PyExplainer: Explaining the Predictions of Just-In-Time Defect Models

Chanathip Pornprasit*, Chakkrit Tantithamthavorn*, Jirayus Jiarpakdee*, Micheal Fu*, Patanamon Thongtanunam†
*Monash University, Australia. †The University of Melbourne, Australia.

***Abstract*—Just-In-Time (JIT) defect prediction (i.e., an AI/ML model to predict defect-introducing commits) is proposed to help developers prioritize their limited Software Quality Assurance (SQA) resources on the most risky commits. However, the explainability of JIT defect models remains largely unexplored (i.e., practitioners still do not know why a commit is predicted as defect-introducing). Recently, LIME has been used to generate explanations for any AI/ML models. However, the random perturbation approach used by LIME to generate synthetic neighbors is still suboptimal, i.e., generating synthetic neighbors that may not be similar to an instance to be explained, producing low accuracy of the local models, leading to inaccurate explanations for just-in-time defect models. In this paper, we propose PyExplainer—i.e., a local rule-based model-agnostic technique for generating explanations (i.e., why a commit is predicted as defective) of JIT defect models. Through a case study of two open-source software projects, we find that our PyExplainer produces (1) synthetic neighbous that are 41%-45% more similar to an instance to be explained; (2) 18%-38% more accurate local models; and (3) explanations that are 69%-98% more unique and 17%-54% more consistent with the actual characteristics of defect-introducing commits in the future than LIME (a state-of-the-art model-agnostic technique). This could help practitioners focus on the most important aspects of the commits to mitigate the risk of being defect-introducing. Thus, the contributions of this paper build an important step towards Explainable AI for Software Engineering, making software analytics more explainable and actionable. Finally, we publish our PyExplainer as a Python package to support practitioners and researchers (http://github.com/awsm-research/PyExplainer).**

***Index Terms*—Explainable AI, Just-In-Time Defect Prediction, Explainable Software Analytics**

I. Introduction

Modern software development projects tend to release software products in rapid cycles. To ensure the quality of all newly arrived commits, developers need to conduct code review and provide feedback prior to merging them into the release branch. However, such code review activities are still time-consuming and expensive. Thus, performing exhaustive code review activities for all commits is infeasible due to the limited Software Quality Assurance (SQA) resources.

Just-In-Time (JIT) defect prediction [17, 18, 29]—an AI/ML model to predict defect-introducing commits—has been proposed to help developers efficiently prioritize their limited SQA resources on the most risky commits. In addition, JIT defect prediction is also used to provide insights about the important characteristics of defect-introducing commits. Such insights can help QA teams and managers to develop proactive software quality improvement plans to prevent pitfalls in the past that lead to software defects in the future [25].

However, the predictions of existing JIT defect prediction approaches are still not explainable, hindering the adoption of JIT defect models in practice [5, 13]. Recent research shows that practitioners still asked many *why*-questions (e.g., why a commit is predicted as defective) [5, 13, 14], since current JIT defect prediction approaches are treated as black-box which only provide the predictions, not the explanations. Such a lack of explainability of JIT defect prediction approaches could lead to suboptimal software quality assurance practices and suboptimal operational decision-makings.

Recently, LIME—a state-of-the-art model-agnostic technique [31]—has been adopted in software engineering research (e.g., line-level just-in-time defect prediction [29], and explainable file-level defect prediction [13]). LIME is a technique that explains a prediction of AI/ML models (i.e., what are the features that influence a given prediction). Generally speaking, given an instance to be explained (e.g., a commit), LIME produces an explanation from a local model (F') that is trained using the randomly generated synthetic instances around the instance to be explained (X') (i.e., *synthetic neighbors*) and the predictions (Y') obtained from the global black-box model. This allows the local model to mimic the behavior of the underlying global black-box models.

The quality of explanation produced by LIME heavily relies on the neighborhood generation process [12]. Ideally, the neighborhood generation process should generate synthetic neighbors that are closely similar to the instance to be explained so that the local model can accurately approximate the prediction of the global models. In LIME, the random perturbation approach is used to generate synthetic neighbors. However, such a simple random perturbation approach may not be suitable for sparse and high dimensional data like JIT datasets [40]. It is possible that the random perturbation approach will generate synthetic neighbors that may not be similar to an instance to be explained, which will lead the local model to inaccurately approximate the predictions of the global model. Thus, these local models may not be effective in generating explanations (e.g., generic).

In this paper, we propose PyExplainer, a local rule-based model-agnostic technique for explaining the predictions of JIT defect prediction models. To produce a more accurate explanation for the prediction of JIT defect models, our PyExplainer generates synthetic neighbors based on the actual character-

istics of defect-introducing commits in the JIT dataset using the crossover and mutation operations. Instead of generating an explanation with a single rule feature with an importance score like LIME (e.g., the importance score of Churn>100 is 0.9), our PyExplainer generates an explanation that accounts for interactions between rule features (e.g., Churn>100 & #Reviewers<2 ⇒ Defect).

To evaluate our PyExplainer, we compare with LIME [31] in three dimensions: (1) *the similarity between synthetic neighbors and an instance to be explained*; (2) *the accuracy of the local models*; and (3) *the effectiveness in generating explanations*. Through a case study of 40,798 commits that span across two large-scale software systems (i.e., OpenStack and Qt), we address the following three research questions:

(RQ1) Does our PyExplainer produce better synthetic neighbors than LIME for JIT defect models?
The synthetic neighours produced by our PyExplainer are 41%-55% more similar to an instance to be explained than LIME, indicating that our PyExplainer produces synthetic neighbors that are more closely similar to an instance to be explained than LIME.

(RQ2) Does our PyExplainer produce higher accuracy of local models than LIME for JIT defect models?
When explaining the RF and LR JIT defect models, PyExplainer produces local models that are 18%-38% more accurate (AUC) than the local models produced by LIME, indicating that the PyExplainer produces local models that have a higher ability to discriminate the characteristics between defect and clean classes.

(RQ3) Is our PyExplainer more effective in generating explanations than LIME for JIT defect models?
The explanations generated by our PyExplainer are 69%-98% more unique (i.e., more specific to an instance to be explained) than LIME. On the other hand, the explanations generated by our PyExplainer are 17%-54% more consistent with the actual defect-introducing commits in the testing data than LIME.

Thus, the explanations generated by PyExplainer could help practitioners to focus on the most important aspects that are associated with the risk of being defect-introducing for a given prediction, instead of focusing on the less important aspects.

Contributions. The contributions of this paper are as follows:

- We propose PyExplainer, a local rule-based model-agnostic technique for explaining the predictions of JIT defect models.
- Our results show that PyExplainer produces (1) synthetic neighbours that are more similar to an instance to be explained; (2) more accurate local models; and (3) explanations that are more unique and more consistent with the actual characteristics of defect-introducing commits in the future than LIME.
- Finally, we developed a proof-of-concept of visual explanations and *what-if* visualizations, and published our PyExplainer as a python package.

II. Related Work & Research Questions

Prior studies pointed out that practitioners often do not understand the reasons behind the predictions of software analytics [5, 14, 24]. Recent work also raises concerns that a lack of explainability of software analytics often hinder the adoption of software analytics in practice [5, 13, 14]. Importantly, Jiarpakdee *et al.* [14] found that 91% of recent defect prediction studies often focus on improving the accuracy, while as few as 4% of recent defect prediction studies focus on making file-level defect prediction models more explainable. However, Jiarpakdee *et al.* [14] found that practitioners perceived that providing the explanations of defect prediction models are as important and useful as improving the accuracy of defect prediction models. Yet, the explainability of JIT defect models remains largely unexplored.

The explanability of software analytics can be achieved at the global and the local levels.

The *global* explanation can be produced using model-specific interpretation techniques that are built in the AI/ML models (e.g., ANOVA for regression analysis, variable importance analysis for random forest). This explanation helps researchers and software practitioners understand what important features that influence the predictions of the models. However, this global explanation is not specific to the prediction of each instance (e.g., a commit) in the testing or unseen data, since the global explanation is derived from the training dataset (i.e., commits in the past) [37]. Hence, the global explanation may not be accurate for a particular prediction.

On the other hand, the *local* explanation is produced for a particular prediction of an instance in the testing or unseen dataset, which allow practitioners better understand the reasons behind the predictions of the AI/ML models [13]. **LIME** [31] is a state-of-the-art model-agnostic technique which has been widely adopted to address various software engineering problems and other domains (5,000+ citations). For example, recent work [29, 37] employed LIME for line-level defect predictions (i.e., identifying defective lines that contain the risky code tokens explained by LIME). Jiarpakdee *et al.* [13] found that LIME [31] is effective in explaining the predictions of file-level defect prediction models (i.e., why a file is predicted as defective). However, LIME has the following limitations.

First, the LIME's neighborhood generation process is still suboptimal. The quality of an explanation for a prediction heavily relies on the quality of the generated synthetic neighbors around the instance to be explained [22]. Thus, if the neighbor generation process is suboptimal, the local model may fail to provide accurate insights about the logical reasoning of the global model. Jia *et al.* [12] found that the size of the neighbourhood has a large impact on the quality of the explanation. Krishnan *et al.* [21] found that when a model is learned from sparse and high dimensional data (e.g., just-in-time defect dataset [40]), the model is often underfitting, failing to capture the phenomenon of the data being trained. Thus, the neighbor generation process should ideally generate synthetic neighbors that are similar to the

instance to be explained. Therefore, we investigate whether our PyExplainer produce better synthetic neighbors (i.e., the synthetic neighbors that are more similar to an instance to be explained) than LIME for JIT defect models. We formulate the following RQ:

(RQ1) Does our PyExplainer produce better synthetic neighbors than LIME for JIT defect models?

Second, the approximation of the LIME local models to the predictions of the global model is still suboptimal. One of the key principles of model-agnostic techniques is to build the best local model to mimic the behavior of the predictions of the global models. Accuracy is often used to measure the extent to which how well the local model can approximate the predictions of the global model [31]. Thus, a high accuracy of local models is desirable in order to derive the highest quality explanation, i.e., the local models can accurately approximate the global model predictions for a subset of the data (e.g., local surrogate models). Since LIME uses the random perturbation method to generate synthetic neighbors, the approximation of the LIME local models to the predictions of the global model may be suboptimal, leading to the inaccurate local models produced by LIME. Thus, we formulate the following RQ:

(RQ2) Does our PyExplainer produce higher accuracy of local models than LIME for JIT defect models?

Third, the explanations generated by LIME are still not specific to an instance to be explained. Another key principle of the model-agnostic techniques is to build a unique explanation that is specific to the prediction of an instance to be explained. Thus, explanations should be unique and highlight the key characteristics (i.e., features) of a commit that leads a global model to predict as defect-introducing. However, LIME uses a Quantile discretization (i.e., 1^{st}, 2^{nd}, 3^{rd} Quantiles) for generating rule features. The three bins used by LIME may not be enough to capture the highly-complex and highly-skewed JIT defect datasets. Thus, the rule features used by LIME may produce generic explanations that may not be specific to the instance to be explained, which may not reflect the actual characteristics of defect-introducing commits. Therefore, we formulate the following RQ:

(RQ3) Is our PyExplainer more effective in generating explanations than LIME for JIT defect models?

III. Our PyExplainer: A Local Rule-Based Model-Agnostic Technique

In this section, we present our PyExplainer, a local rule-based model-agnostic approach for explaining the predictions of JIT defect models.

Overview. Figure 1 illustrates an overview of the PyExplainer approach, which consists of four main steps. First, given an instance to be explained (i.e., a commit) and a global model, our PyExplainer will generate synthetic neighbors around the instance to be explained using the crossover and mutation techniques [35]. Second, our PyExplainer will obtains the predictions of the synthetic neighbors from the global model. Third, our PyExplainer builds a local rule-based regression model in order to learn the associations between the characteristics of the synthetic instances and the predictions from the global model. In the fourth step, our PyExplainer generates an explanation from the local model for the instance to be explained. We now describe each of the four steps below.

(Step 1) Generate Synthetic Neighbors Around the Instance to be Explained. Our PyExplainer will first generate synthetic neighbors (X') around the instance to be explained using the crossover and mutation techniques [35]. To do so, PyExplainer will find an initial set of actual neighbors, i.e., the actual instances around the instance to be explained in the training dataset. To identify the actual neighbors, PyExplainer applies the exponential kernel function (see Eq. 1) to calculate the similarity score between each instance in the training dataset (i_k) and the instance to be explained (i_e).

$$K(i_k, i_e) = \exp(-\frac{\text{dist}(i_k, i_e)^2}{2w^2}) \quad (1)$$

where $\text{dist}(i_k, i_e)$ is the euclidean distance between instances i_k and i_e, and w is the kernel width as the multiplication of 0.75 and the number of features of an instance ($w = 0.75 \times \#\text{features}$) as suggested by Ribeiro *et al.* [31].

Based on the initial set of actual neighbor, PyExplainer generates synthetic neighbors using crossover and mutation techniques to expand the initial set. The calculation of the crossover ($I_{\text{crossover}}$) and mutation (I_{mutation}) techniques can be derived as follows:

$$I_{\text{crossover}} = i_1 + (i_2 - i_1) * \alpha \quad (2)$$

$$I_{\text{mutation}} = i_1 + (i_2 - i_3) * \mu \quad (3)$$

where i_1, i_2, and i_3 are the randomly selected neighbourhood instances, α is a randomly generated number between 0 and 1; and μ is a randomly generated number between 0.5 and 1.

(Step 2) Obtain the Predictions of the Synthetic Instances using the Global Model. In Step 1, only the features of a synthetic neighbor (X') are generated. Hence, PyExplainer uses the global model to obtain the predictions (i.e., whether it is defective or clean given features of a synthetic instance). This allows PyExplainer to learn the behaviour of the underlying global model.

(Step 3) Build a Local Rule-based Regression Model using the RuleFit technique. To build a local model (F'), PyExplainer uses a rule-based logistic regression technique, called RuleFit [7]. RuleFit is a classifier that combines tree ensembles and linear models, which allows us to interpret the model like a traditional regression model, while understanding the logical reasons learnt from the rule features.

Broadly speaking, RuleFit will first generate *rule features* (X'_R), e.g., {Churn $>$ 100 & #Reviewers $<$ 2} based on ensemble decision trees (e.g., Gradient Boosting Trees). Then, RuleFit uses a regression model (i.e., logistic regression for binary outcomes, or linear regression for continuous outcomes) to model the association between the predicted outcomes (Y')

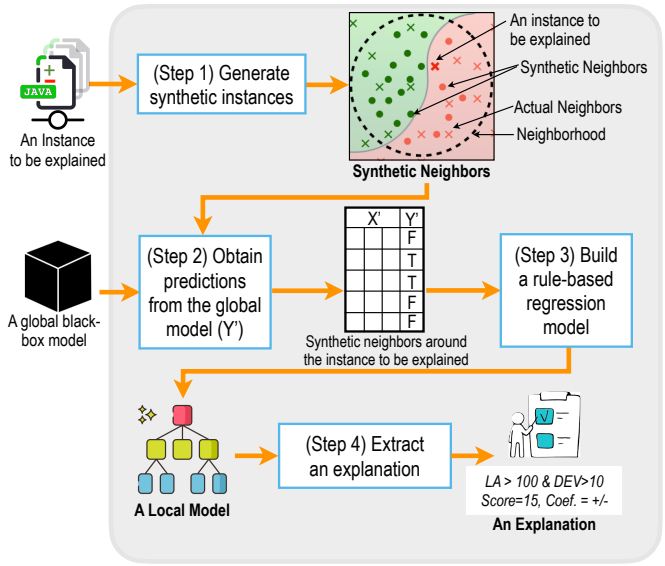


Fig. 1: An overview of the PyExplainer approach. Given an instance to be explained, PyExplainer produces four main component i.e., (1) synthetic neighbors, (2) a local model, and (3) an explanation. Each PyExplainer's explanation produces three pieces of information i.e., (1) a rule-based explanation, (2) an importance score, and (3) the direction of relationship of either supporting (+) or contradicting (-) the prediction.

and the rule features (X'_R) together with the original features (X). Then, the degree of importance and the coefficients of rule features and the original features can be analyzed from this regression model.

The use of RuleFit in our PyExplainer will address the limitation of LIME which does not account for interactions between features (i.e., the combination of rule features). Although existing rule-based model-agnostic techniques (e.g., SQAPlanner [30], Anchors [32], and LORE [9]) have been proposed, these techniques employed association rule mining techniques (e.g., Apriori, FP-Growth) which do not provide the degree of importance of the rules and the coefficients. Without the degree of importance of the rules and the coefficients provided by such techniques, we cannot quantify how strong the association between the rules and the predicted outcome and what the direction of the relationship is.

(Step 4) Extract an Explanation from the Local Rule-based Model. To generate an explanation, our PyExplainer analyzes the local model which is built using the RuleFit technique in Step 3. In the local model, there are three key pieces information: (1) rule features, (2) importance scores, and (3) coefficients. The importance score indicates the strength of the association between the rule feature and the predicted outcome. The coefficient can be used to indicate the direction of the relationship. For example, a positive coefficient indicates that a rule feature has a contribution towards the prediction of the TRUE class (i.e., DEFECT).

Based on the three key pieces of information in the local model, our PyExplainer generates an explanation by identifying the rule feature that has the highest importance score, has a positive coefficient, and satisfies the actual feature values of the instance to be explained. For example, suppose that a rule feature (Churn>100 & #Reviewers<2) has the highest importance score and has a positive coefficient, our PyExplainer will generate the following rule-based explanation: Churn>100 & #Reviewers<2 $\Rightarrow$ DEFECT, which means that a commit is predicted as defective since Churn is greater than 100 and the number of reviewers is less than 2.

IV. Experimental Design

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

Studied JIT Datasets. We select just-in-time defect datasets from two large-scale open-source software projects (i.e., Openstack and Qt) as provided by McIntosh and Kamei [25]. Openstack is an open-source software for cloud infrastructure service. Qt is a cross-platform application development framework. We choose Openstack and Qt datasets for our study, since both datasets (1) are often used as a benchmark in defect prediction studies [10, 11, 25, 29]; and (2) are manually verified for the validity of the SZZ algorithm [34] to reduce the number of false positives and false negatives. Table I provides an overview of the studied datasets.

Commit Features. 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 Appendix.

Experiment Design. Figure 2 presents an overview of our experimental design, which is composed of four main steps.

(Step 1) Split Data into Training and Testing Datasets. To ensure that the evaluation of our just-in-time defect prediction reflects a real-world scenario, we first sort the date of the commits to preserve the order of the commits in a chronological order. Then, we use a time-wise hold-out validation technique (as used by McIntosh and Kamei [25]) 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 model training. Similarly, the commits that appear earlier will not be used in model evaluation.

(Step 2) Build JIT Defect Models. For each training dataset, we first mitigate collinearity using AutoSpearman and handle class imbalance using SMOTE prior to build JIT defect models. Below, we describe each step in detail.

(Step 2-1) Mitigate Collinearity using AutoSpearman. To ensure that the interpretation of our JIT defect models is highly accurate, we mitigate collinearity and multi-collinearity, as suggested by prior studies [15, 16]. We use *AutoSpearman*, an automated feature selection approach to automatically select one feature from each group of highly correlated features that shares the least correlation with the other features that are not in the group [16]. As suggested by Kraemer *et al.* [19], we use a threshold of 0.7 to indicate strong correlation between

TABLE I: An overview of the studied JIT defect datasets provided by McIntosh and Kamei [25].

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%)

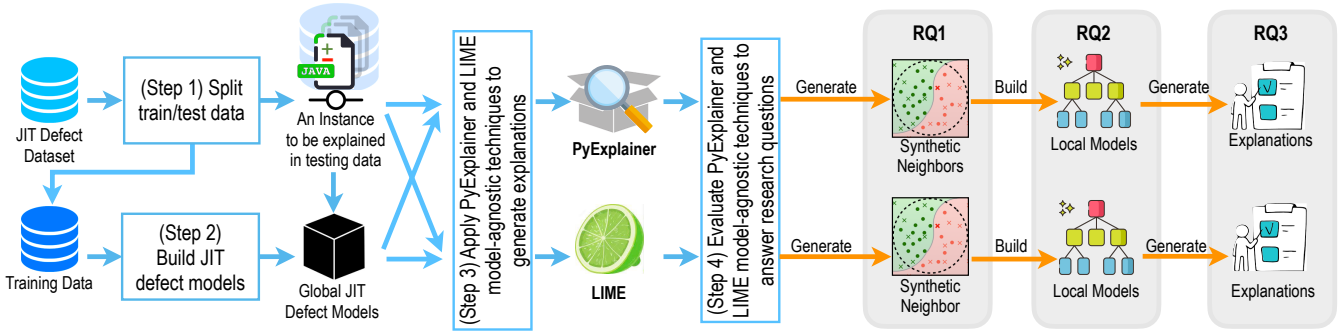


Fig. 2: An overview of the experimental design.

features. As suggested by Fox [6], we use a VIF threshold value of 5 to indicate multicollinearity. We use the implementation of *AutoSpearman* as provided by the `PyExplainer` Python package. After using AutoSpearman, we finally select 7 features that are not highly-correlated with each other.

(Step 2-2) Handle Class Imbalance using SMOTE. To ensure that the predictions of our JIT defect models are highly accurate, we apply a class rebalancing technique, as suggested by prior work [2, 36]. Since the defective ratio of our studied JIT defect datasets are highly imbalanced (i.e., 8%-16%), we apply SMOTE [3] to handle class imbalance *only on the training dataset*. We choose the SMOTE technique, as suggested by prior work [2, 36] who found that the SMOTE technique outperforms other class rebalancing techniques. SMOTE performs the following steps. First, SMOTE calculates the k-nearest neighbors of a set of minority class. Then, SMOTE randomly chooses the neighbors and generates synthetic instances around such neighbors. Finally, SMOTE combines the synthetic instances with the undersampling of the majority class to produce the final set of balanced instances. We use the implementation of SMOTE as provided by `Imbalanced-Learn` Python library [23]. We use the default setting ($k = 5$) of SMOTE, since our experiment with various k settings has shown that varying the k settings has little impact on the performance of our JIT defect models.

(Step 2-3) Evaluate Global JIT Defect Models. We build global JIT defect models using the training data of each project. We select the two classification techniques that are commonly-used in prior studies [8, 17], since they found that Random Forests (RF) and Logistic Regressions (LR) often outperform other classification techniques. Then, we evaluate the global JIT defect models on the testing dataset using 2 effort-aware measures (i.e., Recall@20%effort, and P_{opt}) and 2 traditional performance measures (i.e., Area Under the ROC Curve (AUC) and F1 (with a cutoff threshold of 0.5). We select classifiers using Recall@20%effort to ensure that they are practical when they are deployed in practices [27]. Recall@20%effort measures the percentage of correctly predicted defect-introducing commits that can be found when inspecting the top-20% LOC of the most risky commits.

Our global JIT defect models trained using both RF and LR techniques achieve similar performance for both OpenStack and Qt projects. Table II presents the accuracy of the JIT defectt models. For Openstack, our RF classifier achieves a Recall@20%Effort of 0.56 for RF and 0.54 for LR, indicating that our JIT defect models can correctly predicted 54%-56% of defect-introducing commits when spending only 20% code inspection effort (i.e., LOC). Similarly, for Qt, our RF classifier achieves a Recall@20%Effort of 0.83 for RF and 0.82 for LR, indicating that our JIT defect models can correctly predicted 82%-83% of defect-introducing commits when inspecting only 20% code inspection effort (i.e., LOC).

(Step 3) Apply our PyExplainer and the LIME model-agnostic techniques to generate explanations. For each prediction of JIT defect models, we apply our PyExplainer and LIME to generate an explanation of each prediction. We choose LIME as a baseline comparison, since LIME has been widely used in SE research [13, 28, 29, 37] Similar to PyExplainer, LIME produces three main components: (1) synthetic neighbors; (2) local models; and (3) explanations. LIME performs the following four steps to produce an explanation.

First, LIME randomly generates synthetic neighbors surrounding an instance to be explained using a random perturbation method with an exponential kernel function of euclidean distance. Second, LIME obtains the predictions of the synthetic neighbors from the global JIT defect models. Third, LIME builds a local sparse linear regression model (K-Lasso) using the randomly generated instances and their predictions from the global JIT defect models. Forth, LIME generates an explanation using the coefficients of the local K-Lasso

TABLE II: The accuracy of JIT defect models that are trained using Random Forest (RF) and Logistic Regression (LR).

	OpenStack			
Classification	**Recall@20%Effort**	**Popt**	**AUC**	**F1**
Random Forest	0.56	0.82	0.75	0.36
Logistic Regression	0.54	0.82	0.66	0.36
	Qt			
Classification	**Recall@20%Effort**	**Popt**	**AUC**	**F1**
Random Forest	0.83	0.94	0.74	0.21
Logistic Regression	0.82	0.95	0.64	0.16

model with three key pieces of information: (1) a decision rule feature; (2) the importance score; and (3) the direction of relationship of either supporting (+) or contradicting (-) the prediction towards a TRUE class (i.e., Defect).

(Step 4) Evaluate the PyExplainer and LIME model-agnostic techniques. Both PyExplainer and LIME use different techniques to generate synthetic neighbors (i.e., crossover and mutation vs. random perturbation) and the local models (i.e., RuleFit vs. K-Lasso). Thus, they may produce different explanations. Therefore, we aim to investigate which model-agnostic technique is the best to generate an explanation of the prediction obtained from JIT defect models. To evaluate PyExplainer and LIME, we focus on the three main components along two dimensions. In the first dimension, we focus on the common internal mechanism of the model-agnostic techniques i.e., the synthetic neighbour and the accuracy of their local models, since these two components are used to generate an explanation for a prediction. In the second dimension, we focus on the explanations generated by PyExplainer and LIME. We describe the analysis approach for each research question in Section V.

V. Experimental Results

In this section, we present the approach, and results with respect to our three research questions.

(RQ1) Does our PyExplainer produce better synthetic neighbors than LIME for JIT defect models?

Approach. To address RQ1, we analyze the distance between an instance to be explained and the synthetic instances around the neighbourhood using the Euclidean Distance measure. The Euclidean Distance measure is the calculation of distance between two feature vectors in an n-dimensional feature space (i.e., $d(i_1, i_2) = \sqrt{\sum_{j=1}^{n}(i_{1j} - i_{2j})^2}$, where $d(i_1, i_2)$ is a Euclidean Distance of two instances i_1 and i_2). The smaller the distance, the higher similarity of the both vectors (instances). Thus, the lower distances between an instance to be explained and the synthetic instances indicate that such generated synthetic instances yield high similarity with the instance to be explained.

For each instance to be explained in the testing dataset, we calculate the Euclidean Distance between the instance to be explained and their synthetic instances. Since the data is not normally distributed, we compute the median value (instead of the average) of the Euclidean Distance to approximate the average similarity of the instances around the neighbourhood. Then, we compare the distributions of the median Euclidean Distance of the synthetic neighbors produced by both PyExplainer and LIME.

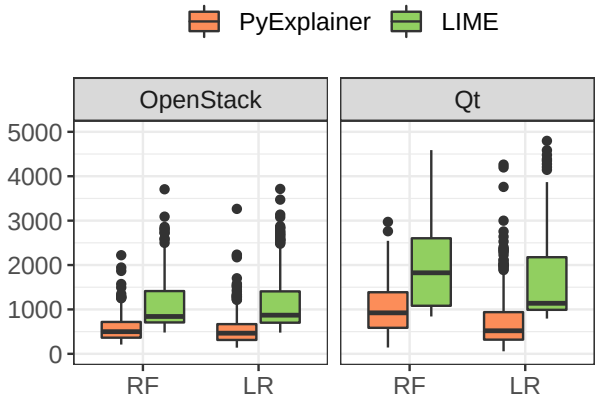


Fig. 3: (RQ1) The Euclidean Distance of neighborhood instances and instances to be explained, obtained from model-agnostic techniques (i.e., PyExplainer and LIME).

Finally, we apply two statistical test (i.e., Wilcoxon singed-rank test and Cliff's $|\delta|$ effect size) to identify whether differences of the Euclidean Distance produced by PyExplainer and LIME are statistically significant. The Wilcoxon signed-rank test is a non-parametric test that measures the difference of distribution between two population (i.e., the Euclidean Distance of PyExplainer and LIME). Cliff's $|\delta|$ is a non-parametric effect size test that measures the magnitude of the differences of the given two distributions. We use the Cliff's $|\delta|$ interpretation of Romano *et al.* [33] as follows, i.e., negligible for $|\delta| \leq 0.147$, small for $|\delta| \leq 0.33$, medium for $|\delta| \leq 0.474$, and large for $|\delta| > 0.474$ Finally, we compute the relative percentage difference using the following equation: $\%\text{diff} = \frac{\text{PyExplainer} - \text{LIME}}{\text{LIME}} \times 100$.

Results. The synthetic neighours produced by our PyExplainer is 41%-55% more similar to an instance to be explained than LIME for both RF and LR JIT defect models. Figure 3 shows that our PyExplainer achieves 41%-49% and 47%-55% lower Euclidean Distance for both RF and LR JIT defect models for both Openstack and Qt. For Openstack, we find that our PyExplainer achieves a median Euclidean Distance of 492, while LIME achieves a median Euclidean Distance 839. For Qt, we find that our PyExplainer achieves a median Euclidean Distance of 492, while LIME achieves a median Euclidean Distance 1,825. The Wilcoxon signed-ranked test confirms that the distributions of the Euclidean Distance of our PyExplainer is statistically significantly smaller than LIME (p-value < 0.05) with a large Cliff's $|\delta|$ effect size for both classifiers and both projects.

This finding indicates that our PyExplainer produces synthetic neighbors that are more closely similar to an instance to be explained than LIME. The less similarity of synthetic neighbors generated by LIME has to do with the use of random

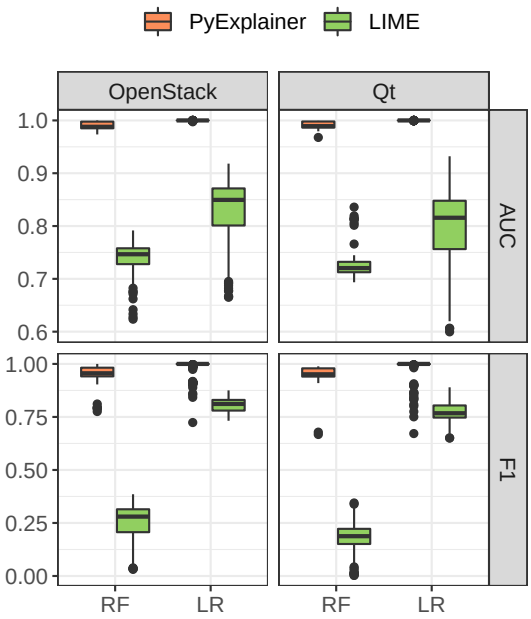


Fig. 4: (RQ2) The accuracy of the local models produced by our PyExplainer and LIME in terms of AUC and F1.

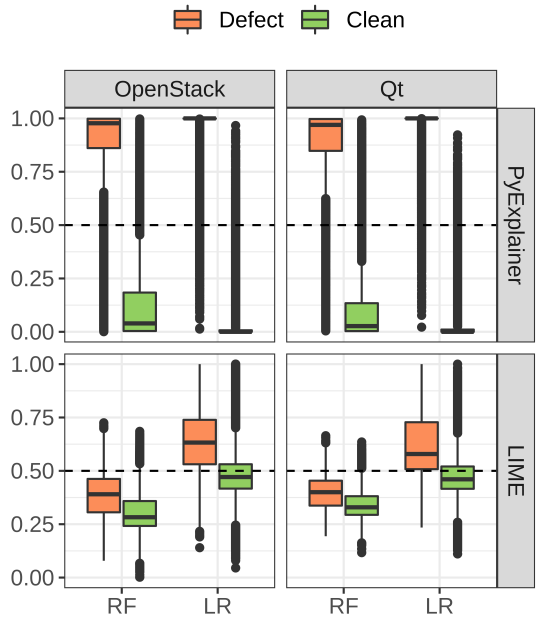


Fig. 5: (RQ2) The probability (y-axis) of synthetic instances predicted by the local models of PyExplainer and LIME, when comparing to the actual class of that instances (i.e., the legend of defect and clean) from the RF and LR global JIT defect models.

perturbation approach. The random perturbation approach perturbs an instance to be explained by adding a value randomly drawn from a normal distribution. However, such a simple random perturbation approach is not suitable for sparse and high dimensional data like JIT datasets [40]. In particular, the random perturbation approach does not account for the characteristics of the actual JIT datasets. Thus, we found that the random perturbation approach often generates synthetic neighbors that are less similar to an instance to be explained than our PyExplainer. On the other hand, our PyExplainer generates synthetic neighbors based on the characteristics of JIT dataset using the crossover and mutation operations, producing a more accurate explanation for the predictions of JIT defect models.

(RQ2) Does our PyExplainer produce higher accuracy of local models than LIME for JIT defect models?

Approach. To address RQ2, we analyze the accuracy of the local models generated by PyExplainer and LIME. The accuracy of local models indicates how well local models can approximate (or mimic) the logic of the global models. To do so, we first obtain the predicted class (i.e., CLEAN and DEFECT) of the synthetic instances from the global JIT defect models. Then, we obtain the probability of CLEAN and DEFECT class of synthetic instances from the local models. Then, we evaluate the accuracy of the predictions between the local models and the global JIT model using two traditional performance measures, i.e., Area Under the ROC Curve (AUC) and F1. Similar to RQ1, we apply the Wilcoxon signed-rank test and the Cliff's $|\delta|$ effect size test to evaluate whether the accuracy of local models of PyExplainer are statistically significantly higher than LIME.

Results. PyExplainer produces local models that are 18%-38% more accurate than the LIME's local models. Figure 4 shows that the local models produced by our PyExplainer achieve a median AUC of 0.99 when explaining the RF and LR JIT defect models for both Openstack and Qt. On the other hand, the local models produced by LIME achieves a median AUC of 0.75 for Openstack and 0.72 for Qt when explaining the RF JIT defect models, while achieving a median AUC of 0.85 for Openstack and 0.82 for Qt when explaining the LR JIT defect models. This indicates that our PyExplainer produces local models that are 32%-38% and 18%-24% more accurate (AUC) than LIME for both RF and LR JIT defect models. Finally, we observe a similar conclusion when using F-measure, i.e., PyExplainer produces local models that are 242.86%-413.5% and 23.46%-29.87% more accurate (F1) than LIME for both RF and LR JIT defect models. The Wilcoxon signed-ranked test confirms that the accuracy of local models produced by our PyExplainer is statistically significantly higher than the accuracy of local models produced by LIME (p-value < 0.05) with a large Cliff's $|\delta|$ effect size.

The more accurate local models (i.e., high AUC) produced by our PyExplainer have to do with the quality of synthetic neighbors generated by our PyExplainer. First, our PyExplainer uses crossover and mutation techniques to generate synthetic neighbors. Thus, the synthetic neighbors are more closely similar to an instance to be explained and more similar to the actual characteristics of defect-introducing commits and clean commits from the training data. Therefore, we perform a deeper investigation to better understand the distribution

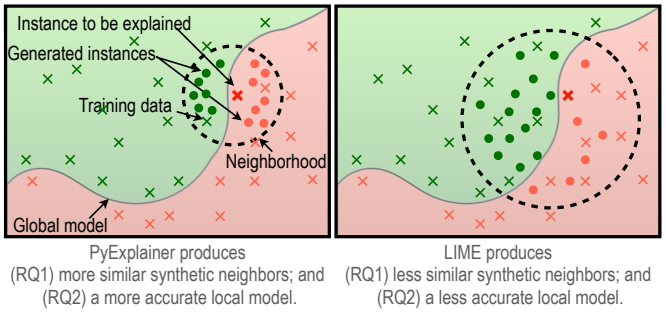


Fig. 6: The characteristics of synthetic neighbors generated by PyExplainer and LIME.

of the probability of synthetic neighbours generated by the PyExplainer and LIME local models. Figure 5 shows that the median probability of defect class (0.98-1.00) and clean class (0.00-0.04) generated by the PyExplainer local models differs by 0.96-0.98. On the other hand, the median probability of defect (0.39-0.63) and clean (0.28-0.47) classes generated by the LIME local models differs by 0.11-0.16. This findings indicates that the PyExplainer local models have a higher ability to discriminate the characteristics between DEFECT and CLEAN classes, producing higher AUC than the LIME local models.

Finally, we illustrate the characteristics of synthetic neighbors generated by PyExplainer and LIME in Figure 6. In RQ1, the smaller Euclidean Distance by PyExplainer indicates that PyExplainer produces synthetic neighbors that are more similar to (1) an instance to be explained; and (2) the actual characteristics of the JIT defect datasets due to the use of crossover and mutation operations on training data. In RQ2, the higher AUC and F1 by PyExplainer indicates that our PyExplainer produces better synthetic neighbors, leading to more accurate local models than LIME. Therefore, the explanation generated by PyExplainer is more closely similar to the explanation of an instance to be explained than the explanation generated by LIME.

(RQ3) Is our PyExplainer more effective in generating explanations than LIME for JIT defect models?

Approach. To address RQ3, we analyze the explanations produced by PyExplainer and LIME using the two measures.

- *%Unique* measures the percentage of unique explanations generated by each technique. The higher percentage of unique explanations indicates that a model-agnostic technique can effectively generate a more specific (i.e., less duplicate) explanation to the instance to be explained.
- *%Consistency* measures the percentage of the defect-introducing commits in the testing data that have characteristics satisfying the rule feature in the generated explanation. The higher percentage of the consistency indicates that a model-agnostic technique can effectively generate an explanation that is consistent with the actual characteristics of defect-introducing commits.

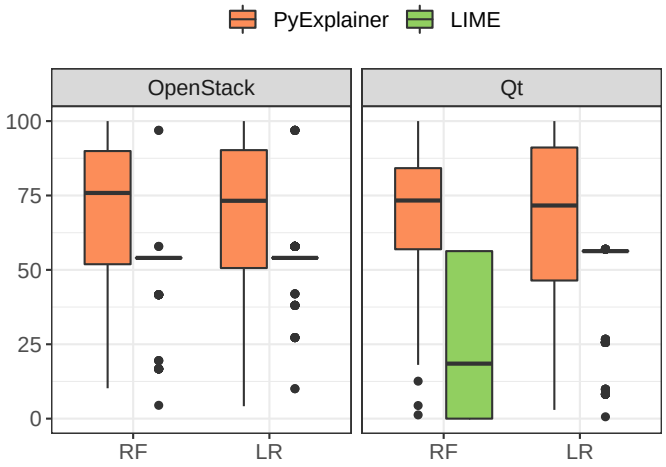


Fig. 7: (RQ3) The percentage of the defect-introducing commits in the testing data that are consistent with the generated explanation.

Results. The explanations generated by our PyExplainer are 69%-98% more unique (i.e., more specific to an instance to be explained) than LIME. We find that PyExplainer can produce 100% unique explanations for all of the instances to be explained for both studied datasets. On the other hand, LIME can produce as few as 2%-4% unique explanations for OpenStack and 3%-31% unique explanations for Qt. In other words, for OpenStack, we find that as much as 72%-86% of defect-introducing commits have the same explanation, despite having different characteristics of the feature values. Similarly, for Qt, we find that as much as 53%-74% of commits have the same explanation, despite having different characteristics of the testing instances. Thus, the less duplicate explanations generated by PyExplainer indicates that PyExplainer can generate explanations that are more specific to an instance to be explained rather than LIME.

The explanations generated by our PyExplainer are 17%-54% more consistent with the actual defect-introducing commits in the testing data than LIME. Figure 7 shows that PyExplainer achieves a median consistency of 73%-75% for OpenStack and 72%-73% for Qt. On the other hand, we find that LIME achieves a median consistency of 54% for OpenStack and 18%-56% for Qt. The experiment result indicates that the explanations generated by our PyExplainer are 19%-21% and 17%-54% more consistent with the actual defect-introducing commits in the testing data than LIME for OpenStack and Qt, respectively. The Wilcoxon signed-ranked test confirms that the percentage consistency value of PyExplainer is statistically significantly higher than LIME (p-value < 0.05) with a large Cliff's $|\delta|$ effect size for both JIT defect models and both studied datasets.

VI. Discussion

In this section, we first discuss the usage scenario of how PyExplainer can be used in practice. Then, we present an analysis of the *What-If* simulation when the explanations were

considered (i.e., *what if* we change this, would it reverse the predictions of the JIT defect models?). Finally, we describe the implementation details of the PyExplainer Python package.

A. A Usage Scenario

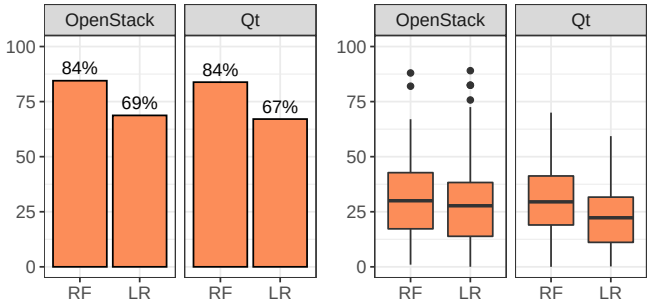
Let's consider Bob as a developer in a large-scale software project that adopts modern code review practices. Bob has his main responsibility to inspect commits that are submitted by other developers to ensure the quality of commits prior to integration into the release branch. Suppose that on average Bob spends one hour to review one commit. Hence, with his average 8 working hours per day, he can review only 8 commits per day. Given a huge number of newly arrived commits everyday (e.g., 100 commits per day), Bob does not know which commits should be reviewed first. With the use of JIT model, the list of commits can be prioritized based on the likelihood of being defect-introducing provided by the JIT defect model so that Bob can efficiently spend his limited time on the most risky commits. However, Bob still may not be convinced by the predictions of JIT defect models, since he *does not understand why a commit is predicted as defect-introducing*. Thus, Bob may not trust the JIT defect models and may decide to ignore the predictions, resulting in suboptimal SQA resource allocation and prioritization.

With PyExplainer, Bob now better understands why a commit is predicted as defect-introducing since PyExplainer provides an explanation (i.e., which feature is the most important for a given prediction). For example, PyExplainer provides an explanation (e.g., Churn $>$ 100 $\Rightarrow$ Defect) that a commit is predicted as defect-introducing because the churn size is greater than 100. This kind of explanation could help Bob to focus on the most important aspects that are associated with the risk of being defect-introducing (i.e., considering reducing the churn size of the commit), instead of focusing on the less important aspects (e.g., inviting more reviewers). However, it remains challenging for Bob to consider which value of a feature that should be changed to mitigate the risk. In particular, given an explanation (e.g., Churn $>$ 100 $\Rightarrow$ Defect), Bob still does not know how small a Churn value should be that could reverse the prediction of JIT models from DEFECT to CLEAN. Thus, an interactive *what-if* visualization tool is needed to help Bob making better decisions of how much the Churn value that should be changed.

B. What-If Analysis

We conducted a *what-if* simulation based on a hypothetical scenario if the explanations of our PyExplainer were considered. In particular, we investigated *what if* we change the value of a feature guided by the explanation, would it reverse the prediction of the JIT defect model?. For example, an explanation (Churn$>$100 $\Rightarrow$ DEFECT) generated by PyExplainer means that a commit is predicted as defect-introducing since Churn is greater than 100. Thus, what if Churn was less than 100, would the prediction be reversed from DEFECT to CLEAN.

To conduct this what-if simulation, we first generate a simulated instance. The *simulated instance* is an instance where the actual value of a feature guided by the rule-based explanation was changed in the opposite direction of the comparison operator of the explanation (i.e., decrease for $>$, increase for $<$) by one SD (a standard deviation of that feature in the training data) from the rule threshold. According to the above example, the simulated instance is the modified original instance where the actual value of a feature (e.g., Churn=120) guided by the rule-based explanation (Churn$>$100 $\Rightarrow$ DEFECT) was changed in the opposite direction of the comparison operator of the explanation (i.e., decrease for $>$) by one SD (e.g., 20) from the rule threshold (100). Thus, the Churn value of the simulated instance is 80 (i.e., 100-20). Then, we input this simulated instance to the global JIT defect model and analyze whether the simulated instance could reverse the predictions of the global JIT defect models.



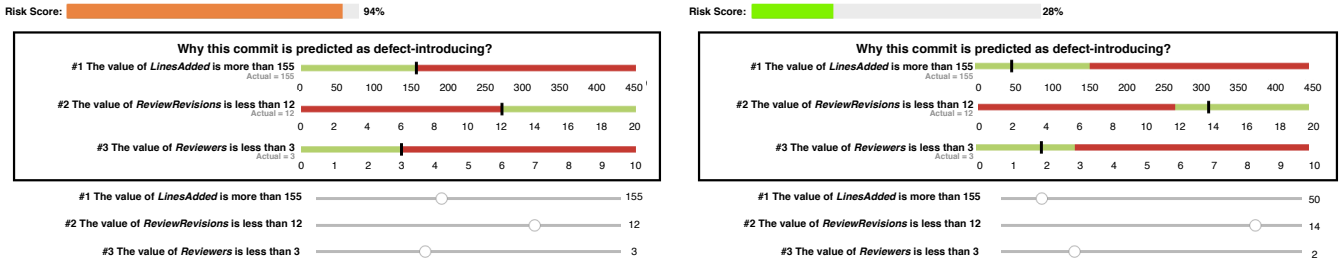
(a) The percentage of the simulated instances that can reverse the prediction from DEFECT to CLEAN.

(b) The difference between the probability of the original instance and the probability of the simulated instance.

Fig. 8: The results of the *what-if* analysis.

We perform this what-if simulation for all the explanations generated by our PyExplainer for all of the commits in the testing dataset that are correctly predicted as defect-introducing by the JIT defect models. Then, we measure (1) *%reversed*, i.e., the percentage of the simulated instances that can reverse the prediction from DEFECT to CLEAN; and (2) *%prob_diff*, i.e., the difference between the probability of the original instance and the probability of the simulated instance.

Figure 8a shows that, when considering the explanations guided by our PyExplainer, 84% (RF) and 67%-69% (LR) of the simulated instances that can reverse the prediction from DEFECT to CLEAN of the global JIT defect models. Furthermore, Figure 8b also shows that, after considering the explanations guided by our PyExplainer, the probability of the simulated instance is decreased by 30% for the RF models and 22% - 28% for the LR models when comparing to the probability of the original instance. This simulation highlights the importance of our PyExplainer for helping practitioners to focus on the most important aspects that are associated with the risk of being defect-introducing for a given commit, instead of focusing on the less important aspects. Nevertheless, the one SD used in this what-if simulations is just an example, the actual changed value should be subject to the domain experts.



(a) The visual explanation of the original instance (predicted as DEFECT with a risk score of 98%).

(b) The visual explanation of the simulated instance when changing the feature values (predicted as CLEAN with a risk score of 28%).

Fig. 9: The proof-of-concept visualization of our PyExplainer consists of (1) the risk score (i.e., the probability of an instance to be explained by the global JIT model); (2) the visual explanation (in the black border); and (3) the interactive what-if visualization for our PyExplainer.

C. The PyExplainer Python package

To ease the adoption of our PyExplainer by practitioners and to facilitate the replication of future research, we developed the PyExplainer Python package. In our PyExplainer Python package, we also developed a proof-of-concept of the visual explanation and the interactive what-if visualization.

The visual explanation is developed to present the rule-based explanation in a form of a bullet plot visualization with textual explanations. Figure 9a shows an example of the visual explanation of an OpenStack commit (`a9a59cc`). Our visual explanation is designed to provide the following key information: (1) textual descriptions that explain why a commit is predicted as defect-introducing; (2) the actual feature values of the commit (i.e., the vertical black bars); and (3) the range of feature values associated with the risk score (i.e., the predicted probability). The green shades indicate the non-risky range values of a feature, while the red shades indicate the risky range values of a feature.

An interactive *what-if* visualization is developed to help practitioners interactively change the value of a feature, while immediately generating the new estimated risk score (i.e., the probability obtained from the JIT defect model). This visualization will allow practitioners to explore different values of a feature prior to making a decision.

Figure 9b shows an example of an interactive what-if visualization for an OpenStack commit (`a9a59cc`). Through the visualization, the user can change the value of the feature (e.g., changing the value of *Lines Added* from 155 to 50, the value of *Review Revisions* from 12 to 14, and the value of *Reviewers* from 3 to 2). Then, the visualization will responsively update the predicted probability generated by the JIT defect model (e.g., from 94% to 28%).

D. Implications to Practitioners and Researchers

The contributions of this paper build an important step towards a new research area of Explainable AI for SE, by making the predictions of just-in-time defect models more explainable and actionable. A lack of explainability and actionability of software analytics has been raised by both practitioners [5, 14, 24] and researchers [4, 13, 20, 28, 30, 38]. For example, Rajapaksha *et al.* [30] proposed an approach to generate actionable suggestions (i.e., counterfactual explanations) for file-level defect prediction. Peng and Menzies [28] also proposed a TimeLIME approach (an extension of LIME model-agnostic technique) to generate actionable suggestions (i.e., defect reduction plans). However, the approaches of both Rajapaksha *et al.* [30] and Peng and Menzies [28] are designed for release-based defect prediction, which require multiple releases for training and evaluation. Thus, they are not applicable to JIT defect prediction models. On the other hand, our results show that our PyExplainer is more effective in generating explanations than LIME for the predictions of JIT defect models, while providing an interactive what-if visualization so practitioners can make better data-informed decisions. Similar effort to other state-of-the-art model agnostic techniques (e.g., LIME [31]), we make our PyExplainer Python package publicly-available to ease the adoption by practitioners and researchers.

E. Threats to Validity

Threats to construct validity relates to the hyperparameter settings of RandomForest, SMOTE, and LIME techniques when conducting our experiment. To ensure the reprehensibility, the used parameter setting of such techniques are reported in the replication package in the Supplementary Material. The final replication package will be in Zenodo upon acceptance.

Threats to internal validity relates to the randomization of our PyExplainer (i.e., the neighbour generation process). To mitigate any conclusion instability threat, we chose to generate 2,000 neighbours. After we repeated the experiment five times, the conclusion of our paper remains the same. Nevertheless, future work can explore what would be the minimum synthetic neighbours that can produce stable local explanations (i.e., the same local explanations when they are regenerated).

Threats to the external validity relates to the generalizability of our PyExplainer approach. PyExplainer is designed for explaining the predictions of any classification problems. However, our experiment only focused on the just-in-time defect prediction problem, the limited number of the studied classification techniques, and the limited number of studied

projects. Thus, future work should explore if our PyExplainer can be used to effectively explain the predictions of other classification problems (e.g., vulnerability prediction, code smell detection).

VII. Conclusion

Prior studies proposed Just-In-Time (JIT) defect prediction, yet its explanability remains largely unexplored (i.e., practitioners still do not know why a commit is predicted as defect-introducing). In this paper, we propose PyExplainer, a novel local rule-based model-agnostic technique for explaining the predictions of JIT defect models. Through a case study of two open-source software projects, we find that our PyExplainer produces: (1) synthetic neighbours that are 41%-45% more similar to an instance to be explained; (2) 18%-38% more accurate local models; and (3) explanations that are 69%-98% more unique and 17%-54% more consistent with the actual characteristics of defect-introducing commits in the future than LIME (a state-of-the-art model-agnostic technique).

Publishing the PyExplainer Python Package. To ease the adoption of our PyExplainer by practitioners and to facilitate the replication of future research, the PyExplainer package is available in both `conda` and `pip` (Package Installer for Python). Our PyExplainer Python package also has a code coverage of 93% measured by CodeCov with an A+ quality graded by LGTM.

Acknowledgement. Chakkrit Tantithamthavorn was supported by ARC DECRA Fellowship (DE200100941). Patanamon Thongtanunam was supported by ARC DECRA Fellowship (DE210101091).

References

[1] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies, "How to"dodge" complex software analytics," *IEEE Transactions on Software Engineering (TSE)*, 2019.

[2] A. Agrawal and T. Menzies, "Is Better Data Better Than Better Data Miners?: On the Benefits of Tuning SMOTE for Defect Prediction," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 1050–1061.

[3] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, pp. 321–357, 2002.

[4] D. Chen, W. Fu, R. Krishna, and T. Menzies, "Applications of Psychological Science for Actionable Analytics," in *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 456–467.

[5] H. K. Dam, T. Tran, and A. Ghose, "Explainable Software Analytics," in *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2018, pp. 53–56.

[6] J. Fox, *Applied regression analysis and generalized linear models*, 2015.

[7] J. H. Friedman, B. E. Popescu *et al.*, "Predictive learning via rule ensembles," *Annals of Applied Statistics*, pp. 916–954, 2008.

[8] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 172–181.

[9] R. Guidotti, A. Monreale, S. Ruggieri, D. Pedreschi, F. Turini, and F. Giannotti, "Local rule-based explanations of black box decision systems," *arXiv preprint arXiv:1805.10820*, 2018.

[10] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2019, pp. 34–45.

[11] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020, pp. 518–529.

[12] Y. Jia, J. Bailey, K. Ramamohanarao, C. Leckie, and M. E. Houle, "Improving the quality of explanations with local embedding perturbations," in *Proceedings of International Conference on Special Interest Group on Knowledge Discovery & Data Mining (SIGKDD)*, 2019, pp. 875–884.

[13] J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models," *IEEE Transactions on Software Engineering (TSE)*, p. To Appear, 2020.

[14] J. Jiarpakdee, C. Tantithamthavorn, and J. Grundy, "Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021, p. To Appear.

[15] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The Impact of Correlated Metrics on Defect Models," *IEEE Transactions on Software Engineering (TSE)*, p. To Appear, 2019.

[16] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 92–103.

[17] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A Large-Scale Empirical Study of Just-In-Time Quality Assurance," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 757–773, 2013.

[18] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting Faults from Cached History," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007, pp. 489–498.

[19] H. C. Kraemer, G. A. Morgan, N. L. Leech, J. A. Gliner, J. J. Vaske, and R. J. Harmon, "Measures of Clinical Significance," *Journal of the American Academy of Child & Adolescent Psychiatry (JAACAP)*, pp. 1524–1529, 2003.

[20] R. Krishna and T. Menzies, "Learning Actionable Analytics from Multiple Software Projects," *Empirical Software Engineering (EMSE)*, pp. 3468–3500, 2020.

[21] R. Krishnan, D. Liang, and M. Hoffman, "On the challenges of learning with inference networks on sparse, high-dimensional data," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2018, pp. 143–151.

[22] T. Laugel, X. Renard, M.-J. Lesot, C. Marsala, and M. Detyniecki, "Defining locality for surrogates in post-hoc interpretablity," *arXiv preprint arXiv:1806.07498*, 2018.

[23] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *Journal of Machine Learning Research*, pp. 1–5, 2017.

[24] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr, "Does Bug Prediction Support Human Developers? Findings from a Google Case Study," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 372–381.

[25] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering (TSE)*, pp. 412–428, 2017.

[26] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 107–116.

[27] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu, "Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction," *IEEE Transactions on Software Engineering*, 2020.

[28] K. Peng and T. Menzies, "Defect Reduction Planning (using TimeLIME)," *IEEE Transactions on Software Engineering (TSE)*, 2021.

[29] C. Pornprasit and C. Tantithamthavorn, "JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021, p. To Appear.

[30] D. Rajapaksha, C. Tantithamthavorn, J. Jiarpakdee, C. Bergmeir, J. Grundy, and W. Buntine, "SQAPlanner: Generating Data-Informed Software Quality Improvement Plans," *IEEE Transactions on Software Engineering (TSE)*, 2021.

[31] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you?: Explaining the Predictions of Any Classifier," in *Proceedings of the International Conference on Knowledge Discovery & Data Mining (KDD)*, 2016, pp. 1135–1144.

[32] ——, "Anchors: High-precision model-agnostic explanations," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[33] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate Statistics for Ordinal Level Data: Should we really be using T-test and Cohen's d for Evaluating group differences on the NSSE and other surveys," in *Annual meeting of the Florida Association of Institutional Research (FAIR)*, 2006, pp. 1–33.

[34] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, 2005, p. 1–5.

[35] M. Srinivas and L. M. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics (TSMC)*, pp. 656–667, 1994.

[36] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The Impact of Class Rebalancing Techniques on The Performance and Interpretation of Defect Prediction Models," *IEEE Transactions on Software Engineering (TSE)*, p. To Appear, 2019.

[37] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Transactions on Software Engineering (TSE)*, 2020.

[38] Y. Yang, D. Falessi, T. Menzies, and J. Hihn, "Actionable analytics for software engineering," *IEEE Software*, pp. 51–53, 2017.

[39] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 157–168.

[40] H. Zhang, A. Nelson, and T. Menzies, "On the value of learning from defect dense components for software defect prediction," in *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, 2010, pp. 1–9.

VIII. Appendix

A. Evaluation Measures of the Global JIT Defect Models

Below, we describe the four evaluation measures (i.e., Recall@20%effort, P_{opt}, AUC, F1) that are used to evaluate the global JIT defect models.

Recall@20%effort measures the proportion of defective lines that are found from the top 20% of changed lines of a codebase. A high value of Recall@20%effort indicates that a model is good at ranking defective commits at the top.

P_{opt} is defined as 1-Δ_{opt}, where Δ_{opt} is the area of the effort-based (i.e., churn) cumulative lift chart between an optimal model and a prediction model. The effort-based (i.e., churn) cumulative lift chart is the relationship between the cumulative percentage of defect-introducing commits from a prediction model (y-axis) and the cumulative percentage of the inspection effort (x-axis). In this paper, we use the normalized version of P_{opt} similar to prior studies [1, 26, 39].

AUC measures how well a model can discriminate test instances into different classes based on various probability threshold. AUC ranges from 0 to 1, where AUC of 0.5 indicates a random guessing model and AUC of 1.0 indicates a perfect discrimination model.

F1 (or F-measure) is a harmonic mean of precision and recall ranging from 0 to 1. The higher value of F1, the better classification ability of a model.

B. A Tutorial of the PyExplainer Python Package.

Below, we present a tutorial of how to use the PyExplainer Python package step-by-step using Code Block 1.

(Step 1) The PyExplainer package is installed using pip (Python Package Management system).

(Step 2) The PyExplainer package is imported.

(Step 3) The data for demonstration is obtained from PyExplainer package. The data is composed of `X_train`, `y_train`, `indep`, `dep`, `blackbox_model`, `X_explain`, `y_explain`. The `X_train` variables are used to generate neighborhood instances. The `indep` and `dep` variables specify the feature names and label, respectively. The `blackbox_model` is the global JIT defect models from Scikit-learn module. The `X_explain` represents an instance to be explained while the `y_explain` is the label of the instance to be explained.

(Step 4) A PyExplainer object is created and an explanation is obtained from the `explain` function.

(Step 5) The visual explanation and the *what-if* visualization are generated by the `visualise` function.

```
# step 1 - install the pyexplainer package
!pip install pyexplainer
# step 2 - import necessary libraries
from pyexplainer.pyexplainer_pyexplainer import
    PyExplainer as pyexp
# step 3 - get the preprocessed data and global
    model to be tuned in to the PyExplainer object
dflt = pyexplainer_pyexplainer.get_dflt()
# step 4 - create a PyExplainer object using the
    preprocessed data and model, and generate rules
    by utilising the built-in local model in
    PyExplainer
exp = pyexp(X_train=dflt['X_train'],
            y_train=dflt['y_train'],
            indep=dflt['indep'],
            dep=dflt['dep'],
            blackbox_model=dflt['blackbox_model'])
rules = exp.explain(X_explain=dflt['X_explain'],
                    y_explain=dflt['y_explain'])
# step 5 - visualise the rules generated by the
    local model and the prediction generated by the
    global model
exp.visualise(rules)
```

Code Block 1: An Example Tutorial of the PyExplainer Python package.