

# Refining Labeling Functions With Limited Labeled Data

Anonymous  
Author(s)

## Abstract

Data programming systems like Snorkel generate large amounts of training data by combining the outputs of a set of user-provided labeling functions (LFs) on unlabeled datapoints with a blackbox ML model. This significantly reduces human effort for labeling data compared to manual labeling. The efficiency of the labeling can be further improved through approaches like Witan that automate (parts of) the LF generation process. However, no matter whether LFs are created with or without the help of such tools, the quality of the generated training data depends directly on the accuracy of the set of LFs. In this work, we study the problem of fixing LFs based on a small set of labeled example datapoints. Our approach complements manual and automatic generation of LFs by improving a given set of LFs. Towards this goal, we develop novel techniques for repairing a set of LFs 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). 2018. Refining Labeling Functions With Limited Labeled Data. In *Proceedings of SIGMOD International Conference on Management of Data (SIGMOD’25)*. ACM, New York, NY, USA, 21 pages. <https://doi.org/XXXXXXX.XXXXXXX>

## 1 Introduction

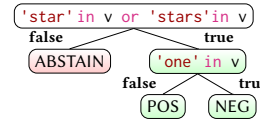
Data programming [23] has emerged as a powerful weak supervision technique [31] that revolutionizes the creation of training datasets by leveraging machine learning (ML) models. Unlike the traditional manual labeling process, where labels are painstakingly assigned by hand to each training datapoint, data programming allows for label assignment through labeling functions (LFs) — heuristics that take an datapoint as input and output a label. This approach dramatically minimizes the human labor required to prepare training data. To push this reduction even further, recent approaches automate the generation of labeling functions [4, 7, 11, 28]. For example, Witan [7] automatically crafts labeling functions from simple predicates that are effective in differentiating datapoints, subsequently guiding users to select and refine sensible LFs, while

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

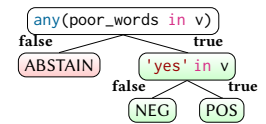
def key_word_waste(v): #LF-2
    return NEGATIVE if ('waste' in v) else ABSTAIN

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

(a) Three example labeling functions for amazon reviews



(b) Refined rule #1



(c) Refined rule #3

Figure 1: LFs before / after refinement by RULECLEANER

another approach by Guan et al. [11] employs large language models (LLMs) to derive labeling functions based on a small amount of labeled training data, further enhancing efficiency and reducing the dependency on human intervention.

Regardless of whether labeling functions (LFs) are manually crafted by domain experts or generated through (semi-)automated techniques, users face significant challenges when it comes to debugging and repairing these LFs to correct issues with the resulting labeled training data. Several factors contribute to this difficulty. In particular, (i) the black-box nature of the model that combines LF results obscures the specific LFs responsible for mislabeling a datapoint; (ii) large training datasets make it difficult for users to manually identify all issues caused by problematic LFs; and (iii) the semantics of complex LFs may be unclear even to domain experts, especially when they evolve over time based on updates made by different curators. Consequently, identifying potentially faulty or inaccurate LFs from the final output, as well as finding effective strategies for correcting them, remains a substantial hurdle to achieving high labeling accuracy.

In this work, we tackle the challenge of automatically suggesting repairs for a set of LFs based on a small subset of labeled datapoints. Our approach focuses on the following type of repair: refining an LF by locally overriding its outputs to align better with expectations for specific datapoints. Rather than replacing human domain expertise or existing automated LF generation tools, our method, RULECLEANER, serves as a complementary tool, providing targeted improvements to an existing set of LFs. Our approach is versatile, operating under minimal assumptions about the LFs or the black-box model that combines their outputs to assign final labels to datapoints. This flexibility allows RULECLEANER to support arbitrarily complex LFs, and various black-box models that combine them such as Snorkel [23] or simpler models like majority voting. Furthermore, our approach is agnostic to the source of LFs, enabling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SIGMOD’25, June 22–27, 2025, Berlin, Germany

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

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: Amazon products reviews with ground truth labels ("P"ositive or "N"egative), predicted labels by Snorkel [23] (before and after rule refinement), and the results of the LFs from Figure 1 ("-" means ABSTAIN). Updated results for the repaired rules are shown in blue in parenthesis.**

the repair of LFs generated by tools like Witan [7], LLMs [11], or those manually crafted by domain experts.

To address the challenge of refining LFs expressed in a general-purpose programming language, we model LFs as *rules*, represented as decision trees. In these trees, inner nodes are *predicates*, Boolean conditions evaluated on datapoints (simple predicates like comparison or set membership, or complex predicates defined by arbitrary functions), and leaf nodes that 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. Although simple, *this model can represent any LF as a rule* by creating predicates that evaluates the LF on the datapoint and matches its result against every possible label.

**EXAMPLE 1.1.** Consider the Amazon Review Dataset from [7, 14] 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 [7] for this task are shown in Figure 1a. For datapoint, key\_word\_star labels reviews as POS that contain either "star" or "stars" and otherwise returns ABSTAIN (the function cannot make a prediction). Some reviews with their ground truth labels (unknown to the user) and the labels predicted by Snorkel [24] are shown in Table 1, which also shows the results of three LFs including the ones 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 an arbitrarily complex algorithm or ML classifier to generate a final label for each datapoint.

Suppose a small set of labeled data is available that shows the true label for each review; a sample of the labeled reviews is shown in Table 1. RULECLEANER uses these ground truth labels to generate a set of repairs for the LFs by deleting or refining LFs to ensure they align with the ground truth. Table 1 also shows the labels produced by the repaired LFs (updated labels are highlighted in blue in parentheses), and the old and new predictions generated by Snorkel before and after refinements of the LFs. 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 (discussed in Section 2) with new predicates highlighted in green. The updated version of rule LF-1 fixes the labels assigned to review 1 and 3 from P to N. Intuitively, this repair is sensible: a review mentioning "one" and "star(s)" is very likely negative. The prediction for review 4 is fixed by checking for the keyword 'yes' in LF-3 which flips the prediction from N to P for this review by both LF-3 and when Snorkel assigns the final label.

The RULECLEANER system we present in this work produces repairs as shown in the example above.

## Our Contributions

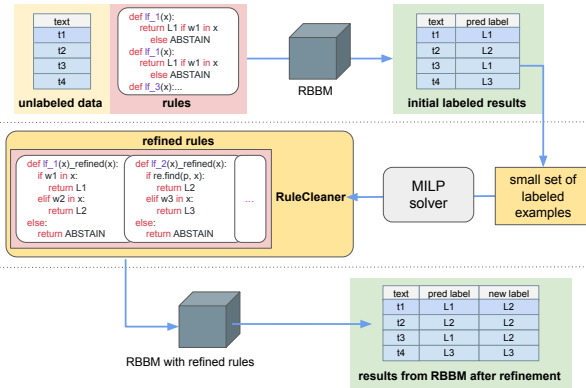
- **General Model for Rule-Based Systems.** We introduce a general model for weak supervision systems, such as Snorkel [23], as rule-based black-box models (RBBM), where interpretable LFs (rules) are combined to predict labels for a dataset  $\mathcal{X}$  (Section 2). Our model accommodates both simple and complex black-box systems and supports arbitrary LFs. We formally analyze the problem of repairing LFs in RBBM with minimal modification based on the ground truth labels provided for a small subset  $\mathcal{X}^*$  of  $\mathcal{X}$ .
- **Efficient Rule Repair Algorithms.** We define the rule repair problem as fixing a set of rule through *refinement* such that the output of the repaired rules aligns with the ground truth on  $\mathcal{X}^*$ . We avoid overfitting by: (i) minimizing the changes to the outputs of the original LFs and (ii) only requiring sufficiently high accuracy on the labeled datapoints instead of requiring all LFs to return the correct labels on all labeled datapoints. While this problem is NP-hard, we benefit from the fact that the set of labeled examples is typically small. It is feasible to solve the problem exactly by formulating it as a mixed-integer linear program (MILP) that determines changes to the outputs of rules on  $\mathcal{X}^*$  (Section 3). To implement these changes to the outputs of the rules, we then refine individual rules to match the desired outputs produced as a solution to the MILP (Section 4). To further decrease the likelihood of overfitting, we want to minimize the number of new predicates we have to add to rules to change their outputs on the labeled examples. As this problem is also NP-hard, we propose a polynomial-time heuristic algorithm for refining rules that minimizes an information-theoretic metric (Section 4.1).
- **Comprehensive Experimental Evaluation.** We validate our approach using Snorkel [23] and demonstrate that our algorithm effectively repairs rules to align RBBM outputs with ground truth labels. The repairs also generalize well to unseen datapoints (Section 5). Additionally, we apply our framework to rules automatically generated by Witan [7] and the LLM-based approach from [11], showing significant improvements in the quality of these automatically generated rules. We conduct experiments on 11 real datasets. Surprisingly even with a tiny labeled training dataset with 20 tuples out of 200000 tuples, the global accuracy can improve up to 19% and with 20-160 labeled tuples, the global accuracy can improve up to 24%, showing the effectiveness of

our approach in suggesting repairs or fixes to labeling functions to human users.

We review related work in **Section 6** and conclude with future work in **Section 7**.

## 2 The RULECLEANER Framework

In this section, we formalize Rule-based Black-Box Models (RBBMs), define the rule repair problem, and examine its computational complexity. Our system, RULECLEANER, enhances RBBMs by fixing rules to improve labeling accuracy based on a small amount of labeled examples. As illustrated in Figure 2, RULECLEANER begins with an existing set of labeling functions and the corresponding labels produced by an RBBM for an unlabeled dataset  $\mathcal{X}$  and a small subset of labeled datapoints  $\mathcal{X}^* \subset \mathcal{X}$ . The original rules may have been manually curated by a human domain expert or be the result of an automated approach for generating labeling functions like Witan [7]. RULECLEANER then refines specific labeling functions based on this input, generating an updated set of rules. Finally, the RBBM applies these revised rules to re-label the dataset, which users can further review and adjust if necessary.



**Figure 2: The RULECLEANER framework for repairing rules in data programming. After running an RBBM (e.g., Snorkel) on the rule results, RULECLEANER refines these rules using a small subset of labeled examples  $\mathcal{X}^*$ . Finally, the model is retrained by running the RBBM on the output of the refined rules on  $\mathcal{X}$  and used to produce new labels for the whole dataset  $\mathcal{X}$ .**

The model of rule-based black-boxes we use can capture various weakly supervised labeling systems [7, 22, 23], and any other system that uses a set of rules to solve a problem that can be modeled as a prediction task. In this work, we use Snorkel [23] as the RBBM.

As previously noted, the process for combining LF outputs into final labels can vary widely, often relying on complex black-box models. This variability presents a challenge: correcting individual LF outputs does not always ensure that the black-box model will adjust its final label. For simpler models like majority voting, ensuring LFs return correct labels or abstain suffices to achieve accurate labeling. However, this is not guaranteed with more complex models like Snorkel. Our approach, therefore, makes a reasonable

assumption that as more LFs agree on a label, the black-box model is more likely to select this correct label.

### 2.1 Black-Box Model, Datapoints, and Labels

Consider a set of input *datapoints*  $\mathcal{X}$  and a set of discrete *labels*  $\mathcal{Y}$ . A RBBM takes  $\mathcal{X}$ , the labels  $\mathcal{Y}$ , and a set of rules  $\mathcal{R}$  (discussed in Section 2.2) as input and produces a *model*  $\mathcal{M}_{\mathcal{R},\mathcal{X}}$  (Definition 2.5) as the output that maps each datapoint in  $\mathcal{X}$  to a label in  $\mathcal{Y}$ , i.e.,

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

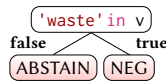
Without loss of generality, we assume the presence of an *abstain* label  $y_0 \in \mathcal{Y}$  that is used by the RBBM or a rule to abstain from providing a label to some datapoints. For a datapoint  $x \in \mathcal{X}$ ,  $\hat{y}_x = \mathcal{M}_{\mathcal{R},\mathcal{X}}(x)$  denotes the label given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

We assume that a datapoint  $x \in \mathcal{X}$  consists of a set of *atomic units*. For datapoint, if Snorkel [23] (the RBBM) is used to generate labels for documents (the training data),  $\mathcal{X}$  is the set of document (datapoints) to be labeled, and each document consists of a set of words as atomic units. If the classification task is sentiment analysis on user reviews (for songs, movies, products, etc.), the set of labels assigned by such a RBBM may be  $\mathcal{Y} = \{\text{POS}, \text{NEG}, \text{ABSTAIN}\}$ , where  $y_0 = \text{ABSTAIN}$ .

**EXAMPLE 2.1.** Table 1 shows a set of reviews (datapoints) with possible labels of POS or P (a review with positive sentiment), NEG or N (a negative review), and ABSTAIN or - (the abstain label). The atomic units of the first review with id 0 are all the words that it contains, e.g., “five”, “stars”, etc.

### 2.2 Rules in the Black-Box Model

Rules are the building blocks of RBBMs. These rules are either designed by a human expert or discovered automatically (e.g., [7, 11]). A rule consists of one or more predicates  $\mathcal{P}_r$  from a set of domain-specific atomic predicates  $\mathcal{P}$  over a single variable  $v$  that represents the datapoint over which the rule is evaluated. These predicates are allowed to access the atomic units of the datapoint. The atomic predicates  $\mathcal{P}$  may contain simple predicates such as comparisons ( $=, \neq, >, \geq, <, \leq$ ) as well as arbitrarily complex black-box functions. As mentioned in Section 1, any LF expressed in a general-purpose programming language like Python can be expressed as a rule by wrapping it as a predicate that compares the label returned by the LF against a constant label. We have implemented an importer for Python LFs that analyzes the LF code. For simple predicates like checking for the existence of words in a sentence, our importer generates a separate predicate for each such check. As a fallback, the importer will wrap the whole LF as predicates, as explained above. We represent a rule as a *tree* where leaf nodes represent labels in  $\mathcal{Y}$  and the non-leaf nodes are labeled with predicates from  $\mathcal{P}$ . Each non-leaf node has two outgoing edges labeled with **true** and **false**. To determine the label assigned by a rule  $r$  to a datapoint  $x$ , the predicate of the root node of the  $r$ ’s tree is evaluated by substituting  $v$  with  $x$  and then evaluating the resulting condition. Based on the outcome, either the edge labeled **true** or **false** is followed to one of the children of the root. The evaluation then continues with this child node until a leaf node is reached. Then, the label  $y$  of the leaf node is returned.



**Figure 3: Rule form of the LF keyword\_word\_waste (Figure 1a)**

DEFINITION 2.2 (RULE). A rule  $r$  over atomic predicates  $\mathcal{P}$  is a labeled directed binary tree where the internal nodes are predicates in  $\mathcal{P}$ , leaves are labels from  $\mathcal{Y}$ , and edges are marked with **true** and **false**. A rule  $r$  takes as input a datapoint  $x \in \mathcal{X}$  and returns a label  $r(x) \in \mathcal{Y}$  for this assignment. Let  $\text{ROOT}(r)$  denote the root of the tree for rule  $r$  and let  $\text{C}_{\text{true}}(n)$  ( $\text{C}_{\text{false}}(n)$ ) denote the child of node  $n$  adjacent to the outgoing edge of  $n$  labeled **true** (**false**). Given a datapoint  $x$ , the result of ruler  $r$  for  $x$  is  $r(x) = \text{EVAL}(\text{ROOT}(r), x)$ . Function  $\text{EVAL}(\cdot, \cdot)$  operates on nodes  $n$  in the rule’s tree and is recursively defined as follows:

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

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

We provide linear time procedures for converting LFs to rules in Appendix A.

In particular, we have the following observation.

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

EXAMPLE 2.4. Figure 3 shows the rule for a labeling function that returns NEG if the review contains the word ‘waste’ and returns ABSTAIN otherwise.

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

DEFINITION 2.5 (BLACK-BOX MODELS). *Given a set of datapoints  $\mathcal{X}$ , a set of labels  $\mathcal{Y}$ , and a set of rules  $\mathcal{R}$ , a RBBM 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  $\mathcal{M}_{\mathcal{R},\mathcal{X}}(x) = \hat{y}_x$ .*

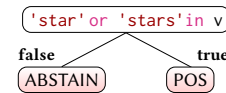
In the following we will often drop  $\mathcal{R}$  and  $X$  from  $\mathcal{M}_{\mathcal{R},X}$  when they are irrelevant to the discussion or clear from the context.

### 2.3 Labelled Examples for RULECLEANER

In this work, we assume a small subset of labeled datapoints that we use for repairing a set of rules. The user interacts with our systems as follows:

**Model Creation.** Based on the rules  $\mathcal{R}$  provided by the user, the RBBM produces a model  $\mathcal{M}_{\mathcal{R}, \mathcal{X}} : \mathcal{X} \rightarrow \mathcal{Y}$ . The rules may be generated by a human expert or generated by an existing technique for label function creation such as Witan [7].

**Ground Truth Labels.** The user provides a small labeled subset  $\mathcal{X}^*$  of  $\mathcal{X}$  for which the true labels  $y_x^* \neq y_0$  are provided. In Table 1, the predictions of the RBBM agree with the ground truth for the



**Figure 4: Rule form of the LF key\_word\_star**

reviews with ids 0, 2 ( $\hat{y}_x = y_x^* = P$ ), and disagree with the ground truth for reviews with ids 1, 3 ( $\hat{y}_x = P, y_x^* = N$ ) and 4 ( $\hat{y}_x = N, y_x^* = P$ ). For convenience we represent the output of a set of  $m$  rules  $\mathcal{R} = \{r_1, \dots, r_m\}$  on a  $\mathcal{X}^* = \{x_1, \dots, x_n\}$  with  $n$  datapoints as an  $n \times m$  matrix  $\mathcal{R}(\mathcal{X}) = \mathbf{O}$  where the entry  $\mathbf{O}_{ij} = r_j(x_i)$  stores the result of applying rule  $r_j$  to datapoint  $x_i$ .

**Rule Repair.** RULECLEANER then generates a repair  $\Phi$  for the rules  $\mathcal{R}$  that updates the rules based on the labelled subset  $\mathcal{X}^*$ .

## 2.4 Rule Refinement

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 a subset of the datapoints  $\mathcal{X}^*$ . For a given path  $P$  in rule  $r$  ending in a leaf node  $n$ , a refinement replaces the leaf with a predicate node  $p$  with children labeled  $y_1, y_2 \in \mathcal{Y}$ .

DEFINITION 2.6 (RULE REFINEMENT). 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  $y_1, y_2$ , i.e.,

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

EXAMPLE 2.7. We show a refinement for rule #1 in Figure 1a. This is the rule version of the labeling function LF-1 from Figure 1a. Suppose we want to refine this rule to change its output on  $t_1$  ( $id=1$ ) in Table 1 to the ground truth label for  $t_1$  which is negative  $y_{t_1}^* = \text{NEG}$ . The rule for LF-1 is shown in Figure 4. Suppose we selected predicate  $p = \text{'one'}$  in  $\vee$  as the refinement (we postpone the discussion of how to come up with such a predicates to Section 4). For instance, we may add the original label  $y = \text{POS}$  as the **false** child for  $p$  and  $y^* = \text{NEG}$  as the **true** child.

## 2.5 The Rule Repair Problem

Given the user feedback on ground truth labels and rule refinement described in the previous section, we now state our desiderata for rule repairs and then formalize the rule repair problem.

**Desiderata.** Given the labelled 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 RBBM without overfitting to the small amount of labelled datapoints in  $\mathcal{X}^*$ . Specifically, we want the repair to fulfill the following desiderata:

- **Sufficient evidence for datapoints:** the repaired rules should provide sufficient evidence for each datapoint  $x_i$ , i.e., the fraction of LFs that abstain on  $x_i$  should be low.
- **Accuracy for datapoints:** the accuracy of the repaired rules that do not abstain should be high on datapoint  $x_i$ .
- **Accuracy for rules:** each repaired rule should have sufficiently high accuracy on the subset of datapoints on which it does not abstain.



Intuitively, uncertainty about an datapoint's label stems either from lack of evidence, e.g., if all rules abstain from providing a label for the datapoint then there is zero information for deciding the datapoint's label, or conflicting evidence, e.g., half of the rules that do not abstain return one label  $y_1$  on the datapoint and the other half a different label  $y_2$ . Thus, the first two desiderata ensure that the uncertainty about an datapoint's label is reduced by the repair. The third desiderata ensures that the repaired rules are effective in predicting correct labels.

We model a repair of a set of rules  $\mathcal{R}$  according to a small labeled training dataset  $\mathcal{X}^*$  as a **repair sequence**  $\Phi = \phi_1, \dots, \phi_k$  of refinement steps  $\phi_i$  and use  $\mathcal{R}^* = \{r'_1, \dots, r'_k\}$  to denote  $\Phi(\mathcal{R})$ . To avoid overfitting and to preserve the information encoded in the rules as much as possible, we want a repair that achieves our desiderata (formalized in the following) with a minimum number of changes to the results of the rules on  $\mathcal{X}^*$ .

**Datapoint Evidence.** We define the *evidence* for an datapoint  $x_i$  as the fraction of non-abstain labels ( $\neq y_0$ ) the datapoint receives from the  $m$  rules in  $\mathcal{R}$ .

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

**Datapoint Accuracy.** The *accuracy* for an datapoint  $x_i$  as shown below.

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

**Rule Accuracy.** In addition, the rules should be of “high” quality, i.e., mostly return correct results. 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.** We use a simple cost model. For a repair sequence  $\Phi$  and  $\mathcal{R}^* = \Phi(\mathcal{R})$  we define the cost of the repair 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  $\mathcal{X}^*$ :

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

We state the rule repair problem as an optimization problem: minimize the number of changes to labeling function results ( $\text{cost}(\Phi)$ ) while ensuring the desiderata of (i) sufficient evidence per datapoint: each datapoint  $x_i$  receives sufficient evidence, i.e., the fraction of non-abstain labels for  $x_i$  is above a threshold  $\tau_{\text{non-abstain}}$ , (ii) datapoint accuracy: the accuracy for each datapoint  $x_i$ , defined as the fraction of non-abstain labels for the datapoint that match the ground truth label  $y_{x_i}^*$ , is above a threshold  $\tau_{x\text{-acc}}$ , and (iii) labeling function accuracy: sufficiently the accuracy for each labeling function  $r_j$ , defined analog to the definition of accuracy for datapoints, is above a threshold  $\tau_{\text{rule-acc}}$ .

**DEFINITION 2.8 (RULE REPAIR PROBLEM).** Consider a black-box model  $\mathcal{M}_{\mathcal{R}, \mathcal{X}}$  that uses a set of rules  $\mathcal{R}$ , an input database  $\mathcal{X}$ , output labels  $\mathcal{Y}$ , and ground truth labels for a subset of datapoints  $\mathcal{X}^*$ . Given an datapoint accuracy threshold  $\tau_{x\text{-acc}} \in [0, 1]$ , datapoint non-abstain threshold  $\tau_{\text{non-abstain}} \in [0, 1]$ , and labeling function accuracy threshold  $\tau_{\text{rule-acc}} \in [0, 1]$ , the rule repair problem aims to

find a repair sequence  $\Phi$  that is a solution to the optimization problem stated below.

$$\begin{array}{lll} \text{argmin}_{\Phi} & \text{cost}(\Phi) \\ \text{subject to} & \\ \forall i \in [1, n] : & \text{Acc}(x_i) & \geq \tau_{x\text{-acc}} \\ \forall i \in [1, n] : & \text{EVIDENCE}(x_i) & \geq \tau_{\text{non-abstain}} \\ \forall j \in [1, m] : & \text{Acc}(r'_j) & \geq \tau_{\text{rule-acc}} \end{array}$$

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

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

### 3 Ruleset Repair Algorithm

In this section, we describe a generic algorithm that repairs a rule set  $\mathcal{R}$  aiming to minimize the cost and fulfill the side constraints. We demonstrate that the problem can be solved in two steps. In the first step we determine desired changes to the outputs of rules and in the second step we implement these changes by refining individual rules to returned the desired results on the datapoints in  $\mathcal{X}^*$ . We focus on the first step and then show that it is always possible to refine rules to achieve a given output on  $\mathcal{X}^*$  as long as the space of predicates is expressive enough in Section 4. We first start by formulating the problem of determining updates to the results of rules as a mixed integer linear program (MILP) that can be solved by off-the-shelf solvers. While solving MILPs is hard in general, the runtime is acceptable if  $|\mathcal{X}^*|$  is below 200 datapoints. Note that the size of the MILP encoding of the optimization problem only depends on the number of rules and number of datapoints in  $\mathcal{X}^*$ , but is independent of the size of  $\mathcal{X}$ . Furthermore, we demonstrate that the accuracy on the whole dataset  $\mathcal{X}$  can be significantly improved with a small number of labeled examples (Section 5.1). Intuitively, this is due to the fact that rules naturally generalize well beyond  $\mathcal{X}^*$  if the refinement is performed with predicates that are not tailored to specific datapoints.

#### 3.1 MILP Formulation

As mentioned above, we model the first step of determining the desired outputs of rules on  $\mathcal{X}^*$  as an MILP. We introduce an integer variable  $o_{ij}$  for each datapoint  $x_i$  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  $r'_j$  that returns updated labels on  $\mathcal{X}^*$  as dictated by the  $o_{ij}$ . We restrict

$$\begin{aligned}
& \text{minimize } \sum_i \sum_j m_{ij} \\
& \text{subject to} \\
& \forall i \in [1, n], j \in [1, m] : o_{ij} \in [0, |\mathcal{Y}| - 1] \\
& m_{ij} = \mathbb{1}[o_{ij} \neq r_j(x_i)] \\
& c_{ij} = \mathbb{1}[o_{ij} = y_{x_i}^*] \\
& e_{ij} = \mathbb{1}[o_{ij} > 0] \\
& \forall i \in [1, n] : \sum_j c_{ij} \geq \sum_j e_{ij} \cdot \tau_{x-acc} \\
& \forall i \in [1, n] : \sum_j e_{ij} \geq m \cdot \tau_{non-abstain} \\
& \forall j \in [1, m] : \sum_i c_{ij} \geq \sum_i e_{ij} \cdot \tau_{rule-acc}
\end{aligned}$$

Figure 5: MILP for the Rule Repair Problem

these variables to take values in  $[0, |\mathcal{Y}| - 1]$  where 0 represent the abstain label  $y_0$  and the values  $i$  represents the label  $y_i \in \mathcal{Y}$ . For example, assume that the possible labels are  $y_1 = \text{SPAM}$  and  $y_2 = \text{HAM}$ , then 1 encodes SPAM and 2 encodes HAM. We use a Boolean indicator variable  $m_{ij}$  that is 1 if  $o_{ij} = r_j(x_i)$ , a variable  $c_{ij}$  which is 1 if  $o_{ij} = y_{x_i}^*$ , and a variable  $e_{ij}$  which is 1 if  $o_{ij} \neq y_0$  (recall that the abstain label is encoded as 0). While such constraints are not linear constraints per se as required for MILPs, there are well known methods for encoding them as linear constraints using the so-called Big M technique [10].

Using these auxiliary variables, we can express the side constraints of the rule repair problem. Note that the number of rules and number of datapoints is a constant not a variable in these constraints. To ensure that the accuracy for each datapoint  $x_i$  is above the threshold  $\tau_{x-acc}$ , we have to ensure that out of rules that do not return the abstain label for  $x_i$ , i.e., all  $j \in [1, m]$  where  $e_{ij} = 1$ , at least a fraction of size  $\tau_{x-acc}$  have the correct label ( $c_{ij}=1$ ). This can be enforced if  $\sum_j c_{ij} - \sum_j e_{ij} \cdot \tau_{x-acc} \leq 0$  or equivalently  $\sum_j c_{ij} \geq \sum_j e_{ij} \cdot \tau_{x-acc}$ . A symmetric condition can be used to ensure the constraint on labeling function accuracy with the only difference being that the threshold  $\tau_{rule-acc}$  is used and that we are summing up over all datapoints instead of all rules. Finally, we need to ensure the condition on the minimal number of non-abstain labels for the each datapoint  $x_i$ . Recall that  $e_{ij}$  encodes whether labeling function  $r'_j$  returns a label different from  $y_0$ . Thus, this condition can be enforced using a constraint  $\forall i \in [1, n] : \sum_j e_{ij} \geq m \cdot \tau_{non-abstain}$ . The full MILP is shown in Figure 5.

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 labeling function results on  $\mathcal{X}^*$  encoded in the variables  $o_{ij}$  can be implemented. That is, as long as there exists a repair sequence  $\Phi$  such that  $\mathcal{R}^* = \Phi(\mathcal{R})$  produces  $o_{ij}$  for all  $i \in [1, n]$  and  $j \in [1, m]$ . 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 3.1 (CORRECTNESS OF THE MILP).** *Given a set of rules  $\mathcal{R}$  and ground truth labels for  $\mathcal{X}^*$  and the outputs  $o_{ij}$  produced*

*as a solution to the MILP from Figure 5, if there exists a repair sequence  $\Phi$  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 any such  $\Phi$  is a solution to the rule repair problem for  $\mathcal{R}$  and  $\mathcal{X}^*$ .*

**MILP Size.** The number of constraints and variables in the MILP is both in  $O(n \cdot m)$  where  $n = |\mathcal{X}^*|$  and  $m = |\mathcal{R}|$ .

**Example.** 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 following results on  $\mathcal{X}^*$  where abstain labels are highlighted in blue and incorrect labels are highlighted in red.

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

Let us assume that all thresholds are set to 50%. That is, each data point should receive at least two non-abstain labels, the accuracy for datapoints is at least 50% (1 correct label if the data point receives 2 non-abstain labels and 2 correct labels if it receives no abstain label), and the accuracy for rules is at least 50% (1 correct label if it returns 1 abstain label, and 2 correct labels if it returns no abstain label). The minimum number of changes required to fulfill these constraints is 4. One possible solution for the MILP is shown below with modified cells shown with black background.

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

### 3.2 Repairing Rules

Given the outputs  $o_{ij}$  of the MILP, we need to find a repair sequence  $\Phi$  such that for  $\mathcal{R}^* = \Phi(\mathcal{R}) = \{r'_1, \dots, r'_m\}$  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 others, we can solve this problem one rule at a time. Specifically, assume the existence of a function `SingleRuleRefine` that takes as input a rule  $r_j$ , the labeled subset  $\mathcal{X}^*$  of  $\mathcal{X}$ , and the expected labels for all datapoints in  $\mathcal{X}^*$ , i.e.,  $o_{*j}$ . `SingleRuleRefine` returns a repair sequence  $\Phi_j$  such that  $r'_j = \Phi_j(r_j)$  returns the expected labels on  $\mathcal{X}^*$ . We will present several algorithms for `SingleRuleRefine` in Section 4 and demonstrate that these algorithms can achieve any desired outcome on  $\mathcal{X}^*$ . The individual repair sequences can then be combined into a single sequence  $\Phi = \Phi_1 \circ \dots \circ \Phi_m$  which is guaranteed to be a solution to the rule repair problem.

## 4 Single Rule Refinement

We now discuss algorithms for `SingleRuleRefine` used to refine a single rule  $r$  and establish some important properties of rule refinement repairs. The input to single rule repair is a set of pairs  $\mathcal{Z} = \{(x_i, y_i)\}_{i=1}^n$  where  $x_i$  is an datapoint and  $y_i$  is the desired label for this datapoint. When repairing rule  $r_j \in \mathcal{R}$  to find a solution to the ruleset repair problem, the input will use the labels encoded in the result variables  $o_{ij}$  of the MILP:

$$\mathcal{Z} = \{(x_i, o_{ij}) \mid x_i \in \mathcal{X}^* \wedge j \in [1, m]\}.$$

**Algorithm 1:** SingleRuleRefine

---

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

```

1  $Y \leftarrow \emptyset, \Phi \leftarrow \emptyset$ 
2  $P_{fix} \leftarrow \{P[r, x] \mid x \in \mathcal{X}\}$ 
3  $r_{cur} \leftarrow r$ 
4 /* Iterate over paths that need to be fixed */
5 foreach  $P \in P_{fix}$  do
6   /* Fix path  $P$  to return correctly labels on  $\mathcal{Z}$  */
7    $\mathcal{X}_P \leftarrow \{(x, y) \mid (x, y) \in \mathcal{Z} \wedge P[r, x] = P\}$ 
8    $\phi \leftarrow \text{RefinePath}(r_{cur}, \mathcal{X}_P)$ 
9    $r_{cur} \leftarrow \phi(r_{cur})$ 
10   $\Phi \leftarrow \Phi.append(\phi)$ 
11 return  $\Phi$ 

```

---

We only operate on datapoints from  $\mathcal{Z}$ . We will abuse notation and use  $\mathcal{X}$  to denote  $\{x \mid (x, y) \in \mathcal{Z}\}$ . Furthermore, we will sometimes write  $\mathcal{Z}(x)$  to denote the label  $y$  associated with  $x$  in  $\mathcal{Z}$ . Before presenting the algorithm for SingleRuleRefine, we will formalize the problem of single rule refinement. As mentioned before, we search for a sequence of refinements that change the results of a rule  $r$  on  $\mathcal{X}$  to match the labels from  $\mathcal{Z}$ . To avoid over-fitting the rule to the labeled datapoints from  $\mathcal{Z}$ , the goal is to minimize the number of new predicates that are added to  $r$ . For that we define a function  $rcost$  that returns the number of new predicates added by a repair sequence  $\Phi = \phi_1, \dots, \phi_k$  for rule  $r$  which is just the number of operations in  $\Phi$ :

$$rcost(\Phi) = |\Phi|$$

**DEFINITION 4.1 (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  $\Phi_{min}$  using predicates from  $\mathcal{P}$  such that for the repaired rule  $r_{fix} = \Phi(r)$  we have:*

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

The pseudocode for SingleRuleRefine is given in Algorithm 1. Given a single rule  $r$ , this algorithm determines a refinement-based repair  $\Phi_{min}$  for  $r$  such that  $\Phi_{min}(r)$  returns the ground truth label  $\mathcal{Z}(x)$  for all datapoints specified by the user in  $\mathcal{Z}$ . We use the following notation in this section.  $\mathcal{P}$  denotes the set of predicates we are considering and  $P_{fix}$  denotes 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  $r$ .

This algorithm uses the two following properties of refinement repairs: **(1) Independence of path repairs (Section 4.0.1):** Let  $P_{fix}$  be the set of paths in  $r$  taken by datapoints from  $\mathcal{X}$ . We show that each such path can be fixed independently. **(2) Existence of path repairs (Section 4.0.2):** We show that it is always possible to repair a given path such that the desired labels of all datapoints

following this path are returned if the space of predicates  $\mathcal{P}$  satisfies a property that we call *partitioning*. **(3) RepairPath algorithms (Section 4.1):** We then present three algorithms for the RefinePath function used in Algorithm 1 to repair a single path  $P$ , i.e., refine the rule to returned the labels from  $\mathcal{Z}$  on  $\mathcal{X}$  with a trade off between runtime and repair cost that utilize these properties to repair individual paths in rule  $r$ . To ensure that all datapoints  $x$  following a given path  $P$  in the rule  $r$  get assigned their desired labels based on  $\mathcal{Z}$ , these algorithms add predicates at the end of  $P$  to “reroute” each datapoint to a leaf node with its ground-truth label. Using SingleRuleRefine with Algorithm BRUTEFORCEPATHREPAIR (one of these algorithms) solves the single rule refinement problem exactly. However, given the high computational cost of the brute force algorithm, we develop two algorithm that may return a solution that is may not be optimal wrt. its cost  $rcost$ .

**4.0.1 Independence of Path Repairs.** The first observation we make is that for refinement repairs, the problem can be divided into subproblems that can be solved independently. As refinements only extend existing paths in  $r$  by replacing leaf nodes with new predicate nodes, in any refinement  $r'$  of  $r$ , the path for a 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 (the proof is shown in Appendix C.1).

**PROPOSITION 4.2 (PATH INDEPENDENCE OF REPAIRS).** *Given a rule  $r$  and  $\mathcal{Z}$ , let  $P_{fix} = \{P_1, \dots, P_k\}$  and let  $\Phi_i$  denote a refinement-based repair of  $r$  for  $\mathcal{X}_{P_i}$  of minimal cost. Then  $\Phi = \Phi_1, \dots, \Phi_k$  is a refinement repair for  $r$  and  $\mathcal{Z}$  of minimal cost.*

**4.0.2 Existence of Path Refinement Repairs.** Next, we will show that is always possible to find a refinement repair for a path if the space of predicates  $\mathcal{P}$  is *partitioning*, i.e., if for any two 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  $w$  (we formally state this claim Lemma C.1 in Appendix C.2).

The following proposition shows that when  $\mathcal{P}$  is partitioning, we can always find a refinement repair. Further, we show an upper bound on the number of predicates to be added to a path  $P \in P_{fix}$  to assign the ground truth labels  $\mathcal{Z}_P$  to all datapoints in  $\mathcal{X}_P$ .

**PROPOSITION 4.3 (UPPER BOUND ON REPAIR COST OF A PATH).** *Consider a rule  $r$ , a path  $P$  in  $r$ , a partitioning predicate space  $\mathcal{P}$ , and ground truth labels  $\mathcal{Z}_P$  for datapoints  $\mathcal{X}_P$  on path  $P$ . Then there exists a refinement repair  $\Phi$  for path  $P$  and  $\mathcal{Z}_P$  such that:  $cost(\Phi) \leq |\mathcal{X}_P|$ .*

**PROOF SKETCH.** Our proof is constructive: we present an algorithm that, given a set of datapoints  $\mathcal{X}_P$  for a path  $P$  ending in a node  $n$  iteratively selects two datapoints  $x_1, x_2 \in \mathcal{X}_P$  such that

$\mathcal{Z}_P(x_1) \neq \mathcal{Z}_P(x_2)$  and adds a predicate that splits  $\mathcal{X}_P$  into  $\mathcal{X}_P^1$  and  $\mathcal{X}_P^2$  such that  $x_1 \in \mathcal{X}_P^1$  and  $x_2 \in \mathcal{X}_P^2$ . The full proof is shown in Appendix C.3.  $\square$

Of course, as we show in Appendix C.4, an optimal repair of  $r$  wrt.  $\mathcal{X}_P$  may require less than  $|\mathcal{X}_P|$  refinement steps. Nonetheless, this upper bound will be used in the brute force algorithm we present in Section 4.1.2 to bound the size of the search space. The above proposition only guarantees that repairs with a bounded cost exist. However, the space of predicates may be quite large or even infinite. Thus, it is not immediately clear how to select a separating predicate for any two given datapoints  $x_1$  and  $x_2$ . Note that if comparisons between atomic units and constants are included in  $\mathcal{P}$ , we can simply find in polynomial time two data points, say  $x_1$  and  $x_2$ , and select  $v[A] = c$  such that  $x_1[A] = c \neq x_2[A]$ . More generally, in Appendix C.5 we show that we can determine equivalence classes of predicates in  $\mathcal{P}$  wrt. a set of datapoints and use only a single member from each equivalence class.

#### 4.1 Path Repair Algorithms

Based on the insights presented in the previous sections, we now present three algorithms for `RefinePath` used in Algorithm 1 to fix a given path  $P \in P_{fix}$ . These algorithms return a sequence of refinements  $\Phi$  for rule  $r$  such that the repaired rule  $r' = \Phi(r)$  returns the desired from  $\mathcal{Z}_P$  for all datapoints  $x \in \mathcal{X}_P$ . These algorithms refine  $r$  by replacing  $last(P)$  with a subtree to generate a refinement repair  $\Phi$ . As shown in Appendix C.5, we only need to consider a finite number of predicates specific to  $\mathcal{X}_P$ . We first briefly explain a greedy algorithm that often requires large number of new predicates to be added, then a brute force exponential time algorithm that generates a repair with the minimal possible number of new predicates, and then our entropy-based algorithm that greedily chooses predicates that minimize an entropy-based measure.

**4.1.1 GreedyPathRepair.** This algorithm (pseudocode in Appendix D.1) 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 an 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 Proposition 4.3, this algorithm terminates after adding at most  $|\mathcal{X}_P|$  new predicates.

**4.1.2 BruteForcePathRepair.** The brute-force algorithm (pseudocode in Appendix D.1) is optimal, i.e., it returns a refinement of minimal cost (number of new predicates added). This algorithm enumerates all possible refinement repairs for a path  $P$ . Each such repair corresponds to replacing the last element on  $P$  with some rule tree. We enumerate such trees in increasing order of their size and pick the smallest one that achieves perfect accuracy on  $\mathcal{Z}_P$  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

---

#### Algorithm 2: EntropyPathRepair

---

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

```

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

---

rule from the queue and extend it in all possible ways by replacing one leaf node, and select the refined rule with minimum cost that satisfies all assignments. As we generate subtrees in increasing size, Proposition 4.3 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  $n$ .

**4.1.3 EntropyPathRepair.** GreedyPathRepair is fast but has the disadvantage that it may use overly specific predicates that do not generalize well (even just on  $\mathcal{X}_P$ ). Furthermore, by randomly selecting predicates to separate two datapoints (ignoring all other datapoints for a path), this algorithm will often yield repairs with a suboptimal cost. In contrast, BruteForcePathRepair produces minimal repairs that also generalize better, but the exponential runtime of the algorithm limits its applicability in practice. We now introduce EntropyPathRepair, a more efficient algorithm that avoids the exponential runtime of BruteForcePathRepair while typically producing more general and less costly repairs than GreedyPathRepair. We achieve this by greedily selecting predicates that best separate datapoints with different labels at each step.

Note that in the context of this algorithm we denote the input path as  $P_{in}$  instead of  $P$  and reserve  $P$  for the current path processed in one iteration of the algorithm. To measure the quality of a split, we employ the entropy-based *Gini impurity score*  $I_G$  [17]. 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  $p$ :

$$\mathcal{Z}_{\text{false}} = \{(x, y) \mid (x, y) \in \mathcal{Z}_P \wedge \neg p(x)\}$$

$$\mathcal{Z}_{\text{true}} = \{(x, y) \mid (x, y) \in \mathcal{Z}_P \wedge p(x)\}$$

Using  $\mathcal{Z}_{\text{false}}$  and  $\mathcal{Z}_{\text{true}}$  we define  $I_G(\mathcal{Z}_P, p)$  for a predicate  $p$  as shown below:

$$I_G(\mathcal{Z}_P, p) = \frac{|\mathcal{Z}_{\text{false}}| \cdot I_G(\mathcal{Z}_{\text{false}}) + |\mathcal{Z}_{\text{true}}| \cdot I_G(\mathcal{Z}_{\text{true}})}{|\mathcal{Z}_P|}$$

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

Note that 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 again 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 create the subsets of datapoints from  $\mathcal{X}_P$ , which contains datapoints fulfilling  $p$  and those that do not. We then generate a refinement repair step  $\phi_{\text{new}}$  for the current version of the rule ( $r_{\text{cur}}$ ) that replaces the last element on  $P_{\text{cur}}$  with predicate  $p$ . The child at the **true** edge of the node for  $p$  is then assigned the most prevalent label  $y_{\text{max}}$  for the datapoints that will end up in this node (the datapoints from  $\mathcal{Z}_{\text{true}}$ ). Finally, unless they only contain one label, new entries for  $\mathcal{Z}_{\text{false}}$  and  $\mathcal{Z}_{\text{true}}$  are appended to the todo queue.

**4.1.4 Correctness.** The following theorem shows that all three path repair algorithms are correct (proof in Appendix D.2).

**THEOREM 4.4 (CORRECTNESS).** *Consider a rule  $r$ , ground-truth labels of a set of datapoints  $\mathcal{Z}_P$ , and partitioning space of predicates  $\mathcal{P}$ . Let  $\Phi$  be the repair sequence produced by `GreedyPathRepair`, `BruteForcePathRepair`, or `EntropyPathRepair` for path  $P$ . Then we have:*

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

## 5 Experiments

In this section, we evaluate runtime of `RULECLEANER` and its effectiveness in improving the accuracy of rules produced by Witan [7] and compare it against a baseline that uses a large language model to repair labeling functions. Furthermore, we evaluate the trade-offs for the three path repair algorithms we propose in this work. `RULECLEANER` is implemented in Python (version 3.8) and runs on top of PostgreSQL (version 14.4). Experiments were run on Oracle Linux Server 7.9 with 2 x AMD EPYC 7742 CPUS, 128GB RAM. We evaluate `RULECLEANER` for weakly-supervised labeling (LF) using `Snorkel` [23] as the RBBM. We evaluate both the runtime and the quality of the refinements produced by our system with respect to several parameters.

dataset	#row	avg #word	$\mathcal{Y}$	# LFs
<i>Amazon</i>	200000	68.9	positive/negative	15
<i>AGnews</i>	60000	37.7	business/technology	9
<i>PP</i>	54476	55.8	physician/professor	18
<i>IMDB</i>	50000	230.7	positive/negative	7
<i>FNews</i>	44898	405.9	true/false	11
<i>Yelp</i>	38000	133.6	negative/positive	8
<i>PT</i>	24588	62.2	professor/teacher	7
<i>PA</i>	12236	62.6	painter/architect	10
<i>Tweets</i>	11541	18.5	positive/negative	16
<i>SMS</i>	5572	15.6	spam/ham	17
<i>MGenre</i>	1945	26.5	action/romance	10

Table 2: LF dataset statistics.

**Datasets and rules.** The datasets used in the experiments are listed in Table 2. We give a brief description on each dataset: *Amazon*: product reviews from Amazon and their sentiment label [15]. *AGnews*: categorized news articles from AG's corpus of news article [34]. *PP*: descriptions of biographies, each labeled as a physician or a professor [6]. *IMDB*: IMDB movie reviews [19]. *FNews*: Fake news identification [1]. *Yelp*: Yelp reviews [34]. *PT*: descriptions of individuals, each labeled as a professor or a teacher.[6]. *PA*: descriptions of individuals each labeled a painter or an architect. [6]. *Tweets*: classification of tweets on disasters [20]. *SMS*: classification of SMS messages [2]. *MGenre*: movie genre classification based on plots [28]. The rules we use in the experiments are generated using Witan [7]. Based on preliminary experiments where we compared the 3 path repair algorithms, `EntropyPathRepair` is the best considering the trade-off between runtime and quality. Unless stated otherwise, the experiments in this section are run with `EntropyPathRepair`. We present a detailed evaluation of these 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. Given the limited scalability of MILP solvers in the number of variables, we used at most  $|\mathcal{X}^*| = 150$  data points. The results are shown in Figure 6. The labeled data points 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 6 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  $\tau_{\mathcal{X}-acc}$ ,  $\tau_{non-abstain}$ ,  $\tau_{rule-acc}$  and new global accuracy. We used *Tweets* with and 20 labeled examples. The effects on global accuracy of

the pairwise relationships of  $\tau_{x-acc}$ ,  $\tau_{non-abstain}$ , and  $\tau_{rule-acc}$  are shown in Figure 7. 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 2.5. 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  $\mathcal{X}^*$ . Based on our experience we recommend setting all of the thresholds to  $\sim 0.7$ .

## 5.2 Runtime

Runtime breakdowns for the experiments from Section 5.1 are shown in Figure 8. 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 data points are randomly sampled from  $\mathcal{X}$  and the complexity in 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 example result in more complex rules even when the sample size is small, increasing the time required for Snorkel to fit a model.

## 5.3 LLM vs RULECLEANER

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 data points. 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 can be found in Appendix E.

We manually inspected the rules returned by the LLM to ensure that they are semantically meaningful. The quality of results are shown in Table 3. *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. We observe that the LLM tends to 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 is `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

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

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

to refine a rule with negative sentiment by adding `else if 'great' in text then POSITIVE else ABSTAIN`

## 5.4 Path Repair Algorithms

Next, we evaluate the path repair algorithms discussed in Section 4.1. In this experiment we used the *Tweets* dataset and randomly selected between 2 and 10 labeled examples. The input datapoints to each path repair algorithm are the same for each input size. We show the repaired rule size in terms of number of nodes in a tree and the runtime in Figure 9. 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. It is obvious that the runtime for brute force 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 RBMMs as well as discuss approaches for automatically generating rules for RBMMs and improving a given rule set.

**Weak supervision & data programming.** Weak supervision is a general technique of learning from noisy supervision signals that has been applied in many contexts [9, 22–25, 28], e.g., for data labeling to generate training data [23, 24, 28] (the main use case we target in this work), for data repair [25], and for entity matching [22]. The main advantage of weak supervision is that it reduces the effort of creating training data without ground truth labels. The data programming paradigm pioneered in Snorkel [23] has the additional advantage that the rules used for labeling are interpretable. However, as such rules are typically noisy heuristics, systems like Snorkel combine the output of LFs using a model.

**Automatic generation and fixing labeling functions.** While the data programming paradigm proves to be effective, asking human annotators to create a large set of high-quality labeling functions requires domain knowledge, programming skills, and time. To this end, automatically generating or improving labeling heuristics has received much attention from the research community.

*Witan* [7] is a system for automatically generating labeling functions. While the labeling functions produced by Witan are certainly useful, we demonstrate in our experimental evaluation that applying RULECLEANER to Witan LFs can significantly improve

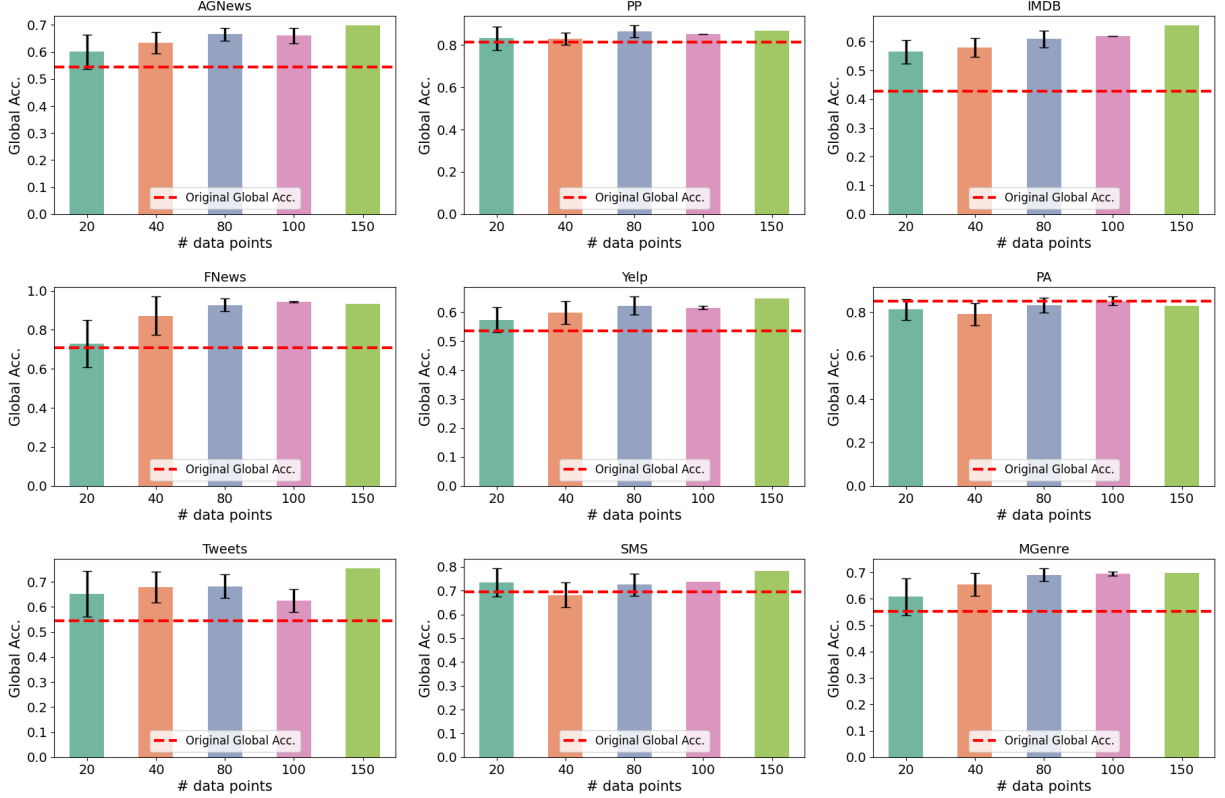


Figure 6: Change in global accuracy, varying labeled dataset size

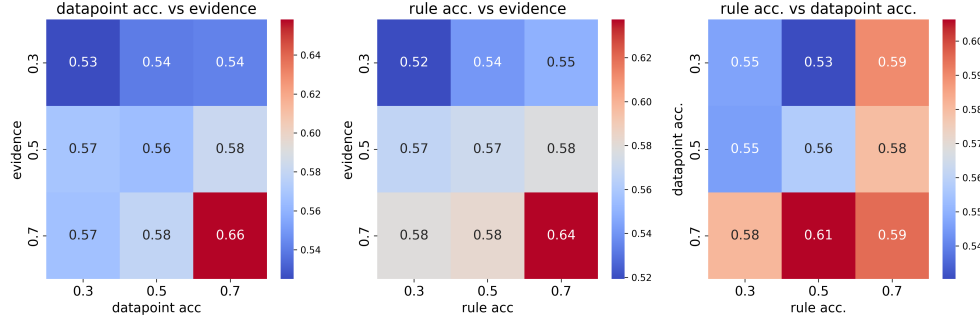


Figure 7: Pairwise interaction heatmaps for New Global Acc.

accuracy. Starting from a seed set of LFs, *Darwin* [8] generates heuristic LFs under a context-free grammar and uses a hierarchy to capture the containment relationship between LFs that helps determine which to be verified by users. *IWS* [3] also selects n-gram-based LFs according to the expert’s annotation on the usefulness of the LF. Unlike *RULECLEANER* that also repairs LFs based on user feedback, *Darwin* will only use the user feedback to update its LF scoring model. *Snuba* [28] fits classification models like decision trees and logistic regressors as LFs on a small labeled training set, followed by a pruner to determine which LFs to finally use. *Datasculpt* [12] uses a large language model (LLM) to generate labeling functions. Given a small set of training data with known ground

truth, the system prompts LLM with in-context examples of labels and keyword-based or pattern-based LFs to generate LFs for an unlabeled example.

Hsieh et al. [16] propose a framework called *Nemo* for selecting data to show to the user for labeling function generation based on a utility metric for LFs and a model of user behavior (given some data how likely is the user to propose a particular LF). Furthermore, *Nemo* specializes LFs to be applicable only to the neighborhood of data (user-developed LFs are likely more accurate to data similar to the data based on which they were created). However, in contrast to *RULECLEANER*, *Nemo* does not provide a mechanism for the user to provide feedback on the result of weakly-supervised training

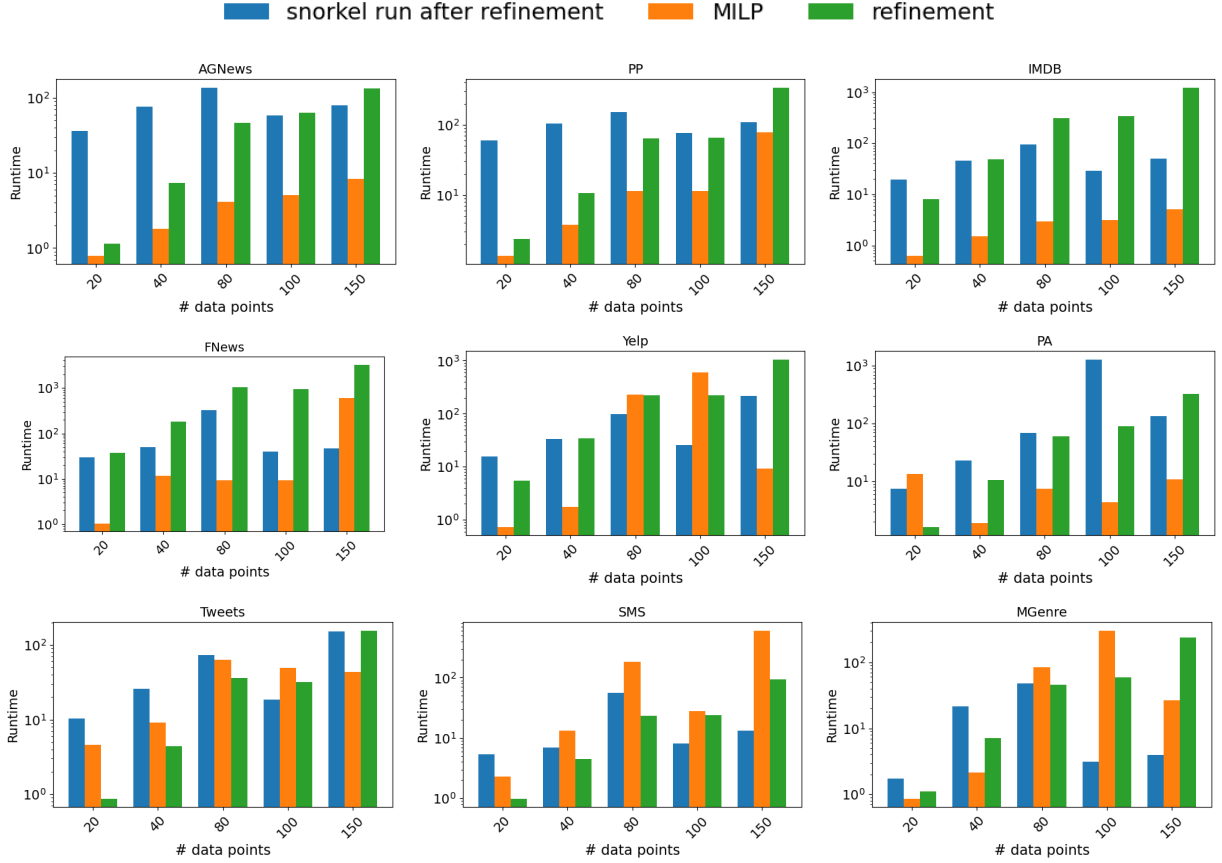


Figure 8: Runtime results, varying training set size

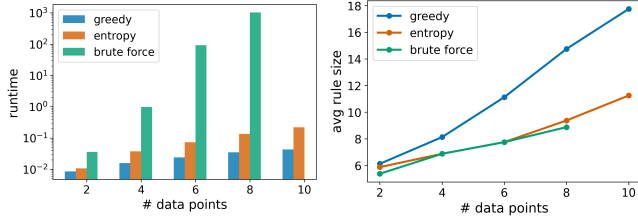


Figure 9: Comparison of 3 path repair algorithms

data generation and to use this information to automatically delete and refine rules. *ULF* [27] is an unsupervised system for fixing labeling functions using k-fold cross-validation, extending previous approaches aimed at compensating for labeling errors [21, 29].

**Explanations for weakly supervised systems.** While there is a large body of work on explaining the results of weak-supervised systems that target improving the final model or better involving human annotators in data programming [3, 5, 30, 32, 33], most of this work has stopped short of repairing the rules of a RBBM and, thus, are orthogonal to our work. However, it may be possible to utilize the explanations provided by such systems to guide a user in

selecting what datapoints to label. People have also studied using human-annotated natural language explanations to build LFs [13].

## 7 Conclusions and Future Work

In this work, we introduced RBBMs, a general model for systems that combine a set of interpretable rules with a model that combines the outputs produced by the rules to predict labels for a set of datapoints in a weakly-supervised fashion. Many important applications can be modeled as RBBMs, including weakly-supervised labeling. We develop a human-in-the-loop approach for repairing a set of RBBM rules based on a small set of ground truth labels generated by the user. Our algorithm is highly effective in improving the accuracy of RBBMs by improving rules created by a human expert or automatically discovered by a system like Witan [7]. In future work, we will explore the application of our rule repair algorithms to other tasks that can be modeled as RBBM, e.g., information extraction based on user-provided rules [18, 26]. In this work, we used ground truth labels for both datapoints and assignments. It will be interesting to investigate whether the rules can be repaired based on the ground truth of only the datapoints. Furthermore, it would be interesting to use explanations for rule outcomes to



guide the user in what datapoints to inspect and to aide the system in selecting which rules to repair, e.g., repair rules that have high responsibility for a wrong result.

## References

- [1] 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*, Matthew R. B. Hardy and Frank Wm. Tompa (Eds.). ACM, 259–262.
- [3] 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*.
- [4] Benedikt Boecking, Willie Neiswanger, Eric P. Xing, and Artur Dubrawski. 2021. Interactive Weak Supervision: Learning Useful Heuristics for Data Labeling. In *ICLR*.
- [5] 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).
- [6] Maria De-Arteaga, Alexey Romanov, Hanna M. Wallach, Jennifer T. Chayes, Christian Borgs, Alexandra Chouldechova, Sahin Cem Geyik, Krishnamurthy Ken- thapadi, 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.
- [7] Benjamin Denham, Edmund M.-K. Lai, Roopak Sinha, and M. Asif Naeem. 2022. Witan: Unsupervised Labelling Function Generation for Assisted Data Program- ming. *PVLDB* 15, 11 (2022), 2334–2347.
- [8] Sainyam Galhotra, Behzad Golshan, and Wang-Chiew Tan. 2021. Adaptive Rule Discovery for Labeling Text Data. In *SIGMOD*. 2217–2225.
- [9] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC Data-Cleaning Framework. *PVLDB* 6, 9 (2013), 625–636.
- [10] Igor Griva, Stephen G Nash, and Ariela Sofer. 2008. *Linear and Nonlinear Opti- mization 2nd Edition*. SIAM.
- [11] Naiqing Guan, Kaiwen Chen, and Nick Koudas. 2023. Can Large Language Models Design Accurate Label Functions? *CoRR* abs/2311.00739 (2023). arXiv:2311.00739
- [12] Naiqing Guan, Kaiwen Chen, and Nick Koudas. 2023. Can Large Language Models Design Accurate Label Functions? *CoRR* abs/2311.00739 (2023). arXiv:2311.00739
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] Sotiris B. Kotsiantis. 2013. Decision Trees: a Recent Overview. *Artif. Intell. Rev.* 39, 4 (2013), 261–283.
- [18] B. Liu, L. Chiticariu, V. Chu, HV Jagadish, and F.R. Reiss. 2010. Automatic Rule Refinement for Information Extraction. *PVLDB* 3, 1 (2010).
- [19] 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.
- [20] Hussein Mouzannar, Yara Rizk, and Mariette Awad. 2018. Damage Identification in Social Media Posts using Multimodal Deep Learning. In *International Confer- ence on Information Systems for Crisis Response and Management*, Kees Boersma and Brian M. Tomaszewski (Eds.). ISCRAM Association.
- [21] 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.
- [22] Fatemah Panahi, Wentao Wu, AnHai Doan, and Jeffrey F Naughton. 2017. Towards Interactive Debugging of Rule-based Entity Matching. In *EDBT*. 354–365.
- [23] 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.
- [24] 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.
- [25] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* 10, 11 (2017), 1190–1201.
- [26] Sudeepa Roy, Laura Chiticariu, Vitaly Feldman, Frederick R Reiss, and Huaiyu Zhu. 2013. Provenance-based dictionary refinement in information extraction. In *SIGMOD*. 457–468.
- [27] Anastasiya Sedova and Benjamin Roth. 2022. ULF: Unsupervised Labeling Function Correction using Cross-Validation for Weak Supervision. *CoRR* abs/2204.06863 (2022).
- [28] Paroma Varma and Christopher Ré. 2018. Snuba: Automating Weak Supervision to Label Training Data. *PVLDB* 12, 3 (2018), 223–236.
- [29] 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.
- [30] Peilin Yu and Stephen Bach. 2023. Alfred: A System for Prompted Weak Supervi- sion. *arXiv preprint arXiv:2305.18623* (2023).
- [31] 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
- [32] 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.
- [33] Xiaoyu Zhang, Xiwei Xuan, Alden Dima, Thurston Sexton, and Kwan-Liu Ma. 2023. LabelVizier: Interactive Validation and Relabeling for Technical Text An- notations. In *2023 IEEE 16th Pacific Visualization Symposium (PacificVis)*. IEEE, 167–176.
- [34] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level Convo- lutional 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 2.2).

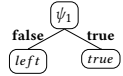
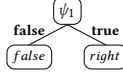
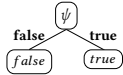
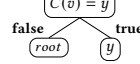
As mentioned before, we support arbitrary labeling functions written in a general purpose programming language. Our imple- mentation of the translation into rules is for Python functions as supported in Snorkel. Detecting the if-then-else rule structure and logical connectives that are supported in our 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 any LF written in Python is we can live if we treat ev- ery code blocks 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 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 a LF into a tree with multiple predi- cates: (i) such a rule will make explicit the logic of the LF and, thus, may be easier to interpret by a user 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 men- tioned 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 la- bel 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 pro- gramming can be decomposed by our approach. Nonetheless, our approach can easily be extended to support additional structures if need be.

**Algorithm 3:** Convert LF to Rule

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

```

1  $V, E = \emptyset, \emptyset;$ 
2  $r_f \leftarrow (V, E);$ 
3 Let  $C$  be the source code of  $f$ ;
4 LF-to-Rule( $r, C$ ):
5   if  $C = \text{if } \text{cond} : B1 \text{ else} : B2 \wedge \text{ispure}(\text{cond}) \wedge$ 
      $\text{purereturn}(B1) \wedge \text{purereturn}(B2)$  then
6      $v_{\text{true}} \leftarrow \text{LF-to-Rule}(B1)$ 
7      $v_{\text{false}} \leftarrow \text{LF-to-Rule}(B2)$ 
8      $n_{\text{root}} \leftarrow \text{Pred-To-Rule}(\text{cond}, v_{\text{false}}, v_{\text{true}})$ 
9   else if  $C = \text{return } y \wedge y \in \mathcal{Y}$  then
10     $n_{\text{root}} = y$ 
11   else
12     $n_{\text{root}} \leftarrow \text{Translate-BBox}(C)$ 
13   return  $n_{\text{root}}$ 
14 Pred-to-Rule( $\psi, n_{\text{false}}, n_{\text{true}}$ ):
15   if  $\psi = \psi_1 \vee \psi_2$  then
16      $n_{\text{left}} \leftarrow \text{Pred-to-Rule}(\psi_2, n_{\text{false}}, n_{\text{true}})$ 
17      $n_{\text{root}} \leftarrow$  
18   else if  $\psi = \psi_1 \wedge \psi_2$  then
19      $\text{right} \leftarrow \text{Pred-to-Rule}(\psi_2, n_{\text{false}}, n_{\text{true}})$ 
20      $n_{\text{root}} \leftarrow$  
21   else
22      $n_{\text{root}} \leftarrow$  
23   return  $n_{\text{root}}$ 
24 Translate-BBox( $C$ ):
25    $\text{root} = y_0;$ 
26   for  $y \in \mathcal{Y} \setminus \{y_0\}$  do
27      $n_{\text{root}} \leftarrow$  
28   return  $n_{\text{root}}$ 

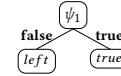
```

**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 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  $\psi$  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:

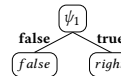
(i) both  $B1$  and  $B2$  have to be pure, i.e., they are side-effect free, and return a label for every input. This is checked using function **purereturn**. This is necessary to ensure that we can translate  $B1$  and  $B2$  into rule fragments that return a label. Furthermore, **cond** has to be 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 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).

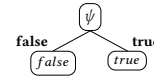
**Translating predicates.** **Pred-to-Rule**, our function for translating predicates (Boolean conditions as used in if statements), takes as input a condition  $\psi$  and the roots of two rule subtrees ( $n_{\text{false}}$  and  $n_{\text{true}}$ ) that should be used to determine an inputs label based on whether  $\psi$  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 that is the case, we decompose the condition and create an appropriate rule fragment implementing the disjunction (conjunction). For disjunctions  $\psi_1 \vee \psi_2$ , the result of  $n_{\text{true}}$  should be returned if  $\psi_1$  evaluates to true. Otherwise, we have to check  $\psi_2$  to determine whether  $n_{\text{false}}$ 's or  $n_{\text{true}}$ 's result should be returned. For that we generate a rule fragments as shown below where  $n_{\text{left}}$  denote the root of the rule tree generated by calling **Pred-to-Rule** to translate  $\psi_2$ .



The case for conjunctions is analog. If  $\psi_1$  evaluates to false we have to return the result of  $n_{\text{false}}$ . Otherwise, we have to check  $\psi_2$  to determine whether to return  $n_{\text{false}}$ 's or  $n_{\text{true}}$ 's result. This is achieved using the rule fragment shown below where  $n_{\text{right}}$  denotes the root of the tree fragment generated for  $\psi_2$  by calling **Pred-to-Rule** on  $\psi_2$ .



If  $\psi$  is neither a conjunction nor disjunction, then we just add predicate node for the whole condition  $\psi$ :



**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 10 shows the structure of the generated rule tree produced by **Translate-BBox** for a set of labels  $\mathcal{Y} = \{y_1, \dots, y_k, \text{ABSTAIN}\}$  where **ABSTAIN** is the default label ( $y_0$ ).

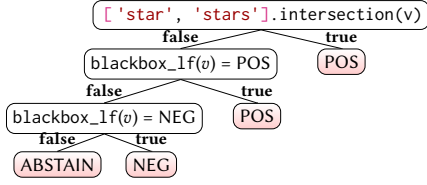


Figure 11: Translating a LF wrapping parts into a blackbox function

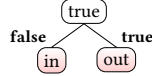


Figure 12: The rule representation of rule  $r$  used in Appendix B

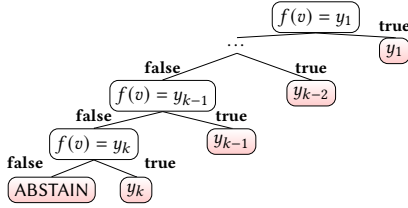


Figure 10: Translating a blackbox function

As mentioned above this translation process produces a valid rule tree  $f$  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 A.1. *Let  $f$  be a python labeling function and let  $r_f = \text{LF-to-Rule}(f)$ . Then for all datapoints  $x$  we have*

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

Function  $\text{LF-to-Rule}$ 's runtime is linear in the size of  $f$ .

PROOF SKETCH. The result is proven through induction over the structure of a labeling function.  $\square$

EXAMPLE A.2 (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. 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 11. 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 2.9

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

- Database: a single table  $R(E)$  with single attribute  $E$ . We consider  $n + 1$  tuples (datapoints)  $\{(e_1), \dots, (e_n), (b)\}$  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  $C_{\text{false}}(p) = in$  and  $C_{\text{true}}(p) = out$  (Figure 12). Hence initially any assignment will end up in  $C_{\text{true}}(p) = out$ .
- 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  $r$ ).
- We disallow deletions (i.e., set  $\tau_{del} = \infty$ ).
- The space of predicates is  $\mathcal{P} = \{v \in S_i \mid i \in [1, m]\}$
- The accuracy threshold is  $\tau_{acc} = 1$ . In other words, we want the correct classification of all datapoints after repairing  $r$ , i.e., for all  $e_i$ ,  $i = 1 \dots n$ , the updated rule should output *in*, and for  $b$ , the updated rule should still output *out*.

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

(only if): Let  $S_{i_1}, \dots, S_{i_k}$  be a set cover. We have to show that there exists a minimal repair  $\Phi$  of size  $\leq k$ . We construct that repair as follows: we replace the **true** child of the single predicate  $p : (v = v)$  in  $r$  with a left deep tree with predicates  $p_{i_j}$  for  $j \in [1, k]$  such that  $p_{i_j}$  is  $(v \in S_{i_j})$  and  $C_{\text{false}}(p_{i_j}) = p_{i_{j+1}}$  unless  $j = k$  in which case  $C_{\text{false}}(p_{i_j}) = out$ ; for all  $j$ ,  $C_{\text{true}}(p_{i_j}) = in$ . The rule tree for this rule is shown in Figure 13 The repair sequence  $\Phi = \phi_1, \dots, \phi_k$  has cost  $k$ . Here  $\phi_i$  denotes the operation that introduces  $p_{i_j}$ .

It remains to be shown that  $\Phi$  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

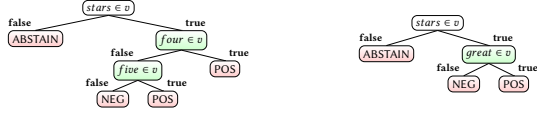


Figure 14: A non-optimal rule repair for the DC from Example C.2 produced by the algorithm from Proposition 4.3 and an optimal repair (right)

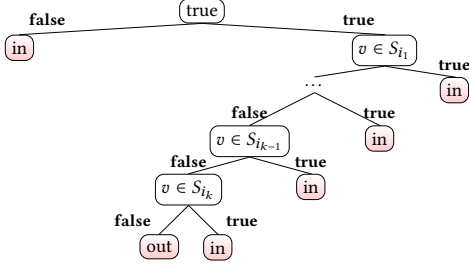


Figure 13: The rule representation of repair of rule  $r$  in Appendix B

$M(e) = r(e)$  and, thus,  $r_{up}(e) = Z(e)$  ensures that  $M(e) = Z(e)$ . Since the model corresponds a single rule, we want  $r_{up}(e_i) = in$  for all  $i = 1, \dots, n$ , 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  $j = 1, \dots, k$ . Since  $S_{i_1}, \dots, S_{i_k}$  constitute a set cover, for every  $e_i, i = 1, \dots, n$ , the check will be true for at least one  $S_{i_j}$ , resulting in a final label of *in*. On the other hand, the check will be false for all  $S_{i_j}$  for  $b$ , and therefore the final label will be *out*. This gives a refined rule with accuracy 1.

(if): Let  $\Phi$  be a minimal repair of size  $\leq k$  giving accuracy 1. Let  $r_{up} = \Phi(r)$ , i.e., in the repaired rule  $r_{up}$ , all  $e_i, i = 1, \dots, n$  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  $p(b) = \text{false}$ .

Let us consider the path  $P = root \xrightarrow{b_1} \dots \xrightarrow{b_{l-1}} \dots \xrightarrow{b_l}$  taken by  $b$ .

As  $r_{root} = p_{root} = \text{true}$ , we know that  $b_1 = \text{true}$  ( $b$  takes the **true** edge of  $r_{root}$ ). 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  $b_i = \text{false}$  for  $i > 1$ . As we have  $(b) = out$  and  $\Phi$  is a repair, we know that  $l = out$ . 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  $r_{up}(e) = out \neq in = (e)$  contradicting the fact that  $\Phi$  is a repair. That is, for each  $e \in U$  there exists  $S_i$  such  $e \in S_i$  and  $p_i : v \in S_i$  appears in the tree of  $r_{up}$ . Thus,  $\{S_i \mid p_i \in r_{up}\}$  is a set cover of size  $\leq k$ .  $\square$

## C Single Rule Refinement - Proofs and Additional Details

### C.1 Proof of Proposition 4.2

PROOF OF PROPOSITION 4.2. The claim can be shown by contradiction. Assume that there exist a repair  $\Phi = \Phi_1, \dots, \Phi_k$ , but  $\Phi$  is not optimal. That is, there exists a repair  $\Phi'$  with a lower cost. First

off, it is easy to show that  $\Phi'$  does not refine any paths  $p \notin P_{fix}$  as based on our observation presented above any such refinement does not affect the label of any assignment in  $\Lambda_{C^\wedge}$  and, thus, can be removed from  $\Phi'$  yielding a repair of lower costs which contradicts the fact that  $\Phi'$  is optimal. However, then we can partition  $\Phi'$  into refinements  $\Phi'_i$  for each  $P_i \in P_{fix}$  such that  $\Phi' = \Phi'_1, \dots, \Phi'_k$ . As  $cost(\Phi') < cost(\Phi)$  there has to exist at least on path  $P_i$  such that  $cost(\Phi'_i) < cost(\Phi_i)$  which contradicts the assumption that  $\Phi_i$  is optimal for all  $i$ . Hence, no such repair  $\Phi'$  can exist.  $\square$

### C.2 Partitioning Predicate Spaces

LEMMA C.1. Consider a space of predicates  $\mathcal{P}$  and atomic units  $\mathcal{A}$ .

- **Atomic unit comparisons:** If  $\mathcal{P}$  contains for every  $a \in \mathcal{A}$ , constant  $c$ , and variable  $v$ , the predicate  $v[a] = c$ , then  $\mathcal{P}$  is partitioning.
- **Labeling Functions:** Consider the document labeling use-case. 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  $\mathcal{P}$ .

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

**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.  $\square$

### C.3 Proof of Proposition 4.3

PROOF OF PROPOSITION 4.3. 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  $x \in X_P$  the correct label. The algorithm starts with  $X_{cur} = X_P$  and in each step finds a predicate  $p$  that “separates” two datapoints  $x_1$  and  $x_2$  from  $X_{cur}$  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  $X_1 = \{x \mid x \in X_{cur} \wedge p(x)\}$  and  $X_2 = \{x \mid x \in X_{cur} \wedge \neg p(x)\}$ . We know that  $x_1 \in X_1$  and  $x_2 \in X_2$ . That means that  $|X_1| < |X_P|$  and  $|X_2| < |X_P|$ . The algorithm repeats this process for  $X_{cur} = X_1$  and  $X_{cur} = X_2$  until all datapoints in  $X_{cur}$  have the same desired label which is guaranteed to be the case if  $|X_{cur}| = 1$ . 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  $X_{cur}$  is reduced by at least one, the algorithm will terminate after adding at most  $|X_P|$  predicates.  $\square$

### C.4 Non-minimality of the Algorithm from Proposition 4.3

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

EXAMPLE C.2. Consider  $X$  as shown below with 3 documents, their current labels (NEG) assigned by a rule and expected labels from  $Z$ .



**Algorithm 4:** GreedyPathRepair

---

**Input** : Rule  $r$   
 Path  $P$   
 datapoints to fix  $X_P$   
 Expected labels for assignments  $Z_P$

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

---

```

1 todo  $\leftarrow [(P, Z_P)]$ 
2  $\Phi = []$ 
3 while todo  $\neq \emptyset$  do
4    $(P, Z_P) \leftarrow \text{pop}(\text{todo})$ 
5   if  $\exists x_1, x_2 \in X : Z_P[x_1] \neq Z_P[x_2]$  then
6     /* Determine predicates that distinguishes
       assignments that should receive different labels
       for a path */
7      $p \leftarrow \text{GetSeperatorPred}(x_1, x_2)$ 
8      $y_1 \leftarrow Z_P[x_1]$ 
9      $\phi \leftarrow \text{refine}(r_{\text{cur}}, P, y_1, p, \text{true})$ 
10     $X_1 \leftarrow \{x \mid x \in P \wedge p(x)\}$ 
11     $X_2 \leftarrow \{x \mid x \in P \wedge \neg p(x)\}$ 
12    todo.push $((P[r_{\text{cur}}, x_1], X_1))$ 
13    todo.push $((P[r_{\text{cur}}, x_2], X_2))$ 
14  else
15     $x \leftarrow X_P.\text{pop}()$  /* All  $x \in Z_P$  have same label */
16     $\phi \leftarrow \text{refine}(r_{\text{cur}}, P, Z_P[x])$ 
17     $r_{\text{cur}} \leftarrow \phi(r_{\text{cur}})$ 
18     $\Phi.\text{append}(\phi)$ 
19 return  $\Phi$ 

```

---

The original rule consists of a single predicate  $\text{stars} \in v$ , assigning all documents that contain the word “stars” the label NEG. The algorithm may repair the rule by first adding the predicate  $\text{four} \in v$  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.,  $\text{five} \in v$ . The resulting rule is shown Figure 14 (left). The cost of this repair is 2. However, a repair with a lower costs exists: adding the predicate  $\text{great} \in v$  instead. This repair has a cost of 1. The resulting rule is shown in Figure 14 (right).

sentence	Current label	Expected labels from $Z$
$d_1$ : 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.  $X_P$  can be exploited to reduce the search space of predicates and present an algorithm that is exponential in  $|X_P|$  which determines an optimal repair independent of the size of the space of all possible predicates. First observe that with  $|X_P| = n$ , there are exactly  $2^n$  possible outcomes of applying a predicate  $p$  to  $X_P$  (returning either true or false for each  $x \in X_P$ ). Thus, with respect to the task of repairing a rule through refinement to return the correct label for each  $x \in X_P$ , two predicates are equivalent if they return the same result on  $\Lambda$  in the sense that in any repair using a predicate  $p_1$ , we

can substitute  $p_1$  for a predicate  $p_2$  with the same outcome and get a repair with the same cost. That implies that when searching for optimal repairs it is sufficient to consider one predicate from each equivalence class of predicates.

**LEMMA C.3 (EQUIVALENCE OF PREDICATES).** *Consider a space of predicates  $\mathcal{P}$ , rule  $r$ , and set of datapoints  $X_P$  with associated expected labels  $Z_P$  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.  $X_P$ , written as  $p \equiv_{X_P} p'$  if  $p(x) = p'(x)$  for all  $x \in X_P$ . 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:

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

**PROOF.** Let  $\Phi = \mathcal{A}(\mathcal{P}_{\equiv}, r, Z_P)$  and  $\Phi_{\equiv} = \mathcal{A}(\mathcal{P}, r, Z_P)$ . Based on the assumption about  $\mathcal{A}$ ,  $\Phi$  ( $\Phi_{\equiv}$ ) are optimal repairs within  $\mathcal{P}$  ( $\mathcal{P}_{\equiv}$ ). We prove the lemma by contradiction. Assume that  $\text{cost}(\Phi_{\equiv}) > \text{cost}(\Phi)$ . We will construct from  $\Phi$  a repair  $\Phi'$  with same cost as  $\Phi$  which only uses predicates from  $\mathcal{P}_{\equiv}$ . This repair then has cost  $\text{cost}(\Phi') = \text{cost}(\Phi) < \text{cost}(\Phi_{\equiv})$  contradicting the fact that  $\Phi_{\equiv}$  is optimal among repairs from  $\mathcal{P}_{\equiv}$ .  $\Phi'$  is constructed by replacing each predicate  $p \in \mathcal{P}$  used in the repair with an equivalent predicate from  $\mathcal{P}_{\equiv}$ . Note that such a predicate has to exist based on the requirement in Eq. (1). As equivalent predicates produce the same result on every  $x \in X_P$ ,  $\Phi'$  is indeed a repair. Furthermore, substituting predicates does not change the cost of the repair and, thus,  $\text{cost}(\Phi') = \text{cost}(\Phi)$ .  $\square$

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  $Z_P$  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  $x \in X$  contains one representative of each equivalence class.<sup>1</sup>

## D Path Refinement Repairs - Proofs and Additional Details

### D.1 GreedyPathRepair

Function GreedyPathRepair is shown Algorithm 4. To ensure that all datapoints ending in path  $P$  get assigned the desired label based on  $Z_P$ , 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 Proposition 4.3: 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  $P$ . For that, we maintain a queue

<sup>1</sup>With the exception of the class of predicates that return false on all  $x \in X_P$ . However, this class of predicates will never be part of an optimal repair as it does only trivially partitions  $P$  into two sets  $P$  and  $\emptyset$ .

**Algorithm 5:** BruteForcePathRepair

---

**Input** :Rule  $r$   
 Path  $P$   
 Datapoints to fix  $\mathcal{X}_P$   
 Expected labels for datapoints  $\mathcal{Z}_P$

**Output**: Repair sequence  $\Phi$  which fixes  $r$  wrt.  $\mathcal{Z}_P$

```

1  $todo \leftarrow [(r, \emptyset)]$ 
2  $\mathcal{P}_{all} = \text{GetAllCandPredicates}(P, \mathcal{X}_P, \mathcal{Z}_P)$ 
3 while  $todo \neq \emptyset$  do
4    $(r_{cur}, \Phi_{cur}) \leftarrow \text{pop}(todo)$ 
5   foreach  $P_{cur} \in \text{leafpaths}(r_{cur}, P)$  do
6     foreach  $p \in \mathcal{P}_{all} - \mathcal{P}_{r_{cur}}$  do
7       foreach  $y_1 \in \mathcal{Y} \wedge y_1 \neq \text{last}(P_{cur})$  do
8          $\phi_{new} \leftarrow \text{refine}(r_{cur}, P_{cur}, y_1, p, \text{true})$ 
9          $r_{new} \leftarrow \phi_{new}(r_{cur})$ 
10         $\Phi_{new} \leftarrow \Phi_{cur}, \phi_{cur}$ 
11        if  $\text{Acc}(r_{new}, \mathcal{Z}_P) = 1$  then
12          return  $\Phi_{new}$ 
13        else
14           $todo.\text{push}((r_{new}, \Phi_{new}))$ 
```

---

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  $P$ . 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 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 leaf node's label to  $\mathcal{Z}_P(x)$ .

**Generating Predicates.** The implementation of `GetCoveringPred` is specific to the type of RBBM. 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 Section 4.0.2, 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 processed in the current iteration into two subsets which each are strictly smaller than . 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 Proof of Theorem 4.4

**PROOF.** Proof of Theorem 4.4 In the following let  $n = |\mathcal{X}_P|$ .

**GreedyPathRepair:** As GreedyPathRepair does implement the algorithm from the proof of Proposition 4.3, 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 size. 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 less predicates which has 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 precessed, strictly decrease in each step. Thus, the algorithm will in worst-case terminate after adding  $n$  predicates.  $\square$

## E LLM as refinement setup

The prompt used in Section 5.3 is shown below

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  $F_{News}$  are

```
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
```

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

Part 2 of the code block

```
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
```

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

```
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 17: Refined *FNews* funcs by LLM (Part 3)

```
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
```

Figure 18: Refined *FNews* funcs by LLM (Part 4)

```
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 19: Refined *FNews* funcs by LLM (Part 5)

```

# 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

```

Figure 20: Refined Amazon funcs by LLM (Part 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 22: Refined Amazon funcs by LLM (Part 3)

```

# func3: No changes needed, it's quite simple but
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
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

```

Figure 21: Refined Amazon funcs by LLM (Part 2)

```

# 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

```

Figure 23: Refined Amazon funcs by LLM (Part 4)



```

# 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

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 24: Refined Amazon funcs by LLM (Part 5)

```

# 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

```

Figure 25: Refined Amazon funcs by LLM (Part 6)

```

# 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 26: Refined Amazon funcs by LLM (Part 7)