Repairing Labeling Functions Based on User Feedback

Chenjie Li
Illinois Institute of Technology
cli112@hawk.iit.edu

Amir Gilad
Hebrew University
amirg@cs.huji.ac.il

Boris Glavic
University of Illinois, Chicago
bglavic@uic.edu

Zhengjie Miao
Simon Fraser University
zhengjie@sfu.ca

Sudeepa Roy
Duke University
sudeepa@cs.duke.edu

ABSTRACT

Data programming systems like Snorkel generate large amounts of training data by combining the outputs of a set of user-provided labeling functions (LFs) on unlabeled instances with a blackbox ML model. This significantly reduces human effort for labeling data compared to manual labeling. The efficiency of the labeling can be further improved through approaches like Witan that automate (parts of) the LF generation process. However, no matter whether LFs are created with or without the help of such tools, the quality of the generated training data depends directly on the accuracy of the set of LFs. In this work, we study the problem of fixing LFs based on user feedback on the correctness of labels for a set of example instances. Our approach complements manual and automatic generation of LFs by improving a given set of LFs. Towards this goal, we develop novel techniques for repairing a set of LFs such that the fixed LFs will return correct labels on the instances on which the user has provided feedback. Typically, LFs are implemented using general purposes programming languages like Python. We model such LFs as conditional rules which enables us to refine them, i.e., to selectively change their output for some inputs. We demonstrate experimentally that our system improves the quality of both manually and automatically generated LFs based on a moderate amount of user feedback.

Boris says: Sudeepa had a comment there?

Furthermore, we evaluate how much feedback is required for improving LFs and generates refined LFs that are intuitive.

Boris says: Better way to phrase this?

PVLDB Reference Format:
Chenjie Li, Amir Gilad, Boris Glavic, Zhengjie Miao, and Sudeepa Roy. Repairing Labeling Functions Based on User Feedback. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

Data programming [40] is weak supervision approach for creating training data that combines evidence about a training instance's label provided by multiple heuristics through a black box machine learning (ML) model. In contrast to manual labeling where a user has to assign a label to each training instance, labels are created based on heuristics implemented as so-called labeling functions (LFs) that take an instance as input and output a label. The main advantage of data programming is that it reduces human effort for training data creation. This effort can further be reduced through techniques that automate (parts of) the label function generation process [7, 14, 21, 46]. For instance, Witan [14] automatically constructs labeling functions from simple predicates that are effective in distinguishing training instances and suggests these functions to a user to select sensible LFs and determine what labels they should return. Another recent approach is the work of Guan et. al. [21] that uses large language models (LLMs) to derive labeling functions from a small set of Labelled training data.

No matter whether LFs are created by a domain expert by hand or through one of the aforementioned automated approaches, it is hard for a user to debug and repair a set of LFs to fix issues with the created training data. This is due to several reasons: (i) the result of LFs is combined using a blackbox model which obfuscates which LFs are responsible for a mislabeling of a training data point; (ii) the training dataset may be large which makes it hard for a user to identify all problems caused by a broken labeling function; (iii) the semantics of automatically generated LFs may be unclear to a domain expert making it hard to figure out why such a function returns incorrect labels.

In this work, we study the problem of automatically fixing a set of **existing** LFs based on **user feedback on the labels** of a small subset of the training data. We consider two types of repairs: (i) refining a LF by locally overriding its result to match the user's expectation about a training data label and (ii) to delete LF that are deemed counterproductive by our approach. Rather than replacing a human domain expert or automated LF generation tool, our approach called RuleCleaner **complements** both approaches by improving an existing set of LFs. Our approach is general in that only makes minimal assumptions about the LFs and the behavior of the black box model that combines the output of the labeling functions to assign a final label to each training data point. In particular, labeling functions can be arbitrary functions implemented in a general purpose programming language such as Python and model can be used to combine the outputs of LFs. Thus, we support Snorkel [40] as well as simpler models, e.g., a majority vote of labeling functions. Furthermore, we are agnostic to how LFs have been created which enables us to repair LFs created by systems like Witan [14] or through large language models [21] as well as functions that were created by human domain experts.

Labeling functions as rules. We address the issue of LFs expressed in a general purpose programming language by modeling LFs as *rules* which are trees where inner nodes represent *predicates*, i.e., Boolean conditions evaluated on a data points and leaf nodes

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

representing labels. Intuitively, such rules encode a cascading set of rules: given a data point, we start at the root and repeatedly navigate to either the *false* or *true* child of the node based on whether the node's predicate evaluates to true. This process is repeated until a leaf node is reached. The label of the leaf node is then the label assigned by the rule to the input data point. While this model is simple, it allows us to encode arbitrary labeling functions as rules. In the worst-case we generate a single predicate for each possible label that compares the function's output against that label. The main reason for modeling labeling functions as rules is that it allows us to *refine* arbitrary LFs by extending their rule tree as follows: we replace one of the leaf nodes with a new predicate. Thus, the refined rule checks an additional condition to decide what label to return.

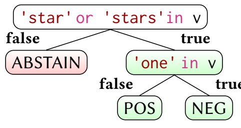
We present an algorithm that takes as input a labeling function, a space of allowable predicates (which does not necessarily include all predicates used in the input set of LFs) and a set of example data points with expected labels and refines the LF to returns the correct label on the example data points. We demonstrate that under reasonable conditions on the space of predicates, such a repair of the labeling function is guaranteed to exist. However, minimizing the number of predicates that have to be added to the LF to fix it is NP-hard. The rationale for minimizing changes to the original labeling function is that this better preserves the domain knowledge encoded in the function and also leads to more interpretable functions. In addition to a brute force optimal algorithm, we introduce an entropy-based heuristic which greedily selects predicates that best partition example data points based on their labels.

Dealing with black-box models. Another challenge of repairing labeling functions is that fixing the output of the LFs does not guarantee that the decision of black box model will change. For simple models like majority vote, ensuring that the labeling functions return the correct labels or abstain for all example data points is sufficient for the model to return the correct label for each example data point labeled by the user. However, for more complex models like the one employed by Snorkel this is not guaranteed to be true. We, therefore, make the reasonable assumption that the model is more likely to return the correct label on a sample data point as the number of labeling functions that return that label for this data point increases. We present an approach that periodically retrains the black box model and validates whether the model's accuracy on the user provided data examples is above a threshold. While simple, we demonstrate that this approach works well in practice.

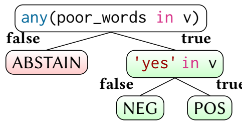
Example 1.1. **(Weakly supervised labeling with Snorkel [41])** *Consider the Amazon Review Dataset from [14, 24] which contains reviews for products bought from Amazon and the task of labeling the reviews as* POS *(positive) or* NEG *(negative). A subset of labeling functions* (LFs) *generated by the Witan system [14] for this task are shown in Fig. 1a. 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 *[41] are shown in Tab. 1. Tab. 1 also shows the results of three LFs including the ones from Fig. 1a. 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 labeling functions (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.*

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

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

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

(a) Three example labeling functions for amazon reviews



(b) Refined rule #1



(c) Refined rule #3

Figure 1: Example LFs before and after refinement by RuleCleaner

Suppose a human annotator labels the subset of the reviews shown in Tab. 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 for a subset of the generated training data to generate a repair Φ for the LFs $\mathcal{R}$ by deleting or refining LFs to ensure they align with the ground truth. Tab. 1 also shows the labels produced by the repaired LFs $\Phi(\mathcal{R})$ (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. 1a by adding new predicates (refinement). Fig. 1b and 1c 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.*

As shown in the example above, our approach is able to generate refined rules that are semantically meaningful such as realizing that mentioning "stars" in a review is only indicative of a negative review if the number of stars mentioned is low. As we demonstrate in our experimental evaluation, this behavior is not limited to this specific dataset. For instance, for classifying individuals as professors or teachers based on a brief description of the individual, RuleCleaner identified the presence of the keyword "research" as a distinguishing factor.

Boris says: Either add an explanation of the overview figure to the intro or at the beginning of section 2

Our Contributions

We make the following contributions in this work:

- **A general model for rule-based systems**. We model weak supervision labeling systems like Snorkel as *rule-based black models* (*RBBM*), i.e., systems that combine the output of a set of

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)

Table 1: Amazon products reviews with ground truth labels ("P"ositive or "N" egative), predicted labels by Snorkel [40] (before and after rule refinement), and the results of the labeling functions from Fig. 1 ("-" means ABSTAIN). Updated results for the repaired rules are highlighted in blue in parenthesis).

interpretable labeling functions (the rules) to predict labels for a set of data points $\mathcal{X}$. This framework is quite general as we support arbitrary labeling functions as input and do not place any restrictions on the blackbox model that generates the final labels. We further study the problem of repairing the LFs $\mathcal{R}$ of a RBBM based on user-provided ground truth labels for a small set of examples while minimizing the changes to $\mathcal{R}$, determining that the minimum repair is NP-hard.

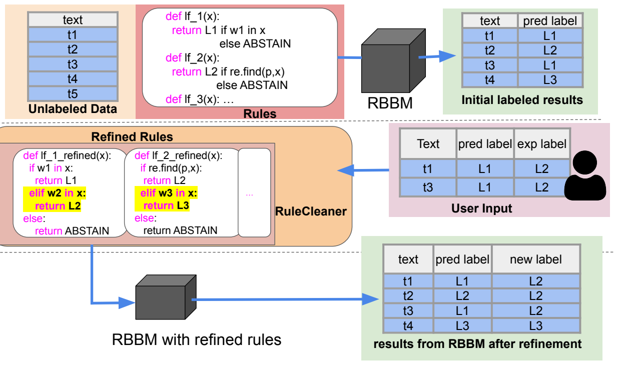
- **Algorithms for rule repair**. As the rule repair problem is NP-hard, we develop a PTIME heuristic algorithm that refines or deletes rules. A core component of this algorithm is a technique for refining rules by introducing new predicates such that the rule returns ground truth labels. We introduce (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 data points with different expected labels.
- **Experimental study**. We evaluate our approach weakly supervised labeling in *Snorkel* [40]. 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 LFs generated automatically using Witan [14] and using the LLM-based approach from [21]. Our approach can significantly improve the quality of such automated rules: as mentioned before, the repairs shown in Ex. 1.1 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 weakly supervised labeling systems [14, 35, 40], 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

Amir says: Replace 'rules' and 'data points' with 'labeling functions' and 'sentences'?



text
t1
t2
t3
t4
t5

```
def lf_1(x):
  return L1 if w1 in x
           else ABSTAIN
def lf_2(x):
  return L2 if re.find(p,x)
           else ABSTAIN
def lf_3(x): …
```

text	pred label
t1	L1
t2	L2
t3	L1
t4	L3

```
def lf_1_refined(x):
  if w1 in x:
    return L1
  elif w2 in x:
    return L2
  else:
    return ABSTAIN
```

```
def lf_2_refined(x):
  if re.find(p,x):
    return L2
  elif w3 in x:
    return L3
  else:
    return ABSTAIN
```

Text	pred label	exp label
t1	L1	L2
t3	L1	L2

text	pred label	new label
t1	L1	L2
t2	L2	L2
t3	L1	L2
t4	L3	L3

Figure 2: The RuleCleaner framework for repairing RBBMs. After running RBBM with rules, the user would identify a subset from initial labeled results, RuleCleaner refines rules based on the user input. Finally, RBBM is retrained with refined rules and produces new predictions on the data

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. 2.5) 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, if Snorkel [40] (the RBBM) is used to generates labels for documents (the training data), $\mathcal{X}$ is the set of document (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 $\mathcal{Y} = \{\mathsf{POS}, \mathsf{NEG}, \mathsf{ABSTAIN}\}$. where $y_0 = \mathsf{ABSTAIN}$. As another example consider labeling tabular data. In this case each tuple is a data point and the atomic units are the tuple's attribute values.

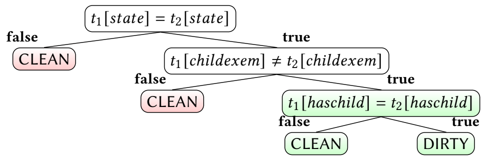
Example 2.1. *Tab. 1 shows a set of reviews (data points) with possible labels of* POS *or* P *(a review with positive sentiment),* NEG *or* N *(a negative review), and* ABSTAIN *or - (the abstain label). The atomic units of the first review with id 0 are all the words that it contains, e.g., "five", "stars", etc. (b) Fig. 3a shows a sample of a Tax dataset with 10 tuples. Here the task is to detect errors in individual tuples using a set of manually curated LFs. The label of each tuple is*

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)

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

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
```

(b) Labeling functions for error detection


$t_1[state] = t_2[state]$
false
true
CLEAN
$t_1[childexem] \neq t_2[childexem]$
false
true
CLEAN
$t_1[haschild] = t_2[haschild]$
false
true
CLEAN
DIRTY

Boris says: Fix this example if we decide to keep it repurposed for labeling individual tuples are dirty or clean based on within tuple problems

Figure 3: Error detection on the Tax dataset [8]

either CLEAN *(the abstain label) 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., [14, 21]). A rule consists of one or more predicates $\mathcal{P}_r$ from a set of domain-specific atomic predicates $\mathcal{P}$ over a single variable v that represents the data point over which the rule is evaluated. These predicates are allowed to access the atomic units of the datapoint. The atomic predicates $\mathcal{P}$ may contain simple predicates such as comparisons ($=, \neq, >, \geq, <, \leq$) as well as arbitrarily complex black box functions. As mentioned in Sec. 1 already, any LF expressed in a general purpose programming language like Python can be expressed as a rule by wrapping it as a predicate that compares the label returned by the rule against a constant label.

Boris says: We may have to show an example for that with 3 labels to clarify that

. We have implemented an importer for Python LFs that analyzes the code of the LF. For simple predicates like checking for the existence of words in a sentence, our importer generates a separate predicate for each such check. As a fall-back the importer will wrap the whole LF as predicates as explained above. 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

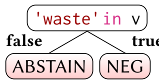


Figure 4: A rule encoding of the LF `keyword_word_waste` from Fig. 1a

over a data point x by substituting the variable v with x in the predicates of the rule. To determine the label for a data point x, the predicate of the root node of the rule's tree is evaluated by substituting v with x 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 x.

Definition 2.2 (Rule). *A rule r over atomic predicates $\mathcal{P}$ is a labeled directed binary tree where the internal nodes are predicates in $\mathcal{P}$, leaves are labels from $\mathcal{Y}$, and edges are marked with* **true** *and* **false***. A rule r takes as input a data point $x \in \mathcal{X}$ and returns a label $r(x) \in \mathcal{Y}$ for this assignment. Let $\textsc{root}(r)$ denote the root of the tree for rule r and let $C_{\mathbf{true}}(n)$ ($C_{\mathbf{false}}(n)$) denote the child of node n adjacent to the outgoing edge of n labeled* **true** (**false**)*. Given a data point x, the result of rule r for x is $r(x) = \textsc{eval}(\textsc{root}(r), x)$. Function $\textsc{eval}(\cdot,\cdot)$ operates on nodes n in the rule's tree and is recursively defined as follows:*

$$\textsc{eval}(n, x) = \begin{cases} y & \text{if } n \text{ is a leaf labeled } y \in \mathcal{Y} \\ \textsc{eval}(C_{\mathbf{true}}(n), x) & \text{if } n(x) \text{ is true} \\ \textsc{eval}(C_{\mathbf{false}}(n), x) & \text{if } n(x) \text{ is false} \end{cases}$$

Here $n(x)$ denotes replacing variable v in the predicate of node n with x and evaluating the resulting predicate.

We provide linear time procedures for converting LFs to rules in App. A. In particular, we have the following observation.

Proposition 2.3. *Any labeling function can be translated to a rule, given a suitable atomic predicate space $\mathcal{P}$.*

Example 2.4. *Fig. 4 shows the rule for a labeling function that returns* POS *if the length of the review contains the word 'waste' and returns* ABSTAIN *otherwise.*

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 2.5 (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 $\mathcal{X}$ (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 $\mathcal{M}_{\mathcal{R}} : \mathcal{X} \to \mathcal{Y}$.

Boris says: Replace with LF example and describe the refinement in example 2.6

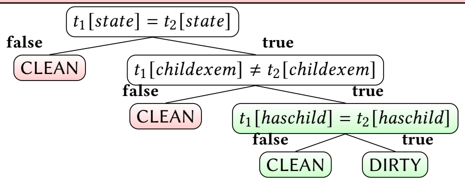

$t_1[state] = t_2[state]$
false
true
CLEAN
$t_1[childexem] \neq t_2[childexem]$
false
true
CLEAN
$t_1[haschild] = t_2[haschild]$
false
true
CLEAN
DIRTY

Figure 5: A refinement of DC r_5 from Fig. 4 (right) that assigns label CLEAN to assignment λ: $\lambda(t_1) = x_5$ and $\lambda(t_2) = x_6$. Nodes added in refinement are highlighted in green.

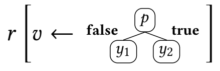
Ground Truth Labeling. The user inspects the labels produced by the RBBM and specifies their expectations about the ground truth for labels as a partial function $C^* : \mathcal{X} \to \mathcal{Y}$ that provides the true label for a subset of the data points from $\mathcal{X}$. We assume that the user only gives ground truth labels to data points x such that $\hat{y_x} \neq y_0$ (ABSTAIN for LFs in our setting), since these data points do not give any useful information about potential errors in the rule set used by the RBBM. $C^*(x)$ is POS, NEG for LFs. 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 Tab. 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 ($\hat{y_x} = N, C^*(x) = P$).

Rule Repair. RuleCleaner then generates a repair Φ 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 with an accuracy above a user-provided threshold. The user can choose to repair the rules according to Φ, 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.

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, a data point x, a predicate p to add to r, and a desired label $y^* \in \mathcal{Y}$. Let P be the path in r taken by x leading to a leaf node n. 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 n with a new node labeled p and adds the new leaf nodes for y_1, y_2, i.e.,*



Example 2.7.

Boris says: *Replace this with an LF example refinement*

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 ϕ 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 Φ is a sequence of repair operations and $\Phi(\mathcal{R})$ denotes the result of applying Φ 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^*. Given an accuracy threshold $\tau_{acc} \in \mathbb{R}$, the rule repair problem aims to find a repair sequence Φ that achieves accuracy $\geq \tau_{acc}$ for the RBBM $\mathcal{M}_{\Phi(\mathcal{R})}$ with the refined rule set on the datapoints from C^*:*

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

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^*.

Boris says: Expand intro discussion here on why repairing the rules to return correct labels is still a reasonable approach

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. B. □

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 ϕ in the repair sequence Φ. 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 Φ, we define $cost(\Phi) = \sum_{\phi \in \Phi} cost(\phi)$. Automated systems for rule generation may generate spurious LFs. 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 τ_{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 [1], we employ the greedy algorithm shown in Algorithm 1 to explore relevant 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^*, the set of rules $\mathcal{R}$, the cost of deletions τ_{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* τ_{pre} based on which the algorithm prunes rules with poor accuracy early on, and a *model-reevaluation frequency* τ_{reeval} to specify how frequently the model is rebuild.

Boris says: Should we keep the pre-deletion threshold as for LFs there is typically no need for aggressive deletion?

(1) Deleting bad rules upfront. As a first step, the algorithm prunes rules that perform poorly on C^*, i.e., whose accuracy on C^* 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.

Boris says: If we want to keep this we should find a example where LF generation techniques produce too many bad rules

(2) Refining or deleting 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 ground truth labels C^*, 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 assigns the ground truth labels for data points from C^*.

(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 $\mathcal{M}$ of the RBBM periodically to test whether the repair we have produced so far is successful (the updated model's accuracy is above τ_{acc} for C^*). For Snorkel, this steps amounts to retraining the model that Snorkel uses to predict labels based on the labels predicted by the rules.

[1]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.

Algorithm 1: Rule Set Repair

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

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

```
1  $\mathcal{R}_{repair} \leftarrow \mathcal{R}$
2  for $r \in \mathcal{R}$ do
3      if Accuracy$(r, C^*) < \tau_{pre}$ then
4          $\mathcal{R}_{repair} \leftarrow \mathcal{R}_{repair} \setminus \{r\}$
5  for $i \in [1, n]$ do
6      $\Phi \leftarrow$ SingleRuleRefine$(r_i, C^*)$;
7      if $cost(\Phi) < \tau_{del}$ then
8          $\mathcal{R}_{repair} \leftarrow \Phi(\mathcal{R}_{repair})$;
9      else
10         $\mathcal{R}_{repair} \leftarrow \mathcal{R}_{repair} \setminus \{r_i\}$;
11     if $i$ % $\tau_{reeval} = 0$ then    /* retrain every $i$ iterations */
12         $\mathcal{M}_{\mathcal{R}_{repair}} \leftarrow$ Reevaluate$(\mathcal{R}_{repair}, \mathcal{X})$
13         if Accuracy$(\mathcal{M}_{\mathcal{R}_{repair}}, C^*) \geq \tau_{acc}$ then
14             return $\mathcal{R}_{repair}$
15 return $\mathcal{R}_{repair}$
```

The cost of regenerating the model is typically 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 τ_{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.

Boris says: Expand this discussion to argue why in practice this often works.

Example 3.1. *Consider Ex. 1.1 and Tab. 1 for refining labeling functions (LFs). Using the LFs from Fig. 1a generated by Witan [14], Snorkel has generated the predicated labels for the reviews shown in Tab. 1. Assume that the user has given the ground truth labels C^* for reviews in this subset of the dataset (column* true label *in Tab. 1). Assume that we set $\tau_{reeval} = 4$ (we rerun Snorkel only once after having repaired 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. 1b (explained in Sec. 4) which adds a predicate* `'one' in x` *to the rule. Even without understanding yet how our algorithm selected the new predicate, intuitively this refinement is sensible: The updated rule returns the correct label (or abstains) for all sentences in C^*. In the following iteration, rule LF-2 is not modified as it does not return incorrect labels. Rule*

Algorithm 2: SingleRuleRefine

```
Input  : Rule $r$
         Labelled Datapoints $C^* \subseteq \mathcal{X} \times \mathcal{Y}$
Output : Repair sequence $\Phi$ such that $\Phi(r)$ fixes $C^*$
$Y \leftarrow \emptyset, \Lambda_P \leftarrow \emptyset$
$P_{fix} \leftarrow \{P[r,x] \mid \exists(x,y) \in C^*\}$
foreach $P \in P_{fix}$ do
    $\Lambda_P[P] \leftarrow \{x \mid \exists(x,y) \in C^\Lambda : P = P[r,x]\}$
$r_{cur} \leftarrow r$
/* Iterate over paths that need to be fixed */
foreach $P \in P_{fix}$ do
    /* Fix path $P$ to return correctly labels on $C^*$ */
    $\phi \leftarrow$ RefinePath($r_{cur}, \Lambda_P[P]$)
    $r_{cur} \leftarrow \phi(r_{cur})$
    $\Phi \leftarrow \Phi$.append($\phi$)
return $\Phi$
```

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 (=τ_{del}) new predicates, none of the rules get deleted.*

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.

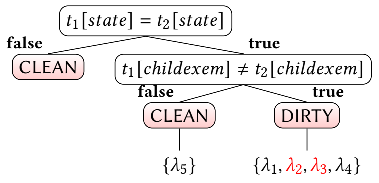
4 SINGLE RULE REFINEMENT

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 $(x, y) \in C^*$ to denote that $C^*(x) = y$ has been specified in the partial ground truth labels provided by the user.

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

$$\Phi_{min} = \underset{\Phi}{\operatorname{argmin}}\ cost(\Phi) \quad \textbf{subject to} \quad \forall(x,y) \in C^* : r_{fix}(x) = y$$

The pseudocode for `SingleRuleRefine` is given in Algorithm 2. Given a single rule r, this algorithm determines a refinement-based repair Φ_{min} for r such that $\Phi_{min}(r)$ returns the ground truth label $C^*(x)$ for *all* assignments specified by the user in C^*. 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^Λ. 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 data points x following a given path P



$t_1[state] = t_2[state]$
false
true
CLEAN
$t_1[childexem] \neq t_2[childexem]$
false
true
CLEAN
DIRTY
$\{\lambda_5\}$
$\{\lambda_1, \lambda_2, \lambda_3, \lambda_4\}$

Figure 6: Example for paths taken by assignments. Assignments with $C^\Lambda(\lambda)$ = DIRTY are highlighted in red.

in the rule r get assigned their desired labels based on C^*, these algorithms add predicates at the end of P to "reroute" each data point to a leaf node with its ground-truth label.

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

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 a data point $x \in \mathcal{X}^*$ has as prefix a path from P_{fix}. That is, for $P_1 \neq P_2 \in P_{fix}$, any refinement of P_1 can only affect the labels for data points in $\mathcal{X}_{P_1}$, but not the labels of data points in $\mathcal{X}_{P_2}$ as all data points in $\mathcal{X}_{P_2}$ are bound to take paths in any refinement r' that start with P_2. Hence we can determine repairs for each path in P_{fix} independently (the proof is shown in App. C.1).

Proposition 4.2 (Path Independence of Repairs). *Given a rule r and C^*, let $P_{fix} = \{P_1, \ldots, P_k\}$ and let Φ_i denote a refinement-based repair of r for $\mathcal{X}_{P_i}$ of minimal cost. Then $\Phi = \Phi_1, \ldots, \Phi_k$ is a refinement repair for r and C^* 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 data points $x_1 \neq x_2$ there exists $p \in \mathcal{P}$ such that:

$$p(x_1) \neq p(x_2)$$

Observe that any two data points $x_1 \neq x_2$ have to differ in at least one atomic unit, say A: $x_1[A] = c \neq x_2[A]$. If $\mathcal{P}$ includes all comparisons of the form $v[A] = c$, then any two data points can be distinguished. In particular, for labeling text documents, where the atomic units are words, $\mathcal{P}$ is partitioning if it contains $w \in v$ for every word w (we formally state this claim Lem. C.1 in App. C.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^Λ to all assignments in $\mathcal{X}^*$.

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 data points on path P. Then there exists a refinement repair Φ 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 data points $\mathcal{X}^n$ at a node n iteratively selects two data points $x_1, x_2 \in \mathcal{X}^n$ such that $C(x_1) \neq C(x_2)$ and adds a predicate that splits $\mathcal{X}^n$ into $\mathcal{X}^n_1$ and $\mathcal{X}^n_2$ such that $x_1 \in \mathcal{X}^n_1$ and $x_2 \in \mathcal{X}^n_2$. The full proof is shown in App. C.3. □

Of course, as we show in App. C.4, an optimal repair of r wrt. $\mathcal{X}^*$ may require less than $|\mathcal{X}^*|$ refinement steps. Nonetheless, this upper bound will be used in the brute force algorithm we present in Sec. 4.3.2 to bound the size of the search space. 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 x_1 and x_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 x_1 and x_2 and select $v[A] = c$ such that $x_1[A] = c \neq x_2[A]$. More generally, in App. C.5 we show that we can determine equivalence classes of predicates in $\mathcal{P}$ wrt. a set of data points 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 Φ for rule r such that the repaired rule $r' = \Phi(r)$ returns the ground truth labels from C^* for all data points $x \in \mathcal{X}^*$. We will again overload C^* to denote the ground truth for P and use $\mathcal{X}_P$ to denote the set of assignments from C^* for P. These algorithms refine r by replacing $last(P)$ with a subtree to generate a refinement repair Φ. As shown in App. C.5 , we only need to consider a finite number of predicates specific to $\mathcal{X}_P$.

4.3.1 GreedyPathRepair. This algorithm (pseudocode in App. D.1) maintains a list of pairs of paths and data points at these paths to be processed. This list is initialized with all data points $\mathcal{X}^*$ from C^* and the path P provided as an input to the algorithm. In each iteration, the algorithm picks two data points x_1 and x_2 from the current set and selects a predicate p such that $p(x_1) \neq p(x_2)$. It then refines the rule with p and appends $\Lambda_1 = \{x \mid x \in \Lambda \wedge p(x)\}$ and $\Lambda_2 = \{x \mid x \in \Lambda \wedge \neg p(x)\}$ with their respective paths to the list. As shown in the proof of Prop. 4.3, this algorithm terminates after adding at most $|\mathcal{X}^*|$ new predicates.

4.3.2 BruteForcePathRepair. The brute-force algorithm (pseudocode in App. D.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 on C^*. We first determine all predicates that can be used in the candidate repairs. As argued in App. C.5, there are only finitely many distinct predicates (up to equivalence) for a given set C^*. 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 $n = |C^*|$ as it may generate all substrees of size up to n.

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. $\Lambda$
1  $todo \leftarrow [(P, C^\Lambda)]$
2  $\Phi \leftarrow []$
3  $r_{cur} \leftarrow r$
4  $\mathcal{P}_{all} \leftarrow$ GetAllCandPredicates($P, \Lambda, C^\Lambda$)
5  while $todo \neq \emptyset$ do
6      $(P_{cur}, C) \leftarrow pop(todo)$
7      $p_{new} \leftarrow \text{argmin}_{p \in \mathcal{P}_{all}} I_G(p, \Lambda_{cur})$
8      $C_{\mathbf{false}} \leftarrow \{(\lambda, y) \mid (\lambda, y) \in C \wedge \neg p(\lambda)\}$
9      $C_{\mathbf{true}} \leftarrow \{(\lambda, y) \mid (\lambda, y) \in C \wedge p(\lambda)\}$
10     $y_{max} \leftarrow \text{argmax}_{y \in \mathcal{Y}} |\{\lambda \mid C_{\mathbf{true}}(\lambda) = y|$
11     $\phi_{new} \leftarrow \mathbf{refine}(r_{cur}, P_{cur}, y_{max}, p, \mathbf{true})$
12     $r_{cur} \leftarrow \phi_{new}(r_{cur})$
13     $\Phi \leftarrow \Phi, \phi_{new}$
14     if $|\mathcal{Y}_{C_{\mathbf{false}}}| > 1$ then
15         $todo.push((P[r_{cur}, C_{\mathbf{false}}], C_{\mathbf{false}}))$
16     if $|\mathcal{Y}_{C_{\mathbf{true}}}| > 1$ then
17         $todo.push((P[r_{cur}, C_{\mathbf{true}}], C_{\mathbf{true}}))$
18 return $\Phi$
```

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^*). Furthermore, by randomly selecting predicates to separate two data points (ignoring all other data points for a path), this algorithm will often yield repairs with a suboptimal cost. In contrast, BruteForcePathRepair produces minimal repairs that also generalize better, but the exponential runtime of the algorithm limits its applicability 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 data points into two sets that best separates data points with different labels at each step.

To measure the quality of a split, we employ the entropy-based *Gini impurity score* I_G [30]. Given a candidate predicate p for splitting a set of data points and their labels C, we denote the subsets of C generated by splitting C based on p:

$$C_{\mathbf{false}} = \{(x, y) \mid (x, y) \in C \wedge \neg p(x)\} \quad C_{\mathbf{true}} = \{(x, y) \mid (x, y) \in C \wedge p(x)\}$$

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

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

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

$I_G(C)$ is minimal if $\mathcal{Y}_C = \{y \mid \exists x : (xy) \in C\}$ contains a single label. Intuitively, we want to select predicates such that all data points 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 using function `GetAllCandPredicates`. Then it iteratively selects predicates until all data points are assigned 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 data points with labels C from the queue. We then determine the predicate p that minimizes the entropy of C. Afterwards, we create the subsets of data points from C which fulfill p and those which do not. We then generate a refinement repair step ϕ_{new} for the current version of the rule (r_{cur}) that replaces the last element on P_{cur} with predicate p. The child at the **true** edge of the node for p is then assigned the most prevalent label y_{max} for the data points 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. D.2).

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

$$\forall (x, y) \in C^* : \Phi(r)(x) = y$$

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.

Weak supervision & data programming. Weak supervision is a general technique of learning from noisy supervision signals that has been applied in many contexts [19, 35, 40–42, 46], e.g., for data labeling to generate training data [40, 41, 46] (the main use case we target in this work), for data repair [19, 42], and for entity matching [35]. The main advantage of weak supervision is that it reduces the effort of creating training data without ground truth labels. The data programming paradigm pioneered in Snorkel [40] has the additional advantage that the rules used for labeling are interpretable. However, as such rules are typically noisy heuristics, systems like Snorkel combine the output of LFs using a model.

Automatic generation and fixing labeling functions. While the data programming paradigm proves to be effective, asking human annotators to create a large set of high-quality labeling functions requires domain knowledge, programming skills, and time. To this end, automatically generating labeling heuristics from unlabeled data has received much attention from the research community. *ULF* [44] is an unsupervised system for fixing labeling functions using k-fold cross validation, extending previous approaches aimed at compensating for labeling errors [34, 50].

Witan [14] 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.

Starting from a seed set of LFs, *Darwin* [**?**] generates heuristic LFs under a context-free grammar and uses a hierarchy to capture the containment relationship between LFs that helps determine which to be verified by users. Unlike RuleCleaner that also repairs LFs based on user feedback, *Darwin* will only use the user feedback to update its LF scoring model. *Snuba* [46] fits classification models like decision trees and logistic regressors as LFs on a small labeled training set, followed by a pruner to determine which LFs to finally use. *Datasculpt* [22] uses a large language model (LLM) to generate labeling functions. Given as input a small set of training data with known ground truth, the system prompts an LLM with in-context examples of labels and keyword-based or pattern-based LFs, asking the LLM to generate LFs for an unlabeled example. Although *Datasculpt* filters the generated LFs based on accuracy and diversity, quality issues remain, and RuleCleaner can further improve the LF accuracy, as discussed in Section 6.6.

Hsieh et al. [25] 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 weakly-supervised training data generation and to use this information to automatically delete and refine rules.

Explanations for weakly supervised systems. While there is a large body of work on explaining the results of weak-supervised systems that target improving the final model or better involving human annotators in data programming [**? ? ? ?**], 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.

Boris says: find snorkel citation

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 weakly-supervised labeling (LF) using *Snorkel* [40] as the RBBM. We evaluate both the runtime and the quality of the repairs produced by our system with respect to several

dataset	avg(#word)	#row	$\mathcal{Y}$	# Witan	# Datasculpt[22]
Amazon [33]	68.86	200,000	positive/negative	15	86
Amazon-0.5	56.01	100,000	positive/negative	15	
Enron [29]	317.03	27,716	ham/spam		
YTSpam [2]	15.60	1,957	ham/spam		65
PT		24,588	professor/teacher	7	110
PA		12,236	painter/architect	10	120

Table 2: LF dataset statistics

parameters. We also use our system to rules generate automatically using Witan [14] and Datasculpt [22].

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. 2. 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. 2).

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 τ_{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 Weakly-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. 7, 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.

As shown in Fig. 7a, the runtime for *Brute Force* is up to ~4 orders of magnitude larger than the runtime of *Entropy* and *Greedy*. As shown in Fig. 7b, *Brute Force* and *Entropy* achieve similar results in terms of new global acc. and both outperform *Greedy*. In Fig. 7c, 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. 7e 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, 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.

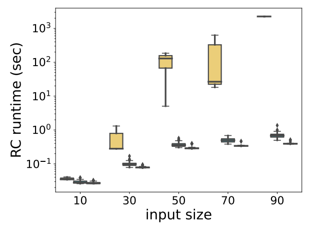
Varying complaint ratio. We also evaluated how the ratio between complaints and confirmations (complaint ratio) affects repairs. Fig. 7f shows the avg. size incr. for different input size values when varying the complaint ratio. As the complaint ratio increases, there's a trend for the size increase that goes up and peaks at around 50% complaint rate and then goes down after the peak. In App. E.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.

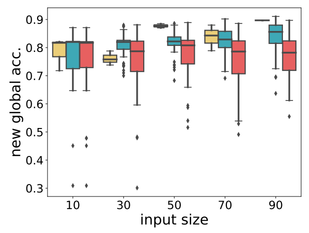
6.2 Scalability

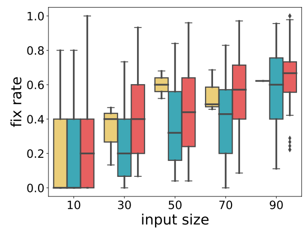
In this section, we evaluate the scalability of RuleCleaner. The summary of the datasets we used are summarized in Tab. 2. For LF repair, we used *Entropy*, with complaint ratio =50% and τ_{del} =0. For each dataset we vary the input size and measure the run time for both the black box model (Snorkel 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. 2) 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.

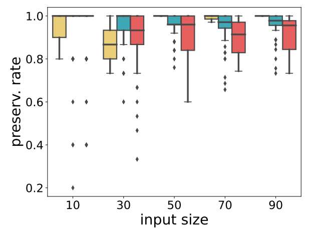
6.3 Varying Reevalation Frequency

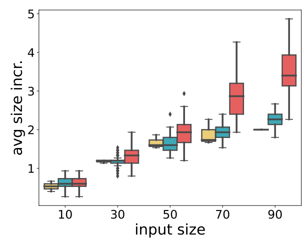
In the experiments discussed so far, we have let RuleCleaner fix all rules and only reevaluate the RBBM once after the repair. We now investigate the impact of the retraining frequency τ_{reeval} on runtime, repair cost, and repair quality. Intuitively, if we retrain more frequently then we may stop 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.


(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 $|C^*|$

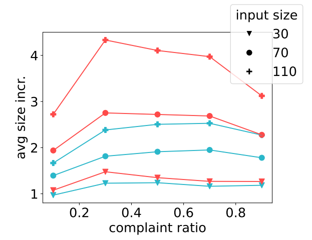
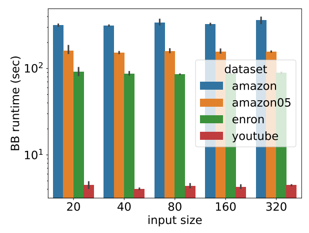

(f) avg. size incr. varying complaint ratio

Figure 7: Quality results for weakly-supervised labeling

Amir says: Chenjie: is the experiment here done only using LFs?

Chenjie says: Yes, we did the same for DCs but they are commented out


(a) Black Box Model Runtime

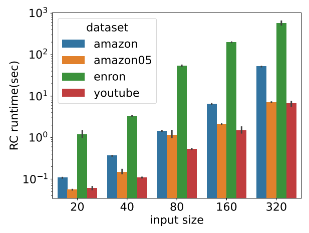

(b) RuleCleaner Runtime

Figure 8: Scalability: weakly-supervised labeling

Fig. 13 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 τ_{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.

6.4 Varying Deletion Cost

We now evaluate the impact of parameter τ_{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 τ_{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.

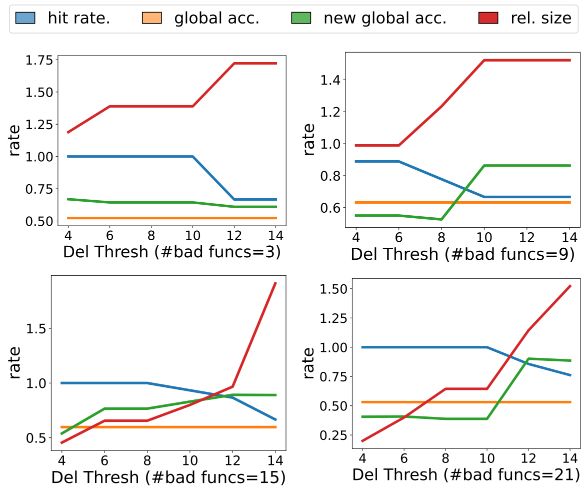
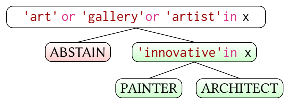
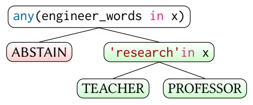


Figure 9: Deletion



(a) Refined rule for PA



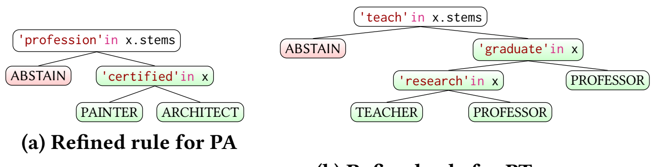
(b) Refined rule for PT

Figure 10: Refined Witan LFs

6.5 Repairing Witan Labeling Functions

To test the effectiveness of RuleCleaner in improving an automatically generated ruleset, we used labeling functions generated by Witan [14] for *Amazon*, *PT*, and *PA* as input. We still use Snorkel [40] as the RBBM. A summary of the datasets and number of labeling functions is shown in Tab. 2. 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.

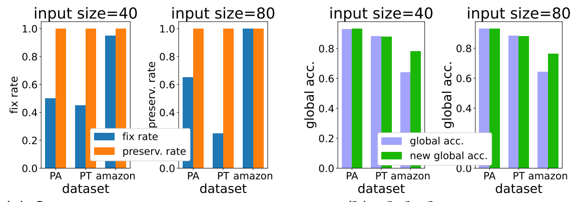
For the *PT* rule shown in Fig. 11b (labeling persons as teachers or professors), the refined rule added a check for word *research* in the person description. This is very sensible since professors tend to be more involved in research compared to school teachers. Fig. 12 shows the quality of our repaired 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.



(a) Refined rule for PA

(b) Refined rule for PT

Figure 11: Refined Datasculpt LFs



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

Figure 12: Witan Case Study Quality

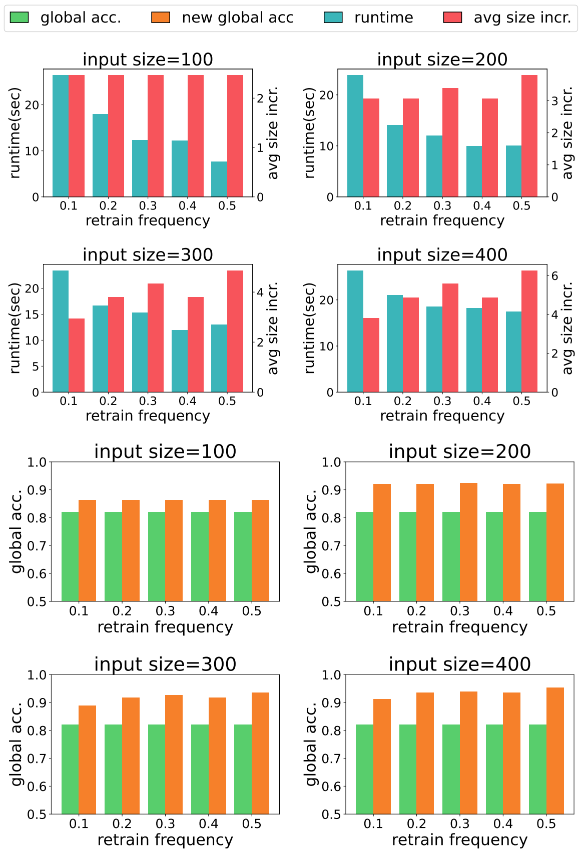
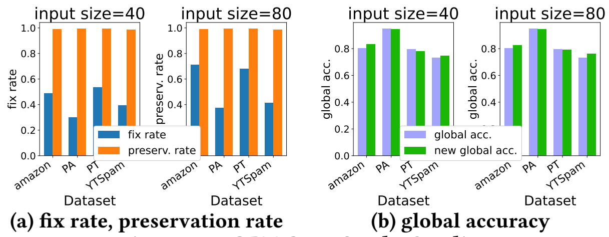


Figure 13: Varying τ_{reeval}, retraining every x%

6.6 Repairing DataSculpt (ChatGPT) Labeling Functions

Recently emerged pre-trained language model(PLM) like GPT-3 and GPT-4 have been shown to achieve impressive performances in various tasks. There have been some recent work developed to utilize this powerful tool in data programming tasks. Datasculpt [22] is an interactive framework that uses PLMs to generated LFs. Using the predefined prompt templates to interacte with PLMs, it



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

Figure 14: GPT Case Study Quality

generates keyword or regular expression based LFs on various tasks. Although PLMs can help design decent keyword based functions, the authors pointed out some limitations of this approach such as the lack of good instance selection strategy for PLM to generate LFs from. In this section, we investigate if RuleCleaner could improve the model performance trained using the LFs generated by PLMs. In this example, we tested on *Amazon*, *YTSpam*, *PA* and *PT* with Snorkel [40] as the RBBM. We followed the default setup in Datasculpt and created some examples for each class label as part of the system prompt. We then sampled the same amount of examples for each class in each dataset as query instances. The number of generated LFs are listed in Tab. 2. As shown in Fig. 14, after the repairs, we achieved decent fix rates among 4 different datasets while preserving the confirmed correct labels. One thing to note is that compared to the Witan case study in Sec. 6.5, the quality of the results is more sensitive to the user input size. This is because the number of LFs we get from [22] is significantly more than the LFs we get from Witan, which indicates that we might need more user input in order to revert some of the predictions.

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 weakly-supervised fashion. Many important applications can be modeled as RBBMs, including weakly-supervised labeling. 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 [14] or Datasculpt [22]. 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 [31, 43]. 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] 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.

[5] 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.

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

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

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

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

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

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

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

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

[14] 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.

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

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

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

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

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

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

[21] Naiqing Guan, Kaiwen Chen, and Nick Koudas. 2023. Can Large Language Models Design Accurate Label Functions? *CoRR* abs/2311.00739 (2023). arXiv:2311.00739

[22] Naiqing Guan, Kaiwen Chen, and Nick Koudas. 2023. Can Large Language Models Design Accurate Label Functions? *CoRR* abs/2311.00739 (2023). https://doi.org/10.48550/ARXIV.2311.00739 arXiv:2311.00739

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

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

[40] 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.

[41] 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.

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

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

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

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

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

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

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

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

[50] 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.

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

[52] 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 LF to Rule

```
Input  : An LF $f$
Output : A rule representation of $f$, $r_f$
1  $V, E = \emptyset, \emptyset$;
2  $r_f \leftarrow (V, E)$;
3  Let $D$ be the flow diagram of $f$ for every possible data
     flow;
4  LF-to-Rule($r, D$):
5      Let $r(D)$ be $\varphi$;
6      if $\varphi \notin \mathcal{Y}$ then
7          Let $\varphi' = P_2 \circ \varphi'$, where $\circ \in \{\vee, \wedge\}$;
8          $V \leftarrow V \cup \{P_1\}$;
9          if $\varphi = P_1 \vee \varphi''$ then
10             $E \leftarrow E \cup \{(P_1, P_2, False)\}$;
11         else if $\varphi = P_1 \vee \varphi'$ then
12             $E \leftarrow E \cup \{(P_1, P_2, True)\}$;
13         LF-to-Rule($r, D_{\varphi'}$);
14     else
15         return $r_f$;
```

A TRANSLATING LABELING FUNCTIONS INTO RULES

Sudeepa says: check

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

For LFs, we assume that they can be written in a hierarchical manner such that each condition φ_1 leads to another condition φ_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. For instance, for Python labeling functions, this structure can be extracted from the code using standard libraries for code introspection like Python's AST library. Let us denote this flow diagram of an LF by D, where D_φ is the sub-diagram rooted at the condition φ, not including φ itself and denote the root of the diagram by $r(D)$. Algorithm 4 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. This approach is extensible, if a Python function uses constructs that our algorithm is not aware of, then we can wrap the whole function f as predicates of the form $f(v) = y$ for all labels $y \in \mathcal{Y}$. For instance, consider a blackbox labeling function f whose structure we were not able to identify and assume that $\mathcal{Y} = \{y_1, \ldots, y_k, \text{ABSTAIN}\}$, then we would create a rule as shown in Fig. 16 that compares the output of f against every possible label $y \in \mathcal{Y}$ and if f. Each such predicate node has a true child that is y, i.e., the rule will return y iff $f(x) = y$. Note that this translation can not just be applied to full labeling functions, but also to branches of conditional statements in labeling functions with code that our algorithm does not know how to translate into predicates.

<

Example A.1 (Translation of Complex Labeling Functions). *Consider the labeling function implemented in Python shown below. This function assigns label* POS *to each data point (sentences in this example) containing the word star or stars. For sentences that do not contain any of these words, the function uses a function* `sentiment_analysis` *to determine the sentence's sentiment and return* POS *if it is above a threshold. Otherwise,* ABSTAIN *is returned. Assuming that our translator supports predicates that check for words, but does not recognize sentiment analysis as a predicate, we can wrap the else branch into a new black box function (under the assumption that* `sentiment_analysis` *is a pure function and that we have determined which variables are in scope of the else branch and need to be passed to the new function). The new function is shown below. We show the generated rule in Fig. 15.*

```
def complex_lf(v):
  words = ['star', 'stars']
  if words.intersection(v):
     return POSITIVE
  else:
     sentiment = sentiment_analysis(v)
     return POSITIVE if sentiment > 0.7 else ABSTAIN
```

```
def blackbox_lf(v):
    sentiment = sentiment_analysis(v)
    return POSITIVE if sentiment > 0.7 else ABSTAIN
```

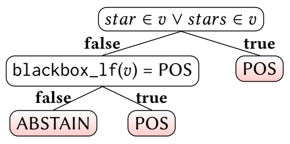


Figure 15: Translating a LF wrapping parts into a blackbox function

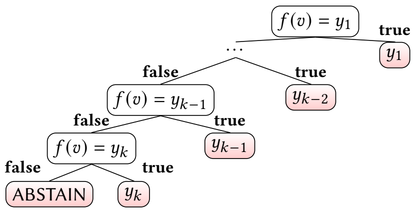


Figure 16: Translating a blackbox function

B PROOF OF THEOREM 2.9

Proof of Thm. 2.9. 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 $b \notin U$.

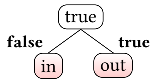


Figure 17: The rule representation of rule r used in App. B

- 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. 17). Hence initially any assignment will end up in $C_{\textbf{true}}(p) = out$.
- There are $n + 1$ assignments, where λ_e corresponds to tuple $e \in R$ (all e_i and b) and let's denote the assignment $\lambda_e(v) = e$. 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 Φ 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. 18

Boris says: Chenjie show such a tree

The repair sequence $\Phi = \phi_1, \ldots, \phi_k$ where ϕ_i introduces p_{i_j} has cost k.

It remains to be shown that Φ 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 Φ 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 λ_b.

As $v_{root} = p_{root} = true$, we know that $b_1 = \textbf{true}$ (λ_b takes the **true** edge of v_{root}. Furthermore, as $p(b)$ for all predicates in $\mathcal{P}$ (as $b \notin U$), λ_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 Φ 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 λ_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 λ_e would take the same path as λ_b and we have $r_{up}(\lambda_e) = out \neq in = C^{\Lambda}(\lambda_e)$ contradicting the fact that Φ 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 $\leq k$. ◻

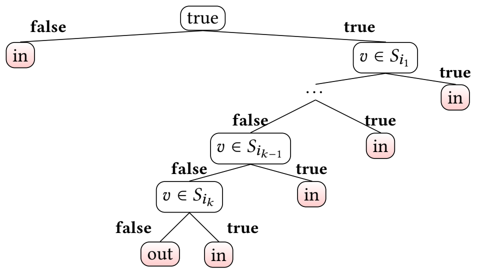


Figure 18: The rule representation of repair of rule r in App. B

C SINGLE RULE REFINEMENT - PROOFS AND ADDITIONAL DETAILS

C.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 Φ is not optimal. That is, there exists a repair Φ' with a lower cost. First off, it is easy to show that Φ' 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 Φ' yielding a repair of lower costs which contradicts the fact that Φ' is optimal. However, then we can partition Φ' into refinements Φ'_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 Φ_i is optimal for all i. Hence, no such repair Φ' can exist. ◻

C.2 Partitioning Predicate Spaces

Lemma C.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.*

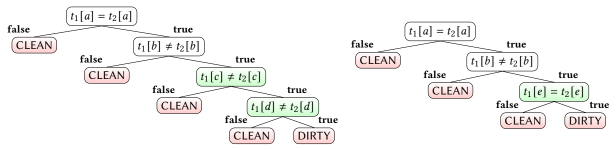


Figure 19: A non-optimal rule repair for the DC from Ex. C.2 produced by the algorithm from Prop. 4.3 and an optimal repair (right)

- ***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 $v \in \textsc{vars}(r)$ 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 λ_1 and λ_2. Thus, given that for any choice of assignment we can find a predicate that holds on λ_1 and does not hold on λ_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 λ_1 and λ_2. Thus, if $\mathcal{P}$ does only contain comparisons between attributes, it is not partitioning. □

C.3 Proof of Prop. 4.3

Proof of Prop. 4.3. We will use y_λ to denote the expected label for λ, 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 λ_1 and λ_2 from Λ_{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 Λ_{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 Λ_{cur} is reduced by at least one, the algorithm will terminate after adding at most $|\Lambda|$ predicates. □

C.4 Non-minimality of the Algorithm from Prop. 4.3

Example C.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 λ_1 and λ_3 and select predicate $t_1[c] \neq t_2[c]$ to distinguish between these assignments. However, λ_2 fulfills this predicate and thus in the next step, the algorithm may select λ_2 and λ_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. 19 (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. 19, 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
$x_1 \longrightarrow$	1	2	1	2
$x_2 \longrightarrow$	1	2	3	1
$x_3 \longrightarrow$	1	3	2	1
$x_4 \longrightarrow$	1	3	1	1

C.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. Λ 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 Λ (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 Λ 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 C.3 (Equivalence of Predicates). *Consider a space of predicates $\mathcal{P}$, rule r, and set of assignments Λ 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. Λ, written as $p \equiv_\Lambda p'$ if $p(\Lambda) = p'(\Lambda)$. Furthermore, consider a reduced space of predicates*

Algorithm 5: 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  $todo \leftarrow [(P, \Lambda)]$
 2  $\Phi = []$
 3  while $todo \neq \emptyset$ do
 4      $(P, \Lambda) \leftarrow pop(todo)$
 5      if $\exists \lambda_1, \lambda_2 \in \Lambda : C[\lambda_1] \neq C[\lambda_2]$ then
 6          /* Determine predicates that distinguishes
               assignments that should receive different labels
               for a path                                        */
 7          $p \leftarrow$ GetSeperatorPred$(\lambda_1, \lambda_2)$
 8          $y_1 \leftarrow C[\lambda_1]$
 9          $\phi \leftarrow$ refine$(r_{cur}, P, y_1, p,$ true$)$
10          $\Lambda_1 \leftarrow \{\lambda \mid \lambda \in \Lambda \wedge \lambda(p)\}$
11          $\Lambda_2 \leftarrow \{\lambda \mid \lambda \in \Lambda \wedge \neg\lambda(p)\}$
12          $todo.push((P[r_{cur}, \lambda_1], \Lambda_1))$
13          $todo.push((P[r_{cur}, \lambda_2], \Lambda_2))$
14      else
15          $\lambda \leftarrow \Lambda.pop()$          /* All $\lambda \in \Lambda$ have same label */
16          $\phi \leftarrow$ refine$(r_{cur}, P, C[\lambda])$
17      $r_{cur} \leftarrow \phi(r_{cur})$
18      $\Phi.append(\phi)$
19  return $\Phi$
```

$\mathcal{P}_{\equiv}$ that fulfills the following condition:

$$\forall p \in \mathcal{P} : \exists p' \in \mathcal{P}_{\equiv} : p \equiv p' \qquad (1)$$

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 $\mathcal{A}$, Φ ($\Phi_{\equiv}$) 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 Φ a repair Φ' with same cost as Φ 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 $\mathcal{P}_{\equiv}$. Φ' 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 $\lambda \in \Lambda$, Φ' 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 Λ 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.[2]

D PATH REFINEMENT REPAIRS - PROOFS AND ADDITIONAL DETAILS

D.1 GreedyPathRepair

Boris deleted: The algorithm explores repairs in increasing order of their cost. Repair candidates are encodes as trees (to replace the leaf node at the single path). The algorithm maintains a queue of repair candidates that is initialized with all single predicate refinements from $\mathcal{P}_{\equiv}$. Note that the label leaf nodes are not part of candidates in this queue but are determine dynamically when evaluating a candidate. In each iteration the algorithm pops a candidate from the queue, evaluates the candidate on all assignments for the path for which we are computing a repair. The result for each leaf node of the candidate is a (possible empty subset) of assignments from Λ which take the path to this leaf node in the candidate's predicate tree. If all of these sets are "pure" (all assignments of a set have the same expected label), then we have found a solution that is generated from the candidate by using the single expected label of all assignments at a leaf node as the label for this leaf node (leaf nodes with empty sets of assignments can be assigned an arbitrary label). Otherwise, we extend the candidate in all possible ways by replacing one of the leaf nodes with a predicate from $\mathcal{P}_{\equiv}$ that does not yet occur in the candidate and append all these new candidates to the end of the queue. The algorithm terminates once a solution has been found, which based on Prop. 4.3 will take at most a number of steps that is equal to the number of candidate of size up to $|\Lambda|$. However, note that this number is exponential in $|\Lambda|$ and polynomial in $|\mathcal{P}_{\equiv}|$. **because This is the optimal algorithm, merge there or remove**

Function GreedyPathRepair is shown Algorithm 5. 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 Λ_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 Λ_C. The algorithm processes sets of assignments until the todo queue is empty. In each iteration, the algorithm greedily selects a pair of assignments λ_1 and λ_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 λ_1 and false on λ_2 (or vice versa). If we extend path P with p, then λ_1 will follow the **true** edge of p and λ_2 will follow the **false** edge (or vice versa). This effectively partitions the set of assignments for path P into two sets Λ_1 and Λ_2 where Λ_1 contains λ_1 and Λ_2 contains λ_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 λ (line 14) and changes the current leave node's label to $C(\lambda)$.

[2]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 Λ into two sets Λ and $\emptyset$.

Algorithm 6: BruteForcePathRepair

```
Input  : Rule $r$
         Path $P$
         Assignments to fix $\Lambda$
         Expected labels for assignments $C$
Output : Repair sequence $\Phi$ which fixes $r$ wrt. $\Lambda$
$todo \leftarrow [(r, \emptyset)]$
$\mathcal{P}_{all}$ = GetAllCandPredicates($P, \Lambda, C$)
while $todo \neq \emptyset$ do
    $(r_{cur}, \Phi_{cur}) \leftarrow pop(todo)$
    foreach $P_{cur} \in leafpaths(r_{cur}, P)$ do
        foreach $p \in \mathcal{P}_{all} - \mathcal{P}_{r_{cur}}$ do
            foreach $y_1 \in \mathcal{Y} \land y_1 \neq last(P_{cur})$ do
                $\phi_{new} \leftarrow$ refine($r_{cur}, P_{cur}, y_1, p$, true)
                $r_{new} \leftarrow \phi_{new}(r_{cur})$
                $\Phi_{new} \leftarrow \Phi_{cur}, \phi_{cur}$
                if ACCURACY($r_{new}, C$) = 1 then
                    return $\Phi_{new}$
                else
                    $todo.push((r_{new}, \Phi_{new})$
```

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 Λ processed in the current iteration into two subsets which each are strictly smaller than Λ. Thus, the algorithm is guaranteed to terminate and by construction assigns each assignments λ in C its desired label $C(\lambda)$.

D.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 λ the label $C(\lambda)$.

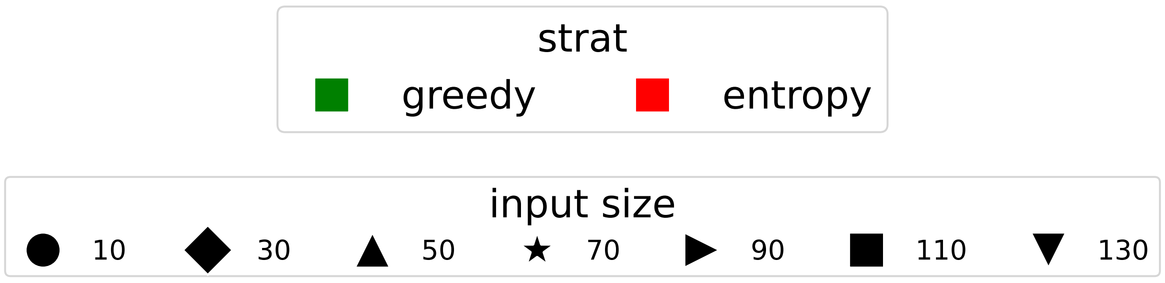
BruteForcePathRepair: The algorithm generates all possible trees build from predicates and leaf nodes in increasing order of their side. It terminates once a tree has been found that returns the correct labels on Λ_C. As there has to exist a repair of size n or less, the algorithm will eventually terminate.

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 Λ_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 Λ into two subsets Λ_1 and Λ_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. □

E ADDITIONAL EXPERIMENTS

E.1 Varying Complaint Ratio

We now focus on evaluating the effects of varying complaint ratio, τ_{del}, and input size on the results of *Entropy* and *Greedy* using *YTSpam*. The results are shown in Fig. 20 to 22. The general trend we observed in Fig. 7f 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. 22. when τ_{del} is set low, *Entropy* outperforms *Greedy*.



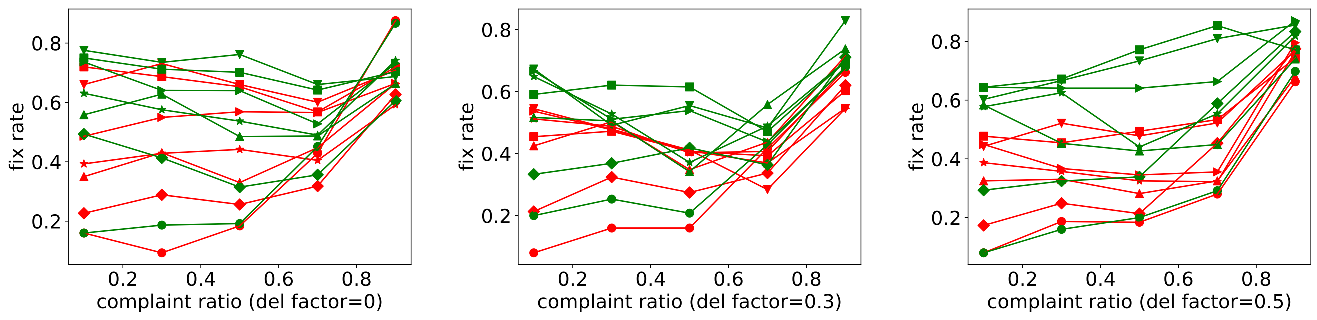


Figure 20: complaint ratio vs fix rate

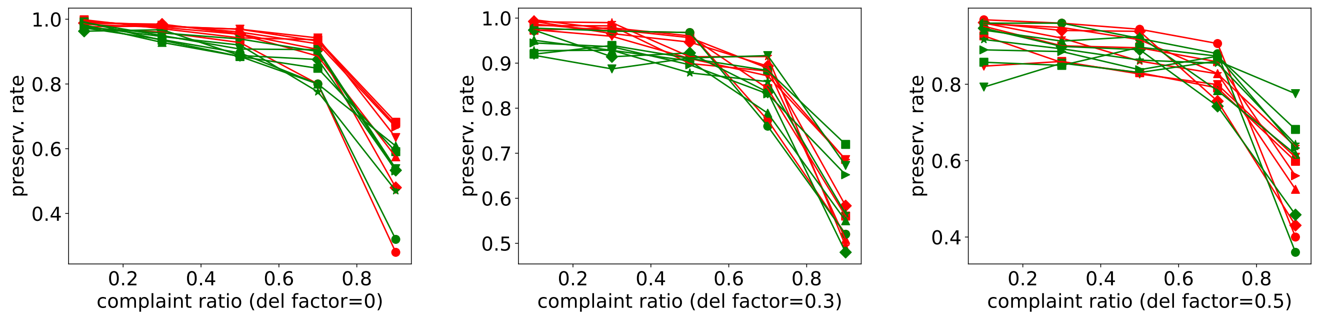


Figure 21: complaint ratio vs preserv. rate

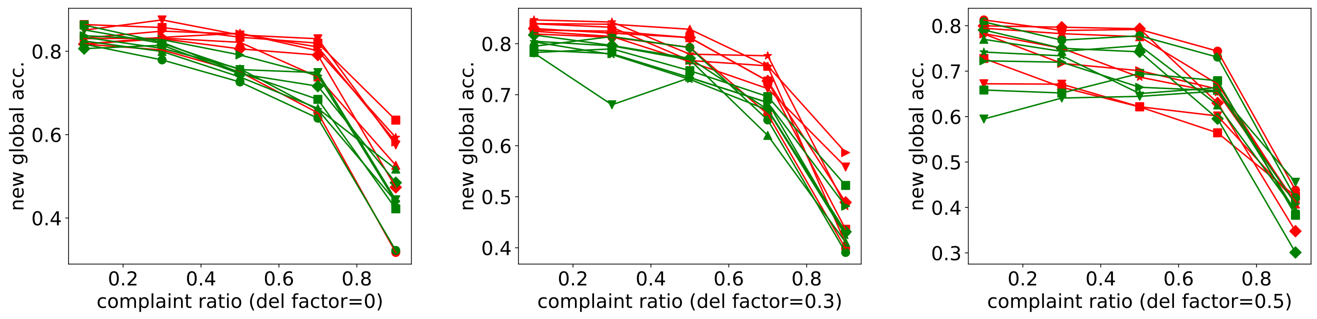


Figure 22: complaint ratio vs new global accuracy