Semi-Supervised Data Cleaning

vorgelegt von
Mohammad Mahdavi Lahijani

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
–Dr.-Ing.–

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr. Klaus-Robert Müller
Gutachter: Prof. Dr. Ziawasch Abedjan
Gutachter: Prof. Dr. Wolfgang Lehner
Gutachter: Prof. Eugene Wu, PhD
Tag der wissenschaftlichen Aussprache: 6. November 2020

Berlin 2020

Abstract

Data cleaning is one of the most important but time-consuming tasks for data scientists. The data cleaning task consists of two major steps: (1) error detection and (2) error correction. The goal of error detection is to identify wrong data values. The goal of error correction is to fix these wrong values. Data cleaning is a challenging task due to the trade-off among correctness, completeness, and automation. In fact, detecting/correcting all data errors accurately without any user involvement is not possible for every dataset.

We propose a novel data cleaning approach that detects/corrects data errors with a novel two-step task formulation. The intuition is that, by collecting a set of base error detectors/correctors that can independently mark/fix data errors, we can learn to combine them into a final set of data errors/corrections using a few informative user labels. First, each base error detector/corrector generates an initial set of potential data errors/corrections. Then, the approach ensembles the output of these base error detectors/correctors into one final set of data errors/corrections in a semi-supervised manner. In fact, the approach iteratively asks the user to annotate a tuple, i.e., marking/fixing a few data errors. The approach learns to generalize the user-provided error detection/correction examples to the rest of the dataset, accordingly.

Our novel two-step formulation of the error detection/correction task has four benefits. First, the approach is configuration free and does not need any user-provided rules or parameters. In fact, the approach considers the base error detectors/correctors as black-box algorithms that are not necessarily correct or complete. Second, the approach is effective in the error detection/correction task as its first and second steps maximize recall and precision, respectively. Third, the approach also minimizes human involvement as it samples the most informative tuples of the dataset for user labeling. Fourth, the task formulation of our approach allows us to leverage previous data cleaning efforts to optimize the current data cleaning task.

We design an end-to-end data cleaning pipeline according to this approach that takes a dirty dataset as input and outputs a cleaned dataset. Our pipeline leverages user feedback, a set of data cleaning algorithms, and a set of previously cleaned datasets, if available. Internally, our pipeline consists of an error detection system (named Raha), an error correction system (named Baran), and a transfer learning engine.

As our extensive experiments show, our data cleaning systems are effective and efficient, and involve the user minimally. Raha and Baran significantly outperform existing data cleaning approaches in terms of effectiveness and human involvement on multiple well-known datasets.

Zusammenfassung

Die Datenbereinigung ist eine der wichtigsten, aber zeitraubendsten Aufgaben für Datenwissenschaftler. Die Datenbereinigungsaufgabe besteht aus zwei Hauptschritten: (1) Fehlererkennung und (2) Fehlerkorrektur. Das Ziel der Fehlererkennung ist es, falsche Datenwerte zu identifizieren. Das Ziel der Fehlerkorrektur ist es, diese falschen Werte in korrekte Werte zu korrigieren. Die Datenbereinigung ist eine anspruchsvolle Aufgabe aufgrund des Kompromisses zwischen Korrektheit, Vollständigkeit und Automatisierung. Tatsächlich ist es nicht für jeden Datensatz möglich, alle Datenfehler ohne Beteiligung des Benutzers genau zu erkennen und zu korrigieren.

Wir schlagen einen neuartigen Datenbereinigungsansatz vor, der Datenfehler mit einer neuartigen zweistufigen Aufgabenformulierung entdeckt/korrigiert. Die Intuition ist, dass wir durch das Sammeln eines Menge von Basisfehler-Detektoren/Korrektoren, die Datenfehler unabhängig voneinander markieren/beheben können, lernen können, sie mit Hilfe einiger informativer Benutzerannotationen zu einem endgültigen Satz von Datenfehlern/Korrekturen zu kombinieren. Zunächst erzeugt jeder Basisfehler-Detektor/Korrektor einen ersten Satz potenzieller Datenfehler/Korrekturen. Dann fasst der Ansatz die Ausgabe dieser Basisfehler-Detektoren/Korrektoren in einer halb-überwachten Weise zu einem endgültigen Satz von Datenfehlern/Korrekturen zusammen. Tatsächlich fordert der Ansatz den Benutzer iterativ auf, ein Tupel zu annotieren, d.h. einige wenige Datenfehler zu markieren/zu korrigieren. Der Ansatz lernt, die vom Benutzer zur Verfügung gestellten Fehlererkennungs-/Korrekturbeispiele entsprechend auf den Rest des Datensatzes zu zu generalisieren.

Unsere neuartige zweistufige Formulierung der Fehlererkennungs-/Fehlerkorrekturaufgabe hat vier Vorteile. Erstens ist der Ansatz konfigurationsfrei und benötigt keine vom Benutzer bereitgestellten Regeln oder Parameter. Tatsächlich betrachtet der Ansatz die Basis-Fehlerdetektoren/-korrektoren als Black-Box-Algorithmen, die nicht unbedingt korrekt oder vollständig sind. Zweitens ist der Ansatz bei der Fehlererkennungs-/Fehlerkorrekturaufgabe effektiv, da seine ersten und zweiten Schritte die Empfindlichkeit bzw. Genauigkeit maximieren. Drittens minimiert der Ansatz auch die menschliche Beteiligung, da er die informativsten Tupel des Datensatzes für die Benutzerbeschriftung auswählt. Viertens ermöglicht uns die Aufgabenformulierung unseres Ansatzes, frühere Datenbereinigungsbemühungen zur Optimierung der aktuellen Datenbereinigungsaufgabe zu nutzen.

Wir entwerfen nach diesem Ansatz eine End-to-End-Datenbereinigungs-Pipeline, die einen fehlerhaften Datensatz als Eingabe nimmt und einen bereinigten Datensatz ausgibt. Unsere Pipeline nutzt das Benutzer-Feedback, einen Satz von Datenbereinigungsalgorithmen und, falls verfügbar, historische Daten. Intern besteht unsere Pipeline aus einem Fehlererkennungssystem (namens Raha), einem Fehlerkorrektursystem (namens Baran) und einer Transferlernmaschine.

Wie unsere umfangreichen Experimente zeigen, sind unsere Datenbereinigungssysteme effektiv und effizient und involvieren den Benutzer nur minimal. Raha und Baran übertreffen bestehende Datenbereinigungsansätze in Bezug auf Wirksamkeit und menschliche Beteiligung bei mehreren bekannten Datensätzen erheblich.

This dissertation is dedicated to my parents, who are my greatest treasure.

Acknowledgements

I have been always rich, not because of money, but due to the awesome colleagues, friends, and family members who have been around me. Here, I would like to express my gratitude to those people who specifically supported me during the PhD journey.

I would like to thank my advisor Professor Ziawasch Abedjan, who trusted me in the first place and continuously supported me during the PhD period. I am also grateful to Professor Volker Markl, who let me scientifically grow up using their infrastructures at the Database Systems and Information Management group.

I would like to also thank my colleagues (actually friends) at the Big Data Management group: Larysa Visengeriyeva, Felix Neutatz, Maximilian Dohlus, Mahdi Esmailoghli, Binger Chen, Milad Abbaszadeh, Hagen Anuth, Jeremy Bilic, and Rajasiman Srinivasan.

My gratitude also goes to all my colleagues and friends at the Database Systems and Information Management group. I would like to specifically thank my colleagues Claudia Gantzer, Lutz Friedel, and Melanie Neumann, who always helped me in administrative or technical tasks.

I also deeply thank my family and friends for their continuous love and support. I definitely could not reach this point without all sacrifices they have made for me.

Finally, I am grateful to whoever has taught me even a word in my life.

Table of Contents

List of Figures

List of Tables

1 Introduction

Data-driven science has been recognized as the fourth paradigm of science after experimental, theoretical, and computational sciences [42]. Data-driven approaches are nowadays popular in many diverse domains, such as healthcare [36], journalism [44], and energy management [97]. Traditionally, the experts of these domains made decisions based on their intuitions and experiences. Alternatively, data-driven approaches extract knowledge and insights from historical data to support decision makers in future decision making processes. This way, business managers can run their business more effectively and efficiently using these decision support systems [49].

Data science is an inter-disciplinary field of science that provides the knowledge and insights for decision support systems [69]. Data science uses various theories and techniques drawn from multiple fields, such as statistics, data mining, and machine learning, to extract knowledge and insights from raw data. A data science pipeline consists of multiple steps, such as data collection, data preprocessing, data analysis, and data visualization.

The data preprocessing step is particularly responsible for preparing high quality data for the rest of the data science pipeline through multiple tasks, such as data cleaning, data transformation, and feature extraction [70]. Data preprocessing is important for data science pipelines to avoid the "garbage in, garbage out" situation, where low quality input data produces low quality output insights.

Working with the low quality datasets is nowadays inevitable. The integrated datasets are usually dirty, i.e., contain erroneous values, because of various reasons, such as typos, inconsistent handling of data, and extraction errors [72]. Processing these dirty datasets in data science pipelines is hard as they are not directly usable for data analysis algorithms or they can lead to wrong knowledge and insights. Therefore, we need to clean datasets in the data preprocessing step to ensure that the quality of data is preserved.

1.1 Problem Overview: Data Cleaning

Data cleaning aims at improving data quality [72]. A data cleaning task consists of two major steps: (1) *error detection* and (2) *error correction* (i.e., data repairing). The goal of error detection is to identify data values that are wrong [2]. Error detection can be considered as a binary classification task, where each data value is classified into two possible classes, namely clean or dirty. The goal of error correction is to fix the wrong values [74]. Error correction is a

more challenging step as the space of possible correction candidates is infinite. In fact, any possible string could theoretically be a correction candidate for an erroneous value.

Data cleaning is particularly one of the most important but time-consuming tasks of data scientists [27] as they spend 60% of their time on this task [25]. Most data cleaning tasks need to incorporate human supervision because the desired data curation might be use-case dependent or subjective. That is why the data cleaning process involves the user in multiple forms of human supervision, such as providing integrity rules, setting statistical parameters, and annotating data values.

1.2 Previous Approaches: The Preconfiguration Paradigm

Existing data cleaning approaches are designed based on the *preconfiguration paradigm*, which requires the user to provide the correct and complete set of integrity rules [22, 20], statistical parameters [93], or both [74]. For example, a data cleaning system might need data patterns, such as the date format "dd.mm.yyyy", or statistical parameters, such as expected mean and standard deviation.

The preconfiguration-based approaches suffer from two general limitations. First, they need the user to provide these input configurations to clean the dataset, accordingly. Providing the correct and complete set of rules and parameters upfront is a major impediment for most non-expert users as they need to know both the dataset and the data cleaning system very well to be able to configure the systems properly [2, 93]. Second, the performance of these approaches heavily depends on the quality of the user-provided rules and parameters. In fact, they cannot achieve high performance unless the user configures them properly for every given dirty dataset.

1.3 Our Approach: The Configuration-Free Paradigm

We propose a novel *configuration-free paradigm* for data cleaning. This paradigm does not require the user to provide any data- or approach-dependent configurations, such as integrity rules or statistical parameters. Instead, our paradigm leverages human supervision in the form of only a few annotated data values.

The configuration-free paradigm is more suitable for three scenarios.

1. The dataset is novel and its data constraints are unknown.
2. The user is a domain expert who is not adept at generating the right rules and parameters for complex data cleaning systems.
3. In addition to providing rules and parameters, the user prefers to also annotate a few data values to further improve the data cleaning performance. In this case, the configuration-free paradigm complements the preconfiguration paradigm.

We design a novel configuration-free approach to detect/correct data errors. The intuition is that, by collecting a set of base error detectors/correctors that can independently mark/fix data errors, we can learn to combine them into a final set of data errors/corrections with a few informative user labels. To this end, we need a set of base error detectors/correctors, a representation method to combine them, a sampling method to select the most informative data values for user labeling, and a learning task to generalize the user-provided error detection/correction operations.

In a nutshell, our approach detects/corrects data errors with a novel *two-step task formulation*. First, each base error detector/corrector generates an initial set of potential

data errors/corrections. This step particularly increases the achievable recall bound of the error detection/correction task. Then, the approach ensembles the output of these base error detectors/correctors into one final set of data errors/corrections in a semi-supervised manner. In fact, the approach iteratively asks the user to annotate a tuple, i.e., marking/fixing a few data errors. The approach learns to generalize the user-provided error detection/correction examples to the rest of the dataset, accordingly. This step particularly preserves high precision of the error detection/correction task.

Our novel two-step formulation of the error detection/correction task has four benefits.

1. The approach is configuration free as it considers the base error detectors/correctors as black-box algorithms that are not necessarily correct or complete. Therefore, the user does not need to configure the base error detectors/correctors upfront.

2. The approach is effective in the error detection and correction tasks as its first and second steps maximize recall and precision, respectively.

3. The approach also minimizes human involvement as it samples the most informative tuples of the dataset for user labeling. That is why the performance of the approach quickly converges with only a few user-provided labels.

4. The task formulation of our approach enables us to leverage historical data cleaning efforts to optimize the current data cleaning task, according to transfer learning [62]. In particular, we can estimate the effectiveness of the base error detectors on the current dataset based on their effectiveness on previously cleaned datasets. Furthermore, we can pretrain the base error correctors on previously cleaned datasets.

1.4 Challenges

We identified two general challenges and multiple subsequent research questions in designing our data cleaning approach.

1.4.1 Optimizing All Data Cleaning Objectives

An ideal data cleaning approach has three main properties: (1) high effectiveness, (2) high efficiency, and (3) low human involvement. The effectiveness itself consists of two sub-properties: (i) being correct and (ii) being complete. Ideally, a data cleaning approach must detect and correct all data errors (completeness) accurately (correctness) in a short time (efficiency) and in an automated manner (human involvement). Although optimizing each of these objectives alone is possible, optimizing all simultaneously is challenging due to the trade-off among them.

Our approach leverages a large set of base error detectors/correctors to clean as many data errors as possible. This approach could create two issues. First, the runtime of the approach is high as it has to run multiple base error detectors/correctors. Second, aggregating these base error detectors/correctors into an accurate final result set is challenging as not all the base algorithms are accurate. To address the first issue, we need to parallelize and prune base error detectors/correctors without harming the result completeness. To address the second issue, we need to design a learning task with minimal human supervision as simple aggregation functions, such as majority voting of the base error detectors/correctors, will lead to false positives [2]. Overall, we need a sophisticated task formulation for data cleaning that maximizes precision and recall and minimizes runtime and user involvement.

In particular, we need to address the following research questions:

- How should we design a set of base error detectors/correctors that can theoretically capture/fix data errors of any types? (Completeness)
- How should we parallelize and prune these base error detectors/correctors for a given dataset to reduce the overall runtime? (Efficiency)
- How should we ensemble the output of these base error detectors/correctors to accurately identify and fix data errors? (Correctness)
- How should we confine all user interventions to just a few tuple annotations and how to find the most informative tuples? (Human involvement)

1.4.2 Transfer Learning on Heterogeneous Datasets

Learning from previous data cleaning experiences enables us to improve the performance of new data cleaning tasks. However, the transfer learning process requires us to capture the similarity of different data cleaning tasks. Two data cleaning tasks are similar if the input dirty datasets to these two tasks are similar. Defining similarity of dirty datasets is not trivial because dirty datasets could be heterogeneously different. Each dirty dataset could have different content, structure, and quality that makes its comparison to other datasets challenging. Therefore, we need to extract and transfer data cleaning knowledge between these heterogeneous datasets.

In particular, we need to answer the following research questions:

- How should we represent a dataset with automatically extractable and domain-independent metadata that describe the dirtiness of the dataset?
- How should we leverage these metadata to define similarity between datasets in terms of data quality issues and required data cleaning treatments?
- How should we leverage the similarity of the current dataset to previous historical datasets to improve the performance of the current data cleaning task?

1.5 Solution Overview: The End-to-End Data Cleaning Pipeline

We have designed an end-to-end data cleaning pipeline based on our two-step data cleaning approach. Figure 1.1 illustrates the high-level architecture of our end-to-end data cleaning pipeline. Given a dirty dataset as input, the goal is to detect and correct data errors and output the cleaned dataset.

To this end, our pipeline leverages three auxiliary data cleaning resources: (1) the user feedback, (2) a data cleaning toolbox, and (3) an optional historical data repository. The user feedback in the form of a few identified and fixed data errors is the only required form of supervision in our data cleaning pipeline. The data cleaning toolbox contains the set of base error detector/corrector algorithms. Optionally, the user can further enrich this default set of base error detectors/correctors with custom dataset-specific algorithms. The historical data repository is an optional resource to boost the error detection/correction performance by learning from previous data cleaning experiences on historical data.

Provided with these data cleaning resources, our end-to-end data cleaning pipeline consists of three main components: (1) the error detection system (named Raha), (2) the error correction system (named Baran), and (3) the transfer learning engine.

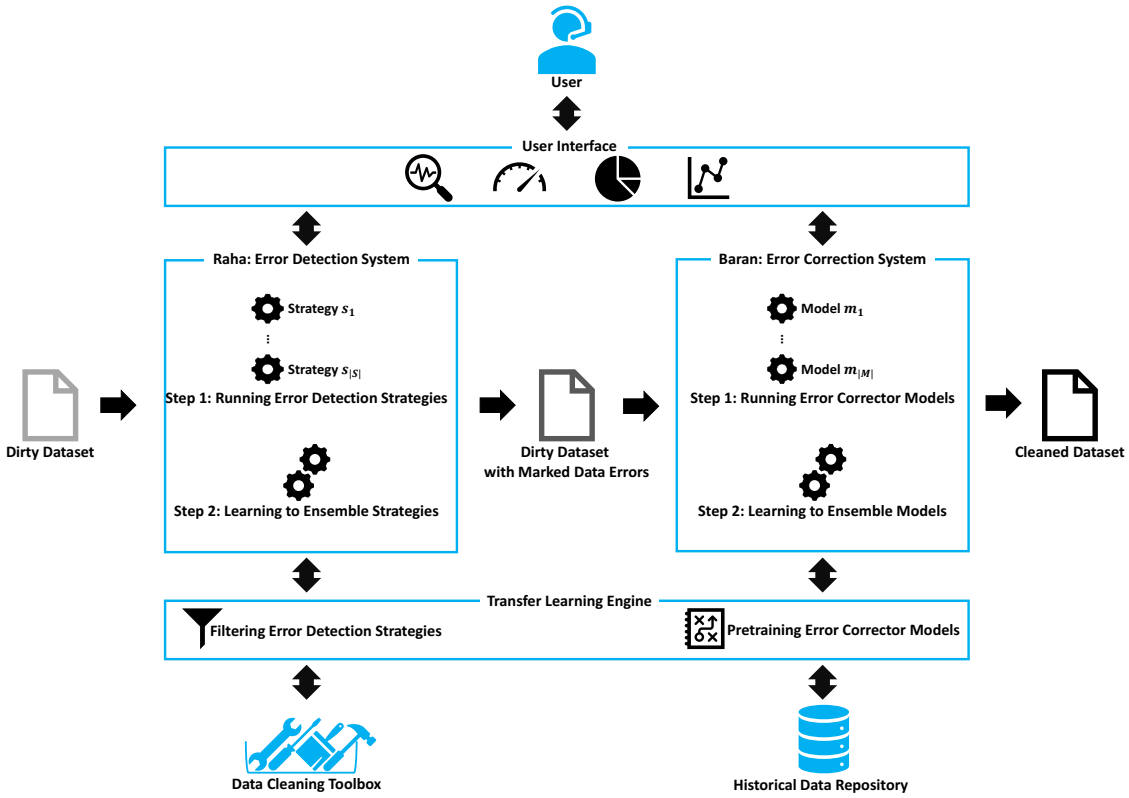


Figure 1.1: The architecture of our end-to-end data cleaning pipeline.

1.5.1 Raha: The Error Detection System

Raha takes the dirty dataset and detects its data errors. Internally, Raha generates and runs a large number of error detection strategies on the dirty dataset. These strategies represent the four main families of traditional error detection algorithms, namely outlier detection, pattern violation detection, rule violation detection, and knowledge base violation detection [2]. Raha collects the output of these error detection strategies to featurize each data cell. The feature vector of each data cell shows which error detection strategies have marked this particular data cell as a data error. Next, Raha clusters data cells of each data column based on the similarity of their feature vectors. Raha iteratively samples a set of tuples that cover as many unlabeled clusters as possible and asks the user to label data cells of these sampled tuples as dirty or clean. Raha propagates each user label to all data cells of the same cluster to boost the number of labeled training data points. Finally, Raha trains one classifier per data column to predict the final label of each data cell inside a data column.

1.5.2 Baran: The Error Correction System

Baran takes the dirty dataset with marked data errors and corrects its previously detected data errors. Internally, Baran trains a set of error corrector models that leverage different contexts of a data error to propose potential corrections. A data error context comprises the value itself, the co-occurring values inside the same tuple, and all values that define the attribute type. Each error corrector model proposes various potential corrections for each data error based on its contexts. Baran featurizes each pair of a data error and a correction candidate by representing the fitness of the correction candidate for the data error. Each component of this feature vector corresponds to the confidence of one error corrector model for replacing the data error with the correction candidate. Then, Baran iteratively samples a set of tuples that

cover various data error types in different data columns and asks the user to fix the marked data errors of these sampled tuples. Baran incrementally updates the error corrector models with these new correction examples. Finally, Baran trains one classifier per data column to predict the actual correction of each data error from the set of all correction candidates.

1.5.3 The Transfer Learning Engine

The transfer learning engine optionally leverages previous data cleaning experiences on historical data to improve the error detection/correction performance of Raha/Baran on the current dataset.

For Raha, the transfer learning engine filters out ineffective error detection strategies to reduce the overall error detection runtime without harming the effectiveness of the error detection process. In fact, the engine calculates the similarity of the current dataset with previously cleaned historical datasets to estimate the effectiveness of error detection strategies based on their effectiveness on similar historical datasets. This way, the engine identifies and filters out the ineffective error detection strategies before Raha spends computational resources to run them.

For Baran, the transfer learning engine pretrains error corrector models to reduce the required dataset-specific user labels on the current dataset. To pretrain the models, the engine extracts correction examples from any general-purpose revision data history, such as the Wikipedia page revision history. Since these pretrained error corrector models learn common typos and mistakes, such as wrong usage of the value "Holland" instead of the value "Netherlands", they generate many correction candidates for the current dataset. This prior set of correction candidates improves the achievable recall bound of Baran.

1.6 Contributions

We make the following contributions:

- **Novel two-step task formulation.** We propose a novel two-step formulation for the error detection and correction tasks, which is effective and efficient, and requires minimal user involvement. Our approach is effective as its first and second steps maximize recall and precision, respectively. Furthermore, our approach displays a competitive runtime due to its parallelization and pruning techniques. Finally, our approach involves the user minimally to annotate a few informative data values due to its effective feature representation and tuple sampling method.
- **Configuration-free data cleaning systems.** We propose two new configuration-free error detection and error correction systems, Raha and Baran. Our configuration-free systems detect/correct data errors without requiring any user-provided data- or system-dependent configurations.
- **Comprehensive set of base error detectors and correctors.** We propose a comprehensive set of base error detectors and correctors that leverage all data error contexts to detect and correct various data error types.
- **Effective feature representations.** We propose effective feature vectors for error detection and correction tasks. In particular, each feature vector combines the output of base error detectors/corrctors to represent data quality issues and required data cleaning treatments.
- **Tuple sampling methods.** We propose two tuple sampling methods for the error detection and correction tasks to sample the most informative tuples for user annotation.

Our clustering-based sampling method clusters similar data cells, samples tuples that cover unlabeled clusters, and propagates each user label through its corresponding cluster to boost the number of labeled training data points. Our informativeness-based sampling method samples those tuples whose erroneous data cells are more informative for the classifier of underlabeled data columns.

- **Transfer learning methods.** We propose transfer leaning methods to learn from previous data cleaning experiences on historical data. In particular, we estimate the effectiveness of error detection strategies based on their effectiveness on similar historical datasets. Furthermore, we pretrain value-based error corrector models based on the extracted value-based corrections from the Wikipedia page revision history.

- **Extensive experiments.** We conducted extensive experiments to evaluate our data cleaning systems in terms of effectiveness, efficiency, and human involvement. As our experiments show, Raha and Baran significantly outperform 10 recent data cleaning systems on 8 well-known datasets. Our experiments particularly show that our two-step approach achieves high precision and recall together with low runtime and user involvement. Furthermore, our experiments show that, the more base error detectors/correctors we use, the higher performance our approach can achieve as we can represent data errors/corrections more effectively.

1.7 Impact of Thesis Contributions

The impact of the thesis contributions consists of several research publications and open source projects.

1.7.1 Research Publications

We have already published the research of this thesis in the following peer-reviewed publications:

1. **Raha: A Configuration-Free Error Detection System [57]**
   Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang
   SIGMOD 2019 (pages 865–882)
   Note 1: This paper won the **ACM SIGMOD most reproducible award**.
   Note 2: Mohammad Mahdavi, as the only PhD student among the co-authors, was the main responsible person for designing the approach and the only responsible person for conducting the experiments.

2. **Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning [54]**
   Mohammad Mahdavi and Ziawasch Abedjan
   PVLDB 2020 (pages 1948–1961)

3. **REDS: Estimating the Performance of Error Detection Strategies Based on Dirtiness Profiles [55]**
   Mohammad Mahdavi and Ziawasch Abedjan
   SSDBM 2019 (pages 193–196)

4. **Semi-Supervised Data Cleaning with Raha and Baran [56]**
   Mohammad Mahdavi and Ziawasch Abedjan
   CIDR 2021

Furthermore, the author also collaborated on the following peer-reviewed related publications:

1. **ED2: A Case for Active Learning in Error Detection [60]**
   Felix Neutatz, **Mohammad Mahdavi**, and Ziawasch Abedjan
   CIKM 2019 (pages 2249–2252)

2. **CLRL: Feature Engineering for Cross-Language Record Linkage [16]**
   Öykü Özlem Çakal, **Mohammad Mahdavi**, and Ziawasch Abedjan
   EDBT 2019 (pages 678–681)

3. **Towards Automated Data Cleaning Workflows [58]**
   Mohammad Mahdavi, Felix Neutatz, Larysa Visengeriyeva, and Ziawasch Abedjan
   LWDA 2019 (pages 10–19)

4. **Data Science für alle: Grundlagen der Datenprogrammierung [1]**
   Ziawasch Abedjan, Hagen Anuth, Mahdi Esmailoghli, **Mohammad Mahdavi**, Felix Neutatz, and Binger Chen
   Informatik Spektrum 2020 (pages 1–8)

1.7.2 Open Source Projects

We have released all the projects implemented based on this thesis under the Apache License 2.0 in the following repositories:

1. `https://github.com/bigdama/raha`
   This repository contains the implementation and documentation of our error detection and error correction systems, Raha and Baran, including their transfer learning methods. The repository also contains our non-proprietary datasets and benchmark scripts to reproduce the experimental results provided in the original research papers. Furthermore, the repository contains interactive Jupyter Notebooks to build end-to-end data cleaning pipelines with Raha and Baran.

2. `https://github.com/bigdama/reds`
   This repository contains the implementation and documentation of our error detection performance estimator system, REDS, including our non-proprietary datasets. In fact, REDS is the first prototype of our transfer learning engine. We created a separate repository for REDS as, in contrast to Raha and Baran, REDS focuses on the narrow task of estimating the performance of error detection strategies.

1.8 Outline

The rest of this thesis is structured as follows. We first review foundations in Chapter 2. We then review related work in Chapter 3. Next, we detail our error detection system in Chapter 4. We then elaborate our error correction system in Chapter 5. Next, we detail our transfer learning methods in Chapter 6. We then experimentally evaluate our proposed approach in Chapter 7. Next, we demonstrate our data cleaning systems in Chapter 8. Finally, we conclude the discussions and provide future directions in Chapter 9.

2

Foundations

We review the foundational concepts of our work in this chapter. We first introduce the input dataset to our approach and our assumptions on it. We then formally define data errors that our approach aims at detecting/correcting. Next, we review the core machine learning and data profiling techniques that our approach is built on. Finally, we summarize the chapter.

2.1 Dataset

The input data to our data cleaning approach could be any relational dataset. More formally, let $d = \{t_1, t_2, ..., t_{|d|}\}$ be a relational dataset of size $|d|$, where each t_i denotes a tuple. Let $A = \{a_1, a_2, ..., a_{|A|}\}$ be the schema of the dataset d, with $|A|$ attributes. Let $d[i, j]$ be the data cell in the tuple t_i of the dataset d and the attribute a_j of the schema A.

We denote the ground truth (i.e., the cleaned version) of the same dataset as d^*. We assume that the dataset d and its ground truth d^* have the same size, i.e., they both have $|d|$ data rows and $|A|$ data columns. In fact, we do not allow to add or remove tuples during the data cleaning process and focus on marking and fixing the existing data values, in accordance with literature [2, 74].

Example 1 (Dataset). Table 2.1 shows a small dirty dataset d and its ground truth d^*. □

2.2 Data Error

Data errors are those data values inside a dataset that deviate from the actual ground truth [2]. Every data cell $d[i, j]$ that is different from the corresponding data cell in the ground truth $d^*[i, j]$ is considered to be a data error. We denote the set of data errors of the dataset d as $E = \{d[i, j] \mid d[i, j] \neq d^*[i, j]\}$.

Example 2 (Data error). Considering the dirty dataset d in Table 2.1, the data cells $d[3, 2] =$ "", $d[4, 2] =$ "", $d[5, 2] =$ "123", and $d[6, 2] =$ "Shire" are erroneous according to the ground truth d^*, i.e., $E = \{d[3, 2], d[4, 2], d[5, 2], d[6, 2]\}$. □

Table 2.1: A dirty dataset d with marked data errors (left) and its ground truth d^* (right).

ID	Lord	Kingdom
1	Aragorn	Gondor
2	Sauron	Mordor
3	Gandalf	
4	Saruman	
5	Elrond	123
6	Théoden	Shire

ID	Lord	Kingdom
1	Aragorn	Gondor
2	Sauron	Mordor
3	Gandalf	N/A
4	Saruman	Isengard
5	Elrond	Rivendell
6	Théoden	Rohan

2.2.1 Data Error Categories

Data errors have been traditionally categorized in different ways. A previous work categorized data errors based on the way they were introduced into the data, such as missing values, typos, formatting issues, and violated attribute dependencies [72]. Another work categorized data errors based on the error detection strategies that identify them, such as outliers, pattern violations, and rule violations [2].

We unify these taxonomies by distinguishing between *data error categories* and *data error types*. We categorize data errors into *syntactic* and *semantic* errors. Syntactic errors are those data values that do not conform to the syntax of the correct values. Semantic errors are those values that, although are syntactically correct, appear in the wrong context. Furthermore, we consider prevalent data error types in literature, such as missing values, typos, formatting issues, and violated attribute dependencies [72]. We aim at detecting/correcting both major syntactic and semantic error categories and the prevalent data error types.

Example 3 (**Data error categories**)**.** Considering the dirty dataset d in Table 2.1, the numerical data value $d[5, 2] =$ "123" is a syntactic error because it does not fit the syntax of any kingdom name. The data value $d[6, 2] =$ "Shire" is a semantic error because, although it is a syntactically correct kingdom in "The Lord of the Rings", the correct value for this data cell is a different kingdom, namely "Rohan". □

2.2.2 Data Error Contexts

Every data error has three contexts that can be leveraged to detect and correct the data error: its value, its vicinity, and its domain.

2.2.2.1 Value Context

Some data errors can be detected and corrected by only taking the data value itself into consideration. In this case, we can identify wrongly formatted values and transform them into the correct value. The first data error context is the value of the data error, i.e., $e_{val} = d[i, j]$.

Example 4 (**Value context**)**.** Suppose an erroneous data cell has the value "16/11/1990". We can identify this data error as it does not comply the desired predefined date format "dd.mm.yyyy". We can also fix this data error by transforming it into the correct format, i.e., "16.11.1990". □

2.2.2.2 Vicinity Context

The detection and correction of some data errors requires information on their vicinity, i.e., information about other clean data values in the same data row. The second data error context contains all the other clean values inside the same tuple, i.e., the vicinity of the data error $e_{vic} = d[i, :]$.

Example 5 (**Vicinity context**)**.** Suppose an erroneous data cell in the data column *Capital* has the value "Paris". We cannot identify/fix this data error because "Paris" is not an erroneous capital name on its own. But, if we check its clean neighboring data values in the same tuple and observe "Germany" in the data column *Country*, then we can change "Paris" to "Berlin" to make it consistent with its vicinity. □

2.2.2.3 Domain Context

Identifying and fixing some data errors needs domain information, i.e., information about other clean data values inside the same data column. The third data error context contains all the other clean values of the same data column, i.e., the domain of the data error $e_{dom} = d[:, j]$.

Example 6 (**Domain context**)**.** Suppose we have an outlying temperature value in the data column *Temperature*. We can use the distribution of other clean values inside this data column to identify this erroneous value and impute the correct value. □

As the above examples show, each data error context can be leveraged in an entirely different way to detect/correct a data error. These detection/correction procedures are not easy to integrate although they can independently lead to the same results.

2.3 Machine Learning

Our data cleaning approach leverages machine learning techniques to optimize the data cleaning process. In particular, Raha and Baran are semi-supervised systems that possess active learning-based sampling methods and can optionally benefit from transfer learning methods.

2.3.1 Semi-Supervised Learning

Machine learning tasks can be categorized into three branches: unsupervised, supervised, and semi-supervised learning [17, 84]. Unsupervised learning is the task of finding structure in unlabeled data points. Supervised learning is the task of learning a function given a set of labeled data points. Semi-supervised learning, which lies between unsupervised and supervised learning, is the task of learning a function given a large number of unlabeled and a small set of labeled data points. Semi-supervised classification approaches are particularly relevant to scenarios where labeled data is scarce [84]. Other data cleaning approaches have been also designed in a semi-supervised manner [51, 60].

We design our data cleaning approach in a semi-supervised manner because obtaining user annotations for the data cleaning tasks is time consuming and expensive. In fact, we usually do not have a given set of annotated data values upfront for a new dirty dataset to train supervised models. On the other hand, we cannot clean data effectively without any human supervision as the unsupervised models can only detect/correct objective data error types, such as typos. That is why Raha and Baran start the data cleaning process with no user labels and ask the user to label a few data points on the fly in a semi-supervised manner.

2.3.2 Active Learning

Active learning is a technique to improve the performance of a machine learning model by allowing it to ask the label of those particular new data points that the model needs to know [78]. Hence, active learning is a natural fit for semi-supervised approaches, which have to

learn a function with a limited number of labeled training data points. Using active learning, the semi-supervised approaches can converge faster by asking the label of a selective sample of data points from the user. Other data cleaning approaches have leveraged active learning as well [94, 51, 60].

We design the tuple sampling methods of our data cleaning approach according to active learning to sample the most informative tuples for user annotation. As a result, the performance of Raha and Baran quickly converges with only a few user-annotated tuples.

2.3.3 Transfer Learning

Transfer learning is the act of gaining knowledge from one task and then applying this knowledge to a different but related task [62]. A natural fit for transfer learning is when training data is scarce and expensive in one domain while it is widely available in a similar domain. In this case, the learning model can be pretrained on the related domain and then be fine-tuned on the current dataset [28]. Other data integration approaches have leveraged transfer learning for entity matching [96, 15] and missing value imputation [88] tasks.

We design our data cleaning approach in a way that enables us to optionally leverage transfer learning. In particular, we estimate the effectiveness of the base error detectors and pretrain the base error correctors via historical data, if available. To the best of our knowledge, Raha and Baran are the first data cleaning systems that can benefit from transfer learning in the error detection and correction tasks.

2.4 Data Profiling

Data profiling is the task of generating metadata for a given dataset [3]. Many data profiling algorithms have been designed to generate various metadata, such as functional dependencies [10], conditional functional dependencies [30], and denial constraints [19]. Other data cleaning approaches leverage these metadata as integrity rules for rule-based data cleaning [22, 20].

Since we design our data cleaning approach configuration free, we do not need the user to discover and input genuine integrity rules to our approach. In fact, Raha and Baran do not require the internal base error detectors/correctors to be configured. We instead leverage data profiling for featurizing and representing the data.

We particularly leverage functional dependencies [3] as base error detectors/correctors in Raha and Baran. Let X and Y be two subsets of the attributes A of the dataset d, i.e., $X, Y \subset A$. A functional dependency $X \rightarrow Y$ states that any two tuples that agree on the left-hand-side attributes X must also agree on the right-hand-side attributes Y. In fact, for each pair of tuples $i_1 \neq i_2 \in d$, if $d[i_1, X] = d[i_2, X]$, then $d[i_1, Y] = d[i_2, Y]$. Therefore, a base error detector can mark as data errors all the left-hand-side and right-hand-side values $d[i_1, X]$, $d[i_2, X]$, $d[i_1, Y]$, and $d[i_2, Y]$ that violate a functional dependency $X \rightarrow Y$. Furthermore, a base error corrector can also propose to replace these marked values with other clean values in the dataset that satisfy the functional dependency $X \rightarrow Y$.

2.5 Summary

We reviewed the preliminary concepts. In particular, we elaborated our definitions and assumptions on datasets and data errors. We also explained the necessary machine learning and data profiling techniques.

3

Related Work

Our research in this thesis is related to three major research areas. We first review related research in data cleaning as addressing the data cleaning problem is the main focus of our work. We then discuss data transformation approaches to clarify their connections to our approach. Next, we differentiate our research problems from their similar counterparts in natural language processing. Finally, we summarize the chapter.

3.1 Data Cleaning

Recent data cleaning approaches can be categorized based on different criteria as shown in Table 3.1. A data cleaning approach might address the error detection task [2, 41, 89], the error correction task [20, 93, 74], or both [22, 21]. It might work as a single-strategy approach [93, 21] or as an aggregator that internally aggregates multiple base data cleaning approaches [68, 2, 74, 86]. A data cleaning approach might leverage only internal signals from the dataset itself [93, 67] or additionally incorporate external data sources [21, 39, 74]. A data cleaning approach might be unsupervised [45, 89], semi supervised [60], or supervised. It might leverage human supervision in the form of integrity rules [22, 20, 74], statistical parameters [9, 93, 67, 74], or data annotations [86, 41]. A data cleaning approach might leverage human supervision in only one user interaction [67, 74] or in an interactive manner [94, 87, 40, 51, 60]. Finally, a data cleaning approach might need data- or approach-dependent configurations [22, 20, 74] or it might be configuration free.

We can position our data cleaning approach using our data cleaning systems, Raha and Baran. Raha and Baran address both the error detection and error correction tasks, respectively. They are both data cleaning aggregators as they internally ensemble multiple base error detectors/correctors. Raha and Baran leverage both the dataset itself together with external data sources. They are both semi-supervised data cleaning systems that use human supervision interactively in the form of only a few annotated data values. Raha and Baran are configuration free, which means they do not need any data- or system-dependent configurations from the user.

We next categorize the most prominent recent data cleaning approaches based on the first criteria, i.e., the data cleaning task, and compare our approach with them.

Table 3.1: Different categorizations of data cleaning approaches.

Criteria	Categories
Data Cleaning Task	Error Detection [2, 41, 89] Error Correction [20, 93, 74] End to End [22, 21]
Aggregation Ability	Single Strategy [93, 21] Aggregator [68, 2, 74, 86]
Data Source Usage	Bound to Internal Signals [93, 67] Able to Leverage External Signals [21, 39, 74]
Human Supervision	Unsupervised [45, 89] Semi Supervised [60] Supervised
Human Supervision Form	Rule Based [22, 20, 74] Parameter Based [9, 93, 67, 74] Annotation Based [86, 41]
Human Supervision Frequency	Single Touch [67, 74] Interactive [94, 87, 40, 51, 60]
Configuration Usage	Configuration Based [22, 20, 74] Configuration Free

3.1.1 Error Detection Approaches

Traditional error detection approaches detect data errors with qualitative or quantitative heuristics. Qualitative rule-based approaches, such as NADEEF [22], take a set of user-provided integrity rules in the form of denial constraints and detect data errors accordingly. Quantitative statistical approaches, such as dBoost [67], take a set of user-provided statistical parameters (i.e., thresholds) to detect outlying data errors.

The performance of all these approaches heavily depends on the quality of the user-provided rules and parameters as they directly enforce these rules and parameters on the data. As a result, without a correct and complete set of integrity rules and statistical parameters, these approaches cannot achieve high precision and recall. This dependency is a major impediment for non-expert users as they need to spend a lot of time to discover useful integrity rules and accurate statistical parameters for each dataset. In fact, they have to run data profiling systems [63] or consult with metadata taxonomies [85] to collect these configurations for the dataset at hand.

Our data cleaning approach does not need any user-provided rules or parameters as it is configuration free. In fact, our two-step formulation of the error detection task allows us to incorporate many base error detectors that are not necessarily correct or complete. Hence, we do not need to discover useful integrity rules and accurate statistical parameters to configure the base error detectors. Raha incorporates each available error detection algorithm with a large set of possible configurations as base error detectors. This way, Raha learns error detection rules itself by combining these base error detector signals.

Error detection aggregators, such as min-k and maximum entropy-based order selection [2], aggregate multiple base error detectors via simple aggregation functions, such as majority voting or precision-based ordering. The aggregators detect data errors more effectively than their base error detectors as aggregation of multiple base error detectors naturally improves the overall precision and recall of the error detection task.

Despite the promise of the aggregation idea, these approaches are limited in terms of their aggregation functions. Their simple aggregation functions, such as majority voting of the base error detectors, can work effectively only if the base error detectors are accurate. Therefore, when the base error detectors generate false positives due to their input configurations,

aggregating them with a simple function, such as majority voting, will not achieve high precision either.

Our data cleaning approach aggregates base error detectors with a learning-based aggregation method. In fact, our two-step formulation of the error detection task allows us to learn the importance of each particular base error detector using a few user labels. As a result, Raha achieves high precision by training classifiers that combine the outputs of base error detectors into one final set of data errors.

Unsupervised error detection approaches, such as Auto-Detect [45] and Uni-Detect [89], aim at detecting data errors in an unsupervised manner. These approaches automatically detect a small portion of data errors as achieving "any recall for free is better than no recall" [89].

Most data cleaning tasks need to incorporate human supervision because the desired data curation might be use-case dependent or subjective. Without any human supervision, these unsupervised error detection approaches can only detect objective data error types, such as typos, which are recognized as data errors by any user in any use case. Therefore, the unsupervised approaches are not expressive enough to detect various real-world data errors accurately.

Our data cleaning approach uses human supervision in a semi-supervised manner to be able to identify any data error types according to user's preferences. In fact, Raha leverages human supervision in the form of a few user labels to detect data errors.

State-of-the-art error detection approaches, such as metadata driven [86] and HoloDetect [41], learn to detect data errors in a semi-supervised manner. Metadata driven learns to detect data errors using the output of error detection tools and metadata features. HoloDetect detects data errors by learning from synthetically generated labeled training data.

Despite the promise, these approaches involve the user a lot due to two main limitations. First, similar to the traditional approaches, these learning-based approaches need the user to provide a correct and complete set of integrity rules, statistical parameters, or both. Metadata Driven needs the user to configure its underlying error detection tools accurately. HoloDetect leverages user-provided data constraints to construct a data representation model. While it is a tedious task for non-expert users to provide these input configurations, the performance of these approaches heavily depends on the quality of these input configurations. Second, they need a large number of labeled training data points that scales with the size of dataset, i.e., $1\% - 10\%$ of the dataset [86, 41]. The reason is that these approaches randomly sample tuples for user labeling as they do not have any data cleaning-specific tuple sampling methods. This user labeling task is particularly tedious in the case of HoloDetect as this approach requires the user to both mark and fix data errors. In fact, HoloDetect needs the correction of user-labeled data errors as well to train an error generator model for generating more training data, according to the data augmentation technique.

Our data cleaning approach minimizes user involvement as it does not suffer from these limitations. First, as mentioned, our approach is configuration free and does not need any user-provided rules or parameters. Second, due to our effective feature representation and tuple sampling methods, the required number of user labels of our approach scales with the number of data error types of dataset, i.e., $10 - 20$ labeled tuples. In fact, this number of labeled tuples is enough for our approach to learn prevalent data error types introduced in Section 2.2. Raha detects data errors effectively using only a few labeled tuples as it represents each data cell with a large and expressive feature vector and samples the most informative tuples for user labeling.

3.1.2 Error Correction Approaches

Traditional error correction approaches, such as Holistic [20] and LLUNATIC [34], are rule based. They take a set of user-provided integrity rules in the form of denial constraints and correct data errors accordingly.

Similar to the traditional error detection approaches, the performance of these rule-based error correction approaches heavily depends on the quality of the user-provided rules as they directly enforce these rules on the data to make the dataset consistent. Therefore, these approaches suffer from the same limitations as they need the user to provide a correct and complete set of integrity rules.

Our configuration-free data cleaning approach does not need any user-provided rules or parameters. As mentioned, our two-step formulation of the error correction task allows us to incorporate many base error correctors that are not necessarily correct or complete. Our error correction system, Baran, incorporates various base error correctors based on different data error contexts. This way, Baran learns error correction rules itself by using these base error corrector signals and a few user labels.

Interactive error correction approaches, such as GDR [94] and Falcon [40], incorporate continuous user feedback to conduct the data cleaning process. Therefore, the user does not have to start the data cleaning process with a complete set of configurations and the user can incrementally update them during the process.

Although continuous user feedback improves the error correction performance, these approaches do not make the best use of it. While GDR uses user feedback to choose the actual correction from a set of potential correction updates, it still needs a given set of integrity rules to generate these correction candidates. Hence, it also suffers from the same downsides of the rule-based approaches, i.e., it relies on user-provided integrity rules. Falcon is confined to SQL-like update queries and cannot learn more complicated correction operations from user feedback.

Our data cleaning approach incorporates interactive user feedback in an example-driven way, taking examples of data errors and their correction from the user. Furthermore, it learns various error correction operations using the value, the vicinity, and the domain contexts of data errors. In particular, Baran learns four types of value-based correction operations in contrast to the simple SQL-like update queries of Falcon.

State-of-the-art error correction approaches, such as SCARE [93] and HoloClean [74], learn to correct data errors. SCARE learns to correct data errors in different data partitions based on statistical likelihoods. HoloClean trains a graphical model to correct data errors with respect to integrity rules, matching dependencies, and statistical signals.

These approaches suffer from two main limitations. First, similar to the traditional approaches, these learning-based approaches need the user to provide a correct and complete set of integrity rules, statistical parameters, or both. While it is a tedious task for non-expert users to provide these input configurations, the performance of these approaches heavily depends on the quality of these input configurations. Second, these approaches heavily depend on data redundancy as they reduce the data cleaning problem to the task of "finding a consistent permutation of data values". In fact, their idea is to minimally swap data values in the dataset with respect to a cost function, such as the number of value changes, until the dataset becomes consistent with respect to the integrity rules and statistical likelihoods. Therefore, the performance of these approaches not only depends on the correctness and completeness of the input rule/parameter set, but also on the amount of redundancy in data. Without duplicate tuples that provide the actual clean value of a data error in the active domain, without correlated data columns that connect data values across domains, and without

a correct and complete set of predefined user-given rules and parameters to define consistency on top of redundant data, these approaches fail to achieve both high precision and recall.

Our data cleaning approach addresses these limitations. First, our configuration-free approach does not need any user-provided rules or parameters. Second, besides leveraging data redundancy to correct data errors, our approach further learns to generate correction candidates using user-provided annotations and historical data. In particular, Baran leverages user-provided corrections and value updates in historical datasets to generate additional correction candidates besides clean values of the dataset.

3.1.3 End-to-End Approaches

Traditional end-to-end data cleaning approaches both detect and correct data errors with qualitative heuristics, such as NADEEF [22], or quantitative heuristics, such as DEC [9]. They check the conformity of data to user-provided rules and parameters, detect inconsistencies, and correct them with minimal changes. Hence, they also depend on the quality of the user-provided rules and parameters and cannot be effective without a correct and complete set of configurations.

A group of data cleaning approaches, such as KATARA [21] and Detective Rules [39], can use out-of-dataset signals from a master dataset, such as the DBpedia knowledge base [8], to detect and correct data errors of the dataset at hand. In fact, they match the dataset at hand to an external data source and detect/correct conflicting data errors in the dataset at hand. This way, they can reduce the required user feedback on the dataset at hand as the external master data might automatically provide some relevant data errors/corrections for the dataset.

Despite the promising idea, these approaches are limited due to their matching methods. The simple matching operation is too naive as a master dataset might not match to all parts of the dataset at hand or it might match wrongly to some data parts due to the ambiguity in concepts. As a result, these approaches cannot achieve high error detection/correction precision and recall.

Our data cleaning approach uses external data sources through learning-based models to avoid wrong mismatches. Using historical datasets as training data, our approach learns to predict the effectiveness of base error detectors on the dataset at hand. This way, it can identify the most effective base error detectors for the dataset at hand without matching datasets. Furthermore, our approach learns the value-based corrections observed in historical data upfront to generate more correction candidates on the dataset at hand. Therefore, both Raha and Baran can leverage transfer learning to transfer error detection/correction knowledge from historical data to the dataset at hand.

There are also other data integration/cleaning approaches that can be considered relevant to our research. Hands-off entity matching approaches, such as Corleone [35], Falcon [24], and CloudMatcher [37], are also configuration free. However, we propose the first configuration-free approach to address the data cleaning problem, which is a more challenging task than entity matching in the absence of user-provided configurations. Machine learning-tailored data cleaning approaches, such as ActiveClean [51], BoostClean [50], and AlphaClean [52], also clean datasets. However, these approaches are not general-purpose data cleaning systems as they rely on a downstream machine learning task to evaluate and guide the data cleaning process. In contrast, our general-purpose data cleaning approach can clean datasets regardless of the downstream applications.

3.2 Data Transformation

Data transformation is the task of transforming data values from one format into another [6]. A data transformation could be syntactic, such as transforming the value "1 m" to "100 cm", or semantic, such as transforming the value "Holland" to "Netherlands" [4]. Programming by example is an approach to address the data transformation problem. Programming by example is the task of synthesizing a program that satisfies a set of input-output examples [38]. Other data integration approaches have leveraged the programming by example paradigm for string transformation [80] and data wrangling [47, 81] tasks.

Our data cleaning approach leverages data transformation and programming by example as one type of error correction, i.e., the value-based correction. In particular, Baran possesses various value-based error corrector models to learn both syntactic and semantic value transformations from user-provided examples.

3.3 Natural Language Processing

Our research problems might seem similar to well-known problems in natural language processing. In particular, data preprocessing seems similar to text preprocessing and error detection/correction seems similar to spell checking/fixing. We clarify the differences and explain why similar natural language processing approaches are not necessarily applicable to the data cleaning tasks.

3.3.1 Text Preprocessing

Text preprocessing is the task of representing unstructured texts in a unified way by multiple techniques, such as Unicode normalization, tokenization, stemming, lemmatization, and stop word removal [77]. Text preprocessing is an essential step in unstructured text processing pipelines as it reduces the vocabulary size by text unification. As a result, machine learning models will avoid out-of-vocabulary words in new text corpora.

Data preprocessing for structured data is the counterpart task of text preprocessing that similarly aims to represent the data more effectively. However, the mentioned text preprocessing techniques do not address the structured data preprocessing needs due to the nature of structured datasets. Structured datasets are intended to store a limited number of unique values in a well-structured format. Thus, having the right value in the right spot of the dataset is more important than vocabulary size reduction. Therefore, the standard text preprocessing approaches are not applicable and we need dedicated data preprocessing approaches for structured data.

3.3.2 Spell Checking and Fixing

Spell checking/fixing is the task of detecting/correcting objective data error types, such as typos and grammar mistakes, in texts [66]. Data quality issues in natural language processing are confined to these objective data error types as defining more specific data quality requirements for unstructured text corpora is not applicable. Spell checking/fixing approaches leverage contextual words in the same sentence to detect/correct a typo or grammar issue.

Data cleaning aims at detecting/correcting various data error types in structured datasets based on user's preferences. Applying typical spell checking/fixing tools from natural language processing is not enough to detect/correct all data error types in structured datasets for two reasons. First, they cannot detect/correct subjective data error types, such as violated attribute

dependencies and formatting issues [72], which do not fit user's specific preferences. Second, they cannot effectively detect/correct objective data errors because contextual information in structured data is not necessarily as rich as unstructured text corpora. In fact, while in unstructured text a typo can be identified and fixed using its surrounding contextual words in the same sentence, the neighboring data values in a structured dataset could be irrelevant to each other. Therefore, identifying/fixing even objective data error types is harder in structured datasets and requires us to redefine the context of a data value for structured data.

3.4 Summary

We reviewed the related research to our work. We first categorized and reviewed the most prominent recent data cleaning approaches. We then clarified the connection of data transformation to our work. Finally, we distinguished our research problems from their similar counterparts in natural language processing.

4

Raha: The Error Detection System

Error detection is the task of identifying data values that are wrong. Although there are many error detection algorithms in literature [2], detecting all data errors accurately with minimum user involvement is not trivial as these error detection algorithms need the user to provide a correct and complete set of data- or algorithm-specific configurations.

Problem 1 (**Error detection**)**.** Given as input a dirty dataset d, a set of available error detection algorithms $B = \{b_1, b_2, ..., b_{|B|}\}$ that require configurations, and a user with a labeling budget θ_{Labels} to annotate tuples, the goal is to identify all data errors within d, i.e., $E = \{d[i,j] \mid d[i,j] \neq d^*[i,j]\}$. □

Raha leverages a novel two-step task formulation to achieve both high error detection precision and recall using a few user labels. The intuition is that, by collecting a set of base error detectors that can independently mark data errors, we can learn to combine them into a final set of data errors using a few informative user labels. First, Raha automatically configures these error detection algorithms and leverages their output to represent data quality issues of data cells. Then, Raha learns to detect data errors using this data representation and a few user labels.

Figure 4.1 illustrates the workflow of Raha. Given as input the dirty dataset, the data cleaning toolbox, and the user feedback, Raha outputs the set of data errors through the following steps.

Step 1: Configuring error detection algorithms. Raha systematically configures each existing algorithm to generate a set of error detection strategies. The set of error detection strategies should be automatically generatable and comprehensive enough to detect various data error types. We detail this step in Section 4.1.

Step 2: Running error detection strategies. Raha runs the error detection strategies on the dataset. Each strategy marks a set of data cells as data errors. We detail this step in Section 4.2.

Step 3: Generating feature vectors. Raha generates a feature vector for each data cell by collecting the output of all the error detection strategies. Each element in the feature vector of a data cell is a binary flag that shows whether a particular strategy marks this data cell as a data error or not. We detail this step in Section 4.2.

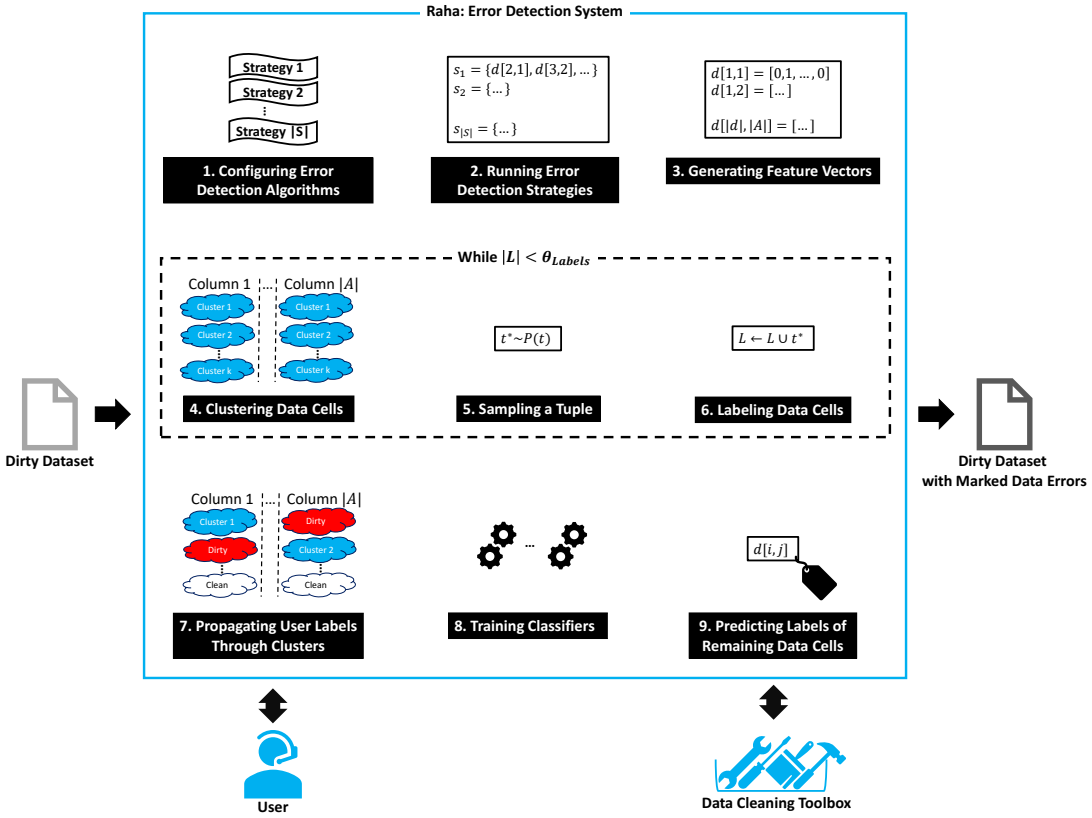


Figure 4.1: The workflow of Raha.

Step 4: Clustering data cells. Raha clusters the data cells of each data column in each iteration, based on the similarity of their feature vectors. The number of clusters per data column needs to be set automatically according to the number of existing data error types. We detail this step in Section 4.3.

Step 5: Sampling a tuple. Raha samples a tuple in each iteration to be labeled by the user. Thus, the total number of iterations is bound by the user labeling budget θ_{Labels}. Since each data column has a separate set of clusters, the sampled tuple should ideally cover as many unlabeled clusters as possible over all the data columns. We detail this step in Section 4.3.

Step 6: Labeling data cells. Raha asks the user to label data cells of the sampled tuple as dirty or clean. Raha iteratively repeats the steps 4, 5, and 6 until the user labeling budget θ_{Labels} is depleted.

Step 7: Propagating user labels through clusters. Raha propagates the user labels through the clusters. These propagated labels are noisy as they have not been verified by the user. We detail this step in Section 4.4.

Step 8: Training classifiers. Raha trains a classifier per data column based on the feature vectors of data cells and the propagated labels, i.e., user labels and noisy labels. We detail this step in Section 4.4.

Step 9: Predicting labels of remaining data cells. The trained classifiers are applied to predict the labels of the remaining unlabeled data cells.

Algorithm 1 also shows the main steps of Raha in pseudocode. Raha first configures error detection algorithms (line 1). Raha then generates the feature vectors (line 2). Next, Raha iteratively clusters data cells and samples tuples for user labeling (lines 3–11). Finally, it propagates the user labels through the clusters, trains a set of classifiers, and applies them to predict the label of unlabeled data cells (lines 12–16).

Algorithm 1: Raha(d, B, θ_{Labels}).

Input: dataset d, set of error detection algorithms B, user labeling budget θ_{Labels}.
Output: set of data errors E.

$S \leftarrow$ generate error detection strategies by automatically configuring all the algorithms $b \in B$;
$V \leftarrow$ generate feature vectors by running all the strategies $s \in S$ on the dataset d;
$k \leftarrow 2$; `// number of clusters per data column`
$L \leftarrow \{\}$; `// set of user labeled tuples`
while $|L| < \theta_{Labels}$ **do**
    for *each data column* $j \in [1, |A|]$ **do**
        $\psi_j \leftarrow$ cluster data cells of the data column j into k clusters;
    $t^* \leftarrow$ draw a tuple with the probability proportional to $P(t)$;
    ask the user to label the tuple t^*;
    $L \leftarrow L \cup \{t^*\}$;
    $k \leftarrow k + 1$;
$E \leftarrow \{\}$; `// set of data errors`
for *each data column* $j \in [1, |A|]$ **do**
    $L'_j \leftarrow$ propagate the user labels through the clusters ψ_j;
    $\phi_j \leftarrow$ train a classifier with the feature vectors V_j and the propagated labels L'_j;
    $E \leftarrow E \cup$ apply the classifier ϕ_j to the feature vectors V_j;

We first elaborate the error detection strategies. We next explain the feature generation process. Then, we detail the tuple sampling and labeling method. Next, we elaborate the label propagation and classification step. Finally, we summarize the chapter.

4.1 Error Detection Strategies

Previous research proposed many error detection algorithms [67, 47, 22, 21] that can be categorized into four families: outlier detection, pattern violation detection, rule violation detection, and knowledge base violation detection algorithms [2].

Each error detection algorithm needs to be configured based on the characteristics of the given dataset. Depending on the error detection algorithm, the configuration might include statistical parameters, such as expected mean and standard deviation; patterns, such as the desired date format "dd.mm.yyyy"; specification of rules, such as the functional dependency $ZIP \rightarrow City$; or providing links to reference datasets, such as DBpedia [8].

We consider each combination of an algorithm and a configuration as one distinct error detection strategy. In fact, an error detection strategy is any configured algorithm that can mark data cells of a dataset as data errors based on a logic. More formally, the set of error detection strategies is $S = \{s = (b, g) \mid b \in B, g \in G_b\}$, where B is the set of error detection algorithms and G_b is the set of finite/infinite space of different configurations of an error detection algorithm $b \in B$.

Our set of base error detectors incorporates all the four families of error detection strategies. This way, we leverage all data error contexts to detect all data error categories. Outlier and pattern violation detection strategies leverage the value and domain contexts of data errors to detect mainly syntactic errors. Rule and knowledge base violation detection strategies leverage the vicinity context of data errors to detect mainly semantic errors. The user can optionally enrich the default set of base error detectors by adding new error detection strategies.

These error detection algorithms take as input either continuous numerical parameters or discrete nominal parameters. For algorithms with numerical parameters, such as outlier detectors, we quantize the continuous range of the parameters. For algorithms with infinite nominal parameters, such as patterns, rules, and knowledge base violation detectors, we identify heuristics to effectively limit the space of parameters. Any other error detection algorithm that does not require any of such parameters, i.e., a black box, can only be used with a single configuration.

Of course, only a subset of the generated error detection strategies may effectively mark data errors on a given dataset and most of them might be imprecise. Nevertheless, as long as each strategy marks data cells using the same logic, Raha can use the output of the strategy as a notion of similarity for comparing data cells.

4.1.1 Outlier Detection Strategies

Outlier detection algorithms [67] assess the correctness of data values in terms of compatibility with the general distribution of values that reside inside the data column. Identification of outliers depends on the type of data values. To detect string outliers, we need to compare the frequency of the string data values. To detect numerical outliers, we need to compare the magnitude of the numerical data values. Thus, we leverage two fundamental Histogram- and Gaussian-based outlier detection algorithms [67] that leverage the occurrence and magnitude of data values, respectively.

A histogram-based strategy builds a histogram distribution based on the frequency of data values in a particular data column. The strategy $s_{\theta_{\text{tf}}}$ marks data cells from the rare bins as data errors, i.e., data cells with a normalized term frequency smaller than a threshold $\theta_{\text{tf}} \in (0, 1)$. Formally,

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

where $TF(d[i,j])$ is the term frequency of the data cell $d[i,j]$ inside the data column j.

Raha generates 9 histogram-based outlier detection strategies by setting the threshold $\theta_{\text{tf}} \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$. In fact, Raha divides the threshold range $\theta_{\text{tf}} \in (0, 1)$ into 0.1 intervals as, in practice, we notice that finer granular thresholds (e.g., $\theta_{\text{tf}} = 0.15$) do not create more distinctive outlier detection strategies in comparison to the coarser granular selection of thresholds.

A Gaussian-based strategy builds a Gaussian distribution based on the magnitude of the numerical values in a particular data column. The strategy $s_{\theta_{\text{dist}}}$ marks as data errors those numerical data cells whose normalized distance to the mean is larger than a threshold $\theta_{\text{dist}} \in (0, \infty)$. Formally,

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

where μ_j is the mean and σ_j is the standard deviation of the numerical data column j.

Raha generates 9 Gaussian-based outlier detection strategies by setting the threshold $\theta_{\text{dist}} \in \{1, 1.3, 1.5, 1.7, 2, 2.3, 2.5, 2.7, 3\}$, according to the *68-95-99.7 rule* [83].

Example 7 (**Outlier detection strategies**)**.** On the dataset d in Table 4.1, the output of two histogram-based outlier detection strategies over the attribute *Kingdom* would be $s_{o1} = \{d[1,2], d[2,2], d[5,2], d[6,2]\}$ by setting $\theta_{\text{tf}} = \frac{2}{6}$ and $s_{o2} = \{d[1,2], d[2,2], d[3,2], d[4,2], d[5,2], d[6,2]\}$ by setting $\theta_{\text{tf}} = \frac{3}{6}$. □

4.1.2 Pattern Violation Detection Strategies

Pattern violation detection algorithms [47] assess the correctness of data values in terms of compatibility with predefined data patterns. In fact, they mark data values that do not match a certain data pattern.

Ideally, we could have domain-specific data patterns for each given dirty dataset. For example, to detect data errors in an IT-related dataset, the user provides a set of regular

Table 4.1: A dirty dataset d with marked data errors (left) and its ground truth d^* (right).

ID	Lord	Kingdom
1	Aragorn	Gondor
2	Sauron	Mordor
3	Gandalf	
4	Saruman	
5	Elrond	123
6	Théoden	Shire

ID	Lord	Kingdom
1	Aragorn	Gondor
2	Sauron	Mordor
3	Gandalf	N/A
4	Saruman	Isengard
5	Elrond	Rivendell
6	Théoden	Rohan

expressions to identify valid and invalid *URLs* and *email addresses*. This way, we can accurately detect data errors as these domain-specific data patterns are designed for the dirty dataset at hand. However, we do not want to involve the user to provide these configurations for each dataset. Therefore, we need a general data pattern representation that can theoretically capture any domain-specific data pattern to not hold any assumption on the data domain.

We leverage the *bag-of-characters representation* [77], which is a general representation for encoding all possible data patterns, i.e., character combinations. This representation can also encode the length and type of data values as it also shows which and how many distinct characters appear in data values.

We generate a set of character checker strategies s_{ch} to check the existence of each character *ch* in data cells. For each character *ch* in the set of all characters in a data column j, the strategy s_{ch} marks a data cell $d[i, j]$ as data error if the data cell contains character *ch*. Formally,

$$s_{\text{ch}}(d[i,j]) = \begin{cases} 1, & \text{iff} \quad d[i,j] \text{ contains } ch; \\ 0, & \text{otherwise.} \end{cases}$$

Overall, we have $\bigcup_{j=1}^{|A|} |\Sigma_j|$ pattern violation detection strategies in our system, where $|A|$ is the number of data columns and Σ_j is the set of all distinct characters appearing in the data column j.

Example 8 (**Pattern violation detection strategies**)**.** On the dataset d in Table 4.1, the output of two pattern violation detection strategies over the attribute *Kingdom* would be $s_{p1} = \{d[1,2], d[2,2]\}$ by setting $ch =$ "o" and $s_{p2} = \{d[5,2]\}$ by setting $ch =$ "1". □

4.1.3 Rule Violation Detection Strategies

Rule violation detection algorithms [22] assess the correctness of data values based on their conformity to integrity rules, such as not being null or being unique. Since the single-column rules, such as value range and length, are implicitly covered by the outlier and pattern violation detection algorithms, we include here only rule violation detection strategies that check inter-column dependencies. In particular, we focus on rules in the form of functional dependencies [3].

We limit the scope of functional dependencies to only those with a single attribute on their left-hand side, in order to reasonably limit the exponential space of all possible functional dependencies. This way, we also avoid potentially unintended dependencies that arise by considering large sets of data columns. As discussed in literature [64], most interesting functional dependencies involve only a few attributes. As we do not know upfront which functional dependencies are useful, we consider all pairs of data columns as potential functional dependencies. With this approach, we are also covering partial functional dependency relationships that were unknown to the user.

For each pair of data columns $\forall j_1 \neq j_2 \in [1, |A|]$, the strategy $s_{j_1 \to j_2}$ marks all data cells $d[i, j]$ that violate the functional dependency $j_1 \to j_2$. Formally,

$$s_{j_1 \to j_2}(d[i,j]) = \begin{cases} 1, & \text{iff} \quad d[i,j] \text{ violates } j_1 \to j_2; \\ 0, & \text{otherwise.} \end{cases}$$

The number of rule violation detection strategies is $|A| \times (|A| - 1)$ as we consider the functional dependencies from and to each attribute.

Example 9 (**Rule violation detection strategies**)**.** On the dataset d in Table 4.1, the output of two rule violation detection strategies over the attribute *Kingdom* would be $s_{r1} = \{\}$ by checking the functional dependency *Lord* $\to$ *Kingdom* and $s_{r2} = \{d[3,2], d[4,2]\}$ by checking the functional dependency *Kingdom* $\to$ *Lord*. □

4.1.4 Knowledge Base Violation Detection Strategies

Knowledge base violation detection algorithms [21] assess the correctness of data values by cross-checking them with data within a knowledge base, such as DBpedia [8].

Knowledge bases contain rich information about the world's entities and their mutual relationships [79]. These knowledge bases are constructed by extracting high-quality structured data from unstructured information [26]. The data inside a knowledge base is usually stored in the form of entity relationships, such as *City isCapitalOf Country*. Here, *City* and *Country* are entity types and *isCapitalOf* is a relationship. The algorithm tries to match each side of a relationship to different data columns in the dataset. If there are two data columns that are matched to both sides of the relationship (e.g., *City* and *Country*), the algorithm marks data values in the matched data columns that contradict the entity relationship inside the knowledge base. Therefore, the knowledge base violation detection algorithms can also identify data errors that violate inter-column dependencies.

For each relationship r inside the knowledge base, the strategy s_r marks all data cells $d[i, j]$ that violate the relationship. Formally,

$$s_r(d[i,j]) = \begin{cases} 1, & \text{iff} \quad d[i,j] \text{ violates } r; \\ 0, & \text{otherwise.} \end{cases}$$

Overall, we have as many knowledge base violation detection strategies as there are relationships inside the DBpedia knowledge base [8], i.e., 2064 strategies.

Example 10 (**Knowledge base violation detection strategies**)**.** On the dataset d in Table 4.1, the output of two knowledge base violation detection strategies over the attribute *Kingdom* would be $s_{k1} = \{d[3,2], d[4,2], d[5,2], d[6,2]\}$ by setting the entity relationship to *Lord isKingOf Kingdom* and $s_{k2} = \{\}$ by setting the entity relationship to *City isCapitalOf Country*. □

4.2 Feature Vector Generation

The mentioned error detection strategies mark data cells as data errors based on different intuitions. Therefore, we can describe data quality issues of a data cell by collecting the output of all the strategies on that particular data cell.

Raha maps each data cell to a feature vector that is composed of the output of error detection strategies. Having a set of error detection strategies $S = \{s_1, s_2, ..., s_{|S|}\}$, Raha runs

Table 4.2: Featurizing data cells of the data column *Kingdom*.

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

each strategy $s \in S$ on the dataset d. Each strategy s either marks a data cell $d[i, j]$ as a data error or not. Formally, Raha stores this information as

$$s(d[i,j]) = \begin{cases} 1, & \text{iff} \quad s \text{ marks } d[i,j] \text{ as a data error;} \\ 0, & \text{otherwise.} \end{cases}$$

The feature vector of the data cell $d[i, j]$ is the vector of all the outputs of the error detection strategies $s \in S$ on this data cell. Formally,

$$v(d[i,j]) = [s(d[i,j]) \mid \forall s \in S]. \tag{4.1}$$

Hence, the set of feature vectors of data cells $v(d[i, j])$ inside a particular data column j is

$$V_j = \{v(d[i,j]) \mid i \in [1, |d|]\}. \tag{4.2}$$

Raha post-processes the feature vectors of each data column V_j to remove non-informative features that are constant for all the data cells of the data column.

Example 11 (**Feature vector generation**)**.** We introduced two configurations for each family of error detection algorithms in Section 4.1. This would result into $|S| = |B \times G_b| = 4 \times 2 = 8$ strategies: $S = \{s_{o1}, s_{o2}, s_{p1}, s_{p2}, s_{r1}, s_{r2}, s_{k1}, s_{k2}\}$. As mentioned, each of these strategies marks a set of data cells as data errors in our running example.

Table 4.2 shows the feature vectors that Raha generates for each data cell in the data column *Kingdom*. For example, the feature vector of the data cell $d[1, 2]$ = "Gondor" is $[1, 1, 1, 0, 0, 0, 0, 0]$, as only the strategies s_{o1}, s_{o2}, and s_{p1} mark this data cell as a data error. Raha removes the features s_{o2}, s_{r1}, and s_{k2} in the post-processing phase as these features are constant for all data cells of the data column *Kingdom*.

We can identify similar clean/dirty data cells as the feature vectors represent data quality issues of data cells. For example, the clean data cells $d[1, 2]$ = "Gondor" and $d[2, 2]$ = "Mordor" have the same feature representation. □

4.3 Tuple Sampling and Labeling

After designing a feature representation, we need to incorporate human supervision to learn various data error types. In particular, we need to ask the user to label a sampled subset of data cells as clean or dirty.

A straightforward sampling method is to uniformly pick n random tuples from the dataset [86, 41]. Therefore, we will have $n \times |A|$ labeled data cells for training classifiers. However, this method is not effective on datasets with low data error rates due to the class imbalance ratio of clean and dirty data cells. In fact, since the number of dirty data cells is much smaller than the number of clean data cells, a small set of uniformly sampled data cells cannot cover various data error types.

Raha follows a clustering-based sampling method to sample a small set of tuples that cover various data error types across all data columns. Raha first clusters data cells of each data column and then samples tuples that cover mostly the unlabeled clusters.

4.3.1 Clustering Data Cells

Using our expressive feature vector, we can boost the number of labeled data cells per data column using the *cluster assumption*, which states that two data points are likely to have the same class label if they belong to one cluster [18]. Therefore, we can cluster data cells and assign the same dirty/clean label to all data cells in each cluster.

Raha builds a separate clustering model for each data column as data values are better comparable within their own domain. Setting the number of clusters k per data column is particularly challenging. Smaller k's yield bigger clusters that are more likely to contain a mix of dirty and clean data cells. Inversely, bigger k's will lead to more clusters, requiring more and potentially unnecessary labels from the user. Although there are clustering algorithms that can automatically choose the number of clusters, such as DBSCAN [29], they need other more non-intuitive input parameters, such as minimum/maximum distances between data points of different clusters.

Raha builds a hierarchical agglomerative clustering model [5] per data column, which allows a flexible specification of the number of clusters. Raha starts with only two clusters per data column and then increases the number of clusters in each iteration. Its iterative process lets the user terminate the clustering-based sampling process at will and in accordance with the labeling budget. As the choice of the similarity metric and the linkage method does not affect Raha's performance, we use the default cosine similarity metric and the average linkage method.

4.3.2 Tuple Selection

So far, Raha clusters data cells of each data column independent of clustered data cells of the other data columns. We can ask the user to label a data cell per cluster. However, the user typically needs to check the whole tuple to label one data cell. Thus, it is more intuitive to sample entire tuples for user labeling than individual data cells per data column.

Finding a minimum set of tuples that covers all the unlabeled clusters is the classical *set cover problem* [48], which is NP-complete. In every clustering iteration, each data column is divided into k clusters of data cells. Some of these clusters are unlabeled, i.e., none of their data cells has been labeled. Ideally, the sampled tuples should cover all unlabeled clusters from each data column. This way, labeling the sampled tuples leads to labeling all the unlabeled data cells.

To address this NP-complete problem, we design an approximate method with two properties. First, we relax the challenge of finding a minimum set of tuples by selecting only one tuple in each iteration. This way, we avoid the challenge of finding tuples that do not cover the same set of clusters. Second, instead of deterministically selecting the tuple that covers the most number of unlabeled clusters, we probabilistically select this tuple. This way, we avoid getting stuck in local optima as probabilistic solutions of this problem have been shown to be more resilient than deterministic greedy heuristics against local optima [31].

Therefore, Raha draws a tuple t^* in each iteration based on the softmax probability function

$$P(t) = \frac{\exp(\sum_{c \in t} \exp(-N_c))}{\sum_{t' \in d} \exp(\sum_{c \in t'} \exp(-N_c))}, \tag{4.3}$$

Table 4.3: Clustering data cells of the data column *Kingdom*.

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

where N_c is the number of user-labeled data cells in the current cluster of data cell c. This scoring formula benefits tuples whose data cells mostly belong to the clusters that have received fewer user labels.

The user takes one sampled tuple t^* in each iteration and labels its data cells as clean or dirty. At the end of each iteration, we have the set of user labeled tuples $L \subset d$. This iterative procedure is repeated in the next iterations with a larger number of clusters $k \in \{2, 3, ...\}$ as long as the labeling budget of the user is not depleted, i.e., $|L| < \theta_{\text{Labels}}$. At the end of the tuple sampling and labeling process, we have $k = \theta_{\text{Labels}} + 1$ clusters per data column and $|L| = \theta_{\text{Labels}}$ labeled tuples.

The proposed clustering-based sampling scheme has two characteristics. First, since the sampling method iteratively clusters and labels underlabeled clusters, the expectation is that, at some point, we will cluster each data column into homogeneously clean or dirty clusters, respectively. In other words, if there is a certain type of data errors inside one data column, the hierarchical clustering method eventually identifies its corresponding cluster. Furthermore, the sampling method addresses the natural class imbalance issue as well, because the rare dirty labels will be propagated through the corresponding clusters.

Example 12 (**Tuple sampling and labeling**)**.** Table 4.3 shows the second iteration of the clustering-based sampling on our running example, when Raha clusters the data cells inside the data column *Kingdom* into 3 groups: a green, a blue, and a purple cluster.

Suppose Raha already sampled the tuple t_1 in the first and the tuple t_3 in this iteration. These two tuples cover the green and the blue clusters over the data column *Kingdom*. Ideally, these two tuples should cover two different clusters in the other data column of the dataset, i.e., the data column *Lord*.

The user labels data cells of these two tuples. In particular, the user labels the data cells $d[1, 1]$, $d[1, 2]$, and $d[3, 1]$ as clean and the data cell $d[3, 2]$ as dirty. □

4.4 Label Propagation and Classification

In semi-supervised settings like ours, having more user labels leads to faster convergence of the models [82]. We thus leverage the clusters to boost the number of labels, according to the *cluster assumption* [18]. Since we cluster data cells based on the expressive feature vectors, the data cells inside one particular cluster are likely to have the same dirty/clean label. Therefore, we can propagate user labels through the clusters.

Raha propagates the user label of each data cell to all data cells in its cluster. Let $L' = \{d[i, j] \mid i \in L, j \in [1, |A|]\}$ be the set of labeled data cells, i.e., all data cells of the labeled tuples L. All the unlabeled data cells $d[i', j'] \notin L'$ that are inside the same cluster of a labeled data cell $d[i, j] \in L'$ get the same dirty/clean label of data cell $d[i, j]$. This way, the number of training labels L' increases significantly.

Since a cluster might have multiple user-labeled data cells with contradicting dirty/clean labels, we need a way to resolve such contradictions. We investigate two conflict resolution functions to propagate user labels in clusters.

1. **Homogeneity-based conflict resolution function.** The *homogeneity-based function* propagates user labels only in clusters that do not contain two data cells with contradicting labels. This method works under the assumption of perfect user labels, in which case, the contradicting user labels inside one cluster indicate that the cluster is not homogeneous enough for label propagation.

2. **Majority-based conflict resolution function.** The *majority-based function* propagates the user labels in clusters with mixed labels as well if a label class has the majority. The intuition is that a homogeneous cluster could also have contradicting user labels due to a user labeling error.

Raha then trains a classifier ϕ_j per data column j to predict the labels for the unlabeled data cells in the same data column. In the training phase, each classifier ϕ_j takes the feature vectors of data cells inside the data column V_j along with the labeled data cells L'. In the prediction phase, each classifier ϕ_j predicts the label of all data cells in the data column j that are not labeled by the user.

We train one classifier for each data column because, similar to the clustering per data column, data cells are better comparable inside their own domain. Although we train a classifier per data column, the classifier can still consider inter-column dependency violations due to the rule and knowledge base violation detection features.

Example 13 (**Label propagation and classification**)**.** Considering the clusters and the labeled data cells of the data column *Kingdom* in Table 4.3, Raha propagates the user labels: The data cell $d[2, 2]$ gets a noisy clean label and the data cell $d[4, 2]$ gets a noisy dirty label due to the user-labeled data cells $d[1, 2]$ and $d[3, 2]$, respectively.

Therefore, the corresponding classifier of the data column *Kingdom* is trained on the first four labeled data points and predicts the label of the last two unlabeled data points. □

4.5 Summary

Raha is a a novel error detection system that relieves the user from the tedious task of selecting and configuring error detection algorithms. Raha systematically generates a wide range of error detection strategies and encodes their output into a feature vector for each data cell. Raha then clusters data cells of each data column and samples those tuples that cover mostly unlabeled clusters. Asking the user to label a few sampled tuples, Raha propagates the user labels through the clusters to boost the number of labeled data points. Finally, Raha trains and applies a classifier per data column to label the rest of unlabeled data cells.

Our novel two-step formulation of the error detection task allows Raha to achieve both high precision and recall using only a few user labels. First, the error detection strategies generate many data error candidates that increase the achievable recall bound of Raha. Then, Raha incorporates human supervision using a few labeled tuples to accurately identify the actual data errors among the set of all data error candidates.

5

Baran: The Error Correction System

Error correction is the task of fixing the detected data errors. Correcting all data errors accurately with a minimum user involvement is not trivial due to the trade-off among correctness, completeness, and automation.

Problem 2 (**Error correction**)**.** Given as input a dirty dataset d, the set of detected data errors E, the set of error corrector models M, and a user labeling budget θ_{Labels} to annotate tuples, the goal is to correct all the detected data errors. □

Baran leverages a novel two-step task formulation to achieve both high error correction precision and recall using a few user labels. The intuition is that, by collecting a set of base error correctors that can independently fix data errors, we can learn to combine them into a final set of corrections using a few informative user labels. First, Baran collects all correction candidates proposed by a set of base error correctors and represents the fitness of each correction candidate for each data error. Then, Baran learns to find the actual correction of each data error using these correction candidate representations and a few user labels.

Figure 5.1 illustrates the workflow of Baran. Given as input a dirty dataset with marked data errors, the data cleaning toolbox, and the user feedback, Baran fixes data errors in the input dirty dataset and returns a cleaned version of the dataset through the following steps.

Step 1: Initializing error corrector models. Baran initializes the error corrector models of the toolbox based on the structural characteristics of the dataset. The models have to learn the co-occurrences of the clean values in data rows and columns. If the error corrector models have been already pretrained, Baran incrementally updates them. Otherwise, Baran trains the error corrector models from scratch on the current dataset. We detail this step in Section 5.1.

Steps 2: Sampling a tuple. Baran samples a tuple in each iteration to be labeled by the user. Thus, the total number of iterations is bound by the user labeling budget θ_{Labels}. To optimally leverage the limited number of user labels, the set of sampled tuples should cover as many data error types as possible across all data columns of the dataset. We detail this step in Section 5.2.

Steps 3: Labeling data cells. Baran asks the user to fix the marked data errors in the sampled tuple.

Step 4: Fine-tuning error corrector models. Baran updates the error corrector models based on the user-corrected data errors. We need to define a unified model for all different

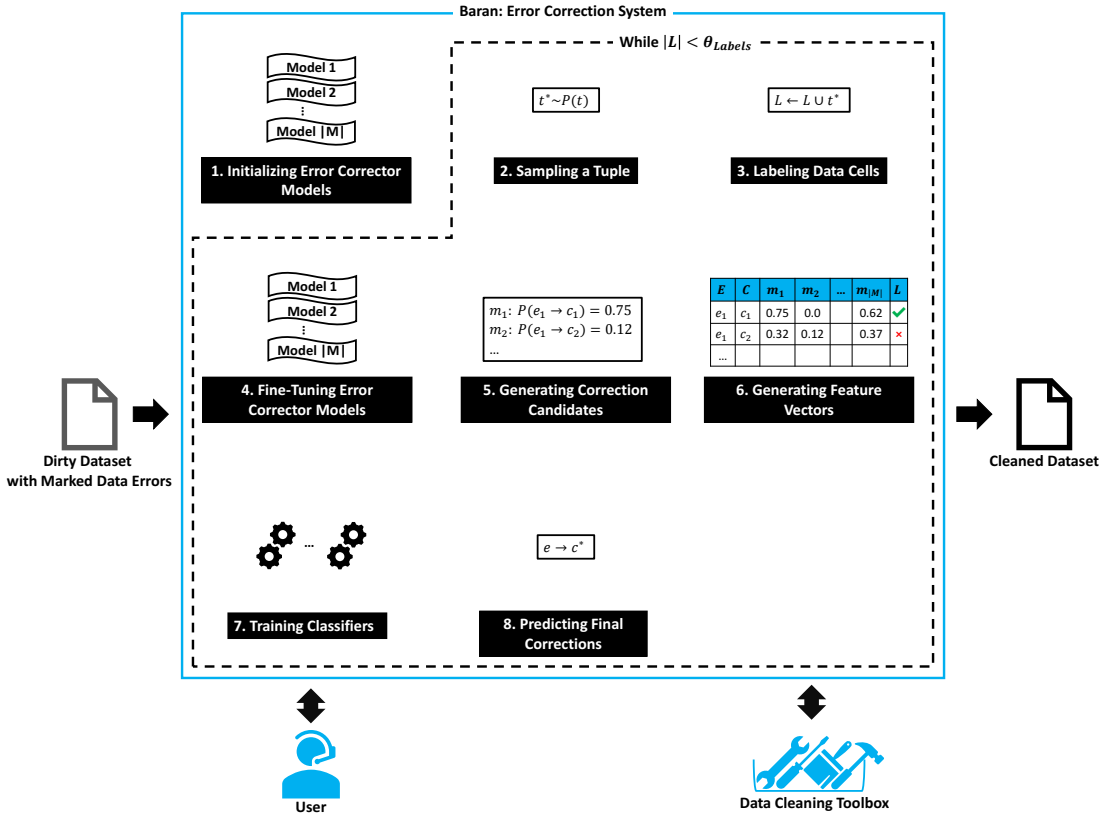


Figure 5.1: The workflow of Baran.

error corrector models so that we can incrementally update all of them in the same way with every new user-corrected data error. We detail this step in Section 5.1.

Steps 5: Generating correction candidates. Each error corrector model proposes various correction candidates for each data error. We need to effectively and efficiently process all these correction candidates for all these data errors. We detail this step in Section 5.3.

Steps 6: Generating feature vectors. Baran generates a feature vector for each pair of a data error and a correction candidate. The feature vector has to represent the mutual fitness of one particular correction candidate for one particular data error. We detail this step in Section 5.3.

Steps 7: Training classifiers. Baran trains a binary classifier per data column based on the feature vectors and the user labels. The binary classifier has to decide whether a correction candidate is the actual correction of a data error or not. We detail this step in Section 5.3.

Steps 8: Predicting final corrections. Baran applies the trained classifiers to predict the final correction for the rest of the data errors. Baran iteratively repeats the steps 2–8 until the user labeling budget θ_{Labels} is depleted.

Algorithm 2 also shows the main steps of Baran in pseudocode. Baran first initializes the error corrector models (line 1). Baran then iterates as long as the user labeling budget is not depleted (lines 4–16). In each iteration, Baran first samples and labels a tuple and then updates the error corrector models accordingly (lines 5–8). Next, it generates a feature vector for each pair of a data error and a correction candidate, trains a classifier per data column, and predicts the final correction of data errors (lines 9–16).

We first elaborate the error corrector models. We then explain the tuple sampling and labeling method. Next, we elaborate the feature vector generation and classification step. Finally, we summarize the chapter.

Algorithm 2: Baran(d, E, M, θ_{Labels}).

Input: dataset d, set of data errors E, set of error corrector models M, user labeling budget θ_{Labels}.
Output: cleaned dataset d^*.

$M \leftarrow$ initialize the error corrector models;
$d^* \leftarrow d$; `// the cleaned dataset`
$L \leftarrow \{\}$; `// set of labeled tuples`
while $|L| < \theta_{Labels}$ **do**
  $t^* \leftarrow$ draw a tuple that maximizes the tuple scoring formula $P(t)$;
  ask the user to fix data errors in the tuple t^*;
  $L \leftarrow L \cup t^*$;
  $M \leftarrow$ update the error corrector models with the tuple t^*;
  for *each data column* $j \in [1, |A|]$ **do**
    $V_j \leftarrow \{\}$; `// feature vectors`
    for *each data error* $e \in E_j$ **do**
      $C_e \leftarrow$ set of all correction candidates for the data error e proposed by all the models $m \in M$;
      for *each correction candidate* $c \in C_e$ **do**
        $V_j \leftarrow V_j \cup$ generate a feature vector for the pair (e, c);
    $\phi_j \leftarrow$ train a classifier using feature vectors V_j and labels L;
    $d^* \leftarrow$ apply the classifier ϕ_j to predict the final correction of data errors E_j;

5.1 Error Corrector Models

An error corrector model is any algorithm that can propose correction candidates to a data error based on a logic that uses a data error context. For example, a simple script that rounds float numbers could be an error corrector model. Baran leverages these error corrector models as base error correctors that propose potential correction candidates for each data error.

Ideally, we want to fix all data errors of the dataset accurately, without any user involvement. Therefore, the set of error corrector models should be ideally complete and contain correct and automated base error correctors. In fact, their combination should be able to fix all the data errors (completeness) accurately (correctness) without any user involvement (automation). Choosing and aggregating base error correctors under these three requirements simultaneously is not possible because there is typically a trade-off among them.

For example, a data scientist may assess a dataset and then write a script to transform wrongly formatted date values "dd/mm/yyyy" to the format "dd.mm.yyyy". Although this base error corrector is accurate, it involves the user (not automated) and may not fix other potential data errors (not complete). Therefore, we have to handle the natural trade-off of correctness, completeness, and automation while designing the set of base error correctors.

We address this trade-off by first ignoring the precision of the base error correctors. Each of our error corrector models is designed to leverage one context of a data error to fix it. The error corrector models automatically propose as many potential correction candidates as possible in the first place. This way, we increase the achievable recall bound by automatically generating a set of potential corrections. Of course, many of these potential data corrections might be irrelevant. However, we will avoid false positives later as well, when we train classifiers on top of these automatically proposed correction candidates.

While we propose a default and general set of base error correctors, which together leverage all data error contexts to identify prevalent real-word data error types, the set of base error correctors can be extended with optional custom user-provided algorithms. In particular, the user can optionally implement data constraints in the form of error correctors and incorporate them into our data cleaning toolbox. This way, Baran generalizes the previous data cleaning aggregators [68, 74] as we consider the base error correctors as black boxes.

We design a set of error corrector models, each of which leverages one context of a data error in the form of a heuristic. To keep the error corrector models simple, general, and incrementally updatable, we define an error corrector model m formally as the conditional

probability

$$P(c|e_m) = \frac{\text{count}(c|e_m)}{\text{count}(e_m)}, \tag{5.1}$$

where e_m is a context of the data error e that the model m uses; count(e_m) is the number of times that the data error context e_m is observed; and count($c|e_m$) is the number of times that the context e_m is leveraged to fix the data error e to the correction c. The intuition is that the more often a data error context e_m is leveraged to fix a data error e to a correction c, the more it is likely that the correction candidate c will be the actual correction of the data error e. Note that we can incrementally update this model by storing count($c|e_m$) and count(e_m) for each pair combination of a data error context e_m and a correction candidate c.

We design three types of error corrector models based on the three data error contexts to be able to fix data errors with respect to all their value-based, vicinity-based, and domain-based contextual information. The abstract definition of error corrector models in Equation 5.1 needs to be implemented for each type of error corrector models, accordingly. In fact, the exact identification of the data error context e_m and the correction candidate c depends on the type of each error corrector model.

5.1.1 Value-Based Error Corrector Models

Value-based error corrector models learn to fix data errors e using only the erroneous value itself [38]. A value-based model applies fine granular substring transformations, which are learned from previous training value-based corrections, to erroneous values. Here, the data error e and its context $e_{val} = d[i, j]$ are identical. Whenever an erroneous value e_{val} is corrected to a value e^*_{val}, Baran updates the value-based error corrector models by encoding the erroneous value and its correction operation.

5.1.1.1 Erroneous Value Encoding

Abstracting erroneous values is an essential technique for training value-based error corrector models as it enables us to generalize correction operations to similar data values. There are different levels of abstraction to encode data values, such as abstracting the character category or the string length. We leverage two simple and general encoders to support both syntactic and semantic value-based corrections.

1. **Identity encoder.** The *identity encoder* encodes the data value with its original characters. This encoding is suitable for fixing semantic errors. For example, to fix the erroneous value “Holland” to “Netherlands”, our value-based error corrector models need to see the exact erroneous value “Holland”.

2. **Unicode encoder.** The *Unicode encoder* encodes each character of a data value with its equivalent Unicode category [90]. For example, an uppercase character will be replaced with its category symbol “<Lu>” and a number will be replaced with its category symbol “<Nd>”. This encoding enables the value-based models to learn syntactic error corrections faster by generalizing the syntax of data errors.

Example 14 (**Erroneous value encoding**)**.** Considering the erroneous value e_{val} = “16/11/1990” and the corrected value e^*_{val} = “16.11.1990”, we have two methods to encode this erroneous value.

The identity encoder encodes the erroneous value with its original characters: Whenever an erroneous value is equal to “16/11/1990”, a potential correction operation could be to replace “/” with “.”. Although the identity encoder is effective in terms of precision, its recall is not satisfying as it is not extendable to other erroneous values.

On the other hand, the Unicode encoder encodes this erroneous value by abstracting its characters to their Unicode category: Whenever an erroneous value is in the format "<Nd><Nd><Po><Nd><Nd><Po><Nd><Nd><Nd><Nd>", a potential correction operation could be to replace "/" with ".". The Unicode encoder is more in favor of recall than precision as all the erroneous values with the same format will be mapped to the same encoding. □

5.1.1.2 Correction Operation Encoding

After encoding the erroneous value, we need to encode the required correction operations. In general, there are four kinds of correction operators that can be applied to any erroneous value.

1. **Remover operator.** The *remover operator* removes substrings. For example, if we want to fix the erroneous value "U.S." to the value "US", the remover operator can remove ".".
2. **Adder operator.** The *adder operator* adds substrings. For example, if we want to fix the erroneous value "US" to the value "U.S.", the adder operator can add ".".
3. **Replacer operator.** The *replacer operator* both removes and adds substrings simultaneously. For example, if we want to fix the erroneous value "16/11/1990" to the value "16.11.1990", the replacer operator can remove "/" and, instead of it, add ".".
4. **Swapper operator.** The *swapper operator*, which is a special case of the replacer operator, substitutes the entire erroneous value with another value. While the previous value-based operators are suitable for syntactic errors, the swapper operator is useful for fixing semantic errors. For example, if we want to fix the erroneous value "Holland" to the value "Netherlands", the swapper operator can substitute these data values.

With this set of operators, we can generate any value-based correction. We train value-based models, each of which learns to perform one of these correction operations on an encoded erroneous value. Given a user-corrected data error, Baran calculates the difference of the erroneous value e_{val} and the corrected value e^*_{val} on the character level, according to the diff checking technique [46], and extracts training data for each operator.

Example 15 (Diff checking). Considering the erroneous value e_{val} = "Chris Edward NoLan" and the user-corrected value e^*_{val} = "Christopher Nolan" as the source and target sequences of characters, the output of diff checker algorithm is as follows:

$$\text{diff}(e_{val}, e^*_{val}) = \begin{cases} \texttt{Add "topher" after "Chris".} \\ \texttt{Remove "Edward ".} \\ \texttt{Replace "L" with "l".} \end{cases}$$

Thus, we can update the corresponding value-based models of all the four operators with this user-corrected example. For example, the value-based model with the identity encoder and the adder operator learns to add the substring "topher" after the substring "Chris" in the erroneous value "Chris Edward NoLan". □

To implement the abstract definition of error corrector models in Equation 5.1, a value-based model considers the encoded erroneous value encode(e_{val}) as the context e_m and the correction operation o that has to be applied to this erroneous value as the correction candidate c. Formally,

$$P(c|e_m) = P(o|\text{encode}(e_{val})) = \frac{\text{count}(o|\text{encode}(e_{val}))}{\text{count}(\text{encode}(e_{val}))}. \tag{5.2}$$

Overall, we have $2 \times 4 = 8$ value-based error corrector models because of 2 erroneous value encoders and 4 correction operators.

Table 5.1: A dirty dataset d with marked data errors (left) and its ground truth d^* (right).

ID	Name	Address
1	H	5th Str
2	Hana	-
3	Gandom	7th Street
4	Chris	9th Str

ID	Name	Address
1	Hana	5th Street
2	Hana	5th Street
3	Gandom	7th Street
4	Christopher	9th Street

Example 16 (Value-based error corrector models). Assume that we have a value-based error corrector model equipped with the identity encoder and the swapper operator. Assume that the model encounters the erroneous value "Holland" 5 times in different parts of the dataset, i.e., $\text{count}(\text{encode}(e_{val})) = 5$. Assume that the user fixes this data error to "Netherlands" 4 times, and to "HL" 1 time. Let us call these two swapping operations "Holland" to "Netherlands" and "Holland" to "HL" o_1 and o_2, respectively.

Therefore, this value-based error corrector model proposes two correction candidates "Netherlands" with $P(o_1|\text{encode}(e_{val})) = 0.8$ and "HL" with $P(o_2|\text{encode}(e_{val})) = 0.2$ for any detected data error e that holds the value "Holland". □

5.1.2 Vicinity-Based Error Corrector Models

Vicinity-based error corrector models learn to fix data errors based on data column relationships. A vicinity-based model proposes clean values of the active domain as potential corrections based on their relationship with clean values of other data columns. Similar to the error detection strategies discussed in Section 4.1, we limit all kinds of data column relationships and correlations [3] to functional dependencies that have one attribute on their left-hand side. This way, we reasonably limit the exponential space of all the functional dependencies as these functional dependencies have been known to be more useful for data cleaning [64].

To implement the abstract definition of error corrector models in Equation 5.1, we consider every $j_1 \rightarrow j_2$ to be a functional dependency for each pair of data columns $\forall j_1 \neq j_2 \in [1, |A|]$. For each functional dependency $j_1 \rightarrow j_2$, a vicinity-based model considers the clean co-occurring value $d[i, j_1]$ in the vicinity context $e_{vic} = d[i, :]$ as the context e_m and the clean value $d[i, j_2]$ as the correction candidate c. Formally,

$$P(c|e_m) = P(d[i, j_2]|d[i, j_1]) = \frac{\text{count}(d[i, j_2]|d[i, j_1]))}{\text{count}(d[i, j_1])}. \tag{5.3}$$

This conditional probability shows how often the left-hand-side value $d[i, j_1]$ determines the right-hand-side value $d[i, j_2]$.

Overall, we have $|A| \times (|A| - 1)$ vicinity-based error corrector models as we consider the functional dependencies from and to each attribute.

Example 17 (Vicinity-based error corrector models). Table 5.1 shows a dirty dataset with its already detected data errors marked in red. The goal is to fix these data errors and generate the depicted cleaned dataset d^*.

Considering the functional dependency *Name* → *Address*, a vicinity-based model learns that a name value "Gandom" must always have the address value "7th Street". Thus, the corresponding vicinity-based model proposes one correction candidate $P(\text{"7th Street"}|\text{"Gandom"}) = 1.0$ for any data error e in the data column *Address* whose neighboring value in the neighboring data column *Name* is "Gandom". □

5.1.3 Domain-Based Error Corrector Models

Domain-based error corrector models learn to fix data errors using the existing values inside their data columns. We consider each data column as a domain and propose a set of domain-based error corrector models, accordingly. A domain-based model proposes the most relevant clean values from the active domain as potential corrections. As tuples inside a dataset are generally independent of each other, the order, distance, or neighborhood of values inside a data column does not indicate any relevance. However, the frequency of clean values can be used as a signal to estimate their relevance. More frequent clean values inside a data column are more likely to also be corrections for a data error in the same data column.

To implement the abstract definition of error corrector models in Equation 5.1, a domain-based model considers the domain context $e_{dom} = d[:, j]$ as the context e_m and each clean value inside this domain as the correction candidate c. Formally,

$$P(c|e_m) = P(d[i,j]|e_{dom}) = \frac{\text{count}(d[i,j]|e_{dom})}{\text{count}(e_{dom})}, \tag{5.4}$$

where $\text{count}(e_{dom})$ is the number of clean values and $\text{count}(d[i,j]|e_{dom})$ is the frequency of the clean value $d[i,j]$ in the active domain of the data error. This conditional probability shows the chance of observing a clean value $d[i,j]$ in the data column j.

Overall, we have one domain-based error corrector model per data column, resulting in $|A|$ domain-based models.

Example 18 (Domain-based error corrector models). Considering the clean values of the data column *Name* in Table 5.1, a domain-based model learns that all the clean values have the same probability to be a correction candidate for each data error inside this data column as they all appear just once.

Thus, the corresponding domain-based model proposes two correction candidates "Hana" with $P(\text{"Hana"}|\text{Name}) = 0.5$ and "Gandom" with $P(\text{"Gandom"}|\text{Name}) = 0.5$ for any data error e in the data column *Name*. □

5.2 Tuple Sampling and Labeling

The trained error corrector models generate various potential corrections for any data error using its value, vicinity, and domain contexts. We need to identify the actual correction among all the proposed potential corrections with respect to user's preferences.

Baran incorporates user supervision in the form of a limited number of manual correction examples. It leverages these examples to update all the error corrector models and to train classifiers. To sample tuples for user labeling, Baran follows an iterative procedure. In each iteration, Baran draws a tuple t^* that maximizes the tuple scoring formula

$$t^* = \underset{t \in d}{\text{argmax}} \prod_{d[i,j] \in t \cap E'_j} \exp(\frac{|E'_j|}{|E_j|}) \exp(\frac{\text{count}(d[i,j]|E'_j)}{|E'_j|}), \tag{5.5}$$

where E_j is the set of all data errors in the data column j; E'_j is the set of those data errors in the data column j that have not been fixed yet; and $\text{count}(d[i,j]|E'_j)$ is the number of unfixed data errors in the data column j whose value is exactly $d[i,j]$.

This scoring formula benefits tuples that (1) contain more unfixed data errors, (2) their data errors reside in data columns that have a high number of unfixed data errors, and (3) their erroneous values are frequent among the unfixed data errors. This way, Baran obtains informative labeled data points for the classifiers of underlabeled data columns. Once the user

fixes data errors of the sampled tuple t^*, the value-based, vicinity-based, and domain-based models will be updated accordingly.

Example 19 (**Updating error corrector models**)**.** Assume that Baran samples the first tuple of the dataset in Table 5.1 and the user fixes the data errors in this tuple.

Therefore, the value-based models will be updated with the new example of the erroneous value "5th Str" that is corrected to the value "5th Street". In particular, a value-based model with the Unicode encoder and the adder operator learns to add the substring "eet" after the substring "Str" for all encounters of erroneous values with a similar pattern to the value "5th Str". The corresponding vicinity-based model of the functional dependency *Name* $\rightarrow$ *Address* will be updated with a new association between the value "Hana" and the value "5th Street". Furthermore, the domain-based model of the data column *Address* will be updated as now we have two clean values in this data column. □

5.3 Feature Vector Generation and Classification

We can now leverage the user labels to define a classification task that predicts the final correction of each data error.

A straightforward approach is to define a multiclass classification task, where each correction candidate resembles a target class and the classifier has to choose one of these target classes for each data error. However, formulating this classification task as a multiclass classification leads to the sparsity issue of the feature vector [12]. In the use case at hand, the feature vector has to encode the probabilities of all the error corrector models for each correction candidate. Formally, the feature vector of a data error e would be $v(e) = [P(c|e_m) \mid \forall m \in M, \forall c \in C]$, where M is the set of all the error corrector models and C is the set of all the correction candidates. Thus, the size of the feature vector would scale with the number of correction candidates, while not every correction candidate is relevant for each data error. As a result, there will be many zero elements in the feature vector.

Example 20 (**The sparsity issue of the multiclass classification**)**.** Assume that we have 20 error corrector models, each proposing 100 correction candidates for any data error. In the multiclass classification task, we have to encode all the $20 \times 100 = 2000$ model probabilities into the feature vector of each data error. □

To avoid the sparsity issue of the multiclass classification, we formulate the classification task as a binary decision. The role of the binary classifier is to decide whether a correction candidate is the actual correction of a data error or not. We generate a feature vector that represents the mutual fitness of one particular correction for one particular data error inside a data column. Thus, for any combination of a data error e and a correction candidate c, we collect all the error corrector model probabilities as a feature vector. Formally,

$$v(e, c) = [P(c|e_m) \mid \forall m \in M], \tag{5.6}$$

where M is the set of all the error corrector models. When a feature vector contains mostly close-to-one probabilities (i.e., $P(c|e_m) \approx 1.0$ for most of the models $m \in M$), it is more likely that the correction candidate c is the actual correction of the data error e; Because, in this case, most error corrector models with high confidence propose this correction candidate for this data error.

Hence, the set of feature vectors of data errors inside a particular data column j is

$$V_j = \{v(e, c) \mid \forall e \in E_j, \forall c \in C_e\}, \tag{5.7}$$

where E_j is the set of all the data errors in the data column j and C_e is the set of all correction candidates for a data error e. The number of feature vectors depends on the number of correction candidates that the error corrector models propose.

This feature representation has three benefits in contrast to the feature representation of the multiclass classification task.

1. This feature vector is small and dense. The number of features is equal to the number of error corrector models and the ratio of non-zero features is higher because the considered models are relevant for the pair at hand.

2. This feature representation leads to fast performance convergence with only a few user labels as we can transform one user label into several training data points. Assume that the user fixes the data error e with the correction c^*. Baran extends this one user label into multiple training data points. Naturally, we have a positive data point that indicates the correction $c^* \in C_e$ is the actual correction of the data error e. Furthermore, we have many negative data points that indicate all the corrections $c \in C_e \backslash \{c^*\}$ are not the actual correction of the data error e.

3. This highly imbalanced training set, with a few positive and a large number of negative data points, makes our classifier conservative in predicting a correction. In fact, our classifier is more biased towards the negative class and prevents false positive corrections. That is why the precision of Baran is generally high.

Instead of training one classifier for the whole dataset, we train one binary classifier per data column because data errors, their required correction techniques, and the usefulness of their contexts are better comparable inside their domain. Although Baran trains one classifier per data column, it preserves all inter-column dependency signals via the vicinity-based error corrector models, which are encoded as features for each pair of a data error and a correction candidate. The vicinity-based models propose correction candidates for a data error based on the functional dependency of the given data column with a different data column.

In each iteration, Baran trains all the classifiers and applies each to all data errors of the corresponding data column. The classifiers do not overwrite the user-provided corrections. For each pair of a data error e and a correction candidate c, the corresponding classifier predicts a label with a confidence score. For a data error e, the classifier can determine zero or multiple correction candidates c as the final corrections. If the binary classifier predicts the label 0 for every correction candidate, no final correction will be selected for the corresponding data error. If it predicts the label 1 for multiple correction candidates, Baran selects the correction candidate with the highest confidence score as the final correction. This iterative procedure is repeated as long as the user labeling budgent is not depleted, i.e., $|L| < \theta_{\text{Labels}}$, where $|L|$ is the number of labeled tuples. Baran considers the output of the last iteration as the final system output.

Example 21 (**Feature vector generation and classification**)**.** Considering the data column *Address* in Table 5.1, the following table contains pairs of data errors and correction candidates and their corresponding feature vectors. For brevity, we just demonstrate a few pairs and features. The features are a value-based model with the Unicode encoder and the adder operator ($m_{\text{Unicode+Adder}}$), a vicinity-based model ($m_{\text{Name}\rightarrow\text{Address}}$), and a domain-based model (m_{Address}). The first two pairs are labeled as the user already validated the first tuple of our toy dataset in Example 19.

Error	Correction	$\mathbf{m_{Unicode+Adder}}$	$\mathbf{m_{Name \rightarrow Address}}$	$\mathbf{m_{Address}}$	Label
5th Str	5th Street	1.0**	1.0*	0.5	1
5th Str	7th Street	0.0	0.0	0.5	0
-	5th Street	0.0	1.0*	0.5	
-	7th Street	0.0	0.0	0.5	
9th Str	5th Street	0.0	0.0	0.5	
9th Str	7th Street	0.0	0.0	0.5	
9th Str	9th Street	1.0**	0.0	0.0	

The classifier of the data column *Address* receives these pairs as data points. It trains with the first two labeled data points and then predicts the label of the rest of unlabeled data points.

The classifier predicts the value "5th Street" as the final correction of the erroneous value "-" because of the vicinity-based feature $m_{\text{Name} \rightarrow \text{Address}}$ (marked with *). This model returns the probability of 1.0 for the pair ("-", "5th Street") because, after labeling the tuple 1 and updating the models in Example 19, the vicinity-based model learned the association of the value "Hana" in the data column *Name* and the value "5th Street" in the data column *Address*.

Furthermore, the classifier predicts the value "9th Street" as the final correction of the erroneous value "9th Str" because of the value-based feature $m_{\text{Unicode+Adder}}$ (marked with **). This model returns the probability of 1.0 for the pair ("9th Str", "9th Street") because the data error "5th Str" matches the erroneous value "9th Str" based on the Unicode encoding and the correction candidate "9th Street" is generated with the same adder operator that generates the correction candidate "5th Street" for the erroneous value "5th Str". □

5.4 Summary

Baran is a a novel error correction system that fixes data errors with respect to their value, vicinity, and domain contexts. Using these data error contexts, Baran trains multiple error corrector models that propose various correction candidates for each data error. Baran then samples a few informative tuples for user labeling. Finally, Baran featurizes each pair of a data error and a correction candidate and trains a classifier per data column to predict the final correction of each data error.

Our novel two-step formulation of the error correction task allows Baran to achieve both high precision and recall using only a few user labels. First, the error corrector models generate many correction candidates that increase the achievable recall bound of Baran. Then, Baran incorporates human supervision using a few labeled tuples to accurately identify the actual correction of each data error among the set of all correction candidates.

6

Data Cleaning Optimization: The Transfer Learning Engine

Although a dataset can be cleaned by itself, it is desirable to learn from previous data cleaning experiences on historical datasets to optimize the current data cleaning task. However, learning from previous data cleaning tasks is challenging as the dirty datasets could be heterogeneously different. In particular, identifying the similarities of these heterogeneous datasets in terms of their data quality issues and required data cleaning treatments is not trivial.

We have designed our data cleaning approach in a way that it can optionally optimize the current data cleaning task by learning from previous data cleaning efforts. In particular, both Raha and Baran can conduct transfer learning to leverage historical datasets in the first step of our two-step task formulation. In error detection, Raha leverages previously cleaned datasets to estimate the effectiveness of error detection strategies on the current dataset. In error correction, Baran leverages previously cleaned datasets to pretrain error corrector models for the current dataset.

We first elaborate our method to estimate the effectiveness of error detection strategies. Then, we detail our method to pretrain error corrector models. Finally, we summarize the chapter.

6.1 Estimating the Effectiveness of Error Detection Strategies

Error detection aggregators internally combine multiple base error detection strategies. Estimating the effectiveness of these base error detection strategies is useful as we can remove ineffective strategies for a given dataset. Filtering out ineffective error detection strategies is beneficial in two different ways. First, it improves the overall effectiveness of existing error detection aggregators, such as maximum entropy-based approach [2], as their overall effectiveness is dependent on the effectiveness of their base error detection strategies. Second, filtering out ineffective base strategies upfront improves the overall efficiency of Raha as it can stop running the irrelevant base error detection strategies on the current dataset.

A straightforward approach is to run all the error detection strategies on the dirty dataset and evaluate their effectiveness on a data sample [2]. This way, the user can select only the most effective error detection strategies for the error detection aggregator. However, this approach

Table 6.1: The features of the dirtiness profile.

Category	Name	Encoding
Content Features	Most Frequent Word Values of Data Columns	One Hot
	Most Frequent Cell Values of Data Columns	One Hot
Structure Features	Fraction of Unique Cell Values	Float
	Fraction of Explicitly Missing Cell Values	Float
	Fraction of Alphabetical Cell Values	Float
	Fraction of Numerical Cell Values	Float
	Fraction of Punctuation Cell Values	Float
	Fraction of Miscellaneous Cell Values	Float
Quality Features	Normalized Raw Output Size of Each Strategy	Float
	Normalized Raw Output Overlap of Each Pair of Strategies	Float
	Sample Precision of Each Strategy (Optional)	Float

suffers from two limitations. First, the user has to run all the error detection strategies on each new dataset, which could be time consuming. Second, the user has to evaluate all the error detection strategies on a data sample, which could be tedious.

Therefore, it is desirable to estimate the effectiveness of error detection strategies upfront, without having to run and evaluate them on any new dataset.

Problem 3 (**Estimating the effectiveness of error detection strategies**)**.** Suppose $D = \{d_1, d_2, ..., d_{|D|}\}$ is a set of historical datasets with the corresponding ground truths $D^* = \{d_1^*, d_2^*, ..., d_{|D|}^*\}$. Suppose $S = \{s_1, s_2, ..., s_{|S|}\}$ is the set of available error detection strategies. Given a new dataset d_{new}, the problem is to estimate the effectiveness (i.e., the F_1 score) of each strategy s on the new dataset d_{new}, i.e., $\hat{F}(s, d_{\text{new}}), \forall s \in S$. □

Our intuition is that, on *similarly dirty* datasets, the same error detection strategies will perform similarly well. To define similarity based on dirtiness, we introduce the novel concept of a *dirtiness profile*, which is composed of various metadata features. Mapping each dataset to its dirtiness profile, we design methods to estimate the effectiveness of the error detection strategies on the new dataset based on the similarity of dirtiness profiles.

We first detail the concept of dirtiness profile. Then, we elaborate our algorithms to estimate the effectiveness of the error detection strategies using the dirtiness profiles. Finally, we detail how estimating the effectiveness of the error detection strategies improves the overall error detection performance.

6.1.1 Dirtiness Profile

Similar datasets should require similar error detection efforts as well. The similarity of datasets can be defined with regard to different dimensions. Content-based and structure-based similarities are typical similarity dimensions for comparing two datasets [71, 3]. However, these similarity dimensions are not enough for comparing datasets as two datasets with almost the same content and structure could suffer from different data error types, such as missing values and typos. Thus, we argue that, in the data cleaning context, the *data quality similarity* is another important similarity dimension.

We propose a dirtiness profile that summarizes the content, structure, and quality of a dataset into metadata features. Table 6.1 shows the features of the dirtiness profile. Our proposed dirtiness profile has two properties. First, it can be generated for each dataset automatically, i.e., without any user involvement. Second, it contains both domain-dependent (i.e., content) features and domain-independent (i.e., structure and quality) features.

6.1.1.1 Content Features

Content features represent the domain of data. Datasets with similar data domains are likely to have similar data errors as well. For example, multiple datasets in literature, such as *Hospital* [74], *Address* [2], and *Beers* [43], contain the data domains *ZIP*, *City*, and *State*. For these content-wise similar datasets, applying the same rule violation detection strategy that marks data cells violating the functional dependencies *ZIP* $\rightarrow$ *State* and *City* $\rightarrow$ *State* as data errors is typically promising. Therefore, to capture the content similarity, the dirtiness profile should leverage features that describe data domains of datasets.

Our dirtiness profile contains the most frequent words and cell values of each data column as features to represent the content of the dataset. This way, datasets with similar data domains (e.g., *City* and *Capital*) would have similar dirtiness profiles because of the same overlapping data values.

6.1.1.2 Structure Features

The structure of a dataset can be represented by various metadata. Error detection strategies are likely to have similar effectiveness on datasets with certain data value structures. In particular, the data type distribution of data columns is a key characteristic to estimate the effectiveness of error detection strategies. For example, on datasets that mainly contain numerical data values, outlier detection strategies could be more effective than rule violation detection strategies. Therefore, to capture the structural similarity, the dirtiness profile should leverage features that describe the distribution of data value types of datasets.

Our dirtiness profile contains the fraction of unique, explicitly missing, alphabetical, numerical, punctuation, and miscellaneous data values as features to represent the structure of each dataset. This way, datasets with similar distributions of data value types would have similar dirtiness profiles.

6.1.1.3 Quality Features

Datasets with similar data error distributions need similar error detection treatments as well. Ideally, the data error distribution should represent the fraction of each existing data error type in the dataset. For example, if we know that 5% of a dataset are outliers and 25% of the same dataset are rule violations, then the rule violation detection strategy is a more effective approach for this dataset rather than the outlier detection strategy.

However, it is not trivial to accurately calculate the distribution of data error types for a given dataset. We can evaluate the output of error detection strategies on a data sample to estimate the distribution of data error types [2].

Example 22 (**Manual evaluation of error detection strategies**)**.** Suppose we evaluate the output of a typo detection strategy on 1% of a dataset. Suppose 10 actual typos are identified by the user on this data sample. Assuming that the dataset has 2500 rows and 20 columns, we can extrapolate our observation to estimate $\frac{10 \times 100}{2500 \times 20} = 2\%$ of data values as typos. Therefore, we could estimate the fraction of typos in the dataset. □

The manual evaluation of error detection strategies is expensive as it needs the user to evaluate the output of all the strategies on a data sample.

Thus, it would be desirable to have an alternative set of automatically extractable features to represent the quality of datasets. To represent data quality, our features should be defined based on the output of error detection strategies as they mark certain data error types. To be automatically extractable, the feature generation process must not involve the user to evaluate

the output of the error detection strategies. Therefore, we design our features based on the raw output of the strategies. In particular, we leverage the raw output size and overlap of error detection strategies on datasets.

The output size of a specific error detection strategy might correlate with the actual number of data errors. For example, the number of marked data cells by a rule violation detection strategy hints at how many actual rule violations exist in the dataset. Thus, datasets that are associated with similar raw output sizes might be similarly dirty. Let $O_{\text{size}}(s, d)$ be the fraction of data cells that the strategy s marks as data errors in the dataset d. For all the available error detection strategies $s \in S$, we consider the $O_{\text{size}}(s, d)$ as a feature in the dirtiness profile of the dataset d.

The output overlap of error detection strategies captures the agreement of error detection strategies on a particular dataset. When two strategies have strongly overlapping raw outputs on a dataset, they should also be similarly effective on this dataset. Let $O_{\text{overlap}}(s_\alpha, s_\beta, d)$ be the fraction of data cells that both strategies s_α and s_β mark in the dataset d as data errors. For all the pairs of strategies $s_\alpha \neq s_\beta \in S$, we consider $O_{\text{overlap}}(s_\alpha, s_\beta, d)$ as a feature in the dirtiness profile of the dataset d.

For the sake of completeness, we can also incorporate sample precision of strategies as *optional* user-provided features. Our system, i.e., REDS [55], is able to adequately estimate the effectiveness of strategies without this feature group. Note that measuring the recall and therefore F_1 score of error detection strategies on a data sample is not possible as we have no information about false and true negatives [2].

6.1.2 Estimation Algorithm

We can now calculate the similarity of datasets using their dirtiness profiles. However, we still need algorithms to estimate the effectiveness of error detection strategies using the similarity of datasets.

We design two methods based on two different assumptions separately. The first method is based on the assumption that the user can run the error detection strategies on the current dataset and optionally evaluate them on a data sample. The second method is based on the assumption that the user cannot run the error detection strategies on the current dataset at all and she needs to estimate their effectiveness without having the raw output of the strategies.

6.1.2.1 Estimation with Running Strategies

In many error detection scenarios, optimizing runtime is not a critical objective. Thus, the user has enough time to run all the error detection strategies on the dirty dataset and collect their raw output. The user might also optionally evaluate the strategies on a data sample. Therefore, we can generate a full dirtiness profile for each dataset, including the content, structure, and quality features.

Having the full dirtiness profiles, we estimate the effectiveness of error detection strategies by training regression models. Let $Z = \{z_1, z_2, ..., z_{|S|}\}$ be the set of $|S|$ independent regression models, as many as the number of error detection strategies. Each regression model z_j learns to estimate the effectiveness of one specific error detection strategy s_j on the new datasets.

In the training phase, which is offline, each regression model z_j takes as input the set of training data points, i.e., the historical dirtiness profiles $V_{\text{train}} = \{\langle v_{d_1}, F(s_j, d_1)\rangle, \langle v_{d_2}, F(s_j, d_2)\rangle, ..., \langle v_{d_{|D|}}, F(s_j, d_{|D|})\rangle\}$. Here, v_{d_i} is the dirtiness profile of the dataset d_i and $F(s_j, d_i)$ is the target value, i.e., the F_1 score of the error detection strategy s_j on the dataset d_i.

In the prediction phase, which is online, each regression model z_j takes as input the new dirtiness profile $v_{d_{\text{new}}}$ of the new dataset d_{new} and estimates the F_1 score of each error detection strategy s_j, i.e., $\hat{F}(s, d_{\text{new}}), \forall s \in S$.

6.1.2.2 Estimation without Running Strategies

In some error detection scenarios, optimizing runtime is also an important objective. Thus, the user cannot run all the error detection strategies on the new dataset to select the most promising ones as running all the strategies could be time consuming. Therefore, we need a method that estimates the effectiveness of error detection strategies without running them on the new dataset.

Estimating the effectiveness of error detection strategies is more challenging in this scenario for two reasons. First, we cannot generate a full dirtiness profile for each new dataset as the quality features of the dirtiness profile require the raw output of error detection strategies. Without the full dirtiness profile features, we cannot leverage the previous regression-based method to estimate the effectiveness of strategies. Second, without having the raw output of error detection strategies, mapping the strategies across datasets is more challenging as we cannot compare the strategies based on their raw outputs on different datasets. In particular, the schema-dependent error detection strategies, such as rule violation detectors, cannot easily be transferred across different datasets as each dataset requires different integrity rules based on its own schema.

To alleviate the first challenge, we define the concept of *column profiles*, which is the dirtiness profile of a data column. In the lack of quality features, each column profile contains the content and structure features of one particular data column. Likewise, the idea is that on similar data domains, similar error detection strategies will perform similarly. For example, on a data column *City*, we need to run only those error detection strategies that performed well on the data column *Capital* of some historical datasets. Instead of generating the dirtiness profile for the whole dataset, we generate a profile for the finer granular data columns because the remaining content and structure features of the dirtiness profile represent a data domain more effective than the whole dataset.

To address the second challenge, we design an algorithm that systematically adapts and predicts the promising error detection strategies for a new data column based on their effectiveness on other similar data columns. This way, Raha can select the top-ranked adapted strategies and filter out the rest.

Algorithm 3 shows how we leverage the similarity between a new data column $d_{\text{new}}[:, j]$ and a historical data column $d[:, j']$ to select the promising error detection strategies for the data column $d_{\text{new}}[:, j]$. The algorithm consists of four main steps.

First, the cosine similarity between the profile of each data column $d_{\text{new}}[:, j]$ of the new dataset d_{new} and the profile of each data column $d[:, j']$ from any historical dataset $d \in D$ is computed (line 7).

Second, the algorithm retrieves the stored F_1 score of the strategy s on the historical data column $d[:, j']$, i.e., $F(s, d[:, j'])$ (line 9).

Third, the algorithm might need to modify the strategy s to make it compatible to run on the new data column $d_{\text{new}}[:, j]$ (line 10). For an outlier detector, a pattern violation detector, or a knowledge base violation detector, a modification is not necessary. Raha can simply run the same strategy on the new data column. However, for a rule violation detection strategy, which is schema dependent, Raha needs a modified strategy as the integrity rules have to be updated based on the schema of the new dataset.

Example 23 (**Error detection strategy adaptation**)**.** Suppose we have a functional dependency checker $s_{j'_1 \to j'_2}$ that has detected data errors on the data columns $d[:, j'_1]$ and $d[:, j'_2]$ of the historical dataset d. Suppose the algorithm identifies the historical data column $d[:, j'_1]$ similar to the new data column $d_{\text{new}}[:, j]$.

Therefore, the algorithm translates the strategy $s_{j'_1 \to j'_2}$ to the strategy $s_{j \to q}$ for the new dataset d_{new}. To adapt the strategy $s_{j'_1 \to j'_2}$, the algorithm replaces the data column $d[:, j'_1]$ with its corresponding data column $d_{\text{new}}[:, j]$ and the data column $d[:, j'_2]$ with its most similar data column $d_{\text{new}}[:, q]$ inside the new dataset. □

Fourth, the algorithm assigns a score to each updated error detection strategy s' (line 11). The score is the product of the similarity of the data columns $d_{\text{new}}[:, j]$ and $d[:, j']$ and the F_1 score of the strategy s on the data column $d[:, j']$. Formally,

$$\text{score}(s') = \text{similarity}(d_{new}[:, j], d[:, j']) \times F(s, d[:, j']). \tag{6.1}$$

The score will be high if the data columns $d_{\text{new}}[:, j]$ and $d[:, j']$ are similar and the strategy s has had a high F_1 score on the data column $d[:, j']$, indicating that the updated strategy s' will be promising for the data column $d_{\text{new}}[:, j]$.

We can sort the scored strategies S'_j for each data column $d_{\text{new}}[:, j]$ to pick only top-scored strategies per data column. A threshold-free approach to select the most promising subset of the strategies is to apply a gain function [4]. The idea is to add the top-scored strategies $s^* \in S'_j$ iteratively to the set of promising strategies S^*_j (lines 13 to 18), until adding the next best strategy decreases the following gain function [4], where the gain reaches a local maxima. Formally,

$$\text{gain}(S^*_j) = \sum_{s \in S^*_j} score(s) - \frac{1}{2} \sum_{s \in S^*_j} \sum_{s' \neq s \in S^*_j} |score(s) - score(s')|. \tag{6.2}$$

In the end, the set of promising error detection strategies S^* is the union of the promising strategies over all the data columns (line 19).

Algorithm 3 iterates over all the historical datasets (line 4) and their data columns (line 5). Alternatively, it is possible to create data column indices such as *MinHash* [14] to quickly find relevant historical datasets and data columns for a new data column. However, we ignore this optimization as the number of historical cleaned datasets is usually limited.

6.1.3 Estimation Benefits

Estimating the effectiveness of error detection strategies allows us to filter out ineffective strategies upfront, without running them on the dataset. In particular, Raha leverages the column profiles and the strategy filterer algorithm to filter out ineffective error detection strategies upfront. Therefore, the runtime of Raha significantly reduces as Raha runs only a promising subset of error detection strategies on the dataset. Although the resulting feature vectors are also smaller, they are still expressive enough to achieve high effectiveness, as we will show in the experiments.

6.2 Pretraining Error Corrector Models

Baran is an error correction system that internally aggregates multiple base error corrector models. The error corrector models can be trained on the current dataset from scratch. In that case, the correction candidates are either provided by the correct values inside the given dataset or through user corrections. This approach might face two general limitations. First, the limited number of user-provided corrections might not be enough to train the error corrector

Algorithm 3: 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^*.

for *each data column* $j \in [1, |A_{new}|]$ *of the dataset* d_{new} **do**
  $S'_j \leftarrow \{\}$; `// set of all adapted strategies for the data column` j
  $p_{d_{\text{new}}[:,j]} \leftarrow$ generate a profile for the data column $d_{\text{new}}[:, j]$;
  for *each dataset* $d \in D$ **do**
    for *each data column* $j' \in [1, |A|]$ *of the dataset* d **do**
      $p_{d[:,j']} \leftarrow$ generate a profile for the data column $d[:, j']$;
      similarity($d_{\text{new}}[:, j]$, $d[:, j']$) $\leftarrow$ similarity of the column profiles $p_{d_{\text{new}}[:,j]}$ and $p_{d[:,j']}$;
      for *each strategy* $s \in S$ **do**
        $F(s, d[:, j']) \leftarrow$ the F_1 score of the strategy s on the data column $d[:, j']$;
        $s' \leftarrow$ adapt the strategy s for the dataset d_{new};
        score(s') $\leftarrow$ similarity($d_{\text{new}}[:, j], d[:, j']$) $\times F(s, d[:, j'])$;
        $S'_j \leftarrow S'_j \cup \{s'\}$;
  $S^*_j \leftarrow \{\}$; `// set of promising adapted strategies for the data column` j
  do
    $s^* \leftarrow \arg\max_{s \in S'_j} \text{score}(s)$;
    $S^*_j \leftarrow S^*_j \cup \{s^*\}$;
    $S'_j \leftarrow S'_j - \{s^*\}$;
  while *adding* s^* *to* S^*_j *does not decrease the gain of the set* S^*_j ;
$S^* \leftarrow \cup_{j=1}^{|A_{\text{new}}|} S^*_j$;

models sufficiently. Second, some out-of-dataset corrections might never be found. Fortunately, our problem formulation allows us to extract additional correction candidates from external sources to pretrain the error corrector models.

As mentioned in Section 5.1, we have three types of error corrector models. Since the vicinity-based and domain-based error corrector models are schema dependent, they should be pretrained on datasets with the same schema. Thus, pretraining them would be straightforward as the historical data would be a structured dataset with the same schema. The value-based error corrector models can be schema independent and hence can be pretrained on any dataset where value-based corrections can be extracted.

Pretraining value-based models on structured datasets with ground truth is again straightforward as we can simply collect the erroneous and clean values by comparing the dataset with its ground truth. However, finding these structured datasets with ground truth is hard as creating ground truth is an expensive task. In the absence of structured datasets with ground truth, we can resort to non/semi-structured datasets with user-committed corrections. There are publicly available general-purpose revision histories, such as the Wikipedia page revision history.

Problem 4 (**Pretraining error corrector models**)**.** Suppose W is a non/semi-structured dataset with a revision history. Suppose $M = \{m_1, m_2, ..., m_{|M|}\}$ is the set of available value-based error corrector models. The problem is to extract new value-based corrections from W and pretrain each value-based model m with them.

Extracting value-based corrections from the revision histories requires two main steps. First, we need to break down the non/semi-structured revision texts into text segments (i.e., chunks). Second, we need to align text segments across subsequent revisions to collect value-based corrections. Here, we briefly discuss the implementation of these steps for the Wikipedia page revision history as a general-purpose and publicly available revision dataset [59, 95]. To apply the same approach to other similar revision histories, such as web pages', we just need to adapt the text segmenter to the corresponding markup language, e.g., HTML. Naturally, we can extract more effective value-based corrections for the current data cleaning task when the historical data is more similar to the dirty dataset at hand.

Figure 6.1: The Wikipedia page view history.

We first briefly introduce the Wikipedia page revision history. We then explain our algorithm to segment the Wikipedia page revision texts. Next, we elaborate our method in aligning the segmented texts to collect new value-based corrections. Finally, we detail how pretraining the error corrector models with these new value-based corrections improves the overall error correction performance.

6.2.1 Wikipedia Page Revision History

The Wikipedia page revision history is a rich semi-structured corpus of revision data that contains terabytes of human-committed revisions. The Wikipedia page revision history is available in Wikipedia released dumps [91] and also on each web page itself under the "View History" menu (Figure 6.1). As shown in Figure 6.1, each page revision consists of the revision data, i.e., the source and the target texts, and the revision metadata, e.g., the timestamp, the author, and the comments.

Formally, let $W = \{P_1, P_2, \ldots, P_{|W|}\}$ be the Wikipedia corpus, where each $P \in W$ is a page. Let each page contain a history of all the revised versions $P = \{r_1, r_2, \ldots, r_{|P|}\}$, where each $r \in P$ is a revised version of the text of the page P. Here, r_1 is the first version and $r_{|P|}$ is the latest updated version of the page P.

Wikipedia pages are written in the Wikitext markup language (also known as Wiki markup or Wikicode). This markup language recognizes a set of entities. For example, `'''Christopher Nolan'''` (a text inside three single quotations) is an entity that makes the text "Christopher Nolan" bold and `[[Inception]]` (a text inside two square brackets) is an entity that creates a link to another Wikipedia page with the name "Inception" [92].

Example 24 (Wikipedia page revision history). A Wikipedia page $P = \{r_1, r_2\}$ with a history of two revisions could be as follows:

r_1 = "`'''Chris Nolan'''` `(born on ''30/07/1970'') is a well-known [[British]] film-maker.`"

r_2 = "`'''Christopher Nolan'''` `(born on ''30.07.1970'') is a well-known [[English]] filmmaker.`" □

6.2.2 Recursive Text Segmentation

A Wikipedia page revision text is semi-structured. Thus, we need to break its free texts down into segments that are useful for training our value-based error corrector models.

Algorithm 4: RecursiveTextSegmenter(r).

```
Input: Wikipedia page revision text r.
Output: list of text segments S.
S ← [];                                                    // list of text segments
recursion(r, S);
def recursion(r′, S′):
    if r′ is text, not a compound entity then
        S′ ← append the text r′ to the list S′;
    else
        for each subentity q ∈ r′ do
            recursion(q, S′);
```

Algorithm 4 shows the recursive text segmentation procedure. The algorithm takes a Wikipedia page revision text in Wikitext markup language as input and recursively breaks the text down into its entities. The algorithm *recursively* segments the text because each Wikitext entity can consist of subentities. For example, `'''[[Inception]]'''` is a bold entity (i.e., `'''bold'''`) that contains a link subentity (i.e., `[[link]]`). In the end, the list of text segments of the Wikipedia page revision is the output of the algorithm.

Example 25 (**Recursive text segmentation**)**.** Considering the two Wikipedia page revision texts r_1 and r_2 from Example 24, the recursive text segmentor algorithm breaks these two revision texts down into two lists of text segments:

$S_1 = [$“`Chris Nolan`”, “`(born on `”, “`30/07/1970`”, “`) is a well-known `”, “`British`”, “` film-maker.`”$]$

$S_2 = [$“`Christopher Nolan`”, “`(born on `”, “`30.07.1970`”, “`) is a well-known `”, “`English`”, “` filmmaker.`”$]$ □

6.2.3 Text Segment Alignment

Having the list of text segments for each Wikipedia page revision, we need to align the corresponding segments in every consecutive segment list to collect new value-based corrections.

Let S_1 and S_2 be the text segment lists of two consecutive revisions r_1 and r_2, respectively. We again perform diff checking but this time on the segment level to identify which segments of the first list are transformed to which segments of the second list. We can also tokenize each segment to have finer granular token alignments.

Overall, we capture value-based corrections accurately, although some of them are not useful for our use case. In particular, we discard value-based corrections that involve null values.

Example 26 (**Text segment alignment**)**.** Considering the two text segment lists S_1 and S_2 from Example 25, the alignment of the text segments is as follows:

$$\text{diff}(S_1, S_2) = \begin{cases} \texttt{Replace "Chris Nolan" with "Christopher Nolan".} \\ \texttt{Replace "30/07/1970" with "30.07.1970".} \\ \texttt{Replace "British" with "English".} \\ \texttt{Replace " film-maker." with " filmmaker.".} \end{cases}$$

Thus, we obtain new value-based corrections as training data points, such as fixing “Chris Nolan” to “Christopher Nolan”. We also tokenize each text segment and increase the training data points with finer granular value-based correction examples, such as fixing “Chris” to “Christopher”. □

6.2.4 Pretraining Benefits

The value-based error corrector models can finally be pretrained with the additional correction examples. The pretrained models generate more effective correction candidates for data errors of the dataset at hand.

Baran can now fix more data errors with the same user labeling budget as more correction candidates are now available and, at the same time, the corresponding features of the pretrained models exhibit more evidence for the classifiers. Note that Baran can still avoid irrelevant correction candidates with the help of the user labels. The user labels let the classifier learn and prioritize across all available correction candidates.

Example 27 (**Pretraining benefits**)**.** Pretraining the value-based error corrector models with the new correction examples extracted in Example 26 makes it possible to fix the last remaining data error of our toy dataset in Table 5.1, without any further user label.

Considering the data column *Name* in Table 5.1, the following table contains pairs of data errors and correction candidates and their corresponding feature vectors. For brevity, we just demonstrate a few pairs and features. The features are a value-based model with the identity encoder and the adder operator ($m_{\text{Identity+Adder}}$) and a domain-based model (m_{Name}). The first two pairs are labeled as the user already validated the tuple 1 of our toy dataset in Example 19.

Error	Correction	$\mathbf{m_{Identity+Adder}}$	$\mathbf{m_{Name}}$	Label
H	Hana	1.0*	0.67	1
H	Gandom	0.0	0.33	0
Chris	Hana	0.0	0.67	
Chris	Gandom	0.0	0.33	

The classifier of the data column *Name* receives these pairs as data points. It trains with the first two labeled data points and then predicts the label of the rest of unlabeled data points. With only these two labeled data points, the classifier has no chance to fix the erroneous value "Chris" to its actual correction "Christopher", because it is not among the correction candidates at all. However, if we pretrain the value-based model $m_{\text{Identity+Adder}}$ with the previously extracted example of fixing the erroneous value "Chris" to the value "Christopher" from Wikipedia, we will have the following new pair of a data error and a correction candidate as a new data point.

Error	Correction	$\mathbf{m_{Identity+Adder}}$	$\mathbf{m_{Name}}$	Label
Chris	Christopher	1.0*	0.0	

The classifier now has enough evidence to fix the erroneous value "Chris" without any further user label. Since the user already fixed the erroneous value "H" to "Hana", the classifier learned that the value-based model $m_{\text{Identity+Adder}}$ (marked with *) is an important feature. The classifier also observes that the same value-based feature $m_{\text{Identity+Adder}}$ has a high probability for the pair ("Chris", "Christopher"), as the model learned this probability in the pretraining phase. Therefore, the classifier can predict the value "Christopher" as the final correction of the erroneous value "Chris". □

6.3 Summary

We proposed new methods to transfer error detection and correction knowledge from previously cleaned datasets to the current dataset. In particular, we proposed methods to estimate

the effectiveness of error detection strategies on new datasets based on their effectiveness on similarly dirty datasets. To define similarity between datasets, we designed a dirtiness profile to represent the dirtiness of a dataset based on automatically extractable features. Furthermore, we proposed a method to pretrain value-based error corrector models on the huge corpus of Wikipedia page revision history. To collect additional value-based corrections, we designed algorithms to segment and align the Wikipedia page revision history. These transfer learning methods improve the overall performance of the error detection/correction tasks.

7 Evaluation

We conducted extensive experiments to evaluate our proposed data cleaning approach in terms of effectiveness, efficiency, and human involvement. We first elaborate our experimental setup. We then detail our error detection, error correction, and end-to-end data cleaning experiments. Finally, we summarize the chapter.

7.1 Experimental Setup

We detail our datasets, baselines, evaluation measures, and default settings.

7.1.1 Datasets

We evaluate our systems on 8 well-known datasets from existing literature as described in Table 7.1. The difficulty level of the error detection/correction task depends on the data error rate, the diversity of data error types, and the availability of error context signals. We manually examined the datasets to identify prevalent data error types and useful contextual information for detecting/correcting the data errors.

1. **Hospital.** *Hospital* [74] is a real-world dataset with randomly imposed data errors. We obtained this dataset along with its ground truth from a previous research project [74]. This dataset has rich contextual information, including a high degree of data redundancy in the form of duplicate tuples and correlated data columns. However, sampling informative tuples is particularly challenging on this dataset as the data errors are scarce and randomly imposed.

2. **Flights.** *Flights* [53, 74] is a real-world dataset with real-world data errors. A previous research project released this dataset and its ground truth [74]. This dataset also has rich contextual information, including a high degree of data redundancy in the form of duplicate tuples and correlated data columns. However, the degree of trustworthy contextual information is low on this dataset due to its high data error rate.

3. **Address.** *Address* is a proprietary dataset with ground truth. This dataset is large and contains various data error types. Its large size, few duplicate tuples, and few correlated data columns lead to a huge search space for detecting/correcting data errors.

Table 7.1: Dataset characteristics. The data error types are missing value (MV), typo (T), formatting issue (FI), and violated attribute dependency (VAD) [72].

Name	Size	Error Rate	Error Types	Data Constraints
Hospital	1000 × 20	3%	T, VAD	city → zip, city → county, zip → city, zip → state, zip → county, county → state, index (digits), provider number (digits), zip (5 digits), state (2 letters), phone (digits)
Flights	2376 × 7	30%	MV, FI, VAD	flight → actual departure time, flight → actual arrival time, flight → scheduled departure time, flight → scheduled arrival time
Address	94306 × 12	14%	MV, FI, VAD	address → state, address → zip, zip → state, state (2 letters), zip (digits), ssn (digits)
Beers	2410 × 11	16%	MV, FI, VAD	brewery id → brewery name, brewery id → city, brewery id → state, brewery id (digits), state (2 letters)
Rayyan	1000 × 11	9%	MV, T, FI, VAD	journal abbreviation → journal title, journal abbreviation → journal issn, journal issn → journal title, authors list (not null), article pagination (not null), journal abbreviation (not null), article title (not null), article language (not null), journal title (not null), journal issn (not null), article journal issue (not null), article journal volume (not null), journal created at (date)
IT	2262 × 61	20%	MV, FI	support level (not null), app status (not null), curr status (not null), tower (not null), end users (not null), account manager (not null), decomm dt (not null), decomm start (not null), decomm end (not null), end users (not 0), retirement (predefined list), emp dta (predefined list), retire plan (predefined list), division (predefined list), bus import (predefined list)
Movies	7390 × 17	6%	MV, FI	id ("tt" + digits), year (4 digits), rating value (float), rating count (digits), duration (digits + "min")
Tax	200000 × 15	4%	T, FI, VAD	zip → city, zip → state, first name → gender, area code → state, gender (predefined list), area code (3 digits), phone (7 formatted digits), state (2 letters), zip (non-zero-leading digits), material status (predefined list), has child (predefined list), salary (digits)

4. **Beers.** *Beers* is a real-world dataset that was collected by scraping the web and manual curation [43]. The lack of data redundancy makes error detection/correction challenging on this dataset.

5. **Rayyan.** *Rayyan* [61] is a real-world dataset that was cleaned by the dataset owner. The lack of data redundancy makes error detection/correction challenging on this dataset.

6. **IT.** *IT* [2] is a real-world dataset that was cleaned by the dataset owner. Prevalent missing values on this dataset decreases the the degree of trustworthy contextual information for the error detection/correction task.

7. **Movies.** *Movies* is a dataset from the Magellan repository [23]. We used the existing labels for the duplicate tuples to generate a ground truth for this dataset. Sampling informative tuples is particularly challenging on this dataset as the data errors are scarce.

8. **Tax.** *Tax* is a synthetic dataset from the BART repository [7] with randomly imposed data errors. This dataset is also large, which makes the error detection/correction search space huge.

7.1.2 Baselines

We compare our data cleaning systems to 10 recent data cleaning approaches.

1. **dBoost.** *dBoost* [67] is an outlier detection tool that contains several algorithms such as histogram and Gaussian modeling. dBoost also provides the tuple expansion feature to expand string values into numerical values and find string outliers as well. We applied grid search to configure dBoost algorithms. In fact, we ran differently configured versions of dBoost algorithms and evaluated the results on a data sample to report the best-configured algorithm with the highest effectiveness numbers.

2. **Min-k.** *Min-k* [2] is a simple error detection aggregator that outputs data cells that are marked by more than $k\%$ of the error detection strategies. We ran the approach with $k \in \{0\%, 20\%, 40\%, 60\%, 80\%\}$ and report the highest effectiveness numbers. Note that $k = 0\%$ corresponds to the union of all strategies' outputs.

3. **Maximum entropy-based order selection.** *Maximum entropy-based order selection* [2] is an error detection aggregator that evaluates the error detection strategies on a data sample. The approach then picks the error detection strategy with the highest precision first, evaluates its output, and picks the next best strategy.

4. **Metadata driven.** *Metadata driven* [86] is an error detection aggregator that combines the output of manually configured stand-alone error detection tools and metadata in a feature vector. The approach then trains ensemble classifiers using the feature vectors and a labeled data sample. We use this aggregator on top of the stand-alone error detection tools, using the same best-effort configurations.

5. **HoloDetect.** *HoloDetect* [41] is a machine learning-based error detection framework that leverages data augmenting. Given a set of labeled tuples, HoloDetect generates additional training data to learn the error detection task more effectively. Since HoloDetect's source code has not been released yet at the time of writing this thesis, we cannot run it on our datasets to conduct a holistic comparison. Instead, we compare HoloDetect's reported numbers on the shared datasets.

6. **KATARA.** *KATARA* [21] is a data cleaning system powered by knowledge bases that takes a set of entity relationships as input and marks/fixes the violating data errors accordingly. We ran KATARA with all the entity relationships that are available in the DBpedia knowledge base [8].

7. **NADEEF.** *NADEEF* [22] is a rule-based data cleaning system that takes integrity rules in the form of denial constraints and marks/fixes the violating data cells. We ran NADEEF with data constraints that are provided by the dataset owners (Table 7.1).

8. **Holistic.** *Holistic* [20] is a rule-based error correction system that uses denial constraints to fix the violating data errors accordingly. We ran Holistic with all the data constraints that are provided by the dataset owners (Table 7.1).

9. **SCARE.** *SCARE* [93] is an error correction system that partitions the dataset and uses the clean values to choose corrections of data errors based on their statistical likelihood. We ran SCARE with random data partitioning. We set its maximum number of value corrections to the number of detected errors to accommodate all data errors.

10. **HoloClean.** *HoloClean* [74] is an error correction system that leverages integrity rules, matching dependencies, and statistical signals simultaneously to fix data errors holistically. We ran HoloClean with all the data constraints and matching dependencies that are provided by the dataset owners (Table 7.1).

7.1.3 Evaluation Measures

We leverage different evaluation measures to evaluate our systems. We report precision, recall, and the F_1 score to evaluate the effectiveness. Formally,

$$\begin{aligned} P &= \frac{\text{The Number of Correctly Detected/Corrected Data Errors}}{\text{The Number of All Detected/Corrected Data Errors}}, \\ R &= \frac{\text{The Number of Correctly Detected/Corrected Data Errors}}{\text{The Number of All Data Errors}}, \\ F_1 &= \frac{2 \times P \times R}{P + R}. \end{aligned} \tag{7.1}$$

We report the runtime in seconds to evaluate the efficiency. We report the number of labeled tuples to evaluate the human involvement. For each evaluation measure, we report the mean of 10 independent runs. For the sake of readability, we omit the standard errors as they are always small numbers close to zero.

7.1.4 Our Default Setting

As the default setting, Raha and Baran incorporate all the error detection strategies and error corrector models, respectively. They do not leverage any historical data for transfer learning. Raha uses the Gradient Boosting classifier [33] and Baran uses the AdaBoost classifier [32] as these advanced ensemble classifiers are less susceptible to overfitting [32]. We set the labeling budget of the user to $\theta_{\text{Labels}} = 20$, i.e., 20 labeled tuples per dataset for each system separately. We assess the sensitivity of our systems to all these choices in our experiments. We run all the experiments on an Ubuntu 16.04 LTS machine with 28 2.60 GHz cores and 264 GB memory.

7.2 Error Detection Experiments

Our error detection experiments aim to answer the following questions on Raha.

1. How does Raha compare to existing error detection approaches? (Section 7.2.1)
2. How does each group of error detection strategies affect Raha's effectiveness? (Section 7.2.2)
3. How does the tuple sampling approach affect Raha's convergence? (Section 7.2.3)
4. How do user labeling errors affect Raha's effectiveness? (Section 7.2.4)
5. How does the performance of Raha scale in the number of data rows and columns? (Section 7.2.5)
6. How does the choice of the classifier affect Raha's effectiveness? (Section 7.2.6)
7. How does our method in estimating the effectiveness of error detection strategies affect Raha's performance? (Section 7.2.7)

7.2.1 Raha Versus the Baselines

We compare Raha to stand-alone and aggregator error detection approaches separately.

Table 7.2: Raha's effectiveness in comparison to the stand-alone error detection approaches.

Approach	Hospital			Flights			Address			Beers		
	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
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
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
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**

Approach	Rayyan			Movies			IT		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
dBoost	0.12	0.26	0.16	0.18	0.72	0.29	0.00	0.00	0.00
NADEEF	0.74	0.55	0.63	0.13	0.43	0.20	0.99	0.78	0.87
KATARA	0.02	0.10	0.03	0.01	0.17	0.02	0.11	0.17	0.14
Raha	0.81	0.78	**0.79**	0.85	0.88	**0.86**	0.99	0.99	**0.99**

7.2.1.1 Stand-Alone Approaches

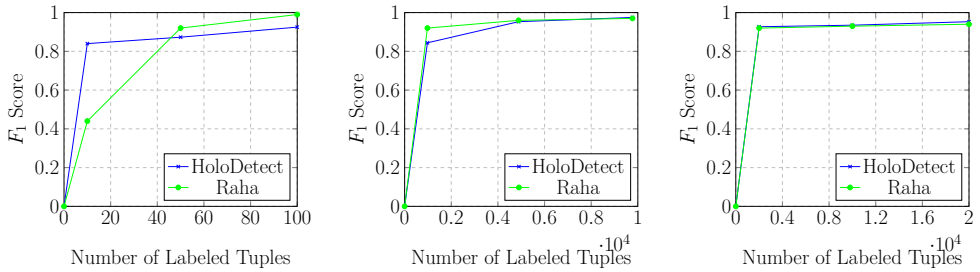
Table 7.2 shows the effectiveness of Raha in comparison to three stand-alone error detection systems. Raha outperforms all these systems on all the datasets in terms of F_1 score. The superiority of Raha is due to its novel task formulation that featurizes each data cell with a large number of signals to learn data errors. Raha achieves this effectiveness with only a limited number of labeled tuples, i.e., $\theta_{\text{Labels}} = 20$ tuples per dataset for the reported numbers.

The other baselines fail to achieve high effectiveness with low human involvement due to their design. They are usually bounded to only one type of data errors and are not expressive enough to accurately differentiate clean and dirty values. dBoost yields low recall because it marks only statistical outliers as data errors. It also achieves low precision because its applied heuristics leads to outlying legitimate values as data errors. This system also needs the user to tune statistical parameters. NADEEF yields low recall because it marks only rule violation data errors. It also achieves low precision because it reports data errors in the coarse granular violation form, which consists of multiple data cells. This system also needs the user to provide the correct and complete set of integrity rules. KATARA yields low recall because it marks only data values that do not conform to the entity relationships inside the knowledge base. It achieves low precision because the ambiguity of concepts leads to a mismatch between the dataset at hand and the external knowledge base. This system also needs the user to provide related knowledge bases.

Figure 7.1 shows the effectiveness of Raha in comparison to HoloDetect. Raha achieves a competitive F_1 score while involving the user much less than HoloDetect for two reasons. First, Raha does not take any user-provided rules or parameters to achieve this performance. Second, Raha requires the user to only mark data errors during the user labeling process. Contrary, HoloDetect requires the user to both mark and fix data errors in the sampled tuples to train error generator models. Note that since HoloDetect's source code has not been released yet, we cannot run it on our datasets. Hence, we instead ran Raha on HoloDetect's datasets to compare our performance numbers to HoloDetect's reported numbers. The superiority of HoloDetect on the *Hospital* dataset is due to the fact that data errors of this dataset are systematically injected typos. Thus, the data augmentation technique of HoloDetect can easily learn the error generation model and detect the rest of the data errors. However, Raha eventually outperforms HoloDetect on the *Hospital* dataset with a few labeled tuples.

7.2.1.2 Aggregator Approaches

Figure 7.2 shows the effectiveness of Raha in comparison to three error detection aggregators. Raha outperforms all the error detection aggregators on all the datasets in terms of F_1 score



(a) Hospital (b) Adult (c) Soccer

Figure 7.1: Raha's effectiveness in comparison to HoloDetect.

and user labels. Raha converges faster requiring fewer labeled tuples due to its expressive feature representation and effective tuple sampling method.

While the baseline aggregator approaches internally combine multiple error detection strategies, they fail to achieve high F_1 score due to the simplicity of their aggregation methods. They leverage simple heuristics as aggregation functions that work only based on the strong assumption of having accurate base error detection strategies. Therefore, their simple aggregation functions fail to achieve high precision when the base error detection strategies are not accurate.

The min-k approach, which simply marks as data errors the majority voted data cells, yields a low precision due to the inaccuracy of the base error detection strategies. Although this approach does not need user labels, it needs the user to configure the base error detection strategies and to set the parameter k.

The maximum entropy-based approach tries to address these shortcomings by aggregating the base error detection strategies based on their sample precision. However, its aggregation function is too simplistic to be able to learn complex data errors. Furthermore, this approach needs the user to configure the base error detection strategies and to manually evaluate the precision of each base strategy on a data sample.

The metadata-driven approach incorporates a learning component that learns to combine multiple base error detection strategies. However, it cannot achieve Raha's precision and recall due to its feature representation and tuple sampling methods. The metadata-driven approach leverages a small set of features, including the output of three base error detection strategies (i.e., dBoost, NADEEF, and KATARA) and metadata. This feature representation is not expressive enough to catch complex data errors. Furthermore, it randomly samples tuples for labeling, which is not an effective method due to the class imbalance ratio between clean and dirty data cells in datasets. This approach also needs the user to configure the base error detection strategies and to label a data sample.

7.2.2 Error Detection Strategy Impact Analysis

We conducted an ablation experiment to analyze the impact of different groups of error detection strategies on the effectiveness of Raha. We run Raha with all the error detection strategies (row *All* in Table 7.3). Then, we exclude each feature group, one at a time, to analyze its impact. For example, *All - OD* means Raha leverages all the error detection strategies as features but the outlier detection ones. Here, we also report the effectiveness of Raha when it uses *TFIDF* features, which is the featurization method of ActiveClean [51].

As shown in Table 7.3, Raha is robust against removing feature groups as its effectiveness does not collapse when a group of error detection strategies is excluded. However, depending on the data error rate and prevalent data error types, removing a feature group could reduce the

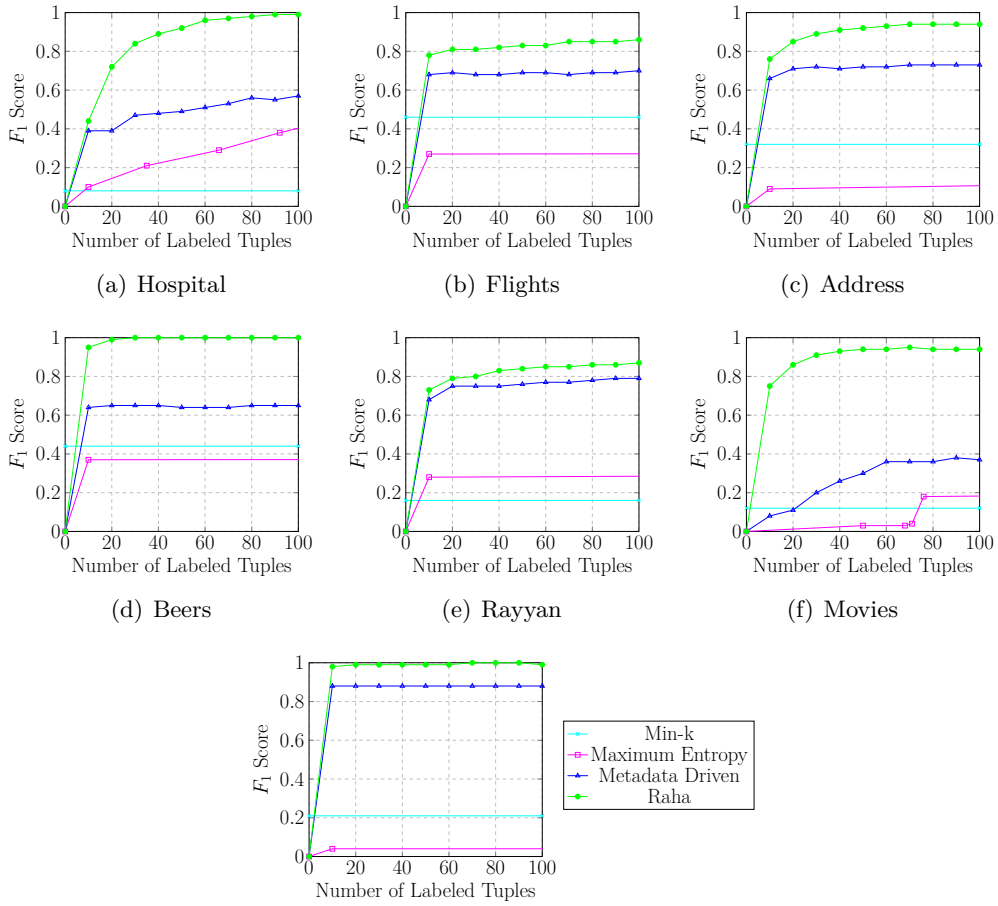


Figure 7.2: Raha's effectiveness in comparison to the error detection aggregators.

effectiveness more significantly. For example on the *Hospital* dataset, where data columns are highly correlated, removing the rule violation detection strategies decreases the effectiveness more severely as functional dependencies are very important signals to detect data errors on this dataset. On the *Movies* dataset, where the data error rate is low and thus the data errors are mainly outliers, removing the outlier detection strategies decreases the effectiveness more severely.

Interestingly, the TFIDF featurization of ActiveClean leads to high F_1 scores using Raha's sampling method. However, the overall effectiveness with TFIDF features is always worse than the full feature set of Raha.

7.2.3 Tuple Sampling Impact Analysis

We analyze the impact of our tuple sampling method on the effectiveness of Raha. In particular, we compare two versions of Raha with two different sampling methods. *Uniform sampling* method selects tuples for user labeling according to a uniform probability distribution. On the other hand, our proposed *clustering-based sampling* method first selects tuples based on the existing clusters of data cells and then propagates the user labels through the clusters.

As shown in Figure 7.3, our clustering-based sampling method speeds up the convergence of Raha. This higher convergence speed is more obvious on datasets with lower data error rates, such as *Hospital* and *Movies*, where finding enough dirty data cells to sufficiently train

Table 7.3: Raha's effectiveness with different groups of error detection strategies as features: 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		
	P	R	F	P	R	F	P	R	F	P	R	F
TFIDF	0.98	0.10	0.18	0.63	0.88	0.73	0.84	0.57	0.68	0.73	0.80	0.76
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
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
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
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
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

Feature Group	Rayyan			Movies			IT		
	P	R	F	P	R	F	P	R	F
TFIDF	0.91	0.58	0.70	0.19	0.04	0.07	0.92	0.97	0.95
All - OD	0.78	0.72	0.75	0.66	0.82	0.73	0.99	0.98	0.98
All - PVD	0.74	0.74	0.74	0.77	0.84	0.80	0.98	0.97	0.97
All - RVD	0.82	0.78	0.80	0.92	0.90	0.91	0.99	0.98	0.98
All - KBVD	0.83	0.76	0.79	0.80	0.88	0.84	0.99	0.97	0.98
All	0.81	0.78	0.79	0.85	0.88	0.86	0.99	0.99	0.99

the classifiers is harder due to the class imbalance ratio of clean and dirty data cells. Our clustering-based sampling method addresses this issue by generating additional noisy labels.

7.2.4 User Labeling Error Impact Analysis

We analyze the impact of user labeling errors on the effectiveness of Raha as the input user labels could be erroneous themselves. In particular, we compare the *homogeneity-* and *majority-based* conflict resolution functions for label propagation in the presence of user labeling errors. The erroneous user labels are randomly distributed across data cells of the sampled tuples.

As shown in Figure 7.4, Raha's effectiveness drops slightly with increased user labeling errors. The decline is more severe on the *Hospital* dataset because this dataset contains many similar data errors due to duplicate tuples. Having wrong user labels for such similar data errors confuses the classifiers.

The majority-based conflict resolution function fares better than its homogeneity-based counterpart in the presence of user labeling errors because the latter does not propagate any user label if user labels are contradictory. The majority-based conflict resolution function is more robust in these situations as it propagates user labels in any clusters where a label class has the majority. Under the assumption of perfect user labels, both conflict resolution functions perform nearly equal.

7.2.5 System Scalability

We analyze the scalability of Raha with respect to the number of data rows and columns. We leverage our large *Tax* and *IT* datasets to measure the effectiveness and efficiency of Raha when the number of data rows and data column increases.

As shown in Figure 7.5, Raha's runtime increases linearly and its F_1 score stays the same when the number of data rows and data columns increases.

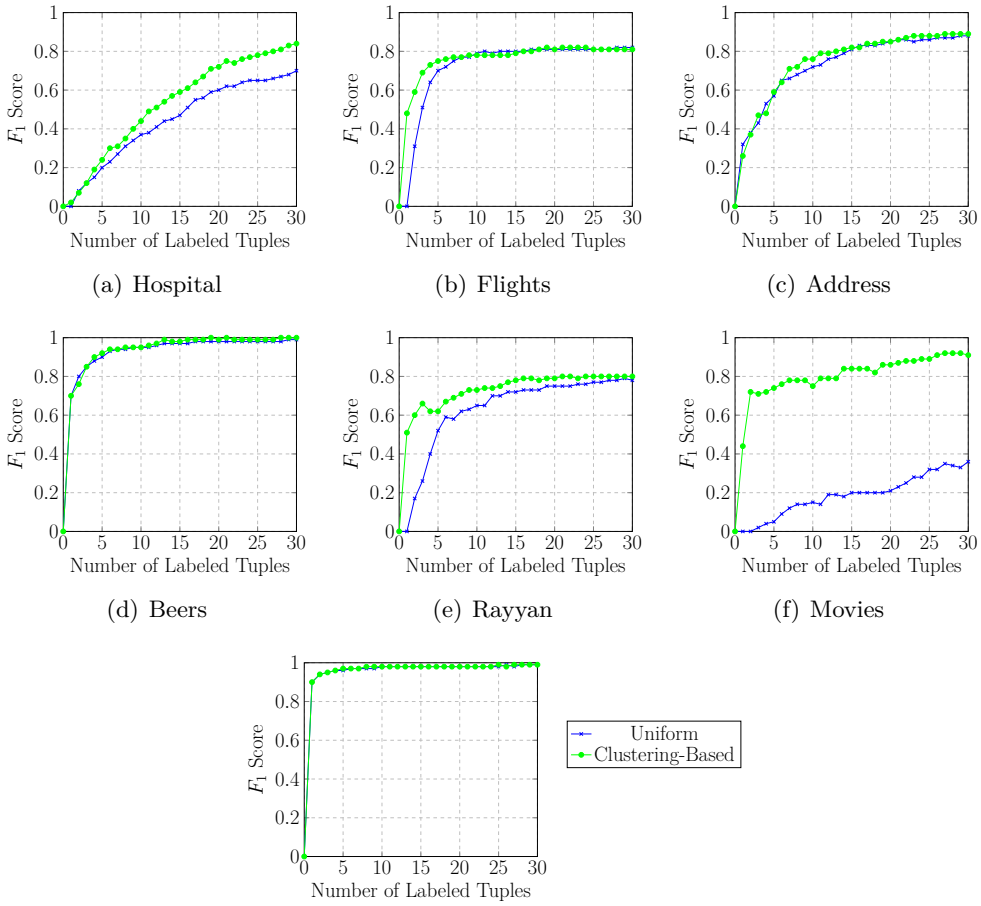


Figure 7.3: Raha's effectiveness with different tuple sampling methods.

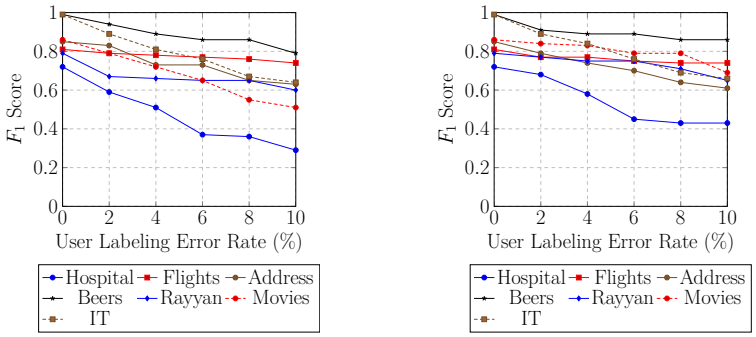
7.2.6 Classifier Impact Analysis

We analyze the impact of the classifier on the effectiveness of Raha. In particular, we tested *AdaBoost*, *Decision Tree*, *Gradient Boosting*, *Gaussian Naive Bayes*, *Stochastic Gradient Descent*, and *Support Vectors Machines*, all implemented in *scikit-learn* Python module [65]. The hyperparameter tuning task can slightly improve the effectiveness of these classifiers. However, as we aim at building a general-purpose system and fine-tuning Raha for each dataset is not our goal, we skip the straightforward hyperparameter tuning task and simply run each classifier with its recommended hyperparameters.

As shown in Table 7.4, the choice of the classifier does not affect the effectiveness of Raha significantly because the main impact of the system comes from the feature representation and the tuple sampling method. In our current prototype, we deploy the Gradient Boosting classifier because it is an advanced ensemble learning model that internally combines multiple simpler base classifiers [33].

7.2.7 Estimating the Effectiveness of Strategies Impact Analysis

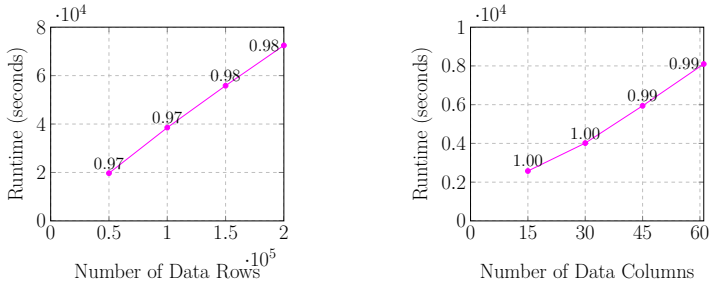
We analyze the effect of estimating the effectiveness of error detection strategies on the performance of Raha. We simulated a scenario where we have several cleaned datasets in our repository with a new dirty dataset arriving. For each run, we consider one of the datasets as the new dirty dataset d_{new} and the rest of the datasets as the set of historical datasets D, according to the well-known *leave-one-out* methodology [75].



(a) Homogeneity-Based Conflict Resolution Function

(b) Majority-Based Conflict Resolution Function

Figure 7.4: Raha's effectiveness in the presence of user labeling errors with the (a) homogeneity-based and (b) majority-based conflict resolution functions.



(a) Scalability with Respect to the Number of Data Rows of the *Tax* Dataset

(b) Scalability with Respect to the Number of Data Columns of the *IT* Dataset

Figure 7.5: Raha's scalability with respect to the number of (a) data rows and (b) data columns. The number on each point depicts the achieved F_1 score.

We evaluate both of our algorithms in estimating the effectiveness of strategies: with full dirtiness profiles and with column profiles. We first show how accurately we can estimate the effectiveness of error detection strategies with full dirtiness profiles, when all metadata features are available. We then show how Raha can filter out ineffective error detection strategies with column profiles, when the quality features of the dirtiness profile are not available.

7.2.7.1 Estimation Accuracy with Full Dirtiness Profiles

As mentioned in Section 6.1, the effectiveness of existing error detection aggregators depends on the effectiveness of their base error detection strategies. Therefore, we can improve their overall effectiveness by estimating the effectiveness of the base error detection strategies and selecting only the most promising strategies.

We analyze the accuracy of estimated effectiveness of error detection strategies with the assumption of having the full dirtiness profiles, including the raw output of all error detection strategies. Thus, we estimate the F_1 score of the error detection strategies on the new dataset with our regression-based method. We use *mean squared error* (MSE) to measure the quality of the estimated effectiveness as

$$MSE = \frac{1}{|S|} \sum_{s \in S} \left(F(s, d_{\text{new}}) - \hat{F}(s, d_{\text{new}}) \right)^2, \tag{7.2}$$

Table 7.4: Raha's effectiveness with different classifiers.

Classifier	Hospital			Flights			Address			Beers		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
AdaBoost	0.96	0.56	0.70	0.83	0.79	0.81	0.92	0.82	0.87	0.99	1.00	1.00
Decision Tree	0.95	0.57	0.71	0.82	0.79	0.80	0.90	0.81	0.85	1.00	0.99	0.99
Gradient Boosting	0.94	0.59	0.72	0.82	0.81	0.81	0.91	0.80	0.85	0.99	0.99	0.99
Gaussian Naive Bayes	0.99	0.45	0.61	0.88	0.69	0.77	0.84	0.84	0.84	1.00	0.97	0.98
Stochastic Gradient Descent	0.89	0.52	0.65	0.84	0.76	0.80	0.83	0.73	0.78	0.99	0.98	0.98
Support Vectors Machines	0.98	0.39	0.56	0.83	0.77	0.80	0.66	0.36	0.47	0.99	0.96	0.98

Classifier	Rayyan			Movies			IT		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
AdaBoost	0.80	0.78	0.78	0.81	0.87	0.84	0.98	0.98	0.98
Decision Tree	0.79	0.75	0.77	0.82	0.87	0.84	0.98	0.98	0.98
Gradient Boosting	0.81	0.78	0.79	0.85	0.88	0.86	0.99	0.99	0.99
Gaussian Naive Bayes	0.79	0.62	0.69	0.53	0.88	0.66	0.99	0.98	0.98
Stochastic Gradient Descent	0.82	0.78	0.80	0.80	0.85	0.82	0.99	0.97	0.98
Support Vectors Machines	0.82	0.79	0.80	0.74	0.85	0.79	0.97	0.98	0.98

where $F(s, d_{\text{new}})$ is the actual and $\hat{F}(s, d_{\text{new}})$ is the estimated F_1 score of the error detection strategy s on the new dataset d_{new}.

We compare three approaches in estimating the effectiveness of error detection strategies.

1. **Maximum entropy-based estimator.** The baseline *maximum entropy-based estimator* [2] asks the user to evaluate error detection strategies on a data sample and then estimates the F_1 score of each strategy by its sample precision.

2. **Unsupervised estimator.** Our *unsupervised estimator* leverage all the dirtiness profile features except the optional sample precision of error detection strategies to estimate the effectiveness of strategies.

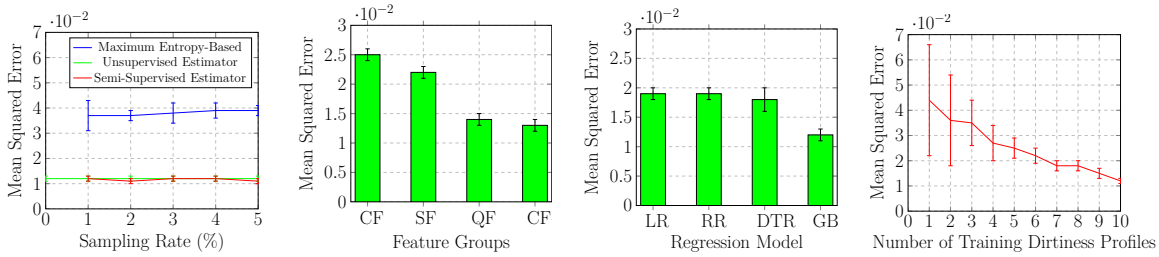
3. **Semi-supervised estimator.** Our *semi-supervised estimator* leverage all the dirtiness profile features. By default, the semi-supervised estimator leverages sample precision features evaluated on 1% of the dataset and the Gradient Boosting regression model [33].

Figure 7.6(a) illustrates the mean squared error of the three approaches in estimating the effectiveness of error detection strategies. Our semi-supervised and unsupervised estimators always outperform the baseline approach. The results particularly show that the unsupervised dirtiness profile works sufficiently effective.

Figure 7.6(b) illustrates the mean squared error of the semi-supervised estimator when it leverages different subsets of feature groups as the dirtiness profile. All the feature groups are informative for the task as the semi-supervised estimator estimates most accurately by incorporating all of the *content*, *structure*, and *quality features* (*CF+SF+QF*). The most informative feature group is data *quality features* (*QF*) because it practically captures the error distributions across datasets.

Figure 7.6(c) illustrates the mean squared error of the semi-supervised estimator when it uses different regression models. In particular, we tested *Linear Regression* (*LR*), *Ridge Regression* (*RR*), *Decision Tree Regression* (*DTR*), and *Gradient Boosting Regression* (*GBR*), all implemented in *scikit-learn* Python module [65]. The choice of the regression model does not significantly affect the overall effectiveness of the semi-supervised estimator as the major effectiveness impact is gained due to the dirtiness profile.

Figure 7.6(d) illustrates the mean squared error of the semi-supervised estimator while increasing the number of training dirtiness profiles in the repository. We start to train the regression models with just one dirtiness profile. As the number of training dirtiness profiles increases, the mean squared error decreases as well, because it becomes more likely to find a similar dirtiness profile in the repository. When having around 7 dirtiness profiles as the



(a) Estimation Accuracy (b) Feature Impact Analysis (c) Regression Model Impact Analysis (d) Repository Size Impact Analysis

Figure 7.6: Mean squared error of estimating the effectiveness of error detection strategies with different (a) approaches and sampling rates, (b) feature groups, (c) regression models, and (d) repository sizes.

training set, its effectiveness almost converges. This is promising as the effectiveness does not require unrealistically large repositories.

7.2.7.2 Strategy Filtering with Column Profiles

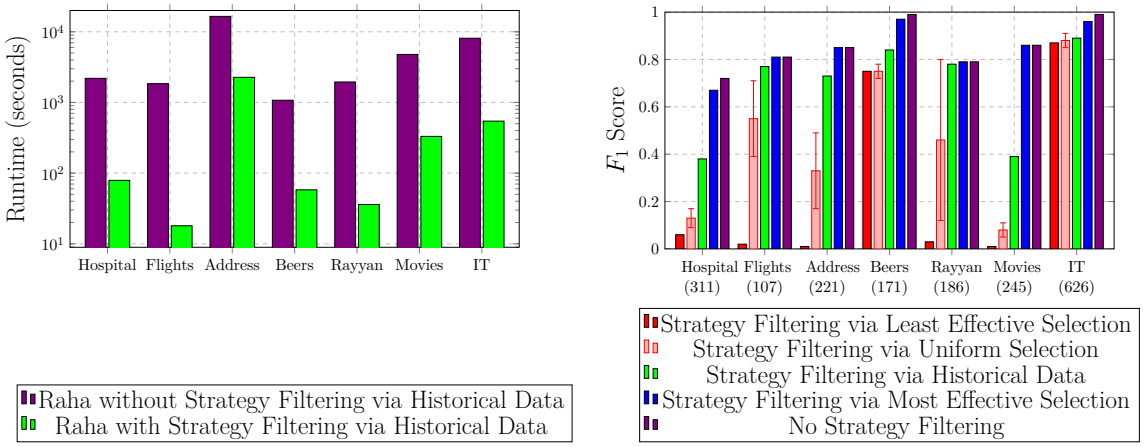
As mentioned in Section 6.1, Raha's runtime depends on the number of base error detection strategies that Raha has to run on the dataset. Thus, we can improve Raha's overall efficiency by estimating the effectiveness of the base error detection strategies upfront and filtering out the ineffective strategies. Since we want to estimate the effectiveness of error detection strategies upfront, we cannot run them on the dataset. Therefore, the quality features of the dirtiness profiles are not available and we need to filter out ineffective error detection strategies with the column profiles and the strategy filtering algorithm introduced in Section 6.1.

We analyze the effect of filtering out irrelevant error detection strategies on the performance of Raha with the assumption of having column profiles, which do not cover the raw output of error detection strategies. In fact, we leverage our strategy filtering algorithm to select the promising error detection strategies upfront and pass these promising strategies to Raha. We report the performance of Raha with and without this strategy filtering step to analyze the effect of limited computational resources for feature extraction.

Figure 7.7(a) shows the runtime of Raha with and without strategy filtering via historical data. The runtime is significantly improved by more than an order of magnitude as Raha needs to run only a fraction of all possible error detection strategies. Note that we exclude the user labeling time as it is user dependent and irrelevant to the machine runtime.

Figure 7.7(b) shows the F_1 score of five different strategy filtering methods. We leverage the ground truth of datasets to evaluate the F_1 score of all the error detection strategies to sort them accordingly. Thus, the first and second methods are Raha with the least and most effective strategies as features. These two extreme methods could be a lower and an upper bound for the effectiveness of any other strategy filtering method. The third method is Raha with a uniform strategy filtering that uniformly picks error detection strategies as the features. Since this method is probabilistic, we repeat it 5 times and report the mean and standard deviation. The fourth and fifth methods are Raha with our strategy filtering via historical data and Raha without any strategy filtering, respectively. Note that the mentioned number of selected strategies is the same for all the strategy filtering methods and is computed by our strategy filtering method.

As shown in Figure 7.7(b), our strategy filtering method via historical data outperforms the least effective strategy selection and the uniform strategy selection methods. The effectiveness of Raha with strategy filtering via historical data is slightly lower than the most effective



(a) Raha's Efficiency with/without Strategy Filtering via Historical Data

(b) Raha's Effectiveness with Different Strategy Filtering Methods

Figure 7.7: Raha's (a) efficiency with/without strategy filtering via historical data and (b) effectiveness with different strategy filtering methods. The numbers of selected strategies are denoted inside the brackets.

strategy selection method. However, our strategy filtering via historical data method can achieve almost the same F_1 score without running or evaluating any strategy on the new dataset. The effectiveness of Raha with no strategy filtering is higher than the most effective strategy selection method. This shows that even ineffective error detection strategies still can add some information to the feature vector.

7.3 Error Correction Experiments

Our error correction experiments aim to answer the following questions on Baran.

1. How does Baran compare to the existing error correction approaches? (Section 7.3.1)
2. How do the error corrector models affect Baran's effectiveness? (Section 7.3.2)
3. How does the tuple sampling approach affect Baran's convergence? (Section 7.3.3)
4. How does the choice of the classifier affect Baran's effectiveness? (Section 7.3.4)
5. How does our method in pretraining the error corrector models affect Baran's effectiveness? (Section 7.3.5)

7.3.1 Baran Versus the Baselines

We compare the performance of Baran with the baselines in terms of effectiveness, efficiency, and human involvement. All the error correction approaches in this section take as input the same correct and complete set of data errors.

7.3.1.1 Effectiveness

Table 7.5 shows the effectiveness of the error correction approaches. Baran outperforms all the baselines in terms of the F_1 score on all the datasets as our two-step formulation of the error correction task achieves both high precision and recall. In fact, since Baran trains a comprehensive set of error corrector models based on the different contexts of data errors,

Table 7.5: Baran's effectiveness in comparison to the baselines.

Approach	Hospital			Flights			Address			Beers		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
KATARA	0.98	0.24	0.39	0.00	0.00	0.00	0.79	0.01	0.02	0.00	0.00	0.00
SCARE	0.67	0.53	0.59	0.57	0.06	0.11	0.10	0.10	0.10	0.16	0.07	0.10
Holistic	0.52	0.38	0.44	0.21	0.01	0.02	0.41	0.31	0.35	0.49	0.01	0.02
HoloClean	1.00	0.71	0.83	0.89	0.67	0.76	0.01	0.01	0.01	0.01	0.01	0.01
Baran	0.88	0.86	**0.87**	1.00	1.00	**1.00**	0.67	0.32	**0.43**	0.91	0.89	**0.90**

Approach	Rayyan			IT			Tax		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
KATARA	0.00	0.00	0.00	0.04	0.01	0.02	0.59	0.01	0.02
SCARE	0.00	0.00	0.00	0.20	0.10	0.13	0.01	0.01	0.01
Holistic	0.85	0.07	0.13	1.00	0.78	0.88	0.96	0.26	0.41
HoloClean	0.00	0.00	0.00	0.01	0.01	0.01	0.11	0.11	0.11
Baran	0.76	0.40	**0.52**	0.98	0.98	**0.98**	0.84	0.78	**0.81**

these models propose a large set of potential corrections that increases the achievable recall bound significantly. Later, when Baran leverages a few user labels to ensemble these correction candidates into the final set of corrections, the high error correction precision is also maintained.

The effectiveness of Baran depends on the amount of value-based, vicinity-based, and domain-based context information that each dataset provides. On context-rich datasets, such as *Hospital* and *Flights*, with many duplicate rows and correlated data columns, Baran can train and ensemble effective error corrector models leveraging all the three data error contexts. Therefore, Baran achieves high F_1 scores on these datasets. In particular, Baran achieves perfect F_1 score on the *Flights* dataset, which has a high degree of redundancy. On the other hand, on datasets with less data error context information, such as *Address*, where the erroneous values are often missing values, Baran cannot fix all the data errors accurately.

Other error correction approaches cannot achieve Baran's F_1 score because they cannot achieve both high precision and recall. KATARA has poor precision because the ambiguity of concepts leads to a mismatch between the dataset and the knowledge base. KATARA also has poor recall because most parts of datasets cannot be matched to knowledge bases. Holistic has poor precision and recall because the provided integrity rules can only fix a portion of data errors accurately. Although HoloClean has relatively high precision and recall on datasets with a high degree of redundancy, such as *Hospital* and *Flights*, it cannot show the same effectiveness on the rest of datasets. On datasets with lower degrees of redundancy or fewer predefined data constraints, HoloClean cannot find the correction of data errors accurately because the actual correction either does not exist anywhere in data or is not covered by data constraints. Similarly, SCARE cannot achieve high precision and recall on datasets with low degrees of data redundancy.

As long as a dataset provides rich contextual information, the data error rate does not affect the effectiveness of Baran significantly. For example, Baran is highly effective on both context-rich datasets *Hospital* and *Flights* although the former has a low (3%) and latter has a high (30%) data error rate. To further analyze the effect of the data error rate, we select our most erroneous dataset *Flights* and generate four without-replacement samples of it with 10%, 30%, 50%, and 70% erroneous data cells. Figure 7.8 shows the F_1 score of the error correction approaches on these four datasets. As the data error rate increases, the F_1 score of all the approaches naturally drops because trustworthy evidence diminishes. However, Baran consistently outperforms all the other approaches because it leverages the remaining scarce trustworthy contexts of data errors more effectively.

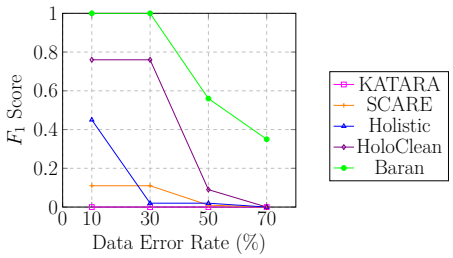


Figure 7.8: Baran's effectiveness on the *Flights* dataset when the data error rate increases.

Table 7.6: Baran's runtime (in seconds) in comparison to the baselines.

Approach	Hospital	Flights	Address	Beers	Rayyan	IT	Tax
KATARA	234	116	5739	180	134	2031	15992
SCARE	76	123	11853	363	216	8717	55495
Holistic	15	10	69	9	8	3	247
HoloClean	148	39	17582	96	112	885	25778
Baran	23	22	11073	114	26	247	11936

7.3.1.2 Human Involvement

Figure 7.9 shows the effectiveness of the error correction approaches with respect to the number of labeled tuples. The F_1 score of Baran quickly converges with only a few labeled tuples and outperforms the F_1 score of all the other approaches. This fast convergence is due to the Baran's tuple sampling and feature generation methods that generate a large number of informative training data points with a few user labels.

Since KATARA, SCARE, Holistic, and HoloClean do not leverage user labels, their F_1 score is independent of the number of labeled tuples. However, we argue that they leverage human supervision in other more tedious forms. KATARA needs the user to provide related knowledge bases. SCARE needs the user to tune statistical parameters. Holistic and HoloClean need the user to provide the correct and complete set of integrity rules and matching dependencies. In contrast, Baran does not need any user-provided rules or parameters and quickly converges with only a few user labels.

7.3.1.3 Efficiency

Table 7.6 shows the runtime of the error correction approaches in seconds. Although efficiency is not the main concern of Baran, it displays a competitive runtime in comparison to the other baselines. The reported runtime captures the online phase of Baran as the offline pretraining phase is totally independent of the input dataset. Furthermore, we exclude the user labeling time for Baran as it is user dependent and irrelevant to the machine runtime.

Optimizing machine runtime efficiency has not been the main goal of data cleaning systems as optimizing effectiveness and human involvement are more important objectives [2, 74]. However, it is important to develop systems that can work in a reasonable runtime.

7.3.2 Error Corrector Model Impact Analysis

We conducted an ablation experiment on the error corrector models to better understand their effect on the overall effectiveness of Baran. First, we run Baran with all the default error corrector models (row *All* in Table 7.7). Then, similar to the study in Section 7.3, we exclude each type of models, one at a time, to analyze its impact. For example, *All - VaM* means that Baran leverages all the error corrector models but the value-based ones. Finally,

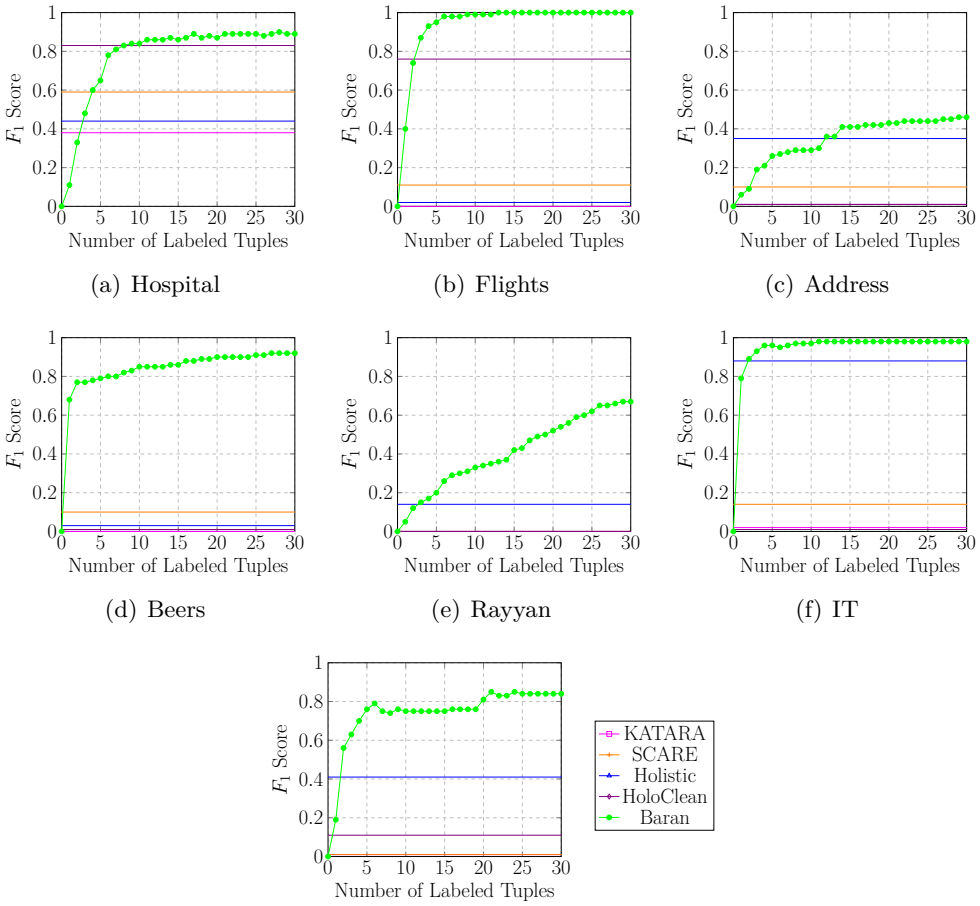


Figure 7.9: Baran's effectiveness with different numbers of labeled tuples.

we also evaluate the effectiveness of Baran with all the default models together with custom dataset-specific models (row *All + CM*) obtained from the data constraints in Table 7.1. Baran leverages these data constraints as hard-coded error corrector models that overwrite our default models, if necessary.

As shown in Table 7.7, Baran has the highest F_1 score with all the default error corrector models on most of the datasets. By collecting the proposed potential corrections from all the error corrector models, Baran has more context information to fix data errors.

However, excluding one type of error corrector models can improve the F_1 score on some datasets. Excluding vicinity-based or domain-based models improves the F_1 score on the *Tax* dataset. On this large dataset with thousands of data rows, these models propose thousands of clean values from the active domain of the data error as potential corrections. Learning to find the actual correction among this huge search space needs more learning iterations and user labels. Excluding value-based models improves the F_1 score on the *Hospital* dataset. Since this dataset has randomly imposed typos, the value-based models cannot effectively learn value-based corrections from this randomness.

We can also observe the importance of vicinity-based models in fixing inter-column dependency violations. Excluding the vicinity-based error corrector models significantly drops the F_1 score on datasets with high inter-column dependencies, such as *Hospital* and *Flights*. This decline shows that Baran effectively fixes inter-column dependency violations with including vicinity-based models.

Table 7.7: Baran's effectiveness with different error corrector models: value-based models (VaM), vicinity-based models (ViM), domain-based models (DoM), all default models (All), and custom dataset-specific models (CM).

Error Corrector Models	Hospital			Flights			Address			Beers		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
All - VaM	0.95	0.88	0.91	1.00	1.00	1.00	0.61	0.06	0.11	0.67	0.42	0.52
All - ViM	0.64	0.31	0.42	0.08	0.06	0.07	0.40	0.25	0.31	0.87	0.86	0.86
All - DoM	0.88	0.87	0.87	1.00	1.00	1.00	0.56	0.25	0.35	0.91	0.88	0.89
All	0.88	0.86	0.87	1.00	1.00	1.00	0.67	0.32	0.43	0.91	0.89	0.90
All + CM	0.90	0.89	0.89	1.00	1.00	1.00	0.67	0.32	0.43	0.91	0.89	0.90

Error Corrector Models	Rayyan			IT			Tax		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
All - VaM	0.19	0.07	0.10	0.98	0.95	0.96	0.44	0.24	0.31
All - ViM	0.48	0.34	0.40	0.98	0.98	0.98	0.88	0.88	0.88
All - DoM	0.54	0.35	0.42	0.98	0.98	0.98	0.90	0.81	0.85
All	0.76	0.40	0.52	0.98	0.98	0.98	0.84	0.78	0.81
All + CM	0.76	0.40	0.52	0.98	0.98	0.98	0.84	0.78	0.81

Adding custom dataset-specific models to the default set of error corrector models does not affect the F_1 score on most of the datasets as our default models are general enough and already cover these prevalent data constraints. For example, one-attribute to one-attribute functional dependencies are already incorporated into Baran due to the vicinity-based models.

7.3.3 Tuple Sampling Impact Analysis

We analyze the impact of our tuple sampling method on the effectiveness of Baran. In particular, we compare two versions of our system with two different sampling methods. the *Uniform sampling* method selects erroneous tuples for user labeling according to a uniform probability distribution. *Our tuple sampling* method selects tuples according to their informativeness for the classifiers as described in Section 5.2.

As shown in Figure 7.10, our tuple sampling method speeds up the convergence of Baran. The superiority of our tuple sampling method is more obvious on datasets with more randomly dispersed data errors, such as *Hospital* and *Rayyan*. On these datasets, the classifiers need to have enough informative labeled data errors from each data column and data error type to be able to fix all the data errors accurately. That is why our tuple sampling method that oversamples the underlabeled data columns and data error types converges faster.

7.3.4 Classifier Impact Analysis

We analyze the impact of the classifier on the effectiveness of Baran. In particular, we tested *AdaBoost*, *Decision Tree*, *Gradient Boosting*, and *Stochastic Gradient Descent*, all implemented in *scikit-learn* Python module [65]. The hyperparameter tuning task can slightly improve the effectiveness of these classifiers. However, as we aim at building a general-purpose system and fine-tuning Baran for each dataset is not our goal, we skip the straightforward hyperparameter tuning task and simply run each classifier with its recommended hyperparameters.

Table 7.8 shows that the choice of the classifier does not have a significant impact on the effectiveness of Baran. Although on some datasets, such as *Address*, the F_1 score varies more, there are always multiple classifiers that achieve almost the same F_1 score. In our current prototype, we deploy AdaBoost because it is an advanced ensemble classifier [32], which is less susceptible to overfitting [76].

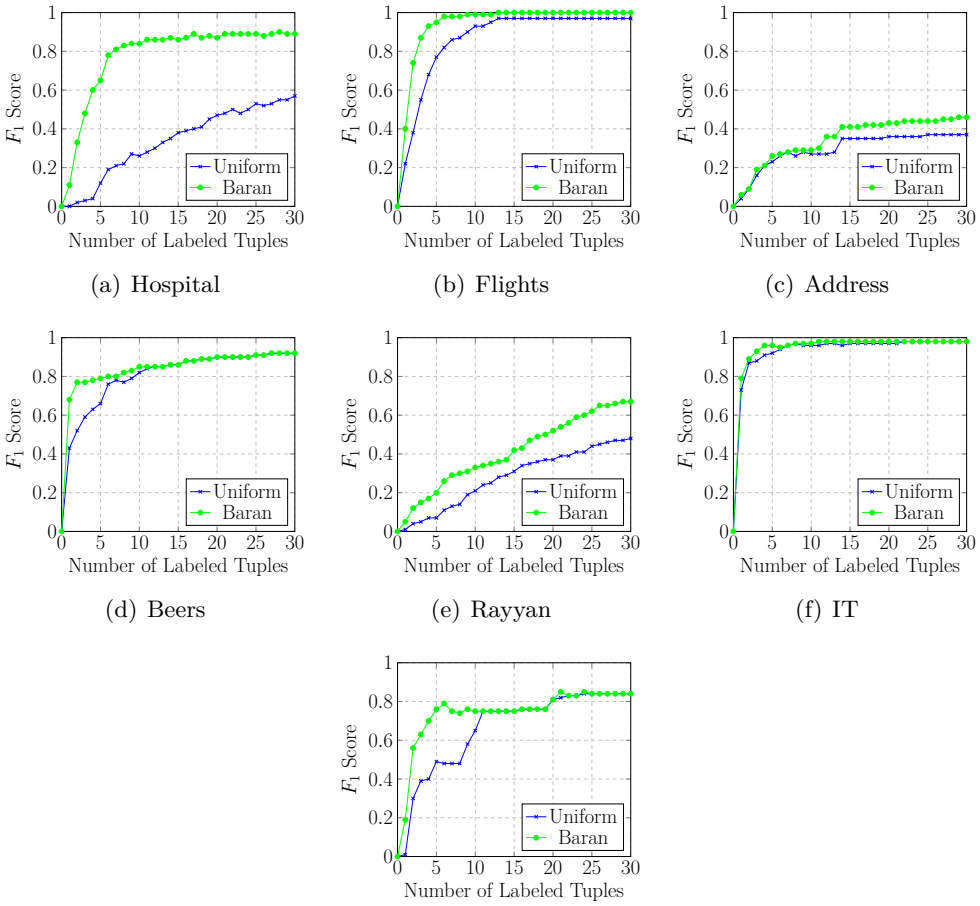


Figure 7.10: Baran's effectiveness with different tuple sampling methods.

7.3.5 Pretraining Models Impact Analysis

We analyze the impact of pretraining the error corrector models on the effectiveness of Baran. We pretrained the value-based models on the revision history of more than 300000 Wikipedia pages (row *Baran with Pretraining* in Table 7.9).

As shown in Table 7.9, pretraining the value-based models improves the F_1 score more significantly on datasets with prevalent syntactic data issues, such as *Hospital* and *Rayyan*. Pretraining provides more evidence for the classifiers to predict the actual correction. In particular, the pretrained value-based models generate additional correction candidates that enable Baran to converge to its reported F_1 score with fewer than 20 labeled tuples for the same F_1 score. On the *Hospital* dataset, pertaining generates 455390 new correction candidates. As a result, we just need 13 labeled tuples to achieve the same F_1 score achieved with 20 labeled tuples. On the *Rayyan* dataset, pertaining generates 77569 new correction candidates. With pretraining, we reach the same F_1 score that previously needed 20 labeled tuples with only 18 labeled tuples. On the *Tax* dataset, pretraining generates 7134254 new correction candidates. As a result, we just need 19 labeled tuples to achieve the same F_1 score achieved with 20 labeled tuples.

Contrary, the effectiveness improvement is minor on datasets like *IT*, where most of the erroneous values are missing values. Here, the value-based models cannot propose potential corrections effectively, regardless of being pretrained or not.

Table 7.8: Baran's effectiveness with different classifiers.

Classifier	Hospital			Flights			Address			Beers		
	P	R	F	P	R	F	P	R	F	P	R	F
AdaBoost	0.88	0.86	0.87	1.00	1.00	1.00	0.67	0.32	0.43	0.91	0.89	0.90
Decision Tree	0.88	0.85	0.86	1.00	1.00	1.00	0.70	0.34	0.46	0.91	0.89	0.90
Gradient Boosting	0.94	0.68	0.79	1.00	1.00	1.00	0.63	0.12	0.20	0.92	0.81	0.86
Stochastic Gradient Descent	0.95	0.92	0.93	1.00	1.00	1.00	0.66	0.25	0.36	0.95	0.87	0.91

Classifier	Rayyan			IT			Tax		
	P	R	F	P	R	F	P	R	F
AdaBoost	0.76	0.40	0.52	0.98	0.98	0.98	0.84	0.78	0.81
Decision Tree	0.62	0.34	0.44	0.98	0.98	0.98	0.74	0.73	0.73
Gradient Boosting	0.66	0.41	0.51	0.99	0.98	0.98	0.97	0.59	0.73
Stochastic Gradient Descent	0.59	0.21	0.31	0.99	0.98	0.98	0.83	0.63	0.72

Table 7.9: Baran's effectiveness with and without pretraining the error corrector models.

Approach	Hospital			Flights			Address			Beers		
	P	R	F	P	R	F	P	R	F	P	R	F
Baran	0.88	0.86	0.87	1.00	1.00	1.00	0.67	0.32	0.43	0.91	0.89	0.90
Baran with Pretraining	0.94	0.88	0.91	1.00	1.00	1.00	0.67	0.32	0.43	0.94	0.87	0.90

Approach	Rayyan			IT			Tax		
	P	R	F	P	R	F	P	R	F
Baran	0.76	0.40	0.52	0.98	0.98	0.98	0.84	0.78	0.81
Baran with Pretraining	0.80	0.44	0.57	0.98	0.98	0.98	0.95	0.73	0.83

7.4 End-to-End Data Cleaning Experiments

Our end-to-end data cleaning experiments aim to answer the following question on Raha and Baran: How does an end-to-end data cleaning pipeline with Raha and Baran compare to other alternative end-to-end pipelines?

Although error detection and error correction have been considered as two orthogonal tasks in literature [2, 41, 74, 93], it is important to also analyze the end-to-end data cleaning effectiveness. Naturally, the effectiveness of the downstream error correction task depends on the effectiveness of the upstream error detection task. Typically, the error detection recall is the upper bound of the error correction recall as we might only fix a subset of data errors that are already detected [74].

We compare the effectiveness of three end-to-end data cleaning scenarios.

1. **Perfect error detection + Baran.** The user perfectly detects all the data errors of the dataset and then Baran corrects them (row *Perfect ED + Baran* in Table 7.10).

2. **Raha + perfect error correction.** Raha detects data errors of the dataset and then the user corrects all the detected data errors perfectly (row *Raha + Perfect EC*). The effectiveness of this virtual error correction approach is the upper bound of error correction systems.

3. **Raha + an error correction system.** Raha detects data errors and then an error correction system corrects the detected data errors. In particular, we study the effectiveness of three end-to-end data cleaning pipelines. All the error correction systems in the following end-to-end pipelines take the same set of detected data errors.
   (a) **Raha + HoloClean.** Raha detects data errors and then HoloClean corrects these detected data errors (row *Raha + HoloClean*).
   (b) **Raha + Baran.** Raha detects data errors and then Baran corrects these detected data errors (row *Raha + Baran*). Here, Raha and Baran work orthogonal and each of them separately asks the user to label 20 tuples per dataset.

Table 7.10: The end-to-end data cleaning effectiveness with imperfect and perfect error detection (ED) and error correction (EC).

Approach	Hospital			Flights			Address			Beers		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Perfect ED + Baran	0.88	0.86	0.87	1.00	1.00	1.00	0.67	0.32	0.43	0.91	0.89	0.90
Raha + Perfect EC	0.98	0.58	0.73	0.97	0.75	0.85	0.83	0.85	0.84	0.98	1.00	0.99
Raha + HoloClean	0.19	0.41	0.26	0.08	0.16	0.11	0.01	0.01	0.01	0.01	0.01	0.01
Raha + Baran	0.89	0.52	0.66	0.88	0.53	0.66	0.57	0.32	0.41	0.93	0.87	0.90
Raha + Baran (In)	0.95	0.52	0.67	0.84	0.56	0.67	0.53	0.32	0.40	0.93	0.87	0.90

Approach	Rayyan			IT			Tax		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Perfect ED + Baran	0.76	0.40	0.52	0.98	0.98	0.98	0.84	0.78	0.81
Raha + Perfect EC	0.83	0.79	0.81	0.99	0.98	0.98	0.97	0.98	0.97
Raha + HoloClean	0.00	0.00	0.00	0.01	0.01	0.01	0.11	0.11	0.11
Raha + Baran	0.50	0.27	0.35	0.98	0.96	0.97	0.84	0.66	0.74
Raha + Baran (In)	0.44	0.21	0.28	0.99	0.97	0.98	0.84	0.77	0.80

(c) **Raha + Baran (integrated).** Raha detects data errors by asking the user to label 20 tuples per dataset and then it passes these user labels along with the detected data errors to Baran (row *Raha + Baran (In)*). Therefore, Baran does not take any other labels from the user in this integrated end-to-end pipeline and just reuses Raha's user labels.

Table 7.10 shows the effectiveness of these end-to-end data cleaning pipelines. Imperfect error detection/correction naturally leads to a slight drop of end-to-end data cleaning effectiveness. In particular, imperfect error detection leads to a slight drop of the Baran's error correction effectiveness. This decline is minor on most of the datasets, such as *Beers* and *IT*, as Baran still achieves a very close F_1 score to the perfect error correction.

Both end-to-end data cleaning pipelines with Raha and Baran achieve almost the same F_1 score and both clearly outperform the pipeline with HoloClean. The semi-supervised learning nature of Baran enables us to learn corrections with respect to all data error contexts and user labels, even if the set of detected data errors is not correct and complete in the first place.

Interestingly, the integrated pipeline with Raha and Baran achieves even higher F_1 score on large datasets, such as *Tax*. This is promising as it shows Raha's clustering-based sampling method is effective enough to sample informative tuples for both error detection and correction tasks and we do not need separate user labels for Baran.

7.5 Summary

We extensively evaluated our proposed data cleaning approach. In summary, we observed the followings:

- **Performance.** Raha and Baran significantly outperform the baseline data cleaning approaches. In terms of effectiveness, Raha and Baran achieve both high precision and recall due to our novel two-step task formulation. In terms of human involvement, Raha and Baran leverage only a few user labels due to their effective feature representation and tuple sampling method. In terms of efficiency, Raha and Baran terminate in a reasonable time due to their parallelization and pruning techniques. Not only we can achieve state-of-the-art error detection and correction performances with Raha and Baran, respectively, but also we can achieve the state-of-the-art end-to-end data cleaning performance with serializing Raha and Baran in an end-to-end data cleaning pipeline.

- **Feature representation.** Raha and Baran effectively represent each potential data error/correction by collecting the output of various base error detectors/correctors. Overall, they achieve the best effectiveness by incorporating all the features as the resulting feature vectors will be more expressive.
- **Tuple sampling.** The performance of Raha and Baran converges faster using our tuple sampling methods as these methods sample the most informative tuples for learning error detection/correction operations.
- **Transfer learning.** Transfer learning optimizes the error detection/correction performance as Raha and Baran can learn from previous data cleaning efforts on historical datasets. The more similar historical datasets we collect, the better performance Raha and Baran can achieve using transfer learning.
- **Classifier.** The performance of Raha and Baran does not depend on the choice of the classifier as the main impact is due to the feature representation and tuple sampling method.
- **User labeling error.** Although user labeling errors naturally decrease the data cleaning performance, Raha's effectiveness does not collapse with limited user labeling errors as Raha can compensate erroneous user labels with more redundant user labels.
- **System scalability.** Raha is scalable as its runtime increases linearly and its F_1 score stays the same when the number of data rows/columns increases.

8

Raha and Baran in Action

We implemented our data cleaning systems, Raha and Baran, together with the transfer learning methods in the Python 3.6 programming language on the top of Python's data science libraries. In particular, we leverage

- the *pandas* library to store and process relational data frames.
- the *NumPy* and *SciPy* libraries to conduct scientific computing operations on multidimensional arrays and matrices.
- the *scikit-learn* library to deploy machine learning algorithms.
- the *Matplotlib* and *PrettyTable* libraries to visualize the results.
- the *py7zr* library, *Beautiful Soup* library, and *MWParserFromHell* package to decompress and parse the Wikipedia page revision dump files.
- the *Natural Language Toolkit (NLTK)* library, *difflib* module, and *re* module to process texts.
- the *multiprocessing* package to parallelize the workflow.
- the *Jupyter Notebook* environment to build an interactive user interface.

The implementations, documentations, and datasets are all available online[1].

We first briefly review the implementation challenges. We then demonstrate how to build an end-to-end data cleaning pipeline with Raha and Baran. Next, we elaborate the user labeling process in a case study. Finally, we summarize the chapter.

8.1 Implementation Challenges

The high-level Python programming language and its data science libraries provide high readability, ease of use, and fast development opportunities. However, it is still challenging to implement Raha and Baran efficiently. The complex workflows of Raha and Baran include a series of algorithmic steps, including running base error detectors/correctors, featurizing their outputs, sampling tuples, and learning to predict data errors/corrections. Implementing

[1] `https://github.com/BigDaMa/raha`

these workflows in a way that all these steps can run efficiently is not trivial in a high-level programming language like Python. This issue is more problematic on large datasets, where these steps must run on thousands of tuples.

We parallelized the workflows of Raha and Baran with the *multiprocessing* package to address this issue. Raha runs all the error detection strategies in parallel processes. Baran featurizes each pair of a data error and a correction candidate in parallel processes. This way, both Raha and Baran terminate the data cleaning tasks, even on large datasets, in a reasonable time.

8.2 Building End-to-End Data Cleaning Pipelines

Figure 8.1 shows an end-to-end data cleaning pipeline with Raha and Baran in a Jupyter Notebook. As depicted, the workflow is modularized and each step has a dedicated application programming interface (API). This modularized design provides the usage flexibility for the user as the user can easily orchestrate different steps of the workflow. For example, the user can decide to run Raha's or Baran's tuple sampling method to sample a tuple.

The user can also continuously monitor various indicators, such as the data cleaning progress and the data error distribution (Figure 8.2). The iterative data cleaning process continues until the user terminates it and stores the final cleaned dataset.

The only step that needs user involvement is the tuple labeling process. In the depicted pipeline, we leverage a *label_with_ground_truth* method to label the sampled tuples via the ground truth of the dataset. Since the ground truth is usually not available in practice, the user has to instead label the sampled tuples manually.

8.3 Case Study: The User Labeling Process

We provide a case study to elaborate the user labeling procedure. Table 8.1 shows the 20 tuples that Raha sampled on the *Flights* dataset. The user takes one tuple at a time and labels its data cells. Ideally, the user should label all the red data cells as dirty and the rest as clean for Raha. For Baran, the user should provide the correct value of the marked data errors as well.

The *Flights* dataset contains various data error types. Missing values, such as the scheduled departure time in tuple *3*, and formatting issues, such as "11:25aDec 1" as the scheduled departure time in tuple *4*, can easily be identified by the user. The user usually identifies and fixes these data error types using either domain expertise or some sort of master data.

However, the tuples also include erroneous values that are not easily detectable. For example, the reported departure time "2:03 p.m." in tuple *3* does not look erroneous without further investigation. Although this data value seems correct, it is not the correct departure time of the flight "AA-1279-DFW-PHX". This value violates the functional dependency *Flight* $\rightarrow$ *Actual Departure Time*. In this particular case, the user needs to look at master data and check the actual departure time of the flight "AA-1279-DFW-PHX". Note that knowing the violated functional dependency is not a guarantee for a correct label as typically multiple data cells are involved in the violation of a functional dependency, e.g., left-hand-side or right-hand-side value. Thus, not every involved data cell in a violation can be considered as a data error.

Raha and Baran highlight low-level information that support the user in the tuple annotation process. Raha visualizes the clusters of data cells in each data column (Figure 8.3). Once the user clicks on a data point, the user can drill down the results and see the detailed information

An End-to-End Data Cleaning Pipeline

Error Detection with Raha

In [ ]:

```
import raha

app_1 = raha.Detection()
app_1.LABELING_BUDGET = 20

dataset_dictionary = {
    "name": "flights",
    "path": "/home/mohammad/Desktop/raha/datasets/flights/dirty.csv",
    "clean_path": "/home/mohammad/Desktop/raha/datasets/flights/clean.csv"
}
d = app_1.initialize_dataset(dataset_dictionary)

app_1.run_strategies(d)
app_1.generate_features(d)
app_1.build_clusters(d)
while len(d.labeled_tuples) < app_1.LABELING_BUDGET:
    app_1.sample_tuple(d)
    if d.has_ground_truth:
        app_1.label_with_ground_truth(d)
app_1.propagate_labels(d)
app_1.predict_labels(d)
```

Error Correction with Baran

In [ ]:

```
app_2 = raha.Correction()
app_2.LABELING_BUDGET = 20

d = app_2.initialize_dataset(d)

app_2.initialize_models(d)
while len(d.labeled_tuples) < app_2.LABELING_BUDGET:
    app_2.sample_tuple(d)
    if d.has_ground_truth:
        app_2.label_with_ground_truth(d)
    app_2.update_models(d)
    app_2.generate_features(d)
    app_2.predict_corrections(d)
```

Figure 8.1: An end-to-end data cleaning pipeline with Raha and Baran in a Jupyter Notebook.

related to that particular data cell, such as which error detection strategies have marked this data cell as a data error. Baran reports the likelihood of user-provided corrections based on the confidence of the error corrector models and the prevalence of the observed correction in the indexed Wikipedia page revision history. For example in Figure 8.4, three error corrector models propose the value “7:45 p.m.” as the correction of the erroneous value “7:45pm”. The most confident model (depicted as *Model 12*) is a substring adder that has learned to add substrings “ ” and “.” to the erroneous values like “7:45pm” in order to fix the format. This particular correction matches exactly to 3 corrections and matches due to a similar pattern to 142 corrections in the Wikipedia logs. These statistics come from the value-based error corrector models that have been pretrained on the Wikipedia page revision history.

Figure 8.5 shows the F_1 score of Raha on the erroneous data columns of the *Flights* dataset. Whenever a labeled tuple covers a new unseen data error type, the F_1 score is improved

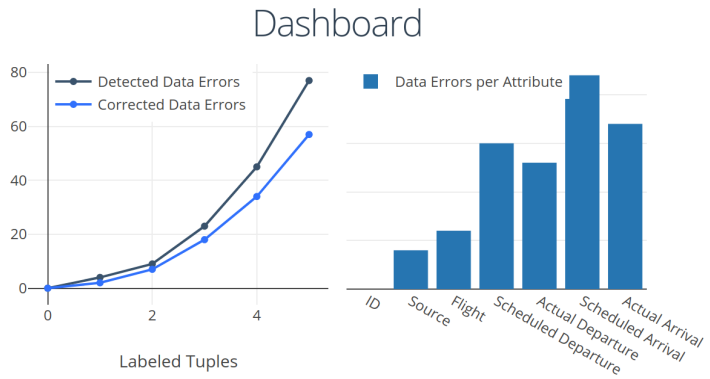


Figure 8.2: The user dashboard displays the data cleaning progress and the data error distribution.

Table 8.1: The 20 tuples that the user labeled on the *Flights* dataset for Raha. The red data cells are dirty and the rest are clean.

ID	Source	Flight	Scheduled Departure Time	Actual Departure Time	Scheduled Arrival Time	Actual Arrival Time
1	aa	AA-3-JFK-LAX	12:00 p.m.	12:11 p.m.	3:15 p.m.	3:16 p.m.
2	usatoday	AA-1886-BOS-MIA	10:45 a.m.		2:20 p.m.	
3	airtravelcenter	AA-1279-DFW-PHX		2:03 p.m.		3:13 p.m.
4	mytripandmore	AA-1544-SAN-ORD	11:25aDec 1	11:20aDec 1	5:25 p.m.	4:56 p.m.
5	weather	UA-2704-DTW-PHX	11:15 a.m.		1:40 p.m.	
6	flightaware	AA-431-MIA-SFO	8:35 a.m.	8:51 a.m.	11:22 a.m.	11:33 a.m.
7	panynj	AA-404-MIA-MCO	6:45 a.m.	6:58 a.m.	7:45 a.m.	7:32 a.m.
8	quicktrip	AA-3823-LAX-DEN	9:00 p.m.	9:06 p.m. (Estimated)	12:15 a.m.	11:49 p.m. (Estimated)
9	foxbusiness	AA-3-JFK-LAX	12:00 p.m.	12:12 p.m.	3:15 p.m.	3:16 p.m.
10	orbitz	UA-6273-YYC-SFO	7:35aDec 1	7:27aDec 1	9:43aDec 1	8:45aDec 1
11	flightarrival	UA-828-SFO-ORD	11:08 p.m.		5:11 a.m.Dec 02	
12	travelocity	UA-3515-IAD-MSP	Not Available	8:24 a.m.	Not Available	9:56 a.m.
13	aa	AA-3063-SLC-LAX	8:20 p.m.	8:39 p.m.	9:20 p.m.	
14	helloflight	AA-1733-ORD-PHX		7:59 p.m.		10:31 p.m.
15	orbitz	AA-616-DFW-DTW	10:00aDec 1	9:59aDec 1	12:35 p.m.	1:27 p.m.
16	gofox	UA-2906-PHL-MCO	3:50 p.m.	4:46 p.m.	6:23 p.m.	6:35 p.m.
17	weather	AA-2268-PHX-ORD	7:15 a.m.	7:23 a.m.	11:35 a.m.	11:04 a.m.
18	flylouisville	AA-466-IAH-MIA	6:00 a.m.	6:08 a.m.	9:20 a.m.	9:05 a.m.
19	panynj	AA-3-JFK-LAX	12:00 p.m.	12:12 p.m.	3:15 p.m.	3:16 p.m.
20	flylouisville	UA-1500-IAH-GUA		9:16 a.m.		11:45 a.m.

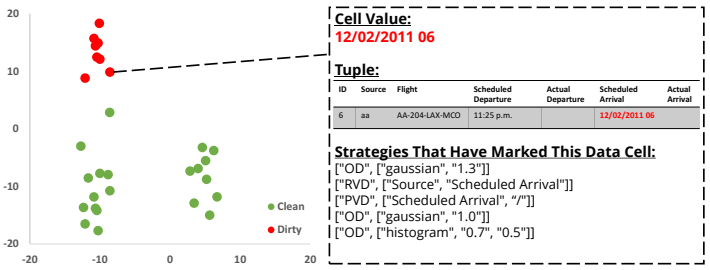


Figure 8.3: These 2D projected clusters contain either clean (green) or dirty (red) data cells for one data column. By clicking on a point, the user can inspect the actual value and the error detection strategies that marked this particular data cell as a data error.

significantly. For example, by labeling tuple 2, the F_1 score improves significantly on the data columns *Actual Departure Time* and *Actual Arrival Time*, because Raha learns the missing value data error type for these data columns. Labeling tuple 3 improves the F_1 score on the data column *Actual Departure Time* as well, because the classifier of this data column learns that the value is erroneous due to the violation of a functional dependency. In fact, the classifier learns that the functional dependency checker feature $s_{\text{Flight}\rightarrow\text{Actual Departure Time}}$ is an important feature for identifying data errors in the data column *Actual Departure Time*. Labeling tuple 4 also improves the F_1 score on the data column *Actual Departure Time*, because

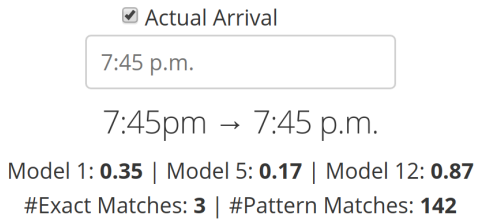


Figure 8.4: The user can inspect the confidence of error corrector models for a particular correction and the number of exactly/approximately matched corrections from the Wikipedia page revision history.

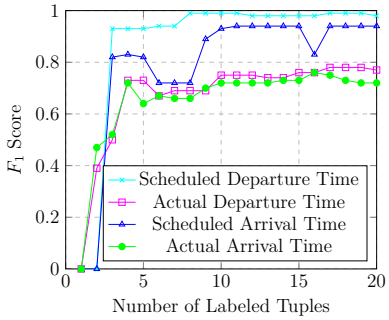


Figure 8.5: Raha's effectiveness on different data columns of the *Flights* dataset.

the classifier of this data column learns that the values with the same type of formatting as "11:20aDec 1" are not desired. Interestingly, after labeling these four tuples, the F_1 score of Raha on the data column *Actual Departure Time* almost converges, because there are no more different data error types in this data column.

The F_1 score of Raha might also slightly drop on some data columns upon labeling some tuples. For example, labeling tuple 6 slightly drops the F_1 score of Raha on the data column *Scheduled Arrival Time*. While the classifier of this data column learns that the value "11:22 a.m." is a data error, the classifier overfits and labels some clean values with a similar format as data errors. However, the F_1 score of the classifier again increases in later iterations by observing a few more similar clean values. In fact, the classifier learns that, in contrast to other clean data cells, the data cell with the value "11:22 a.m." is erroneous due to a functional dependency violation.

8.4 Summary

We demonstrated our data cleaning systems, Raha and Baran, in practice. In particular, we elaborated their implementation challenges, their usage in an end-to-end data cleaning pipeline, and their user labeling process in a case study.

9 Conclusion

We proposed a novel approach for the data cleaning problem. We first discuss our approach and its limitations. We then mention potential future research directions.

9.1 Discussion

We proposed a novel data cleaning approach that effectively represents data errors/corrections and learns to generalize the error detection/correction operation from a few user-annotated data values to the rest of the dataset.

Our data cleaning approach addresses the error detection and correction tasks with a novel two-step formulation. First, our approach runs a set of base error detectors and correctors on the dataset. These base algorithms leverage various data error contexts to detect and correct data errors. Our approach collects the output of these base error detector and correctors into feature vectors that represent the data quality issues and the required data cleaning treatments. Then, the approach incorporates human supervision by asking the user to annotate the most informative tuples. The approach leverages these user labels to train classifiers that predict the rest of data errors/corrections. Our approach can also leverage historical data to optimize this two-step data cleaning procedure by estimating the effectiveness of the base error detectors and pretraining the base error correctors.

Our data cleaning approach is semi supervised, example driven, and configuration free. The approach is semi supervised as it learns to clean data with a large number of unlabeled and a few labeled data points. The approach is example driven as it receives these a few labeled data points from the user through examples. Finally, our approach is configuration free as the user does not need to provide any input rules or parameters to configure it.

We designed an end-to-end data cleaning pipeline according to this approach that takes a dirty dataset as input and outputs a cleaned dataset. Our pipeline leverages user feedback, a set of data cleaning algorithms, and historical data, if available. Internally, our pipeline consists of an error detection system (i.e., Raha), an error correction system (i.e., Baran), and a transfer learning engine.

Our data cleaning systems are effective and efficient, and involve the user minimally in our experiments. Raha and Baran achieve both high precision and recall due to our novel two-step task formulation that significantly outperforms existing data cleaning approaches. Raha and

Baran achieve competitive runtime as well due to our parallelization and transfer learning optimizations. Finally, Raha and Baran minimize the whole human supervision tasks into only annotating a few tuples due to our effective feature representation and tuple sampling methods.

While our end-to-end data cleaning pipeline delivers outstanding performance in a hassle-free way, it has sill some limitations. These limitations are mainly due to the assumptions that we have to hold on the inputs of our end-to-end data cleaning pipeline, i.e., the user feedback, the dirty dataset, and the base error detectors/correctors.

9.1.1 User Annotation Correctness

Most data cleaning tasks need to incorporate human supervision as the desired data curation might be use-case dependent or subjective. The data cleaning process might leverage human supervision in multiple tasks, such as providing integrity rules, statistical parameters, and data annotations.

Similar to other data cleaning approaches, our systems are also dependent on the quality of human supervision. In other words, if human supervision is erroneous itself, the data cleaning task will be flawed as well. This problem has been identified in the previous rule-based approaches in the form of obsolete [87] or inaccurate [11] integrity constraints. In our example-driven systems, this problem might appear in the form of wrong user-provided error/correction examples. However, we argue that Raha and Baran are more robust against this issue than rule-based approaches because of three reasons.

1. Example-based supervision is less complicated and thus less error-prone than rule generation. For example, it is more intuitive for the user to fix the erroneous value "November (16, 1990)" to "16 November 1990" instead of writing a regular expression rule to do so. Note that a rule is always a generalization of many examples and has to undergo several tests before it can be considered as a trustworthy business rule [85].

2. Raha and Baran limit all means of human supervision, such as providing rules, parameters, and annotations, to just labeling a few tuples. While, in existing approaches, the number of user labels scales with the size of dataset (e.g., $1\% - 10\%$ of the dataset [41, 86]), in our systems, the number of user labels scales with the number of data error types of dataset (i.e., $10 - 20$ tuples). This limited human interaction naturally diminishes the possibility of human mistakes as well. At the same time, the user can always skip examples that are hard to label.

3. When learning through examples, one can compensate human labeling errors by simply considering more user-provided examples. Redundant user-provided examples allow the learning model to refine the underlying error detection/correction logic. rule-based approaches however are not that flexible as they directly enforce input rules. Thus, wrong input rules are harder to be compensated by other user-provided refining rules.

We assume that the user is accurate, i.e., the user does not annotate values wrongly, which is a realistic assumption considering the low required number of user labels. However, similar to previous approaches, this assumption could create a single point of failure in our pipeline, i.e., the user annotation procedure. However, as we showed in the experiments, the limited number of user labeling errors does not significantly decrease the performance of our data cleaning systems.

Alternatively, we can switch our one accurate user with a set of non-accurate crowd workers whose provided noisy annotations need to be aggregated. This problem, which has been widely studied in the crowdsourcing [13] and weak supervision [73] areas, is orthogonal to our data cleaning task and therefore it is beyond the scope of this thesis.

9.1.2 Assumptions on the Input Dataset

We hold a minimal set of basic assumptions on the input datasets and their data errors to allow our data cleaning systems to work with the prevalent types of datasets and data errors. However, even this minimal set of assumptions could still exclude a few types of datasets and data errors.

We consider the input dataset to be a relational dataset and define the data errors, their contexts, and their categories based on the structure of this data model, accordingly. Therefore, for any other data model, such as nested structures, we need to redefine these concepts. Even for the relational datasets, we assume that both the dataset and its ground truth must have the same size, which hinders tuple generation and removal operations. In accordance with literature, our data cleaning systems do not support these operations as they are more relevant to the orthogonal duplicate detection and entity matching tasks.

We also define data errors as data values inside a dataset that deviate from the ground truth. While this is the case in many data cleaning tasks, we might need to redefine data errors based on data quality requirements of the user. For example, in statistical data cleaning tasks, such as cleaning time series data, it is often does not matter that a sensor dataset reports a temperature value as 22.4 or 22.5 degree; Both are correct values although they might slightly deviate from the ground truth. Therefore, our general definition of data errors, which considers even a slight deviation between the dataset and its ground truth as a data error, might not be appropriate for those data cleaning tasks.

9.1.3 Completeness of the Base Error Detectors/Correctors

Our data cleaning systems are as complete in error detection and correction as the set of our base error detectors and correctors is. In fact, Raha and Baran can only detect and correct those data errors that have been marked and fixed by at least one base error detector and corrector. If our base error detectors/correctors are not able to mark/fix a data error type, then our data cleaning systems cannot detect/correct this data error type either.

We design a set of simple and general base error detectors/correctors that leverage all data error contexts to fix prevalent data error types. As we showed in the experiments, this default set is enough to achieve high performance across several real-world datasets. However, by any means, we do not claim that our proposed set of base error detectors and correctors are complete for any arbitrary data cleaning task. In some especial data cleaning tasks, such as cleaning time series or satellite data, where the notion of data error is different, the user might need to enrich our default set of base error detectors/correctors with domain-specific base algorithms. Therefore, like many other machine learning approaches, there is no guarantees that the user can achieve the desired performance with these default set of base error detectors/correctors and a limited number of user labels in any arbitrary data cleaning tasks.

9.2 Future Work

Despite the promises of our novel data cleaning approach, there are still possible future research directions for improvement.

9.2.1 Supporting the User During the Tuple Annotation Task

Our data cleaning systems expect the user to take a tuple at a time and annotate its data cells. While it is not unrealistic, it could be still tedious for the user to identify/fix some semantic data error types, such as functional dependency violations, in the absence of contextual metadata.

Thus, supporting the user during the tuple annotation task could be a future research direction. We need approaches that collect and visualize metadata evidence for each data error/correction candidate. This way, the user can annotate the tuples more effectively.

9.2.2 Handling User Annotation Errors

Our data cleaning systems expect one accurate user to annotate a few sampled tuples. However, in some data cleaning scenarios, we might prefer to instead have multiple non-accurate crowd annotators as they could be cheaper to hire rather than one expert user.

Thus, extending our data cleaning systems to handle noisy annotations from crowd workers could be another future research direction. We need approaches that aggregates these weak supervisions into a final set of accurate annotations for our data cleaning systems.

9.2.3 Extending Transfer Learning Methods

Our data cleaning systems can benefit from transfer learning in both error detection and correction tasks to estimate the effectiveness of the base error detection strategies and to pretrain the base error corrector models. In addition to these transfer learning methods, there are still other opportunities to leverage historical data.

Thus, extending the transfer learning methods to more challenging tasks could be another research direction. We can leverage the Wikipedia page revision history to train error generator models. These models can be used to impose real-world data errors to clean parts of datasets and generate more training data points, according to the data augmentation technique. Furthermore, we can design approaches to pretrain schema-dependent error corrector models, i.e., the vicinity- and domain-based models, on the Wikipedia page revision history.

9.2.4 Scaling the Systems

Our data cleaning systems achieve outstanding effectiveness, efficiency, and human involvement in typical settings where data size is in the order of thousands of tuples. However, when the data size goes beyond millions of tuples, we need new approaches to scale the systems.

Thus, leveraging big data processing systems, such as Apache Flink and Apache Spark, to make our data cleaning systems scalable could be another research direction. We can particularly leverage these big data processing systems to generate and process huge feature matrices with millions of data points and features.

References

[1] Ziawasch Abedjan, Hagen Anuth, Mahdi Esmailoghli, Mohammad Mahdavi, Felix Neutatz, and Binger Chen. "Data Science für alle: Grundlagen der Datenprogrammierung". In: *Informatik Spektrum* (2020), pp. 1–8.

[2] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. "Detecting data errors: Where are we and what needs to be done?" In: *PVLDB* 9.12 (2016), pp. 993–1004.

[3] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. "Profiling relational data: A survey". In: *VLDBJ* 24.4 (2015), pp. 557–581.

[4] Ziawasch Abedjan, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. "Dataxformer: A robust transformation discovery system". In: *ICDE*. 2016, pp. 1134–1145.

[5] Charu C Aggarwal and Chandan K Reddy. *Data clustering: Algorithms and applications*. CRC Press, 2013.

[6] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. "Learning string transformations from examples". In: *PVLDB* 2.1 (2009), pp. 514–525.

[7] Patricia C Arocena, Boris Glavic, Giansalvatore Mecca, Renée J Miller, Paolo Papotti, and Donatello Santoro. "Messing up with bart: Error generation for evaluating data-cleaning algorithms". In: *PVLDB* 9.2 (2015), pp. 36–47.

[8] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. "Dbpedia: A nucleus for a web of open data". In: *ISWC*. 2007, pp. 722–735.

[9] Laure Berti-Equille, Tamraparni Dasu, and Divesh Srivastava. "Discovery of complex glitch patterns: A novel approach to quantitative data cleaning". In: *ICDE*. 2011, pp. 733–744.

[10] Laure Berti-Equille, Hazar Harmouch, Felix Naumann, Noël Novelli, and Saravanan Thirumuruganathan. "Discovery of genuine functional dependencies from relational data with missing values". In: *PVLDB* 11.8 (2018), pp. 880–892.

[11] George Beskales, Ihab F Ilyas, Lukasz Golab, and Artur Galiullin. "On the relative trust between inconsistent data and inaccurate constraints". In: *ICDE*. 2013, pp. 541–552.

[12] Mathieu Blondel, Kazuhiro Seki, and Kuniaki Uehara. "Block coordinate descent algorithms for large-scale sparse multiclass classification". In: *Machine learning* 93.1 (2013), pp. 31–52.

[13] Daren C Brabham. *Crowdsourcing*. MIT Press, 2013.

[14] Andrei Z Broder. "On the resemblance and containment of documents". In: *Proceedings of the compression and complexity of sequences*. 1997, pp. 21–29.

[15] Ursin Brunner and Kurt Stockinger. "Entity matching with transformer architectures-a step forward in data integration". In: *EDBT*. 2020.

[16] Öykü Özlem Çakal, Mohammad Mahdavi, and Ziawasch Abedjan. "Clrl: Feature engineering for cross-language record linkage." In: *EDBT*. 2019, pp. 678–681.

[17] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. *Semi-supervised learning*. MIT Press, 2006.

[18] Olivier Chapelle, Jason Weston, and Bernhard Schölkopf. "Cluster kernels for semi-supervised learning". In: *NIPS*. 2003, pp. 601–608.

[19] Xu Chu, Ihab F Ilyas, and Paolo Papotti. "Discovering denial constraints". In: *PVLDB* 6.13 (2013), pp. 1498–1509.

[20] Xu Chu, Ihab F Ilyas, and Paolo Papotti. "Holistic data cleaning: Putting violations into context". In: *ICDE*. 2013, pp. 458–469.

[21] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. "Katara: A data cleaning system powered by knowledge bases and crowdsourcing". In: *SIGMOD*. 2015, pp. 1247–1261.

[22] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F Ilyas, Mourad Ouzzani, and Nan Tang. "Nadeef: A commodity data cleaning system". In: *SIGMOD*. 2013, pp. 541–552.

[23] Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, and Pradap Konda. *The magellan data repository*. `https://sites.google.com/site/anhaidgroup/projects/data`. 2015.

[24] Sanjib Das, Paul Suganthan GC, AnHai Doan, Jeffrey F Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. "Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services". In: *SIGMOD*. 2017, pp. 1431–1446.

[25] *Data science report*. `https://visit.figure-eight.com/rs/416-ZBE-142/images/CrowdFlower_DataScienceReport_2016.pdf`. 2016.

[26] Christopher De Sa, Alex Ratner, Christopher Ré, Jaeho Shin, Feiran Wang, Sen Wu, and Ce Zhang. "Deepdive: Declarative knowledge base construction". In: *ACM SIGMOD Record* 45.1 (2016), pp. 60–67.

[27] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. "The data civilizer system". In: *CIDR*. 2017.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805*. 2018.

[29] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: *SIGKDD*. 1996, pp. 226–231.

[30] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. "Discovering conditional functional dependencies". In: *TKDE* 23.5 (2010), pp. 683–698.

[31] Thomas A Feo and Mauricio GC Resende. "A probabilistic heuristic for a computationally difficult set covering problem". In: *Operations research letters* 8.2 (1989), pp. 67–71.

[32] Yoav Freund and Robert E Schapire. "A decision-theoretic generalization of on-line learning and an application to boosting". In: *JCSS* 55.1 (1997), pp. 119–139.

[33] Jerome H Friedman. "Greedy function approximation: A gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[34] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. "The llunatic data-cleaning framework". In: *PVLDB* 6.9 (2013), pp. 625–636.

[35] Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F Naughton, Narasimhan Rampalli, Jude Shavlik, and Xiaojin Zhu. "Corleone: Hands-off crowdsourcing for entity matching". In: *SIGMOD*. 2014, pp. 601–612.

[36] David Gotz and David Borland. "Data-driven healthcare: Challenges and opportunities for interactive visualization". In: *CG&A* 36.3 (2016), pp. 90–96.

[37] Yash Govind, Erik Paulson, Palaniappan Nagarajan, AnHai Doan, Youngchoon Park, Glenn M Fung, Devin Conathan, Marshall Carter, and Mingju Sun. "Cloudmatcher: A hands-off cloud/crowd service for entity matching". In: *PVLDB* 11.12 (2018), pp. 2042–2045.

[38] Sumit Gulwani. "Programming by examples: Applications, algorithms, and ambiguity resolution". In: *IJCAR*. 2016, pp. 9–14.

[39] Shuang Hao, Nan Tang, Guoliang Li, and Jian Li. "Cleaning relations using knowledge bases". In: *ICDE*. 2017, pp. 933–944.

[40] Jian He, Enzo Veltri, Donatello Santoro, Guoliang Li, Giansalvatore Mecca, Paolo Papotti, and Nan Tang. "Interactive and deterministic data cleaning". In: *SIGMOD*. 2016, pp. 893–907.

[41] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. "Holodetect: Few-shot learning for error detection". In: *SIGMOD*. 2019, pp. 829–846.

[42] Tony Hey, Stewart Tansley, Kristin Tolle, et al. *The fourth paradigm: Data-intensive scientific discovery*. Vol. 1. Microsoft Research Redmond, WA, 2009.

[43] Jean-Nicholas Hould. *Craft beers dataset*. `https://www.kaggle.com/nickhould/craft-cans`. Version 1. 2017.

[44] Alexander Benjamin Howard. *The art and science of data-driven journalism*. Columbia Journalism School, 2014.

[45] Zhipeng Huang and Yeye He. "Auto-detect: Data-driven error detection in tables". In: *SIGMOD*. 2018, pp. 1377–1392.

[46] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[47] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. "Wrangler: Interactive visual specification of data transformation scripts". In: *SIGCHI*. 2011, pp. 3363–3372.

[48] Richard M Karp. "Reducibility among combinatorial problems". In: *Complexity of computer computations*. 1972, pp. 85–103.

[49] Peter GW Keen. "Decision support systems: A research perspective". In: *Decision support systems: Issues and challenges: Proceedings of an international task force meeting*. 1980, pp. 23–44.

[50] Sanjay Krishnan, Michael J Franklin, Ken Goldberg, and Eugene Wu. "Boostclean: Automated error detection and repair for machine learning". In: *arXiv preprint arXiv:1711.01299* (2017).

[51] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. "Activeclean: Interactive data cleaning for statistical modeling". In: *PVLDB* 9.12 (2016), pp. 948–959.

[52] Sanjay Krishnan and Eugene Wu. "Alphaclean: Automatic generation of data cleaning pipelines". In: *arXiv preprint arXiv:1904.11827* (2019).

[53] Xian Li, Xin Luna Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. "Truth finding on the deep web: Is the problem solved?" In: *arXiv preprint arXiv:1503.00303* (2015).

[54] Mohammad Mahdavi and Ziawasch Abedjan. “Baran: Effective error correction via a unified context representation and transfer learning”. In: *PVLDB* 13.11 (2020), pp. 1948–1961.

[55] Mohammad Mahdavi and Ziawasch Abedjan. “Reds: Estimating the performance of error detection strategies based on dirtiness profiles”. In: *SSDBM*. 2019, pp. 193–196.

[56] Mohammad Mahdavi and Ziawasch Abedjan. “Semi-supervised data cleaning with raha and baran”. In: *CIDR*. 2021.

[57] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. “Raha: A configuration-free error detection system”. In: *SIGMOD*. 2019, pp. 865–882.

[58] Mohammad Mahdavi, Felix Neutatz, Larysa Visengeriyeva, and Ziawasch Abedjan. “Towards automated data cleaning workflows”. In: *LWDA*. 2019, pp. 10–19.

[59] Aurélien Max and Guillaume Wisniewski. “Mining Naturally-occurring Corrections and Paraphrases from Wikipedia’s Revision History.” In: *LREC*. 2010.

[60] Felix Neutatz, Mohammad Mahdavi, and Ziawasch Abedjan. “Ed2: A case for active learning in error detection”. In: *CIKM*. 2019, pp. 2249–2252.

[61] Mourad Ouzzani, Hossam Hammady, Zbys Fedorowicz, and Ahmed Elmagarmid. “Rayyan—a web and mobile app for systematic reviews”. In: *Systematic reviews* 5.1 (2016), p. 210.

[62] Sinno Jialin Pan and Qiang Yang. “A survey on transfer learning”. In: *TKDE* 22.10 (2009), pp. 1345–1359.

[63] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. “Data profiling with metanome”. In: *PVLDB* 8.12 (2015), pp. 1860–1863.

[64] Thorsten Papenbrock and Felix Naumann. “A hybrid approach to functional dependency discovery”. In: *SIGMOD*. 2016, pp. 821–833.

[65] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. “Scikit-learn: Machine learning in Python”. In: *JMLR* 12.Oct (2011), pp. 2825–2830.

[66] Tommi A Pirinen and Krister Lindén. “State-of-the-art in weighted finite-state spell-checking”. In: *CICLing*. 2014, pp. 519–532.

[67] Clement Pit–Claudel, Zelda Mariet, Rachael Harding, and Sam Madden. *Outlier detection in heterogeneous datasets using automatic tuple expansion*. Tech. rep. MIT-CSAIL-TR-2016-002. CSAIL, MIT, 2016.

[68] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J Miller, and Divesh Srivastava. “Combining quantitative and logical data cleaning”. In: *PVLDB* 9.4 (2015), pp. 300–311.

[69] Foster Provost and Tom Fawcett. “Data science and its relationship to big data and data-driven decision making”. In: *Big data* 1.1 (2013), pp. 51–59.

[70] Dorian Pyle. *Data preparation for data mining*. Morgan Kaufmann, 1999.

[71] Erhard Rahm and Philip A Bernstein. “A survey of approaches to automatic schema matching”. In: *VLDBJ* 10.4 (2001), pp. 334–350.

[72] Erhard Rahm and Hong Hai Do. “Data cleaning: Problems and current approaches”. In: *DE* 23.4 (2000), pp. 3–13.

[73] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason A. Fries, Sen Wu, and Christopher Ré. “Snorkel: Rapid training data creation with weak supervision”. In: *VLDBJ* 29.2-3 (2020), pp. 709–730.

[74] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. "Holoclean: Holistic data repairs with probabilistic inference". In: *PVLDB* 10.11 (2017), pp. 1190–1201.

[75] Claude Sammut and Geoffrey I Webb. *Encyclopedia of machine learning*. Springer Science & Business Media, 2011.

[76] Robert E Schapire. "Explaining adaboost". In: *Empirical inference*. 2013, pp. 37–52.

[77] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*. Cambridge University Press, 2008.

[78] Burr Settles. *Active learning literature survey*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 2009.

[79] Wei Shen, Jianyong Wang, and Jiawei Han. "Entity linking with a knowledge base: Issues, techniques, and solutions". In: *TKDE* 27.2 (2014), pp. 443–460.

[80] Rishabh Singh. "Blinkfill: Semi-supervised programming by example for syntactic string transformations". In: *PVLDB* 9.10 (2016), pp. 816–827.

[81] Rishabh Singh and Sumit Gulwani. "Transforming spreadsheet data types using examples". In: *ACM SIGPLAN Notices*. 2016, pp. 343–356.

[82] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. "Revisiting unreasonable effectiveness of data in deep learning era". In: *ICCV*. 2017, pp. 843–852.

[83] Graham Upton and Ian Cook. *A dictionary of statistics 3e*. Oxford University Press, 2014.

[84] Jesper E Van Engelen and Holger H Hoos. "A survey on semi-supervised learning". In: *Machine Learning* 109.2 (2020), pp. 373–440.

[85] Larysa Visengeriyeva and Ziawasch Abedjan. "Anatomy of metadata for data curation". In: *JDIQ* 12.3 (2020).

[86] Larysa Visengeriyeva and Ziawasch Abedjan. "Metadata-driven error detection". In: *SSDBM*. 2018, pp. 1–12.

[87] Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J Miller. "Continuous data cleaning". In: *ICDE*. 2014, pp. 244–255.

[88] Guanjin Wang, Jie Lu, Kup-Sze Choi, and Guangquan Zhang. "A transfer-based additive ls-svm classifier for handling missing data". In: *IEEE transactions on cybernetics* 50.2 (2018), pp. 739–752.

[89] Pei Wang and Yeye He. "Uni-detect: A unified approach to automated error detection in tables". In: *SIGMOD*. 2019, pp. 811–828.

[90] Ken Whistler and Laurentiu Iancu. *Unicode character database*. `http://www.unicode.org/reports/tr44/`. Version Unicode 12.0.0. Accessed: 12.09.2019. 2019.

[91] *Wikipedia:Database download*. `https://en.wikipedia.org/wiki/Wikipedia:Database_download`. Accessed: 12.09.2019. 2019.

[92] *Wikitext*. `https://www.mediawiki.org/wiki/Wikitext`. Accessed: 12.09.2019. 2019.

[93] Mohamed Yakout, Laure Berti-Équille, and Ahmed K Elmagarmid. "Don't be scared: Use scalable automatic repairing with maximal likelihood and bounded changes". In: *SIGMOD*. 2013, pp. 553–564.

[94] Mohamed Yakout, Ahmed K Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F Ilyas. "Guided data repair". In: *PVLDB* 4.5 (2011), pp. 279–289.

[95] Torsten Zesch. "Measuring contextual fitness using error contexts extracted from the wikipedia revision history". In: *EACL*. 2012, pp. 529–538.

[96] Chen Zhao and Yeye He. "Auto-em: End-to-end fuzzy entity-matching using pre-trained deep models and transfer learning". In: *WWW*. 2019, pp. 2413–2424.

[97] Kaile Zhou, Chao Fu, and Shanlin Yang. "Big data driven smart energy management: From big data to big insights". In: *Renewable and Sustainable Energy Reviews* 56 (2016), pp. 215–225.