [46]和 ActiveClean [32])来概括用户反馈。人类参与框架面临的一个普遍挑战是类别失衡问题,即脏数据值的数量远低于干净数据值。这种类别失衡通常会导致需要大量用户标注。相比之下, Raha 利用基于聚类的采样和标签传播,大大减少了所需的用户标签数量。

数据剖析。许多数据剖析算法被用于检测 FDs [2, 10]、CFDs [24]和 DCs [15]。然而,从脏数据中发现规则会产生许多误报[15],需要用户从发现的候选规则中挑选出真正的规则。这就是为

什么 Raha 不需要完美的错误检测策略。

数据修复。在数据修复方面已经有很多工作,例如 ActiveClean [32]、HoloClean [39]和 NADEEF [18]。我们的工作与数据修复是互补的, 因为我们专注于错误检测。Raha 的输出,即检测到的数据错误,稍后会输入到这些数据修复方法中。然而,我们将 Raha 与这些系统的错误检测组件进行了比较。

8 结论

我们提出了一种新颖的错误检测系统,该系统可帮助用户摆脱选择和配置错误检测算法的繁琐任务。Raha 系统地生成各种算法配置,并将其输出编码为所有数据单元的特征向量。然后, Raha 结合每个数据列的聚类、标签传播和分类,并学习预测每个数据列中所有数据单元的标签。此外, 我们提出了一种策略过滤方法, 根据可选的历史数据过滤不相关的错误检测策略。我们的实验清楚地表明了 Raha 如何超越现有的基准。

虽然 Raha 以一种轻松的方式提供了卓越的性能,但它也有一些局限性。Raha 的无配置特性要求我们运行许多错误检测策略。与许多机器学习方法一样, Raha 无法保证用户通过多少算法、配置甚至用户标签可以获得所需的性能。Raha 试图通过一些启发式方法来限制所有可能算法/配置的无限集合。这些搜索空间剪枝启发式方法排除了某些潜在的相关策略,例如时间函数依赖性。

Raha 有几种可能的改进。第一种是通过并行化工作流程进行运行时优化。虽然 Raha 支持策略过滤, 但其流水线仍然是顺序的。第二种是通过众包取代专家用户来降低用户标记成本, 这需要处理嘈杂的标签。处理用户标记错误,并为用户标记过程提供重要的上下文和元数据,是未来重要的研究方向。目前, Raha 要求用户一次标记一个元组, 这可能无法提供足够的上下文来识别某些语义错误类型, 例如功能依赖性冲突和系统性数据错误。最后, 我们还计划扩展我们的系统,使其能够同时进行数据修复。