Refining Labeling Functions With Limited Labeled Data

Anonymous
Author(s)

Abstract

Programmatic weak supervision (PWS) significantly reduces human effort for labeling data by combining the outputs of user-provided labeling functions (LFs) on unlabeled datapoints. However, the quality of the generated labels depends directly on the accuracy of the LFs. In this work, we study the problem of fixing LFs based on a small set of labeled examples. Towards this goal, we develop novel techniques for repairing a set of LFs by minimally changing their results on the labeled examples such that the fixed LFs ensure that (i) there is sufficient evidence for the correct label of each labeled datapoint and (ii) the accuracy of each repaired LF is sufficiently high. We model 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 LFs based on surprisingly small sets of labeled datapoints.

ACM Reference Format:
Anonymous Author(s). 2025. Refining Labeling Functions With Limited Labeled Data. In *Proceedings of 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '25).* ACM, New York, NY, USA, 24 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Programmatic weak supervision (PWS) [26, 35] is a powerful technique for creating training data. Unlike manual labeling, where labels are painstakingly assigned by hand to each training datapoint, data programming assigns labels by combining the outputs of labeling functions (LFs) — heuristics that take a datapoint as input and output a label — using a model. This approach dramatically reduces the human effort required to label data. To push this reduction even further, recent approaches automate the generation of LFs [6, 9, 12, 31]. For example, Witan [9] creates LFs from simple predicates that are effective in differentiating datapoints, subsequently guiding users to select and refine sensible LFs. Guan et al. [12] employ large language models (LLMs) to derive LFs based on a small amount of labeled data, further reducing the dependency on human intervention. One advantage of PWS over weak supervision with a black box model is that LFs are inherently interpretable.

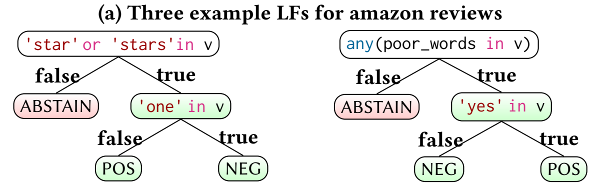
Regardless of whether LFs are manually crafted by domain experts or generated by automated techniques, users face significant challenges when it comes to repairing these LFs to correct issues with the resulting labeled data. The black-box nature of the model that combines LFs results obscures which specific LFs are responsible for mislabeling a datapoint, and large training datasets make it difficult for users to manually identify effective repairs. While explanation techniques for PWS [13, 36] can identify LFs responsible for erroneous labels, determining how to repair the LFs to fix these errors remains a significant challenge.

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

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

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

(a) Three example LFs for amazon reviews



(b) Refined rule for LF-1 **(c) Refined rule for LF-3**

Figure 1: LFs before / after refinement by RuleCleaner

In this work, we tackle the challenge of automatically suggesting repairs for a set of LFs based on a small set of labeled datapoints. Our approach refines an LF by locally overriding its outputs to align with expectations for specific datapoints. Rather than replacing human domain expertise or existing automated LF generation, our method, RuleCleaner, improves an existing set of LFs. Our approach is versatile, supporting arbitrarily complex LFs, and various black box models that combine them such as Snorkel [26] or simpler models like majority voting. RuleCleaner is agnostic to the source of LFs, enabling the repair of LFs generated by tools like Witan [9], LLMs [12], and those created by domain experts.

To address the challenge of refining LFs expressed in a general-purpose programming language, we model LFs as *rules*, represented as trees. In these trees, inner nodes are *predicates*, Boolean conditions evaluated on datapoints, and leaf nodes correspond to labels. Such a tree encodes a cascading series of conditions, starting at the root, each predicate directs navigation to a *true* or *false* child until a leaf node is reached, which assigns the label to the input datapoint. *This model can represent any LF as a rule* by creating predicates that match the result of the LF against every possible label (see App. A).

Example 1. *Consider the Amazon Review Dataset from [9, 15] which contains reviews for products bought from Amazon and the task of labeling the reviews as* POS *or P (positive), or* NEG *or N (negative). A subset of LFs generated by the Witan system [9] for this task are shown in Figure 1a.* `key_word_star` *labels reviews as* POS *that contain either* "star" *or* "stars" *and otherwise returns* ABSTAIN

KDD '25, Toronto, Canada

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

id	text	true label	old predicted label by Snorkel	new predicted label by Snorkel	LF labels: old (new) 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: Products reviews with ground truth labels ("P"ositive or "N" egative), predicted labels by Snorkel [26] (before and after rule repair), and the results of the LFs from Figure 1 ("-" means ABSTAIN). Results for repaired rules are shown in blue.

(the function cannot make a prediction). Some reviews with their ground truth labels (unknown to the user) and the labels predicted by Snorkel *[27] are shown in Table 1, which also shows the three LFs from Figure 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 LFs. We treat systems like Snorkel as a blackbox that can use any algorithm or model to generate labels.*

Suppose that for a small set of reviews, the true label is known (Table 1). RuleCleaner *uses these ground truth labels to generate a set of repairs for the LFs by deleting or refining LFs to align them with the ground truth. Table 1 also shows the labels produced by the repaired LFs (updated labels shown in blue), and the predictions generated by Snorkel before and after LF repair.* RuleCleaner *repairs LF-1 and LF-3 from Figure 1a by adding new predicates (refinement). Figure 1b and 1c show the refined rules in tree form with new nodes highlighted in green. Consider the repair for LF-1. Intuitively, this repair is sensible: a review mentioning* "one" *and* "star(s)" *is likely negative.*

Our RuleCleaner system produces repairs as shown in the example above. We make the following contributions.

- **The PWS Repair Problem**. We introduce a general model for PWS as programmatic weak supervision systems (PWSSs), where interpretable LFs (rules) are combined to predict labels for a dataset $\mathcal{X}$ (**Section 2**). We formalize the problem of repairing LFs in PWSS through refinement and deletion, showing the problem is NP-hard. To avoid overfitting, we (i) minimize the changes to the outputs of the original LFs and (ii) allow some LFs to return incorrect labels for some labeled datapoints.
- **Efficient Rule Repair Algorithm**. In spite of its hardness, in practice it is feasible to solve the repair problem exactly as the number of labeled examples is typically small. We formalize this problem as a mixed-integer linear program (MILP) that determines changes to the LF output on the labeled examples (**Section 3**). To implement these changes, we refine individual rules to match the desired outputs (**Section 4**). To further decrease the likelihood of overfitting and limit the complexity of the fixed rules, we want to minimize the number of new predicates that are added. This problem is also NP-hard. We propose a PTIME information-theoretic heuristic algorithm (**Section 4.2**).
- **Comprehensive Experimental Evaluation**. We conduct experiments on 11 real datasets using Snorkel [26] over LFs generated by Witan [9] or LLMs [12] (**Section 5**). Furthermore, we compare against using LLMs directly for labeling and for repairing LFs. RuleCleaner significantly improves labeling accuracy using a small number of labeled examples. While direct labeling with LLMs achieves impressive accuracy for advanced models like GPT-4o, it is also prohibitively expensive.

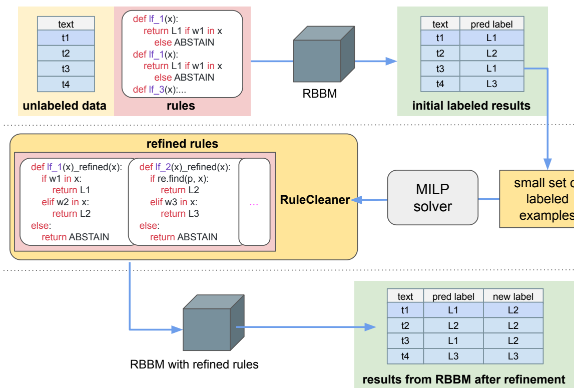


Figure 2: The RuleCleaner framework for repairing LFs (rules $\mathcal{R}$). After running an PWSS (e.g., Snorkel) on the rule outputs, RuleCleaner fixes the rules $\mathcal{R}$ using labeled examples $\mathcal{X}^*$. Finally, the PWSS is rerun on the output of the refined rules $\mathcal{R}^*$ on the whole dataset $\mathcal{X}$ to produce the repaired labels.

2 The RuleCleaner Framework

As shown in Figure 2, we assume as input a set of LFs modeled as rules $\mathcal{R}$, the corresponding labels produced by an PWSS for an unlabeled dataset $\mathcal{X}$, and a small subset of labeled datapoints $\mathcal{X}^* \subset \mathcal{X}$. RuleCleaner refines specific LFs based on this input, generating an updated set of rules $\mathcal{R}^*$. Finally, the PWSS, Snorkel [26] in this work, applies these revised rules to re-label the dataset.

2.1 Rules and PWSSs

To be able to repair a LF by selectively overriding its output based on conditions that hold for an input datapoint, we model LFs as a set of cascading conditions. A rule r is a *tree* where leaf nodes represent labels from a set of labels $\mathcal{Y}$ and the non-leaf nodes are labeled with Boolean predicates from a space of predicates $\mathcal{P}$. Each non-leaf node has two outgoing edges labeled with **true** and **false**. A rule r assigns a label $r(x)$ to an input datapoint x by evaluating the predicate at the root, following the outgoing edge **true** if the predicate evaluates to true and the **false** edge otherwise. Then the predicate of the node at the end of the edge is evaluated. This process is repeated until a leaf node is reached. The label of the leaf node is the label assigned by r to x.

Example 2. *Figure 3 shows the rule for a LF that returns* NEG *if the review contains the word 'waste' and returns* ABSTAIN *otherwise. In Proposition 2, we show how to translate any LF written in a general-purpose programming language into a rule in* PTIME*. We have implemented this algorithm for LF expressed as Python functions.*

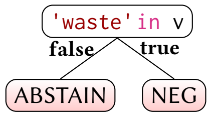


Figure 3: Rule form of the LF `keyword_word_waste` (Figure 1a)

In the sequel, we use the terms LF and rule interchangeably.

Consider a set of input *datapoints* $\mathcal{X}$ and a set of discrete *labels* $\mathcal{Y}$. For a datapoint $x \in \mathcal{X}$, y^*_x denotes the datapoint's (unknown) true label. A PWSS takes $\mathcal{X}$, the labels $\mathcal{Y}$, and a set of rules $\mathcal{R}$ as input and produces a *model* $\mathcal{M}_{\mathcal{R},\mathcal{X}}$ as the output that maps each datapoint in $\mathcal{X}$ to a label in $\mathcal{Y}$. Without loss of generality, we assume the presence of an *abstain label* $y_0 \in \mathcal{Y}$ that is used by the PWSS or a rule to abstain from providing a label to some datapoints.

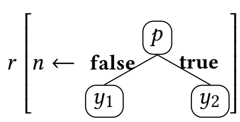
Definition 1 (PWSS). *Given a set of datapoints $\mathcal{X}$, a set of labels $\mathcal{Y}$, and a set of rules $\mathcal{R}$, a PWSS takes $\mathcal{R}(\mathcal{X})$ as input and produces a model $\mathcal{M}_{\mathcal{R},\mathcal{X}} : \mathcal{X} \rightarrow \mathcal{Y}$ that maps datapoints $x \in \mathcal{X}$ to labels*

$$\mathcal{M}_{\mathcal{R},\mathcal{X}}(x) = \hat{y_x}.$$

In the following, we will often drop $\mathcal{R}$ and $\mathcal{X}$ from $\mathcal{M}_{\mathcal{R},\mathcal{X}}$ when they are irrelevant to the discussion.

2.2 The Rule Repair Problem

Rule Refinement Repairs. We model a repair of a set of rules $\mathcal{R}$ as a **repair sequence** $\Phi = \phi_1, \dots, \phi_k$ of refinement steps ϕ_i and use $\mathcal{R}^* = \{r'_1, \dots, r'_k\}$ to denote $\Phi(\mathcal{R})$. We repair rules by *refining* them by replacing a leaf node with a new predicate to achieve a desired change to the rule's result on some datapoints. Consider a rule r, a path P ending in a node n, and a predicate p and two labels y_1 and y_2. The refinement **refine**(r, P, 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:



For example, Figure 1b shows the result of refinement where a leaf POS was replaced with the subtree highlighted in green.

Desiderata. Given the labeled training data $\mathcal{X}^*$, we would like the repaired rules to provide sufficient information about the true labels for datapoints in $\mathcal{X}^*$ to the PWSS without overfitting to the small number of labeled datapoints in $\mathcal{X}^*$. Specifically, we want the repair to fulfill the following desiderata:

Datapoint Evidence. We define the *evidence* for a datapoint x_i as the fraction of non-abstain labels ($\neq y_0$) the datapoint receives from the m rules in $\mathcal{R}$. The repaired rules should provide sufficient evidence for each datapoint x_i, such that the PWSS can make an informed decision about x_i's label.

$$\text{Evidence}(x_i) = \frac{\sum_j \mathbb{1}[r'_j(x_i) \neq y_0]}{m}$$

Datapoint Accuracy. The *accuracy* for a datapoint x_i as shown below. The accuracy of the repaired rules that do not abstain on x_i should be high.

$$\text{Acc}(x_i) = \frac{\sum_{j:r'_j(x_i)\neq y_0} \mathbb{1}[r'_j(x_i) = y^*_{x_i}]}{m}$$

Rule Accuracy. In addition, the rules should have high accuracy. The *accuracy* of a rule $r_j \in \mathcal{R}^*$ is defined as the fraction of the n datapoints in $\mathcal{X}^*$ on which it returns the ground truth label.

$$\text{Acc}(r'_j) = \frac{\sum_{i:r'_j(x_i)\neq y_0} \mathbb{1}[r'_j(x_i) = y^*_{x_i}]}{n}$$

Repair Cost. For a repair sequence Φ and $\mathcal{R}^* = \Phi(\mathcal{R})$ we define its cost as the number of labels that differ between the results of $\mathcal{R} = \{r_1, \dots, r_m\}$ and $\mathcal{R}^* = \{r'_1, \dots, r'_m\}$ on $\mathcal{X}^*$. Optimizing for low repair cost avoids overfitting to $\mathcal{X}^*$ and preserves rule semantics where feasible.

$$cost(\Phi) = \sum \mathbb{1}[r_j(x_i) \neq r'_j(x_i)]$$

We state the rule repair problem as an optimization problem: minimize the number of changes to labeling function results ($cost(\Phi)$) while ensuring the desiderata enforced by thresholds τ_E (evidence), τ_{acc} (accuracy), and τ_{racc} (rule accuracy).

Definition 2 (Rule repair problem). *Consider a black-box model $\mathcal{M}_{\mathcal{R},\mathcal{X}}$ that uses a set of m rules $\mathcal{R}$, a dataset of n datapoints $\mathcal{X}$, output labels $\mathcal{Y}$, and ground truth labels for a subset of datapoints $\mathcal{X}^*$. Given thresholds $\tau_{acc} \in [0,1]$, $\tau_E \in [0,1]$, and $\tau_{racc} \in [0,1]$, the* rule repair problem *is to find a repair sequence Φ such that:*

$$\begin{aligned} \mathbf{argmin}_{\Phi} \quad & cost(\Phi) \\ \mathbf{subject\ to} \quad & \forall\, i \in [1,n] : \text{Acc}(x_i) \geq \tau_{acc} \wedge \text{Evidence}(x_i) \geq \tau_E \\ & \forall\, j \in [1,m] : \text{Acc}(r'_j) \geq \tau_{racc} \end{aligned}$$

Note that since we treat the PWSS as a black box, we can, in general, not guarantee that the PWSS's performance on the unlabeled dataset $\mathcal{X}$ will improve. Nonetheless, we will demonstrate experimentally in Section 5 that significant improvements in the accuracy of rules on $\mathcal{X}$ can be achieved based on 10s of training examples. This is due to the use of predicates in rule repairs that generalize beyond $\mathcal{X}^*$. While finding an optimal repair is NP-hard, we can still solve this problem exactly as $\mathcal{X}^*$ is expected to be small.

Theorem 1. *The rule repair problem is NP-hard in the size of $\mathcal{R}$.*

3 Ruleset Repair Algorithm

We now present an algorithm that solves the rule repair problem in two steps. In the first step, we use an MILP to determine desired changes to the outputs of rules, and in the second step, described in Section 4, we implement these changes by refining individual rules to return the desired output on $\mathcal{X}^*$.

3.1 MILP Formulation

In the MILP, we use an integer variable o_{ij} for each datapoint $x_i \in \mathcal{X}^*$ and r_j that stores the label that the repaired rule r'_j should assign to x_i. That is, in combination these variables store the desired changes to the results of rules that we then have to implement by refining each rule r_j to a rule r'_j. We restrict these variables to take values in $[0, |\mathcal{Y}| - 1]$ where value i represents the label $y_i \in \mathcal{Y}$ with

0 encoding y_0. To encode the objective (minimizing the changes to the outputs of rules on $\mathcal{X}^*$) we use a Boolean variable m_{ij} for each rule r_j and datapoint x_i that is 1 iff $o_{ij} \neq r_j(x_i)$ (the output of r'_j on x_i is different from $r_j(x_i)$. The objective is then to minimize the sum of these indicators.

To encode the side constraints of the rule repair problem, we introduce additional indicators: c_{ij} is 1 if $o_{ij} = y^*_{x_i}$, and e_{ij} is 1 if $o_{ij} \neq y_0$. To ensure that the accuracy for each datapoint x_i is above τ_{acc}, we have to ensure that out of rules that do not return $y_0$on x_i, i.e., all $j \in [1, m]$ where $e_{ij} = 1$, at least a fraction of τ_{acc} have the correct label (c_{ij}=1). This can be enforced if $\sum_j c_{ij} - \sum_j e_{ij} \cdot \tau_{acc} \leq 0$ or equivalently $\sum_j c_{ij} \geq \sum_j e_{ij} \cdot \tau_{acc}$. A symmetric condition is used to ensure LF accuracy using the threshold τ_{racc} and summing up over all datapoints instead of over all rules. Finally, we need to ensure that each datapoint x_i receives a sufficient number of labels $\neq y_0$. Recall that e_{ij} encodes whether LF r'_j returns a non-abstain label. Thus, for m rules we have to enforce: $\forall i \in [1, n] : \sum_j e_{ij} \geq m \cdot \tau_E$. The full MILP is shown below. The non-linear constraints for indicator variables can be translated into linear constraints using the so-called Big M technique [11].

$$\textbf{minimize } \sum_i \sum_j m_{ij} \quad \textbf{subject to}$$

$$\begin{aligned}
&\forall i \in [1, n], j \in [1, m] : && \forall i \in [1, n] : \sum_j c_{ij} \geq \sum_j e_{ij} \cdot \tau_{acc} \\
&\quad o_{ij} \in [0, |\mathcal{Y}| - 1] && \\
&\quad m_{ij} = \mathbb{1}[o_{ij} \neq r_j(x_i)] && \forall i \in [1, n] : \sum_j e_{ij} \geq m \cdot \tau_E \\
&\quad c_{ij} = \mathbb{1}[o_{ij} = y^*_{x_i}] && \\
&\quad e_{ij} = \mathbb{1}[o_{ij} > 0] && \forall j \in [1, m] : \sum_i c_{ij} \geq \sum_i e_{ij} \cdot \tau_{racc}
\end{aligned}$$

As we show next, the solution of the MILP is a solution for the rule repair problem as long as the expected changes to the LF results on $\mathcal{X}^*$ encoded in the variables o_{ij} can be implemented as a repair sequence Φ. As we will show in Section 4 such a repair sequence is guaranteed to exist as long as we choose the space of predicates to use in refinements carefully.

Proposition 1. *Consider rules $\mathcal{R}$, $\mathcal{X}^*$, and the output o_{ij} produced as a solution to the MILP. If there exists a repair sequence Φ such that for $\mathcal{R}^* = \Phi(\mathcal{R})$ the output on $\mathcal{X}^*$ is equal to o_{ij} for all $i \in [1, n]$ and $j \in [1, m]$, then Φ is a solution to the rule repair problem.*

MILP Size. The number of constraints and variables in the MILP is both in $O(n \cdot m)$ where $n = |\mathcal{X}^*|$ and $m = |\mathcal{R}|$. While solving MILPs is hard in general, we demonstrate experimentally that the runtime is acceptable for $|\mathcal{X}^*| \leq 200$.

Example 3. *Consider a set of 3 datapoints $\mathcal{X}^* = \{x_1, x_2, x_3\}$ with ground truth labels $y^*_{x_1} = 2$, $y^*_{x_2} = 1$, $y^*_{x_3} = 2$, and three rules r_1 to r_3 labels $\mathcal{Y} = \{0, 1, 2\}$ where $y_0 = 0$ and assume that these rules return the results on $\mathcal{X}^*$ shown below on the left where abstain (incorrect) labels are highlighted in blue (red). Assume that all thresholds are set to 50%. That is, each datapoint should receive at least two labels $\neq y_0$, and the accuracy for datapoints and rules is at least 50% (1 correct label if 2 non-abstain labels are returned and 2 correct labels for no abstain label). The minimum number of changes required to fulfill these constraints is 4. One possible solution for the MILP is shown below on the right with modified cells shown with a black background.*

	r_1	r_2	r_3
x_1	1	1	2
x_2	0	1	0
x_3	0	1	0

o_{ij}	$i=1$	$i=2$	$i=3$
$j=1$	2	1	2
$j=2$	0	1	1
$j=3$	2	2	0

Given the outputs o_{ij} of the MILP, we need to find a repair sequence Φ such that for $\mathcal{R}^* = \Phi(\mathcal{R}) = \{r'_1, \ldots, r'_m\}$ we have $r'_j(x_i) = o_{ij}$ for all $i \in [1, n]$ and $j \in [1, m]$. An important observation regarding this goal is that as rules operate independently of each other, we can solve this problem one rule at a time.

4 Single Rule Refinement

We now detail our approach for refining a single rule.

4.1 Rule Repair

We now formalize the problem of generating a sequence of refinement steps Φ of minimal size for a rule r_j such that for a set of datapoints and labels $\mathcal{Z} = \{(x_i, y_i)\}_{i=1}^n$, $r'_j = \Phi(r_j)$ we have $r'_j(x_i) = y_i$ for all $(x_i, y_i) \in \mathcal{Z}$. We can use this algorithm to implement the changes to rule outputs computed by the MILP from the previous section using: $\mathcal{Z} = \{(x_i, o_{ij}) \mid x_i \in \mathcal{X}^*\}$. For a sequence of refinement repairs Φ we define its cost as $rcost(\Phi) = |\Phi|$.

Definition 3 (The Single Rule Refinement Problem). *Given a rule r, a set of datapoints with desired labels $\mathcal{Z}$, and a set of allowable predicates $\mathcal{P}$, find a sequence of refinements Φ_{min} using predicates from $\mathcal{P}$ such that for $r_{fix} = \Phi(r)$ we have:*

$$\Phi_{min} = \underset{\Phi}{\text{argmin}}\ rcost(\Phi) \quad \textbf{subject to} \qquad \forall (x, y) \in \mathcal{Z} : r_{fix}(x) = y$$

Let P_{fix} denote the set of paths (from the root to a label on a leaf) in rule r that are taken by the datapoints from $\mathcal{X}$. For $P \in P_{fix}$, $\mathcal{X}_P$ denotes all datapoints from $\mathcal{X}$ for which the path is P, hence also $\mathcal{X} = \bigcup_{P \in P_{fix}} \mathcal{X}_P$. Similarly, $\mathcal{Z}_P$ denotes the subset of $\mathcal{Z}$ for datapoints x with path P in rule r. The algorithm for solving the single rule repair problem we will present in the following exploits two important properties of this problem.

Independence of path repairs. as any refinement in a minimal repair will only extend paths in P_{fix} (any other refinement does not affect the labels for $\mathcal{X}$) and refinements at any path P_1 do not affect the labels of datapoints in $\mathcal{X}_{P_2}$ for a path $P_2 \neq P_1$, a solution to the single rule repair problem can be constructed one path at a time (see Appendix C.1 for the formal proof).

Existence of path repairs. In appendix C.3, we show that path repairs with a cost of at most $|\mathcal{X}_P|$ are guaranteed to exist as long as the space of predicates is partitioning. That is, for any two datapoints x_1 and x_2 we can find a predicate p such that $p(x_1) \neq p(x_2)$. Note that for textual data, even a simple predicate space that only contains predicates of the form $w \in x$ where w is a word is partitioning as long as any two datapoints (documents in the case of text data) will differ in at least one word. Intuitively, this guarantees the existence of a repair as for any two datapoints x_1 and x_2 with $\mathcal{Z}(x_1) \neq \mathcal{Z}(x_2)$ that share the same path (and, thus, also label) in a rule r we can refine r using an appropriate predicate p with $p(x_1) \neq p(x_2)$ to assign the desired labels to x_1 and x_2.

The pseudocode for `SingleRuleRefine` is given in Algorithm 1. Given a single rule r, this algorithm determines a refinement-based repair Φ_{min} for r such that $\Phi_{min}(r)$ returns the designed label $\mathcal{Z}(x)$

Algorithm 1: SingleRuleRefine

Input : Rule r, Labelled datapoints $\mathcal{Z}$.
Output : Repair sequence Φ such that $\Phi(r)$ fixes $\mathcal{Z}$

$Y \leftarrow \emptyset, \Phi \leftarrow \emptyset$
$P_{fix} \leftarrow \{P[r,x] \mid x \in \mathcal{X}\}$
$r_{cur} \leftarrow r$
foreach $P \in P_{fix}$ **do** /* Fix one path at a time */
  /* Fix path P to return correctly labels on $\mathcal{Z}$ */
  $\mathcal{Z}_P \leftarrow \{(x,y) \mid (x,y) \in \mathcal{Z} \land P[r,x] = P\}$
  $\phi \leftarrow$ `RefinePath`$(r_{cur}, P, \mathcal{Z}_P)$
  $r_{cur} \leftarrow \phi(r_{cur})$
  $\Phi \leftarrow \Phi$.append(ϕ)
return Φ

Algorithm 2: EntropyPathRepair

Input : Rule r, Path P_{in}, Ground truth labels $\mathcal{Z}_{P_{in}}$
Output : Repair sequence Φ which fixes r wrt. $\mathcal{Z}_{P_{in}}$

$todo \leftarrow [(P_{in}, \mathcal{Z}_{P_{in}})]$
$\Phi \leftarrow []$
$r_{cur} \leftarrow r$
$\mathcal{P}_{all} \leftarrow$ `GetAllCandPredicates`$(P_{in}, \mathcal{Z}_{P_{in}})$
while $todo \neq \emptyset$ **do**
  $(P, \mathcal{Z}_P) \leftarrow pop(todo)$
  $p_{new} \leftarrow \text{argmin}_{p \in \mathcal{P}_{all}} I_G(\mathcal{Z}_P, p)$
  $Z_{\textbf{false}} \leftarrow \{(x,y) \mid (x,y) \in \mathcal{Z}_P \land \neg p(x)\}$
  $Z_{\textbf{true}} \leftarrow \{(x,y) \mid (x,y) \in \mathcal{Z}_P \land p(x)\}$
  $y_{max} \leftarrow \text{argmax}_{y \in \mathcal{Y}} \mid \{x \mid Z_{\textbf{true}}(x) = y\} \mid$
  $\phi_{new} \leftarrow$ **refine**$(r_{cur}, P, p, Y[P], y_{max})$
  $r_{cur} \leftarrow \phi_{new}(r_{cur})$
  $\Phi \leftarrow \Phi.append(\phi_{new})$
  if $\mid \mathcal{Y}_{Z_{\textbf{false}}} \mid > 1$ **then**
    $todo.push((P[r_{cur}, Z_{\textbf{false}}], Z_{\textbf{false}}))$
  if $\mid \mathcal{Y}_{Z_{\textbf{true}}} \mid > 1$ **then**
    $todo.push((P[r_{cur}, Z_{\textbf{true}}], Z_{\textbf{true}}))$
return Φ

for *all* datapoints specified in $\mathcal{Z}$ by refining one path at a time using a function `RefinePath`. The problem solved by `RefinePath` is NP-hard. Next, we introduce an algorithm implementing `RefinePath` that utilizes an information-theoretic heuristic that does not guarantee that the returned repair is minimal but works well in practice.

4.2 Path Repair: EntropyPathRepair

Given a rule r, a path $P_{in} \in P_{fix}$, and the datapoints and desired labels for this path $\mathcal{Z}_{P_{in}}$, our algorithm EntropyPathRepair avoids the exponential runtime of an optimal brute force algorithm BruteForcePathRepair that enumerates all possible refinements (see Appendix D.2). We achieve this by greedily selecting predicates that best separate datapoints with different labels at each step. To measure the quality of a split, we employ the entropy-based *Gini impurity score* I_G [18]. Given a candidate predicate p for splitting a set of datapoints and their labels at path P ($\mathcal{Z}_P$), we denote the subsets of $\mathcal{Z}_P$ generated by splitting $\mathcal{Z}_P$ based on p:

$$Z_{\textbf{false}} = \{(x,y) \mid (x,y) \in \mathcal{Z}_P \land \neg p(x)\}$$
$$Z_{\textbf{true}} = \{(x,y) \mid (x,y) \in \mathcal{Z}_P \land p(x)\}$$

Using $Z_{\textbf{false}}$ and $Z_{\textbf{true}}$ we define the score $I_G(\mathcal{Z}_P, p)$ for p:

$$I_G(\mathcal{Z}_P, p) = \frac{\mid Z_{\textbf{false}} \mid \cdot I_G(Z_{\textbf{false}}) + \mid Z_{\textbf{true}} \mid \cdot I_G(Z_{\textbf{true}})}{\mid \mathcal{Z}_P \mid}$$

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

For a set of ground truth labels Z, $I_G(Z)$ is minimal if $\mathcal{Y}_Z = \{y \mid \exists x : (x,y) \in Z\}$ contains a single label. Intuitively, we want to select predicates such that all datapoints 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 $I_G(\mathcal{Z}_P, p)$.

Algorithm 2 first determines all candidate predicates using function `GetAllCandPredicates`. Then, it iteratively selects predicates until all datapoints are assigned the expected label by the rule. For that, we maintain a queue of paths paired with a set $\mathcal{Z}_P$ of datapoints with 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 and datapoints with labels $\mathcal{Z}_P$ from the queue. We then determine the predicate p that minimizes the entropy of $\mathcal{Z}_P$. Afterward, we determine two subsets of datapoints from $\mathcal{X}_P$: datapoints fulfilling p and those that 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 ($Y[P]$ denotes the label of the node the end of P). The child at the **true** edge of the node for p is then assigned the most prevalent label y_{max} for the datapoints at this node (the datapoints from $Z_{\textbf{true}}$). Finally, unless they only contain one label, new entries for $Z_{\textbf{false}}$ and $Z_{\textbf{true}}$ are appended to the queue. As shown below, EntropyPathRepair is correct (the proof is shown in Appendix D.3).

Theorem 2 (Correctness). *Consider a rule r, ground-truth labels of a set of datapoints $\mathcal{Z}_P$, and partitioning space of predicates $\mathcal{P}$. Let Φ be the repair sequence produced by EntropyPathRepair for path P. Then we have:*

$$\forall (x,y) \in \mathcal{Z}_P : \Phi(r)(x) = y$$

5 Experiments

We evaluate the runtime of RuleCleaner and its effectiveness in improving the accuracy of rules produced by Witan [9] and LLMs. Furthermore, we evaluate the trade-offs for the three path repair algorithms we propose in this work. RuleCleaner is implemented in Python. The source code can be found here [3]. Experiments were run on Oracle Linux Server 7.9 with 2 x AMD EPYC 7742 CPUS, 128GB RAM. We evaluate RuleCleaner using *Snorkel* [26] as the PWSS. We evaluate both the runtime and the quality of the refinements produced by our system with respect to several parameters.

Datasets and rules. The datasets used in the experiments are listed in Table 2. Note that because of the complexity of and the nature of multi-class labels, the LFs we used for *CmPt* are all from [34], which has 26 LFs. We give a brief description of each dataset:

Dataset	#row	#word	$\mathcal{Y}$	$\#LFs_{witan}$	$\#LFs_{llm}$
Amazon	200000	68.9	pos/neg	15	23
AGnews	60000	37.7	busi/tech	9	21
PP	54476	55.8	physician/prof	18	20
IMDB	50000	230.7	pos/neg	7	20
FNews	44898	405.9	true/false	11	20
Yelp	38000	133.6	neg/pos	8	20
PT	24588	62.2	prof/teacher	7	19
CmPt	16075	27.9	10 relations	-	-
PA	12236	62.6	painter/architect	10	18
Tweets	11541	18.5	pos/neg	16	18
SMS	5572	15.6	spam/ham	17	16
MGenre	1945	26.5	action/romance	10	14

Table 2: LF dataset statistics.

Amazon: product reviews from Amazon and their sentiment label [16]. *AGnews*: categorized news articles from AG's corpus of news articles. For this dataset, we chose a binary class version from [9]. *PP*: descriptions of biographies, each labeled as a physician or a professor [8]. *IMDB*: IMDB movie reviews [22]. *FNews*: Fake news identification [1]. *Yelp*: Yelp reviews [39]. *PT*: descriptions of individuals, each labeled as a professor or a teacher.[8]. *PA*: descriptions of individuals, each labeled a painter or an architect. [8]. *Tweets*: classification of tweets on disasters [23]. *SMS*: classification of SMS messages [2]. *MGenre*: movie genre classification based on plots [31]. *CmPt*: chemical-protein relationship classification from [19]. Unless stated otherwise, the experiments in this section are run with EntropyPathRepair. We present a detailed evaluation of these all path repair algorithms in Section 5.4.

5.1 Refining labelling functions

In this experiment, we investigate the effects of several parameters on the performance and quality of the rules repaired with RuleCleaner for several datasets.

Varying the number of labeled examples. We evaluate how the size of $\mathcal{X}^*$ affects global accuracy. Global accuracy is defined as the accuracy of the labels predicted by the trained Snorkel model using the LFs compared to the ground truth labels. Given the limited scalability of MILP solvers in the number of variables, we used at most $|\mathcal{X}^*| = 150$ datapoints. The labeled datapoints are randomly sampled from $\mathcal{X}$, with 50% correct predictions by Snorkel and 50% wrong predictions within each sample. The reason for sampling in this manner is to provide sufficient evidence for correct predictions and predictions that need to be adjusted. Even if we have no control over the creation of $\mathcal{X}^*$, we can achieve this by sampling from a larger set of labeled examples. Figure 5a shows the global accuracy after retraining a Snorkel model with the rules refined by RuleCleaner. The repairs improve the global accuracy on 8 out of 9 datasets, even for very small sample sizes. The variance of the new global accuracy also decreases as the amount of labeled examples increases.

Varying thresholds. We evaluate the relationships between τ_{acc}, τ_E, τ_{racc} and new global accuracy. We used *Tweets* with 20 labeled examples. The details of the experiments and analysis are shown in Appendix E.2. Based on the experiments, we recommend setting all of the thresholds to ~ 0.7.

5.2 Runtime

Runtime breakdowns for a subset of the experiments from Section 5.1 are shown in Figure 6. For the breakdown of the other datasets, please refer to Appendix E.1. The total runtime increases as we increase the amount of labeled examples. The runtime of the refinement step is strongly correlated with the average length of the texts in the input dataset, i.e., the longer the average text length (as presented in *average # words* in Table 2), the more time is required to select the best predicate using EntropyPathRepair.

It is important to note that the runtime changes for both *snorkel run after refinement* and *MILP* do not exhibit a strictly linear pattern. The reason for such non-linearity arises from the fact that the labeled datapoints are randomly sampled from $\mathcal{X}$, and the complexity of solving the MILP problem depends on the sparsity of the solution space. The same reason applies for retraining with Snorkel using the refined rules. Some sets of labeled examples result in more complex rules even when the sample size is small, increasing the time required for Snorkel to fit a model.

5.3 LLMs vs RuleCleaner

In this section, we compare our approach against LLMs. We consider three setups: (i) using the LLM as a labeler (without any use of LFs) and (ii) using the LLM to generate LFs based on with labeled examples; and (iii) using the LLM to repair labeling functions (refer to Appendix F). For (ii), we then investigate whether RuleCleaner can successfully improve the LFs generated by the LLM.

The LLM as a Labeler. We compare the performance and quality of RuleCleaner with Snorkel and LLMs as a standalone labeler. In this experiment, we use `GPT-4o` and `Llama-3-8B-instruct` (Llama 3 8B), using a zero-shot prompt to describe the task. Both LLMs receive the possible labels along with the sentences, but not the LFs. To optimize API usage, we batch 10 datapoints per call for `GPT-4o`, whereas for `Llama-3-8B-instruct`, we label one sentence per call to maintain response validity. Experiments were conducted on a Mac Studio (Apple M2 Max, 12-core CPU, 64GB unified memory, SSD storage). The setup with RuleCleaner (RC for short shown in the plot) is the default setup from Section 5.1. The runtime and quality comparisons are presented in Figure 4. We did set a 48-hour time limit, including only datasets where all three competitors completed the task within this time limit. `GPT-4o` achieves the highest accuracy in 6 out of 7 datasets. However, for dataset *CmPt*, RuleCleaner with Snorkel (37.9%) outperforms `GPT4-o` (27.4%). Upon further analysis, we speculate that GPT-4o's poor performance could stem from the specialized terminology and complex domain knowledge required for labeling in ChemProt. Unlike general-purpose datasets where LLMs excel, ChemProt contains highly domain-specific biomedical entity interactions, which may be challenging for zero-shot prompting. Using LLMs as an end-to-end labler comes at an unacceptable computational/monetary cost. The experiments with `GPT-4o` did cost $254.42 for API usage. While we do not know the precise computational resources that were required, our local experiments with `Llama-3-8B-instruct`, a significantly smaller model that also cannot compete with RuleCleaner in terms of accuracy on most datasets, demonstrate the high computational cost of using an LLM for this purpose. In fact, RuleCleaner is ~ 2 to ~ 4 orders of magnitude faster than `Llama-3-8B-instruct`. In summary, while large models like `GPT-4o`, but not smaller models like `Llama-3-8B-instruct`, can achieve high accuracy as labelers, this comes at a prohibitively high computational cost. RuleCleaner outperforms `Llama-3-8B-instruct` in terms of accuracy on most datasets and between 56x to 1,312x in terms of runtime. Furthermore, LFs have the additional advantage of being inherently interpretable which is not the case for labeling with LLMs.

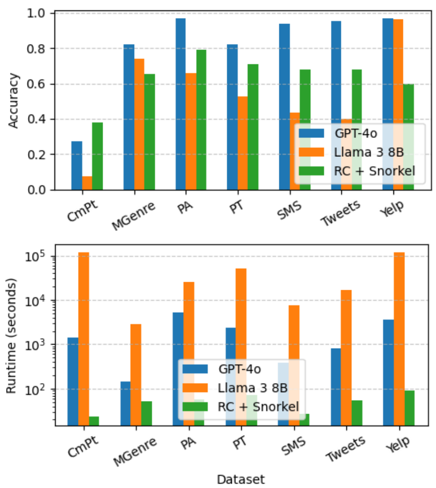


Figure 4: Comparison of 3 labelers

The LLM as a LF Generator. We investigate whether LLMs can generate effective LF and whether RuleCleaner can fix such functions to improve their accuracy. The use of LLMs for generating LFs has been explored in recent years [12, 20]. We adapted these existing methods to generate LFs. For an example prompt, see Appendix E.3. In each prompt, we sample a small set of sentences along with their ground truth labels and provide two LF templates based on keywords and regular expressions. Since the number of LFs is determined by how many times we query the LLM, we scale the number of LFs logarithmically in the dataset size. The number of LFs produced for each dataset is reported as $\#LFs_{llm}$ in Table 2.

To evaluate the quality of generated LFs_{witan}, we use the same experimental setup described in Section 5.1. The results are shown in Figure 5b. Out of the nine datasets used in this experiment, three exhibit higher original global accuracy after training with Snorkel compared to training with LFs_{witan}. After refinement using labeled datapoints, we observe improvements in eight out of nine datasets.

Two datasets, *SMS* and *Tweets*, show significant improvement. Upon further inspection, we observed that some of the LFs generated by the LLM assign incorrect labels. For example, in *SMS*, one LF is defined as follows: `return HAM if any(x in text for x in ['sorry', 'please', 'home', 'call', 'message', 'buy', 'talk', 'problem', 'help', 'ask'])else ABSTAIN`. Some of these keywords, such as "call" and "message", frequently appear in spam messages. RuleCleaner successfully refines this LF by correcting its label assignment, thereby improving accuracy.

5.4 Path Repair Algorithms

Next, we compare the three path repair algorithms discussed in Section 4.2. In this experiment, we used the *Tweets* dataset and randomly selected between 2 and 10 labeled examples. We show *cost*, the repaired rule size in terms of number of nodes and the runtime in Figure 7. Note that for 10 labeled examples, the repair runtime for BruteForcePathRepair exceeded the time limit we set for this experiment (600 secs) and, thus, is absent from the plot. The runtime of BruteForcePathRepair is prohibitory large even for just 8 datapoints. EntropyPathRepair achieves almost the same repair cost as BruteForcePathRepair while being significantly faster. While GreedyPathRepair is the fastest algorithm, this comes that the cost of a significantly higher repair cost.

It is obvious that the runtime for BruteForcePathRepair is significantly higher than the other 2 algorithms. GreedyPathRepair is the fastest since it picks the first available predicate without any additional computation. In terms of rule sizes after the repairs, BruteForcePathRepair generates the smallest rules since it will exhaustively enumerate all the possible solutions and is guaranteed to find the smallest possible solution. It is worth noting that EntropyPathRepair is only slightly worse than BruteForcePathRepair while being significantly faster.

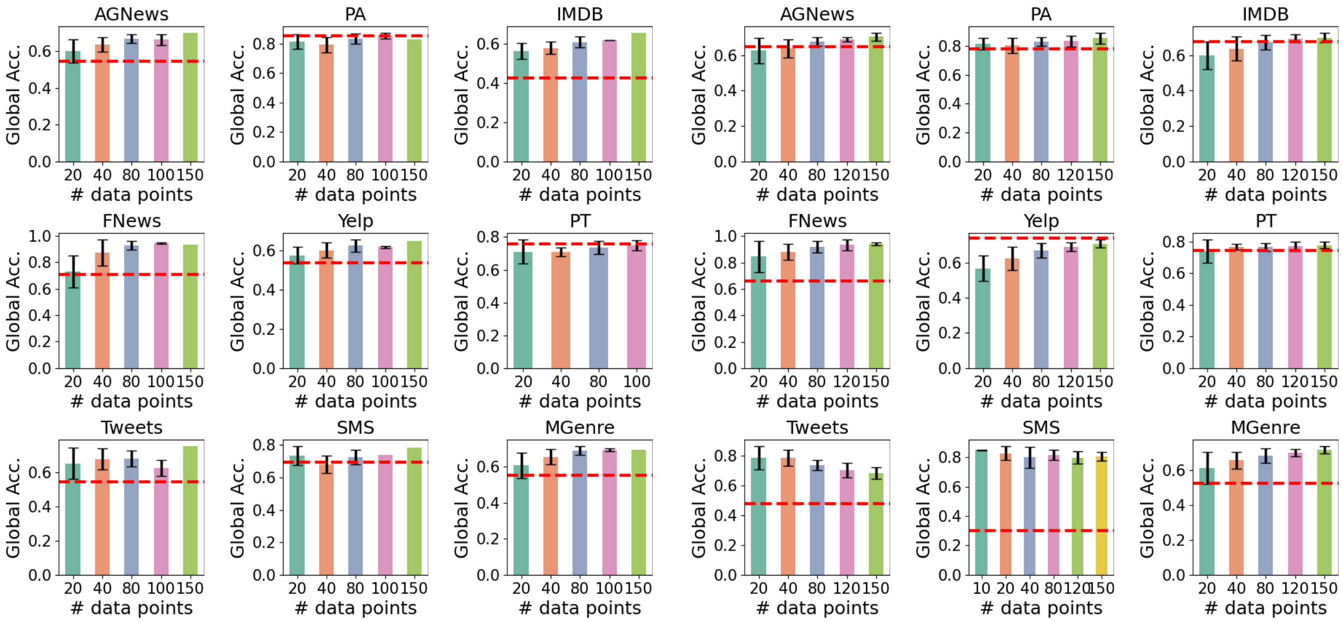
6 Related Work

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

Programmatic weak supervision (PWS). Weak supervision is a general technique of learning from noisy supervision signals, widely applied for data labeling to generate training data [26, 27, 31] (the main use case we target in this work), data repair [28], and entity matching [25]. Its main advantage is reducing the effort of creating training data without ground truth labels. The programmatic weak supervision paradigm pioneered in Snorkel [26] has the additional advantage that the labeling rules 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 PWS proves effective, asking human annotators to create a large set of high-quality labeling functions requires domain knowledge, programming skills, and time. As a result, the automatic generation or improvement of labeling heuristics has received much attention from the research community. Some existing methods demand interactive user feedback in creating labeling functions [5, 10]. *Witan* [9] asks a domain expert to select the automatically generated LFs and assign labels to the LFs. While the LFs produced by Witan are certainly useful, we demonstrate in our experimental evaluation that applying RuleCleaner to Witan LFs can significantly improve accuracy. Other methods generate LFs without requiring user annotations. *Snuba* [31] fits classification models, such as decision trees and logistic regressions, as LFs on a small labeled training set, followed by a pruning process to determine which LFs for final use. *Datasculpt* [12] prompts a large language model (LLM) with a small set of labeled training data and keyword- or pattern-based LFs as in-context examples. The LLM then generates LFs for unlabeled examples based on this input. Evaporate [4] uses an LLM to generate data extraction functions, and then it applies weak supervision to filter out low-quality functions and aggregate the results.

Hsieh et al. [17] propose *Nemo*, a framework for selecting data to guide users in developing LFs. It estimates the likelihood of users



(a) Change in Global accuracy, WITAN LFs

(b) Change in Global accuracy, GPT-4 generated LFs

Figure 5: Impact of repairs on global accuracy (the red dotted line is accuracy before the repair) for LFs generated by Witan and GPT-4. We vary the number of labeled examples $\mathcal{X}^*$.

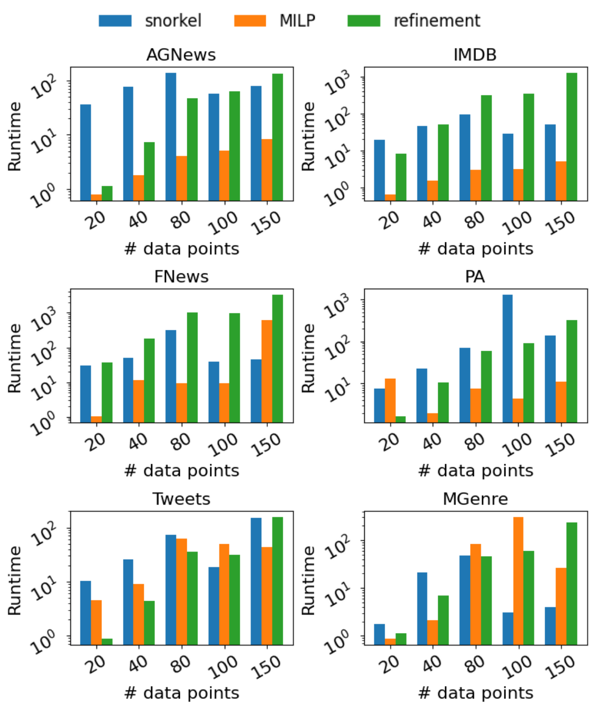


Figure 6: Runtime, varying the size of $\mathcal{X}^*$.

proposing specific LFs using a utility metric for LFs and a model of user behavior. Nemo tailors LFs to the neighborhood of the data, assuming that user-developed LFs are more accurate for data similar to those used for LFs creation. However, unlike RuleCleaner, Nemo lacks a mechanism for the user to provide feedback on the labeling results, preventing the automatic deletion and refinement

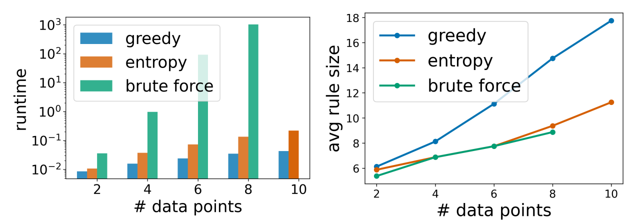


Figure 7: Comparing path repair algorithms

of LFs. *ULF* [30] is an unsupervised system for adjusting LFs assignment for unlabeled samples (instead of repairing them) using k-fold cross-validation, extending previous approaches addressing labeling errors [24, 32].

Explanations for weakly supervised systems. 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 [5, 7, 13, 33, 36–38]. For instance, [36] uses influence function to identify LFs responsible for erroneous labels; WeShap [13] measures the shapley value of LFs to rank and prune LFs. However, most of this work has stopped short of repairing the rules in a PWSS and, thus, are orthogonal to our work. Still, explanations provided by such systems might guide users in selecting what datapoints to label. People have also studied using human-annotated natural language explanations to build LFs [14].

7 Conclusions and Future Work

We study repairs for LFs in PWS based on a small set of labeled examples. Our algorithm is highly effective in improving the accuracy of PWSSs by improving rules created by a human expert or automatically discovered by a system like Witan [9]. In future work, we will explore the application of our rule repair algorithms to other tasks that can be modeled as PWSS, e.g., information extraction based on user-provided rules [21, 29].

References

[1] Hadeer Ahmed, Issa Traoré, and Sherif Saad. 2018. Detecting opinion spams and fake news using text classification. *Secur. Priv.* 1, 1 (2018).

[2] Tiago A. Almeida, José María Gómez Hidalgo, and Akebo Yamakami. 2011. Contributions to the study of SMS spam filtering: new collection and results. In *ACM Symposium on Document Engineering*. ACM, 259–262.

[3] anonymous. [n. d.]. Rule Cleaner Source Code. https://anonymous.4open.science/r/rulecleaner_kdd_anonymized-6D6A.

[4] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. *Proceedings of the VLDB Endowment* 17, 2 (2023), 92–105.

[5] Benedikt Boecking, Willie Neiswanger, Eric Xing, and Artur Dubrawski. 2021. Interactive Weak Supervision: Learning Useful Heuristics for Data Labeling. In *International Conference on Learning Representations.*

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

[7] Bradley Butcher, Miri Zilka, Darren Cook, Jiri Hron, and Adrian Weller. 2023. Optimising Human-Machine Collaboration for Efficient High-Precision Information Extraction from Text Documents. *arXiv preprint arXiv:2302.09324* (2023).

[8] Maria De-Arteaga, Alexey Romanov, Hanna M. Wallach, Jennifer T. Chayes, Christian Borgs, Alexandra Chouldechova, Sahin Cem Geyik, Krishnaram Kenthapadi, and Adam Tauman Kalai. 2019. Bias in Bios: A Case Study of Semantic Representation Bias in a High-Stakes Setting. In *FAT*, danah boyd and Jamie H. Morgenstern (Eds.). 120–128.

[9] 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.

[10] Sainyam Galhotra, Behzad Golshan, and Wang-Chiew Tan. 2021. Adaptive Rule Discovery for Labeling Text Data. In *SIGMOD*. 2217–2225.

[11] Igor Griva, Stephen G Nash, and Ariela Sofer. 2008. *Linear and Nonlinear Optimization 2nd Edition.* SIAM.

[12] Naiqing Guan, Kaiwen Chen, and Nick Koudas. 2025. DataSculpt: Cost-Efficient Label Function Design via Prompting Large Language Models. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025.* OpenProceedings.org, 226–232.

[13] Naiqing Guan and Nick Koudas. 2024. Weshap: Weak Supervision Source Evaluation With Shapley Values. *CoRR* abs/2406.11010 (2024). arXiv:2406.11010

[14] Braden Hancock, Martin Bringmann, Paroma Varma, Percy Liang, Stephanie Wang, and Christopher Ré. 2018. Training classifiers with natural language explanations. In *ACL*, Vol. 2018. 1884.

[15] 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.

[16] Ruining He and Julian J. McAuley. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *WWW*, Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao (Eds.). 507–517.

[17] 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.

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

[19] Martin Krallinger, Obdulia Rabal, Saber A Akhondi, Martın Pérez Pérez, Jesús Santamaría, Gael Pérez Rodríguez, Georgios Tsatsaronis, Ander Intxaurrondo, José Antonio López, Umesh Nandal, et al. 2017. Overview of the BioCreative VI chemical-protein interaction Track. In *Proceedings of the sixth BioCreative challenge evaluation workshop*, Vol. 1. 141–146.

[20] Chenjie Li, Dan Zhang, and Jin Wang. 2024. LLM-assisted Labeling Function Generation for Semantic Type Detection. In *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases, VLDB 2024, Guangzhou, China, August 26-30, 2024.*

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

[22] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Association for Computational Linguistics: Human Language Technologies*, Dekang Lin, Yuji Matsumoto, and Rada Mihalcea (Eds.). 142–150.

[23] Hussein Mouzannar, Yara Rizk, and Mariette Awad. 2018. Damage Identification in Social Media Posts using Multimodal Deep Learning. In *International Conference on Information Systems for Crisis Response and Management*, Kees Boersma and Brian M. Tomaszewski (Eds.). ISCRAM Association.

[24] 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.

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

[26] 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.

[27] 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.

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

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

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

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

[32] 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.

[33] Peilin Yu and Stephen Bach. 2023. Alfred: A System for Prompted Weak Supervision. *arXiv preprint arXiv:2305.18623* (2023).

[34] Yue Yu, Simiao Zuo, Haoming Jiang, Wendi Ren, Tuo Zhao, and Chao Zhang. 2020. Fine-tuning pre-trained language model with weak supervision: A contrastive-regularized self-training approach. *arXiv preprint arXiv:2010.07835* (2020).

[35] Jieyu Zhang, Cheng-Yu Hsieh, Yue Yu, Chao Zhang, and Alexander Ratner. 2022. A Survey on Programmatic Weak Supervision. *CoRR* abs/2202.05433 (2022). arXiv:2202.05433

[36] Jieyu Zhang, Haonan Wang, Cheng-Yu Hsieh, and Alexander J. Ratner. 2022. Understanding Programmatic Weak Supervision via Source-aware Influence Function. In *NeurIPS*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.).

[37] Jieyu Zhang, Haonan Wang, Cheng-Yu Hsieh, and Alexander J Ratner. 2022. Understanding programmatic weak supervision via source-aware influence function. *Advances in Neural Information Processing Systems* 35 (2022), 2862–2875.

[38] Xiaoyu Zhang, Xiwei Xuan, Alden Dima, Thurston Sexton, and Kwan-Liu Ma. 2023. LabelVizier: Interactive Validation and Relabeling for Technical Text Annotations. In *2023 IEEE 16th Pacific Visualization Symposium (PacificVis)*. IEEE, 167–176.

[39] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level Convolutional Networks for Text Classification. In *NeurIPS*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 649–657.

A Translating Labeling Functions Into Rules

In this section, we detail simple procedures for converting labeling functions to our rule representation (Definition 4).

Definition 4 (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 datapoint $x \in \mathcal{X}$ 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 datapoint 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.

Algorithm 3: Convert LF to Rule

Input : The code C of a LF f
Output : r_f, a rule representation of f

$V, E = \emptyset, \emptyset$;
$r_f \leftarrow (V, E)$;
Let C be the source code of f;
LF-to-Rule(r, C):
  if $C =$ `if cond: B1 else: B2` $\wedge$ `ispure`*(cond)* $\wedge$ `purereturn`*(B1)* $\wedge$ `purereturn`*(B2)* **then**
    $v_{true} \leftarrow$ `LF-to-Rule` $(B1)$
    $v_{false} \leftarrow$ `LF-to-Rule` $(B2)$
    $n_{root} \leftarrow$ `Pred-To-Rule` $(cond, v_{false}, v_{true})$
  else if $C =$ `return y` $\wedge\ y \in \mathcal{Y}$ **then**
    $n_{root} = y$
  else
    $n_{root} \leftarrow$ `Translate-BBox` (C)
  return n_{root}
Pred-to-Rule$(\psi, n_{false}, n_{true})$:
  if $\psi = \psi_1 \vee \psi_2$ **then**
    $n_{left} \leftarrow$ `Pred-to-Rule` $(\psi_2, n_{false}, n_{true})$
    $n_{root} \leftarrow$ [ψ_1; **false** → v_{left}; **true** → v_{true}]
  else if $\psi = \psi_1 \wedge \psi_2$ **then**
    $v_{right} \leftarrow$ `Pred-to-Rule` $(\psi_2, v_{false}, v_{true})$
    $n_{root} \leftarrow$ [ψ_1; **false** → v_{false}; **true** → v_{true}]
  else
    $n_{root} \leftarrow$ [ψ; **false** → v_{false}; **true** → v_{true}]
  return n_{root}
Translate-BBox(C) :
  $v_{root} = y_0$;
  for $y \in \mathcal{Y}/\{y_0\}$ **do**
    $n_{root} \leftarrow$ [$C(v) = y$; **false** → v_{root}; **true** → y]
  return n_{root}

As mentioned before, we support arbitrary labeling functions written in a general-purpose programming language. Our implementation of the translation into rules is for Python functions, as supported in Snorkel. Detecting the if-then-else rule structure and logical connectives that are supported in our rules for an arbitrary Python function is undecidable in general (can be shown through a reduction from program equivalence). However, our algorithm can still succeed in any LF written in Python if we can live if we treat every code block that our algorithm does not know how to compose as a black box that we call repeatedly on the input and compare its output against all possible labels from $\mathcal{Y}$. In the worst case, we would wrap the whole LF in this way. Note that this does not prevent us from refining such labeling functions. However, there are several advantages in decomposing an LF into a tree with multiple predicates: (i) such a rule will make explicit the logic of the LF and, thus, may be easier to interpret by a use,r and (ii) during refinement we have more information to refine the rule as there may be multiple leaf nodes corresponding to a label y, each of which corresponds to a different set of predicates evaluating to true.

Our translation algorithm knows how to decompose a limited number of language features into predicates of a rule. As mentioned above, any source code block whose structure we cannot further decompose will be treated as a blackbox and will be wrapped as a predicate whose output we compare against every possible label from $\mathcal{Y}$. Furthermore, when translating Boolean conditions, i.e., the condition of an if statement, we only decompose expressions that are logical connectives and treat all other subexpressions of the condition as atomic. While this approach may sometimes translate parts of a function's code into a black-box predicate, most LFs we have observed in benchmarks and real applications of data programming can be decomposed by our approach. Nonetheless, our approach can easily be extended to support additional structures if need be.

Translating labeling functions. Pseudo code for our algorithm is shown in Algorithm 3. Function `LF-to-Rule` is applied the code `C` of the body of labeling function f. We analyze code blocks using the standard libraries for code introspection in Python, i.e., Python's AST library. If the code is an if then else condition (for brevity we do not show the the case of an if without else and other related cases) that fulfills several additional requirements, then we call `LF-to-Rule` to generate rule trees to the if and the else branch. Afterwards, we translate the condition using function `Pred-To-Rule` described next that takes as input a condition ψ and the roots of subtrees to be used when the condition evaluates to false or true, respectively. For this to work, several conditions have to apply: (i) both `B1` and `B2` have to be pure, i.e., they are side-effect free, and return a label for every input. This is checked using function `purereturn`. This is necessary to ensure that we can translate `B1` and `B2` into rule fragments that return a label. Furthermore, `cond` has to be pure (checked using function `ispure`). Note that both `ispure` and `purereturn` have to check a condition that is undecidable in general. Our implementations of these functions are is sound, but not complete. That is, we may fail to realize that a code block is pure (and always returns a label in case of `purereturn`, but will never falsely claim a block to have this property.

If the code block returns a constant label y, then it is translated into a rule fragment with a single node y. Finally, if the code block `C` is not a conditional statement, then we fall back to use our black box translation technique (function `Translate-BBox` explained below).

Translating predicates. `Pred-to-Rule`, our function for translating predicates (Boolean conditions as used in if statements), takes as input a condition ψ and the roots of two rule subtrees (v_{false} and v_{true}) that should be used to determine an inputs label based on whether ψ evaluates to false (true) on the input. The function checks whether the condition is of the form $\psi_1 \vee \psi_2$ or $\psi_1 \wedge \psi_2$. If

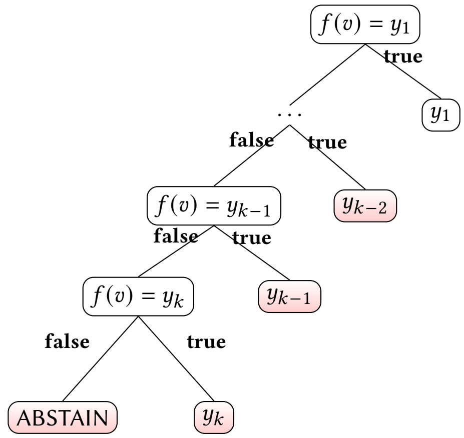
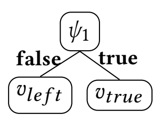
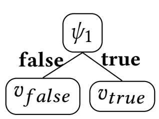


Figure 8: Translating a blackbox function

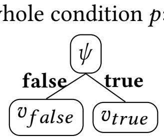
that is the case, we decompose the condition and create an appropriate rule fragment implementing the disjunction (conjunction). For disjunctions $\psi_1 \lor \psi_2$, the result of v_{true} should be returned if ψ_1 evaluates to true. Otherwise, we have to check ψ_2 to determine whether v_{false}'s or v_{true}'s result should be returned. For that we generate a rule fragments as shown below where v_{left} denote the root of the rule tree generated by calling Pred-to-Rule to translate ψ_2.



The case for conjunctions is analog. If ψ_1 evaluates to false we have to return the result of v_{false}. Otherwise, we have to check ψ_2 to determine whether to return v_{false}'s or v_{true}'s result. This is achieved using the rule fragment shown below where v_{right} denotes the root of the tree fragment generated for ψ_2 by calling Pred-to-Rule on ψ_2.



If ψ is neither a conjunction nor disjunction, then we just add predicate node for the whole condition p:



Translating blackbox code blocks. Function Translate-BBox is used to translate a code block C that takes as input a datapoint x (assigned to variable v), treating the code block as a black box. This function creates a rule subtree that compares the output of C on v against every possible label $y \in \mathcal{Y}$. Each such predicate node has a true child that is y, i.e., the rule fragment will return y iff $C(x) = y$. Note that this translation can not just be applied to full labeling functions, but also code blocks within a labeling function's code that our algorithm does not know how to decompose into predicates. Figure 8 shows the structure of the generated rule tree produced by Translate-BBox for a set of labels $\mathcal{Y} = \{y_1, \ldots, y_k, \text{ABSTAIN}\}$ where ABSTAIN is the default label (y_0).

As mentioned above, this translation process produces a valid rule tree r_f that is equivalent to the input LF f in the sense that it returns the same results as f for every possible input. Furthermore, the translation runs in PTIME. In fact, it is linear in the size of the input.

Proposition 2. *Let f be a python labeling function and let r_f =* LF-to-Rule(f). *Then for all datapoints x we have*

$$f(x) = r_f(x)$$

The function LF-to-Rule*'s runtime is linear in the size of f.*

Proof Sketch. The result is proven through induction over the structure of a labeling function. □

Example 4 (Translation of Complex Labeling Functions). *Consider the labeling function implemented in Python shown below. This function assigns label* POS *to each datapoint (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. Our translation algorithm identifies that this function implements an if-then-else condition. The condition if pure and both branches are pure and return a label for every input. Thus, we translate both branches using* LF-to-Rule *and the condition using* Pred-to-Rule. *The if branch is translated into a rule fragment with a single node* POS. *The else branch contains assignments that our approach currently does not further analyze and, thus, is treated as a blackbox by wrapping it in a new function, say* blackbox_lf *(shown below), whose result is compared against all possible labels. Finally, the if statement's condition is translated with* Pred-to-Rule. *We show the generated rule in Figure 9. Note that technically the comparison of the output of* blackbox_lf *with label* NEG *is unnecessary as this function does not return this label for any input. This illustrates the trade-off between adding additional complexity to the translation versus simplifying the generated rules.*

```
def complex_lf(v):
  if ['star', 'stars'].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
```

B Proof of Theorem 1

We now prove the hardness of the rule repair problem. Afterward, we demonstrate that even repairing a single rule with refinement to produce a specific result on a set of datapoints NP-hard if the goal is to minimize the number of predicates that are added to the rule.

Proof of Theorem 1. We prove the theorem through a reduction from the set cover problem. Recall that the set cover problem is: given a set $U = \{e_1, \ldots, e_n\}$ and sets S_1 to S_m such that $S_i \subseteq U$ for each i, does there exist $i_1, \ldots, i_k$ such that $\bigcup_{j=1}^{k} S_{i_j} = U$.

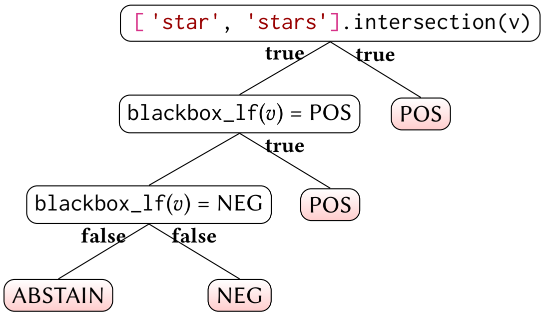


Figure 9: Translating a LF wrapping parts into a blackbox function

Based on an instance of the set cover problem, we construct an instance of the rule repair problem as follows:

- The instance $\mathcal{X}^*$ contains the following datapoints:
  - b_i for $i \in [1, M]$ where M is a large constant, say $M = 3 \cdot n \cdot m^2$ and $\forall i : b_i \notin U$.
  - a_{ij} for $i \in [1, m]$ and $j \in [1, n \cdot m]$ and $\forall i, j : a_{ij} \notin U$
  - e_i for $e_i \in U$
- Labels $\mathcal{Y} = \{out, in\}$
- Rules $\mathcal{R} = \{r_i\}$ for $i \in [1, m]$ where r_i has a single predicate $p_i : v \in S_i \lor \exists j : v = a_{ij}$ with two children that are leaves with labels $C_{\textbf{false}}(p_i) = out$ and $C_{\textbf{true}}(p_i) = out$ (Figure 10). Hence, initially, any datapoint will receive the label out from each rule r_i. Furthermore, the is an additional rule r_{xin} with a single predicate $p_{xin} : \exists i, j : v = a_{i,j}$ with $C_{\textbf{true}}(p_{xin}) = in$ and $C_{\textbf{false}}(p_{xin}) = out$.
- The expected labels $\mathcal{Z}$ are:

$$\mathcal{Z}(x) = \begin{cases} in & \text{if } x \in U \\ out & \text{if } \exists i : x = b_i \\ out & \text{if } \exists i, j : x = a_{ij} \end{cases}$$

- The space of predicates is $\mathcal{P} = \{(v = v)\}$, i.e., a single predicate that returns **true** on all inputs.
- The thresholds are set as follows:
  - $\tau_E = \frac{1}{|\mathcal{X}^*|}$, that is each datapoint has to receive a correct label by at least one of the rules.
  - $\tau_{racc} = \frac{M}{n+(n \cdot m^2)+M}$
  - $\tau_E = 1$, i.e., no rule can abstain on any datapoint.

We claim that there exists a set cover of size k or less iff there exists a rule repair Φ of $\mathcal{R}$ with a cost of less than or equal to $n \cdot m + k \cdot (n \cdot m)$. Before proving this statement, we first state several properties that any solution to the instance of the rule repair problem defined above is guaranteed to fulfill. First off, observe that as all datapoints fulfill the predicate $v = v$, a refinement can only switch the label of every datapoint that fulfills (does not fulfill) the predicate of a rule r_i's root node. Furthermore, observe that for each rule r_i, the left child of the root cannot be refined to assign a different label to any datapoint not fulfilling the root's predicate $v \in S_i \lor \exists j : v = a_{ij}$ as in particular every b_i does not fulfill the predicate and assigning the label in to all b_i's causes the rule to not fulfill the accuracy threshold $\text{Acc}(r_i) \geq \tau_{racc}$ as there are $M > n + n \cdot m^2$ datapoints in $\{b_i\}$. That is, for every set S_i represented by rule r_i, we either have the choice to refine the right child of the root by replacing it with a predicate $v = v$ with a true child in (and any label for the false child as no datapoint will end up at the false child). We will interpret refining a rule r_i as including the set S_i in the set cover. Refining rule r_{xin} has cost $n \cdot m^2$ (all datapoints a_{ij} receive an incorrect label) and, thus, no solution can refine this rule. In any solution, for each e_i, there has to be at least one refined rule r_j for $e_i \in S_j$ to ensure that the $\text{Acc}(e_i) \geq \frac{1}{m+1}$ (this datapoint receives at least one in label). All other datapoints already fulfill $iaccuracy$ through rule r_{xin} that, as explained before, cannot be refined in any solution. In summary, any solution for the rule repair problem encodes a set cover that includes every S_i such that r_i got refined.

$\Rightarrow$: Assume that there exists a set cover $S_{i_1}, \ldots, S_{i_k}$ of size k. We have to show that there exists a minimal repair Φ of cost $cost(\Phi) \leq n \cdot m + k \cdot (n \cdot m)$. We construct this repair by refining r_{i_j} for each S_{i_j}. As $S_{i_1}, \ldots, S_{i_k}$ is a set cover of size k, the repair cost of Φ is $\sum_{j=1}^{k} |S_{i_j}| + n \cdot m = \left(\sum_{j=1}^{k} |S_{i_j}|\right) + k \cdot (n \cdot m) \leq n \cdot m + k \cdot (n \cdot m)$, because for each refined rule r_i, the output of r_i on all datapoints $e_j \in S_i$ and all datapoints a_{ij} for $\forall j \in [1, n \cdot m]$ are changed resulting in a cost of $|S_i| + n \cdot m$ per refined rule and the claimed cost for $cost(\Phi)$.

$\Leftarrow$: Assume that there exists a rule repair Φ of cost less than or equal to $n \cdot m + k \cdot (n \cdot m)$. Recall that for each S_i there exist $n \cdot m$ datapoints a_{ij} that fulfill the root predicate of rule r_i. That is, if r_i is refined in a solution then all there are $n \cdot m + |S_i|$ datapoints (all a_{ij} for $j \in [1, n \cdot m]$ and e_j for all $e_j \in S_i$) whose label is updated. As $\sum_{j:r_j \text{ was refined}} |S_j| \leq n \cdot m$, $cost(\Phi) \leq n \cdot m + k \cdot (n \cdot m)$ implies that at most k rules are refined by Φ and, thus, $|\{S_i \mid r_i \text{ was refined by } \Phi\}| \leq k$ and $\{S_i \mid r_i \text{ was refined by } \Phi\}$ is a set cover of size $\leq k$. □

Theorem 1 states that the rule repair problem is NP-hard. However, as we demonstrate experimentally, for rule sets of typical size we can determine exactly how to change the outputs of rules such that rules with this updated output are a solution to the rule repair problem in sec. 3. Then we can apply any one of our single rule repair algorithms to determine a refinement repair sequence Φ such that the updated rules $\Phi(\mathcal{R})$ return the desired outcome on $\mathcal{X}^*$. This is guaranteed to succeed based on Lemma 3 that shows that as long as the predicate space is partitioning, then any change to a rule's output on a set of datapoints can be achieved by a refinement repair. However, achieving such an update with a minimal number of refinement steps (new predicates added to the rules) is computationally hard as the following theorem shows.

Theorem 3 (Minimal Single Rules Refinement Repairs). *Consider a rule r and a set of datapoints $\mathcal{X}^*$ and labels $\mathcal{Z} : \mathcal{X}^* \to \mathcal{Y}$. The following problem is NP-hard:*

$$\underset{\Phi:\forall x \in \mathcal{Z}:\Phi(r)(x)=\mathcal{Z}(x)}{\text{argmin}} \quad rcost(\Phi)$$

Proof. 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 an instance of the rule repair problem as follows:

- The instance $\mathcal{X}^*$ has $n+1$ datapoints $\{e_1, \ldots, e_n, b\}$ where $b \notin U$.
- Labels $\mathcal{Y} = \{out, in\}$
- Rules $\mathcal{R} = \{r\}$ where r has a single predicate $p : (v = v)$ (i.e., it corresponds to truth value *true*) with two children that are leaves with labels $C_{\mathbf{false}}(p) = in$ and $C_{\mathbf{true}}(p) = out$ (Figure 11). Hence, initially, any assignment will end up in $C_{\mathbf{true}}(p) = out$.
- The ground truth labels $\mathcal{Z}$ assigns a label to every datapoint as shown below.

$$\mathcal{Z}(e) = \begin{cases} in & \text{if } e \neq b \\ out & \text{otherwise}(e = b) \end{cases}$$

- 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).
- The space of predicates is $\mathcal{P} = \{v \in S_i \mid i \in [1, m]\}$

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.

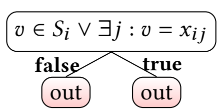


Figure 10: Rule r_i used in Appendix B

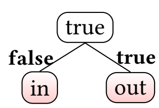


Figure 11: The rule representation of rule r used in Theorem 3

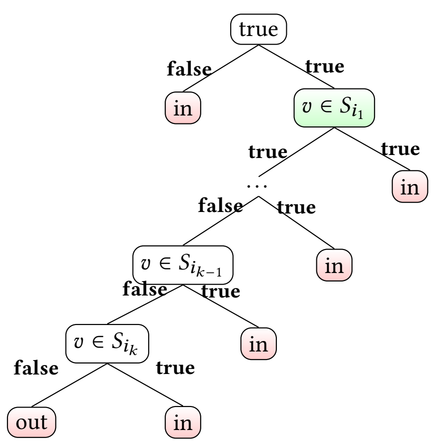


Figure 12: The rule representation of repair of rule r in Appendix B

(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 $C_{\mathbf{false}}(p_{i_j}) = p_{i_{j+1}}$ unless $j = k$ in which case $C_{\mathbf{false}}(p_{i_j}) = out$; for all j, $C_{\mathbf{true}}(p_{i_j}) = in$. The rule tree for this rule is shown in Figure 12 The repair sequence $\Phi = \phi_1, \ldots, \phi_k$ has cost k. Here ϕ_i denotes the operation that introduces p_{i_j} .

It remains to be shown that Φ is a valid repair with accuracy 1. Let $r_{up} = \Phi(r)$, we have $r_{up}(e) = \mathcal{Z}(e)$ for all $(e) \in R$. Recall that $\mathcal{M}(e) = r(e)$ and, thus, $r_{up}(e) = \mathcal{Z}(e)$ ensures that $\mathcal{M}(e) = \mathcal{Z}(e)$. 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 $p(b) = \mathbf{false}$. 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 b.

As $v_{root} = p_{root} = true$, we know that $b_1 = \mathbf{true}$ (b takes the **true** edge of v_{root}. Furthermore, as $p(b)$ for all predicates in $\mathcal{P}$ (as $b \notin U$), b follows the **false** edge for all remaining predicates on the path. That is $b_i = \mathbf{false}$ for $i > 1$. As we have $\mathcal{Z}(b) = out$ and Φ is a repair, we know that $v_l = out$. For each element $e \in U$, we know that $r_{up}(e) = in$ which implies that the path for 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 e would take the same path as b and we have $r_{up}(e) = out \neq in = \mathcal{Z}(e)$ 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 $\leq k$. □

C Single Rule Refinement - Proofs and Additional Details

C.1 Independence of Path Repairs

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 datapoint $x \in \mathcal{X}$ 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 datapoints in $\mathcal{X}_{P_1}$, but not the labels of datapoints in $\mathcal{X}_{P_2}$ as all datapoints 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.

Lemma 1 (Path Independence of Repairs). *Given a rule r and $\mathcal{Z}$, 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 $\mathcal{Z}$ of minimal cost.*

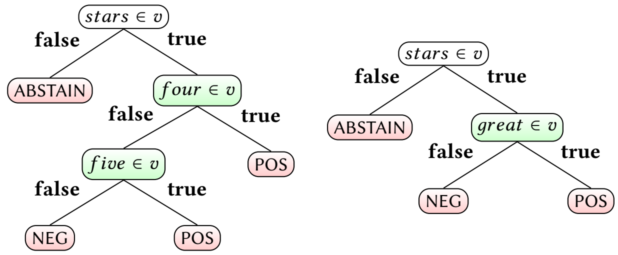


Figure 13: A non-optimal rule repair for the LF from Example 5 produced by the algorithm from Lemma 3 and an optimal repair (right)

Proof. 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 $\Lambda_{C^{\wedge}}$ 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 2. *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 document labeling usecase. If $\mathcal{P}$ contains predicate $w \in v$ every word w, then $\mathcal{P}$ is partitioning.*

Proof. **Atomic unit comparisons.** Consider an arbitrary pair of datapoints $x_1 \neq x_2$ for some rule r over $\mathcal{P}$. Since, $x_1 \neq x_2$ it follows that 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 $\mathcal{P}$.

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

Document Labeling. Recall that the atomic units for the text labeling usecase are words in a sentence. Thus, the claim follows from the atomic unit comparisons claim proven above.

□

C.3 Existence of Path 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 datapoints $x_1 \neq x_2$ there exists $p \in \mathcal{P}$ such that:

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

Observe that any two datapoints $x_1 \neq x_2$ 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 datapoints 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. 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 $\mathcal{Z}_P$ to all datapoints in $\mathcal{X}_P$.

Lemma 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 $\mathcal{Z}_P$ for datapoints $\mathcal{X}_P$ on path P. Then there exists a refinement repair Φ for path P and $\mathcal{Z}_P$ such that: $cost(\Phi) \leq |\mathcal{X}_P|$.*

Proof. We will use y_x to denote the expected label for x, i.e., $y_x = Z(x)$. Consider the following recursive greedy algorithm that assigns to each $x \in \mathcal{X}_P$ the correct label. The algorithm starts with $\mathcal{X}_{cur} = \mathcal{X}_P$ and in each step finds a predicate p that "separates" two datapoints x_1 and x_2 from $\mathcal{X}_{cur}$ with $y_{x_1} \neq y_{x_2}$. That is, $p(x_1)$ is true and $p(x_2)$ is false. As $\mathcal{P}$ is partitioning such a predicate has to exist. Let $\mathcal{X}_1 = \{x \mid x \in \mathcal{X}_{cur} \wedge p(x)\}$ and $\mathcal{X}_2 = \{x \mid x \in \mathcal{X}_{cur} \wedge \neg p(x)\}$. We know that $x_1 \in \mathcal{X}_1$ and $x_2 \in \mathcal{X}_2$. That means that $|\mathcal{X}_1| < |\mathcal{X}_P|$ and $|\mathcal{X}_2| < |\mathcal{X}_P|$. The algorithm repeats this process for $\mathcal{X}_{cur} = \mathcal{X}_1$ and $\mathcal{X}_{cur} = \mathcal{X}_2$ until all datapoints in $\mathcal{X}_{cur}$ have the same desired label which is guaranteed to be the case if $|\mathcal{X}_{cur}| = 1$. 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 $\mathcal{X}_{cur}$ is reduced by at least one, the algorithm will terminate after adding at most $|\mathcal{X}_P|$ predicates. □

C.4 Non-minimality of the Algorithm from Lemma 3

We demonstrate the non-minimality by providing an example on which the algorithm returns a non-minimal repair of a rule.

Example 5. *Consider $\mathcal{X}$ as shown below with 3 documents, their current labels (NEG) assigned by a rule and expected labels from $\mathcal{Z}$. The original rule consists of a single predicate $stars \in v$, assigning all documents that contain the word "stars" the label NEG. The algorithm may repair the rule by first adding the predicate $four \in v$ which separates d_1 and d_3 from d_2. Then an additional predicate has to be added to separate d_1 and d_3, e.g., $five \in v$. The resulting rule is shown Figure 13 (left). The cost of this repair is 2. However, a repair with a lower costs exists: adding the predicate $great \in v$ instead. This repair has a cost of 1. The resulting rule is shown in Figure 13 (right).*

sentence	*Current label*	*Expected labels from $\mathcal{Z}$*
d_1: I rate this one stars. This is bad.	NEG	NEG
d_2: I rate this four stars. This is great.	NEG	POS
d_3: I rate this five stars. This is great.	NEG	POS

C.5 Equivalence of Predicates on Datapoints

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. $\mathcal{X}_P$ can be exploited to reduce the search space of predicates and present an algorithm that is exponential in $|\mathcal{X}_P|$ which determines an optimal repair independent of the size of the space of all possible predicates. First observe that with $|\mathcal{X}_P| = n$, there are exactly 2^n

Algorithm 4: GreedyPathRepair

Input : Rule r
Path P
datapoints to fix $\mathcal{X}_P$
Expected labels for assignments $\mathcal{Z}_P$
Output: Repair sequence Φ which fixes r wrt. $\mathcal{Z}_P$

```
1  todo ← [(P, Z_P)]
2  Φ = []
3  while todo ≠ ∅ do
4      (P, Z_P) ← pop(todo)
5      if ∃x_1, x_2 ∈ X : Z_P[x_1] ≠ Z_P[x_2] then
6          /* Determine predicates that distinguish
              assignments that should receive different labels
              for a path                                       */
7          p ← GetSeperatorPred(x_1, x_2)
8          y_1 ← Z_P[x_1]
9          φ ← refine(r_cur, P, y_1, p, true)
10         X_1 ← {x | x ∈ X_P ∧ p(x)}
11         X_2 ← {x | x ∈ X_P ∧ ¬p(x)}
12         todo.push((P[r_cur, x_1], X_1))
13         todo.push((P[r_cur, x_2], X_2))
14     else
15         φ ← refine(r_cur, P, Z_P[x])
16     r_cur ← φ(r_cur)
17     Φ.append(φ)
18 return Φ
```

possible outcomes of applying a predicate p to $\mathcal{X}_P$ (returning either true or false for each $x \in \mathcal{X}_P$). Thus, with respect to the task of repairing a rule through refinement to return the correct label for each $x \in \mathcal{X}_P$, 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 4 (Equivalence of Predicates). *Consider a space of predicates $\mathcal{P}$, rule r, and set of datapoints $\mathcal{X}_P$ with associated expected labels $\mathcal{Z}_P$ 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. $\mathcal{X}_P$, written as $p \equiv_{\mathcal{X}_P} p'$ if $p(x) = p'(x)$ for all $x \in \mathcal{X}_P$. Furthermore, consider a reduced space of predicates $\mathcal{P}_\equiv$ that fulfills the following condition:*

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

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

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

Proof. Let $\Phi = \mathcal{A}(\mathcal{P}_\equiv, r, \mathcal{Z}_P)$ and $\Phi_\equiv = \mathcal{A}(\mathcal{P}, r, \mathcal{Z}_P))$. 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 $x \in \mathcal{X}_P$, 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 $\mathcal{Z}_P$ 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.[1]

D Path Refinement Repairs - Proofs and Additional Details

D.1 GreedyPathRepair

The function GreedyPathRepair is shown Algorithm 4. This algorithm maintains a list of pairs of paths and datapoints at these paths to be processed. This list is initialized with all datapoints $\mathcal{X}_P$ from $\mathcal{Z}_P$ and the path P provided as input to the algorithm. In each iteration, the algorithm picks two datapoints 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 $\mathcal{X}_1 = \{x \mid x \in \mathcal{X}_P \wedge p(x)\}$ and $\mathcal{X}_2 = \{x \mid x \in \mathcal{X}_P \wedge \neg p(x)\}$ with their respective paths to the list. As shown in the proof of Lemma 3, this algorithm terminates after adding at most $|\mathcal{X}_P|$ new predicates.

To ensure that all datapoints ending in path P get assigned the desired label based on $\mathcal{Z}_P$, we need to add predicates to the end of P to "reroute" each datapoint to a leaf node with the desired label. As mentioned above, this algorithm implements the approach from the proof of Lemma 3: for a set of datapoints taking a path with prefix P ending in a leaf node that is not pure (not all datapoints in the set have the same expected label), we pick a predicate that *"separates"* the datapoints, i.e., that evaluate to true on one of the datapoints and false on the other. Our algorithm applies this step until all leaf nodes are pure wrt. the datapoints from $\mathcal{X}_P$. For that, we maintain a queue of path and datapoint set pairs which tracks which combination of paths and datapoint sets still have to be fixed. This queue is initialized with P and all datapoints for P from $\mathcal{X}_P$. The algorithm processes sets of datapoints until the todo queue is empty. In each iteration, the algorithm greedily selects a pair of datapoints x_1 and x_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 x_1 and false on x_2 (or vice versa). If we extend path P with p, then x_1 will follow the **true** edge of p and x_2 will follow the **false** edge (or vice versa). This effectively partitions the set of datapoints for path P into two sets $\mathcal{X}_1$ and $\mathcal{X}_2$ where $\mathcal{X}_1$ contains x_1 and $\mathcal{X}_2$ contains x_2. We then have to continue to refine the paths ending in the

[1] With the exception of the class of predicates that return false on all $x \in \mathcal{X}_P$. However, this class of predicates will never be part of an optimal repair as it does only trivially partitions $\mathcal{X}_P$ into two sets $\mathcal{X}_P$ and $\emptyset$.

dataset	repairer	fix%	preserv%	global acc.	new global acc.
FNews	RC	1	1	0.71	0.92
FNews	LLM	0.9	0.65	0.71	0.81
Amazon	RC	1	1	0.6	0.77
Amazon	LLM	0.9	0.5	0.6	0.7

Table 3: LLM vs RuleCleaner (RC) quality rule refinement comparison

Algorithm 5: BruteForcePathRepair

Input : Rule r
Path P
Datapoints to fix $\mathcal{X}_P$
Expected labels for datapoints $\mathcal{Z}_P$
Output: Repair sequence Φ which fixes r wrt. $\mathcal{Z}_P$

$todo \leftarrow [(r, \emptyset)]$
$\mathcal{P}_{all}$ = `GetAllCandPredicates`$(P, \mathcal{X}_P, \mathcal{Z}_P)$
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 $y_1 \in \mathcal{Y} \wedge y_1 \neq last(P_{cur})$ **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 Acc$(r_{new}, \mathcal{Z}_P) = 1$ **then**
          return Φ_{new}
        else
          $todo.push((r_{new}, \Phi_{new}))$

two children of p wrt. these sets of datapoints. This is ensured by adding these sets of datapoints with their new paths to the todo queue (lines 12 and 13). If the current set of datapoints does not contain two datapoints with different labels, then we know that all remaining datapoints should receive the same label. The algorithm picks one of these datapoints x (line 14) and changes the current leave node's label to $\mathcal{Z}_P(x)$.

Generating Predicates. The implementation of `GetCoveringPred` is specific to the type of PWSS. We next present implementations of this procedure for weak supervised labeling that exploit the properties of these two application domains. However, note that, as we have shown in Appendix C.3, as long as the space of predicates for an application domain contains equality and inequality comparisons for the atomic elements of datapoints, it is always possible to generate a predicate for two datapoints such that only one of these two datapoints fulfills the predicate. The algorithm splits the datapoint set $\mathcal{X}^*$ processed in the current iteration into two subsets, which each are strictly smaller than $\mathcal{X}^*$. Thus, the algorithm is guaranteed to terminate and by construction assigns each datapoints x in $\mathcal{X}_P$ its desired label $\mathcal{Z}_P(x)$.

D.2 BruteForcePathRepair

The brute-force algorithm (Algorithm 5) 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 $\mathcal{Z}_P$ wrt. $\mathcal{Z}_P$. We first determine all predicates that can be used in the candidate repairs. As argued in Appendix C.5, there are only finitely many distinct predicates (up to equivalence) for a given set $\mathcal{X}_P$. 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 selecting the refined rule with minimum cost that satisfies all assignments. As we generate subtrees in increasing size, Lemma 3 implies that the algorithm will terminate and its worst-case runtime is exponential in $n = |\mathcal{X}_P|$ as it may generate all subtrees of size up to n.

D.3 Proof of Theorem 2

Proof. Proof of Theorem 2 In the following let $n = |\mathcal{X}_P|$.

GreedyPathRepair: As GreedyPathRepair does implement the algorithm from the proof of Lemma 3, it is guaranteed to terminate after at most n steps and produce a repair that assign to each x the label $\mathcal{Z}_P(x)$.

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 $\mathcal{X}_P$. 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 datapoints from $\mathcal{X}_P$ 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 datapoints $\mathcal{X}_P$ into two subsets $\mathcal{X}_1$ and $\mathcal{X}_2$ with fewer predicates which have a lower Gini impurity score than splitting into $\mathcal{X}_1 = \mathcal{X}_P$ and $\mathcal{X}_2 = \emptyset$, the size of the datapoints that are being processed, strictly decrease in each step. Thus, the algorithm will, in the worst-case, terminate after adding n predicates. □

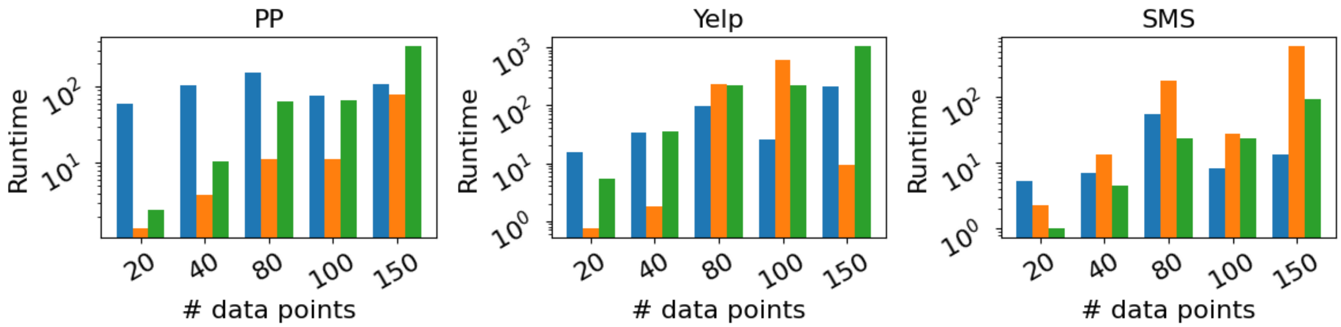
E Additional Experiment Details

E.1 runtime breakdown

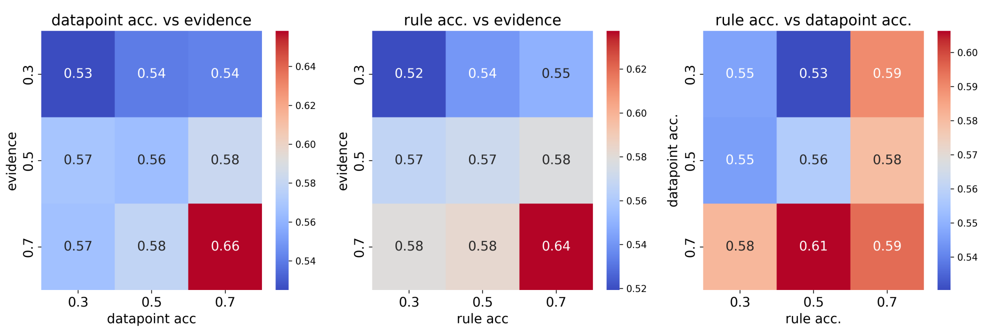
The runtime breakdown of the rest of the datasets from Section 5.2 are shown in Figure 14a.

E.2 MILP thresholds

The effects on global accuracy of the pairwise relationships of τ_{acc}, τ_E, and τ_{racc} are shown in Figure 14b. The color of a square represents global accuracy after the repair. Based on these results, it is generally preferable to set the thresholds higher as discussed in Section 2.2. However, larger thresholds reduce the amount of viable solutions to the MILP and, thus, can significantly increase the runtime of solving the MILP and lead to overfitting to $\mathcal{X}^*$.



(a) Runtime, varying the size of X^*.



(b) Pairwise interaction heatmaps for New Global Acc.

Figure 14: Additional experimental results

dataset	repairer	fix%	preserv%	global acc.	new global acc.
FNews	RC	1	1	0.71	0.92
FNews	LLM	0.9	0.65	0.71	0.81
Amazon	RC	1	1	0.6	0.77
Amazon	LLM	0.9	0.5	0.6	0.7

Table 4: LLM vs RuleCleaner (RC) quality rule refinement comparison

E.3 LLM as LF generator

The prompt used to generate LFs mentioned in Section 5.3 is shown in Appendix F.

F Use LLM as refinement

In this section, we compare RuleCleaner against a baseline using a large language model (LLM). We designed a prompt instructing the LLM to act as an assistant and refine the LFs given a set of labeled datapoints. In this experiment, we used the *FNews* dataset with 40 labeled datapoints and *Amazon* dataset with 20 labeled examples. We used `GPT-4-turbo` as the LLM. A detailed description of the prompt and responses from the LLM are presented in the following.

We manually inspected the rules returned by the LLM to ensure that they are semantically meaningful. The quality of results after running Snorkel with the refined LFs from LLM and RuleCleaner are shown in Table 4. `fix%` measures the percentage of the wrong predictions by Snorkel are fixed after retraining Snorkel with the refined rules. `preserv %` measures what percentage of the input correct predictions by Snorkel remain the same after retraining with refined rules. RuleCleaner outperforms the LLM in both global accuracy and accuracy on labeled input data. Note that the purpose of comparing with LLMs is not to test if LLM can reproduce our algorithm but to test how good is LLM in performing the refinement task given the description. We observe that the LLM tends preserve the semantic meaning of the original LFs in the repairs it produces. For example, in one of the rules from *FNews*, the original rule is `if 'talks'in text then REAL otherwise ABSTAIN` and the repaired rule was `if any(x in text for x in ['discussions ', 'negotiations', 'talks']`. In one of the rules from *Amazon*, the original rule `if any (x in text for x in ['junk',' disappointed','useless'])then NEGATIVE else ABSTAIN` mainly covers negative reviews. The LLM did add more negative words such as `'defective'` whereas RuleCleaner could possibly add opposite sentiment conditions based on the solutions provided by the MILP. For example, it is possible for RuleCleaner to refine a rule with negative sentiment by adding `else if 'great'in text then POSITIVE else ABSTAIN`

> You will help me refine a list of python functions that are being used on a classification task. I will provide you a list of function definitions and a dataset table. The classification process is done in the following way: Given the dataset and the list of rules, there will be a black box aggregator that

makes decisions by combining the results of the rules on the dataset.

the dataset contains 2 columns: "text" and "expected label". You should apply the function to the given dataset and then modify the functions as you like. For this task, you can add new "if-else" statements or adjust the existing logic as you like.

return the refined labeling functions using the same formats as provided. Please add comments on why you changed the original function.

The returned refined functions for *FNews* are shown in Figure 15 to 19.

```python
def func1(text):
    # Expanded keyword coverage
    keywords = ["video", "featured",
    "clip", "footage"]
    if any(x in text for x in keywords):
        return 0
    else:
        return -1
def func2(text):
    # Expanded keyword coverage
    keywords1 = ["getty", "screenshot", "image"]
    keywords2 = ["registry", "database"]

    if any(x in text for x in keywords1):
        if any(x in text for x in keywords2):
            return 1
        else:
            return -1
    else:
        return -1

def func3(text):
    # Expanded keyword coverage
    keywords = ["minister", "ministry", "parliament",
                "government", "official"]

    if any(x in text for x in keywords):
        return 0
    else:
        return -1

def func4(text):
    # Expanded keyword coverage
    keywords = ["pic", "photo", "image", "snapshot"]

    if any(x in text for x in keywords):
        return 1
    else:
        return -1

def func5(text):
    # Expanded keyword coverage
    keywords1 = ["wednesday", "spokesman",
    "thursday", "representatives",
    "nov", "tuesday", "monday"]
    keywords2 = ["legislation", "bill",
    "law"]

    if any(x in text for x in keywords1):
        if any(x in text for x in keywords2):
            return 0
        else:
            return -1
    else:
        return -1

def func6(text):
    # Expanded keyword coverage
    keywords = ["korea", "missile", "region",
                "regional", "authorities",
                "conflict", "war", "border"]

    if any(x in text for x in keywords):
        return 0
    else:
        return -1
```

Figure 15: Refined *FNews* funcs by LLM (Part 1)

```python
def func7(text):
    # Expanded keyword coverage
    keywords1 = ["korea", "missile", "region",
                 "regional", "authorities", "conflict",
                 "war", "border"]
    keywords2 = ["korean", "diplomatic"]

    if any(x in text for x in keywords1):
        if any(x in text for x in keywords2):
            return 0
        else:
            return -1
    else:
        return -1

def func8(text):
    # Expanded keyword coverage
    keywords = ["getty", "watch", "image",
                "com", "https", "don", "woman",
                "didn", "gop", "site", "web"]

    if any(x in text for x in keywords):
        return 1
    else:
        return -1

def func9(text):
    keywords = ["talks", "negotiations",
    "discussions"]

    if any(x in text for x in keywords):
        return 0
    else:
        return -1

def func10(text):
    keywords = ["rex", "rex images", "rex news"]

    if any(x in text for x in keywords):
        return 1
    else:
        return -1

def func11(text):
    # Expanded geopolitical terms
    keywords = ["northern", "turkey",
                "britain", "forces", "ruling",
                "european", "alliance", "nato", "eu"]

    if any(x in text for x in keywords):
        return 0
    else:
        return -1
```

Figure 16: Refined *FNews* funcs by LLM (Part 2)

```
# func1: Improved to capture more
# variations of ratings, like "rating",
# "out of five", etc.
def func1(text):
    keywords = ['star', 'stars',
    'rating', 'out of five', 'rated']
    if any(x in text for x in keywords):
        return 0
    else:
        return -1

# func2: Expanding product-related terms
def func2(text):
    keywords = ['product', 'fit',
    'quality', 'size', 'cheap', 'wear',
    'material', 'design', 'durable']
    if any(x in text for x in keywords):
        return 1
    else:
        return -1

# func3: No changes needed, it's quite
# simple but effective
def func3(text):
    keywords = ['great']
    if any(x in text for x in keywords):
        return 0
    else:
        return -1

# func4: Simplified to capture
# more positive combinations
def func4(text):
    keywords1 = ['great', 'excellent', 'amazing']
    keywords2 = ['stars', 'works', 'fantastic',
    'love']
    if any(x in text for x in keywords1):
        if any(x in text for x in keywords2):
            return 0
        else:
            return -1
    else:
        return -1

# func5: Added more phrases for negative sentiment
def func5(text):
    keywords = ['waste', 'not worth',
    'disappointed', 'regret',
    'poor value']
    if any(x in text for x in keywords):
        return 1
    else:
        return -1

# func6: Added more keywords
# related to product
# discomfort or bad quality
def func6(text):
    keywords = ['shoes', 'item', 'price',
    'comfortable', 'plastic',
    'uncomfortable', 'bad quality']
    if any(x in text for x in keywords):
        return 0
    else:
        return -1
```

Figure 17: Refined *Amazon* funcs by LLM (Part 1)

```python
# func7: Expanded list for
# disappointment and dissatisfaction
def func7(text):
    keywords = ['junk', 'bought', 'like',
    'dont', 'just', 'use', 'buy',
    'work', 'small', 'didnt',
    'did', 'disappointed', 'bad',
    'terrible', 'horrible', 'awful', 'useless']

    if any(x in text for x in keywords):
        return 1
    else:
        return -1

# func8: Also improving dissatisfaction
# detection with more negative keywords
def func8(text):
    keywords1 = ['junk', 'bought', 'like',
    'dont', 'just', 'use', 'buy',
    'work', 'small', 'didnt',
    'did', 'disappointed', 'bad',
    'terrible', 'horrible',
    'awful', 'useless']

    keywords2 = ['shoes', 'metal',
    'fabric', 'replace', 'battery',
    'warranty', 'plug', 'defective',
    'broken']
    if any(x in text for x in keywords1):
        if any(x in text for x in keywords2):
            return 1
        else:
            return -1
    else:
        return -1

# func9: Added more positive words
def func9(text):
    keywords = ['love', 'perfect',
    'loved', 'nice', 'excellent',
    'works', 'loves', 'awesome',
    'easy', 'fantastic', 'recommend']

    if any(x in text for x in keywords):
        return 0
    else:
        return -1

# func10: Added more combination
# cases for positive reviews
def func10(text):
    keywords1 = ['love', 'perfect',
    'loved', 'nice', 'excellent',
    'works', 'loves', 'awesome',
    'easy', 'fantastic', 'recommend']

    keywords2 = ['stars', 'soft',
    'amazing', 'beautiful']

    if any(x in text for x in keywords1):
        if any(x in text for x in keywords2):
            return 0
        else:
            return -1
    else:
        return -1
```

Figure 18: Refined *Amazon* funcs by LLM (Part 2)

```python
# func11: Adding more
# product-related terms
# and combinations
def func11(text):
    keywords1 = ['love', 'perfect',
    'loved', 'nice', 'excellent',
    'works', 'loves', 'awesome',
    'easy', 'fantastic', 'recommend']

    keywords2 = ['shoes', 'bought',
    'use', 'purchase', 'purchased',
    'colors', 'install', 'clean',
    'design', 'pair', 'screen',
    'comfortable', 'products', 'item']

    if any(x in text for x in keywords1):
        if any(x in text for x in keywords2):
            return 0
        else:
            return -1
    else:
        return -1

# func12: No change; this already targets
# product-related complaints well.
def func12(text):
    keywords = ['returned', 'broke',
    'battery', 'cable', 'fits',
    'install', 'sturdy', 'ordered',
    'usb', 'replacement', 'brand',
    'installed', 'unit', 'batteries',
    'box', 'warranty', 'defective',
    'cheaply', 'durable', 'advertised']

    if any(x in text for x in keywords):
        return 1
    else:
        return -1

# func13: Adding a few more
# fun product terms for cuteness
def func13(text):
    keywords = ['cute', 'shirt',
    'adorable', 'lovely', 'sweet']

    if any(x in text for x in keywords):
        return 0
    else:
        return -1

# func14: Improved negative terms
# related to fabric and poor quality
def func14(text):
    keywords = ['fabric', 'return',
    'money', 'poor', 'garbage',
    'poorly', 'terrible', 'useless',
    'horrible', 'returning',
    'flimsy', 'falling apart']

    if any(x in text for x in keywords):
        return 1
    else:
        return -1

# func15: Added more keywords related to
# specific product types and issues
def func15(text):
    keywords = ['pants', 'looks',
    'toy', 'color', 'camera',
    'water', 'phone', 'bag', 'worked',
    'arrived', 'lasted', 'fabric',
    'material', 'build quality', 'finish']

    if any(x in text for x in keywords):
        return 1
    else:
        return -1
```


Figure 19: Refined *Amazon* funcs by LLM (Part 3)

LLM LF generation prompt template

Instructions for Generating Labeling Functions
You are assisting in the generation of labeling functions based on a set of sentences, each provided with its ground truth labels.

Label Information
Below are the available labels and their corresponding label numbers:
`{class_mapping_string}`

Task
Your objective is to derive labeling functions from the given sentences and their associated labels. The labeling functions should follow one of two formats:

(1) Keyword-based functions
(2) Regular expression-based functions

Each labeling function should be designed to capture meaningful patterns from the text while ensuring generalizability.

Constraints

- Each labeling function should contain no more than **10** keywords or regular expressions.
- Regular expressions should be used only when a keyword-based function cannot adequately capture a pattern.
- Ensure that the selected keywords and patterns are generalizable rather than overly specific.

Templates for Labeling Functions
The two accepted templates for defining labeling functions are shown below:

- **Keyword Template:**

```
def keyword_[label_name][label_number](x):
    ABSTAIN = -1
     keywords = [list of identified keywords]
    return [label_number] if any(keyword in x for keyword in keywords)
    else ABSTAIN
```

- **Regular Expression Template:**

```
def regex_[label_name][label_number](x):
     ABSTAIN = -1
     return [label_number] if [regular expression related condition]
    else ABSTAIN
```

Sentence Format
The sentences are presented in the following format:
`Sentence [Sentence ID]: [Sentence Content]. Label: [Label Text]`
Below are the sentences for this task:
`{formatted_sentences}`
(where `formatted_sentences` is the list of sentences in the specified format)

Final Step
Using the provided templates, generate appropriate labeling functions based on the given sentences. Ensure that the functions adhere to the constraints and effectively classify the provided text.