A Formal Explainer for Just-In-Time Defect Predictions

JINQIANG YU, Monash University, Australia and Australian Research Council OPTIMA ITTC, Australia
MICHAEL FU, Monash University, Australia
ALEXEY IGNATIEV, Monash University, Australia
CHAKKRIT TANTITHAMTHAVORN, Monash University, Australia
PETER J. STUCKEY, Monash University, Australia and Australian Research Council OPTIMA ITTC, Australia

Just-In-Time (JIT) defect prediction has been proposed to help teams to prioritize the limited resources on the most risky commits (or pull requests), yet it remains largely a black-box, whose predictions are not explainable nor actionable to practitioners. Thus, prior studies have applied various model-agnostic techniques to explain the predictions of JIT models. Yet, explanations generated from existing model-agnostic techniques are still not formally sound, robust, and actionable. In this paper, we propose FoX, a Formal eXplainer for JIT Defect Prediction, which builds on formal reasoning about the behaviour of JIT defect prediction models and hence is able to provide provably correct explanations, which are additionally guaranteed to be minimal. Our experimental results show that FoX is able to efficiently generate provably-correct, robust, and actionable explanations while existing model-agnostic techniques cannot. Our survey study with 54 software practitioners provides valuable insights into the usefulness and trustworthiness of our FoX approach. 86% of participants agreed that our approach is useful, while 74% of participants found it trustworthy. Thus, this paper serves as an important stepping stone towards trustable explanations for JIT models to help domain experts and practitioners better understand why a commit is predicted as defective and what to do to mitigate the risk.

ACM Reference Format:
Jinqiang Yu, Michael Fu, Alexey Ignatiev, Chakkrit Tantithamthavorn, and Peter J. Stuckey. 2024. A Formal Explainer for Just-In-Time Defect Predictions. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2024), 31 pages. https://doi.org/10.1145/3664809

1 INTRODUCTION

Modern software companies often continuously release software products at a rapid pace in short-term cycles. Such rapid-release software development often poses the greatest challenges to under-resourced Software Quality Assurance (SQA) teams due to the exponential growth of highly-complex source code. Since various SQA activities are time-consuming and expensive (e.g., code review), developers cannot exhaustively ensure that all newly developed code commits or pull requests are of the highest quality given the limited time and resources.

Just-in-time (JIT) defect prediction [26, 29, 35, 43] has been proposed to predict if a commit will introduce software defects in the future, enabling teams to prioritize their limited SQA resources on the most risky commits/pull requests. In the past decades, various Artificial Intelligence/Machine

Authors' addresses: Jinqiang Yu, jinqiang.yu@monash.edu, Monash University, Australia and Australian Research Council OPTIMA ITTC, Australia; Michael Fu, yeh.fu@monash.edu, Monash University, Australia; Alexey Ignatiev, alexey.ignatiev@monash.edu, Monash University, Australia; Chakkrit Tantithamthavorn, chakkrit@monash.edu, Monash University, Australia; Peter J. Stuckey, peter.stuckey@monash.edu, Monash University, Australia and Australian Research Council OPTIMA ITTC, Australia.

1049-331X/2024/1-ART1 $15.00
https://doi.org/10.1145/3664809

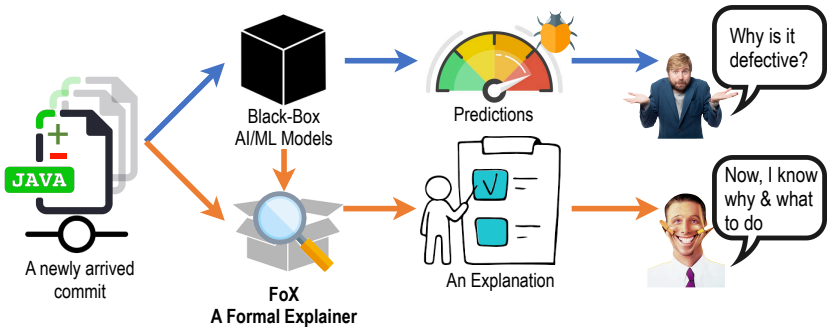

Fig. 1. A motivation on the need of Explainable AI for Just-In-Time defect prediction. Currently, practitioners and domain experts often ask many *why?* questions (e.g., why a commit is predicted as defective) and *how?* questions (e.g., how should developers mitigate the risk) [25].

Learning (AI/ML) techniques have been used to build JIT defect prediction models. Such defect prediction technologies have empowered many recent AI-powered code quality tools (e.g., Microsoft's AI Code Defect,[1] Amazon's CodeGuru,[2] CQSE GmbH's TeamScale,[3] Sealight's Code Quality Intelligence,[4] and CodeScene's Code Quality Analytics[5]), demonstrating the significance of Just-in-time (JIT) defect prediction to real-world practice.

Recently, the *explainability* and *actionability* of AI/ML models in SE have become one of the most challenging research problems that remain largely unsolved. Importantly, most of the current JIT defect prediction models are often treated as a black-box. Their predictions are not explainable nor actionable to practitioners (see Figure 1)—i.e., they are not able to provide clear evidence to explain why the given instance is predicted as defective or not defective. Particularly, practitioners and domain experts often asked many *why?* questions (e.g., why a commit is predicted as defective) [12, 24, 25, 58] and *how?* questions (e.g., how should developers mitigate the risk) [7, 32, 34, 42, 44, 45, 63]. In addition, due to the growing size and complexity of modern AI/ML-based JIT defect models, it is also very difficult for domain experts to understand how the models work and whether the models are trained correctly. Thus, a lack of explainability and actionability can lead to a lack of trust in JIT defect prediction models, hindering their adoption in practice [12, 24, 25, 57].

The state-of-the-art in the explainability of JIT defect prediction models is represented by local model-agnostic approaches. One successful local model-agnostic approach referred to as PyExplainer has been recently proposed in an award-winning paper at ASE'21 [44]. PyExplainer's explanations are greatly beneficial to help developers better understand what features contributed the most to the predictions, enabling teams to focus on the most important aspects that are associated with software defects instead of focusing on the less important ones. The key principle of model-agnostic techniques (these also include the prominent explainers LIME [47], SHAP [36] and Anchors [48] among many others) is to generate explanations based on extensive sampling in the vicinity of the concrete instance being explained. The sampling procedure randomly generates a large number of synthetic instances and results in either directly reporting a conjunction of

[1]https://www.microsoft.com/en-us/ai/ai-lab-code-defect
[2]https://aws.amazon.com/codeguru/
[3]https://www.cqse.eu/en/teamscale/overview/
[4]https://www.sealights.io/product/release-quality-analysis/
[5]https://codescene.com/

feature values deemed relevant for the prediction [48], or training a surrogate local model on these synthetic instances [36, 47], which aims at approximating the original model.

Unfortunately, the reliance on extensive sampling although a vital component in making these explainers model-agnostic also makes them in general *incorrect* [18, 19, 41] and in particular susceptible to *out-of-distribution attacks* [33, 52]. Finally, the random nature of the sampling procedure results in the model-agnostic explanations being *non-robust* [2, 24, 51, 53], i.e. running the same explanation method on the same instance multiple times may produce different explanations. These issues may negatively impact the operational decision-makings of under-resourced SQA teams and exacerbate the problem of trust in AI.

Motivated by the aforementioned limitations of the sampling-based methods, *in this paper*, we propose FoX, a Formal eXplainer for Just-In-Time Defect Prediction, which builds on *formal reasoning* about the behavior of JIT defect prediction models and hence is able to provide a user with *provably correct* explanations, which are additionally guaranteed to be minimal. FoX leverages the formal explainability principles that have not been explored in software engineering [21, 22, 37, 50] in order to compute a single *abductive* explanation, i.e. answering *why a model predicted (or not) a commit as defect-introducing*, and a single *contrastive* explanation (aka. counterfactual explanations), i.e. saying *what developers should do to reverse the prediction of the model*. Furthermore, FoX can provide a user with a given number of both abductive and contrastive explanations, or enumerate them exhaustively [20] as well as report *formal feature attribution* [65], i.e. a list of numeric values representing *how important* the corresponding features are for a given prediction. Note that FoX hinges on the use of formal methods applied to the original JIT defect prediction models, and so on the success of modern propositional satisfiability (SAT) solving [3]. In particular, given a model and an instance both encoded into propositional logic, FoX makes a series of SAT oracle calls for computing the requested number of abductive and/or contrastive explanations for the prediction made. Thanks to the use of logic, the approach is able to capture the behavior of the original model and hence the explanations it reports are guaranteed to be formally correct, i.e. they hold in the *entire feature space*.

In light of the theoretical correctness guarantees of formal explanations [38], we experimentally evaluate FoX along two additional dimensions: robustness and speed, and compare with four state-of-the-art model-agnostic techniques i.e., LIME [47], SHAP [36], Anchor [48], PyExplainer [44]. By evaluating FoX and the competitors on two JIT defect prediction models (i.e., Logistic Regression and Random Forest) that are trained on a total of 40,798 commits from two large-scale software systems (i.e., OpenStack and Qt), the results show that (1) model-agnostic explanations cannot compete with formal explanations generated by FoX in terms of formal correctness; (2) in stark contrast to heuristic explanations, formal explanations of FoX are 100% robust (i.e., deterministic); and (3) explanations of FoX can be efficiently computed and enumerated, i.e. explanation generation for both LRs and RFs requires less than a second. Thus, we conclude that FoX is able to efficiently generate provably-correct, robust, and actionable explanations, addressing various limitations raised by prior studies of explainable defect prediction [2, 24, 51, 53]. These explanations are expected to help domain experts and practitioners better understand why a commit is predicted as defective through abductive explanations and what to do to mitigate the risk of having defects in the future thanks to contrastive explanations. To explore the practicality of our FoX approach, we conduct a user study with 54 software practitioners. Specifically, we evaluated the usefulness and trustworthiness of our approach. Our survey indicates that the explanation generated by FoX for JIT defect predictions improves the usefulness from 72% to 86% and improves the trustworthiness from 53% to 74% based on practitioners' perceptions. These results highlight the practicality of our FoX approach that can enhance the trust between JIT defect prediction and software practitioners.

Novelty. To the best of our knowledge, this paper is the first to:

- Present FoX – The first Formal eXplainer for Just-In-Time Defect Prediction that leverages the formal explainable AI principles [13, 20–22, 37, 50], providing provably correct and succinct explanations that can answer not only *why?* and *how?* questions but also giving formally defined feature importance scores [65].
- Empirically demonstrate that FoX is able to efficiently generate robust and actionable explanations while the four state-of-the-art model-agnostic techniques that were previously used in defect prediction (i.e., LIME [47], SHAP [36], Anchor [48], PyExplainer [44]) cannot, besides lacking formal correctness.

Open Science. To facilitate the reusability and replicability of our work, we provide a public replication package[6]. To ease the adoption of FoX by practitioners, we publish FoX as a Python Package[7], which is available in both `conda` and `pip` (Package Installer for Python). The FoX Python package is also well-tested, achieving a code coverage of 98% measured by CodeCov with A+ quality and 0 alerts graded by LGTM.

Paper Organization. Section 2 motivates this work and discusses the limitations of prior approaches. Section 3 defines the necessary concepts and argues how formal explanations address the aforementioned limitations. Section 4 describes the proposed approach. Section 5 details the experimental setup while Section 6 presents the results. Section 8 outlines related work. Section 9 discusses the limitations and future work, while Section 10 concludes the paper.

2 BACKGROUND & MOTIVATION

In this section, we motivate and formulate the problem with respect to the findings of prior work based on a few illustrative examples.

The explainability of AI models is one of the grand research challenges [56] for AI in SE (see http://xai4se.github.io), since practitioners often do not trust the predictions [12, 57, 58, 58, 63], hindering the adoption of AI-powered software development tools in practice. Recently, Explainable AI has been actively investigated in the domain of defect prediction [7, 24, 25, 32, 42, 44, 45, 58]. For example, recent works have shown some successful case studies to make defect prediction models more practical [43, 61], explainable [24, 28], and actionable [44, 45].

The heart of Explainable AI for SE is the use of model-agnostic techniques to explain the predictions of AI/ML models. Examples of widely-used model-agnostic techniques in SE include LIME [47], SHAP [36], Anchors [48], and PyExplainer [44]. Since many AI/ML models are treated as a black-box, the primary objective of model-agnostic techniques is to build a local explainable model that mimics the behaviors of the global black-box model for the prediction of an instance to be explained.

Such model-agnostic techniques often consists of four steps: (1) generate synthetic instances around an instance to be explained; (2) obtain the predictions of such synthetic instances using the global model; (3) build a local explainable model to learn the relationship between synthetic instances and their predictions from the global model; and (4) generate an explanation from the local explainable model. Depending on the form of explanations model-agnostic approaches offer, they are conventionally classified as *feature selection* or *feature attribution* approaches briefly discussed below. Both lines of work aim at identifying the *most important features*, which are a set of features (i.e., independent variables) that have the strongest influence on the model prediction for a given

[6]https://github.com/trustablefox/exp_replication
[7]https://github.com/trustablefox/foxplainer

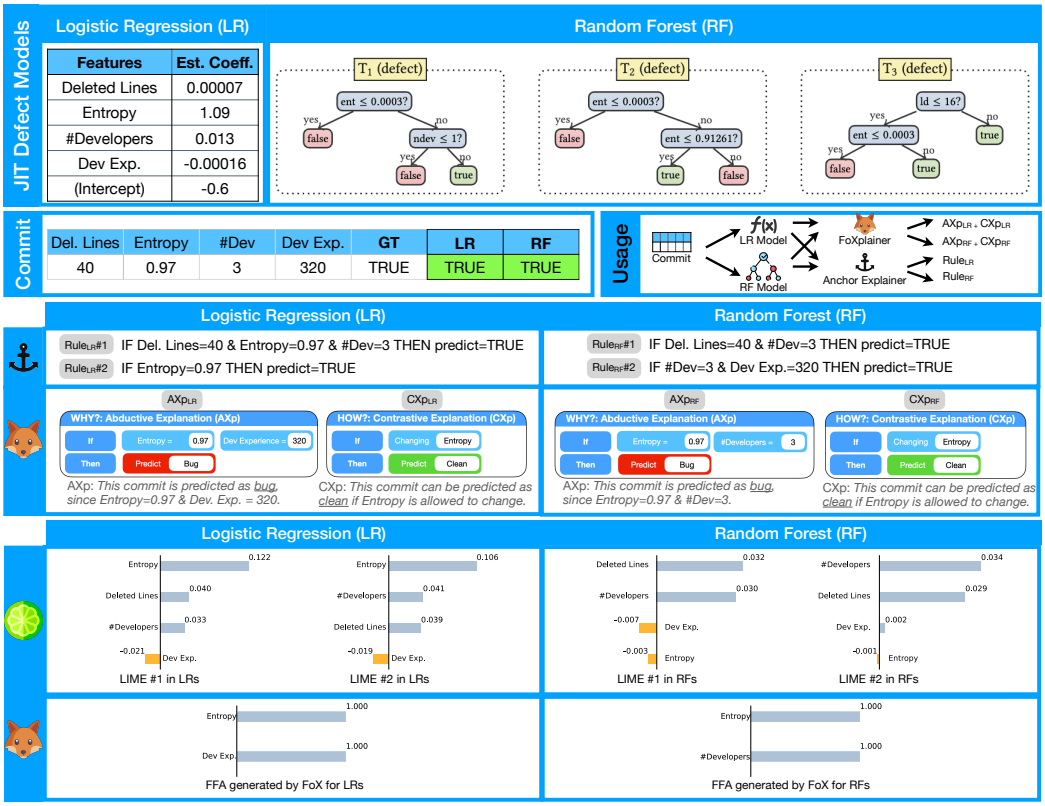

Fig. 2. An illustrative example of explanations generated by FoX, Anchor [48], and LIME [47] for Just-In-Time defect prediction. FFA refers to Formal Feature Attribution (see Definition 3).

instance (i.e., the strongest relationship between features and a prediction outcome), meaning that features that are not important must have little to no influence on the model prediction. Note that both forms of model-agnostic techniques often suffer from a few key limitations also detailed below.

Feature Selection. A feature selection[8] approach identifies subsets of features that are deemed *responsible* for a given prediction. (Two well-known examples of feature selection approaches are Anchor [48] and PyExplainer [44].) The sufficiency of the selected set of features for a given prediction is determined by these explainers statistically based on extensive sampling around the instance of interest, by assessing a few measures like *fidelity*, *precision*, among others.

Feature Attribution. A different view on post-hoc explanations is provided by feature attribution approaches, e.g. LIME [47] and SHAP [36]. Based on random sampling in the neighborhood of the target instance, these approaches attribute responsibility to all model's features by assigning a numeric value of importance to each feature. Given these importance values, the features can then be ranked from most important to least important. As a result, given a set of features $\mathcal{F}$, a feature attribution explanation is conventionally provided as a linear form $\sum_{i \in \mathcal{F}} w_i \cdot x_i$, which can be also

[8]Hereinafter, the term *feature selection* denotes the principles underlying a family of ML explanation approaches rather than the techniques *of the same name* used in the context of data mining.

seen as approximating the original black-box model in the *local* neighborhood of instance being explained. Among other feature attribution approaches, SHAP [36] is often claimed to stand out as it aims at approximating Shapley values, a powerful concept originally introduced by L. Shapley in the context of cooperative games in game theory [49] as a mechanism for assessing player importance in cooperative games.

Limitation ①: Explanations are not formally correct. Explanations generated by existing model-agnostic techniques heavily rely on extensive sampling and various interpretable ML models (e.g., linear regression, LASSO, decision tree) to be used to explain the behavior of the original black-box model. However, due to their statistical nature such model-agnostic techniques do not have a mechanism to provably guarantee that explanations are correct with respect to the original model. For instance, Ignatiev [19] has recently argued that in the context of general AI classification tasks, most prominent model-agnostic explainers report explanations that do not logically capture the semantics of the original models and are likely to be incorrect. Similarly, Lakkaraju *et al.* [33, 52] showed how out-of-distribution sampling can be used to fool a model-agnostic explainer. Finally, Huang and Marques-Silva [18] argued that SHAP may often fail to give a correct estimation of the actual feature importance thus failing to produce reasonable explanations, both in terms of attribution values and in terms of feature ranking. As a result, the set of most important features that such explainers claim to be the reason for the prediction may in fact be insufficient for the prediction (i.e. some of the other important features are missing), or it may be too conservative (i.e. it contains features that are irrelevant for the prediction).

An Illustrative Running Example. Consider example logistic regression (LR) and random forest (RF) based JIT defect prediction models shown in Figure 2. These models are trained using 4 selected features (i.e., *ld*, *ent*, *ndev* and *dev_exp*) on the training dataset of the Qt project.[9] Given an example commit **v** (#lines_deleted = 40, entropy = 0.97, #developers = 3, #developer_experience = 320), the commit is predicted as *true* by both JIT defect models, meaning that this commit is likely to introduce defects in the future based on the given characteristics.

Example 1 (Incorrect Explanation)**.** *First, let us consider a model-agnostic feature selection explanation generated by Anchor [48] for the LR prediction generated as "IF* ld = 40 & ent = 0.97 & ndev = 3 *THEN prediction is* true*", meaning that the LR model shown in Figure 2 predicts this commit as* true *due to these three features. This indicates that these three features alone (i.e.,* ld, ent, *and* ndev*) are* sufficient *for the given commit prediction made by the model. In other words, other features* can be excluded *from consideration as having no effect on the prediction. However, this explanation is* incorrect *because there exists at least one* counterexample *instance compatible with the explanation on all the selected features that is still predicted differently. For example, by changing the value 320 of* dev_exp *to 3,400, we get the JIT model prediction* false*. This illustrates that feature* ld*, which is claimed by Anchor to be irrelevant, still affects the model's prediction for the instance being explained. Observe how the counterexample constructed reveals incorrectness of the explanation.*

Now, consider a feature attribution explanation that LIME [47] computes for the RF prediction: {ld: 0.032, ndev: 0.030, dev_exp: -0.007, ent: -0.003}*. This explanation indicates that all features contribute to the prediction although two of them exhibit negative contributions. Observe that the feature* dev_exp *is not included in the example RF model and thus it should not have any contribution to the prediction. However, LIME fails to generate a correct explanation since it assigns a non-zero attribution value to the feature* dev_exp *in the explanation.* □

Limitation ②: Explanations are not robust. Explanations generated by existing model-agnostic techniques heavily rely on synthetic instances that are randomly generated around

[9]For simplicity, all but 4 features are discarded here. Note that the experimental results shown in Section 6 address the *original* datasets, with no simplification involved.

an instance to be explained. Similarly, prior studies [2, 24, 51, 53] also raised concerns that such model-agnostic techniques are often non-deterministic. Thus, the randomness within the neighbourhood generation process may produce different sets of synthetic instances, which often leads to producing different explanations.

Example 2 (Non-robust Explanations)**.** *Given our running example commit (#lines_deleted = 40, entropy = 0.97, #developers = 3, #developer_experience = 320) and the RF model, Figure 2 shows that if Anchor [48] runs from scratch twice, two different explanations are computed, i.e. "IF* ld = 40 & ndev = 3 *THEN prediction is* true*" and "IF* ndev = 3 & dev_exp = 320 *THEN prediction is* true*", which exemplifies the issue of non-robustness. A similar issue can be observed in LIME [47]. As depicted in Figure 2, LIME generates different feature attribution explanations when running from scratch twice, i.e.* {ld: 0.032, ndev: 0.030, dev_exp: -0.007, ent: -0.003} *and* {ld: 0.029, ndev: 0.034, dev_exp: 0.002, ent: -0.001}*. Importantly, these explanations not only offer different feature attributions but also result in different feature rankings based on those attributions.* □

Limitation ③: Explanations are not actionable. Explanations generated by existing model-agnostic techniques for JIT defect predictions only focus on answering *why?* questions (e.g., why a commit is predicted as defective) [12, 24, 25, 58]. However, the explanations for the *how?* questions (e.g., how should developers proceed to mitigate the risk) that are desirable by practitioners [7, 32, 34, 42, 44, 45, 63] have not yet been effectively generated. Observe that Anchor's explanations in the examples above, while being incorrect and non-robust, may help a user understand *why* the corresponding prediction was made by the model – but they provide the user with no clue of *how* the prediction could be changed.

3 PRELIMINARIES

Given the significant limitations yet high impact of prior work [44, 47, 48], we propose FoX, a practical, formal reasoning-based approach capable of generating both provably correct *abductive* and *contrastive explanations* for JIT defect models. Below, we present the usage scenario of the proposed approach followed by its technical details.

3.1 Usage Scenario

In software development, the ML-based defect prediction bot could offer significant value in improving code quality. In particular, our approach can be integrated seamlessly with GitHub commit actions, which functions as a proactive tool to identify potentially defective commits in real time.

Upon a developer pushing a commit to a repository on GitHub, FoX is automatically triggered to analyze whether this commit is defective using ML models. Subsequently, the prediction results are displayed directly within the GitHub interface (as shown in Figure 10), offering insightful explanations for defective commits. Specifically in this example, FoX offers model explanations including factors such as low relative reviewer experience, low reviewer awareness, and high commit age. These explanations help clarify why the ML model generates such a prediction. Furthermore, actionable guidance is offered to support developers mitigate identified risks. In particular, developers are advised to assign more experienced developers for review, enhance reviewer awareness through training and communication, and ensure frequent updates for ongoing scrutiny and validation.

Upon receiving the prediction results, explanations, and mitigation strategies, developers can acknowledge the suggestions and take necessary actions accordingly. In conclusion, by integrating the ML-based explainable defect prediction into the software development workflow, development

teams can proactively identify and address potential defects, thereby enhancing the overall quality of their software products.

3.2 Necessary Notation

First, let us formally define a JIT defect prediction model. A JIT defect prediction model is defined under the standard classification scenario characterized by a set of features $\mathcal{F} = \{1, \ldots, m\}$ and a set of classes $\mathcal{K} = \{c_0, c_1\}$, where $c_0 = \mathit{false}$ and $c_1 = \mathit{true}$, i.e. non-defective and defective. Let the domain of feature $i \in \mathcal{F}$ be $\mathcal{D}_i$ and so the complete space of feature values is $\mathbb{F} = \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_m$. A specific point in feature space $\mathbb{F}$, also referred to as an *instance*, is denoted by $\mathbf{v} = \langle v_1, \ldots, v_m \rangle$ and represents an individual commit. In general, an arbitrary point in feature space is denoted by $\mathbf{x} = \langle x_1, \ldots, x_m \rangle$, where each variable $x_i \in \mathbf{x}$ takes values from its domain $\mathcal{D}_i$ and represents feature $i \in \mathcal{F}$. As a result, an ML classifier is assumed to define a classification function: $\tau : \mathbb{F} \to \mathcal{K}$.

In this paper, we focus on the two following classes of ML models. First, the *(binary) logistic regression* (LR) model predicts a commit $\mathbf{v} \in \mathbb{F}$ by evaluating the probability $p(\mathbf{v})$ of class *true*, given the log-odds $f(\mathbf{v})$ of this class, which is a linear combination of a set of features $\mathcal{F}$ and an intercept w_0, where each feature $i \in \mathcal{F}$ is associated with a coefficient $w_i \in \mathbb{R}$. Second, the *random forest* (RF) model [4] is an ensemble model that consists of decision trees T for a set of classes $\mathcal{K}$. Given a commit $\mathbf{v} \in \mathbb{F}$, each decision tree assigns a class to $\mathbf{v}$, and the final RF prediction for $\mathbf{v}$ is determined as the majority vote over all the classes assigned by the individual trees.

3.3 Formal Explanations

Let us define formal *abductive* (AXp) and *contrastive* (CXp) explanations and argue how they address the aforementioned limitations. Observe that both types of formal explanations below exploit the concept of *subset-minimality*. A set $\mathcal{S}$ is called subset-minimal with respect to some property $\mathfrak{P}$ if property $\mathfrak{P}$ holds for $\mathcal{S}$ but it does not hold for any of its *strict* subsets, i.e. for $\mathcal{S}' \subsetneq \mathcal{S}$. Note that the role of property $\mathfrak{P}$ in the definitions below is played by the corresponding conditions (1) and (2).

Definition 1: Abductive Explanation (AXp). Building on earlier work [13, 21, 38, 50], an AXp is defined as a subset-minimal set of features *sufficient* to explain the JIT defect prediction made by an ML model τ. More formally, given an instance $\mathbf{v} \in \mathbb{F}$ and a JIT defect prediction $c = \tau(\mathbf{v})$, an AXp $\mathcal{X}$ is a subset-minimal set of features, such that the following holds:

$$\forall (\mathbf{x} \in \mathbb{F}). \left[\bigwedge\nolimits_{i \in \mathcal{X}} (x_i = v_i) \right] \rightarrow (\tau(\mathbf{x}) = c) \tag{1}$$

Informally, (1) states that given an AXp $\mathcal{X}$ for the prediction $c = \tau(\mathbf{v})$, for any instance $\mathbf{x}$ in the *entire* feature space $\mathbb{F}$ *compatible* with instance $\mathbf{v}$ on the features of $\mathcal{X}$, classifier τ is guaranteed to predict c, no matter what values the other features (excluded from AXp $\mathcal{X}$) take in instance $\mathbf{x}$.

Example 3:. *Consider our running example. By examining the Qt dataset, the lower and upper bounds for the domains of the features considered are as follows:* $0 \leq x_{ld} \leq 309,728$, $0.0 \leq x_{ent} \leq 1.0$, $0 \leq x_{ndev} \leq 313$, *and* $0 \leq x_{dev\_exp} \leq 3,419$. *Given the possible values of the features, observe that a valid AXp* $\mathcal{X}$ *for the LR model's* τ_{lr} *prediction is shown in Figure 2. This AXp indicates that specifying* $ent = 0.97$ *and* $dev\_exp = 320$ *guarantees that the prediction for any compatible instance must be* $true$*, independently of the values of the other features, i.e. features* ld *and* ndev. *Indeed, even when* x_{ld} *and* x_{ndev} *take their lower bound values (both are 0), although the probability* $p(\mathbf{x})$ *of* true *gets smaller, it is still greater than 0.5 and, thus,* $\tau_{lr}(\mathbf{x}) = true$. *Similar reasoning can be applied in the case of the RF model, where the majority of the trees are guaranteed to predict* true *as long as* $ent = 0.97$ *and* $ndev = 3$. □

Clearly, the requirement imposed by (1) guarantees that the feature values of explanation $\mathcal{X}$ are sufficient for prediction c to hold in the entire feature space $\mathbb{F}$, i.e. there is no counterexample instance in $\mathbb{F}$ that would be compatible with explanation $\mathcal{X}$ but predicted differently than c. Hence, by definition, abductive explanations are *formally correct*. This is not the case for model-agnostic explanations, as was illustrated above for Anchor.

Note that multiple AXps may exist given an instance $\mathbf{v} \in \mathbb{F}$. Although not guaranteed to be robust by definition, AXps computed by the proposed algorithms are robust due to the deterministic nature of the underlying reasoners used. This is also confirmed by our experimental results provided in Section 6.

Finally, although AXps address 2 out of 3 limitations above, they can only provide a user with an indication for *why* a certain prediction was made, i.e. they are not designed to answer a *how?* question. This is what contrastive explanations defined below are capable of.

Definition 2: Contrastive Explanation (CXp). Following recent work [20, 38, 40], a CXp is a subset-minimal set of features that, if allowed to change their values, are *necessary* to flip the prediction provided by the JIT defect prediction model. Formally, given a JIT defect prediction $c = \tau(\mathbf{v})$, a CXp $\mathcal{Y}$ is a subset-minimal set of features, such that the following holds:

$$\exists(\mathbf{x} \in \mathbb{F}). \left[\bigwedge\nolimits_{i \notin \mathcal{Y}} (x_i = v_i)\right] \land (\tau(\mathbf{x}) \neq c) \tag{2}$$

In other words, if (2) holds for prediction $c = \tau(\mathbf{v})$, it means that there exists an instance $\mathbf{x}$ in the feature space $\mathbb{F}$ where all the features $i \notin \mathcal{Y}$ are equal to the values of instance $\mathbf{v}$ being explained but the features of $\mathcal{Y}$ are different, such that $\tau(\mathbf{x}) \neq c$.

Example 4. *Consider the instance* $\mathbf{v}$ *from* (3) *classified as* true *by both LR and RF models shown in Figure 2. Figure 2 shows a valid CXp* $\mathcal{Y} = \{ent\}$ *computed for the prediction made by the LR model* τ_{lr}*, since this prediction can be flipped if this feature is allowed to take another value from its domain despite other features equating to the values of* $\mathbf{v}$*. Namely, if the value of* ent *is changed to* 0.0 *given* $0.0 \leq x_{ent} \leq 1.0$*, the probability of* true *prediction is* $p(\mathbf{x}) < 0.5$*, i.e. the prediction is changed to* false*. Similarly, the same CXp (Figure 2) for the RF model indicates that changing the value of feature* ent *to* 0.0 *forces the 3 trees of the RF model* τ_{rf} *to predict* false, false *and* true*, respectively, which results in the majority vote (and so the overall RF prediction)* false. □

Observe that the *formal correctness* observations can be made with respect to contrastive explanations too, using the same rationale. Indeed, the validity of (2) guarantees the existence of an instance that is classified differently even though it is close to the original instance $\mathbf{v}$. The algorithms we apply for computing CXps have a deterministic nature and thus the CXps we compute are guaranteed to be *robust* (this is also confirmed by our experimental results). Finally, contrastive explanations are designed to provide a user with an answer to the *how?* question, i.e. how the instance needs to be changed in order to change the prediction. Furthermore, subset-minimality of a CXp implies that the changes suggested by the CXp are *minimal* with respect to the original instance $\mathbf{v}$.

Finally, recent work has shown that given a prediction $c = \tau(\mathbf{v})$, AXps and CXps are related through the *minimal hitting set duality* [20, 46]. Given a set of sets $\mathbb{S}$, a *hitting set* of $\mathbb{S}$ is a set H such that $\forall S \in \mathbb{S}, S \cup H \neq \emptyset$, i.e. H "hits" every set in $\mathbb{S}$. A hitting set H for $\mathbb{S}$ is *minimal* if none of its strict subsets is also a hitting set. The minimal hitting set duality represents the fact that given a prediction $c = \tau(\mathbf{v})$, each AXp for the prediction is a *minimal hitting set* (MHS) of the set of all CXps $\mathbb{C}_\tau(\mathbf{v}, c)$ for that prediction, and the other way around: each CXp is an MHS of the set of all AXps $\mathbb{A}_\tau(\mathbf{v}, c)$. By slightly abusing the notation, $\mathbb{A}_\tau(\mathbf{v}, c) = \text{MHS}(\mathbb{C}_\tau(\mathbf{v}, c))$ and $\mathbb{C}_\tau(\mathbf{v}, c) = \text{MHS}(\mathbb{A}_\tau(\mathbf{v}, c))$. The algorithms we use in our approach heavily rely on this duality relation.

Algorithm 1 MARCO-like Anytime Explanation Enumeration

```
1: procedure XpEnum(τ, v, c)
2:     (𝔸, ℂ) ← (∅, ∅)                          ▷ Sets of AXp's and CXp's to collect.
3:     while resources available do
4:         𝒴 ← MinimalHS(𝔸, ℂ)                  ▷ Get a new MHS of 𝔸 subject to ℂ.
5:         if 𝒴 = ⊥ then break                  ▷ Stop if none is computed.
6:         if ∃(x ∈ 𝔽). ⋀_{i∉𝒴}(x_i = v_i) ∧ (τ(x) ≠ c) then   ▷ Check CXp condition (2) for 𝒴.
7:             ℂ ← ℂ ∪ {𝒴}                      ▷ 𝒴 appears to be a CXp.
8:         else                                 ▷ There must be a missing AXp 𝒳 ⊆ ℱ \ 𝒴.
9:             𝒳 ← ExtractAXp(ℱ \ 𝒴, τ, v, c)   ▷ Get AXp 𝒳 by iteratively checking (1) [21].
10:            𝔸 ← 𝔸 ∪ {𝒳}                      ▷ Collect new AXp 𝒳.
    return 𝔸, ℂ
```

Example 5. *Consider our running example. There is a single AXp for the target instance and so the full set of AXps for the RF prediction of instance* $\mathbf{v}$ *is* $\mathbb{X} = \{\{ent, ndev\}\}$. *The full set of CXps for the same prediction is* $\mathbb{Y} = \{\{ent\}, \{ndev\}\}$. *Observe how the only AXp minimally hits each CXp from* $\mathbb{Y}$ *and, vice versa, each CXp minimally hits the AXp from* $\mathbb{X}$. □

Definition 3: Formal Feature Attribution (FFA). Given the definition of AXp's above, we can now illustrate the *formal feature attribution* (FFA) function by Yu *et al* [65]. Denoted as $\text{ffa}_\tau(i, (\mathbf{v}, c))$, it returns for a classification $\tau(\mathbf{v}) = c$ how important feature $i \in \mathcal{F}$ is in making this classification, defined as the proportion of AXp's for the classification $\mathbb{A}_\tau(\mathbf{v}, c)$, which include feature i, i.e.

$$\text{ffa}_\tau(i, (\mathbf{v}, c)) = \frac{|\{\mathcal{X} \mid \mathcal{X} \in \mathbb{A}_\tau(\mathbf{v}, c), i \in \mathcal{X})|}{|\mathbb{A}_\tau(\mathbf{v}, c)|} \tag{3}$$

Yu *et al* [65] define an anytime algorithm for computing FFA shown in Algorithm 1. The algorithm collects AXp's $\mathbb{A}$ and CXp's $\mathbb{C}$. They are initialized to empty. While we still have resources, we generate a minimal hitting set $\mathcal{Y} \in \text{MHS}(\mathbb{A})$ of the current known AXp's $\mathbb{A}$ and not already in $\mathbb{C}$ with the call MinimalHS($\mathbb{A}, \mathbb{C}$). If no (new) hitting set exists then we are finished and exit the loop. We then check if (2) holds in which case we add the candidate to the set of CXp's $\mathbb{C}$. Otherwise, we know that $\mathcal{F} \setminus \mathcal{Y}$ is a correct (non-minimal) abductive explanation, i.e. it satisfies (1). We use the call ExtractAXp to minimize the resulting explanation, returning an AXp $\mathcal{X}$ which is added to the collection of AXp's $\mathbb{A}$. ExtractAXp tries to remove features i from $\mathcal{F} \setminus \mathcal{Y}$ one by one while still satisfying (1). When resources are exhausted, the loop exits and we return the set of AXp's and CXp's currently discovered.

Example 6. *Consider our running example in Figure 2. As illustrated in Example 5, the complete set of AXps for the RF prediction of instance* $\mathbf{v}$ *is* $\mathbb{X} = \{\{ent, ndev\}\}$. *Therefore, with Equation* (3) *we can compute that the FFA for* $\mathbf{v}$ *is* {ent: 1, ndev: 1}, *indicating that both the features* ent *and* ndev *make an equal contribution to the RF prediction, while the other two features, i.e.* ld *and* dev_exp, *hold no contribution.* □

3.4 On Propositional Satisfiability

Due to the need to logically reason about the behavior of an ML model of interest, e.g. to be able to check the validity of (1) and (2) in the enumeration process described above, the general setup of FoX makes use of the propositional satisfiability (SAT) reasoning. As a result, the notation and standard definitions widely used in SAT solving are assumed [3]. Namely, we assume formulas to be propositional. A propositional formula φ is said to be in *conjunctive normal form* (CNF) if

it is a conjunction of clauses, where each *clause* is a disjunction of literals. A *literal* is either a Boolean variable taking values 0 or 1, or its negation. Whenever convenient, a clause is considered to be a set of literals. A *truth assignment* μ is a mapping from the variables in φ to $\{0, 1\}$. Namely, $\mu(i) = b$, $b \in \{0, 1\}$ represents the fact the assignment μ sets the i'th variable to b. By slightly abusing the notation, we will use μ_i to represent either positive or negative literal on the i'th variable, depending on whether $\mu(i) = 1$ or $\mu(i) = 0$. A clause is satisfied by a truth assignment μ if at least one of literals in the clause is assigned value 1; otherwise, the clause is falsified. If there exists an assignment μ that satisfies all clauses in a CNF formula φ then formula φ is satisfiable; otherwise, it is unsatisfiable.

4 FOX: FORMALLY EXPLAINING JIT DEFECT PREDICTIONS

In this section, we introduce the method we use for extracting formal explanations for JIT defect predictions made by LR and RF models. Note that it builds on the existing principled approach to computing formal abductive and contrastive explanations studied in the general context of XAI [13, 21, 38, 50] and their enumeration [20]. By building on the above, we describe FoX as follows.

Overview. Figure 3 depicts an overview of FoX, which comprises 3 main steps. First, a user (a software developer) selects an ML model (LR or RF) and a target instance, i.e. a commit, such that an explanation for the model's prediction is sought for the instance. Second, FoX encodes both the model and the instance into a logical formalism (into a SAT formula) such that formal reasoning about satisfiability of the corresponding propositional formula can be applied. Note that, as was shown in [37], LR models admit effective reasoning procedures that operate *directly* on the original linear representation of the classifier, i.e. no logical encoding is necessary for LR models (more on this below), which is still needed for RF models. Given a suitable representation of the classifier, this step then iteratively extracts either a required number of AXps and/or CXps or enumerates them exhaustively. All generated explanations are stored in a solution pool. (We should say here that there can be a multitude of valid explanations for each model prediction, and some of them may be less interesting to the user than the other. As such, explanation enumeration helps the user obtain multiple possible reasons for the prediction and opt for the best ones depending on given criteria.) In the third step, explanations are selected from the pool based on the users' explanation preferences. Alternatively, the tool reports a feature importance explanation in the form of FFA based on all the AXps and CXps enumerated. Afterwards, these explanations are sent back to the user to help them prioritize QA resources on the most risky commits.

A few words should be said regarding the extraction of a *single* formal explanation (i.e. ExtractAXp or ExtractCXp in Algorithms 1 and 2). In general [21], explanation extraction works as follows. Given a target instance $\mathbf{v} \in \mathbb{F}$, the procedure (i) considers a working set of features $\mathcal{S}$ (for instance, $\mathcal{S}$ can be initially set to include all features $\mathcal{F}$) and (ii) traverses each element $i \in \mathcal{S}$ *one-by-one* checking if feature i can be discarded from $\mathcal{S}$. Depending on the type of explanation we are interested in (AXp or CXp), this check is implemented by calling an *oracle* deciding whether or not set $\mathcal{S} \setminus \{i\}$ satisfies the explanation condition (conditions (1) or (2)). If it does, then feature i is discarded; otherwise, it is kept in $\mathcal{S}$. The algorithm proceeds until all features of $\mathcal{S}$ are tested and ends up making $\mathcal{S}$ a subset-minimal AXp or CXp (depending on the condition checked). Note that in general, the checks above involve calling a formal reasoner, e.g. a SAT solver, which solves an NP-hard problem [10] for each of the features tested. However, as was mentioned above, some ML models admit polynomial-time procedures that can replace a potentially expensive NP-oracle call. These include LR models discussed in Section 4.1.

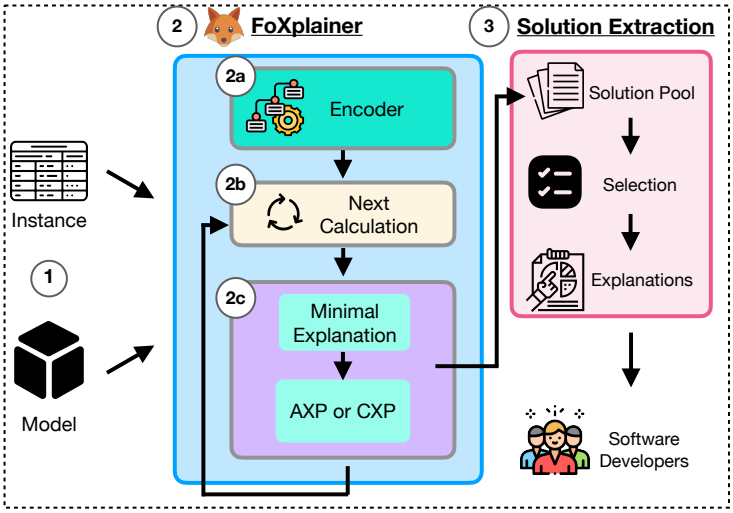

Fig. 3. An overview of FoX to extract explanations. Given an instance to be explained and an ML model, FoX consists of 3 main components: (1) formal encoding, (2) explanation enumeration, and (3) solution pool with the ability to select explanations given user preferences. FoX extracts two types of formal explanations, i.e. AXps and CXps. It can also report FFA.

Algorithm 2 SAT-Based Formal Explanation Enumeration [37]

procedure XpEnum(τ, $\mathbf{v}$, c)
  $(\mathbb{A}, \mathbb{C}) \leftarrow (\emptyset, \emptyset)$ ▷ *Sets of AXp's and CXp's to collect.*
  $\varphi \leftarrow \emptyset$ ▷ *Space to explore.*
  while resources available **do**
    $\mu \leftarrow \text{SAT}(\varphi)$ ▷ *Another unexplored subset?*
    if $\mu = \bot$ **then break** ▷ *Stop if none is computed, i.e. φ is unsatisfiable.*
    $\mathcal{S} \leftarrow \{i \in \mathcal{F} \mid \mu_i = 1\}$ ▷ *Extract the set from model μ.*
    if $\forall(\mathbf{x} \in \mathbb{F}). [\bigwedge_{i \in \mathcal{S}}(x_i = v_i)] \rightarrow (\tau(\mathbf{x}) = c)$ **then** ▷ *Check AXp condition (1) for $\mathcal{S}$.*
      $\mathcal{X} \leftarrow \text{ExtractAXp}(\mathcal{S}, \tau, \mathbf{v}, c)$ ▷ *Get AXp $\mathcal{X}$ by iteratively checking (1) [21].*
      $\mathbb{A} \leftarrow \mathbb{A} \cup \{\mathcal{X}\}$ ▷ *Collect new AXp $\mathcal{X}$.*
      $\varphi \leftarrow \varphi \cup \{(\bigvee_{i \in \mathcal{X}} \neg\mu_i)\}$ ▷ *Block new AXp $\mathcal{X}$.*
    else ▷ *There must be a missing CXp $\mathcal{Y} \subseteq \mathcal{F} \setminus \mathcal{S}$.*
      $\mathcal{Y} \leftarrow \text{ExtractCXp}(\mathcal{F} \setminus \mathcal{S}, \tau, \mathbf{v}, c)$ ▷ *Get CXp $\mathcal{Y}$ by iteratively checking (2) [20].*
      $\mathbb{C} \leftarrow \mathbb{C} \cup \{\mathcal{Y}\}$ ▷ *Collect new CXp $\mathcal{Y}$.*
      $\varphi \leftarrow \varphi \cup \{(\bigvee_{i \in \mathcal{Y}} \mu_i)\}$ ▷ *Block new CXp $\mathcal{Y}$.*
  return $\mathbb{A}$, $\mathbb{C}$

4.1 Special Case of Formal LR Explanations

Recent work [37] proposed polynomial-time algorithms to extracting a single AXp or CXp for *monotonic* classifiers. Note that monotonicity of the target classifier (on each feature) is the only requirement on the classifier imposed by [37]. Also observe that LR classifiers, as being a weighted linear sum, are monotonic on all features. As a result, we can apply the approach of [37] to LR classifiers and consider a set of ordered classes $\mathcal{K} = \{c_0 = false, c_1 = true\}$ in JIT defect predictions, where $c_0 < c_1$. Without loss of generality, assume that our LR classifier τ makes use of the first m

features with coefficients $\mathbf{w} = \{w_1, \ldots, w_m\}$, $m \leq |\mathcal{F}|$, s.t $w_i \geq 0\ \forall i \in \{1, \ldots, m\}$. Then, given two concrete instances $\underline{\mathbf{v}} \in \mathbb{F}$ and $\overline{\mathbf{v}} \in \mathbb{F}$ such that $\underline{\mathbf{v}}_i \cdot w_i \leq \overline{\mathbf{v}}_i \cdot w_i$ for any $i \in \{1, \ldots, m\}$, monotonicity of τ ensures that $\tau(\underline{\mathbf{v}}) \leq \tau(\overline{\mathbf{v}})$.

Given the above, also observe that in our setup, each feature $i \in \mathcal{F}$ takes values from a domain $\mathcal{D}_i$ and so has concrete lower bound $\lambda_i = \min \mathcal{D}_i$ and upper bound $\mu_i = \max \mathcal{D}_i$. (Since by assumption, each $w_i \geq 0$, the values of λ_i and μ_i also define the lower and upper bounds on $w_i \cdot \lambda_i$ and $w_i \cdot \mu_i$.) Recall that we seek an explanation for the prediction $c = \tau(\mathbf{v})$ for a particular instance $\mathbf{v} \in \mathbb{F}$ and given feature i let us denote by $\underline{\mathbf{v}}$ and $\overline{\mathbf{v}}$ the instances where the value $v_i \in \mathcal{D}_i$ is replaced by λ_i and μ_i, respectively.

The key observation of [37] is that checking whether or not a feature $i \in \mathcal{S} = \mathcal{F}$ should be included in the explanation can be replaced by a simple test of whether making this feature *free* allows τ to change its value when all the other features from $\mathcal{S} \setminus \{i\}$ are fixed to the values of $\mathbf{v}$. Essentially, this can be done by testing whether $\tau(\underline{\mathbf{v}}) = \tau(\overline{\mathbf{v}})$. If the equality is violated then feature i is important as changing its value also changes the value of τ. In this case, it is put back to set $\mathcal{S}$. Otherwise, feature i is made free, i.e. it is dropped from $\mathcal{S}$. Afterwards, we proceed by considering feature $i + 1$ in a similar fashion. Importantly, if i is made free then the lower and upper bound instances $\underline{\mathbf{v}}$ and $\overline{\mathbf{v}}$ are kept updated as *extreme* instances for our target instance $\mathbf{v}$. This way, during the overall process, the values of the features i in $\underline{\mathbf{v}}$ ($\overline{\mathbf{v}}$, resp.) are either kept equal to v_i or take value λ_i (μ_i, resp.).

Traverse	Working set $\mathcal{S}$	Feature i	$\mathcal{S} \setminus i : \tau(\underline{\mathbf{v}}) = \tau(\overline{\mathbf{v}})$?	Discard?
1	$\{ld, ent, ndev, dev\_exp\}$	ld	true	true
2	$\{ent, ndev, dev\_exp\}$	ent	false	false
3	$\{ent, ndev, dev\_exp\}$	$ndev$	true	true
4	$\{ent, dev\_exp\}$	dev_exp	false	false
—	$\{ent, dev\_exp\}$	—	—	—

Fig. 4. Single AXp Extraction in LR Models.

Example 7. *Consider the commit* $\mathbf{v}$ *and LR model* τ *described in Figure 2. Figure 4 illustrates the process of a single AXp extraction in LR models.*[10] *The procedure initializes the working set* $\mathcal{S} = \{ld, ent, ndev, dev\_exp\}$ *and then traverses each feature* $i \in \mathcal{S}$. *Feature* i *is discarded if* $\tau(\underline{\mathbf{v}}) = \tau(\overline{\mathbf{v}})$ *when fixing* $\mathcal{S} \setminus \{i\}$, *i.e.* $\mathcal{S} \setminus \{i\}$ *satisfies* (1). *As can be observed in the first row in Figure 4 where feature ld is traversed,* $\underline{\mathbf{v}} = \langle \underline{v}_{ld} = \lambda_{ld} = 0, \underline{v}_{ent} = 0.97, \underline{v}_{ndev} = 3, \underline{v}_{dev\_exp} = 320 \rangle$ *and* $\overline{\mathbf{v}} = \langle \underline{v}_{ld} = \mu_{ld} = 309,728, \underline{v}_{ent} = 0.97, \underline{v}_{ndev} = 3, \underline{v}_{dev\_exp} = 320 \rangle$ *for* $\mathcal{S} \setminus \{i\} = \{ent, ndev, dev\_exp\}$ *such that* $\tau(\underline{\mathbf{v}}) = \tau(\overline{\mathbf{v}})$, *and thus feature* ld *is discarded. Similarly, the third row indicates that feature* ndev *is discarded since* $\tau(\underline{\mathbf{v}}) \neq \tau(\overline{\mathbf{v}})$ *for* $\mathcal{S} \setminus \{i\} = \{ent, dev\_exp\}$. □

4.2 Formal Explanation Enumeration

Note that in the explanation extraction procedure outlined above (both the general SAT-based case and the simplified case of LR models), we start from the complete set of features $\mathcal{F}$. In practice, however, one can bootstrap the procedure with a subset of features $\mathcal{S} \subseteq \mathcal{F}$ that is guaranteed to satisfy the corresponding explanation condition (either (1) or (2)). This fact is exploited by the formal explanation enumeration [20, 37]. A high-level view on formal explanation enumeration is outlined in Algorithm 2 (readers are referred to [20, 37] for more details). Note that it differs from the procedure of Algorithm 1 as it does not rely on minimal hitting set enumeration when

[10] In a similar vein, when extracting a CXp, the procedure replaces (2) checks for subsets $\mathcal{S} \setminus \{i\}$ with checks testing whether or not $\tau(\underline{\mathbf{v}}) \neq \tau(\overline{\mathbf{v}})$ for $\mathcal{S} \setminus \{i\}$.

Table 1. An overview of the studied JIT defect datasets provided by McIntosh and Kamei [39].

Project	Training Data				Testing Data			
	Start Date	End Date	# Commits	# Defective Commits	Start Date	End Date	# Commits	# Defective Commits
Openstack	11/30/2011	08/13/2013	9,246	980 (11%)	08/13/2013	02/28/2014	3,963	646 (16%)
Qt	06/18/2011	05/08/2013	19,312	1,577 (8%)	05/08/2013	03/18/2014	8,277	476 (6%)

computing candidate features subsets to test. This is because in some practical scenarios, e.g. for LR models, it is cheaper to extract a subset-minimal explanation by iteratively checking either condition (1) or (2) than to delegate this task to the minimal hitting set enumerator in use.

To make the enumeration work, one needs to keep track of all subsets of $\mathcal{F}$ that are already seen and iteratively identify yet unseen subset $\mathcal{S} \subseteq \mathcal{F}$ (see lines 5–7) to check whether or not this new subset $\mathcal{S}$ satisfies (1) (cf. line 8). If it does then AXp extraction can be performed starting from set $\mathcal{S}$ and applying the linear search feature traversal augmented with oracle calls checking (1). Otherwise, $\mathcal{S}$ does not satisfy (1). Observe that in this case subset $\mathcal{F} \setminus \mathcal{S}$ satisfies (2) (cf. line 12), i.e. CXp extraction can be performed starting from subset $\mathcal{F} \setminus S$. In either case, a subset-minimal explanation is extracted and collected in each iteration of the enumeration algorithm.

Note that the space of all possible subsets to explore is modeled in Algorithm 2 with the use a CNF formula φ on Boolean variables μ_i, $i \in \mathcal{F}$ (see line 3). (Recall from Section 3.4 that we use μ to represent a truth assignment, and μ_i are meant to represent individual Boolean literals as defined by this assignment.) The meaning of each variable μ_i is that the corresponding feature i is *selected* for inclusion in the target subset $\mathcal{S}$ if and only if $\mu_i = 1$. At the beginning, formula φ has no clauses, which makes *any* subset of features initially possible. As soon as an AXp $\mathcal{X}$ is computed, the variables μ_i, $i \in \mathcal{X}$, are used to construct a clause $(\bigvee_{i \in \mathcal{X}} \neg\mu_i)$ and add it to formula φ, thus, forbidding the same explanation to be computed in the next iterations of the algorithm (line 11). On the other hand, if a CXp $\mathcal{Y}$ is computed, the corresponding variables u_i are used to add to formula φ another clause $(\bigvee_{i \in \mathcal{X}} \mu_i)$, which forces the next iterations to include some of the features of CXp $\mathcal{Y}$ (line 15). Overall, this algorithmic setup ensures that the enumeration process is guided by the already computed CXps and that no previously computed AXp is repeated. Observe that this is correct thanks to the known minimal hitting set duality between AXps and CXps [20] illustrated in Example 5 and also exploited in Algorithm 1. Finally, depending on our setup, the process can proceed until we complete explanation enumeration or until available computational resources, e.g. time, are exhausted. As a result, we can provide a user with a computed set of AXps or CXps we well as compute and report FFA values.

Iteration	$\mathcal{S}$	$\mathcal{S}$ satisfies (1)?	$\mathcal{X}$	$\mathcal{Y}$	Clause added to φ
1	$\{\}$	false	—	$\{ndev\}$	(u_{ndev})
2	$\{ndev\}$	false	—	$\{ent\}$	(u_{ent})
3	$\{ent, ndev\}$	true	$\{ent, ndev\}$	—	$(\neg u_{ent} \vee \neg u_{ndev})$

Fig. 5. Explanation Enumeration.

Example 8. *Consider our running commit* **v** *and RF model* τ *depicted in Figure 2. Figure 5 illustrates the AXp and CXp enumeration process of Algorithm 2. In the first iteration, an empty target set* $\mathcal{S}$ *is identified. Since* $\mathcal{S}$ *is not able to satisfy* (1)*,* $\mathcal{F} \setminus \mathcal{S} = \{ld, ent, ndev, dev\_exp\}$ *is the initial working set for the single CXp extraction, and thus a CXp* $\mathcal{Y} = \{ndev\}$ *is extracted and feature* ndev *must be included for the target sets in the next iterations. Similarly, the next candidate* $\mathcal{S}' = \{ndev\}$ *fails to satisfy* (1) *and so its complement* $\mathcal{F} \setminus \mathcal{S}'$ *is used to extract a CXp* $\mathcal{Y}' = \{ent\}$*; observe that feature* ent *has to be selected for the later candidate sets. Afterwards,* $\mathcal{S}'' = \{ent, ndev\}$ *is selected as the target set*

Table 2. The commit-level software features for Just-In-Time Defect Prediction provided by McIntosh and Kamei [39]. The asterisk symbol indicates the features selected by AutoSpearman that ensure non-collinearity among features.

Category	Name	Description
Size	LA*	The number of lines added by a commit.
	LD*	The number of lines deleted by a commit.
Diffusion	NS*	The number of modified subsystems.
	ND*	The number of modified directories.
	NF	The number of modified files.
	Complexity	The entropy of modified lines that spread across changed files.
History	Unique changes	The number of prior changes to the modified files.
	Developers	The number of developers who have changed the modified files in the past.
	Age*	The time interval between the last and current changes.
Experience	Prior changes	The number of prior changes that an author has participated in.
	Recent changes	The number of prior changes that an author has participated in weighted by the age of the changes.
	Subsystem changes	The number of prior changes to the modified subsystem(s) that an author has participated in.
	Awareness	The proportion of the prior changes to the modified subsystem(s) that an author has participated in.
Review	Iterations	The number of times that a change was revised prior to integration.
	Reviewers*	The number of reviewers who have voted on whether a change should be integrated or abandonded.
	Comments	The number of non-automated, non-owner comments posted during the review of a change.
	Review window*	The length of time between the creation of a review request and its final approval for integration.

in the third iteration. As $\mathcal{S}''$ satisfies (1)*, it is then used for extracting a single AXp* $\mathcal{X} = \{ent, ndev\}$*, and either* ent *or* ndev *is forbidden in next iterations. Finally, formula* φ *becomes unsatisfiable indicating that there exists no unseen set to explore, and the enumeration process terminates. At this point, we can report either the only AXp identified or two CXps enumerated. Finally, we can use the AXp to figure out that* $\mathit{ffa}_\tau(1, (\mathbf{v}, c)) = 0$*,* $\mathit{ffa}_\tau(2, (\mathbf{v}, c)) = 1$*,* $\mathit{ffa}_\tau(3, (\mathbf{v}, c)) = 1$*, and* $\mathit{ffa}_\tau(4, (\mathbf{v}, c)) = 0$ *as features 2 and 3 (*ent *and* ndev*) participate in the only AXp available for instance* $\mathbf{v}$ *while features 1 and 4 (*ld *and* dev_exp*) do not.* □

5 EXPERIMENTAL DESIGN

In this section, we present the studied datasets and explain the details of our experimental design.

Studied JIT Datasets. In our experiment, we build JIT defect models based on the datasets provided by McIntosh and Kamei [39], which were previously used in the PyExplainer paper [44]. These JIT defect datasets represent two large-scale open-source software projects, i.e. Openstack and Qt. They are chosen in order to ensure a fair comparison with the PyExplainer paper [44]. In addition, these datasets are widely-used as a benchmark JIT defect suite [16, 17, 39, 43, 44] and have been manually verified to ensure the validity of the SZZ algorithm [54] to reduce the number of false positives and false negatives [11, 64]. Openstack is an open-source software for cloud infrastructure service. Qt is a cross-platform application development framework.

Table 1 provides an overview of the studied datasets. For each dataset, there are 17 commit-level features that span across 5 dimensions, i.e., Size (e.g., lines added, lines deleted), Diffusion (e.g., #modified files), History (#developers), Experience, and Code Review Activities. The list of the studied features is provided in Table 2.

Experiment Design. To ensure a fair comparison, we use the same experimental setup as the PyExplainer paper [44] as follows:

(Step 1) Data Splitting. To avoid any temporal bias in our experiment, we first preserve the order of commits by sorting based on their commit date [43, 59]. Then, we use a time-wise hold-out validation technique (as used by McIntosh and Kamei [39]) to split the dataset into training (70%)

and testing (30%) datasets. The use of the time-wise hold-out validation technique ensures that the commits that appear later will not be used in the model training. Similarly, the commits that appear earlier will not be used in the model evaluation.

(Step 2) Build Global JIT Defect Models. For each training dataset, we first mitigate collinearity using AutoSpearman and handle class imbalance using SMOTE prior to building JIT defect models. After using AutoSpearman, we finally select 7 features that are not highly-correlated with each other. To ensure that the predictions of our JIT defect models are highly accurate, we apply a class rebalancing technique, as suggested by prior work [1, 55]. Since the defective ratio of our studied JIT defect datasets are highly imbalanced (i.e., 8%-16%), we apply SMOTE [6] to handle class imbalance *only on the training dataset*. Then, we build global JIT defect models using the training data of each project. Same as the PyExplainer paper [44], we use two classification techniques, i.e., Logistic Regression (LR) and Random Forest (RF). We find that our global JIT defect models achieve an AUC of 0.66 (LR) and 0.75 (RF) for OpenStack and an AUC of 0.64 (LR) and 0.74 (RF) for Qt, which are the same as the PyExplainer paper [44].

(Step 3) Explanation Generation. Then, we apply FoX to generate all (abductive and contrastive) explanations for each prediction of JIT defect models. Similarly, given a prediction, we also generate an explanation using the four baseline model-agnostic techniques, namely, LIME [47], SHAP [36], Anchor [48], PyExplainer [44]. These four approaches have been previously used in many prior studies of explainable defect prediction models [7, 24, 25, 32, 42, 44, 45].

6 EXPERIMENTAL RESULTS

In this section, we compare FoX with four model-agnostic techniques, i.e., LIME [47], SHAP [36], Anchor [48], PyExplainer [44] with respect to correctness, robustness and efficiency. Anchor and PyExplainer are feature selection approaches, generating features that are deemed sufficient for a given prediction , while LIME and SHAP produce feature attributions, revealing the contributions of individual features to the prediction. Note that FoX can compute features adequate for a given prediction and also compute FFA indicating the contributions of features.

(RQ1) Does FoX generate more correct explanations than existing approaches for Just-In-Time defect predictions?

Approach. To answer RQ1, we analyze the percentage of provably correct explanations computed by the four model-agnostic techniques. For feature selection approaches, e.g. PyExplainer and Anchor, the provably correct explanations mean *why?* explanations that satisfy (1) or *how?* explanations that satisfy (2), while for feature attribution approaches, e.g. LIME and SHAP, FFA serves as provably correct feature attributions.

Therefore, feature attributions generated by LIME and SHAP are compared with FFA using three metrics, namely errors, Kendall's Tau [27] and rank-based overlap (RBO) [62]. The *error* is quantified by the sum of absolute differences across all features, i.e. Manhattan distance, while Kendall's Tau and RBO are used to compare rankings of feature attributions.[11] Kendall's Tau and RBO are assessed within the ranges of [-1, 1] and [0, 1], respectively. A higher value for both metrics signifies stronger alignment or closeness between a ranking and FFA.

Since existing model-agnostic techniques are computationally expensive, we only randomly select 500 different instances in each testing dataset for evaluation. As PyExplainer generates at most 2,000 explanations answering *why* questions in each instance to be explained, we only analyze the correctness of the explanation with the highest importance score. For Anchor, we measure the

[11]Kendall's Tau is a correlation coefficient that evaluates the ordinal relationship between two ranked lists, providing a way to measure the similarity in the order of values. RBO is a metric that measures the closeness between two ranked lists, considering both the order and the depth of the overlap.

Table 3. (RQ1) Comparison between FoX's FFA against LIME and SHAP. (↘) Error = Better. (↗) Higher Kendall's Tau and RBO = Better.

Approach	Model	Openstack			Qt		
		Error	Tau	RBO	Error	Tau	RBO
LIME	LR	4.818	0.047	0.551	5.630	-0.086	0.447
	RF	6.089	0.064	0.516	7.847	0.063	0.358
SHAP	LR	5.098	0.003	0.531	5.215	-0.129	0.437
	RF	5.645	0.118	0.596	6.867	0.134	0.306

correctness of the only explanation computed in each instance. FoX enumerates all explanations for an instance, including AXps and CXps. Thus, we analyze the correctness of all the AXps and CXps computed. To compare LIME and SHAP with FFA, we take their outcomes substituting negative attributions by their positive counterpart (i.e. taking the absolute value) and then normalize the values into the range of [0, 1], which corresponds to the same scale as FFA.

Result. **In contrast to the explanations provided by FoX, heuristic explanations for Just-In-Time defect predictions are susceptible to the lack of correctness.** First, let us consider feature selection explanations. By definition, explanations computed by FoX are guaranteed to 100% correct for all instance predictions made by LRs and RFs for both studied datasets. On the other hand, according to our results, none of the explanations, i.e. 0%, computed by Anchor and PyExplainer are correct for either of the models in the two selected datasets. This stark difference clearly demonstrates that FoX is advantageous over the model-agnostic feature selection competition with respect to explanation correctness.

As for feature attribution approaches, the importance of features as reported by model-agnostic LIME and SHAP is compared against FFA generated by FoX and detailed in Table 3. For each dataset, we compute the metric for each individual instance and afterwards average the outcomes to acquire the final result of the dataset. Observe that the errors of LIME's feature attributions in LRs are 4.818 and 5.630 for Openstack and Qt, respectively, while the corresponding values in SHAP are 5.098 and 5.215. On the other hand, the errors in RFs are higher compared to LRs. LIME's errors are 6.089 and 7.847 in RFs in these two datasets, whereas SHAP's errors are 5.645 and 6.867. LIME and SHAP also demonstrate similar performance with respect to the two ranking comparison metrics. The values of Kendall's Tau for LIME (resp. SHAP) range from -0.086 to 0.064 (resp. from -0.129 to 0.134). In terms of RBO values, LIME shows results between 0.358 and 0.551, whereas the values in SHAP span from 0.306 to 0.596. In summary, as indicated by Table 3, both LIME and SHAP fail to achieve strong closeness to FFA.

(RQ2) Does FoX generate more robust explanations than existing approaches for Just-In-Time defect predictions?

Approach. To answer RQ2, we evaluate the percentage of robust explanations generated by all the competitors. A higher percentage of robust explanations in an approach indicates that the approach is more robust. We run each experiment twice and compare the explanations computed for the same instance afterwards. For feature selection approaches, if a method can compute *identical explanations* in two separate experiments then we conclude that the approach computes a robust explanation for the instance. On the other hand, an explanation in feature attribution approaches is deemed robust if the approach consistently produces *the same ranking* of feature attributions for a given prediction in the two experiments. Similar to RQ1, we consider the most important

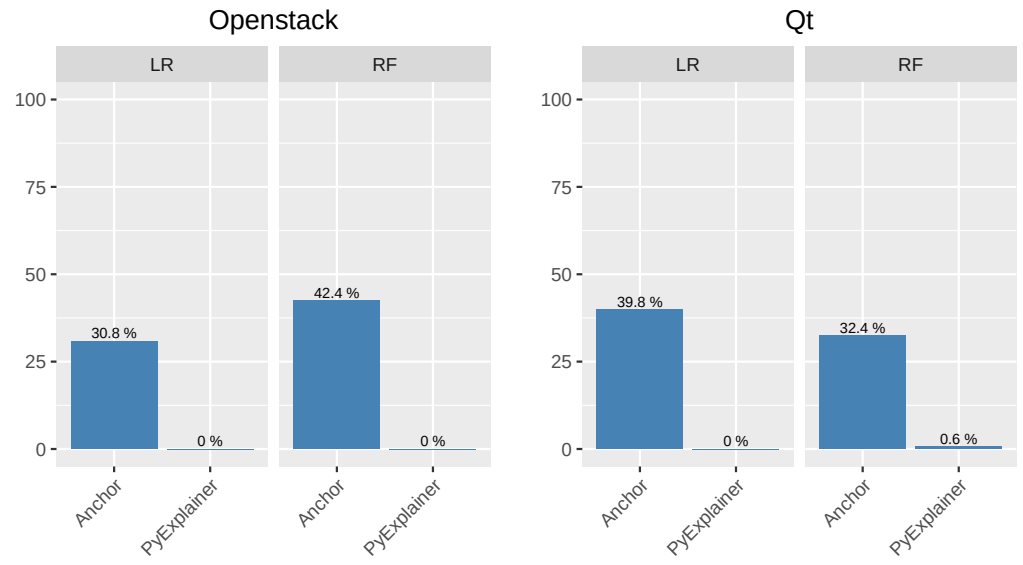

Fig. 6. (RQ2) Robustness of feature selection methods. FoX achieves perfect robustness (i.e., 100%) on both datasets and thus is not shown in the bar chart. (↗) Higher Robustness = Better.

explanation for an instance in PyExplainer, while we replace the solutions of LIME and SHAP by their absolute values.

Result. **FoX is guaranteed to compute 100% robust explanations.** The results of feature selection approaches regarding the percentage of robust explanations for instances to be explained are shown in Figure 6. We should emphasize that FoX always generates robust explanations for all LR and RF predictions for two studied datasets. Due to the deterministic nature of the modern SAT solvers, FoX computes not just a single robust explanation but instead it enumerates all explanations in the same order, and the order can be dictated by user-defined preferences.

In contrast, PyExplainer is not able to extract robust explanations for the studied ML models and datasets. For Openstack dataset Anchor computes 30.8% and 42.4% robust explanations for the LR and RF models respectively, while this approach generates 39.8% and 32.4% robust explanations for the predictions of instances in Qt dataset for the two models. Figure 7 illustrates the percentage of robust explanations for instances in feature attribution approaches. As FoX always computes robust explanations in the same order, FoX always generates robust FFA. 100% robust explanations can also be generated by SHAP which is deterministic. On the contrary, LIME fails to generate any robust explanation. In summary, due to the deterministic nature of the modern SAT solvers, the explanations generated by FoX are guaranteed to be robust and this achievement is only demonstrated in one of the four other model-agnostic techniques, i.e. SHAP.

(RQ3) Does FoX generate explanations faster than existing approaches for Just-In-Time defect predictions?

Approach. To answer RQ3, we assess the average runtime of generating one explanation per instance measured for FoX and the other four techniques. Lower runtime for computing an explanation denotes that the approach can more efficiently explain a JIT defect prediction. In contrast to

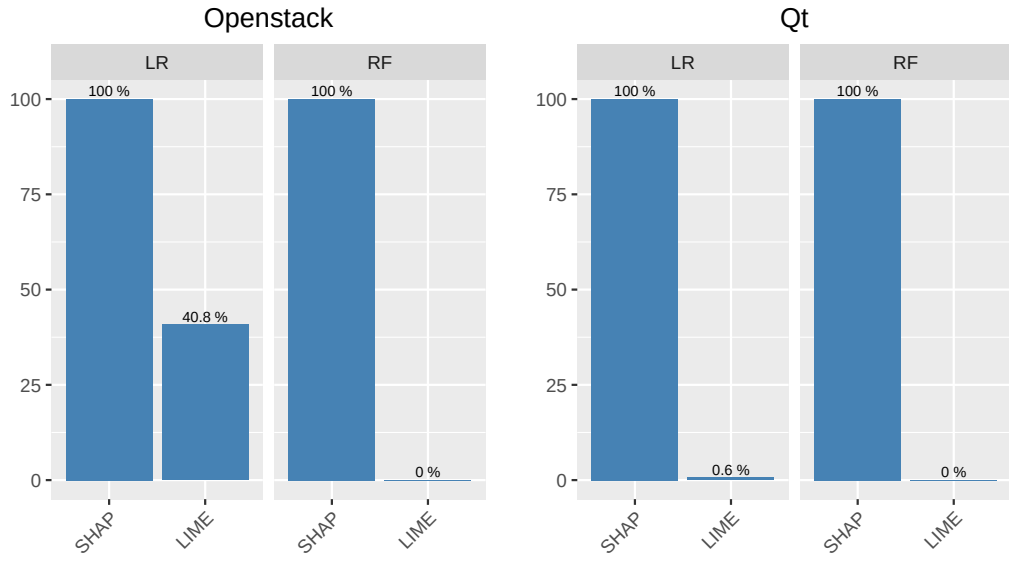

Fig. 7. (RQ2) Robustness of feature attribution methods. FoX achieves perfect robustness (i.e., 100%) on both datasets and thus is not shown in the bar chart. (↗) Higher Robustness = Better.

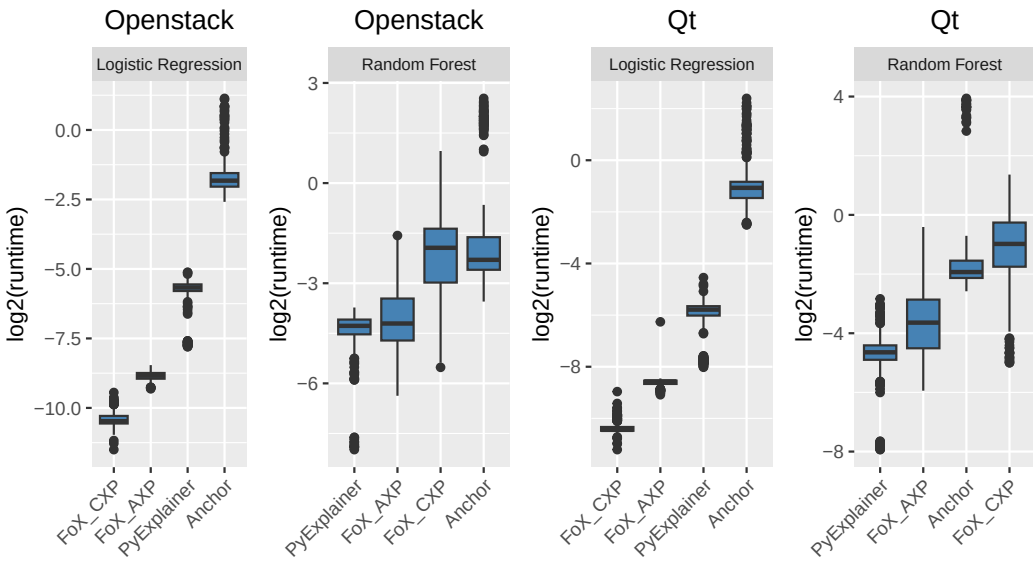

Fig. 8. (RQ3) Runtime of feature selection methods. (↘) Lower Runtime = Better.

other feature selection approaches that are only able to compute explanations answering *why* questions, FoX enumerates both AXps and CXps for a prediction. As a result, we analyze the time

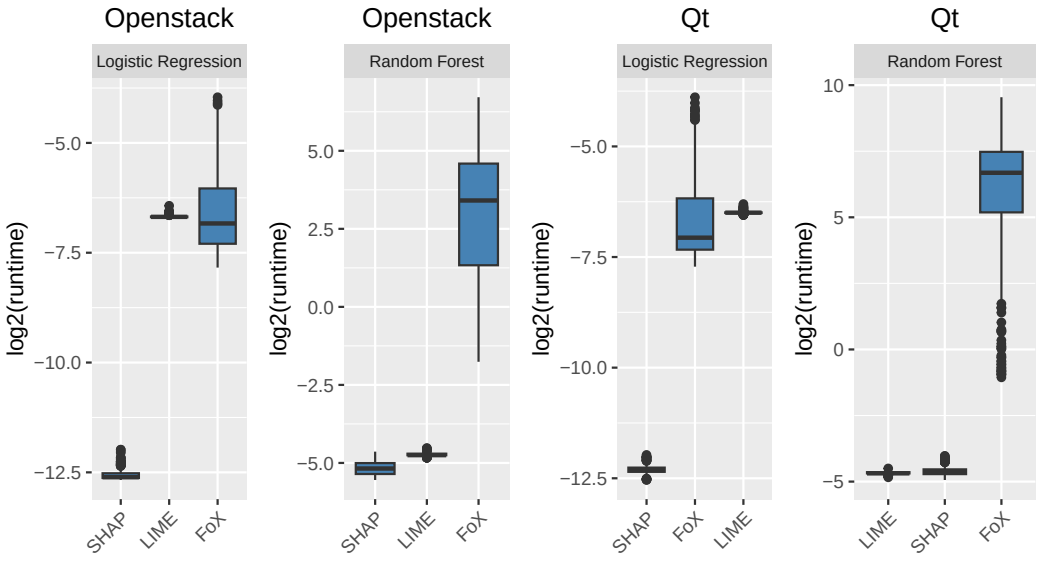

Fig. 9. (RQ3) Runtime of feature feature attribution methods. (↘) Lower Runtime = Better.

spent on generating an AXp and CXp separately represented by FoX_{AXp} and FoX_{CXp}, respectively. Moreover, we also evaluate the time taken by FoX to generate FFA.

Result. **FoX takes less than 0.003 seconds and less than 0.835 seconds to generate a feature selection based explanation for LRs and RFs on average, respectively, while 0.013s and 30.347s are required to generate an FFA for these two models.** Figure 8 depicts the runtime of computing an explanation for two models and both studied datasets in feature selection approaches. For LR models, FoX demonstrates high efficiency with the median time for generating a single AXp being 0.0022 seconds for Openstack and 0.0026 seconds for Qt. Also, FoX takes 0.0008 seconds to produce a CXp for most instances in both datasets. The median runtime of producing an explanation for Openstack by PyExplainer and Anchor is 0.0197 and 0.2825 seconds, whereas the runtime is 0.0181 and 0.4764 seconds for Qt. For RF models, PyExplainer spends 0.0516 seconds to generate an explanation for most instances to be explained in Openstack, while the runtime drops to 0.0400 seconds for Qt. The median runtime of computing an AXp by FoX is 0.0542 seconds for Openstack and 0.0801 seconds for Qt, while CXp extraction takes 0.2609 and 0.5066 seconds for these datasets, respectively. Observe that the runtime increase exhibited by FoX for RF models is caused by the need to represent the models logically and make a large series of SAT oracle calls, as described in Section 4.2. Anchor takes 0.2033 and 0.2620 seconds to generate an explanation for most instances in Openstack and Qt.

The runtime of generating a feature attribution based explanation for selected models and datasets is illustrated in Figure 9. For LR models, FoX remains high efficiency, with the median time of 0.0088 seconds to generate a FFA for an instance in Openstack and 0.0075 seconds for Qt. The median runtime required to generate a feature attribution by LIME and SHAP is 0.0104 and 0.0002 seconds, respectively, for Openstack, and 0.0113 and 0.0002 seconds, respectively, for Qt. For RF models, FoX takes a median time of 10.6197 seconds to generate FFA for an instance in the Openstack dataset and 102.8650 seconds in the Qt dataset. In contrast, the median runtime

for generating a feature attribution in the Openstack dataset using LIME and SHAP is 0.0362 and 0.0276 seconds, respectively. In the Qt dataset, the corresponding runtime is 0.0374 seconds for LIME and 0.0403 seconds for SHAP.

7 A USER STUDY OF EXPLAINABLE JUST-IN-TIME DEFECT PREDICTION

The quantitative evaluation of FoX is demonstrated in Section 6. However, it is essential to delve into the practical utility of explainable Just-In-Time (JIT) defect prediction for software practitioners, a vital aspect yet to be explored. To bridge this gap in knowledge, we conducted a user study to assess software developers' perceptions regarding the usefulness and trustworthiness of JIT defect prediction with and without explanations. This investigation seeks to unveil the practicality and real-world implications of our approach.

Following the guidelines provided by Kitchenham and Pfleeger [30], we conducted our study according to the following steps: (1) design and develop a survey, (2) recruit and select participants, and (3) verify data and analyze data. We explain the details of each step below.

7.1 Survey Design

Step 1 – Design and development of the survey: We designed our survey as a cross-sectional study where participants provided their responses at one fixed point in time. The survey consists of 7 closed-ended questions and 5 open-ended questions. We use multiple-choice questions and a Likert scale from 1 to 5 for closed-ended questions. Our survey consists of two parts: preliminary questions and participants' perceptions of AI-generated software vulnerability repairs.

Part I: Demographics. The survey commences with a query, "(D1) What is your role in your organization?", to ensure that our survey captures responses from the intended target participants. Subsequently, a demographics question, "(D2) What is the level of your professional experience?", is presented to ensure a diverse distribution of responses across software practitioners with varying degrees of professional experience.

Part II-A: JIT Defect Prediction Without Explanations. To simulate a realistic code review scenario, we asked participants to imagine themselves as software developers immersed in the continuous task of reviewing numerous commits and pull requests. In the dynamic world of software development, introducing new code can inadvertently lead to software bugs and errors. This is where AI-based defect prediction comes into play. The AI-based defect prediction is designed to predict potentially defective commits, providing a proactive approach to prioritize code reviews efficiently. Specifically, we told the participants that the defect prediction model will provide a warning for a defective commit as presented in the upper part of Figure 10.

We then asked the participants to assess the usefulness and trustworthiness of the defect prediction, along with providing reasons for their assessments. In particular, four inquiries were presented to the participants, commencing with "(Q1.1) How do you perceive the usefulness of the AI-based defect prediction?"; followed by "(Q1.2) Please justify your answer to Q1.1."; then "(Q1.3) Do you trust the AI-based defect prediction?"; and concluded with "(Q1.4) Please justify your answer to Q1.3.".

Part II-B: JIT Defect Prediction With Explanations. In this part of the survey, we asked participants to imagine that they were in the same situation as in the previous Part II-A. However, this time, the AI-based defect prediction provides both predictions and explanations of why the model generated such a prediction along with actionable guidance to help you mitigate the defective commit as presented in the lower part of Figure 10.

We then asked the participants to assess the usefulness and trustworthiness of the defect prediction with explanations and actionable guidance, while also providing reasons for their assessments. In particular, four inquiries were presented to the participants. These began with "(Q2.1) How

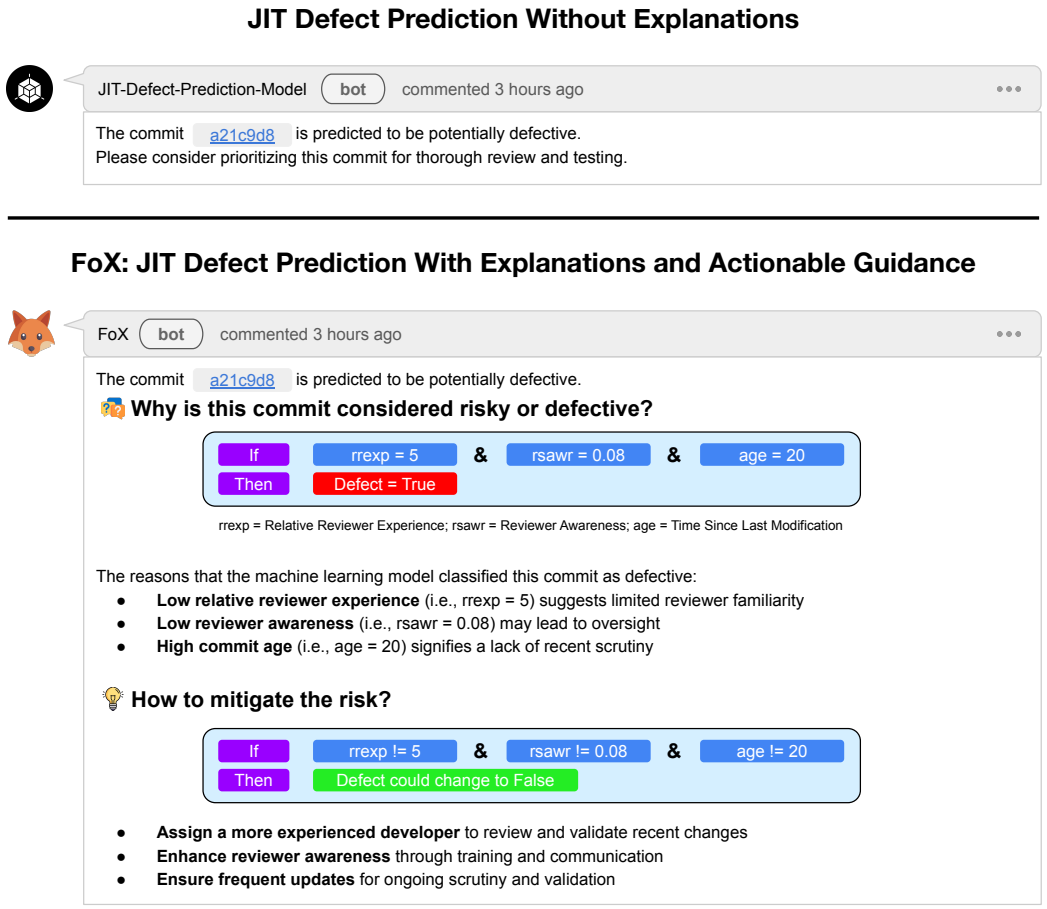

Fig. 10. The upper part shows the Just-In-Time (JIT) defect prediction without any explanations, as presented in Part II-A of our user study. The lower part shows FoX, which presents the JIT defect prediction with explanations and actionable guidance, as presented in Part II-B of our user study.

do you perceive the usefulness of the AI-based defect prediction with "Explanations (WHY)" and "Actionable Guidance (HOW)"?"; followed by "(Q2.2) Please justify your answer to Q2.1."; then "(Q2.3) Do you trust the AI-based defect prediction with "Explanations (WHY)" and "Actionable Guidance (HOW)"?"; then "(Q2.4) Please justify your answer to Q2.3."; then "(Q2.5) Would you consider adopting AI-based defect prediction with explanations if they are integrated into your CI/CD pipeline (e.g., a GitHub action) for free with no conditions?"; and concluded with "(Q2.6) What is your expectation of explainable defect prediction and how can we improve them?"

We employed Google Forms as the platform for our online survey. Upon accessing the landing page, each participant was welcomed with a detailed introductory statement. This statement clarified the study's objectives, the reasoning behind participant selection, potential benefits and risks, and our dedication to maintaining confidentiality. The survey was intentionally brief, taking approximately 15 minutes to complete, and guaranteed complete anonymity for all participants. It is worth noting that our survey underwent a thorough evaluation process and obtained ethical approval from the Monash University Human Research Ethics Committee (MUHREC ID: 41735).

Step 2: Recruit and select participants: We reached out to practitioners with software engineering-related experience through LinkedIn and Facebook platforms. Specifically, we advertised our survey by posting on LinkedIn and Facebook, providing accessible links. Additionally, we reached out directly to target groups via direct messages. Ultimately, we received a total of 54 responses over a two-week recruitment period.
Step 3: Verify data and analyze data: To ensure the integrity of our survey responses, especially concerning open-ended questions, we carried out a comprehensive manual review. We identified and excluded 0 invalid responses, such as those with unanswered open-ended questions or incomprehensible responses, from the total of 54 received. Consequently, we included all 54 responses for analysis. Closed-ended responses underwent quantitative analysis and were depicted using Likert scales in stacked bar plots. Furthermore, we conducted a detailed manual examination of responses to open-ended questions to gain deeper insights into participants' perspectives.

7.2 Survey Results

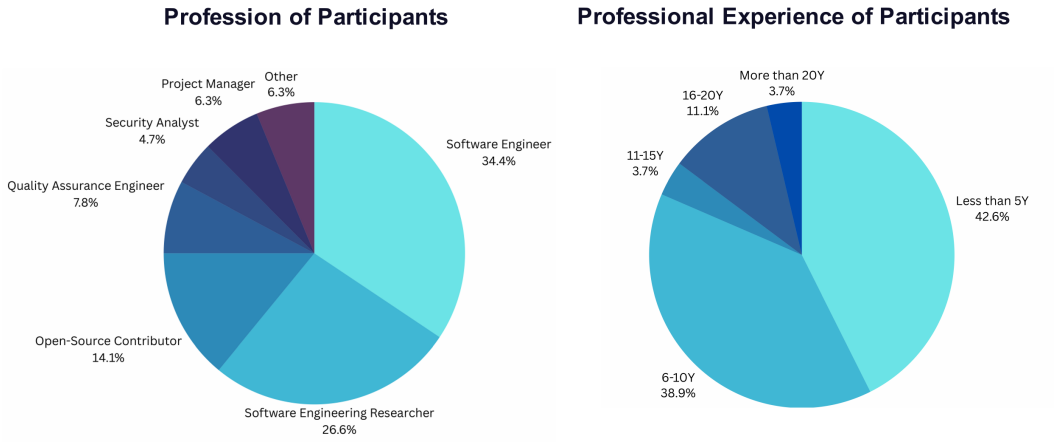

Fig. 11. The demographics of our survey participants in terms of their profession and professional experience.

Part I: Demographics. Figure 11 presents the overall respondent demographic. In terms of the profession of the participants, 34% ($\frac{19}{54}$) of them are software engineers, 27% ($\frac{15}{54}$) of them are software engineering researchers, 14% ($\frac{7}{54}$) of them are open-source contributors, while the other 25% ($\frac{13}{54}$) are software quality assurance engineers, software project managers, and software security analysts. In terms of the level of their professional experience, 42% ($\frac{23}{54}$) of them have less than 5 years of experience, 39% ($\frac{21}{54}$) have 6-10 years of experience, 15% ($\frac{8}{54}$) have 11-20 years of experience, while the other 4% ($\frac{2}{54}$) has more than 20 years of experience.
Part II-A: JIT Defect Prediction Without Explanations. Figure 12 summarizes the answers to (Q1.1)-(Q1.4) regarding participants' perception of the JIT defect prediction without explanations. **As presented in (Q1.1) and (Q1.3) results, 72% ($\frac{39}{54}$) of participants perceived the defect prediction presented in the upper part of Figure 10 as useful, while only 53% ($\frac{29}{54}$) of participants trusted the prediction.**

Some participants expressed strong support for the JIT defect prediction. For example, a software engineering (SE) researcher with 6-10 years of experience stated, *"A good code defect detection tool can help developers or users avoid risks. For instance, unauthorized individuals may exploit the loopholes created by code defects to engage in illicit activities. Particularly in commercial applications,*

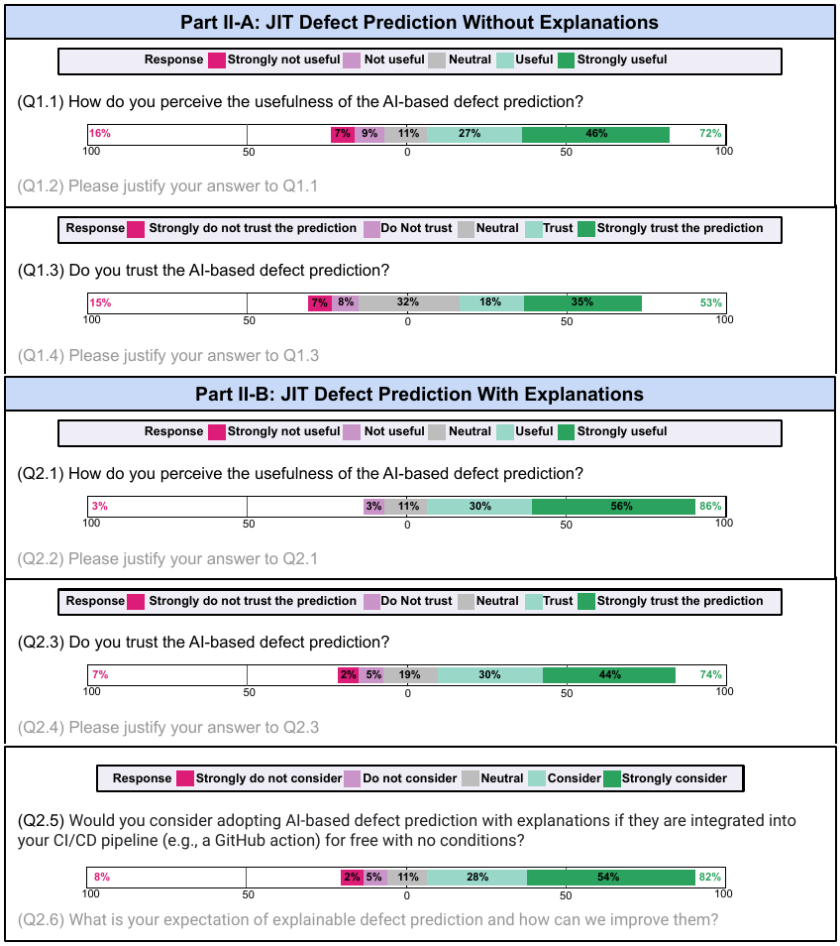

Fig. 12. (Survey Results) A summary of the survey questions (i.e., Part II-A: Q1.1-Q1.4 and Part II-B: Q2.1-2.6) and the results obtained from 54 participants.

a reliable code defect detection model can not only reduce a significant amount of ineffective human effort but also ensure the security of the application." Another SE researcher with less than 5 years of experience mentioned, *"With the development of AI technology, the accuracy of AI-based code review is getting higher and higher. Although it is not 100% reliable, it can still detect some potential bugs."*

However, some participants raised concerns about the explainability and transparency of the defect prediction model, which are important factors in building trust between developers and end users. A SE researcher with less than 5 years of experience noted, *"Without an explanation of the prediction, it cannot reduce the time to review the commit."* Another software testing engineer with 6-10 years of experience expressed, *"It's a black box. We can't know the reason behind the prediction. Hard to trust the prediction."* Additionally, an experienced software security analyst with 16-20 years of experience commented, *"Only prediction seems suspicious, I don't think this would help automate my workflow".*

Overall, most participants found the JIT defect prediction to be valuable as it can streamline human effort and identify potential bugs. However, only half expressed trust in the prediction, citing

concerns about the opaque nature of machine learning models and the absence of explanations. This underscores the practical necessity of our proposed explainable JIT defect prediction approach.
Part II-B: JIT Defect Prediction With Explanations. Figure 12 provides an overview of participants' perceptions of our JIT defect prediction with explanations, covering questions (Q2.1)-(Q2.6). **As illustrated in the findings from (Q2.1) and (Q2.3), 86% ($\frac{46}{54}$) of participants regarded the defect prediction depicted in the lower portion of Figure 10 as useful. This marks a 14% increase compared to the defect prediction without explanations. Furthermore, 74% ($\frac{40}{54}$) of participants expressed trust in the prediction, representing a notable 21% increase compared to the scenario without explanations.** These results underscore the enhanced usefulness and trustworthiness of our approach among software practitioners. Last but not least, 82% (($\frac{44}{54}$)) of participants indicated their willingness to adopt FoX if integrated into the CI/CD pipeline, such as GitHub action. This result suggests a strong inclination towards the adoption of our approach within software development workflows.

Specifically, the participants perceived that FoX is useful and trustworthy due to various reasons stated in (Q2.2) and (Q2.4):

- **Time & Efficiency** – R7 (an open-source contributor with less than 5 years of experience): *AI-based defect prediction and prevention leverage advanced algorithms to proactively identify potential software defects, enabling organizations to save time, reduce costs, and deliver higher-quality software*; R18 (a SE researcher with 6-10 years of experience): *Save my time to double check and provide me with how*; R45 (a full-stack software engineer with 6-10 years of experience): *Increased accuracy and efficiency: AI can analyze vast amounts of data to identify patterns and anomalies that humans might miss, potentially leading to more accurate defect prediction. This can save time and resources by identifying potential issues early in the development process.*
- **Insight & Understanding** – R19 (a full-stack software engineer with less than 5 years of experience): *This gives a more detailed explanation of the detection, and a brief suggestion on how the risk can be mitigated*; R25 (a software testing engineer with 6-10 years of experience): *Understand why we get this prediction. Explanations are provided*; R32 (a SE researcher with less than 5 years of experience): *The information provided is more than just prediction*; R44 (a full-stack software engineer with 6-10 years of experience): *Providing explanations for why a certain commit or piece of code is flagged as potentially defective helps developers understand the underlying reasons behind the prediction*; R46 (a full-stack software engineer with 6-10 years of experience): *Detecting and predicting a defect is not enough, the explanation gives the reason behind the defect and gives possible solutions to mitigate the defect and give guidance to writing a more effective code.*
- **Trust & Confidence** – R15 (a SE researcher with less than 5 years of experience): *As depicted in Q2.2., providing explanations makes me trust the prediction*; R17 (a SE researcher with less than 5 years of experience): *With the explanation, the prediction of AI system becomes more trustable*; R43 (a software security analyst with 16-20 years of experience): *Given that the information is transparent, I would trust this prediction more than the one provided in the previous section*; R54 (a full-stack software engineer with 6-10 years of experience): *Predicting a defect is only one end of the game and is incomplete without explanations or actionable guidance. The explanation and actionable guidance offers solutions to the defects predicted, why they occur and how it can be sorted out thereby rendering additional boost to the confidence in the prediction.*
- **Practicality & Guidance** – R17 (a SE researcher with less than 5 years of experience): *The explanation is clear and the guidance looks convincing*; R32 (a SE researcher with less than 5

years of experience): *It's practical to have explanations and guidance along with the predictions*; R44 (a full-stack software engineer with 6-10 years of experience): *Providing explanations for why a certain commit or piece of code is flagged as potentially defective helps developers understand the underlying reasons behind the prediction. In addition to explanations, actionable guidance offers concrete suggestions or recommendations on how to address potential defects identified by the AI-based system. This could include specific code changes, best practices, or alternative approaches to mitigate the risk of introducing bugs*; R47 (a full-stack software engineer with 6-10 years of experience): *The reason why I strongly trust the prediction is that it guides the action of the masses.*

Furthermore, the participants' expectations regarding explainable defect prediction, as indicated in their responses to (Q2.6), can be summarized as follows:

- **Role-Specific Defect Prediction** – R11 (a full-stack software engineer with 6-10 years of experience): *We might consider that different roles in the IT team might need different information for defect prediction (developer vs tester vs auditor).*
- **Privacy** – R19 (a full-stack software engineer with less than 5 years of experience): *Reliability and explainability. Privacy issues.*
- **Guiding Corrections** – R25 (a software testing engineer with 6-10 years of experience): *We need to understand why we get the prediction and get hints to change the prediction to the desired one, i.e. non-defective.*
- **More Accurate Models** – R28 (a SE researcher with less than 5 years of experience): *I would expect this system would be able to at least accurately predict 70% to 80% of defects.*
- **User Feedback Mechanisms** – R44 (a full-stack software engineer with 6-10 years of experience): *Implementing feedback mechanisms that enable developers to provide input on the relevance and usefulness of explanations. Gathering feedback from users helps improve the quality and effectiveness of explanations over time by addressing common misunderstandings or areas of confusion.*

Summary. Our survey study with 54 software practitioners provides valuable insights into the usefulness and trustworthiness of our proposed explainable JIT defect prediction. In the scenario where only the defect prediction was presented, 72% ($\frac{39}{54}$) of participants deemed JIT defect prediction useful, even without explanations. However, only 53% ($\frac{29}{54}$) of participants trusted the prediction. This discrepancy underscores the pressing need for our proposed explainable JIT defect prediction approach to bridge the trust gap between defect prediction and end users. In contrast, in the scenario where our defect prediction with explanations and actionable guidance was presented, 86% ($\frac{46}{54}$) of participants found our explainable defect prediction useful. Furthermore, 74% ($\frac{40}{54}$) of participants trusted our explainable defect prediction. They attributed their value to increased efficiency, prediction understanding, trust, and practicality of the actionable guidance. In addition, participants voiced expectations for enhancements, emphasizing the importance of improving model accuracy, integrating a feedback loop to incorporate input from human experts, and tailoring the defect prediction model to users' roles and requirements. Our findings highlight the potential and significance of explainable JIT defect prediction, while also pinpointing areas that require further development and refinement.

8 RELATED WORK

Explainable AI in SE. The explainability of AI models in SE is increasingly important, since prior studies point out that practitioners often do not understand the reasons behind the predictions of software analytics [12, 25, 34, 58]. Such a lack of trust in the predictions hinders the adoption of AI-powered software development tools in practice. Thus, Explainable AI for SE (XAI4SE) is a

newly emerging research topic which aims to increase the explainability and actionability of AI models in SE. Various Explainable AI approaches have been explored in software engineering.

In stark contrast to prior work on explainable AI for SE and to the best of our knowledge, this paper is the first to leverage the formal approach to XAI, a quickly developing area of research that makes a bridge between explainable AI and formal reasoning applied to ML models by exploiting propositional and first-order logic [13, 21, 22, 37, 50]. This way, our paper serves as an example of the synergy between software engineering and formal methods, similar to what has been observed in the recent past in the area of formal software verification [5, 9, 14, 23, 31].

Explainable AI for Defect Prediction. Recent works have shown some successful use cases to make defect prediction models more practical [25, 43, 61], explainable [24, 28], and actionable [45]. However, some of these studies only apply existing model-agnostic techniques from the Explainable AI domain. Recently, Pornprasit *et al.* [44] proposed PyExplainer, a rule-based model-agnostic technique for Just-In-Time defect prediction. However, there exist many limitations (e.g., correctness, robustness, actionability) that have not been addressed.

Different from prior studies of explainable AI for defect prediction, to the best of our knowledge, this paper is the first to address the key limitations of explainability techniques (e.g., PyExplainer, LIME, SHAP, etc) for Just-In-Time defect prediction, demonstrating the significant advancement of explainable AI for defect prediction.

Existing Delta-Debugging Techniques By integrating existing delta-debugging techniques [8, 15, 60] proposed for practical application in CI/CD pipelines, software development teams could significantly enhance the regression fault localization and resolution processes. Chen *et al.* [8] emphasized understanding and predicting performance regressions in code commits and proposed an approach to recovering field-representative workload that can be used to detect performance regression. Wang *et al.* [60] introduced a trace-based approach for identifying regression faults and tracing their propagation. This approach further involves constructing a causality graph, known as an explanation, by employing a technique termed alignment slicing and mending. This method aims to isolate the changes that induce failures and provide a clear explanation of the failure. Hassan *et al.* [15] presented UniLoc, a unified fault localization technique for both source code and build scripts. UniLoc further refines fault localization using an information retrieval (IR) strategy, enabling more precise analysis of detected failures and ultimately streamlining the diagnosis and resolution of issues in CI/CD pipelines. The integration of these techniques into CI/CD pipelines shows potential for enhancing software reliability and efficiency across the entire development lifecycle.

9 LIMITATIONS AND FUTURE WORK

Though experimental results demonstrate that FoX can efficiently compute provably-correct, robust, and actionable explanations, outperforming the four compared state-of-the-art model-agnostic approaches, some limitations may constrict the application of FoX.

Potential Overheads When Enumerating All Explanations. While the features of an AXp returned by FoX are guaranteed to contribute to the prediction, the explanation may not contain all the important features. In a scenario where users would like to identify all important features using FoX, they can achieve this by enumerating all explanations. However, enumerating all explanations to identify all important features could be limited by several factors. Firstly, it may incur a substantial computational cost and demand significant computational resources, particularly for complex models or datasets with numerous features. This approach could also pose challenges in interpretability, as the abundance of explanations may overwhelm users, especially those lacking expertise in both the domain and the model. Additionally, there is a trade-off between coverage and precision, potentially leading to difficulties in distinguishing truly important features from less

relevant ones. These limitations underscore the need for users to carefully consider the overhead associated with enumerating all explanations when seeking to identify all important features.

Application to other classifiers. Conceptually, FoX can apply to all kinds of classifiers [21, 38]. Nothing prevents one to apply this technology to any classifier that admits a logical representation in some decidable fragment of first-order logic. In practice, however, some classifiers are hard to reason about formally, e.g. deep neural networks, and so it may be computationally expensive to generate provably correct explanations. In this sense, the success and future applicability of formal explainability depends on the future advances in the underlying formal reasoning technology (SAT, SMT, MILP, CP, etc.). In recent JIT defect prediction studies, language models (e.g., BERT) have been employed to directly extract characteristics of defective entities, showcasing promising performance compared to traditional machine learning models. With the rising adoption of language models in JIT defect prediction, it is crucial to explore efficient methods to produce formal explanations tailored for these models. In the future, this investigation will be one of our research directions, aimed at enhancing the practicability of our proposed approach for the advanced JIT defect prediction models. By improving the applicability of our method, practitioners can leverage it effectively to produce correct and robust explanations for language models.

Scalability. Related to the above, although FoX has been empirically demonstrated to be computationally efficient, it can suffer from some scalability problems if a target RF model is extremely complex and its propositional encoding is accordingly large. This is because SAT is a well-known example of an NP-complete problem [10], and the search space a SAT solver has to deal with is determined by the number of propositional variables in the target formula, which can be further affected by the number of clauses. Furthermore, exact FFA computation requires one to enumerate *all* formal explanations, which is computationally even more challenging, but Yu *et al* [65] show that we can *approximate* FFA very closely (i.e. much closer than the results for alternate methods in Table 3) using Algorithm 1 with a small cutoff time. Hence the disadvantages in feature attribution runtime shown in Figure 9 can easily be ameliorated. Note that while extracting a single formal explanation for LR models can be done in polynomial time [37], exact FFA computation is still quite competitive due to the need to enumerate all such explanations. In the future, we will aim at addressing the scalability problems associated with RF models by proposing alternative (simpler) RF encodings into propositional logic as well as investigating alternative ways for effective FFA approximation.

Explanation Ordering. Users may opt to select only the first generated explanation even though FoX can generate a pool of candidate explanations. Although the generic explanation enumeration approach applied in FoX, supports generating explanations with the preference of small size, the bespoke alternative for the case of monotonic classifiers (see Section 4.1) is unable to do so. Nevertheless, one may still apply the same generic approach to monotonic classifiers as well (although the performance of explanation enumeration may be slightly affected). Thus, our future work will also include the extension of explanation size ordering for monotonic classifiers and the support for more feature orderings in FoX.

Actionability of Explanations. In this article, our focus is on proposing, evaluating, and implementing a more robust formal technique for explaining JIT defect predictions. Additionally, we conduct a user study to assess the qualitative aspects of our proposed approach. While recognizing the importance of clarifying the actionability of these explanations—specifically, how helpful feature-level JIT explanations are to developers—we acknowledge that this depends on how metrics are operationalized, which falls outside the scope of our current work. Thus, in future research, we plan to conduct a user study involving debugging tasks within a CI/CD pipeline. This study aims to evaluate the extent to which metrics provided by JIT explanations aid developers in saving

time and effort, with a focus on recording human behaviors to understand their responses to the reported explanations more comprehensively.

10 CONCLUSION

In this paper, we propose FoX, the first formal explainer for Just-In-Time defect prediction. The FoX explainer builds on the use of formal reasoning by exploiting propositional logic in order to efficiently generate provably-correct, robust, and subset-minimal explanations answering the *why?* questions, e.g. why a commit is predicted as defective. Our formal explainer can also generate provably-correct, robust, and subset-minimal contrastive *how?* explanations, e.g. how should developers mitigate the risk, which are more actionable than the usual abductive explanations, serving as an important step towards actionable software analytics. As a result, the generated explanations are expected to help researchers better design actionable defect prediction approaches and help practitioners better focus on the most important aspects associated with software defects to improve the operational decisions of SQA teams.

REFERENCES

[1] Amritanshu Agrawal and Tim Menzies. 2018. Is Better Data Better Than Better Data Miners?: On the Benefits of Tuning SMOTE for Defect Prediction. In *ICSE*. 1050–1061.

[2] Reem Aleithan. 2021. Explainable just-in-time bug prediction: are we there yet?. In *ICSE-Companion*. 129–131.

[3] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2021. *Handbook of Satisfiability*. IOS Press.

[4] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32.

[5] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[6] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research* (2002), 321–357.

[7] Di Chen, Wei Fu, Rahul Krishna, and Tim Menzies. 2018. Applications of Psychological Science for Actionable Analytics. In *ESEC/FSE*. 456–467.

[8] Jinfu Chen. 2020. Performance regression detection in devops. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 206–209.

[9] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. 2018. *Model checking*.

[10] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *STOC*. ACM, 151–158.

[11] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-introducing Changes. *IEEE Transactions on Software Engineering (TSE)* 43, 7 (2017), 641–657.

[12] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2018. Explainable Software Analytics. In *ICSE-NIER*. 53–56.

[13] Adnan Darwiche and Auguste Hirth. 2020. On the Reasons Behind Decisions. In *ECAI*. 712–720.

[14] Vijay Victor D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 27, 7 (2008), 1165–1178.

[15] Foyzul Hassan, Na Meng, and Xiaoyin Wang. 2023. UniLoc: Unified Fault Localization of Continuous Integration Failures. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–31.

[16] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *MSR*. 34–45.

[17] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *ICSE*. 518–529.

[18] Xuanxiang Huang and Joao Marques-Silva. 2023. The Inadequacy of Shapley Values for Explainability. *CoRR* abs/2302.08160 (2023).

[19] Alexey Ignatiev. 2020. Towards Trustable Explainable AI. In *IJCAI*. 5154–5158.

[20] Alexey Ignatiev, Nina Narodytska, Nicholas Asher, and Joao Marques-Silva. 2020. From Contrastive to Abductive Explanations and Back Again. In *AI*IA*. 335–355.

[21] Alexey Ignatiev, Nina Narodytska, and Joao Marques-Silva. 2019. Abduction-Based Explanations for Machine Learning Models. In *AAAI*. 1511–1519.

[22] Yacine Izza and João Marques-Silva. 2021. On Explaining Random Forests with SAT. In *IJCAI*. 2584–2591.

[23] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Comput. Surv.* 41, 4 (2009), 21:1–21:54.

[24] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Khanh Dam, and John Grundy. 2020. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering (TSE)* (2020), 166–185.
[25] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John Grundy. 2021. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *MSR*. 432–443.
[26] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-In-Time Quality Assurance. *IEEE Transactions on Software Engineering (TSE)* 39, 6 (2013), 757–773.
[27] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.
[28] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. 2020. JITBot: An Explainable Just-In-Time Defect Prediction Bot. In *ASE*. 1336–1339.
[29] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting Faults from Cached History. In *ICSE*. 489–498.
[30] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*. Springer, 63–92.
[31] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. 2013. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*. 846–862.
[32] Rahul Krishna and Tim Menzies. 2020. Learning Actionable Analytics from Multiple Software Projects. *Empirical Software Engineering (EMSE)* (2020), 3468–3500.
[33] Himabindu Lakkaraju and Osbert Bastani. 2020. "How do I fool you?": Manipulating User Trust via Misleading Black Box Explanations. In *AIES*. 79–85.
[34] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. 2013. Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *ICSE*. 372–381.
[35] Dayi Lin, Chakkrit Tantithamthavorn, and Ahmed E Hassan. 2021. The Impact of Data Merging on the Interpretation of Cross-Project Just-In-Time Defect Models. *IEEE Transactions on Software Engineering* (2021).
[36] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS*. 4765–4774.
[37] João Marques-Silva, Thomas Gerspacher, Martin C. Cooper, Alexey Ignatiev, and Nina Narodytska. 2021. Explanations for Monotonic Classifiers. In *ICML*. 7469–7479.
[38] Joao Marques-Silva and Alexey Ignatiev. 2022. Delivering Trustworthy AI through Formal XAI. In *AAAI*. 12342–12350.
[39] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering (TSE)* (2017), 412–428.
[40] Tim Miller. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artif. Intell.* 267 (2019), 1–38.
[41] Nina Narodytska, Aditya A. Shrotri, Kuldeep S. Meel, Alexey Ignatiev, and Joao Marques-Silva. 2019. Assessing Heuristic Machine Learning Explanations with Model Counting. In *SAT*. 267–278.
[42] Kewen Peng and Tim Menzies. 2021. Defect Reduction Planning (using TimeLIME). *IEEE Transactions on Software Engineering (TSE)* (2021).
[43] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In *MSR*. 369–379.
[44] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. 2021. PyExplainer: Explaining the Predictions of Just-In-Time Defect Models. In *ASE*. 407–418.
[45] Dilini Rajapaksha, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Christoph Bergmeir, John Grundy, and Wray Buntine. 2021. SQAPlanner: Generating Data-Informed Software Quality Improvement Plans. *IEEE Transactions on Software Engineering (TSE)* (2021).
[46] Raymond Reiter. 1987. A Theory of Diagnosis from First Principles. *Artif. Intell.* 32, 1 (1987), 57–95.
[47] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why should I trust you?: Explaining the Predictions of Any Classifier. In *KDD*. 1135–1144.
[48] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-Precision Model-Agnostic Explanations. In *AAAI*. 1527–1535.
[49] LLoyd S. Shapley. 1953. A Value of n-Person Games. *Contributions to the Theory of Games* 2, 28 (1953), 307–317.
[50] Andy Shih, Arthur Choi, and Adnan Darwiche. 2018. A Symbolic Approach to Explaining Bayesian Network Classifiers. In *IJCAI*. 5103–5111.
[51] Jiho Shin, Reem Aleithan, Jaechang Nam, Junjie Wang, and Song Wang. 2021. Explainable Software Defect Prediction: Are We There Yet? *arXiv preprint arXiv:2111.10901* (2021).
[52] Dylan Slack, Sophie Hilgard, Emily Jia, Sameer Singh, and Himabindu Lakkaraju. 2020. Fooling LIME and SHAP: Adversarial Attacks on Post hoc Explanation Methods. In *AIES*. 180–186.

[53] Dylan Z Slack, Sophie Hilgard, Sameer Singh, and Himabindu Lakkaraju. 2021. Reliable Post hoc Explanations: Modeling Uncertainty in Explainability. In *NeurIPS*.
[54] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *MSR*. 1–5.
[55] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2020. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering (TSE)* 46, 11 (2020), 1200–1219.
[56] Chakkrit Tantithamthavorn and Jirayus Jiarpakdee. 2021. Explainable AI for Software Engineering. In *ASE*. 1–2.
[57] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. 2020. Explainable AI for Software Engineering. *arXiv preprint arXiv:2012.01614* (2020).
[58] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. 2021. Actionable Analytics: Stop Telling Me What It Is; Please Tell Me What To Do. *IEEE Software* 38, 4 (2021), 115–120.
[59] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering (TSE)* 43, 1 (2017), 1–18.
[60] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jinsong Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2421–2437.
[61] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting Defective Lines Using a Model-Agnostic Technique. *IEEE Transactions on Software Engineering (TSE)* (2020).
[62] William Webber, Alistair Moffat, and Justin Zobel. 2010. A similarity measure for indefinite rankings. *ACM Transactions on Information Systems (TOIS)* 28, 4 (2010), 1–38.
[63] Ye Yang, Davide Falessi, Tim Menzies, and Jairus Hihn. 2017. Actionable analytics for software engineering. *IEEE Software* (2017), 51–53.
[64] Suraj Yathish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining Software Defects: Should We Consider Affected Releases?. In *ICSE*. 654–665.
[65] Jinqiang Yu, Alexey Ignatiev, and Peter J. Stuckey. 2023. On Formal Feature Attribution and Its Approximation. *CoRR* abs/2307.03380 (2023). https://doi.org/10.48550/arXiv.2307.03380 arXiv:2307.03380