# Repairing Rules in Black-Box Models Based on User Feedback

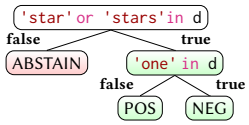

## ABSTRACT

By combining interpretable rules with complex model (that may be based on machine learning or any other complex algorithms), rule-based semi-supervised systems have been shown to be effective for tasks where insufficient ground-truth information is available. Examples include generating labels in semi-supervised labeling (like Snorkel) that uses a set of labeling functions (LFs) as rules, and any data cleaning systems that uses a set of denial constraints (DCs) as rules. When such a system produces a wrong result, it is hard for an analyst to debug and fix the outcome, because of the complex interaction between the rules and model. In this work, we treat the complex model as a *black box* and present a framework called RuleCleaner that suggests repairs for the rules to improve the accuracy of the model based on a partial set of ground truth provided by user. We demonstrate experimentally that RuleCleaner can efficiently generate meaningful fixes for a set of rules (LFs or DCs) based on a small number of ground truth labels, which is particularly useful for cleaning input rules that are automatically mined from a dataset resulting in a large number of spurious or erroneous LFs or DCs.

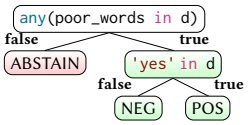

## 1 INTRODUCTION

Semi-supervised systems that use *interpretable rules* whose output is combined by a black box model are used for training data generation, data curation & cleaning, and many other use cases. We refer to such systems as *Rule-based Black-Box Models* (*RBBMs*). Consider a *data cleaning* system that repairs a noisy dataset to comply with a set of *integrity constraints*, e.g., denial constraints (DCs). Such a system can be modeled as a RBBM where the constraints constitute the rules in the data cleaning system, the output of the rules is a set of violations, and the final output determines which tuples to delete or to fix. Systems like *HoloClean* [43] use a black-box machine learning (ML) model over the rule outputs to predict repairs while others, e.g., *MuSe* [21], do not use ML. A very different example of RBBMs is semi-supervised labeling as in *Snorkel* [41] where human-generated rules, called *labeling functions* (LFs), are combined by a black box model to generate training data labels. The RBBM approach is attractive, because (a) it uses *interpretable* rules stemming from domain knowledge, which are simple enough to be created by domain experts and can be understood by humans even if they are automatically generated, and (b) it formalizes the integration of rules with a complex algorithm (whether ML-based or not). These properties make RBBMs more interpretable and configurable than only using a black box model or algorithm. Moreover, it has been observed that RBBMs can achieve high accuracy [42, 43].

Since rules are a critical component of RBBMs, the accuracy of a RBBM depends significantly on the accuracy of the input rules. If the rules are correct and exhaustive, a RBBM is likely to produce more accurate answers (correct repair or labels). On the other hand, inaccurate rules and rules with low coverage (that are only applicable to a few data points) lead to poor performance. Manually generating a set of highly accurate rules for a RBBM is a highly non-trivial and time-consuming task. While there have been attempts to automate this process, e.g., through unsupervised or semi-supervised generation of LFs [8, 15, 47] or automated discovery of DCs and FDs for data cleaning [38, 39], the quality of the rules produced by such systems is far from perfect. If these automatically or manually generated rules are not refined, they are likely to degrade the accuracy of the RBBM. Debugging rules by tracing incorrect outputs of a RBBM, however, is not easy since the RBBM may combine the outputs of the rules in arbitrarily complex and opaque ways using machine learning or other complex algorithms.




(a) Refined rule #1




(b) Refined rule #3

```
def key_word_star(d): #LF-1
  words = ['star', 'stars']
  return POSITIVE if words.intersection(d) else ABSTAIN

 def key_word_waste(d): #LF-2
   return NEGATIVE if ('waste' in d) else ABSTAIN

def key_word_poor(d): #LF-3
  words = ['poorly', 'useless', 'horrible', 'money']
  return NEGATIVE if words.intersection(d) else ABSTAIN
```

(c) Some of the amazon reviews labeling functions

| id | text | true label | pred label | new label | LF labels 1 | 2 | 3 |
|---|---|---|---|---|---|---|---|
| 0 | five stars. product works fine | P | P | P | P | - | - |
| 1 | one star. rather poorly written needs more content and an editor | N | P | N | ~~P~~N | - | N |
| 2 | five stars. awesome for the price lightweight and sturdy | P | P | P | P | - | - |
| 3 | one star. not my subject of interest, too dark | N | P | N | ~~P~~N | - | - |
| 4 | yes, get it! the best money on a pool that we have ever spent. really cute and holds up well with kids constantly playing in it | P | N | P | - | - | ~~N~~P |

**Figure 1: Amazon products reviews with ground truth labels ("P"ositive or "N" egative), predicted labels by Snorkel [41] (before and after rule refinement), and the results of the labeling functions from Fig. 1c ("-" means ABSTAIN). Updated results for the repaired rules are highlighted in blue).**

In this paper, we present the novel RuleCleaner framework that uses a human-in-the-loop approach to address the above challenges and repair rules for RBBMs. Given a black-box system modeled as a RBBM, a set of rules $\mathcal{R}$, and ground truth labels for a subset of the input to RBBM, RuleCleaner generates a set of *repairs* to fix $\mathcal{R}$ to improve the accuracy of the RBBM on the labeled subset of the data. Indeed, there may be other factors such as noisy data or errors in the model/algorithm used in the RBBM that contribute to an erroneous output with a goal to repair erroneous rules. However, we treat the RBBM as a black-box and focus only on errors caused by imprecise or incorrect rules. We illustrate the use of RuleCleaner for semi-supervised labeling (LFs as rules) and data cleaning (DCs as rules) in the examples shown below.

Example 1.1. **(Semi-supervised labeling with Snorkel [42])** *Consider the Amazon Review Dataset from [15, 23] which contains*

*reviews for products bought from Amazon and the task of labeling the reviews as* POS *(positive) or* NEG *(negative). A subset of labeling functions (*LFs*) generated by the recent Witan system [15] for this task are shown in Fig. 1c. For instance,* `key_word_star` *labels reviews as* POS *that contain either* "star" *or* "stars" *and otherwise returns* ABSTAIN *(the function cannot make a prediction). Some reviews with their ground truth labels (unknown to the user) and the labels predicted by* Snorkel *[42] are shown in Fig. 1. Fig. 1 also shows the results of three LFs including the ones from Fig. 1c. Reviews 1,3, and 4 are mispredicted by the model trained by Snorkel over the LF outputs. Our goal is to reduce such misclassifications by refining the rules. We treat Snorkel as a blackbox that can use an arbitrarily complex algorithm or ML classifier to generate a final label for each data point.*

*Suppose a human annotator labels the subset of the reviews shown in Fig. 1 to* confirm *correct labels produced by Snorkel and/or* complain *about mispredictions. Reviews 1 and 3 were labeled as* POS *even though they are obviously negative.* RuleCleaner *uses these ground truth labels to generate a repair $\Phi$ for the rules $\mathcal{R}$ by deleting or refining rules to ensure they align with the ground truth. Fig. 1 also shows the labels for the repaired rules $\Phi(\mathcal{R})$ (updated labels are highlighted in blue), and the updated predictions generated by rerunning Snorkel on $\Phi(\mathcal{R})$.* RuleCleaner *repairs LF-1 and LF-3 from Fig. 1c by adding new predicates. Fig. 1a and 1b show the rules in tree form (discussed in Sec. 2) with new predicates highlighted in green. The updated version of rule LF-1 fixes the labels assigned to review 1 and 3 from P(ositive) to N(egative). Intuitively, this repair is sensible: a review mentioning* "one" *and* "star(s)" *is very likely negative. The prediction for review 4 is fixed by checking in LF-3 for* `'yes'` *which flips the prediction.*

Example 1.2. ***(Data cleaning with MuSe [21] and DCs)*** *Consider a simplified version of the US tax dataset from [9] shown in Fig. 2a — each record describes an individual. Fig. 2b shows a sample of the DCs generated by DCFINDER [38], which may be erroneous since the input data is dirty. There are some erroneous cells in this dataset (highlighted in red). We use* MuSe *[21] to compute a minimal set of tuples to be deleted such that the repaired database fulfills the constraints. Consider, for example, the DC $dc_5$ (highlighted in Fig. 2b), a functional dependency that checks that any two persons in the same state have to have the same child exemption value:*

$$dc_5 : \texttt{state} \rightarrow \texttt{childexem}$$

*This DC is not correct as the child exemption also depends on whether the person has children (attribute* `haschild`*).*

*MuSe decides to delete the tuples with* `tid` *$x_1$, $x_4$, and $x_5$ (highlighted in blue in Fig. 2a). Manually debugging why some tuples are correctly deleted ($x_1, x_4$) and why some tuples are incorrectly deleted ($x_5$) is hard, especially when the dataset is big and/or the number of denial constraints is large. Given the user feedback that $x_1, x_4$ were correctly deleted and $x_5$ was incorrectly deleted,* RuleCleaner *generates a refinement for $dc_5$ (highlighted in orange) as shown in Fig. 2c (new predicates highlighted in green). The refined DC corresponds to the FD*

$$dc_5' : \texttt{state}, \texttt{haschild} \rightarrow \texttt{childexem}$$

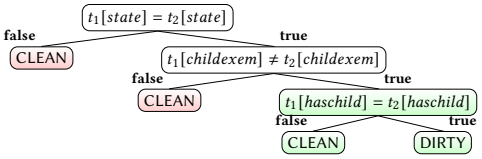
*After rerunning Muse with $dc_5'$, it successfully reverted the incorrect deletion of tuple $x_5$ while keeping the correct deletions of $x_1$ and $x_2$.*

| tid | zip | state | city | haschild | childexem | areacode | phone | ... |
|---|---|---|---|---|---|---|---|---|
| $x_1$ | 29447 | IL | GROVER | Y | 3300 | 864 | 1000815 | ... |
| $x_2$ | 63099 | IL | FENTON | Y | 1200 | 417 | 1001143 | ... |
| $x_3$ | 47032 | CA | MOORES HILL | Y | 1000 | 260 | 1000014 | ... |
| $x_4$ | 37056 | CA | DICKSON | Y | 0 | 901 | 1001648 | ... |
| $x_5$ | 35580 | AL | PARRISH | Y | 300 | 251 | 1004712 | ... |
| $x_6$ | 35802 | AL | HUNTSVILLE | N | 0 | 334 | 1001867 | ... |
| $x_7$ | 83823 | ID | DEARY | N | 0 | 208 | 1015287 | ... |
| $x_8$ | 83467 | ID | SALMON | Y | 3300 | 208 | 1001990 | ... |
| $x_9$ | 2873 | RI | ROCKVILLE | Y | 0 | 401 | 1015810 | ... |
| $x_{10}$ | 2873 | RI | ROCKVILLE | Y | 0 | 401 | 1006274 | ... |

**(a) Tax dataset (simplified)**

| Num. | rule |
|---|---|
| $dc_1$ | $\neg$ ($t_1$.areacode=$t_2$.areacode $\wedge$ $t_1$.phone = $t_2$.phone) |
| $dc_2$ | $\neg$ ($t_1$.zip=$t_2$.zip $\wedge$ $t_1$.city $\neq$ $t_2$.city) |
| $dc_3$ | $\neg$ ($t_1$.areacode=$t_2$.areacode $\wedge$ $t_1$.state $\neq$ $t_2$.state) |
| $dc_4$ | $\neg$ ($t_1$.zip=$t_2$.zip $\wedge$ $t_1$.state $\neq$ $t_2$.state) |
| $dc_5$ | $\neg$ ($t_1$.state=$t_2$.state $\wedge$ $t_1$.childexem $\neq$ $t_2$.childexem) |

**(b) Some of the DCs generated by DCFINDER [38] for the tax dataset**




**(c) Refined version of $dc_5$ from Fig. 2b produced by** RuleCleaner

**Figure 2: Data cleaning on the Tax dataset [9]**

## Our Contributions

We make the following contributions in this work:

- *RBBM*: We introduce *rule-based black models* (*RBBM*) as a framework for describing systems that combine the output of a set of interpretable rules to predict labels for a set of data points $\mathcal{X}$. This framework is quite general as we demonstrate by modeling semi-supervised labeling and DC-based data cleaning as RBBMs.
- *Rule repair*: We study the problem of repairing the rules $\mathcal{R}$ of a RBBM based on user-provided ground truth labels while minimizing the changes to $\mathcal{R}$ and show that the problem is NP-hard.
- *Efficient rule repair*: We develop a `PTIME` heuristic algorithm for rule repairs that refines or deletes rules. A core component of this algorithm is a technique for refining rules by introducing new predicates such that the rule returns ground truth labels.
- *Predicate selection algorithms*: We introduce (i) a brute-force algorithm that minimizes the number of predicates required to assign the expected labels; (ii) a greedy algorithm which may return non-minimal repairs; and (iii) an algorithm based on information-theoretic metrics which greedily selects predicates that best separate assignments with different expected labels.
- *Experimental results*: We evaluate our approach for two instances of RBBM: semi-supervised labeling in *Snorkel* [41] and constraint-based data cleaning in *MuSe* [21]. We demonstrate that our algorithm is effective in repairing rules such that the RBBM returns the ground truth label for most inputs. The repairs we generate typically generalize well to data points for which the ground truth label is not provided by the user. Furthermore, we apply our framework to rules generated automatically using Witan [15] (labeling function generation) and DCFINDER [38] (denial constraint discovery). Our approach can significantly improve the quality of such automated rules: the repairs shown in Ex. 1.1 and 1.2 were produced by our system.

## 2 THE RULECLEANER FRAMEWORK

We now formalize *Rule-based Black-Box Models* (*RBBMs*), define the rule repair problem, and study the complexity of this problem. Our system RuleCleaner proposes fixes to the rules of a RBBM that improve its accuracy on a user-provided set of labeled examples. The model of rule-based black-box we use can capture various data repair systems [1, 5, 14, 18, 20, 32, 43] and semi-supervised labeling [15, 35, 41], and any other system that uses a set of rules to solve a problem that can be modeled as a prediction task.

### 2.1 Black-Box Model, Data Points, and Labels

Consider a set of input *data points* $\mathcal{X}$ and a set of discrete *labels* $\mathcal{Y}$. A RBBM takes $\mathcal{X}$, the labels $\mathcal{Y}$, and a set of rules $\mathcal{R}$ (discussed in Sec. 2.2) as input and produces a *model* $\mathcal{M}_\mathcal{R}$ (Def. 2.4) as the output that maps each data point in $\mathcal{X}$ to a label in $\mathcal{Y}$, i.e.,

$$\mathcal{M}_\mathcal{R} : \mathcal{X} \rightarrow \mathcal{Y}$$

Without loss of generality, we assume the presence of an abstain label $y_0 \in \mathcal{Y}$ that is used by the RBBM or a rule to abstain from providing a label to some input data points. For a data point $x \in \mathcal{X}$, $y_x = \mathcal{M}_\mathcal{R}(x)$ denotes the label given by the RBBM model $\mathcal{M}_\mathcal{R}$ to datapoint $x$ and $y^*_x$ denotes the data point's (unknown) true label.

We assume that a data point $x \in \mathcal{X}$ consists of a set of *atomic units*. For instance, for a RBBM that generates labels for training data (e.g., Snorkel [41]), $\mathcal{X}$ is the set of documents (data points) to be labeled, and each document consists of a set of words as atomic units. If the classification task is sentiment analysis on user reviews (for songs, movies, products, etc.), the set of labels assigned by such a RBBM may be $\mathcal{Y} = \{\text{POS}, \text{NEG}, \text{ABSTAIN}\}$. where $y_0 = \text{ABSTAIN}$.

For a RBBM for data cleaning, $\mathcal{X}$ is the set of input tuples in the database to be cleaned (to simplify the exposition we assume a single relation/table in the database, although our approach applies to multiple tables as well). Each tuple $t$ consists of a set of cells, denoted by $t[A]$ where $A$ is an attribute in the table. These cells are the atomic units of the tuple. The set of labels is $\mathcal{Y} = \{\text{CLEAN}, \text{DIRTY}\}$ (discussed further in Sec. 2.2), and here $y_0 = \text{CLEAN}$[1].

Example 2.1. *(a) Fig. 1 shows a set of reviews (data points) with possible labels of* POS *or* P *(a review with positive sentiment),* NEG *or* N *(a negative review), and* ABSTAIN *or - (the abstain label). The atomic units of the first review with id 0 are all the words that it contains, e.g., "*five*", "*stars*", etc. (b) Fig. 2a shows a sample of the Tax dataset with 10 tuples. The label of each tuple is either* CLEAN *or* DIRTY. *For instance, the atomic units of the first tuple $t_1$ are the cells '29447', 'IL', 'GROVER', 'Y', 3300, 864, ….*

### 2.2 Rules in the Black-Box Model

Rules are the building blocks of RBBMs. These rules are either designed by a human expert or discovered automatically (e.g., [4, 7, 13, 15, 31, 38, 52]). A rule consists of one or more predicates $\mathcal{P}_r$ from a set of domain-specific atomic predicates $\mathcal{P}$ over variables from a set $\mathcal{V}$ that represent data points. The atomic predicates $\mathcal{P}$ may contain simple predicates such as comparisons ($=, \neq, >, \geq, <, \leq$) as well as arbitrarily complex predicates such as black box functions.

[1]RBBMs like Holoclean [43] and Llunatic [20] update cells/attribute values instead of deleting tuples. We assume a model where tuples are labeled, i.e., if a tuple has any cell that is updated by the system, we consider the tuple to be labeled DIRTY.

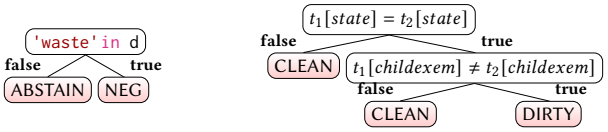



**Figure 3: A rule encoding of the LF `keyword_word_waste` from Fig. 1c (left) and rule encoding $dc_5$ from Fig. 2b (right)**

We represent a rule as a *tree* where leaf nodes represent labels in $\mathcal{Y}$ and the non-leaf nodes are labeled with predicates from $\mathcal{P}$. Each non-leaf node has two outgoing edges labeled by **true** and **false**. A rule $r$ is evaluated over assignments $\lambda$ that substitute the variables $\mathcal{V}_r$ of the rule with data points from $\mathcal{X}$. To determine the label for an assignment $\lambda$, the predicate of the root node of the rule's tree is evaluated by substituting the variables of the predicate based on $\lambda$ and then evaluating the resulting condition. Based on the outcome, either the edge labeled **true** or **false** is followed to one of the children of the root. The evaluation then continues with this child node until a leaf node is reached. Then the label of the leaf node is returned as the label for $\lambda$.

Definition 2.2 (Rule). *A rule $r$ with a set of variables $\mathcal{V}_r$ and atomic predicates $\mathcal{P}_r$ is a labeled directed binary tree where the internal nodes are predicates in $\mathcal{P}_r$, leaves are labels from $\mathcal{Y}$, and edges are marked with* **true** *and* **false**. *A rule $r$ takes as input an assignment $\lambda : \mathcal{V}_r \rightarrow \mathcal{X}$ mapping variables to data points and returns a label $r(\lambda) \in \mathcal{Y}$ for this assignment. Let $\text{ROOT}(r)$ denote the root of the tree for rule $r$ and let $C_{\textbf{true}}(v)$ ($C_{\textbf{false}}(v)$) denote the child of node $v$ adjacent to the outgoing edge of $v$ labeled* **true** *(***false***). Given an assignment $\lambda$, the result of rule $r$ for $\lambda$ is $r(\lambda) = \text{EVAL}(\text{ROOT}(r), \lambda)$. Function $\text{EVAL}(\cdot, \cdot)$ operates on nodes $v$ in the rule's tree and is recursively defined as follows:*

$$\text{EVAL}(v, \lambda) = \begin{cases} y & \text{if } v \text{ is a leaf labeled } y \in \mathcal{Y} \\ \text{EVAL}(C_{\textbf{true}}(v), \lambda) & \text{if } \lambda(v) \text{ is true} \\ \text{EVAL}(C_{\textbf{false}}(v), \lambda) & \text{if } \lambda(v) \text{ is false} \end{cases}$$

*Here $\lambda(v)$ denotes replacing the variables in the predicate of node $v$ with data points based on assignment $\lambda$.*

Our definition captures multiple models of rules used in RBBMs as we illustrate below, including denial constraints (DCs) and labeling functions (LFs). We provide linear time procedures for converting DCs and LFs to rules in App. A.

Example 2.3. *(a)* **Labeling Functions:** *Fig. 3 (left) shows the rule for a labeling function that returns* POS *if the length of the review contains the word '*waste*' and returns* ABSTAIN *otherwise. This rule has a single variable $w$ denoting a review (data point).*

*(b)* **Denial Constraints:** *Fig. 3 (right) shows a rule for the DC $dc_5$ from Fig. 2b. The set of atomic predicates including comparison operators of the form $t_1.A \diamond t_2.A, t_1.A = a$, etc., where $\diamond \in \{=, \neq, \geq, >, \leq, <, \cdots\}$. This rule has two variables $\mathcal{V}_r = \{t_1, t_2\}$. In Fig. 2a, suppose we assign $t_1 \rightarrow x_9$ and $t_2 \rightarrow x_{10}$. The output for this assignment is* CLEAN *as this assignment satisfies the first predicate of the rule ($t_1.state = t_2.state$), but not the second ($t_1.childexem \neq t_2.childexem$). The rule returns* DIRTY *for assignment $t_1 \rightarrow x_1$ and $t_2 \rightarrow x_2$ due to different values of* `childexem`.

We treat the model $\mathcal{M}_{\mathcal{R}}$ of an RBBM as a black box and do not make any assumption on how the labels generated for rules are combined, i.e., the RBBM may use any algorithm (combinatorial, ML-based, etc.) to compute the final labels for data points.

Definition 2.4 (Black-box model in a RBBM). *Given a set of data points $\mathcal{X}$, a set of labels $\mathcal{Y}$, and a set of rules $\mathcal{R}$, a black-box model $\mathcal{M}_{\mathcal{R}}$ takes $\mathcal{X}$, $\mathcal{R}$, and a data point $x \in \mathcal{X}$ as input, and returns a label $\mathcal{M}_{\mathcal{R}}(\mathcal{R}, \mathcal{X}, x) = \hat{y_x} \in \mathcal{Y}$ for $x$, by combining the labels generated by rules in $\mathcal{R}$ for assignments over $\mathcal{X}$ (Def. 2.2). We write $\mathcal{M}_{\mathcal{R}}(x)$ instead of $\mathcal{M}_{\mathcal{R}}(\mathcal{R}, \mathcal{X}, x)$ when $\mathcal{R}$ and $\mathcal{X}$ are understood from the context.*

## 2.3 User Feedback as Inputs to RuleCleaner

In this work, we employ a human-in-the-loop approach for repairing a set of rules. The user interacts with our systems as follows:

**Model Creation.** Based on the rules $\mathcal{R}$ provided by the user, the RBBM produces a model $\mathcal{M}_{\mathcal{R}} : \mathcal{X} \rightarrow \mathcal{Y}$.

**Ground Truth Labeling.** The user inspects the labels produced by the RBBM and specifies their expectations about the ground truth for labels as a partial function $C^* : \mathcal{X} \rightarrow \mathcal{Y}$ that provides the true label for a subset of the data points from $\mathcal{X}$. We assume that the user only gives ground truth labels to data points $x$ such that $\hat{y_x} \neq y_0$ (ABSTAIN for LFs, CLEAN for DCs), since these data points do not give any useful information about potential errors in the ruleset used by the RBBM. $C^*(x)$ can be POS, NEG for LFs and CLEAN, DIRTY for DCs. If $\hat{y_x} = C^*(x)$, then we call $C^*(x)$ a **confirmation**, i.e., the user *confirms* that the RBBM has returned the correct label for $x$. If $\hat{y_x} \neq C^*(x)$, then we call $C^*(x)$ a **complaint**, i.e., the RBBM returned a different label than expected. In Fig. 1, we have confirmations for reviews with ids 0, 2 ($\hat{y_x} = C^*(x) = P$), and complaints for reviews with ids 1, 3 ($\hat{y_x} = P, C^*(x) = N$) and 4 ($\hat{y_x} = N, C^*(x) = P$).

**Eliciting Labels for Assignments.** For RBBMs that have rules with more than one variable (e.g., DCs), we also require feedback on a subset of the assignments involving data points from $C^*$ through a partial function $C^{\Lambda} : \Lambda \rightarrow \mathcal{Y}$ (discussed further for DCs below).

**Rule Repair.** RuleCleaner then generates a repair $\Phi$ for the rules $\mathcal{R}$ that updates and deletes rules such that the fixed rules $\Phi(\mathcal{R})$ match the ground truth labels given by the user. The user can choose to repair the rules according to $\Phi$, or rerun RuleCleaner with additional ground truth labels. The rationale for this approach is that RBBMs are used in situations where obtaining the set of ground truth labels for all data points in $\mathcal{X}$ is infeasible. However, a human expert is typically capable of identifying the right label for a data point and for an assignment for a small subset, which can be used to refine the rules and improve the accuracy of the RBBM.

Note that we do not need labels for assignments $C^{\Lambda}$ for refining LFs in semi-supervised labeling. Here each rule $r$ uses a single variable (Fig. 3, left), i.e., there is only a single assignment for each data point $x$. Hence, $C^{\Lambda}$ can be derived automatically as $C^{\Lambda}(\lambda) = C^*(x)$ where $\lambda$ is the unique assignment for $x$.

**Labels for assignments $C^{\Lambda}$ for repairing DCs in data cleaning.** For DCs, the situation is more complicated as most DCs (like FDs) have more than one variable (see Fig. 3, right). Thus, there will be many assignments involving a single tuple (data point) and a data cleaning system will make choices based on the set of labels for all of these assignments. Recall that we only expect ground truth labels for data points where $x \neq y_0$. Thus, for repairing DCs, users are only asked for ground truth for data points $x$ such that $\hat{y_x}$ = DIRTY. We have two cases: (i) $C^*(x)$ = CLEAN (the tuple was incorrectly repaired) and (ii) $C^*(x)$ = DIRTY (the tuple was correctly repaired). Note that a "clean" tuple (true label = CLEAN) may still be involved in violations with other "dirty" tuples (true label = DIRTY). For instance, in Fig. 2a tuples $x_3$ and $x_4$ violate $dc_5$, but only $x_4$ is DIRTY (it's `childexem` value should be 1000 (the same as for $x_3$ which is a person in the same state and has the same `haschild` value). However, two tuples the user expects to be clean should never be in violation (e.g., $x_7$ and $x_8$ in this example).

Therefore, we can determine the ground truth label $C^{\Lambda}$ automatically for assignments $\lambda$ over two clean tuples in $C^*$ (with predicted labels DIRTY), i.e., if $\lambda(v_1) = x_1$ and $\lambda(v_2) = x_2$, we have:

$$C^*(x_1) = \text{CLEAN} \wedge C^*(x_2) = \text{CLEAN}) \Rightarrow C^{\Lambda}(\lambda) = \text{CLEAN}$$

For tuples $x$ such that $C^*(x) = \hat{y_x}$ = DIRTY (the user confirms that the tuple is dirty), we make the assumption that there is at least one assignment $\lambda_1$ such that $r(x)$ = DIRTY for some rule $r$. The motivation behind this assumption is that most data cleaning systems will only repair a tuple $x$ (flag $x$ as dirty) if it is involved in at least one violation wrt. to the input set of DCs. Note that the reverse may not be true - as mentioned earlier, a clean tuple may be in violation with other dirty tuples, and data cleaning systems like Holoclean [43] rely on other signals like statistical properties from the data to flag a tuple as dirty and may choose to not label some ground truth dirty tuples as dirty even if they are involved in violations (assignments leading to DIRTY labels for some rules).

The following example illustrates why user feedback on assignment labels $C^{\Lambda}$ is important in addition to data point labels $C^*$:

Example 2.5. *Consider Ex. 1.2, the incorrect $dc_5$ : `state` → `childexem` from Fig. 2b (in rule form in Fig. 3 (right)), and the repaired FD by* RuleCleaner *$dc_5'$ : `state`, `haschild` → `childexem`. For the instance shown below (only relevant attributes are shown), $x_2$ is in violation with both $x_1$ and $x_3$. Let us assume that the cleaning system has deleted $x_2$ and $x_3$, i.e., $\hat{y_{x_1}}$ = CLEAN and $\hat{y_{x_2}} = \hat{y_{x_3}}$ = DIRTY. The user has confirmed $x_2$, i.e., $C^*(x_2)$ = DIRTY. Before the repair for rule $r$ corresponding to $dc_5$, there are two assignments involving $x_2$ (ignoring symmetric assignments) that give $r(\lambda)$ = DIRTY in Fig. 3 (right): (1) $\lambda_1(t_1) = x_2, \lambda_1(t_2) = x_1$, and (2) $\lambda_2(t_1) = x_2, \lambda_2(t_2) = x_3$. Assignment $\lambda_1$ is a ground truth violation as it is also a violation for $dc_5'$ (both $x_1$ and $x_2$ are in the same state and do not have children). However, $\lambda_2$ is only a violation for the erroneous FD $dc_5$, but not with the correct FD $dc_5'$ ($x_2$ does not have children while $x_3$ has children, so their child exemptions may not be the same). Note that trying to preserve the results of non ground-truth assignments is likely to lead to incorrect repairs. If we ensured that the repaired version of $dc_5$ returns* DIRTY *for $\lambda_1$ and $\lambda_2$ (instead of only for the ground truth violation $\lambda_1$), then the repair cannot use the correct attribute* `haschild` *but has to use an incorrect attribute* `areacode`.

| | *state* | *haschild* | *childexem* | *areacode* | $\hat{y_x}$ | $C^*(x)$ |
|---|---|---|---|---|---|---|
| $x_1$ | *CA* | *0* | *0* | *901* | CLEAN | |
| $x_2$ | *CA* | *0* | *2500* | *901* | DIRTY | DIRTY |
| $x_3$ | *CA* | *1* | *3000* | *901* | DIRTY | |

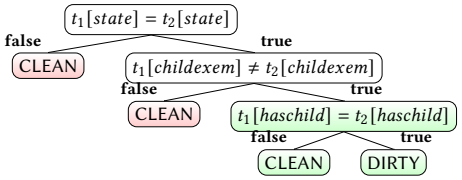



**Figure 4: A refinement of DC $r_5$ from Fig. 3 (right) that assigns label CLEAN to assignment $\lambda$: $\lambda(t_1) = x_5$ and $\lambda(t_2) = x_6$. Nodes added in refinement are highlighted in green.**

## 2.4 Rule Refinement

We consider two types of repair operations: (i) *deleting a rule*, and (ii) *refining a rule* by replacing a leaf node with a new predicate to match the desired ground truth labels of data points and assignments.

Definition 2.6 (Rule Refinement). *Consider a rule $r$, an assignment $\lambda$, a predicate $p$ to add to $r$, and a desired label for the assignment $y^* \in \mathcal{Y}$. Let $P$ be the path in $r$ taken by $\lambda$ leading to a leaf node $v$. Furthermore, consider two labels $y_1$ and $y_2$ such that $y_1 = y^*$ and $y_2 = y$ or $y_1 = y$ and $y_2 = y^*$. The refinement* **refine**$(r, \lambda, p, y_1, y_2)$ *of $r$ replaces $v$ with $p$ and adds the new leaf nodes for $y_1, y_2$, i.e.,*

$$r\left[v \leftarrow \begin{array}{c} p \\ \text{false} \quad \text{true} \\ y_1 \quad y_2 \end{array}\right]$$

Example 2.7. *We show a refinement in Fig. 4 for rule $r$ for DC $dc_5$ in Fig. 3 (right). Tuples $x_5$ and $x_6$ in dataset in Fig. 2a violate this constraint for assignment $\lambda : \{t_1 \mapsto x_5, t_2 \mapsto x_6\}$, since we get label $y = r(\lambda)$ = DIRTY from the rule. Assume the user has marked both tuples as clean $C^*(x_5) = C^*(x_6)$ = CLEAN. As described in Section 2.3, we infer that $C^\Lambda(\lambda)$ = CLEAN = the desired label $y^*$. Suppose we selected the predicate $p$ ($t_1[haschild] = t_2[haschild]$) to refine this rule (we postpone the discussion of how to come up with such a predicate to Sec. 4). We add the original label $y$ = DIRTY as the* **true** *child for $p$ and $y^*$ = CLEAN as the* **false** *child. The refined rule corresponds to the correct FD $dc_5'$ shown in Ex. 1.2.*

## 2.5 The Rule Repair Problem

Given the user feedback on ground truth labels and rule refinement described in the previous section, we now define the rule repair problem. Each repair operation $\phi$ can be either (i) deleting a rule from the ruleset $\mathcal{R}$, or (ii) refining a rule $r \in \mathcal{R}$ (Def. 2.6). A **repair sequence** denoted by $\Phi$ is a sequence of repair operations and $\Phi(\mathcal{R})$ denotes the result of applying $\Phi$ to $\mathcal{R}$. Note that multiple refinement operations may be required to repair a rule. In the problem definition shown below we make use of cost metric *cost* and accuracy measure for repairs. We will define these in the following. Intuitively, a repaired ruleset $\mathcal{R}'$ should be optimized to: (i) maximize the number of assignments from $C^*$ that are assigned the correct label by the RBBM using $\mathcal{R}'$ and (ii) minimize the changes to the input rules $\mathcal{R}$ to preserve the domain knowledge encoded in the rules.

Definition 2.8 (Rule repair problem). *Consider a black-box model $\mathcal{M}_\mathcal{R}$ that uses a set of rules $\mathcal{R}$, an input database $\mathcal{X}$, output labels $\mathcal{Y}$, and ground truth labels for a subset of data points $C^*$ and for a subset of assignments $C^\Lambda$. Given an accuracy threshold $\tau_{acc} \in \mathbb{R}$, the rule repair problem aims to find a repair sequence $\Phi$ that gives accuracy $\geq \tau_{acc}$ for the RBBM $\mathcal{M}_{\Phi(\mathcal{R})}$ with the refined rule set in terms of the ground truth labels $C^*$, i.e, find*

$$\begin{aligned} &\textbf{argmin}_\Phi && cost(\Phi) \\ &\textbf{subject to} && \textsc{Accuracy}(\mathcal{M}_{\Phi(\mathcal{R})}, C^*) \geq \tau_{acc} \end{aligned}$$

Note that since we treat the RBBM as a black box, there may not be a feasible solution to the above problem even if we fix all rules to match all the ground truth labels in $C^\Lambda$. In addition, the following theorem shows that finding an optimal repair is NP-hard, even for very simple RBBM models, hence we design algorithms that give good rule repairs in practice.

Theorem 2.9. *The rule repair problem is NP-hard in $\sum_{r\in\mathcal{R}} size(r)$.*

Sketch. We prove the theorem through a reduction from the Max-3SAT problem. The full proof is shown in App. C. □

**Accuracy.** We define the Accuracy of a repair as the number of data points from $C^*$ that receive the correct label by the RBBM over the repaired rules $\Phi(\mathcal{R})$. Let $\mathcal{X}_{C^*} \subseteq \mathcal{X}$ denotes the subset of data points for which the user has provided a ground truth label in $C^*$, and $\mathbb{1}(e)$ denote the indicator function that returns 1 if its input condition $e$ evaluates to true and 0 otherwise. Then

$$\textsc{Accuracy}(\mathcal{M}_{\Phi(\mathcal{R})}, C^*) = \frac{1}{|C^*|} \cdot \sum_{x \in \mathcal{X}_{C^*}} \mathbb{1}(\mathcal{M}_{\Phi(\mathcal{R})}(x) = C^*(x))$$

**Repair Cost.** We use a simple cost model. For a single repair operation $\phi$ in the repair sequence $\Phi$. For rule refinement operation (Sec. 2.4), since only one predicate is added at a time, we count cost = 1. For rule deletion operation, we count a fixed cost $\tau_{del} \in \mathbb{N}$:

$$cost(\phi) = \begin{cases} \tau_{del} & \text{if } \phi \text{ deletes } r \\ 1 & \text{if } \phi \text{ refines } r \text{ to } r' \end{cases}$$

For a repair sequence $\Phi$, we define $cost(\Phi) = \sum_{\phi\in\Phi} cost(\phi)$. Automated systems for rule generation may generate many spurious rules. For instance, DC discovery algorithms [7] often return many overly specific DCs when applied to noisy data. For such applications, the user may prefer deletion of rules, while for rule sets that were carefully curated by a human, refinement may be preferable to not loose the domain knowledge encoded in the rules. This can be controlled by changing $\tau_{del}$.

# 3 RULESET REPAIR ALGORITHM

In this section we describe a generic algorithm that repairs a ruleset $\mathcal{R}$ aiming to minimize the cost and maximize the accuracy of the RBBM. Since the optimization problem is intractable (Thm. 2.9), and the RBBM does not provide us with any information about its model $\mathcal{M}_\mathcal{R}$ except through the final labels it returns [2], we employ the greedy algorithm shown in Algorithm 1 to explore relevant

[2]Since we do not assume any property of the model $\mathcal{M}$ used in the RBBM, it is possible for the RBBM to return labels that do to match the expected labels from $C^*$ even if we repair each rule to return the correct labels for assignments for data points in $C^*$. However, as the labels produced by the rules are the only part of the system we can directly control and the model $\mathcal{M}_\mathcal{R}$ takes the output of these rules as input, we assume that fixing the predictions by the rules also improves the accuracy of the model for a practical system.

parts of the search space of rule deletion and refinement repairs. The algorithm takes as input the ground truth labels for a subset of data points $C^*$ and for a subset of assignments $C^\Lambda$, the set of rules $\mathcal{R}$, the cost of deletions $\tau_{del}$ (Sec. 2.5) that encodes the user's preference for deleting rules over fixing them, and two other configuration parameters – *a pre-deletion threshold* $\tau_{pre}$ based on which the algorithm prunes rules with poor accuracy early on, and a *model-reevaluation frequency* $\tau_{reeval}$ to specify how frequently the model is rebuild.

**(1) Deleting bad rules upfront.** As a first step, the algorithm prunes rules that perform poorly on $C^\Lambda$, i.e., whose accuracy on $C^\Lambda$ is less than $\tau_{pre} \in [0, 1]$. The rationale for this step is that some automated rule generation techniques may produce a large number of mostly spurious rules which should be removed early-on. For example, in [38], using the default setup on *Tax* dataset would generate over 7874 denial constraints whereas the number of known ground truth denial constraints are less than 10.

**(2) Refine or delete individual rules.** The key function in our greedy algorithm is `SingleRuleRefine`. We discuss this function in detail in Sec. 4 and present several implementations that trade-off between cost and runtime. `SingleRuleRefine` takes a single rule $r_i$ and refines it with respect to the assignment labels $C^\Lambda$, possibly by adding multiple predicates to $r_i$ in several refinement steps. After these refinements, if the total cost is $< \tau_{del}$, then the refinement of $r_i$ is accepted, otherwise, $r_i$ is removed from $\mathcal{R}$. We show a non-trivial property of `SingleRuleRefine` in Sec. 4 – under some mild assumptions on the space of predicates for refinement, `SingleRuleRefine` is guaranteed to succeed in repairing a rule such that the rule is consistent with every assignment for this rule specified in the ground truth labels in $C^\Lambda$.

**(3) Regenerate and test the model $\mathcal{M}$ periodically:** As we do not know whether repairing a subset of the rules is sufficient for causing the updated model to have high enough accuracy on $C^*$, we have to regenerate the model $\mathcal{M}$ in the RBBM periodically to test whether the repair we have produced so far is successful (the updated model's accuracy is above $\tau_{acc}$ for $C^\Lambda$). For Snorkel this steps amounts to retraining the model that Snorkel uses to predict labels based on the labels predicted by the rules. For data cleaning systems, this means rerunning the system with the updated set of DCs encoded by the rules to produce an updated repair.[3]

For most RBBM, the cost of regenerating the model is significantly higher than the cost of repairing the rules. To trade the high runtime of more frequent updates of the model for potentially less costly repairs, we update and evaluate the model every $\tau_{reeval}$ iterations. Infrequent retraining may result in repairs that affect more rules than necessary to achieve the desired accuracy. Note that Algorithm 1 always terminates after processing all rules, but, as discussed before, even repairing all rules may not guarantee that the desired accuracy will be achieved.

Example 3.1. *Consider Ex. 1.1 and Fig. 1 for refining labeling functions (LFs). Using the LFs from Fig. 1c generated by Witan [15], Snorkel has generated the predicated labels for the reviews shown in Fig. 1. Assume that the user has given the ground truth labels $C^*$ for reviews in this subset of the dataset (column* true label *in Fig. 1), which also forms the assignment labels $C^\Lambda$ (Sec. 2.3). Assume we set $\tau_{reeval} = 4$ (we rerun Snorkel only once after considering all 4 rules) and $\tau_{del} = 2$ (rules that require refinements with more than two new predicates get deleted instead of refined). Incorrectly predicted labels are highlighted in red (reviews 1, 3, 4).*

*In the first iteration we refine rule LF-1 (checking for stars). This rule returns incorrect labels for reviews 1 and 3. Function* `SingleRuleRefine` *generates the refinement of this rule shown in Fig. 1a (explained in Sec. 4) which adds a predicate* `'one' in x` *to the rule. The updated rule returns the correct label (or abstains) for all sentences in $C^* = C^\Lambda$. In the following iteration, rule LF-2 is not modified as they do not return incorrect labels. Rule LF-3 however, is refined in iteration 3 with an additional predicate* `'yes' in x` *to separate reviews 1 and 4. As all refinements require less than 2 (=$\tau_{del}$) new predicates, none of the rules get deleted.*

[3]Note that a refined version of a rule encoding a DC may not be representable as a DC. However, as we show in App. B a rule can be translated into a set of DCs that are equivalent to the rule.

**Algorithm 1:** Rule Set Repair

**Input :** Partial ground truth labels: $C^*$ for data points and $C^\Lambda$ for assignments, a set of rules $\mathcal{R} = \{r_1, \ldots, r_n\}$, the RBBM model $\mathcal{M}_\mathcal{R}$, an accuracy threshold $\tau_{acc}$, a deletion cost $\tau_{del}$, a pre-deletion threshold $\tau_{pre}$, and a model-reevaluation frequency $\tau_{reeval}$

**Output :** Repaired ruleset $\mathcal{R}_{repair}$

```
1  R_repair ← R
2  for r ∈ R do
3      if Accuracy(r, C*) < τ_pre then
4          R_repair ← R_repair \ {r}
5  for i ∈ [1, n] do
6      Φ ← SingleRuleRefine(r_i, C^Λ);
7      if cost(Φ) < τ_del then
8          R_repair ← Φ(R_repair);
9      else
10         R_repair ← R_repair \ {r_i};
11     if i % τ_reeval = 0 then    /* retrain every i iterations */
12         M_{R_repair} ← Reevaluate(R_repair, X)
13         if Accuracy(M_{R_repair}, C*) ≥ τ_acc then
14             return R_repair
15 return R_repair
```

**Complexity.** Algorithm 1 makes at most $n = |\mathcal{R}|$ calls to both the `SingleRuleRefine` procedure and the RBBM system to regenerate the model. We will analyze `SingleRuleRefine` in Sec. 4 and 4.3.

## 4 SINGLE RULE REFINEMENT

We now discuss algorithms for `SingleRuleRefine` used in Algorithm 1 to refine a single rule $r$ and establish some important properties of rule refinement repairs. We use $(\lambda, y) \in C^\Lambda$ to denote that $C^\Lambda(\lambda) = y$ has been specified in the partial ground truth for assignments provided by the user. As we are considering a single rule, we only consider assignments in $C^\Lambda$ that relate to $r$, but overload notation and use $C^\Lambda$ to denote this subset for the sake of simplicity.

**Algorithm 2:** SingleRuleRefine

**Input** : Rule $r$
Labelled Datapoints $C^\Lambda \subseteq \Lambda \times \mathcal{Y}$
**Output**: Repair sequence $\Phi$ such that $\Phi(r)$ fixes $C^\Lambda$

```
Y ← ∅, Λ_P ← ∅
P_fix ← {P[r, λ] | ∃(λ, y) ∈ C^Λ}
foreach P ∈ P_fix do
    Λ_P[P] ← {λ | ∃(λ, y) ∈ C^Λ : P = P[r, λ]}
r_cur ← r
/* Iterate over paths that need to be fixed                    */
foreach P ∈ P_fix do
    /* Fix path P to return correctly labels on C^Λ            */
    φ ← RefinePath(r_cur, Λ_P[P])
    r_cur ← φ(r_cur)
    Φ ← Φ.append(φ)
return Φ
```

Definition 4.1 (The Single Rule Refinement Problem). *Given a a rule $r$, a set of ground truth labels of assignments $C^\Lambda$, and a set of allowable predicates $\mathcal{P}$, find a sequence of refinement repairs $\Phi$ using predicates from $\mathcal{P}$ such that for the repaired rule $r' = \Phi(r)$ we have:*

$$\Phi = \underset{\Phi'}{\operatorname{argmin}}\, cost(\Phi') \quad \textbf{subject to} \quad \forall (\lambda, y) \in C^\Lambda : r'(\lambda) = y$$

The pseudocode for `SingleRuleRefine` is given in Algorithm 2. Given a single rule $r$, this algorithm determines a refinement-based repair $\Phi$ for $r$ such that $\Phi(r)$ returns the ground truth label $C^\Lambda(\lambda)$ for *all* assignments specified by the user in $C^\Lambda$ applicable to $r$. This algorithm uses the two following properties of refinement repairs: **(1) Independence of path repairs (Sec. 4.1):** Let $P_{fix}$ be the set of paths in $r$ taken by assignments from $C^\Lambda$. We show that each such path can be fixed independently. **(2) Existence of path repairs (Sec. 4.2):** We show that it is always possible to repair a given path such that it satisfies the desired labels of all assignments following this path if the space of predicates $\mathcal{P}$ satisfies a property that we call *partitioning*. **(3)** `RepairPath` **algorithms (Sec. 4.3):** We then present three algorithms with a trade off between runtime and repair cost that utilize these properties to repair individual paths in rule $r$. To ensure that all assignments following a given path $P$ in the rule $r$ get assigned their desired labels based on $C^\Lambda$, these algorithms add predicates at the end of $P$ to "reroute" each assignment to a leaf node with the ground-truth label.

We use the following notation in this section. $\mathcal{P}$ denotes the set of predicates we are considering. $\Lambda$ denotes the set of assignments in $C^\Lambda$ involving rule $r$, i.e., $\Lambda = \bigcup_{(\lambda,y)\in C^\Lambda}\{\lambda\}$. $P_{fix}$ denotes the set of paths (from the root to a label on a leaf) in rule $r$ that are taken by the assignments in $\Lambda$. For $p \in P_{fix}$, $\Lambda_p$ denotes all assignments from $\Lambda$ for which the path is $p$, hence also $\Lambda = \bigcup_{p\in P_{fix}} \Lambda_p$.

## 4.1 Independence of Path Repairs

The first observation we make is that for refinement repairs, the problem can be divided into subproblems that can be solved independently. As refinements only extend existing paths in $r$ by replacing leaf nodes with new predicate nodes, in any refinement $r'$ of $r$, the path for an assignment $\lambda \in \Lambda$ has as prefix a path from $P_{fix}$. That is, for $P_1 \neq P_2 \in P_{fix}$, any refinement on $P_1$ can only affect the labels for assignments in $\Lambda_{P_1}$, but not the labels of assignments in $\Lambda_{P_2}$ as all assignments in $\Lambda_{P_2}$ are bound to take paths in any refinement $r'$ that start with $P_2$. Hence we can apply repairs for all paths sequentially as stated below (the proof is shown in App. D.1).

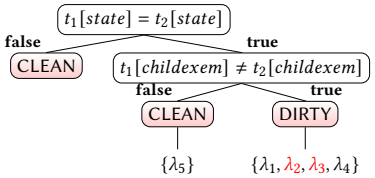



**Figure 5: Example for paths taken by assignments. Assignments with $C^\Lambda(\lambda)$ = DIRTY are highlighted in red.**

Proposition 4.2 (Path Independence of Repairs). *Given a rule $r$ and $C^\Lambda$, let $P_{fix} = \{P_1, \ldots, P_k\}$ and set $\Phi_i$ denote a refinement-based repair of $r$ for $\Lambda_{P_i}$ of minimal cost. Then $\Phi = \Phi_1, \ldots, \Phi_k$ is a refinement repair for $r$ and $C^\Lambda$ of minimal cost.*

## 4.2 Existence of Path Refinement Repairs

Next, we will show that is always possible to find a refinement repair for a path if the space of predicates $\mathcal{P}$ is *partitioning*, i.e., if for any two assignments $\lambda_1 \neq \lambda_2$ there exists $p \in \mathcal{P}$ such that:

$$\lambda_1(p) \neq \lambda_2(p)$$

Observe that for any two assignments $\lambda_1 \neq \lambda_2$, we have to assign different data points $x_1$ and $x_2$ to at least one variable $v$. These two data points differ in at least one atomic unit, say $A$, and $x_1[A] = c \neq x_2[A]$. If $\mathcal{P}$ includes all comparisons of the form $v[A] = c$, then any two assignments can be distinguished. In particular, for semi-supervised labeling with LFs as rules, where the atomic units are words, $\mathcal{P}$ is partitioning if $\mathcal{P}$ contains $w \in v$ for any variable $v$ and word $w$. For data cleaning where DCs are rules, the atomic units are attribute values. Thus, if $\mathcal{P}$ contains comparisons between attribute values and constants, then $\mathcal{P}$ is partitioning (we formally state this claim Lem. D.1 in App. D.2). The following proposition shows that when $\mathcal{P}$ is partitioning, we can always find a refinement repair. Further, we show an upper bound on the number of predicates to be added to a path $P \in P_{fix}$ to assign the ground truth labels $C^\Lambda$ to all assignments in $\Lambda$.

Proposition 4.3 (Upper Bound on Repair Cost of a Path). *Consider a rule $r$, a path $P$ in $r$, a partitioning predicate space $\mathcal{P}$, and ground truth labels $C$ for assignments for path $P$. Then there exists a refinement repair $\Phi$ for path $P$ and $C$ such that: $cost(\Phi) \leq |C|$.*

Proof Sketch. Our proof is constructive: we present an algorithm that given a set of assignments $\Lambda$ at a node iteratively selects two assignments $\lambda_1, \lambda_2 \in \Lambda_C$ such that $C(\lambda_1) \neq C(\lambda_2)$ and adds a predicate that splits $\Lambda$ into $\Lambda_1$ and $\Lambda_2$ such that $\lambda_1 \in \Lambda_1$ and $\lambda_2 \in \Lambda_2$. The full proof is shown in App. D.3. □

Of course, as we show in App. D.4, an optimal repair of $r$ wrt. $\Lambda$ may require less than $|\Lambda|$ refinement steps. Nonetheless, this upper bound will be used in the brute force algorithm we present in Sec. 4.3.2. The above proposition only guarantees that repairs with

a bounded cost exist. However, the space of predicates may be quite large or even infinite. Thus, it is not immediately clear how to select a separating predicate for any two given assignments $\lambda_1$ and $\lambda_2$. Note that if comparisons between atomic units and constants are included in $\mathcal{P}$, we can simply find in linear time variables which are assigned different data points, say $x_1$ and $x_2$ in $\lambda_1$ and $\lambda_2$ and select $v[A] = c$ such that $x_1[A] = c \neq x_2[A]$. More generally, in App. D.5 we show that we can determine equivalence classes of predicates in $\mathcal{P}$ wrt. a set of assignments and use only a single member from each equivalence class.

## 4.3 Path Repair Algorithms

Based on the insights presented in the previous sections we now present three algorithms for `RefinePath` used in Algorithm 2 to fix a given path $P \in P_{fix}$. These algorithms return a refinement-only repair sequence $\Phi$ for rule $r$ such that the repaired rule $r' = \Phi(r)$ returns the ground truth labels from $C^\Lambda$ for all assignments $\lambda \in \Lambda_P$. We will again overload $C^\Lambda$ to denote the ground truth assignments for $P$ and use $\Lambda$ to denote the set of assignments from $C^\Lambda$ for $P$. These algorithms refine $r$ by replacing $last(P)$ with a subtree to generate a refinement repair $\Phi$. As shown in App. D.5 , we only need to consider a finite number of predicates specific to $\Lambda$.

*4.3.1 GreedyPathRepair.* This algorithm (pseudocode in App. E.1 ) maintains a list of sets of assignments to be processed. This list is initialized with all assignments $\Lambda$. In each iteration, the algorithm picks two assignments $\lambda_1$ and $\lambda_2$ from $\Lambda$ and selects a predicate $p$ such that $p(\lambda_1) \neq p(\lambda_2)$. It then refines the rule with $p$ and appends $\Lambda_1 = \{\lambda \mid \lambda \in \Lambda \wedge p(\lambda)\}$ and $\Lambda_2 = \{\lambda \mid \lambda \in \Lambda \wedge \neg p(\lambda)\}$ to the list. As shown in the proof of Prop. 4.3, this algorithm terminates after adding at most $|\lambda|$ new predicates.

*4.3.2 BruteForcePathRepair.* The brute-force algorithm (pseudocode in App. E.1 ) is optimal, i.e., it returns a refinement of minimal cost (number of new predicates added). This algorithm enumerates all possible refinement repairs for a path $P$. Each such repair corresponds to replacing the last element on $P$ with some rule tree. We enumerate such trees in increasing order of their size and pick the smallest one that achieves perfect accuracy. We first determine all predicates that can be used in the candidate repairs. As argued in App. D.5, there are only finitely many distinct predicates (up to equivalence) for a given set $C^\Lambda$. We then process a queue of candidate rules each paired with the repair sequence that generated the rule. In each iteration, we process one rule from the queue and extend it in all possible ways by replacing one leaf node, and select the refined rule with minimum cost that satisfies all assignments. As we generate subtrees in increasing size, Prop. 4.3 implies that the algorithm will terminate and its worst-case runtime is exponential in $n = |\Lambda|$ as it may generate all substrees of size up to $n$.

*4.3.3 EntropyPathRepair.* GreedyPathRepair is fast, but has the disadvantage that it may use overly specific predicates that do not generalize well (even just on $C^\Lambda$). Furthermore, by randomly selecting predicates to separate two assignments (ignoring all other assignments for a path), this algorithm will often yield repairs with a higher than minimal cost. On the other hand, BruteForcePathRepair produces minimal repairs that also naturally generalize better, but the exponential runtime of the algorithm limits its applicability

**Algorithm 3:** EntropyPathRepair

**Input :** Rule $r$
Path $P$
Assignments to fix $\Lambda$
Expected labels for assignments $C^\Lambda$
**Output :** Repair sequence $\Phi$ which fixes $r$ wrt. $\Lambda$

$todo \leftarrow [(P, C^\Lambda)]$
$\Phi \leftarrow []$
$r_{cur} \leftarrow r$
$\mathcal{P}_{all} \leftarrow$ `GetAllCandPredicates`$(P, \Lambda, C^\Lambda)$
**while** $todo \neq \emptyset$ **do**
  $(P_{cur}, C) \leftarrow pop(todo)$
  $p_{new} \leftarrow \text{argmin}_{p \in \mathcal{P}_{all}} I_G(p, \Lambda_{cur})$
  $C_{\textbf{false}} \leftarrow \{(\lambda, y) \mid (\lambda, y) \in C \wedge \neg p(\lambda)\}$
  $C_{\textbf{true}} \leftarrow \{(\lambda, y) \mid (\lambda, y) \in C \wedge p(\lambda)\}$
  $y_{max} \leftarrow \text{argmax}_{y \in \mathcal{Y}} |\{\lambda \mid C_{\textbf{true}}(\lambda) = y|$
  $\phi_{new} \leftarrow$ **refine**$(r_{cur}, P_{cur}, y_{max}, p,$ **true**$)$
  $r_{cur} \leftarrow \phi_{new}(r_{cur})$
  $\Phi \leftarrow \Phi, \phi_{new}$
  **if** $|\mathcal{Y}_{C_{\textbf{false}}}| > 1$ **then**
    $todo.push((P[r_{cur}, C_{\textbf{false}}], C_{\textbf{false}}))$
  **if** $|\mathcal{Y}_{C_{\textbf{true}}}| > 1$ **then**
    $todo.push((P[r_{cur}, C_{\textbf{true}}], C_{\textbf{true}}))$
**return** $\Phi$

in practice. We now introduce EntropyPathRepair, a more efficient algorithm that avoids the exponential runtime of BruteForcePathRepair while typically producing more general and less costly repairs than GreedyPathRepair. We achieve this by greedily selecting predicates to split a set of assignments into two sets that best separates assignments with different labels at each step.

To measure the quality of a split, we employ the entropy-based *Gini impurity score* $I_G$ [29]. Given a candidate predicate $p$ for splitting a set of assignments and their labels $C$, we denote the subsets of $C$ generated by splitting $C$ based on $p$:

$$C_{\textbf{false}} = \{(\lambda, y) \mid (\lambda, y) \wedge \neg p(\lambda)\} \quad C_{\textbf{true}} = \{(\lambda, y) \mid (\lambda, y) \wedge p(\lambda)\}$$

Using $C_{\textbf{false}}$ and $C_{\textbf{true}}$ we define $I_G(p)$ for a predicate $p$ as shown below:

$$I_G(p) = \frac{|C_{\textbf{false}}| \cdot I_G(C_{\textbf{false}}) + |C_{\textbf{true}}| \cdot I_G(C_{\textbf{true}})}{|C|}$$

$$I_G(C) = 1 - \sum_{y \in \mathcal{Y}_C} p(y)^2 \qquad p(y) = \frac{|\{\lambda \mid C(\lambda) = y\}|}{|C|}$$

$I_G(C)$ is minimal if $\mathcal{Y}_C = \{y \mid \exists \lambda : (\lambda, y) \in C\}$ contains a single label. Intuitively, we want to select predicates such that all assignments that reach a particular leaf node are assigned the same label. At each step the best separation is achieved by selecting a predicate $p$ that minimizes $I_G(p)$.

Algorithm 3 first determines all candidate predicates we can use using function `GetAllCandPredicates`. Then it iteratively selects predicates until all assignments achieve the expected label by the

rule. For that we maintain again a queue of paths paired with a map $C$ from assignments to expected labels that still need to be processed. In each iteration of the algorithm's main loop, we pop one pair of a path $P_{cur}$ and assignments with labels $C$ from the queue. We then determine the predicate $p$ that minimizes the entropy of $C$. Afterwards, we create the subsets of assignments from $C$ which fulfill $p$ and those who do not. We then generate a refinement repair step $\phi_{new}$ for the current version of the rule ($r_{cur}$) that replaces the last element on $P_{cur}$ with predicate $p$. The node at the **true** edge of the node for $p$ is then assigned the most prevalent label $y_{max}$ for the assignments which will end up in this node (the assignments from $C_{\textbf{true}}$). Finally, unless they only contain one label, new entries for $C_{\textbf{false}}$ and $C_{\textbf{true}}$ are appended to the todo queue.

*4.3.4 Correctness.* The following theorem shows that all three path repair algorithms are correct (proof in App. E.2 ).

Theorem 4.4 (Correctness). *Consider a rule $r$, ground-truth labels of a set of assignments $C^{\Lambda}$, and partitioning space of predicates $\mathcal{P}$. Let $\Phi$ be the repair sequence produced by GreedyPathRepair, BruteForcePathRepair, or EntropyPathRepair. Then we have:*

$$\forall(\lambda, y) \in C^{\Lambda} : \Phi(r)(\lambda) = y$$

## 5 RELATED WORK

We next survey related work on tasks that can be modeled as RBBMs as well as discuss approaches for automatically generating rules for RBBMs and improving a given rule set.

**Semi-supervised systems.** Semi-supervised systems have been proposed in various contexts [20, 35, 41–43, 47], e.g., for classification [41, 42, 47], for data repair [20, 43], or for entity matching [35]. The main advantage of such systems is interpretability.

**Automated rule discovery.** As manual ganeration of rules for a RBBM can be time-consuming, there has been a significant amount of work on automatically discovering such rules. In the context of data cleaning, many techniques [7, 13, 19, 25–27, 36–38, 40, 46, 53, 54] have been proposed for automatically discovering constraints from data. However, datasets are rarely 100% clean. Thus, any constraint discovery algorithm will either not discover some ground truth constraints as they are violated in the input data or has to allow for some level of approximation (constraints are allowed to be violated) which can lead to the discovery of many spurious constraints. If automatically discovered constraints are used for data cleaning [20, 21, 43] this can lead to hard to trace errors as the data is made to confirm to the erroneous constraints. Perhaps surprisingly, combined constraint and data repair is a less studied problem [6, 11, 12, 48]. In constrast, RuleCleaner applies a human-in-the-loop approach to repair rules based on user-provided feedback about the ground truth for a subset of the data. It, thus, is suited well for repairing the rules generated by constraint discovery.

**Semi-supervised labeling.** In the context of labeling functions, ULF [45] is an unsupervised system for fixing labeling functions using k-fold cross validation, extending previous approaches aimed at compensating for labeling errors [34, 51]. Witan [15] is a system for automatically generating labeling functions. While the labeling functions produced by Witan are certainly useful, we demonstrate in our experimental evaluation that applying RuleCleaner to Witan LFs can significantly improve accuracy. Hsieh et al. [24] propose a framework called Nemo for selecting data to show to the user for labeling function generation based on a utility metric for LFs and a model of user behavior (given some data how likely is the user to propose a particular LF). Furthermore, Nemo specializes LFs to only be applicable to the neighborhood of data (user developed LFs are likely more accurate to data similar to the data based on which they were created). However, in contrast to RuleCleaner, Nemo does not provide a mechanism for the user to provide feedback on the result of semi-supervised training data generation and to use this information to automatically delete and refine rules.

| dataset | avg(#word) | #row | $\mathcal{Y}$ | # Witan LF |
|---|---|---|---|---|
| *Amazon* [33] | 68.86 | 200,000 | positive/negative | 15 |
| *Amazon-0.5* | 56.01 | 100,000 | positive/negative | 15 |
| *Enron* [28] | 317.03 | 27,716 | ham/spam | |
| *YTSpam* [2] | 15.60 | 1,957 | ham/spam | |
| *PT* | | 24,588 | professor/teacher | 7 |
| *PA* | | 12,236 | painter/architect | 10 |

**Table 1: LF dataset statistics**

**Explanations for semi-supervised systems.** While there is a large body of work on explaining the results of semi-supervised systems, most of this work has stopped short of repairing the rules of a RBBM and, thus, are orthogonal to our work. However, it may be possible to utilize the explanations provided by such systems to guide a user in selecting what data points to label. In the context of data cleaning, prior work proposed approaches for explanations / repairs that involve both the data and constraints [3, 10, 11, 16, 17, 22, 49, 50]. Importantly, none of these works has developed an approach for fixing the constraints in order to improve it for black-box repair algorithms.

## 6 EXPERIMENTS

RuleCleaner is implemented in Python (version 3.8) and runs on top of PostgreSQL (version 14.4). Experiments were run on Oracle Linux Server 7.9 with 2 x AMD EPYC 7742 CPUS, 128GB RAM. We evaluate RuleCleaner for semi-supervised labeling (LF) using *Snorkel* [41] as the RBBM and constraint-based data cleaning (DC) using *MuSe* [21] as the RBBM. We evaluate both the runtime and the quality of the repairs produced by our system with respect to several parameters. We also use our system to rules generate automatically using Witan [15] and DCFinder [38].

**Datasets and rules.** We use the following datasets for the LF experiments. *YTSpam*: comments from youtube videos, *Enron*: emails sent from/to employees of the company Enron, *Amazon*: product reviews from Amazon and their sentiment label (POS/NEG), *PT*: descriptions of individuals each labeled as a professors or a teacher, and *PA*: descriptions of individuals each labeled a painter or an architect. Additional information about these datasets is shown in Tab. 1. To generate LF rules, we implemented a rule generator that uses known ground truth to generate "good" and "bad" rules by identifying tokens that occur frequently (infrequently) in sentences with a given ground truth label (all LF datasets we use have ground truth labels). For experiments with Witan, we use the labeling functions produced by that system (number of labeling functions shown in Tab. 1). For the DC use case, we used the Tax [9] dataset using a set of ground truth DCs and also DCs generated using DCFinder.

As ground truth labels for tuples are not available, we randomly inject ~ 5% errors some of which violate ground truth constraints and some of which violate spurious constraints.

**Parameters and Metrics.** In our experiments we vary $|C^*|$, the *input size*, the *complaint ratio* (fraction of $C^*$ that are complaints, i.e., $C^*(x) \neq \mathcal{M}_{\mathcal{R}}(x)$, and parameter $\tau_{del}$. To evaluate the quality of a rule repair, we measure the *fix rate* (fraction of data points from $C^*$ that are complaints and receive the right label after the repair) and the *preserv. rate* (fraction of data points from $C^*$ that are confirmations and receive the right label after the repair). Furthermore, to evaluate how well our rules generalize, we measure the *global acc.* (accuracy of the original rules on the full dataset) and the *new global acc.* (accuracy of the updated rules $\Phi(\mathcal{R})$ on the full dataset). We measure the cost of a repair as the *avg. size incr.* (the cost of the repair relative to the number of rules).

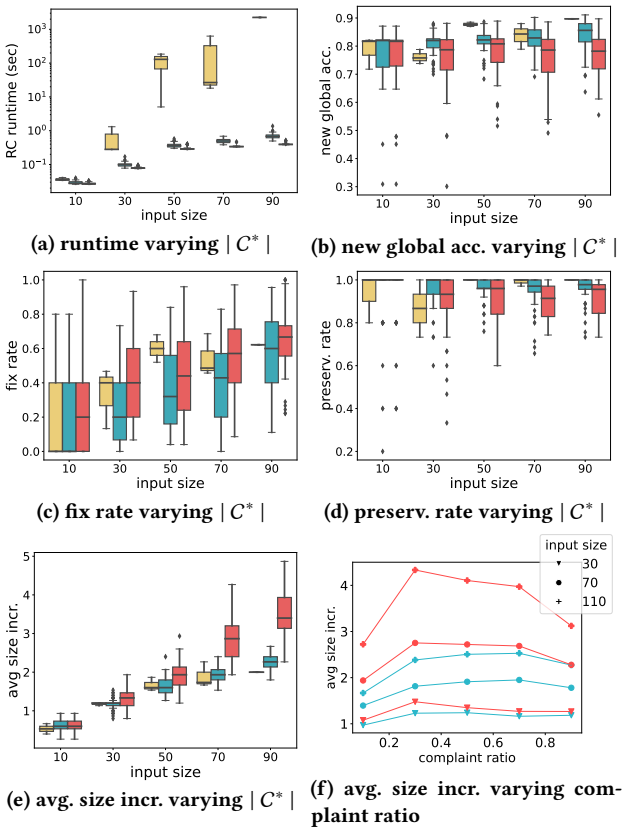
**Competitors & Baselines.** We compare all three predicate selection strategies from Sec. 4.3: *Greedy* (GreedyPathRepair), *Entropy* (EntropyPathRepair), and *Brute Force* (BruteForcePathRepair).

## 6.1 Semi-Supervised Labeling

We first evaluate the effectiveness of the three algorithms for selecting predicates. For each run, we selected the same number of user inputs which are 50% complaints and 50% confirmations (complaint ratio = 50%) based on the model-predicted labels and ground truth labels. For experiments on labeling functions Fig. 6, we used *YTSpam* dataset, with 30 keyword labeling functions generated using our keyword function generator as input (20 functions are of high accuracy while 10 are low quality). We repeated each experiment twice for *Brute Force* (given the long runtime of this algorithm more repetitions are not feasible) and 50 times for *Entropy* and *Greedy*. We determined the number of required repetitions to stabilize results in preliminary experiments. For DC experiments, we used a sample of Tax dataset with 7 denial constraints (some which were designated as ground truth based on constraints that have been used for this dataset in the literature). We simulated the user by finding pairs of tuples in violation wrt. the ground truth DCs as confirmations and violations of spurious DCs as complaints. The results from DC experiments are relatively stable and, thus, 10 repetitions were sufficient.

As shown in Fig. 6a, the runtime for *Brute Force* is up to ~4 orders of magnitude larger than the runtime of *Entropy* and *Greedy*. As shown in Fig. 6b, *Brute Force* and *Entropy* achieve similar results in terms of new global acc. and both outperform *Greedy*. In Fig. 6c, the fix rate for *Greedy* is the highest among the 3 algorithms. The reason behind this result is that *Greedy* tends to overfit to the user input by selecting predicates that distinguish individual data points without considering whether they generalize to other data points.

Finally, Fig. 6e shows the avg. size incr.. The results confirm our assumption that *Greedy* generates overly specific repairs that lead to a large rule size increase. For all algorithms, the avg. size incr. increases when we increase the input size, which is expected since larger input size entails that we have more labels to comply with, which typically requires more refinement steps. In summary, while *Greedy* has a high average fix rate, it tends to overfit to the user input. For *Brute Force*, although it finds the best repairs in terms of size increase and has the best overall new global acc. after the fix,




(a) runtime varying $|C^*|$ (b) new global acc. varying $|C^*|$

(c) fix rate varying $|C^*|$ (d) preserv. rate varying $|C^*|$

(e) avg. size incr. varying $|C^*|$ (f) avg. size incr. varying complaint ratio

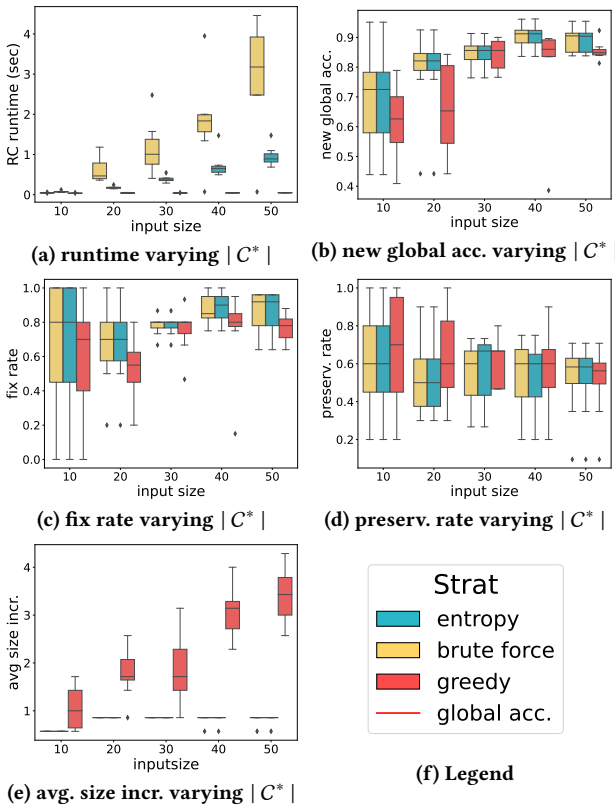
**Figure 6: Quality results for semi-supervised labeling**

the runtime is OOM higher than the other 2 algorithms. Given the poor runtime performance of *Brute Force*, we focus on *Greedy* and *Entropy* for the remaining experiments.

**Varying complaint ratio.** We also evaluated how the ratio between complaints and confirmations (complaint ratio) affects repairs. Fig. 6f shows the avg. size incr. for different input size values when varying the complaint ratio. As the complaint ratio increases, there's a trend for the size increase that goes up and peaks at around 50% complaint rate and then goes down after the peak. In App. F.1 we present a thorough evaluation of the impact of complaint ratio on the quality and size of the repaired ruleset. Based on these results we choose a complaint ratio of 50% for the remaining experiments as it provides a good trade-off between the different metrics.

## 6.2 Data Cleaning

Fig. 7 shows results for varying input size for the DC use case. *Entropy* and *Brute Force* behave mostly the same on the Tax dataset. This is due to the relatively small number and relatively few refinement steps that are needed for fixing the rules. The global accuracy before the fix is low as a few constraints that result in a lot of violations can significantly impact accuracy. All three algorithms improve global acc. and more significantly so for larger input size. *Entropy* and *Brute Force* outperform *Greedy* on new global acc.. In general the trends for DC are similar to those we observed for LF. One exception is preserv. rate, which is lower for DC. This is




(a) runtime varying | $C^*$ | (b) new global acc. varying | $C^*$ |

(c) fix rate varying | $C^*$ | (d) preserv. rate varying | $C^*$ |

(e) avg. size incr. varying | $C^*$ | (f) Legend

**Figure 7: Quality results for data cleaning**

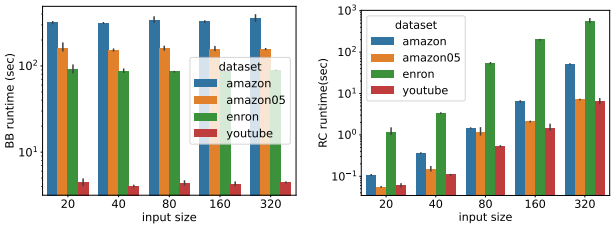
due to the fact that a single data point (tuple) may be involved in several violations and after the repair, MuSe may change which of the tuples involved in a violation will be deleted leading to some tuples involved in violations confirmed by the user to no longer be deleted by MuSe.

## 6.3 Scalability

In this section, we evaluate the scalability of RuleCleaner. The summary of the datasets we used are summarized in Tab. 1. For LF repair, we used *Entropy*, with complaint ratio =50% and $\tau_{del}$ =0. For each dataset we vary the input size and measure the run time for both the black box model (Snorkel or MuSe) and runtime of the repair generation in RuleCleaner. As shown in Fig. 8b, the runtime of Snorkel is linear in the size of the data. However, for RuleCleaner, the *avg word cnt* of sentences (reported in Tab. 1) plays a huge factor in affecting the runtime. Recall that in *Entropy*, when generating the best repair candidate predicate, we iterate over all words included in data points in $C^*$. Even though dataset size does play a factor in affecting the runtime (*YTSpam* dataset being the fastest), sentence size is the more important factor in affecting the runtime of RuleCleaner.

## 6.4 Varying Reevalation Frequency

In the experiments discussed so far, we have let RuleCleaner fix all rules and only reevaluate the RBBM once after the repair. We now investigate the impact of the retraining frequency $\tau_{reeval}$ on




(a) Black Box Model Runtime (b) RuleCleaner Runtime

**Figure 8: Scalability: semi-supervised labeling**

runtime, repair cost, and repair quality. Intuitively, if we retrain more frequently then we may stop early once a repair with high enough accuracy has been found which reduces repair cost (we leave some rules untouched) but will typically result in lower new global acc.. Intuitively, it is easier for the user to interpret the rules with smaller tree size. We use the *YTSpam* dataset and 30 labeling functions generated using our keyword labeling function generator. Fig. 12 shows the total runtime (RuleCleaner + black box model) and average tree size increase when we set the accuracy threshold to 0.6. In addition to the $\tau_{reeval}$ (x axis) we also vary the user input size (50% complaint ratio). For example, if the retrain frequency=0.1, we will rerun the RBBM after fixing 10% of rules and compute the quality of the updated rules. The more frequent we regenerate the black box model, the larger the total runtime. Furthermore, more frequent retraining decreases the repair cost.

## 6.5 Varying Deletion Cost

We now evaluate the impact of parameter $\tau_{del}$. We prepared several sets of LFs (30 LFs each) using our keyword LF generator. Each set contains a certain number of "good" functions (predictive of a label) and "bad" functions (not predictive). We used *YTSpam* and did set input size to 200. We present the results of 4 different sets of input LFs in Fig. 9. They have 3,9,15, and 21 bad LFs, respectively. Recall that if a refinement repair for a rule requires more than $\tau_{del}$ new predicates, then we delete the rule. In addition to new global acc. we also measure *rel. size*, the size of the complete ruleset after the repair relative to the size of the rules before the repair, and *hit rate*, the percentage of the "bad" functions that get deleted. As shown in Fig. 9 more aggressive deletion leads to a higher hit ratio and lower repair cost at the cost of lower global accuracy. For larger fractions of bad functions, we can delete a large fraction of bad functions without significant degradation of accuracy.

## 6.6 Repairing Witan Labeling Functions

To test the effectiveness of RuleCleaner in improving an automatically generated ruleset, we used labeling functions generated by Witan [15] for *Amazon*, *PT*, and *PA* as input. We still use Snorkel [41] as the RBBM. A summary of the datasets and number of labeling functions is shown in Tab. 1. Ex. 1.1 shows two of the repaired rules generated by RuleCleaner for Amazon dataset. Fig. 10 shows repaired rules for *PT* and *PA*. For the *PA* rule shown in Fig. 10a (distinguishing architects from painters), our system added a check for word *innovative*. As high emphasis is placed on innovation in architecture, this is sensible.

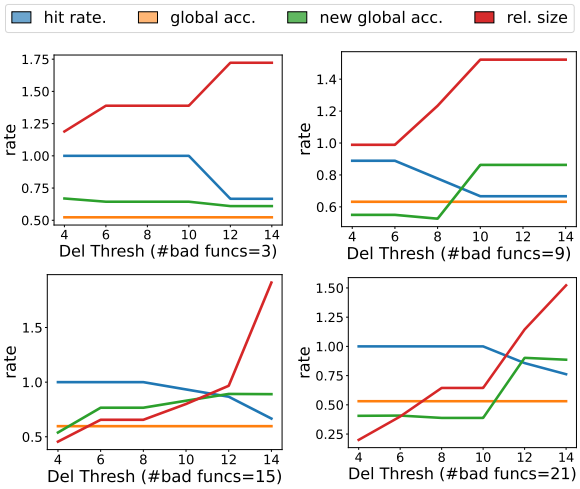
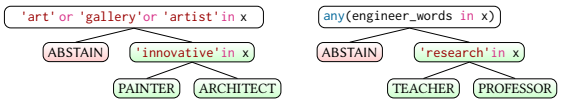
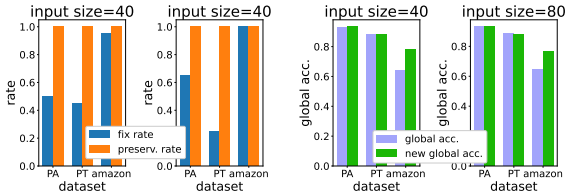



**Figure 9: Deletion**




**(a) Refined rule for PA** **(b) Refined rule for PT**

**Figure 10: Refined Witan LFs**




**(a) fix rate, preservation rate** **(b) global accuracy**

**Figure 11: Witan Case Study Quality**

For the *PT* rule shown in Fig. 10b (labeling persons as teachers or professors), the refined rule added a check for word *research* in the person description. This is very sensible since professors tend to be more involved in research compared to school teachers. Fig. 11 shows the quality of our repaired rule sets for two input sizes. We improved or kept the global accuracy after the fixes. We also achieve a decent fix rate and preserv. rate.

## 6.7 Reparing DCFinder results

In this experiment, we apply RuleCleaner to a mix of DCs generated by DCFinder [38] as well as ground truth DCs from the data cleaning literature for the Tax dataset. DCFinder produced 7874 DCs for this dataset. To increase the challenge for RuleCleaner, pruned out a set of DCs that are obviously wrong by limiting the number of predicates a valid DC can have. We also filtered out DCs that have meaningless operators such as $t_1.\texttt{areacode} < t_2.\texttt{areacode}$. This pruning steps reduced the number of constraints to 35 constraints. We combined this set with a set of 8 ground truth constraints for the Tax dataset. We first evaluate the impact of setting $\tau_{pre} = 0.5$ which deleted all but 7 of the "bad" constraints, but kept all 8 ground truth constraints. In an additional experiment, we evaluated whether our approach can successfully recover the ground truth denial constraint using *Entropy*. We *"polluted"* two

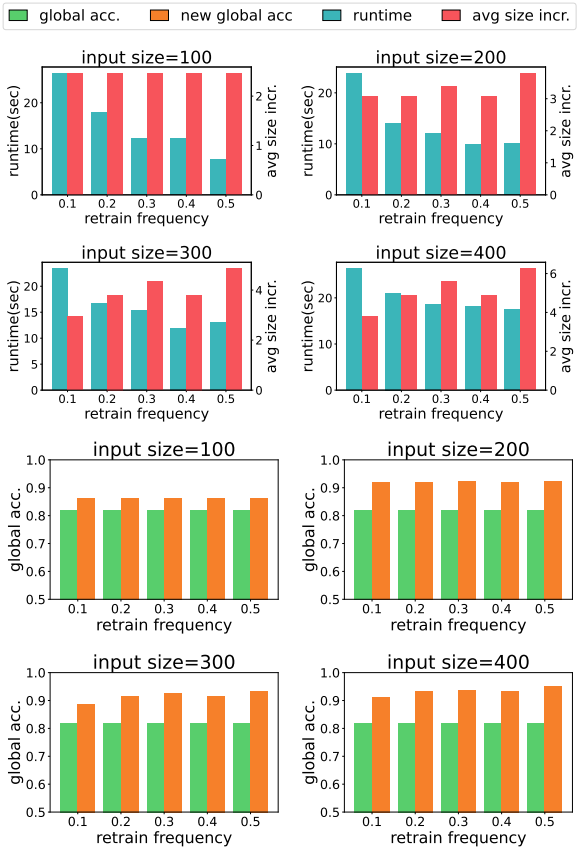



**Figure 12: Varying $\tau_{reeval}$, retraining every x%**

ground truth constraints by removing one of their predicates. RuleCleaner successfully recovered the correct original DCs. One result was shown in Ex. 1.2 already. The other *"polluted"* DC was generated by removing predicate $t_1$.state=$t_2$.state from $\neg$ ($t_1$.salary=$t_2$.salary $\wedge$ $t_1$.state=$t_2$.state $\wedge$ $t_1$.rate $\neq$ $t_2$.rate) to generate $\neg$ ($t_1$.salary=$t_2$.salary $\wedge$ $t_1$.rate $\neq$ $t_2$.rate).

## 7 CONCLUSIONS AND FUTURE WORK

In this work, we introduced RBBMs, a general model for systems that combine a set of interpretable rules with a model that combines the outputs produced by the rules to predict labels for a set of datapoints in a semi-supervised fashion. Many important applications including semi-supervised labeling and constraint-based data cleaning can be modeled as RBBMs. We develop a human-in-the-loop approach for repairing a set of RBBM rules based on a small set of ground truth labels generated by the user. Our algorithm is highly effective in improving the accuracy of RBBMs by improving rules created by a human expert or automatically discovered by a system like Witan [15] or DCFinder [38]. In future work, we will explore the application of our rule repair algorithms to other tasks that can be modeled as RBBM, e.g., information extraction based on user-provided rules [30, 44]. In this work, we used ground truth labels for both data points and assignments. It will be interesting to investigate whether the rules can be repaired based on the ground truth of only the data points. Furthermore, it would be interesting to use explanations for rule outcomes to guide the user in what

data points to inspect and to aide the system in selecting which rules to repair, e.g., repair rules that have high responsibility for a wrong result.

## REFERENCES


[1] Foto N. Afrati and Phokion G. Kolaitis. 2009. Repair Checking in Inconsistent Databases: Algorithms and Complexity. In *ICDT*. 31–41.

[2] Túlio C. Alberto, Johannes V. Lochter, and Tiago A. Almeida. 2015. TubeSpam: Comment Spam Filtering on YouTube. In *ICML*. 138–143.

[3] Laure Berti-Équille and Ugo Comignani. 2021. Explaining Automated Data Cleaning with CLeanEX. In *IJCAI-PRICAI 2020 Workshop on Explainable Artificial Intelligence (XAI)*.

[4] Laure Berti-Équille, Hazar Harmouch, Felix Naumann, Noël Novelli, and Saravanan Thirumuruganathan. 2018. Discovery of Genuine Functional Dependencies from Relational Data with Missing Values. *PVLDB* 11, 8 (2018), 880–892.

[5] Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. 2013. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. *Theory Comput. Syst.* 52, 3 (2013), 441–482.

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

[7] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient Denial Constraint Discovery with Hydra. *PVLDB* 11, 3 (2017), 311–323.

[8] Benedikt Boecking, Willie Neiswanger, Eric P. Xing, and Artur Dubrawski. 2021. Interactive Weak Supervision: Learning Useful Heuristics for Data Labeling. In *ICLR*.

[9] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional Functional Dependencies for Data Cleaning. In *ICDE*. 746–755.

[10] Anup Chalamalla, Ihab F Ilyas, Mourad Ouzzani, and Paolo Papotti. 2014. Descriptive and prescriptive data cleaning. In *SIGMOD*. 445–456.

[11] Fei Chiang and Renée J. Miller. 2011. A unified model for data and constraint repair. In *ICDE*. 446–457.

[12] Fei Chiang and Siddharth Sitaramachandran. 2016. Unifying Data and Constraint Repairs. *ACM J. Data Inf. Qual.* 7, 3 (2016), 9:1–9:26.

[13] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *PVLDB* 6, 13 (2013), 1498–1509.

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

[15] Benjamin Denham, Edmund M.-K. Lai, Roopak Sinha, and M. Asif Naeem. 2022. Witan: Unsupervised Labelling Function Generation for Assisted Data Programming. *PVLDB* 15, 11 (2022), 2334–2347.

[16] Daniel Deutch, Nave Frost, Amir Gilad, and Oren Sheffer. 2020. T-REx: Table Repair Explanations. In *SIGMOD*. 2765–2768.

[17] Daniel Deutch, Nave Frost, Amir Gilad, and Oren Sheffer. 2021. Explanations for Data Repair Through Shapley Values. In *CIKM*. 362–371.

[18] Ronald Fagin, Benny Kimelfeld, and Phokion G. Kolaitis. 2015. Dichotomies in the Complexity of Preferred Repairs. In *PODS*. 3–15.

[19] Wenfei Fan, Floris Geerts, Laks V. S. Lakshmanan, and Ming Xiong. 2009. Discovering Conditional Functional Dependencies. In *ICDE*. 1231–1234.

[20] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC Data-Cleaning Framework. *PVLDB* 6, 9 (2013), 625–636.

[21] Amir Gilad, Yihao Hu, Daniel Deutch, and Sudeepa Roy. 2020. MuSe: Multiple Deletion Semantics for Data Repair. *PVLDB* 13, 12 (2020), 2921–2924.

[22] Daniel Haas, Sanjay Krishnan, Jiannan Wang, Michael J Franklin, and Eugene Wu. 2015. Wisteria: Nurturing scalable data cleaning infrastructure. *PVLDB* 8, 12 (2015), 2004–2007.

[23] Ruining He and Julian McAuley. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *WWW*. 507–517.

[24] Cheng-Yu Hsieh, Jieyu Zhang, and Alexander J. Ratner. 2022. Nemo: Guiding and Contextualizing Weak Supervision for Interactive Data Programming. *PVLDB* 15, 13 (2022), 4093–4105.

[25] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: an Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.* 42, 2 (1999), 100–111.

[26] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *SIGMOD*. 647–658.

[27] Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. 2023. Discovering Similarity Inclusion Dependencies. *Proc. ACM Manag. Data* 1, 1 (2023), 75:1–75:24.

[28] Bryan Klimt and Yiming Yang. 2004. The Enron Corpus: A New Dataset for Email Classification Research. In *Proceedings of the 15th European Conference on Machine Learning*. 217–226.

[29] Sotiris B. Kotsiantis. 2013. Decision Trees: a Recent Overview. *Artif. Intell. Rev.* 39, 4 (2013), 261–283.

[30] B. Liu, L. Chiticariu, V. Chu, HV Jagadish, and F.R. Reiss. 2010. Automatic Rule Refinement for Information Extraction. *PVLDB* 3, 1 (2010).

[31] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. 2020. Approximate Denial Constraints. *PVLDB* 13, 10 (2020), 1682–1695.

[32] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. 2018. Computing Optimal Repairs for Functional Dependencies. In *PODS*. 225–237.

[33] Julian J. McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-Based Recommendations on Styles and Substitutes. In *SIGIR*. 43–52.

[34] Curtis G. Northcutt, Lu Jiang, and Isaac L. Chuang. 2021. Confident Learning: Estimating Uncertainty in Dataset Labels. *J. Artif. Intell. Res.* 70 (2021), 1373–1411.

[35] Fatemah Panahi, Wentao Wu, AnHai Doan, and Jeffrey F Naughton. 2017. Towards Interactive Debugging of Rule-based Entity Matching.. In *EDBT*. 354–365.

[36] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: an Experimental Evaluation of Seven Algorithms. *PVLDB* 8, 10 (2015), 1082–1093.

[37] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *SIGMOD*. 821–833.

[38] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *PVLDB* 13, 3 (2019), 266–278.

[39] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. 2021. Fast Detection of Denial Constraint Violations. *PVLDB* 15, 4 (2021), 859–871.

[40] Eduardo H. M. Pena, Fábio Porto, and Felix Naumann. 2022. Fast Algorithms for Denial Constraint Discovery. *PVLDB* 16, 4 (2022), 684–696.

[41] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason A. Fries, Sen Wu, and Christopher Ré. 2020. Snorkel: rapid training data creation with weak supervision. *VLDBJ* 29, 2-3 (2020), 709–730.

[42] Alexander J. Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data Programming: Creating Large Training Sets, Quickly. In *NIPS*. 3567–3575.

[43] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* 10, 11 (2017), 1190–1201.

[44] Sudeepa Roy, Laura Chiticariu, Vitaly Feldman, Frederick R Reiss, and Huaiyu Zhu. 2013. Provenance-based dictionary refinement in information extraction. In *SIGMOD*. 457–468.

[45] Anastasiya Sedova and Benjamin Roth. 2022. ULF: Unsupervised Labeling Function Correction using Cross-Validation for Weak Supervision. *CoRR* abs/2204.06863 (2022).

[46] S. Song and L. Chen. 2009. Discovering matching dependencies. In *CIKM*. 1421–1424.

[47] Paroma Varma and Christopher Ré. 2018. Snuba: Automating Weak Supervision to Label Training Data. *PVLDB* 12, 3 (2018), 223–236.

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

[49] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data x-ray: A diagnostic tool for data errors. In *SIGMOD*. 1231–1245.

[50] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. 2017. QFix: Diagnosing errors through query histories. In *SIGMOD*. 1369–1384.

[51] Zihan Wang, Jingbo Shang, Liyuan Liu, Lihao Lu, Jiacheng Liu, and Jiawei Han. 2019. CrossWeigh: Training Named Entity Tagger from Imperfect Annotations. In *EMNLP-IJCNLP*. 5153–5162.

[52] Ziheng Wei, Sven Hartmann, and Sebastian Link. 2021. Algorithms for the discovery of embedded functional dependencies. *VLDBJ* 30, 6 (2021), 1069–1093.

[53] Renjie Xiao, Zijing Tan, Haojin Wang, and Shuai Ma. 2022. Fast Approximate Denial Constraint Discovery. *PVLDB* 16, 2 (2022), 269–281.

[54] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. 2020. A Statistical Perspective on Discovering Functional Dependencies in Noisy Data. In *SIGMOD*. 861–876.

**Algorithm 4:** Convert DC to Rule

**Input** : A DC $\varphi = \neg(P_1 \wedge \ldots, \wedge P_k)$
**Output**: A rule representation of $\varphi$, $r_\varphi$

$V, E = \emptyset, \emptyset$;
$r_\varphi \leftarrow (V, E)$;
**for** $i = 1, \ldots, k$ **do**
  $V \leftarrow V \cup \{P_i\}$;
  $E \leftarrow E \cup \{(P_{i-1}, P_i, True), (P_i, \emptyset, False)\}$;
$E \leftarrow E \cup \{(P_k, dirty, True)\}$;
**return** $r_\varphi$;

**Algorithm 5:** Convert LF to Rule

**Input** : An LF $f$
**Output**: A rule representation of $f$, $r_f$

$V, E = \emptyset, \emptyset$;
$r_f \leftarrow (V, E)$;
Let $D$ be the flow diagram of $f$ for every possible data flow;
**LF-to-Rule**$(r, D)$**:**
  Let $r(D)$ be $\varphi$;
  **if** $\varphi \notin \mathcal{Y}$ **then**
    Let $\varphi' = P_2 \circ \varphi'$, where $\circ \in \{\vee, \wedge\}$;
    $V \leftarrow V \cup \{P_1\}$;
    **if** $\varphi = P_1 \vee \varphi''$ **then**
      $E \leftarrow E \cup \{(P_1, P_2, False)\}$;
    **else if** $\varphi = P_1 \vee \varphi'$ **then**
      $E \leftarrow E \cup \{(P_1, P_2, True)\}$;
    LF-to-Rule$(r, D_{\varphi'})$;
  **else**
    **return** $r_f$;

# A TRANSLATING LABELING FUNCTIONS AND DENIAL CONSTRAINTS INTO RULES

In this section, we detail simple procedures for converting DCs and labeling functions to our rule representation (Def. 2.2).

For converting a DC into a rule, Algorithm 4 simply iterates over its predicates from left to right and add each one after another node in the tree, creating a right-deep tree, where the left leaf of each inner node is an empty set and the right one is another predicate.

For LFs, we assume that they can be written in a hierarchical manner such that each condition $\varphi_1$ leads to another condition $\varphi_2$, or a label $y \in \mathcal{Y}$. Therefore, such functions have a natural recursive structure that can be utilized for converting them into rule representations. Let us denote this flow diagram of an LF by $D$, where $D_\varphi$ is the sub-diagram rooted at the condition $\varphi$, not including $\varphi$ itself and denote the root of the diagram by $r(D)$. Algorithm 5 basically traverses the LF according to its flow diagram and, for each condition, splits it into literals and builds the rule by iterating over the literals in order. In both use cases, it's evident that the algorithm's running time is directly proportional to the size of the rule, which indicates a linear time complexity.

# B TRANSLATING REFINED RULES INTO DCS

Recall that a denial constraint is a formula $\forall \vec{x} : \neg\Psi(\vec{x})$ where $\Psi$ is a conjunction $\psi_1 \wedge \ldots \wedge \psi_n$ of relational and comparison atoms. That means, that the translation of a DC into a rule as explained in App. A is tree such that every predicate has a left child that is a leaf labeled CLEAN and a right child that is either another predicate or the single leaf node labeled DIRTY. Put differently, the DC is only violated if all $\psi_i$ evaluate to true on an assignment. Our refinement repair may generate rules from such a DC rule that have more than one leaf node labeled DIRTY. Intuitively, such rules encode disjunctive condition and, thus, in general cannot be expressed as a single DC. However, most data cleaning systems require DCs or even more restrictive classes of constraints such as functional dependencies (FDs). Fortunately, we can translate any refined rule into an equivalent set of denial constraints as follows. First observe that a rule corresponds to a disjunction over the paths $P_i = p_1 \xrightarrow{b_1} p_2 \xrightarrow{b_2} \ldots \xrightarrow{b_n}$ DIRTY, i.e., all paths that end up in a node labeled DIRTY. An assignment violates the constraint encoded by the rule iff it fulfills the following condition:

$$\bigvee_{P_i} \left( \bigwedge_{j \in \{1,\ldots,n\} \wedge b_j = \textbf{true}} \psi_i \wedge \bigwedge_{j \in \{1,\ldots,n\} \wedge b_j = \textbf{false}} \neg\psi_i \right)$$

Intuitively, this condition tests whether the assignments will end up in a leaf labeled DIRTY. This condition is equivalent to a set of DCs one for one of the conjunct of the formula. Thus, we can translate the rule into the following set of DCs $\{dc_i\}$ where:

$$dc_i = \forall \vec{x} : \neg : \left( \bigwedge_{j \in \{1,\ldots,n\} \wedge b_j = \textbf{true}} \psi_i \wedge \bigwedge_{j \in \{1,\ldots,n\} \wedge b_j = \textbf{false}} \neg\psi_i \right)$$

# C PROOF OF THEOREM 2.9

Proof of Thm. 2.9. [4] We prove this theorem by reduction from the NP-complete Set Cover problem. Recall the set cover problem is given a set $U = \{e_1, \ldots, e_n\}$ and subsets $S_1$ to $S_m$ such that $S_i \subseteq U$ for each $i$, does there exist a $i_1, \ldots, i_k$ such that $\bigcup_{j=1}^{k} S_{i_j} = U$. Based on an instance of the set cover problem we construct and instance of the rule repair problem as follows:

- Database: a single table $R(E)$ with single attribute $E$. We consider $n+1$ tuples (data points) $\{(e_1), \ldots, (e_n), (b)\}$ where $b \notin U$.
- Labels $\mathcal{Y} = \{out, in\}$
- Rules $\mathcal{R} = \{r\}$ where $r$ has a single predicate $p : (v = v)$ (i.e., it corresponds to truth value $true$) with two children that are leaves with labels $C_{\textbf{false}}(p) = in$ and $C_{\textbf{true}}(p) = out$ (Fig. 13). Hence initially any assignment will end up in $C_{\textbf{true}}(p) = out$.
- There are $n+1$ assignments, where $\lambda_e$ corresponds to tuple $e \in R$ (all $e_i$ and $b$) and let's denote the assignment $\lambda_e(v) = e$.

[4] We note that the proof sketch in the main body is outdated as it mentions a reduction from Max-3-SAT. We apologize for this oversight.

The $n+1$ ground truth labels for assignments and $n+1$ ground truth labels for data points provided as input are

$$C^{\Lambda}(\lambda_e) = \begin{cases} in & \text{if } e \neq b \\ out & \text{otherwise}(e = b) \end{cases}$$

and

$$C^{*}(e) = \begin{cases} in & \text{if } e \neq b \\ out & \text{otherwise}(e = b) \end{cases}$$

- Furthermore, the model $\mathcal{M}_{\mathcal{R}}$ is defined as $\mathcal{M}(x) = r(x)$ (the model returns the labels produced by the single rule $r$).
- We disallow deletions (i.e., set $\tau_{del} = \infty$).
- The space of predicates is $\mathcal{P} = \{v \in S_i \mid i \in [1, m]\}$
- The accuracy threshold $\tau_{acc} = 1$. In other words, we want the correct classification of all data points after repairing $r$, i.e., for all $e_i$, $i = 1 \cdots n$, the updated rule should output $in$, and for $b$, the updated rule should still output $out$.

We claim that there exists a set cover of size $k$ or less iff there exists a minimal repair of $\mathcal{R}$ with a cost of less than or equal to $k$.

(only if): Let $S_{i_1}, \ldots S_{i_k}$ be a set cover. We have to show that there exists a minimal repair $\Phi$ of size $\leq k$. We construct that repair as follows: we replace the **true** child of the single predicate $p : (v = v)$ in $r$ with a left deep tree with predicates $p_{i_j}$ for $j \in [1, k]$ such that $p_{i_j}$ is $(v \in S_{i_j})$ and $C_{\textbf{false}}(p_{i_j}) = p_{i_{j+1}}$ unless $j = k$ in which case $C_{\textbf{false}}(p_{i_j}) = out$; for all $j$, $C_{\textbf{true}}(p_{i_j}) = in$. A graph reprentation of this is shown in Fig. 14 The repair sequence $\Phi = \phi_1, \ldots, \phi_k$ where $\phi_i$ introduces $p_{i_j}$ has cost $k$.

It remains to be shown that $\Phi$ is a valid repair with accuracy 1. Let $r_{up} = \Phi(r)$, we have $r_{up}(\lambda_e) = C^{\Lambda}(\lambda_e)$ for all $(e) \in R$. Recall that $\mathcal{M}(e) = r(\lambda_e)$ and, thus, $r_{up}(\lambda_e) = C^{\Lambda}(\lambda_e)$ ensures that $\mathcal{M}(e) = C^{*}(e)$. Since the model corresponds a single rule, we want $r_{up}(e_i) = in$ for all $i = 1, \cdots, n$, and $r_{up}(b) = out$. Note that the final label is $out$ only if the check $(v \in S_{i_j})$ is false for all $j = 1, \cdots, k$. Since $S_{i_1}, \ldots S_{i_k}$ constitute a set cover, for every $e_i$, $i = 1, \cdots, n$, the check will be true for at least one $S_{i_j}$, resulting in a final label of $in$. On the other hand, the check will be false for all $S_{i_j}$ for $b$, and therefore the final label will be $out$. This gives a refined rule with accuracy 1.

(if):Let $\Phi$ be a minimal repair of size $\leq k$ giving accuracy 1. Let $r_{up} = \Phi(r)$, i.e., in the repaired rule $r_{up}$, all $e_i$, $i = 1, \cdots, n$ get the $in$ label and $b$ gets $out$ label. We have to show that there exists a set cover of size $\leq k$. First, consider the path taken by element $b$. For any predicate $p \in \mathcal{P}$ of the form $(v \in S_i)$, we have $p(b) = \textbf{false}$. Let us consider the path $P = v_{root} \xrightarrow{b_1} v_1 \xrightarrow{b_2} v_2 \ldots v_{l-1} \xrightarrow{b_l} v_l$ taken by $\lambda_b$.

As $v_{root} = p_{root} = true$, we know that $b_1 = \textbf{true}$ ($\lambda_b$ takes the **true** edge of $v_{root}$. Furthermore, as $p(b)$ for all predicates in $\mathcal{P}$ (as $b \notin U$), $\lambda_b$ follows the **false** edge for all remaining predicates on the path. That is $b_i = \textbf{false}$ for $i > 1$. As we have $C^{\Lambda}(\lambda_b) = out$ and $\Phi$ is a repair, we know that $node_l = out$. For each element $e \in U$, we know that $r_{up}(\lambda_e) = in$ which implies that the path for $\lambda_e$ contains at least one predicate $v \in S_i$ for which $e \in S_i$ evaluates to true. To see why this has to be the case note that otherwise $\lambda_e$ would take the same path as $\lambda_b$ and we have $r_{up}(\lambda_e) = out \neq in = C^{\Lambda}(\lambda_e)$ contradicting the fact that $\Phi$ is a repair. That is, for each $e \in U$ there exists $S_i$ such $e \in S_i$ and $p_i : v \in S_i$ appears in the tree of $r_{up}$. Thus, $\{S_i \mid p_i \in r_{up}\}$ is a set cover of size $\leq k$. □

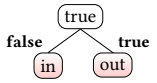



**Figure 13: The rule representation of rule r used in App. C**

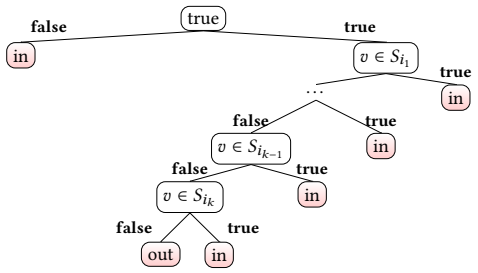



**Figure 14: The rule representation of repair of rule r in App. C**

# D SINGLE RULE REFINEMENT - PROOFS AND ADDITIONAL DETAILS

## D.1 Proof of Prop. 4.2

Proof of Prop. 4.2. The claim can be shown by contradiction. Assume that there exist a repair $\Phi = \Phi_1, \ldots, \Phi_k$, but $\Phi$ is not optimal. That is, there exists a repair $\Phi'$ with a lower cost. First off, it is easy to show that $\Phi'$ does not refine any paths $p \notin P_{fix}$ as based on our observation presented above any such refinement does not affect the label of any assignment in $\Lambda_{C^{\Lambda}}$ and, thus, can be removed from $\Phi'$ yielding a repair of lower costs which contradicts the fact that $\Phi'$ is optimal. However, then we can partition $\Phi'$ into refinements $\Phi'_i$ for each $P_i \in P_{fix}$ such that $\Phi' = \Phi'_1, \ldots, \Phi'_k$. As $cost(\Phi') < cost(\Phi)$ there has to exist at least on path $P_i$ such that $cost(\Phi'_i) < cost(\Phi_i)$ which contradicts the assumption that $\Phi_i$ is optimal for all $i$. Hence, no such repair $\Phi'$ can exist. □

## D.2 Partitioning Predicate Spaces

Lemma D.1. *Consider a space of predicates $\mathcal{P}$ and atomic units $\mathcal{A}$.*

- ***Atomic unit comparisons**: If $\mathcal{P}$ contains for every $a \in \mathcal{A}$, constant $c$, and variable $v$, the predicate $v[a] = c$, then $\mathcal{P}$ is* partitioning.
- ***Labeling Functions**: Consider the labeling function usecase. If $\mathcal{P}$ contains predicate $w \in v$ for any variable $v$ and word $w$, then $\mathcal{P}$ is partitioning.*
- ***Denial Constraints**: If $\mathcal{P}$ contains comparisons between attribute values and constants, then $\mathcal{P}$ is partitioning. However, if $\mathcal{P}$ does only contain comparisons between attribute values and attribute values, then in general it is not partitioning.*

Proof. **Atomic unit comparisons.** Consider an arbitrary pair of assignments $\lambda_1 \neq \lambda_2$ for some rule $r$ over $\mathcal{P}$. Since, $\lambda_1 \neq \lambda_2$ it follows that there exists $v \in \text{VARS}(r)$ such that $\lambda_1(v) = x_1 \neq x_2 = \lambda_2(v)$. Because $x_1 \neq x_2$, there has to exist $a \in \mathcal{A}$ such that

$x_1[a] = c \neq x_2[a]$. Consider the predicate $p = (v[a] = c)$ which based on our assumption is in $\mathcal{P}$.

$$\lambda_1(v[a] = c) = (x_1[a] = c) = \textbf{true}$$
$$\lambda_2(v[a] = c) = (x_2[a] = c) = \textbf{false}$$

Note that we did not restrict the choice of $\lambda_1$ and $\lambda_2$. Thus, given that for any choice of assignment we can find a predicate that holds on $\lambda_1$ and does not hold on $\lambda_2$, it follows that $\mathcal{P}$ is partitioning.

**Labeling Functions.** Recall that the atomic units for the semi-supervised labeling usecase are words in a sentence. Thus, the claim follows from the atomic unit comparisons claim proven above.

**Denial Constraints.** Analog to the case for labeling functions, if comparisons with constants are allowed, then $\mathcal{P}$ is partitioning based on the proof of the general version of this claim shown above. To see why just comparison between attribute values is not enough consider the following two single variable assignments for tuples with schema $(A, B)$:

$$\lambda_1(v) = x_1 = (1, 1)$$
$$\lambda_2(v) = x_2 = (2, 2)$$

It is easy to verify that any comparison between any attributes evaluates to the same for both $\lambda_1$ and $\lambda_2$. Thus, if $\mathcal{P}$ does only contain comparisons between attributes, it is not partitioning. □

## D.3 Proof of Prop. 4.3

Proof of Prop. 4.3. We will use $y_\lambda$ to denote the expected label for $\lambda$, i.e., $y_\lambda = C(\lambda)$. Consider the following recursive greedy algorithm that assigns to each $\lambda \in \Lambda$ the correct label. The algorithm starts with $\Lambda_{cur} = \Lambda$ and in each step finds a predicate $p$ that "separates" two assignments $\lambda_1$ and $\lambda_2$ from $\Lambda_{cur}$ with $y_{\lambda_1} \neq y_{\lambda_2}$. That is, $p(\lambda_1)$ is true and $p(\lambda_2)$ is false. As $\mathcal{P}$ is partitioning such a predicate has to exist. Let $\Lambda_1 = \{\lambda \mid \lambda \in \Lambda_{cur} \wedge p(\lambda)\}$ and $\Lambda_2 = \{\lambda \mid \lambda \in \Lambda_{cur} \wedge \neg p(\lambda)\}$. We know that $\lambda_1 \in \Lambda_1$ and $\lambda_2 \in \Lambda_2$. That means that $|\Lambda_1| < |\Lambda|$ and $|\Lambda_2| < |\Lambda|$. The algorithm repeats this process for $\Lambda_{cur} = \Lambda_1$ and $\Lambda_{cur} = \Lambda_2$ until all assignments in $\Lambda_{cur}$ have the same desired label which is guaranteed to be the case if $|\Lambda_{cur}| = 1$. In this case, the leaf node for the current branch is assigned this label. As for each new predicate added by the algorithm the size of $\Lambda_{cur}$ is reduced by at least one, the algorithm will terminate after adding at most $|\Lambda|$ predicates. □

## D.4 Non-minimality of the Algorithm from Prop. 4.3

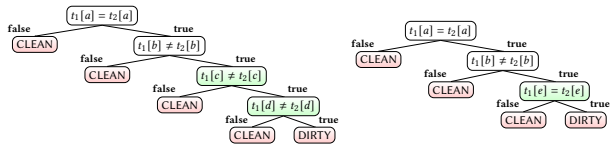



**Figure 15: A non-optimal rule repair for the DC from Ex. D.2 produced by the algorithm from Prop. 4.3 and an optimal repair (right)**

Example D.2. *Consider a DC* $\neg(t_1[a] = t_2[a] \wedge t_1[b] \neq t_2[b])$*, i.e., a functional dependency* $a \rightarrow b$ *and the example table shown below. There are 4 violations to this DC in the example table:* $\lambda_1 = (x_1, x_3)$*,* $\lambda_2 = (x_1, x_4)$*,* $\lambda_3 = (x_2, x_3)$*, and* $\lambda_4 = (x_2, x_4)$ *(For conciseness we write* $\lambda = (x_i, x_j)$ *instead of* $\lambda(t_1) = x_i$ *and* $\lambda(t_2) = x_j$*). Let us assume that the user has marked the assignments for the first two violations as* CLEAN *and the remaining two as* DIRTY*. The algorithm outlined above may first pick* $\lambda_1$ *and* $\lambda_3$ *and select predicate* $t_1[c] \neq t_2[c]$ *to distinguish between these assignments. However,* $\lambda_2$ *fulfills this predicate and thus in the next step, the algorithm may select* $\lambda_2$ *and* $\lambda_4$ *and predicate* $t_1[d] = t_2[d]$ *leading to "pure" sets of assignments wrt. to their expected labels at each node (a valid repair). The resulting repair is shown in Fig. 15 (left). Note that this repair has a cost of 2 (2 new predicate nodes are introduced). However, there exists a smaller rule repair the requires only one additional predicate* $t_1[e] = t_2[e]$ *(Fig. 15, right). That is, for this example, the algorithm has found a repair with a cost of less than* $|\Lambda| = 4$*, but as demonstrated above there exists an optimal repair for this example that has a cost of* 1.

| | A | B | C | D |
|---|---|---|---|---|
| $x_1 \rightarrow$ | 1 | 2 | 1 | 2 |
| $x_2 \rightarrow$ | 1 | 2 | 3 | 1 |
| $x_3 \rightarrow$ | 1 | 3 | 2 | 1 |
| $x_4 \rightarrow$ | 1 | 3 | 1 | 1 |

## D.5 Equivalence of Predicates on Assignments

Our results stated above only guarantee that repairs with a bounded cost exist. However, the space of predicates may be quite large or even infinite. We now explore how equivalences of predicates wrt. $\Lambda$ can be exploited to reduce the search space of predicates and present an algorithm that is exponential in $|\Lambda|$ which determines an optimal repair independent of the size of the space of all possible predicates. First observe that with $|\Lambda| = n$, there are exactly $2^n$ possible outcomes of applying a predicate $p$ to $\Lambda$ (returning either true or false for each $\lambda \in \Lambda$). Thus, with respect to the task of repairing a rule through refinement to return the correct label for each $\lambda \in \Lambda$, two predicates are equivalent if they return the same result on $\Lambda$ in the sense that in any repair using a predicate $p_1$, we can substitute $p_1$ for a predicate $p_2$ with the same outcome and get a repair with the same cost. That implies that when searching for optimal repairs it is sufficient to consider one predicate from each equivalence class of predicates.

Lemma D.3 (Equivalence of Predicates). *Consider a space of predicates* $\mathcal{P}$*, rule* $r$*, and set of assignments* $\Lambda$ *with associated expected labels and assume the existence of an algorithm* $\mathcal{A}$ *that computes an optimal repair for* $r$ *given a space of predicates. Two predicates* $p \neq p' \in \mathcal{P}$ *are considered equivalent wrt.* $\Lambda$*, written as* $p \equiv_\Lambda p'$ *if* $p(\Lambda) = p'(\Lambda)$*. Furthermore, consider a reduced space of predicates* $\mathcal{P}_\equiv$ *that fulfills the following condition:*

$$\forall p \in \mathcal{P} : \exists p' \in \mathcal{P}_\equiv : p \equiv p' \tag{1}$$

*For any such* $\mathcal{P}_\equiv$ *we have:*

$$cost(\mathcal{A}(\mathcal{P}, r, \Lambda)) = cost(\mathcal{A}(\mathcal{P}_\equiv, r, \Lambda))$$

Proof. Let $\Phi = \mathcal{A}(\mathcal{P}_\equiv, r, \Lambda)$ and $\Phi_\equiv = \mathcal{A}(\mathcal{P}, r, \Lambda))$. Based on the assumption about $\mathcal{A}$, $\Phi$ ($\Phi_\equiv$) are optimal repairs within $\mathcal{P}$ ($\mathcal{P}_\equiv$). We prove the lemma by contradiction. Assume that $cost(\Phi_\equiv) > cost(\Phi)$.

We will construct from $\Phi$ a repair $\Phi'$ with same cost as $\Phi$ which only uses predicates from $\mathcal{P}_{\equiv}$. This repair then has cost $cost(\Phi') = cost(\Phi) < cost(\Phi_{\equiv})$ contradicting the fact that $\Phi_{\equiv}$ is optimal among repairs from $\mathcal{P}_{\equiv}$. $\Phi'$ is constructed by replacing each predicate $p \in \mathcal{P}$ used in the repair with an equivalent predicate from $\mathcal{P}_{\equiv}$. Note that such a predicate has to exist based on the requirement in Eq. (1). As equivalent predicates produce the same result on every $\lambda \in \Lambda$, $\Phi'$ is indeed a repair. Furthermore, substituting predicates does not change the cost of the repair and, thus, $cost(\Phi') = cost(\Phi)$. □

If the semantics of the predicates in $\mathcal{P}$ is known, then we can further reduce the search space for predicates by exploiting these semantics and efficiently determine a viable $p_{\equiv}$. For instance, predicates of the form $A = c$ for a given atomic element $A$ only have linearly many outcomes on $\Lambda$ and the set of $\{v.A = c\}$ for all atomic units $A$, variables in $\mathcal{R}$, and constants $c$ that appear in at least one datapoint $x \in \mathcal{X}$ contains one representative of each equivalence class.[5]

# E PATH REFINEMENT REPAIRS - PROOFS AND ADDITIONAL DETAILS

## E.1 GreedyPathRepair

Function GreedyPathRepair is shown Algorithm 6. To ensure that all assignments ending in path $P$ get assigned the desired label based on $C$, we need to add predicates to the end of $P$ to "reroute" each assignment to a leaf node with the desired label. As mentioned above this algorithm implements the approach from the proof of Prop. 4.3: for a set of assignments taking a path with prefix $P$ ending in a leaf node that is not pure (not all assignments in the set have the same expected label), we pick a predicate that *"separates"* the assignments, i.e., that evaluate to true on one of the assignments and false on the other. Our algorithm applies this step until all leaf nodes are pure wrt. the assignments from $\Lambda_C$. For that, we maintain a queue of path and assignment set pairs which tracks which combination of paths and assignment sets still have to be fixed. This queue is initialized with $P$ and all assignments for $P$ from $\Lambda_C$. The algorithm processes sets of assignments until the todo queue is empty. In each iteration, the algorithm greedily selects a pair of assignments $\lambda_1$ and $\lambda_2$ ending in this path that should be assigned different labels (line 5). It then calls method `GetSeperatorPred` (line 7) to determine a predicate $p$ which evaluates to true on $\lambda_1$ and false on $\lambda_2$ (or vice versa). If we extend path $P$ with $p$, then $\lambda_1$ will follow the **true** edge of $p$ and $\lambda_2$ will follow the **false** edge (or vice versa). This effectively partitions the set of assignments for path $P$ into two sets $\Lambda_1$ and $\Lambda_2$ where $\Lambda_1$ contains $\lambda_1$ and $\Lambda_2$ contains $\lambda_2$. We then have to continue to refine the paths ending in the two children of $p$ wrt. these sets of assignments. This is ensured by adding these sets of assignments with their new paths to the todo queue (lines 12 and 13). If the current set of assignments does not contain two assignments with different labels, then we know that all remaining assignments should receive the same label. The algorithm picks one of these assignments $\lambda$ (line 14) and changes the current leave node's label to $C(\lambda)$.

[5]With the exception of the class of predicates that return false on all $\lambda \in \Lambda$. However, this class of predicates will never be part of an optimal repair as it does only trivially partitions $\Lambda$ into two sets $\Lambda$ and $\emptyset$.

**Algorithm 6:** GreedyPathRepair

**Input :** Rule $r$
Path $P$
Assignments to fix $\Lambda$
Expected labels for assignments $C$
**Output :** Repair sequence $\Phi$ which fixes $r$ wrt. $\Lambda$

```
1  todo ← [(P, Λ)]
2  Φ = []
3  while todo ≠ ∅ do
4      (P, Λ) ← pop(todo)
5      if ∃λ1, λ2 ∈ Λ : C[λ1] ≠ C[λ2] then
6          /* Determine predicates that distinguishes
               assignments that should receive different
               labels for a path                          */
7          p ← GetSeperatorPred(λ1, λ2)
8          y1 ← C[λ1]
9          φ ← refine(r_cur, P, y1, p, true)
10         Λ1 ← {λ | λ ∈ Λ ∧ λ(p)}
11         Λ2 ← {λ | λ ∈ Λ ∧ ¬λ(p)}
12         todo.push((P[r_cur, λ1], Λ1))
13         todo.push((P[r_cur, λ2], Λ2))
14     else
15         λ ← Λ.pop()      /* All λ ∈ Λ have same label */
16         φ ← refine(r_cur, P, C[λ])
17     r_cur ← φ(r_cur)
18     Φ.append(φ)
19 return Φ
```

**Generating Predicates.** The implementation of `GetCoveringPred` is specific to the type of RBBM. We next present implementations of this procedure for semi-supervised labeling and constraint-based data cleaning that exploit the properties of these two application domains. However, note that, as we have shown in Sec. 4.2, as long as the space of predicates for an application domain contains equality and inequality comparisons for the atomic elements of data points, it is always possible to generate a predicate for two assignments such that only one of these two assignments fulfills the predicate. The algorithm splits the assignment set $\Lambda$ processed in the current iteration into two subsets which each are strictly smaller than $\Lambda$. Thus, the algorithm is guaranteed to terminate and by construction assigns each assignments $\lambda$ in $C$ its desired label $C(\lambda)$.

## E.2 Proof of Thm. 4.4

Proof. Proof of Thm. 4.4 In the following let $n = |\Lambda_C|$.

GreedyPathRepair: As GreedyPathRepair does implement the algorithm from the proof of Prop. 4.3, it is guaranteed to terminate after at most $n$ steps and produce a repair that assign to each $\lambda$ the label $C(\lambda)$.

BruteForcePathRepair: The algorithm generates all possible trees build from predicates and leaf nodes in increasing order of their side. It terminates once a tree has been found that returns the correct

**Algorithm 7:** BruteForcePathRepair

**Input :** Rule $r$
Path $P$
Assignments to fix $\Lambda$
Expected labels for assignments $C$

**Output :** Repair sequence $\Phi$ which fixes $r$ wrt. $\Lambda$

```
todo ← [(r, ∅)]
P_all = GetAllCandPredicates(P, Λ, C)
while todo ≠ ∅ do
    (r_cur, Φ_cur) ← pop(todo)
    foreach P_cur ∈ leafpaths(r_cur, P) do
        foreach p ∈ P_all − P_{r_cur} do
            foreach y_1 ∈ 𝒴 ∧ y_1 ≠ last(P_cur) do
                φ_new ← refine(r_cur, P_cur, y_1, p, true)
                r_new ← φ_new(r_cur)
                Φ_new ← Φ_cur, φ_cur
                if Accuracy(r_new, C) = 1 then
                    return Φ_new
                else
                    todo.push((r_new, Φ_new)
```

labels on $\Lambda_C$. As there has to exist a repair of size $n$ or less, the algorithm will eventually terminate.

EntropyPathRepair: This algorithm greedily selects a predicate in each iteration that minimizes the Gini impurity score. The algorithm terminates when for every leaf node, the set of assignments from $\Lambda_C$ ending in this node has a unique label. That is, if the algorithm terminates, it returns a solution. It remains to be shown that the algorithm terminates for every possible input. As it is always possible to find a separator predicate $p$ that splits a set of assignments $\Lambda$ into two subsets $\Lambda_1$ and $\Lambda_2$ with less predicates which has a lower Gini impurity score than splitting into $\Lambda_1 = \Lambda$ and $\Lambda_2 = \emptyset$, the size of the assignments that are being precessed, strictly decrease in each step. Thus, the algorithm will in worst-case terminate after adding $n$ predicates. □

# F ADDITIONAL EXPERIMENTS

## F.1 Varying Complaint Ratio

We now focus on evaluating the effects of varying complaint ratio, $\tau_{del}$, and input size on the results of *Entropy* and *Greedy* using *YTSpam*. The results are shown in Fig. 16 to 18. The general trend we observed in Fig. 6f still holds for these experiments: all three metrics (fix rate, preserv. rate, and new global acc.) increase if we increase the size the input size. For fix rate, *Greedy* slightly outperforms *Entropy*, and both algorithms have higher fix rate when the complaint ratio approaches 90%. However, when the complaint ratio increases, the preserv. rate decreases approaching 0% when the complaint ratio approaches 90%. Based on these results we recommend setting complaint ratio to 50% for best overall accuracy. For the global accuracy of the retrained model after the fix, we could see that the high deletion factor is affecting the global accuracy in a negative way, which indicates that aggressively deleting the rules is not good for improving the model performance. As shown in Fig. 18. when $\tau_{del}$ is set low, *Entropy* outperforms *Greedy*.

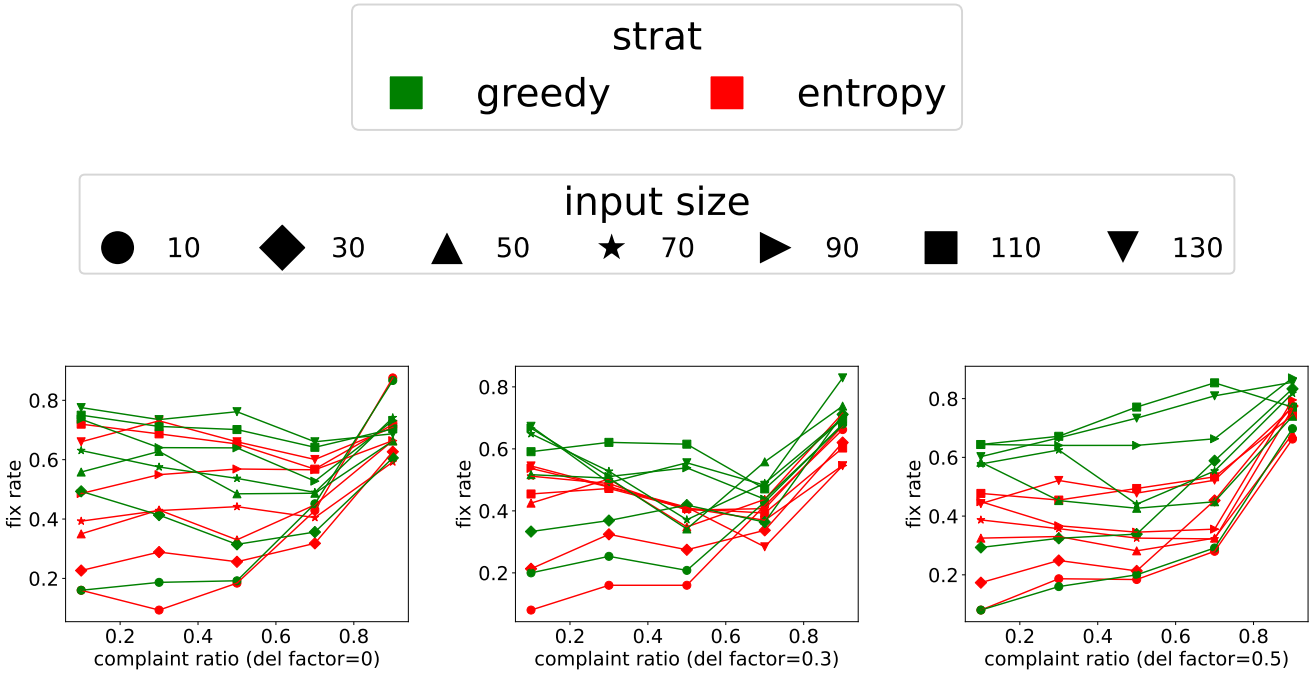



**Figure 16: complaint ratio vs fix rate**

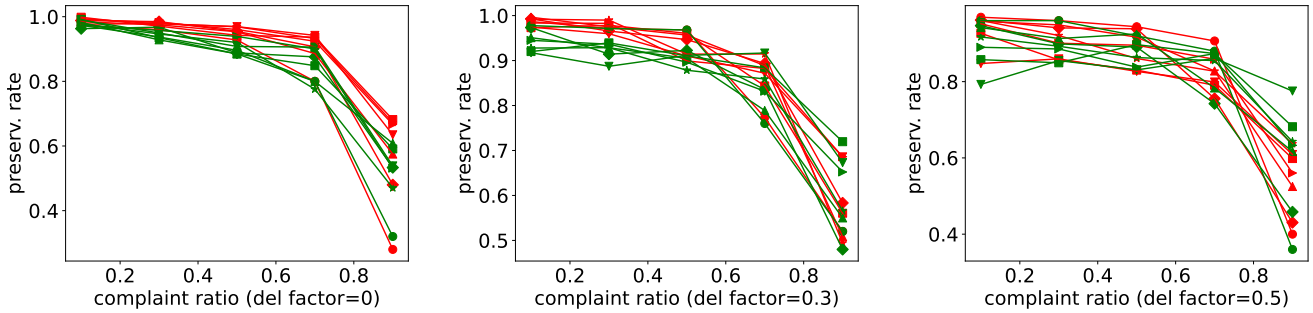



**Figure 17: complaint ratio vs preserv. rate**

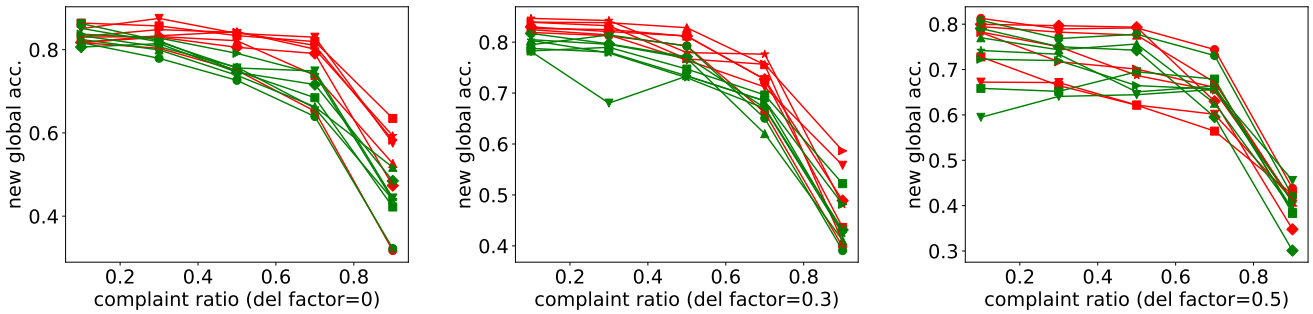



**Figure 18: complaint ratio vs new global accuracy**