Let’s Make it Dirty with BART!

Donatello Santoro¹, Patricia C. Arocena², Boris Glavic³, Giansalvatore Mecca¹, Renée J. Miller², and Paolo Papotti⁴

¹Università della Basilicata - Italy, ²University of Toronto - Canada ³Illinois Inst. of Technology - USA, ⁴Eurecom - France

(Discussion Paper)

Abstract. In the last few years many automatic or semi-automatic data-repairing algorithms have been proposed in order to improve the quality of a given database. Due to the richness of research proposals, it is important to conduct experimental evaluations to assess each tool’s potential. Bart is an open-source error-generation system conceived to support thorough experimental evaluations of these data-repairing systems. In this paper we discuss how generating errors in data is a complex problem, with several facets. We introduce the important notions of detectability and repairability of an error, that stand at the core of Bart. Then, we show how, by changing the features of errors, it is possible to influence quite significantly the performance of the tools. Finally, we concretely put to work five data-repairing algorithms on dirty data of various kinds generated using Bart, and discuss their performance.

1 Introduction

Data quality is a very important concern in data management. To date, many (disparate) automatic and semi-automatic data-cleaning algorithms have been proposed in the database community [13][12][8][5][6][4]. These algorithms come from different inspirations. Most of them are constraint-based: they assume that the target database comes with a set of data-quality rules – for example, functional dependencies (FDs) or conditional functional dependencies (CFDs) – and data is repaired to remove violations to these constraints. Others, on the contrary, do not rely on constraints, but rather on statistics-based approaches to identify suspect values or outliers and try to repair them.

Due to the richness of research proposals, it is important to conduct thorough and fair experimental evaluations to assess each tool’s potential. In fact, other fields like entity resolution, record linkage, schema mapping and data exchange have worked to develop consolidated tools and benchmarks for empirically evaluating algorithms [14][1][3]. Thorough evaluation of data-cleaning systems requires systematic control over the amount of errors in a dataset, and over how hard these errors are to repair. Dirty datasets must be paired with a ground-truth clean dataset to enable evaluation of the quality of a repair produced by a cleaning algorithm. To support rigorous empirical evaluations, an
error-generation system must be able to generate multiple dirty versions of a dataset with low user effort, and scale to large datasets.

**Bart.** **BART**\(^1\)\([2][17]\) is the first error-generation tool explicitly conceived to support empirical evaluations of data-repairing algorithms as per the requirements outlined above. It takes as input a clean database and a set of data-quality rules, and injects errors into the database. Rules are expressed using the powerful language of *denial constraints* \([15]\) and errors can be of several kinds, such as typos, duplicated values, nulls, and outliers. We show the major components of the system in Figure 1. A user interacts with **BART** by creating error-generation tasks, using a GUI or CL interface. These tasks are then interpreted by **BART**’s error-generation engine to create dirty versions of a clean database.

The system provides the highest possible level of control over the error-generation process. Among other parameters, it allows users to choose the percentage of errors, whether they want a guarantee that errors are detectable using the given constraints, and even provides an estimate of how hard it will be to restore the database to its original clean state. **BART** is open-source: its codebase is available on GitHub\(^2\) and can be further extended by the community to develop new features and functionalities.

**Evaluation Overview.** The empirical evaluation convey three primary insights about the system.

(i) First, we will discuss how the fine-level control over the features of errors distinguishes **BART** from previous error-generating techniques used in evaluating data-repairing systems.

(ii) Then, we will discuss how the characteristics of errors may significantly influence the quality of repairs generated by a system.

(iii) Finally, we will demonstrate five different algorithms in action on dirty data generated using **BART**, to reveal new insights on their (relative) performance as the characteristics of the errors are varied by **BART**.

Overall, the attendees will learn how the availability of a tool like **BART** may help to level the field and raise the bar for evaluation standards in data cleaning.

The paper is organized as follows. Section 2 introduces the main motivation for the system, and the notions of detectability and repairability of errors. Section 3 provides an overview of the system and of its main use cases. Finally,

---

\(^1\) **BART**: Benchmarking Algorithms for data Repairing and Translation

\(^2\) [https://github.com/dbunibas/BART](https://github.com/dbunibas/BART)
Section 4 discusses the empirical evaluation, and the main lessons that can be learned from it.

2 Concept and Motivation

Assume we are given a database about soccer players, shown in Figure 2, and we want to assess the performance of repair algorithms according to a few data-quality rules.

(i) A first FD stating that Name and Season are a key for the table: $d_1 : \text{Name, Season} \rightarrow \text{Team, Stadium, Goals}$.

(ii) And, a second FD stating that Team implies Stadium: $d_2 : \text{Team} \rightarrow \text{Stadium}$.

We specify these rules in BART using the language of denial constraints. Denial constraints are a very expressive language, capable of capturing most data-quality rules used for data-repairing, including FDs, CFDs, cleaning equality-generating dependencies, editing rules, fixing rules, and ordering constraints [13]. For the sake of simplicity, here we omit the technical details about the syntax and semantics of denial constraints, and show example data-quality rules in the more familiar syntax of FDs.

To evaluate data-repair systems, we proceed as follows.

(i) We start with a clean instance $I$, like the one in Figure 2, and the set of constraints $\Sigma = \{d_1, d_2\}$ discussed above.

(ii) We inject errors by applying a set of cell changes; each cell change $\text{ch} = \langle t_i, A := v \rangle$ updates the value of attribute $A$ in tuple $t_i$ to a new value $v$, e.g., $\langle t_1, \text{Season} := 2011-12 \rangle$. By applying a set of cell changes $\text{Ch}$ to $I$, we obtain a new instance $I_d = \text{Ch}(I)$, named the dirty instance.

(iii) We run a data-repairing algorithm over $I_d$ to obtain a repaired instance $I_{rep}$. We measure the quality of the algorithm using precision and recall. Intuitively, we count how many changes in $\text{Ch}$ have been restored to their original values in $I_{rep}$. Further details are in the full paper [2].

We want now to emphasize how different ways to change the cells of the clean instance may lead to errors that show completely different features when considered from the perspective of a data-repairing tool.

<table>
<thead>
<tr>
<th>Player Name</th>
<th>Season</th>
<th>Team</th>
<th>Stadium</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$ : Giovinco</td>
<td>2013-14</td>
<td>Juventus</td>
<td>Juv.Stadium</td>
<td>3</td>
</tr>
<tr>
<td>$t_2$ : Giovinco</td>
<td>2014-15</td>
<td>Toronto</td>
<td>BMO Field</td>
<td>23</td>
</tr>
<tr>
<td>$t_4$ : Pirlo</td>
<td>2015-16</td>
<td>N.Y.City</td>
<td>Yankee St.</td>
<td>0</td>
</tr>
<tr>
<td>$t_5$ : Vidal</td>
<td>2014-15</td>
<td>Juventus</td>
<td>Juv.Stadium</td>
<td>8</td>
</tr>
<tr>
<td>$t_6$ : Vidal</td>
<td>2015-16</td>
<td>Bayern</td>
<td>Allianz Ar.</td>
<td>3</td>
</tr>
</tbody>
</table>

Fig. 2. Example Clean Database
2.1 Detectability

When evaluating a constraint-based repair algorithm, we want to make sure the errors we inject are detectable by the system in question. After all, an error that cannot be detected, cannot be repaired. To reason about detectability, we need a notion for determining whether a cell change is involved in a constraint violation. Consider the following cell change: $ch_1 = \langle t_1.\text{Season} := 2012-13 \rangle$ that updates tuple $t_1$ as follows:

<table>
<thead>
<tr>
<th>Player Name</th>
<th>Season</th>
<th>Team</th>
<th>Stadium</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giovinco</td>
<td>2012-13</td>
<td>Juventus</td>
<td>Juv.Stadium</td>
<td>3</td>
</tr>
</tbody>
</table>

This change does not introduce a violation to any of the constraints in $\Sigma$, i.e., after the cell change the modified database instances does fulfill all the constraints. Therefore, any data-repairing tool that relies on the constraints to detect dirtiness in the database will not be able to detect the change. We call this an undetectable change.

More formally, a cell change $ch = \langle t_i.A := v \rangle$ in $Ch$ introduces a detectable error in $I$ for constraint $dc$ if: (i) cell $t_i.A$ is involved in a violation with $dc$ in instance $I_d$, and (ii) cell $t_i.A$ was not involved in a violation with $dc$ in instance $I$. Here “involved in a violation” is defined based on the fact that a constraint can be associated with a query that returns sets of cells that cause a violation. We call this type of queries violation-detection queries. A cell is involved in a violation with a constraint $dc$ if it is in the result of the violation-detection query for $dc$ [2]. For example, the violation-detection query for $d_2$ is $Q_{d_2}(i, i', t, a, a') = \text{Player}(i, n, s, t, a, g), \text{Player}(i', n', s', t, a', g'), a \neq a', i \neq i'$. This query returns the Team and Stadium attributes of pairs of Player tuples with the same team, but different stadiums.\(^3\) Using the tuple $id$’s, sets of cells involved in violations can be determined based on the result of this query.

BART allows users to control whether changes to the clean database are detectable when using the constraints in $\Sigma$. BART may be configured to generate random changes, that do not need to be detectable, or errors that are guaranteed to be detectable using the constraints. Note this requires BART to reason efficiently and holistically about a set of changes to ensure that they are detectable using a given set of constraints.

Interestingly, this latter requirement significantly increases the complexity of the error-generation process. In fact, generating a given number of errors in a clean database that are detectable using a set of constraints $\Sigma$ is an NP-complete problem [2]. To deal with this complexity BART implements several novel optimizations [2] that balance the need for control over the nature of errors and scalability.

\(^3\) We assume every tuple has a unique identifier that per convention is the first attribute. Queries are expressed in a notation similar to Datalog.
2.2 Repairability

An alternative change that indeed introduces a detectable error is the following:
\[ \text{ch}_2 = \langle t_1.\text{Season} := 2014-15 \rangle. \] After this update, tuples \( t_1 \) and \( t_2 \) violate FD \( d_1 \), which states that \( \text{Name} \) and \( \text{Season} \) are a key for the table:

<table>
<thead>
<tr>
<th>Player Name</th>
<th>Season</th>
<th>Team</th>
<th>Stadium</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 ) : Giovinco</td>
<td>\textbf{2014-15}</td>
<td>Juventus</td>
<td>Juv.Stadium</td>
<td>3</td>
</tr>
<tr>
<td>( t_2 ) : Giovinco</td>
<td>2014-15</td>
<td>Toronto</td>
<td>BMO Field</td>
<td>23</td>
</tr>
</tbody>
</table>

This change is easily detected using the constraints. Still, it is quite difficult for an automatic data-repairing algorithm to restore the database to its clean state. Notice, in fact, that after this change, the original value 2013-14 has been removed from the active domain of the dirty database. A correct repair cannot be found by any repair algorithm that uses the values in the database as the candidates for repair. BART uses the notion of repairability of an error to characterize this aspect. In the case above, it would assign repairability 0 to change \( \text{ch}_2 \). Different detectable changes may have quite different repairability values. As an example, consider now change \( \text{ch}_3 = \langle t_1.\text{Stadium} := \text{Delle Alpi} \rangle. \) The change is detectable using FD \( d_2 \). In addition, the redundancy in the dirty database may be used to repair the database:

<table>
<thead>
<tr>
<th>Player Name</th>
<th>Season</th>
<th>Team</th>
<th>Stadium</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 ) : Giovinco</td>
<td>2013-14</td>
<td>Juventus</td>
<td>\textbf{Delle Alpi}</td>
<td>3</td>
</tr>
<tr>
<td>( t_3 ) : Pirlo</td>
<td>2014-15</td>
<td>Juventus</td>
<td>Juv.Stadium</td>
<td>5</td>
</tr>
<tr>
<td>( t_5 ) : Vidal</td>
<td>2014-15</td>
<td>Juventus</td>
<td>Juv.Stadium</td>
<td>8</td>
</tr>
</tbody>
</table>

The new, dirty tuple \( t_1 \) is involved in two violations to \( d_2 \), one with \( t_3 \), another with \( t_5 \). In both cases, the change is in violation with \( \text{Juventus.Stadium} \). By a straightforward probabilistic argument, BART would calculate a 2/3 repairability for this error, and rank it as a medium-repairability error.

Errors may have higher repairability, even 1 in some cases. Consider, for example, an additional rule \( d_3: \text{Team}[\text{Juventus}], \text{Season}[2013-14] \rightarrow \text{Stadium}[\text{Juventus.Stadium}] \). This CFD rule states unequivocally that Juventus has played its home games for season 2013–14 in the Juventus Stadium. Since this knowledge is part of the constraint, the dirty cell can easily be restored to its original, clean state.

2.3 Other Kinds of Errors

To conclude this discussion about the features of errors, we notice that the notions of detectability and repairability, that are centered around detecting violations to constraints, are not the only ones supported by BART.

Consider, for example, change \( \text{ch}_4 = \langle t_1.\text{Goals} := 123 \rangle. \) This change is not detectable using the constraints. However, it might be detected by a statistics-based data-repairing algorithm, because it introduces an outlier into the distribution of values of the \( \text{Goals} \) attribute. BART can be configured in order to generate changes of this kind as well.
3 Overview of the System

BART provides users with the graphical user interface shown in Figure 3 to handle error-generation tasks. An error-generation task, $E$, is composed of four key elements: (i) a database schema $S$; (ii) a set $\Sigma$ of denial constraints (DCs) encoding data quality rules over $S$; (iii) an instance $I$ of $S$ that is clean with respect to $\Sigma$; (iv) a set of configuration parameters $\text{Conf}$ (shown in Figure 3.1) to control the error-generation process. These parameters specify, among other things, which relations can be changed, how many errors should be introduced, and how many of these errors should be detectable. They also let the user control the degree of repairability of the errors.

Based on this, BART supports several uses cases. The main one consists of generating a desired degree of detectable errors for each constraint. In addition, users may also specify a range of repairability values for each constraint; BART will estimate the repairability of changes, and only generate errors with estimated repairability within that range. In addition to detectable errors, BART may also generate random errors of several kinds: typos (e.g., ‘database’), duplicated values, bogus or null values (e.g., ‘999’, ‘***’). Random errors may be freely mixed with constraint-induced ones. Finally, BART can introduce outliers in numerical attributes. BART provides sophisticated features to analyze the characteristics of errors that are introduced in the data. It generates charts to analyze the number of errors detected by each constraint, and their estimated repairability (shown in Figure 3.3). It also offers a versioning system, that allows users to generate different dirty databases for the given scenario, and compare the characteristics of their errors.

Finally, BART offers a flexible set of metrics to measure the quality of the repairs generated by a data-repairing tool. In fact, different algorithms can repair data in different ways. For example, some algorithms can produce repairs that mark dirty cells using a variable, while others always restore the dirty instance.
with a constant value. Different metrics have been proposed and are implemented in BART to uniformly evaluate these heterogeneous changes in the data [4, 11].

4 Empirical Evaluation

We conduct [17] an empirical evaluation of several data-repairing algorithms over dirty data generated using BART.

We used two publicly available tools, Llunatic [11] and Nadeef [8], to run four rule-based data-repairing algorithms: (i) Greedy [5, 7]; (ii) Holistic [6]; (iii) Llunatic [10]; and (iv) Sampling [4]. In addition, we evaluated (v) SCARE [18], a statistics-based tool.

The tools were tested with several repair tasks, based on synthetic and real datasets, some of them constraint-based and some statistics-based. We briefly list them here: (i) Employees is a synthetic scenario in the full paper [2]; (ii) Customers is a synthetic scenario from Geerts et al. [10]; (iii) Tax is a synthetic scenario from Fan et al. [9]; (iv) Bus is a real-world scenario from Dallachiesa et al. [8]; and (v) Hospital is a real-world scenario used in several data-repairing papers (e.g., [8, 10]).

Datasets and constraints have been chosen to exhibit different characteristics. Some have high redundancy in their data. Others contain numerical attributes, and constraints containing ordering (<, >) comparisons. Some datasets have master-data [16] and CFDs, while others have only FDs. All these differences help to validate our techniques and the tools under exam.

Notice the purpose of the evaluation was not to assess the quality of the repair algorithms, rather to show how BART can be used to uncover new insights into the data-repairing process. Some important insights are the following.

Lesson 1: Data-repairing is not yet Mature. We expect that a wide degree of variability in quality among all algorithms will emerge from our evaluations. This variability does not clearly emerge from evaluations reported in the literature, suggesting there is no definitive data-repairing algorithm yet.

Lesson 2: Repairability Matters. We observe different trends with respect to repairability. Typically, repair algorithms return very good repairs when sufficient information is available (i.e., high repairability); however, their quality tends to degrade quickly as repairability decreases.

A key observation is that repairability has a strong correlation with the quality of the repairs. In this respect, we believe it nicely captures the “hardness” of the data-repairing problem and it helps in getting a concrete intuition of the power and the limits of existing solutions.

Lesson 3: We Need to Document Our Dirty Data. We may conclude that tools exhibit quite different performance on data-repairing problems of different nature, and the repairability is a natural proxy to characterize how “difficult” a data-repairing problem is.

In light of this and to level the field, we believe it is crucial to have at our disposal systematic error-generation tools and to properly document the
characteristics of the dirty data used in empirical evaluations of data-repairing solutions. BART is a concrete step in this direction.

**Lesson 4: Generating Errors is Hard.** The problem of systematically generating errors, however, is not an easy one. We will show how different configurations of the error-generation task affect the overall scalability of the system, and discuss the main optimizations that BART relies on in order to tame the complexity of the process.

**References**