Explaining Mispredictions of Machine Learning Models using Rule Induction

Jürgen Cito
TU Wien and Facebook
Austria

Isil Dillig
UT Austin†
U.S.A.

Seohyun Kim
Facebook
U.S.A.

Vijayaraghavan Murali
Facebook
U.S.A.

Satish Chandra
Facebook
U.S.A.

ABSTRACT

While machine learning (ML) models play an increasingly prevalent role in many software engineering tasks, their prediction accuracy is often problematic. When these models do mispredict, it can be very difficult to isolate the cause. In this paper, we propose a technique that aims to facilitate the debugging process of trained statistical models. Given an ML model and a labeled data set, our method produces an interpretable characterization of the data on which the model performs particularly poorly. The output of our technique can be useful for understanding limitations of the training data or the model itself; it can also be useful for ensembling if there are multiple models with different strengths. We evaluate our approach through case studies and illustrate how it can be used to improve the accuracy of predictive models used for software engineering tasks within Facebook. We also compare our algorithm against related rule induction techniques to illustrate its advantages in the context of explaining mispredictions of machine learning models.

CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies** → **Rule learning**;

KEYWORDS

explainability, rule induction, machine learning

ACM Reference Format:
Jürgen Cito, Isil Dillig, Seohyun Kim, Vijayaraghavan Murali, and Satish Chandra. 2021. Explaining Mispredictions of Machine Learning Models using Rule Induction. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468614

† Work done at Facebook as visiting scientist.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468614

1 INTRODUCTION

Over the last decade, machine learning models have started playing an increasingly prevalent role in automating many types of software engineering tasks. For instance, machine learning has become a popular tool for automating code completion tasks [44], predicting which tests to run [28], finding relevant code fragments from large corpora [8], isolating crash inducing event sequences from telemetry [32], performing type inference [20, 34, 47], and synthesizing or repairing programs [5, 6, 13, 46].

While software engineering tools powered by machine learning achieve remarkably good results overall, it can be frustrating to understand when they do not produce the intended result. For instance, consider a predictive model that, given a crash report, predicts which of the recent code commits was the likely culprit. If such a tool does not produce the expected result, how do we understand why it behaves as it does? Is it because of some limitation of the model; or is it because of a lack of training data; or is it something else? Thus, in order to understand and improve modern software engineering tools (or, more generally, any program that uses machine learning), we need a way to *debug* trained statistical models. Unfortunately, manually debugging modern machine learning models can be an extremely daunting task due to the opaque nature of these models as well as the high dimensionality of the input data.

Motivated by this observation, there has been significant recent work on improving the interpretability of ML models. For example, *local interpretability* techniques like LIME [38], Shap [21], and Integrated Gradients [43] aim to provide accompanying evidence for predictions on a *specific* input. On the other hand, *global interpretability* techniques such as [18, 26, 45] try to shed light on the *global* behavior of a model either by highlighting which features are the most important or by constructing a simpler, surrogate model that emulates a more complex model. However, with the possible exception of [14], there is almost no work that can explain the *shortcomings* of machine learning models. That is, given a machine learning model, how can we identify the salient characteristics of the input data on which the model's predictions are particularly off? We believe that techniques for understanding the shortcomings of ML models are particularly important in the context of software engineering due to the growing popularity of complex ML models being used to support routine software engineering tasks like debugging, program repair, and more.

Based on this observation, the goal of this paper is to take a step towards semi-automatically debugging machine learning models

and shedding light on when a model does not work well. Given a labeled data set D and a trained model M, our technique automatically identifies *properties* of the data on which M performs particularly poorly. For instance, our technique might uncover that the model is much more likely to mispredict when the crash report pertains to a certain module or when it is excessively long, or indeed, a combination of such properties. As we demonstrate in this paper, this kind of information can be invaluable for improving the model or identifying problems in the training data. This information constitutes an *explanation* in the same sense as in *delta debugging* [49]: delta debugging isolates failure inducing part of the input, and analogously, our method finds properties of inputs on which a model mispredicts.

One of the appealing aspects of our proposed technique is that it is *model agnostic*. In other words, our method can be applied with equal ease to a complex deep learning model as it can be applied to a simple model like an SVM or a decision tree. Furthermore, our method produces *interpretable* results that are easy for a human to understand and act upon. From a technical perspective, the key idea underlying our technique is to learn simple rules (i.e., properties of the input data) that are *highly correlated* with mispredictions of the machine learning model. In particular, our method learns a set of rules that (a) collectively cover a large portion of the model's mispredictions, and (b) each of which correlate strongly with model mispredictions. The learnt rules are conjunctions of predicates over the input features and are therefore easily interpretable. Since these rules are intended to be read by a human, an attempt is made to keep rules simple. Our specific technical method is an instance of *rule induction*. The learnt rules explain the mispredictions of a model "globally", over a large number of inputs, as opposed to explaining misprediction "locally", for a specific input. Given the importance of this problem, there are other works that pursue similar model debugging strategies (e.g., [14]), but as explained in more detail in Sections 5.1 and 6, our primary objective is precision as opposed to accuracy, which has been shown to work well in our case studies.

We have implemented our method in a tool called MD and evaluate it in the context of software engineering tasks, such as code completion and predicting code crashes. In particular, we consider models used within Facebook and provide case studies illustrating how MD has been useful for improving these models. We also empirically evaluate the rules synthesized by our method against those inferred by other rule learners and show that our technique learns rules that correlate more strongly with mispredictions while still keeping rules simple and interpretable.

Key Contributions.

- We introduce the *model misprediction diagnosis* problem for improving the accuracy of machine learning models, adding to the emerging literature on model debugging.
- We present a model-agnostic technique for finding interpretable rules highly correlated with mispredictions.
- We apply our method to ML-powered software engineering tools and provide case studies to illustrate how our method has led to useful insights or improvements in these tools.
- We compare our method against two existing rule induction techniques and show that it yields rules that are better suited to the task of explaining mispredictions.

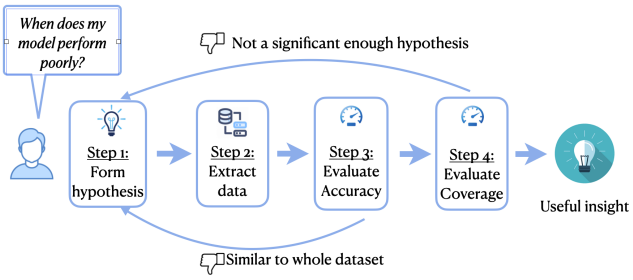

Figure 1: Manual ML model debugging process

Table 1: Samples from a dataset used to train a machine learning model that predicts whether a commit is likely to lead to a crash. The % of diffs that crashed are 8.9%, whereas the % of mispredictions ("crashed" XOR "pred") are 23.5%

loc	experience	modules	...	crashed	pred
3	medium	8		False	True
37	medium	4		False	False
6	low	6		False	False
35	low	6		True	True
38	high	7		False	True

2 OVERVIEW

Imagine you have trained a machine learning model for predicting whether a code commit will lead to a crash. The model might base its predictions on lines of code changed ("loc"), experience level of the developer ("experience"), number of modules touched in the process ("modules"), and potentially dozens of additional columns. Table 1 provides a small sample of the dataset, including columns that show the model's prediction ("pred") and the ground truth outcome ("crashed").

The accuracy of this model on the test set[1] is 76.5%. How would you go about improving the accuracy of this model? In most cases, a first step in this direction is to understand the types of input on which the model does not perform well. In other words, what are some salient characteristics of 23.5% of the test data for which the model mispredicts?

To answer this question, one would first need to formulate a hypothesis such as *the model performs poorly when the lines of code changed (loc) is rather small*. Next, to test this hypothesis, one needs to write a query to extract this subset of the data, evaluate the model on this subset, and compare accuracy α on this subset against that α' of the entire dataset. If α is not substantially lower than α', the hypothesis is incorrect and should be discarded. In this case, the user would need to formulate a different hypothesis and restart the process. On the other hand, if accuracy is indeed much lower on this subset of the data, one would likely want to investigate what percentage of the mispredictions the hypothesis explains. For example, if this hypothesis explains 90% of all mispredictions, it would be very useful to investigate this hypothesis in more detail.

[1]We use the term test data generically to mean the data set used for validation of the model.

Table 2: Different ways a user can act on misprediction explanations produced by MD

Scenario	Possible action	Case study
Training data not rich enough	Improve training data	Bug2Commit (Sec 5.2.1)
Enough training data for E but model still mispredicts	Improve model algorithm	Autocomplete (Sec 5.2.3)
Cannot improve training data or model	Suppress model's output if input belongs in E	Oncall prediction (Sec 5.3)
Multiple models with different misprediction explanations	Select from models based on input	Diff review (Sec 5.2.2)

On the other hand, if it only explains 1% of the mispredictions, one may want to find a better hypothesis. As illustrated by this discussion and depicted schematically in Figure 1, this process is extremely inefficient and requires significant human insight and manual labor. This is particularly true when the model is trained on datasets with a large number of features.

In the particular setting of Table 1, it turns out that a hypothesis that explains mispredictions particularly well is *experience!="low" & loc<=13 & modules>6*, which has a precision of 85.7% and a recall of 30.6%. This means that for this combination of properties in the data, the model mispredicts in 85.7% of cases and it explains 30.6% of all mispredictions that occur in the dataset.

The techniques proposed in this paper are intended to simplify this kind of manual effort involved in debugging machine learning models. In particular, our method largely automates Steps 1-4 depicted in Figure 1 and produces *tested hypotheses* that explain when the model performs poorly. Our method, dubbed *Misprediction Diagnoser* or *MD* for short, takes as input a data set D (labeled with ground truth predictions), machine learning model M, and a minimum target coverage parameter δ and automatically produces a misprediction explanation E such that (1) M's accuracy is much lower for inputs that conform to E compared to the whole data set, and (2) if we discard inputs that conform to E, then M's accuracy improves a significant amount (i.e., E explains at least a minimum threshold δ of all mispredictions). Hence, the explanations produced by our method pass the checks shown in Steps 3 and 4 of Figure 1 and obviate the need for manually constructing and testing these hypotheses.

We envision a number of ways in which the output of MD may be helpful to its users. In particular, as summarized in Table 2, the output of MD may be useful for improving the training data or the model itself, and it may also pinpoint opportunities where suppressing the model's output or ensembling different models may be beneficial. For instance, suppose that MD produces E as a misprediction explanation but the training data does not contain sufficiently many samples that conform to E. This would suggest a distribution shift between the training and test data, which in turn calls for improving the training data or for performing data augmentation. If this is not the case, it is likely due to a shortcoming of the model itself, which can be improved by understanding why the model mispredicts on the subset of the data conforming to E. If there is no room for improving either the model or the data, the output of MD can still be beneficial. For example, if we know where a code completion engine is likely to mispredict, we can choose not to make any predictions rather than confusing developers with incorrect completions. Similarly, if we have access to two models M_1 and M_2 with different misprediction explanations E_1, E_2, we can use this information to obtain a useful ensemble. We have selected concrete case studies from software engineering to illustrate each of these scenarios (Section 5.2).

3 PROBLEM FORMULATION

In this paper, we focus on machine learning models used for classification. A *misprediction explanation* E for such a model M is a boolean function such that inputs x satisfying E are likely to be misclassified by M with high probability. In the remainder of this section, we formalize the desired properties of E in terms of an optimization problem (Section 3.1) and then describe our hypothesis space (Section 3.2).

3.1 Optimization Objective

Let $D : \mathcal{X} \rightarrow \mathcal{Y}$ be a labeled dataset , and let $M : \mathcal{X} \rightarrow \mathcal{Y}$ be a (trained) statistical model. For $x \in \mathcal{X}$, we define the following indicator function $\mathbb{I} : \mathcal{X} \rightarrow \{0, 1\}$:

$$\mathbb{I}(x) = \begin{cases} 1 & \text{if } D(x) \neq M(x) \\ 0 & \text{otherwise} \end{cases}$$

In other words, $\mathbb{I}(x)$ is 1 iff the machine learning model produces the wrong label for x.

Our goal is to find a boolean function $E^* : \mathcal{X} \rightarrow \{0, 1\}$ that maximizes the following optimization objective:

$$E^* = \underset{E}{\text{argmax}}\ P(\mathbb{I}(x) = 1 \mid E(x) = 1, x \sim \mathcal{X}) \tag{1}$$

In other words, the misprediction explanation E^* should maximize the probability that a given input is misclassified by the model.

In addition, because explanations that only explain a tiny fraction of the mispredictions are not very useful, we additionally require that E^* should explain some minimum fraction δ of the mispredictions:

$$P(E^*(x) = 1 \mid \mathbb{I}(x) = 1, x \sim \mathcal{X}) \geq \delta \tag{2}$$

We refer to the value $P(E(x) = 1 \mid \mathbb{I}(x) = 1, x \sim \mathcal{X})$ as the *coverage* of explanation E.

Thus, putting these together, given a model M and parameter δ, our problem in this paper is to find a boolean function E that solves the following constrained optimization problem for inputs x drawn from distribution $\mathcal{X}$:

$$\begin{aligned} &\text{maximize } P(\mathbb{I}(x) = 1 \mid E(x) = 1) \\ &\text{subject to :} \\ &\qquad P(E(x) \mid \mathbb{I}(x) = 1) \geq \delta \end{aligned}$$

3.2 Hypothesis Space

In principle, *any* boolean function $\mathcal{X} \rightarrow \{0, 1\}$, including a deep neural network, can serve as a misprediction explanation. However,

because our goal is to help humans diagnose problems in their classifiers, it is very important that such explanations be easily interpretable. Thus, in this work, we restrict our hypothesis space to *decision lists* (also called *ordered rule sets*) defined by the following grammar:

$$\begin{array}{lcl} E & \rightarrow & \text{if}(\phi) \text{ then 1 else } E \mid 0 \\ \phi & \rightarrow & x_c = c \mid x_c \neq c \mid x_n \leq c \mid x_n > c \mid \phi \wedge \phi \end{array}$$

In other words, misprediction explanations we consider in this work are simple `if - else if` "programs" where each conditional is a conjunction of atomic predicates of the form x op c where x is a feature and c is a constant. We use the notation x_c, x_n to indicate categorical and numeric/ordinal features respectively. For categorical features, we only allow the equality and disequality operators, whereas for numerical/ordinal features, we allow the $\leq$ and $>$ operators. In the remainder of this paper, we use the term *rule* to refer to a conjunction of atomic predicates.

We believe that decision lists are well-suited to our problem setting because they essentially correspond to an ordered sequence of (non-overlapping) rules that explain different problem cases for the model. Thus, an end-user can inspect each rule in order, perform one of the possible actions listed in Table 2 to address the corresponding problem, and move on to the next one.

Discussion. The reader may wonder if our hypothesis space is too restrictive in that atomic predicates refer to individual attributes (features) of the input data. However, it is worth noting that these attributes need *not* necessarily correspond to features of the input data that the model was trained on. For instance, consider a classifier that that is trained on a data set that has features like *mass* and *volume*, and the user wants to know whether there is some correlation between the model's mispredictions and density. In that case, the user can augment the data set to include such a *density* feature. Taking this one step further, it might even be the case that the data set that the model is trained on is entirely different from the data set used for computing misprediction explanations. For instance, consider an image classification model where attributes in the original data set are individual pixels. Since such features may be too low-level to be useful for debugging, the user may choose to replace the original data set D with another data set D' where features in D' correspond to outputs of simpler ML models (e.g., for detecting basic shapes like triangles).

4 GENERATING MISPREDICTION EXPLANATIONS

The problem formulation that we presented in Section 3 can be addressed by a *rule induction* technique. In this section, we present a rule induction technique that can be viewed as an instance of *subgroup discovery* [2]. In Section 5.1 we compare against a couple of other approaches to rule induction and give a broader overview Section 6.

At a high level, our approach produces decision lists that solve the constrained optimization problem from Section 3.1. However, since exactly solving this optimization problem is computationally intractable for large datasets, our approach approximates this objective using a greedy approach. Note that our approach differs from other techniques for learning decision lists in that it optimizes a different objective function.

1: **procedure** Explain(D, M, δ)
   input: Labeled data set $D : \mathcal{X} \rightarrow \mathcal{Y}$ (ground truth)
   input: ML model $M : \mathcal{X} \rightarrow \mathcal{Y}$
   input: Target coverage δ
   output: Misprediction explanation for M
2: $\mathcal{L} \leftarrow$ LabelData(D, M)
3: $\mathcal{A} \leftarrow$ GenAtoms($\mathcal{L}$)
4: $rs \leftarrow []$ ▷ A list of rules representing decision list
5: $cvg \leftarrow 0$ ▷ Current coverage for rs
6: $cur \leftarrow \mathcal{L}$
7: **while** $cvg \leq \delta$ **do**
8: $\phi \leftarrow$ LearnRule($cur, \mathcal{A}$)
9: $rs \leftarrow rs :: [\phi]$
10: $cur \leftarrow$ Filter(cur, ϕ)
11: $cvg \leftarrow$ ComputeCoverage($\mathcal{L}, rs$)
12: **return** rs

Algorithm 1: Top-level algorithm

4.1 Top-Level Algorithm

Algorithm 1 presents our top-level procedure called Explain for generating misprediction explanations. This procedure takes as input a labeled data set D containing ground truth labels, an ML model M, and a target coverage δ. We now explain how this procedure generates a misprediction explanation.

Construct new data set. The Explain algorithm starts by calling a procedure called LabelData that constructs a new dataset $\mathcal{L} : \mathcal{X} \rightarrow \{0, 1\}$ such that:

$$\mathcal{L}(x) = 1 \Leftrightarrow (D(x) \neq M(x))$$

In other words, the new data set $\mathcal{L}$ maps each input in D to a boolean value indicating whether on not it is mispredicted by the given model M.

Generate atomic predicates. Next, our algorithm calls a procedure named GenAtoms to generate a universe of candidate atomic predicates of the form x_i *op* c where x_i is a feature and c is a constant. If x_i is a categorical variable, we generate all predicates of the form $x_i = c_j$ and $x_i \neq c_j$ where $c_j \in \text{Domain}(x_i)$. For numerical and ordinal features, we use the operators $\leq, >$ and generate the constants c_j using *equal frequency binning* [22]. In particular, let x_i be a numerical feature that takes on values $C = \{c_1, \ldots, c_n\}$ in the data set. To generate atoms of the form x_i *op* c_j, we first partition the (sorted) set C into k bins[2] where each bin has roughly equal size and then use the highest value in each bin as one of the constants in our predicates.

Main learning loop. After the initialization phase (lines 2–6), the algorithm enters a loop that iteratively adds new rules to the decision list (lines 6–10) until the learned decision list achieves the desired coverage parameter δ. The algorithm represents the

[2]The value of k is a hyper-parameter and is set to 4 by default.

learned decision list rs as a list of predicates, so, for example, the list $[\phi_1, \phi_2]$ corresponds to the following misprediction explanation:

$$\texttt{if}(\phi_1)\ \texttt{then}\ 1\ \texttt{else}\ \texttt{if}(\phi_2)\ \texttt{then}\ 1\ \texttt{else}\ 0$$

At a high level, the learning loop synthesizes the target decision list using a standard *sequential covering* approach [10]. In particular, it first learns a rule ϕ_1 for the whole data set, then filters out elements satisfying ϕ_1, then learns another rule ϕ_2 for the remaining elements, and so on, until the target coverage is reached. Thus, the predicate ϕ_i learned during the i'th iteration corresponds to the i'th branch in the final misprediction explanation. Intuitively, the predicate in the i'th branch is the best predictor for mispredictions in the subset of the data not covered by the earlier predicates.

In more detail, each iteration of the loop considers a subset of the data called *cur* (initialized to $\mathcal{L}$ in the beginning) and learns a new rule ϕ that serves as a (conjunctive) misprediction explanation for *cur*. This is done at line 8 via the call to LearnRule (discussed in detail in the next subsection) and added as a new branch of the decision list (line 9). Next, it removes from *cur* all inputs $\mathcal{X}' \subseteq \mathcal{X}$ satisfying predicate ϕ and computes coverage *cvg* for the misprediction explanation rs as follows (line 11):

$$\frac{\left|\{x \in \mathcal{X} \mid rs(x) = 1 \wedge \mathcal{L}(x) = 1\}\right|}{\left|\{x \in \mathcal{X} \mid \mathcal{L}(x) = 1\}\right|} \tag{3}$$

which is exactly the probability from Eqn 2. Since the algorithm terminates only when *cvg* exceeds δ, the output of the Explain procedure is guaranteed to satisfy the coverage constraint from Eqn 2.

4.2 Rule Learning

We now describe the LearnRule procedure for learning a conjunction of predicates over $\mathcal{A}$ for a given data set $\mathcal{L}$.[3] As stated in Eq. 1, our goal is to learn a rule ϕ such that:

$$P_{x \sim \mathcal{X}}(\mathcal{L}(x) = 1 \mid \phi(x))$$

is maximized — i.e., we want to learn rules that are *highly* correlated with mispredictions. Observe that this optimization problem is equivalent to maximizing the following alternative optimization objective:

$$p = \frac{|\{x \in \mathcal{X} \mid \phi(x) \wedge \mathcal{L}(x) = 1\}|}{|\{x \in \mathcal{X} \mid \phi(x)\}|} \tag{4}$$

which corresponds to *precision* or *positive predictive value*.

However, if our rule learning algorithm aims to maximize *solely* p, the top-level procedure from Algorithm 1 may, in practice, take many iterations to converge. In particular, recall that Algorithm 1 aims to find a set of rules that collectively satisfy the coverage threshold. But if each individual rule (i.e., branch) has very low coverage, we may need to learn *many* rules in order to reach the coverage threshold. Beyond making the algorithm slow to converge, this would also result in misprediction explanations that are very large, thereby compromising interpretability. Thus, rather than optimizing only precision, our rule learning algorithm also takes into account recall as well as rule size. In particular, our optimization

[3]Recall that $\mathcal{L}$ indicates whether a data point is mispredicted or not.

1: **procedure** LearnRule($\mathcal{L}, \mathcal{A}$)
input: Labeled data set $\mathcal{L}$ where a label $l \in \{0, 1\}$ indicates misprediction or not
input: A set of atomic predicates
output: Rule ϕ (conjunct over atomic predicates)
2: $B \leftarrow [true, \ldots, true]$ ▷ beam initialization
3: $done \leftarrow false$
4: **while** $\neg done$ **do**
5: $done \leftarrow true$
6: **for all** $(\phi_i, p_i) \in B \times \mathcal{A}$ **do**
7: $\phi \leftarrow \phi_i \wedge p_i$; $\sigma \leftarrow \mathsf{Eval}(\phi, \mathcal{L})$
8: $(\phi', \sigma') \leftarrow \mathsf{GetWorst}(B)$
9: **if** $\sigma' < \sigma$ **then**
10: $B \leftarrow (B \backslash \{\phi'\}) \cup \phi$; $done \leftarrow false$
11: **return** $\mathsf{GetBest}(B)$

Algorithm 2: Rule learning algorithm based on beam search

objective O is a linear combination of precision, recall, and rule size:

$$O = \lambda_1 \cdot p + \lambda_2 \cdot r + \lambda_3 \cdot \frac{1}{size(\phi)} \tag{5}$$

where r is the *recall* (or *coverage*) defined as in Eqn. 3 and $\lambda_1, \lambda_2, \lambda_3$ are tunable hyper-parameters. Since our primary goal is to find rules that are highly correlated with mispredictions, precision is the primary factor and is therefore given a higher weight by default; the other hyperparameters are mainly used for regularization and accelerating convergence.[4]

Our rule learning algorithm maximizes this optimization objective using standard beam search as shown in Algorithm 2. In particular, given a beam size of n, Algorithm 2 initializes all n rules in the beam to *true* (line 2). Then, it enters a while loop that terminates when we fail to improve the objective value of any of the rules in the beam. In particular, in each iteration of the while loop, we construct new rules by adding a single atomic predicate p_i to one of the rules ϕ_i in the beam. If the resulting rule ϕ yields a better value σ of the objective function from Eqn. 5 compared to the worst rule ϕ' in the beam (line 9), then we replace ϕ' with the new rule ϕ. At the end of the loop, the algorithm returns the best rule (i.e., one with the highest objective value) among all the rules in the beam.

Running example. We illustrate the algorithm on the example model on relating diffs to crashes we introduced in Sec 2. Table 3 shows the universe of atomic predicates generated based on the input feature space illustrated in Table 1.

In every step of our beam search in Algorithm 2, we build up new rules that consist of conjunctions of predicates from our atomic predicate universe (e.g., *modules>6 & loc<=13 & experience=="medium"*). We evaluate new rule candidates and replace them with the worst performing rule present in our current beam. In our main learning loop in Algorithm 1, we iteratively add the best rule in our beam to the decision list until it achieves our the

[4]We have also experimented with other optimization objectives such as F1 score but we found that it does not achieve the right balance between different goals in our context.

Table 3: Generated universe of atomic predicates based on the dataset in Table 1

Atomic Predicates		
experience=="low"	modules<=3	loc<=13
experience!="low"	modules>3	loc>13
experience=="medium"	modules>6	loc<=26
experience!="medium"	modules>9	loc<=39
experience=="high"	modules<=6	loc>39
experience!="high"	modules<=9	loc>26

desired coverage parameter (e.g., **if** *(modules>6 & loc<=13 & experience=="medium")* **elseif** *(exp!="low" & loc<=13)*). Table 4 shows the final results produced by MD.

4.3 Implementation

We have implemented MD as a Python library that takes as input a Pandas dataframe, a target variable, target coverage, and a set of (optional) parameters and returns a set of decision lists paired with precision, recall, and coverage metrics. The main parameters to be provided are the number of bins (default is 4) and the beam width (default is 10).

Despite taking a greedy approach to solving our optimization objective, the proposed method can still be quite slow on large data sets, especially where the number of features is very large. Thus, our method uses several optimizations to make this computation tractable. Most of the "heavy" computation in our implementation involves producing subsets of the data by applying filters and comparing metrics of the subset with metrics of the overall data (and those of other subsets). We were able to achieve these significant efficiency improvements by introducing proper indexing (and eventually using Pandas built-in indexing utility). Further, we have a set of hyper-parameters that dictate when we discard predicates and rules that do not meet certain precision and relevance thresholds. An implementation is available as an open source project on GitHub: https://github.com/facebookresearch/mmd.

5 EVALUATION

In this section, we describe a series of evaluations that are designed to answer the following research questions:

- **RQ1:** How does the rule learning algorithm of MD compare with other techniques with respect to finding conditions for mispredictions?
- **RQ2:** Can MD identify problems in ML models used in software engineering tasks?
- **RQ3:** Is MD useful for improving these models?

To answer RQ2 and RQ3, we deployed MD within Facebook and report on case studies with selected teams using proprietary data and models. For experiments comparing MD to existing rule learners (RQ1), we additionally use public datasets and models from Kaggle.

5.1 Comparison with Other Rule Learners

Generally speaking, rule induction techniques differ along three dimensions: (1) shape of the learnt rules, (2) the quality metric used for evaluating the rules (i.e., optimization objective), and (3) the search technique. These characteristics have implications of how well-suited the results of the different rule induction techniques are for any particular task.

In this section, we compare our proposed algorithm with other rule learning techniques. In particular, we compare against contrast set mining (CSM) and a decision-list learner implementing RIPPER. CSM has been previously used to diagnose problems in software engineering tasks [9, 35]. Ripper is a well-known ruleset learning algorithm that has been used in a variety of classification tasks [16, 17, 24].

We briefly explain where the STUCCO, a CSM algorithm, and RIPPER algorithms lie along the axes of rule shape, quality metric and search algorithm.

STUCCO Algorithm. The STUCCO algorithm aims to discover rules that maximally contrast the target categories. In the remainder of this discussion, we only consider the scenario where there are two target categories.

Shape of the learnt rules. In contrast to our technique that learns decision lists, STUCCO only learns conjunctions of predicates (i.e., rules instead of rule sets).

Quality metric. The STUCCO algorithm evaluates the quality of a rule based on two metrics: *largeness* and *significance*. Largeness means that the difference of *support* for each of the two target groups must be larger than a certain threshold, where support is the percent of rows in a group that match a rule. The other consideration is significance, which measures whether the difference in support is statistically significant, and not just random fluctuation, based on a chi-squared test. Unlike a direct emphasis on precision and recall within the misprediction group, this algorithm optimizes (as the name says) for contrast between two groups.

Search algorithm. Similar to our LearnRule procedure, CSM's STUCCO algorithm is also based on beam search.

RIPPER Algorithm. RIPPER is rule learning algorithm tailored primarily for classification (we focus on binary classification).

Shape of the learnt rules. Similar to our approach, RIPPER learns decision lists rather than conjunctions of predicates.

Quality metric. The quality metric used by the RIPPER algorithm is information-theoretic gain [37]. In particular, it favors rules that provide the maximum information gain.

Search algorithm. Similar to our top-level algorithm, RIPPER learns decision lists using a sequential covering approach. However, its procedure for learning rules is *not* based on beam search and involves three steps, namely (1) grow, (2) prune, and (3) optimize. The grow step adds conjuncts to a rule based on the information gain criterion. The second step prunes rules that do not reduce entropy. These grow and prune steps are repeated until a stopping criterion is reached. Finally, the optimize step attempts to improve the learnt rules using a variety of heuristics.

5.1.1 Evaluation Subjects. As evaluation subjects, we use two models from our case studies: Bug2Commit (Sec 5.2.1) and Diff Review (Sec 5.2.2). We also consider two publicly available ML models from

Table 4: Overview of final decision lists generated for dataset introduced in Section 2 (Table 1)

Learned Rulesets	Precision	Recall
if modules>6 & loc<=13 & experience=="medium"	0.906977	0.165957
if experience!="low" & loc<=13 & modules>6	0.857143	0.306383
if (modules>6 & loc<=13 & experience=="medium") **elseif** (experience!="low" & loc<=13)	0.734884	0.672340
if (experience!="low" & modules>6) **elseif** (experience!="low" & loc<=13)	0.588732	0.889362

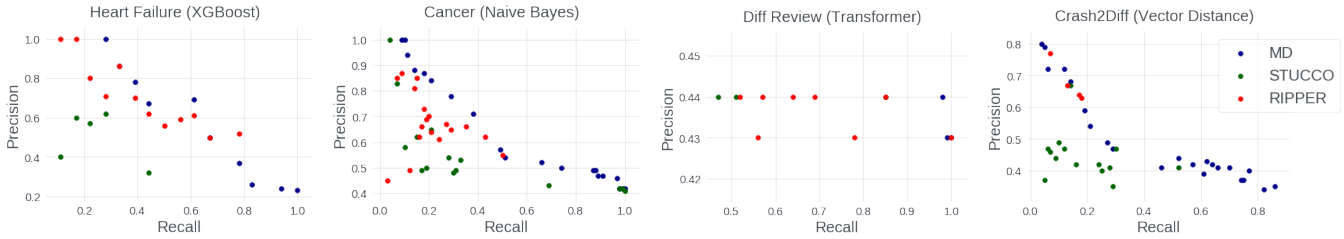

Figure 2: Precision-recall curves comparing different rule learners on four different tasks. For Diff Review, we see few data points for MD because it finds rules with high recall even if we specify the coverage parameter to be small.

Kaggle, with the goal of both providing a replicable study (as it is not possible for us to share company internal data), as well as to evaluate whether our quantitative results extend to external models. For Kaggle, we want to select models that still have room for improvement, so we use models that have an accuracy of less than 90%. Based on these criteria, we evaluate MD on the the following pairs of data sets and models: (1) Heart Failure dataset with an XGBoost model [29] (85% Accuracy), and (2) Cervical Cancer Risk Classification dataset with a Naive Bayes model [11] (59.8% Accuracy).

5.1.2 Results. Our main results are presented in Figure 2, which shows precision-recall curves for the different misprediction explanation tasks. In particular, remember that MD takes as input a parameter that denotes target recall (i.e., percentage of mispredictions covered by the explanation). To generate these precision-recall curves, we run MD with different coverage thresholds. However, since the other techniques do not have such a parameter, we simply show all rules produced by STUCCO and RIPPER (after filtering out redundant rules). Also, as mentioned earlier, STUCCO only generates conjunctive rules.

As we can see from Figure 2, the explanations produced by MD outperform STUCCO and RIPPER in most cases. That is, for similar recall rates, the explanations from STUCCO and RIPPER often have lower precision compared to MD. In some cases, this is not true for RIPPER. However, in these cases, the explanations produced by RIPPER are significantly more complex than those of MD. Table 5 provides an illustrative overview of the rule complexity produced by each of the techniques that we have observed throughout our study: RIPPER finds rules with high precision and recall by sacrificing interpretability, producing rules with very high complexity (i.e., number of conditions). While both STUCCO and MD produce quite succinct rules, we find that MD finds a better balance of precision and recall on the pareto frontier.

While there is no good way of illustrating rule complexity in the plots, we separately report mean and standard deviation of total rule size for each dataset and technique combination in Table 6. We see that MD generally produces rules with lower complexity, especially compared to RIPPER.

Table 5: Representative rule from each of the techniques

	Rule	Prec.	Rec.
MD	**if** exp!="low" & mod>6 & loc<=18 **elseif** exp!="low" & loc<=9	0.84	0.71
CSM	**if** exp=="m" & loc>=1.0 & loc<=10.0	0.92	0.34
Ripper	**if** exp=="high" & loc>=1 & loc<=4 **elseif** loc>=4 & loc<=8 & exp=="m" **elseif** exp=="high" & loc>=4 & loc<=8 **elseif** loc>=1 & loc<=4 & exp=="m" **elseif** loc>=8 & loc<=13 & exp=="m" & mod=="8" **elseif** loc>=1 & loc<=4 & mod=="7"	0.97	0.58

Table 6: Overview of total rule size mean and standard deviation across techniques and datasets

Method	Dataset	Mean	Std
Ripper	Cervical Cancer	9.32	1.90
MD		3.61	1.85
CSM		2.94	2.18
Ripper	Heart Failure	6.51	2.35
MD		3.84	1.81
CSM		5.95	1.73
Ripper	Bug2Commit	8.70	2.11
MD		4.75	3.57
CSM		3.12	1.32
Ripper	Diff Review	6.71	2.62
MD		3.22	1.63
CSM		2.67	1.15

5.2 Case Studies

In this section, we present case reports applying MD to four ML models used within Facebook. We discuss the insights MD revealed and how developers were able to improve their tools by acting on these insights.

One of the common themes in applying MD to software engineering models was the need to augment the original data set with additional, purely *diagnostic* features. That is, in addition to the input data, we worked with model developers to come up with additional features that *could* provide useful diagnostic values. We illustrate this aspect of our case study in the subsections that follow.

5.2.1 Bug2Commit. Bug2Commit [31] is a lightweight ML model developed at Facebook. Given a crash report in the form of text (including stack frames when available), and a large set of code commits in the form of meta data from the commit (but possibly some content as well), Bug2Commit aims to answer the following question: Which commit is most likely the cause of the crash? We expect Bug2Commit to find the true "blame" commit among the top 10 in a ranked list of suspect commits. The model is based on information retrieval and computes the cosine distance between vector representations of both the crash report and each of the commits. Bug2Commit is known to miss the true blame commit in more than 30% of the cases.

We set out to find out whether there is a characterization of crashes that are particularly prone to misprediction. We identified several attributes that could be of interest: length of the crash report, the repository to which it pertains (iOS, Android, etc.), the number of files modified in the true blame commit (which are known for this dataset) and so on.

MD found the following rule (among the top 5) with high precision: *repository=="ios" & length_trace>57*. That is, if the commit is for iOS and the length of the crash trace is greater than 57 lines, it leads to misprediction much more often than in the overall dataset. This rule's precision is 0.68 and recall is 0.14. Our algorithm discovered this rule without requiring any human insight.

To gain better intuition about how to improve the model, we used our domain knowledge to augment the dataset with an additional, diagnostic feature, namely the overlap of words between the crash report and the blame commit, as a percentage of top words in the crash report. In particular, since the model is based on vector distance between the "top weighted" words (BM25), we conjectured that overlap could be a useful diagnostic feature. With this additional feature, MD found: *repository=="ios" & length_trace > 57 & overlap<=35*, with precision of 0.72 and recall of 0.12. This now gives an actionable idea: can we improve word overlap?

Based on the insights we obtained using MD, we augmented the content for each commit with the names of the modified functions in the changed files. With this augmentation and retraining the pipeline, we got significant reduction in iOS misses, without deteriorating anything else. On a dataset with about 5000 instances, we had 32% mispredictions (17% from iOS). With training over augmented data, we had 28.5% mispredictions (13% from iOS.) The data indicates that there is more scope to improve overlap between words, which we will explore going forward.

5.2.2 Diff Review. The "Diff Review" project builds ML models to assess the quality of review that a diff (code commit) has gone through. A diff contains various features associated with it, such as (i) a title and summary of the commit, (ii) files that are modified, and (iii) the actual code changes. Different models were built based on each of these three features, in addition to a fourth model that combines these features using a deep neural network. The deep model performs the best overall in terms of precision but it is also the slowest in terms of inference time. In addition, there are some inputs that are predicted correctly by one of the three simpler models but mispredicted by the deep neural network model. In this case study, we use MD to better understand these "blind spots" for the deep model and construct an ensemble with both better precision and faster inference.

To use MD, we again came up with additional diagnostic features that could be useful. In particular, we used as attributes the number of modified files of each type in each diff. For instance, if a diff modified 4 Java files and 2 Python files, it would have attributes *modified_java=4* and *modified_python=2*. We then ran MD three times, once for each of the simpler models, with a target where the simpler model predicted correctly and the deep model did not. MD revealed several predicates that indicated that the deep model mispredicted when a significantly large number of files of particular types are modified (e.g., *modified_javascript>31*). We confirmed with the designer of the deep model that, when a large number of files are modified in a diff, it ignores the code changes (due to memory constraints) and switches to only using the other features.

We proceeded to see if this is something that can be fixed "online" without re-training models. Using the predicates surfaced by MD, we implemented a quick model selection routine that switches to using the simpler models when the predicates surfaced by MD were satisfied. This eliminated 80 mispredictions compared to the original model. As an additional benefit, we were able to improve inference time by over 3%.

5.2.3 Autocomplete. We used MD to generate misprediction explanations for a machine learning model used for code auto-completion. This is a classical sequence prediction setting, where a standard RNN learns a "language model" over a token stream, considering a context window of size up to 1000. The model outputs a probability distribution (via softmax) over the top-N tokens and is known to predict the correct next token among the top-5 results in approximately 52% of the cases.

To apply MD for this purpose, we first came up with diagnostic features that could be potentially useful for improving the model. These features involve the *vocabulary* of the corpus as well as properties of the token predicted by the model. Some of our diagnostic features include: (1) frequency rank of the target token in the vocabulary, (2) whether the top prediction is out of vocabulary (OOV, indicated by special "unk" token), (3) whether the receiver of attribute access (the token that comes before the `.`) is out of vocabulary, (4) the probability delta between the top prediction and the next one, (5) whether the target occurs in the recent context window, and (6) number of tokens in the context.

MD uncovered that the feature that is most highly correlated with misprediction to be whether the top prediction is out of vocabulary (precision of 84.9% and recall rate of 59.9%). This means that when

the top prediction is "unk", the correct token is not in the top-5 results in 84.9% of cases. Some of the other highly-correlated features included (1) whether the probability delta between the top prediction and next one was < 0.5 (precision 68.6%), and (2) if the receiver is OOV (precision 65%).

This information was useful in two ways. First, we were able to use the misprediction explanations of MD to decide when to suppress the output. That is, if the model's predictions exhibit the characteristics uncovered by MD, we chose not to make predictions rather than frustrating users with incorrect predictions (e.g. suppressing predictions on incidence of the above delta condition would reduce misprediction rate from 48% to 40%.) In addition, the output of MD was also useful for inspiring the developers of the auto-completion tool to improve their model. In particular, given the findings of MD with respect to out-of-vocabulary tokens, they are currently exploring the use of pointer networks and copy mechanisms to improve precision [4].

5.3 Oncall Recommendation

In this last case study, we consider a machine learning model for assigning points of contact for various tasks in the software engineering life-cycle [1] (e.g., debugging, code review, oncall rotation). We used MD with the goal of improving the Rank-1 accuracy of a particular model that ranks a set of oncall candidates on files in the source code repository. The test set for the model is composed of ground truth data on previous oncall assignments.

MD uncovered the following rule which has high precision (93.6%) and recall (39.4%):

$$legacy\_system\_score \leq 0.14 \wedge inline\_comment\_ratio \leq 0.46 \wedge decision\_members\_ratio \leq 0.80$$

This rule includes a feature that relies on a score retrieved by a legacy subsystem that was previously responsible for suggesting oncall assignments. We investigated retraining the model without using the legacy feature. While we were able to see improvements in model accuracy, they were relatively small (~1% improvement). However, the rules uncovered by MD (paired with domain knowledge) helped the team uncover sources of noise introduced by one of the features on the relationship of Rank-1 accuracy and ground truth. Specifically, there can be multiple contenders for a Rank-1 prediction (if more than one person has the same probability score assigned by the model). When investigating the impact of the *legacy_system_score* feature, they uncovered that this feature added just enough noise to rank someone as Rank-2, even though they should have also been at Rank-1. Upon closer inspection, the team learned that this noisiness was only relevant for some parts of the involved repositories. A possible solution that was discussed was suppressing predictions for this particular part of the input space.

6 RELATED WORK

The work presented in the paper relates to several topics.

Rule learning techniques. The misprediction explanation finding technique presented in this paper can be viewed as an instance of rule-based methods used in machine learning and data mining. In particular, since our method produces decision lists, it is particularly related to techniques for learning decision lists [40] and, more recently, *decision sets* [25]. However, the goal of these prior techniques is to perform accurate classification rather than characterizing mispredictions of ML models. Due to these differing goals, our method learns rules that optimize a different objective that prioritizes *precision* rather than *accuracy* Due to this difference, our approach learns decision lists that are highly correlated with mispredictions rather than learning rules that strike a good balance between explaining both correct and incorrect predictions. The difference between our primary objective and the one in classification is shown schematically in Figure 3. Here, the green box serves as a better explanation than the yellow box in our case since all data points inside it are mispredicted (indicated by +), so it has higher precision. On the other hand, the yellow box has higher accuracy and would therefore be preferable for classification purposes.

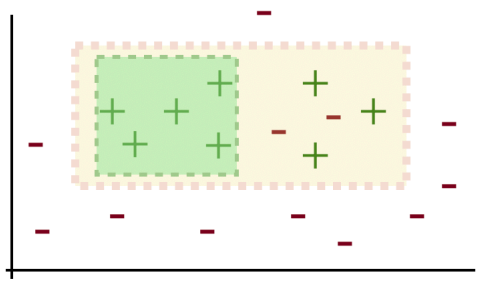

Figure 3: Illustration of different objectives in our case vs classification. Since our primary objective is precision, the green box serves as a better misprediction explanation than the yellow box for our purposes. On the other hand, the yellow box has higher accuracy[5] compared to the green box (90% vs. 85%) and is therefore better for classification. However, it has lower precision (80% vs. 100%).

More generally, rule learning techniques can be classified as either *descriptive rule discovery* which aims to find patterns in data or *predictive rule discovery* which is used for making predictions for new data; ID3 [37], CN2 [15] and Ripper are in the second class. Since our goal is to discover which types of inputs are misclassified by a model, our method is much more closely related to descriptive rule discovery, which can be further categorized into three main classes: (1) contrast set mining (CSM), (2) emerging pattern mining (EPM), and (3) subgroup discovery (SD). While these techniques are closely related, they differ in the following ways: CSM aims to find statistically meaningful differences between multiple groups [7]; EPM aims to find new patterns that emerge in new versions of the same dataset [33], and SD aims to find statistically interesting subgroups with respect to some property of interest [3]. Since our goal is to find subgroups in the data for which misprediction ratio is particularly high, our approach can be cast as an instance of subgroup discovery, which has traditionally been useful in identifying sub-populations that are at risk for certain medical conditions [19, 23, 30]. While many subgroup discovery algorithms have been proposed, depending on the application domain, they vary with respect to their optimization objectives, shape of the

[5] *accuracy* is defined as (true-positives + true-negatives) / all.

learnt subgroups, and search strategies. In contrast to existing subgroup discovery algorithms, our method learns decision lists using a sequential covering approach because we want to identify a *set* of subgroups that collectively cover a high percentage of the mispredictions.

Among different rule learning techniques, ours is perhaps most similar to that of Lakkaraju et al. [25] in that their objective function is also a linear combination of different desiderata that aim to strike a good balance between the primary optimization objective and other factors such as conciseness and interpretability. However, as mentioned earlier and illustrated in Figure 3, their primary objective is accuracy whereas ours is precision. Furthermore, we choose decision lists rather than decision sets due to the additive nature of the discovered rules in terms of recall.

Rule Learning in Software Engineering. Rule learning techniques have also been applied in the context of software engineering. Contrast set mining has been used to help debug crashes by identifying properties unique to sets of crash reports [9, 36]. Castelluccio et al. use STUCCO to find statistically significant correlations in crash groups at Mozilla [9]. Qian et al. use CSM to learn what distinguishes groups of crashes at Facebook. They extend the algorithm to be directly applicable to continuous variables instead of using discretization that can lead to scalability issues [36]. Different rule learning techniques have also been used in defect prediction [41, 42]. Rodriguez et al. study the use of subgroup discovery algorithms to obtain rules identifying defect prone modules [41]. Song et al. uses descriptive rules obtained from association rule learning to predict defect associations and correction efforts [42].

ML model interpretability. While our primary goal is to help users *debug* their machine learning models, our approach is nonetheless related to the fast growing field of *model interpretability*. Efforts in this space can be classified as focusing on either *local* or *global* interpretability. Techniques for local interpretability, such as LIME [38], ANCHORS [39] and integrated gradients [43], aim to provide evidence to justify a *specific* prediction made by the model. In contrast, techniques for global interpretability aim to shed light on the overall behavior of a model. For example, GALE [45] aims to compute globally important features whereas other methods such as [18, 26] construct a simpler and more interpretable *surrogate* model for a much more complex model. More relevant to the software engineering community, [12] automatically extracts rules that emulate deep learning models and applies this technique to software engineering tasks such as binary analysis and malware detection.

Debugging ML models. More similarly to this paper, there has also been recent interest in developing techniques to *debug* machine learning models. For example, Wu et al. [48] propose a methodology for debugging ML models used in natural language processing (NLP). Specifically, they propose a domain-specific language for formulating and testing hypotheses about NLP models; however, it is up to the users to manually formulate these hypotheses in the proposed DSL and decide whether they constitute good misprediction explanation. In contrast, the technique proposed here is intended to automate the task of finding misprediction explanations to a large extent.

Another related approach is *data slicing*, where the goal is to find a "slice" (i.e., subset) of the data where the difference in error loss is statistically significant and the so-called "effect size" is above a certain threshold [14]. Key applications of data slicing include evaluating model fairness and fraud detection. While the goal of this work is similar to ours, our method differs from theirs in that (a) we only assume access to the classification result (as opposed to error loss), (b) we optimize a different objective function consisting of precision and recall that are easier to evaluate, (c) the rules we produce are additive in terms of coverage (as opposed to the top-k rules), and (d) we use different techniques for optimizing our objective function.

Recently, there has also been been a proposal for automatically repairing neural models [27]. In particular, they propose a technique (MODE) for identifying so-called *faulty neurons* and then use this information to select inputs that have a high presence of features that are important for misclassification. In contrast to MODE where the goal is to automatically generate additional training data, our goal is to come up with an interpretable explanation of when a model mispredicts. As discussed earlier, this information can be used for data augmentation, but it may also be useful for other purposes such as ensembling, output suppression, or improving model architecture. In addition, our method is model agnostic and does not focus solely on neural models.

7 LIMITATIONS

While we have shown that MD can help explaining mispredictions of models, it does have certain limitations in its ability to do so. Firstly, MD searches the space of attributes for likely explanations, but the attributes themselves have to be defined by the model designer. Typically, one can start with metadata that already exists for the problem from the input space (e.g., "length of trace" for Bug2Commit), and if the generated explanations are not actionable, add further diagnostic features (e.g., "overlap percentage"). In some cases, this process might require human intervention to converge to the right set of actionable attributes. Secondly, MD does not guarantee that its explanations will lead to significant improvements for the underlying model. Its main purpose is to help the model designer understand dark corners of the model and take appropriate action (as in Table 2). MD should not be relied on as a means to produce an improved model, which is only a possible side-effect of the action. Third, we designed MD to be model-agnostic so that it can be applied to different types of ML models. While this design choice allows broader applicability, it also prevents our approach from taking advantage of white-box knowledge that could potentially allow better scalability.

8 CONCLUSION

We have proposed *misprediction explanations* as a useful concept for debugging and improving machine learning models. We also presented a model-agnostic technique for generating useful and interpretable misprediction explanations. We demonstrated through case studies that misprediction explanations are useful for improving ML models used in machine learning, and we also demonstrated the advantages of our technique compared to other rule learning techniques such as RIPPER and STUCCO.

REFERENCES

[1] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Shan He, Ralf Lämmel, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2020. Ownership at Large: Open Problems and Challenges in Ownership Management. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 406–410. https://doi.org/10.1145/3387904.3389293

[2] Martin Atzmueller. 2015. Subgroup discovery. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 5, 1 (2015), 35–49.

[3] Martin Atzmueller. 2015. Subgroup discovery. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 5, 1 (2015), 35–49.

[4] G. Aye, S. Kim, and H. Li. 2021. Learning Autocompletion from Real-World Datasets. , 131-139 pages. https://doi.org/10.1109/icse-seip52600.2021.00022

[5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[6] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 168:1–168:27. https://doi.org/10.1145/3360594

[7] Stephen D Bay and Michael J Pazzani. 1999. Detecting change in categorical data: Mining contrast sets. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. 302–306.

[8] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974. https://doi.org/10.1145/3338906.3340458

[9] Marco Castelluccio, Carlo Sansone, Luisa Verdoliva, and Giovanni Poggi. 2017. Automatically analyzing groups of crashes for finding correlations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 717–726. https://doi.org/10.1145/3106237.3106306

[10] Jadzia Cendrowska. 1987. PRISM: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies* 27, 4 (1987), 349–370.

[11] Vanessa Chantreau. 2021. Cervical Cancer Risk Classification. https://www.kaggle.com/ebobette/cancer-full-study?scriptVersionId=20118243.

[12] Simin Chen, Soroush Bateni, Sampath Grandhi, Xiaodi Li, Cong Liu, and Wei Yang. 2020. *DENAS: Automated Rule Generation by Knowledge Extraction from Neural Networks*. Association for Computing Machinery, New York, NY, USA, 813–825. https://doi.org/10.1145/3368089.3409733

[13] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *International Conference on Computer Aided Verification*. Springer, 587–610. https://doi.org/10.1007/978-3-030-53291-8_30

[14] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. 2020. Automated Data Slicing for Model Validation: A Big Data - AI Integration Approach. *IEEE Transactions on Knowledge and Data Engineering* 32, 12 (2020), 2284–2296. https://doi.org/10.1109/TKDE.2019.2916074

[15] P. Clark and T Niblett. 1989. The CN2 induction algorithm. *Machine Learning* (1989), 261–283.

[16] William W Cohen et al. 1996. Learning rules that classify e-mail. In *AAAI spring symposium on machine learning in information access*, Vol. 18. Stanford, CA, 25.

[17] William W Cohen and Yoram Singer. 1999. Context-sensitive learning methods for text categorization. *ACM Transactions on Information Systems (TOIS)* 17, 2 (1999), 141–173.

[18] Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint* (2017).

[19] Dragan Gamberger, Nada Lavrač, and Goran Krstačić. 2003. Active subgroup mining: a case study in coronary heart disease risk group detection. *Artificial Intelligence in Medicine* 28, 1 (2003), 27–57.

[20] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 152–162. https://doi.org/10.1145/3236024.3236051

[21] Ehud Kalai and Dov Samet. 1987. On weighted Shapley values. *International journal of game theory* 16, 3 (1987), 205–222.

[22] Sotiris Kotsiantis and Dimitris Kanellopoulos. 2006. Discretization techniques: A recent survey. *GESTS International Transactions on Computer Science and Engineering* 32, 1 (2006), 47–58.

[23] Petra Kralj, Nada Lavrac, Blaz Zupan, and Dragan Gamberger. 2005. Experimental comparison of three subgroup discovery algorithms: Analysing brain ischemia data. *Information Society* (2005), 220–223.

[24] Milan Kumari and Sunila Godara. 2011. Comparative study of data mining classification methods in cardiovascular disease prediction 1. (2011).

[25] Himabindu Lakkaraju, Stephen H. Bach, and Jure Leskovec. 2016. Interpretable Decision Sets: A Joint Framework for Description and Prediction. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1675–1684. https://doi.org/10.1145/2939672.2939874

[26] Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Jure Leskovec. 2017. Interpretable & explorable approximations of black box models. *arXiv preprint arXiv:1707.01154* (2017).

[27] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 175–186. https://doi.org/10.1145/3236024.3236082

[28] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*. IEEE Press, Piscataway, NJ, USA, 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[29] Shiva Raj Mishra. 2021. ANN vs Rest: Endgame in Heart Failure Prediction. https://www.kaggle.com/shivarajmishra/ann-vs-rest-endgame-for-heart-failure-prediction?scriptVersionId=54717793.

[30] Marianne Mueller, Rómer Rosales, Harald Steck, Sriram Krishnan, Bharat Rao, and Stefan Kramer. 2009. Subgroup discovery for test selection: a novel approach and its application to breast cancer diagnosis. In *International Symposium on Intelligent Data Analysis*. Springer, 119–130. https://doi.org/10.1007/978-3-642-03915-7_11

[31] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. 2021. Industry-scale IR-based Bug Localization: A Perspective from Facebook. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '21)*. https://doi.org/10.1109/ICSE-SEIP52600.2021.00028

[32] Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. 2021. Scalable Statistical Root Cause Analysis on App Telemetry. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '21)*. https://doi.org/10.1109/ICSE-SEIP52600.2021.00038

[33] Petra Kralj Novak, Nada Lavrač, and Geoffrey I Webb. 2009. Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining. *Journal of Machine Learning Research* 10, 2 (2009).

[34] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220. https://doi.org/10.1145/3368089.3409715

[35] Rebecca Qian, Yang Yu, Wonhee Park, Vijayaraghavan Murali, Stephen Fink, and Satish Chandra. 2020. Debugging Crashes Using Continuous Contrast Set Mining. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 61–70. https://doi.org/10.1145/3377813.3381369

[36] Rebecca Qian, Yang Yu, Wonhee Park, Vijayaraghavan Murali, Stephen Fink, and Satish Chandra. 2020. Debugging crashes using continuous contrast set mining. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 61–70. https://doi.org/10.1145/3377813.3381369

[37] J Ross Quinlan. 1987. Decision trees as probabilistic classifiers. In *Proceedings of the Fourth International Workshop on Machine Learning*. Elsevier, 31–37.

[38] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144. https://doi.org/10.1145/2939672.2939778

[39] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-Precision Model-Agnostic Explanations. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*.

[40] Ronald L Rivest. 1987. Learning decision lists. *Machine learning* 2, 3 (1987), 229–246.

[41] Daniel Rodriguez, Roberto Ruiz, Jose C Riquelme, and Rachel Harrison. 2013. A study of subgroup discovery approaches for defect prediction. *Information and Software Technology* 55, 10 (2013), 1810–1822. https://doi.org/10.1016/j.infsof.2013.05.002

[42] Qinbao Song, Martin Shepperd, Michelle Cartwright, and Carolyn Mair. 2006. Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering* 32, 2 (2006). https://doi.org/10.1109/tse.2006.1599417

[43] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*. PMLR, 3319–3328.

[44] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1433–1443. https://doi.org/10.1145/3368089.3417058

[45] Ilse van der Linden, Hinda Haned, and Evangelos Kanoulas. 2019. Global aggregations of local explanations for black box models. *arXiv preprint arXiv:1907.03039* (2019).

[46] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. *arXiv preprint* (2017).

[47] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161* (2020).

[48] Tongshuang Wu, Marco Tulio Ribeiro, Jeffrey Heer, and Daniel Weld. 2019. Errudite: Scalable, Reproducible, and Testable Error Analysis. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 747–763. https://doi.org/10.18653/v1/P19-1073

[49] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498