An Empirical Study on the Stability of Explainable Software Defect Prediction

Jiho Shin*, Reem Aleithan*, Jaechang Nam†, Junjie Wang‡, Nima Shiri Harzevili*, Song Wang*
*York University; †Handong Global University; ‡Institute of Software, Chinese Academy of Sciences
{jihoshin, reem1100, nshiri, wangsong}@yorku.ca; jcnam@handong.edu; junjie@iscas.ac.cn

***Abstract*—Explaining the results of software defect prediction (SDP) models is practical but challenging. Jiarpakdee et al. proposed using two model-agnostic techniques (i.e., *LIME* and *BreakDown*) to explain prediction results. They showed that model-agnostic techniques can achieve remarkable performance and that the generated explanations can assist developers in understanding the prediction results. However, the fact that they examined these model-agnostic techniques only under a specific SDP setting calls into question their reliability on SDP models under various settings.**

In this paper, we set out to investigate the reliability and stability of model-agnostic-based explanation generation approaches on SDP models under different settings, e.g., different data sampling techniques, machine learning classifiers, and prediction scenarios used when building SDP models. We use model-agnostic techniques to generate explanations for the same instance under various SDP models with different settings and then check the stability of the generated explanations for the instance. We reused the same defect data and experiment configurations from Jiarpakdee et al. in our experiments. The results show that the examined model-agnostic techniques generate inconsistent explanations under different SDP settings for the same test instances. Our user case study further confirms that inconsistent explanations can significantly affect developers' understanding of the prediction results, which implies that the model-agnostic techniques can be unreliable for practical explanation generation under different SDP settings. To conclude, we urge a revisit of existing model-agnostic-based studies in software engineering and call for more research in explainable SDP toward achieving stable explanation generation.

***Index Terms*—Software bugs, static detection, machine learning libraries**

I. Introduction

Software Defect Prediction (SDP) models have been actively studied to allocate testing resources efficiently to reduce development costs. Most SDP models use various code and development metrics as features to classify a target code fragment as buggy or not. However, a major issue they face is lacking actionable messages for the developers to act upon [1], making it very difficult for practical usage.

To address this issue, studies investigating explainable artificial intelligence (XAI) in the domain of SDP have been explored recently [2]–[5] but most of these approaches target a global explanation, which summarizes a prediction of a whole model (i.e., the relationship between SDP features and the bug proneness). Since the global explanation does not provide a detailed interpretation of each prediction result, Jiarpakdee et al. [6] proposed to use the model-agnostic methods, i.e., *LIME* [7] and *BreakDown* [8], [9] to generate an instance explanation to explain the prediction of each target code fragment. Note that Jiarpakdee et al. [6] also reported that LIME can generate different explanations when re-generating explanations of the same instance because of its randomness, they further proposed an improved variant of LIME, i.e., LIME with Hyper Parameter Optimisation (LIME-HPO). The explanation is defined as a list of ordered features. Their experiments and user case studies showed that both *LIME-HPO* and *BreakDown* achieve promising performance and the generated explanations can assist developers by showing actionable guidance for practical usage.

However, in Jiarpakdee et al. [6], *LIME-HPO* and *BreakDown* were only examined on a single SDP setting which leaves unanswered the more directly relevant question: *Are model-agnostic techniques stable under SDP models with different settings?* The answer to this question is critical. First, many studies conduct SDP under different settings. The explanations generated by model-agnostic techniques are expected to be consistent across different settings to make them stable and reliable. Second, we have seen many studies follow Jiarpakdee et al. [6] to use model-agnostic techniques for other tasks, e.g., defective line prediction [3], online buggy commit prediction [10], and software quality assurance planning [11]. Understanding the stability of model-agnostic techniques will help confirm the findings from inline studies and benefit future research.

In this study, we investigate the reliability and stability of model-agnostic techniques (i.e., *LIME-HPO* and *BreakDown*) on SDP models under different settings. Specifically, we consider three different settings when building SDP models, i.e., data sampling techniques, machine learning (ML) classifiers, and prediction scenarios. Data sampling techniques are used in SDP studies [12]–[14] to solve the data imbalance issue. In this work, we experiment with five widely used sampling methods (details are in Section III-C). Various ML classifiers, e.g., Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), etc., have been used to build SDP models [15]–[18]. In this work, following Jiarpakdee et al. [6], we experiment with six common ML classifiers (details are in Section III-D). For training SDP models, one can choose different versions of historical data as the training data, i.e., cross-version defect prediction [19]. In this work, we also examine the reliability and stability of *LIME-HPO* and *BreakDown* when using different versions of data to build the SDP model.

For our analysis, we reuse the same dataset from Jiarpakdee et al. [6], which contains 32 versions of defect data from nine large-scale open-source Java projects. We follow the experiment settings described in Jiarpakdee et al. [6] to run our experiments for generating explanations under SDP settings and then we use two metrics, i.e., hit_rate and $rank\_diff$ from [6], to evaluate the consistency of two explanations for the same instance. Our experimental results show that explanations generated by both *LIME-HPO* and *BreakDown* are significantly inconsistent when different settings are applied, which makes them unreliable to use in practice. Our user case study further confirms that inconsistent explanations can significantly affect developers' understanding of the prediction results, which implies that the model-agnostic techniques can be unreliable for practical explanation generation under different SDP scenarios. Overall, with this study, we urge to revisit other explainable software analytic studies that adopt model-agnostic techniques and call for more research in explainable SDP towards achieving consistent explanation generation across different SDP settings.

This paper makes the following contributions:

- We perform the first study to analyze the reliability and stability of state-of-the-art model-agnostic explanation generation techniques, i.e., *LIME-HPO* and *BreakDown* on SDP models with three typical settings, i.e., data sampling techniques, ML classifiers, and prediction scenarios.
- We show neither *LIME-HPO* nor *BreakDown* can generate consistent explanations and the generated explanation under different SDP settings. In addition, our user case study further confirms that inconsistent explanations can significantly affect developers' understanding of the prediction results.
- We release the source code and the dataset of this work to help other researchers replicate and extend our study[1].

We organized the rest of this paper as follows. Section II presents the background and motivation of this study. Section III shows the experimental setup. Section IV presents the evaluation results. Section V discusses open questions related to our study. Section VI shows our user case study. Section VII presents the threats to the validity of this work. Section VIII presents the related studies. Section IX concludes this paper.

II. Background and Motivation

This section introduces the background of SDP models and the explanation generation techniques studied in Jiarpakdee et al. [6] and our motivation example.

A. File-level SDP Models Studied in Jiarpakdee et al. [6]

The objective of a file-level SDP model is to determine risky files for further software quality assurance activities [20]–[25]. A typical release-based file-level SDP model mainly has three steps. The first step is to label the files in an early version as buggy or clean based on post-release defects for each file. Post-release defects are defined as defects that are revealed within a post-release window period (e.g., six months) [21], [26]. One could collect these post-release defects from a Bug Tracking System (BTS) via linking bug reports to its bug-fixing changes. Files related to these bug-fixing changes are considered buggy. Otherwise, the files are labeled as clean. The second step is to collect the corresponding defect features to represent these files. Instances with features and labels are used to train ML classifiers. Finally, trained models are used to predict files in a later version as buggy or clean.

Following Jiarpakdee et al. [6], this paper also focuses on file-level SDP.

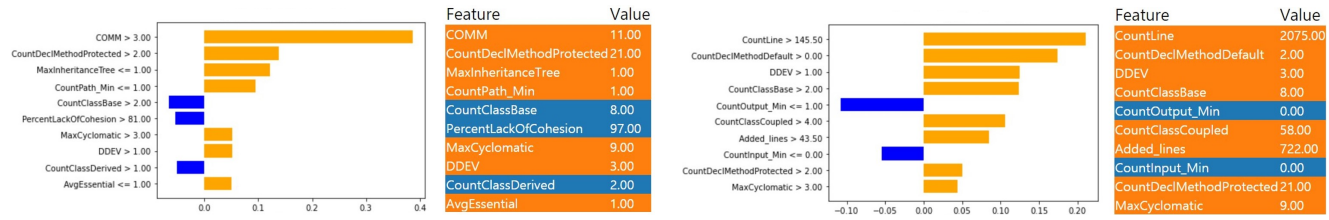
B. Model-agnostic Explanation Generation Techniques

Model-agnostic techniques were originally introduced to explain the prediction of black-box AI/ML algorithms by identifying the contribution that each metric has to the prediction of an instance according to a trained model [27]. *LIME* [7] and *BreakDown* [8], [9] are two state-of-the-art model-agnostic explanation techniques.

LIME [7] mimics a black-box model it aims to explain. To generate an explanation of an instance, *LIME* follows four major steps. First, it creates synthetic instances around the instance to be explained. Then, it generates predictions of all the synthetic instances generated in the step above. After that, it creates a local regression model with the synthetic instances and their predictions made in the step above. Finally, using the regression model, *LIME* ranks the contribution of each metric to the predictions aligning with the black-box model. Since *LIME* randomly generates instances to construct local regression models, the generated explanations are different when they re-generate explanations for the same instance. To mitigate this limitation, Jiarpakdee et al. [6] proposed *LIME-HPO*, which uses a differential evolution algorithm to find an optimal value of the number of randomly generated instances for the local regression models of original *LIME*. *LIME-HPO* was shown to generate stable explanations for the same instances when re-generating explanations. *BreakDown* [8], [9] measures the additive contribution of each feature of an instance sequentially, summing up to the final black-box prediction result. In our study, we used the ag-break version of the *BreakDown* technique, which works for non-additive models following Jiarpakdee et al. [6].

Jiarpakdee et al. [6] are the first to leverage model-agnostic techniques to generate explanations of a prediction, which refer to an explanation of why the SDP model predicts each file as a defective file. The techniques define explanations as a list of ordered features. The authors explore the usefulness of explanations generated by these techniques in answering three types of why questions, i.e., Property-contrast questions (e.g., why file A is defective rather than clean?), Object-contrast questions (e.g., why file A is defective, while file B is clean?), and Time-contrast questions (e.g., why was file A not classified as defective in version 1.2, but was subsequently classified as defective in version 1.3?). In this work, we empirically evaluate the reliability and stability of

[1]https://github.com/shinjh0849/stability_of_XDP.git



(a) Explanation with LR-based SDP model

(b) Explanation with DT-based SDP model

Fig. 1: Explanations from *LIME-HPO* for a buggy file *"ActiveMQConnection.java"* in activemq-5.0.0 on two models.

model-agnostic explanation techniques (i.e., *LIME-HPO* and *BreakDown*) on SDP models under different settings.

C. Motivation Example

In this section, we introduce an example to illustrate the challenge of explanations generated by a model-agnostic technique, i.e., *LIME-HPO*, which motivates us to further explore the reliability and stability of these techniques.

Figure 1 shows the explanations generated by *LIME-HPO* for file *"ActiveMQConnection.java"* with different SDP models (i.e., LR in Figure 1(a) and DT in Figure 1(b)) from version 5.0.0 of project ActiveMQ. The figures list the ranking of features that contribute to the prediction, i.e., explanations of the prediction. Figures on the left side are the probability and explanation of features that contribute to a prediction. On the right side, the figure depicts the actual value of the feature. For example, in Fig. 1(a), "COMM" contributes 0.39 buggy-prone because the value is 11, which is over 3. The orange color shows that a feature contributes to predicted as buggy and blue shows it contributes to predicted as clean. Although under both SDP models, the file is predicted as buggy, the generated explanations are significantly different. Specifically, among the ten features selected by *LIME-HPO* on the LR-based SDP model, only two were also selected on the DT-based SDP model, i.e., "DDEV" and "MaxCylomatic". However, "DDEV" and "MaxCylomatic" have different ranks. Such different explanations can affect users' understanding of the answers to the three types of why questions described in II-B. In this work, we use hit_rate and $rank\_diff$ proposed in [6] (details are in Section III-F) to evaluate the stability of the generated explanations.

Motivated by this example, in this work, we perform a comprehensive assessment and in-depth analysis of the state-of-the-art model-agnostic explanation generation techniques, i.e., *LIME-HPO* and *BreakDown* on SDP models with different settings. Note that the goal of this study is to evaluate the stability of a model-agnostic technique against itself under different SDP settings, not to evaluate the accuracy of the generated explanations or compare one model-agnostic technique against another.

III. Empirical Study Setup

This section describes our experiment method for evaluating the reliability and stability of model-agnostic explanation generation techniques, i.e., *LIME-HPO* and *BreakDown*, on SDP models in various settings. Note that, to remove potential bias, we conduct our experiments by strictly following the experiment process described in Jiarpakdee et. al [6], including using a fixed random seed for LIME (i.e., *LIME-HPO*), applying AutoSpearman to mitigate collinearity, tuning parameters of the examined ML algorithms with AUC, etc. By dropping irrelevant and correlated features using AutoSpearman and using the top-10 features, we can say that the comparison of generated features is in regards to the strong features which intuitively should be more stable than the weak features.

A. Research Questions

To achieve the mentioned goal, we have designed experiments to answer the following research questions regarding the reliability and stability of each studied model-agnostic explanation generation technique (i.e., *LIME-HPO* and *BreakDown*) under different SDP settings:

RQ1: *Are the generated explanations from the same technique consistent under different data sampling techniques?*

RQ2: *Are the generated explanations from the same technique consistent under different ML classifiers?*

RQ3: *Are the generated explanations from the same technique consistent under cross-version SDP scenarios?*

In RQ1, we investigate whether a model-agnostic technique-based explanation technique can generate consistent explanations for instances under the SDP model with applied different data sampling techniques. In RQ2, we examine whether a model-agnostic explanation technique can generate consistent explanations for instances under the SDP model trained with different ML classifiers. Following Jiarpakdee et al. [6], we examine six widely used ML classifies (details are in Section III-D). In RQ3, we explore whether a model-agnostic explanation tool can generate consistent explanations for the instances under the SDP models trained on different history releases from the same project, i.e., prediction of fixed target version with different versions as a training set.

B. Experiment Data

In this paper, to avoid potential bias introduced by experiment data, we reuse the same defect data from Jiarpakdee et al. [6], which comprises 32 releases that span 9 open-source software systems. Table I shows the statistical information of the dataset. We also reuse the same software metrics used in Jiarpakdee et al. [6] for building SDP models. In total, 65

TABLE I: Subjects studied in this work

Project	#Files	#KLOC	Bug rate	Studied Releases
ActiveMQ	1.8K-3.4K	142-299	6%-15%	5.0,5.1,5.2,5.3,5.8
Camel	1.5K-8.8K	75-383	2%-18%	1.4,2.9,2.10,2.11
Derby	1.9K-2.7K	412-533	14%-33%	10.2,10.3,10.5
Groovy	0.7K-0.9K	74-90	3%-8%	1.5.7,1.6.0.b1,1.6.0.b2
HBase	10K-18K	246-534	20%-26%	0.94,0.95.0,0.95.2
Hive	14K-27K	287-563	8%-19%	0.9,0.10,0.12
JRuby	0.7K-16K	105-238	5%-18%	1.1,1.4,1.5,1.7
Lucene	0.8K-28K	101-342	3%-24%	2.3,2.9,3.0,3.1
Wicket	16K-28K	109-165	4%-7%	1.3.b1,1.3.b2,1.5.3

software metrics along 3 dimensions are used, i.e., 54 code metrics (describe the relationship between properties extracted from source code and software quality), 5 process metrics (describe the relationship between the development process and software quality), and 6 human metrics (describe the relationship between the ownership of instances and software quality). Note that, Jiarpakdee et al. [6] have applied AutoSpearman [28] to remove irrelevant and correlated metrics before experiments. As a result, only 22-27 of the 65 metrics were used in the experiments. We follow the same process in this study to avoid any potential bias introduced by data pre-processing.

C. Studied Data Sampling Techniques

In this study, we examine the consistency of an explanation generation tool under five widely used data sampling methods, which are shown as follows.

- **Cluster Centroids [29]**: performs an under-sampling by using centroids as the new majority samples made by k-means clusters.
- **Repeated Edited Nearest Neighbors (RENN) [30]**: applies the nearest-neighbor algorithm to edit the samples by removing instances that are not similar to their neighbors.
- **Random under-sampling (RUS) [31]**: randomly picks samples from the majority class to match the minority class.
- **Random over-sampling (ROS) [32]**: over-samples the minority class by picking random samples with replacement.
- **SMOTE [32], [33]**: is the synthetic minority over-sampling technique (*SMOTE*). This method creates synthetic examples of the minority class rather than over-sampling with replacements.

Researchers have widely used all the above data sampling techniques in software defect prediction tasks to solve the data imbalance issue in SDP datasets [34]–[40]. In this work, we use the implementations of these data sampling techniques from the widely used imbalanced-learn Python library [41].

D. Studied SDP Models

Jiarpakdee et al. [6] showed that the model-agnostic techniques can be applied to many ML classifiers for explanation generation tasks. In this study, we use the same six ML classifiers mentioned in Jiarpakdee et al. [6] to build the SDP models, i.e., Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), Averaged Neural Network (AVNNet), Gradient Boosting Machine (GBM), and Extreme Gradient Boosting Tree (xGBTree). In this work, we used the implementation of the above six ML classifiers developed in the Scikit-learn library [42] and xgboost[2] as the implementation of Jiarpakdee et al [6] were not publicly available. Note that we have also tuned each of the six classifiers with its parameters and used the ones that can achieve the best AUC values to build prediction models in our experiments, results show that we can achieve similar performance as reported in Jiarpakdee et al [6].

E. Studied SDP Scenarios

For building SDP, one can choose different history versions as the training data, which we call cross-version SDP [43]–[45]. In this study, we also investigate the consistency of explanations generated by the same approach under the cross-version SDP scenario. Specifically, to perform a cross-version SDP scenario, for each project, we use its latest version as the test version and randomly select two earlier versions as the training data to build SDP models respectively. We then compare the generated explanations for the test data by using the two SDP models built on the two different training datasets.

F. Evaluation Measures

In this work, given a model-agnostic explanation generation technique (i.e., *LIME-HPO* and *BreakDown*), we use the following two metrics to evaluate the consistency of two explanations generated by it under two different SDP models, i.e., hit_rate and $rank\_diff$, which are proposed in [6].

Hit_rate is the percentage of features that match between the two explanations (i.e., a set of ranked features). Jiarpakdee et al. [6] leveraged the top-10 features ranked by model-agnostic techniques as the explanation to interpret the prediction results. If N ($N >= 0$ and $N <= 10$) out of the ten features are found in two explanations generated under two different SDP models, the value of hit_rate between these two explanations is $\frac{N}{10}$. Hit_rate indicates how similar the two explanations are without considering the ranking orders of features in the explanations. The higher the hit_rate, the better the consistency of an explanation generation technique is. In our experiments, we use the top-10 features for *LIME* and *BreakDown* to calculate the hit_rate as used in Jiarpakdee et al. [6].

Since the hit_rate does not consider the order of features in the explanations, we also use $rank\_diff$ (introduced in Jiarpakdee et al. [6]), which compares two explanations by using the orders of features in the explanations. Specifically, $rank\_diff$ measures the average difference of feature rankings between two explanations. For instance, if a feature is ranked Mth and Hth in two different explanations, the ranking difference of it is $abs\,(M - H)$. $rank\_diff$ is reported as the

[2]https://xgboost.readthedocs.io/en/latest/python/index.html

TABLE II: Average hit_rate and $rank\_diff$ of techniques with and without applying data sampling techniques.

Data sampling techniques	LIME-HPO		BreakDown	
	hit_rate	$rank\_diff$	hit_rate	$rank\_diff$
Cluster Centroids	0.577	5.564	0.655	4.725
Repeated Edited NN	0.608	5.328	0.692	4.292
Random under-sampling	0.574	5.655	0.686	4.382
Random over-sampling	0.641	4.892	0.769	3.505
SMOTE	0.632	4.948	0.762	3.577
Average	0.606	5.277	0.713	4.096

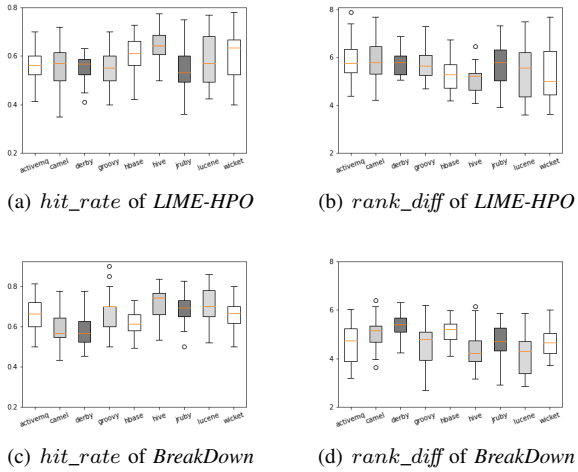
average ranking difference of all features in two explanations. If a feature is not in the ranking, the difference is set to top-N. Higher $rank\_diff$ means more different the explanations are by rankings. The range of the $rank\_diff$ is from zero (all features match all ranking orders) to the number of total features considered in the explanation, i.e., 10 (all features don't appear in the top 10). The smaller the $rank\_diff$, the more consistent an explanation generation technique is.

IV. Results and Analysis

A. RQ1: Explanation Consistency Under Different Data Sampling Techniques

Approach: To investigate the consistency of the generated explanations of a model-agnostic technique under different data sampling approaches, we rebuild the SDP models (i.e., each of the classifiers listed in Section III-D in the within-version SDP scenario) from Jiarpakdee et al. [6] with applying different data sampling techniques on the training data. Following Jiarpakdee et al. [6], we also use the out-of-sample bootstrap validation technique to create the training and test data on each version of each project listed in Table I. On the same test dataset, we run both *LIME-HPO* and *BreakDown* on SDP models with and without applying data sampling techniques to generate explanations for each test instance. We use the hit_rate and $rank\_diff$ to evaluate the consistency of explanations generated by a model-agnostic technique. In total, we have 60 runs on each project, i.e., 6 classifiers * 5 data sampling * 2 options (with or without sampling), for both *LIME-HPO* and *BreakDown*. We report the average values of hit_rate and $rank\_diff$ of explanations generated by the same model-agnostic technique under SDP models with or without different data sampling techniques applied.

Result: Table II shows the average hit_rate and $rank\_diff$ of explanations generated from the same model-agnostic technique with and without applying each data sampling technique. We take Cluster Centroids as an example to illustrate the detailed distributions of hit_rate and $rank\_diff$ on each project, which is shown in Figure 2. Overall, as we can see from the figure, explanations generated by both *LIME-HPO* and *BreakDown* are inconsistent on SDP models with and without applying Cluster Centroids, we have also observed a similar trend in other data sampling techniques. Specifically, the average hit_rate values of *LIME-HPO* and *BreakDown* range from 0.574 (using Random under-sampling) to 0.641 (using Random over-sampling) and 0.655 (using Cluster Centroids) to 0.769 (using Random over-sampling), respectively, which implies almost 40% and 29% of the features in the generated explanations of *LIME-HPO* and *BreakDown* are different with and without data sampling techniques applied.



(a) hit_rate of *LIME-HPO* (b) $rank\_diff$ of *LIME-HPO*

(c) hit_rate of *BreakDown* (d) $rank\_diff$ of *BreakDown*

Fig. 2: The distributions of hit_rate and $rank\_diff$ on each project before and after applying Cluster Centroids.

Regarding $rank\_diff$, on average, 5 out of the 10 features in the explanations from *LIME-HPO* and 4 out of 10 features from *BreakDown* have different ranks, which implies on average 50% and 40% of features in the explanations generated by *LIME-HPO* and *BreakDown* have a different order under SDP models with and without data sampling applied.

In addition, we have also checked that for both *LIME-HPO* and *BreakDown*, 100% of test instances have different feature orders with and without applying data sampling techniques. From these observations, we can see that explanations generated by *LIME-HPO* and *BreakDown* are inconsistent when data sampling is applied, which makes them unstable.

> Both *LIME-HPO* and *BreakDown* are inconsistent when data sampling is applied. On average, 40% of the explanations from *LIME-HPO* and 29% from *BreakDown* have different rankings. In addition, around 50% and 40% of explanations by *LIME-HPO* and *BreakDown* have different orders.

B. RQ2: Explanation Consistency Under Different Classifiers

Approach: Following Jiarpakdee et al. [6], we use six widely-used ML classifiers as our experiment subjects (details are in Section III-D). Note that, to avoid potential bias, we do not apply any data sampling technique in RQ2. For each classifier, we reuse the process described in Jiarpakdee et al. [6] to create the training and test data. We use the Logistic Regression (LR) based SDP model as the baseline as suggested in [6] for the comparison. On the same test dataset, we run a model-agnostic technique on both the baseline (LR-based SDP model) and each of the other five examined classifiers, i.e., AVNNet, DT,

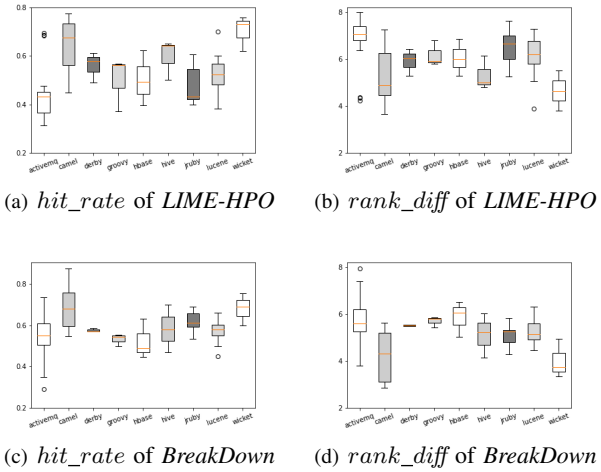
TABLE III: Average hit_rate and $rank\_diff$ of the explanations generated by *LIME-HPO* and *BreakDown* with SDP models with different classifiers.

Classifier	LIME-HPO		BreakDown	
	hit_rate	$rank\_diff$	hit_rate	$rank\_diff$
AVNNet	0.613	5.325	0.681	4.339
DT	0.515	6.185	0.609	5.241
GBM	0.559	5.714	0.638	4.826
RF	0.557	5.712	0.649	4.739
XGB	0.570	5.564	0.641	4.778
Average	0.563	5.700	0.644	4.785

GBM, RF, and xGBTree, to generate explanations for test instances. When different ML models are applied, prediction results of the same instance vary as buggy or clean. So, we only consider instances that have the same predicted results in both compared defect predictors for a fair comparison. To measure the consistency, we use hit_rate and $rank\_diff$ to evaluate *LIME-HPO* and *BreakDown* on different classifiers. We report the average values of hit_rate and $rank\_diff$ across all the experiment projects when comparing two classifiers.

Result: Table III shows the average hit_rate and $rank\_diff$ of the two explanation generation tools on different ML classifiers. Overall, both *LIME-HPO* and *BreakDown* generate inconsistent explanations between different ML classifiers. For *LIME-HPO*, the average hit_rate on these projects ranges from 0.515 (i.e., DT) to 0.613 (i.e., AVNNet), which means around 44% of the features in *LIME-HPO*'s explanations are different when a different ML classifier is applied for SDP compared to LR based SDP model. *BreakDown* has a slightly higher hit_rate, around 36% of the features in *BreakDown*'s explanations are different when different ML classifiers are applied. In addition, all the $rank\_diff$ values of *LIME-HPO* and *BreakDown* are higher than 4 and our analysis further reveals that on average there are more than 5 and 4 features in the explanations generated by *LIME-HPO* and *BreakDown* that have different ranks under SDP models with different classifiers, which indicates 50% and 40% features in the generated explanations have different orders. Note that, because of the space limitation, we only show the results of experiments whose base model is LR, we have also used each of the studied ML classifiers as the base model, and we observe similar findings, which indicate *LIME-HPO* and *BreakDown* consistently generate inconsistent explanations when different classifiers are applied.

Both *LIME-HPO* and *BreakDown* generate inconsistent explanations under SDP models with different classifiers. Specifically, 44% of the features in *LIME-HPO* and 36% of the features in *BreakDown*'s explanations are different when different ML classifiers are applied. In addition, more than 50% and 40% of the features in the explanations generated by *LIME-HPO* and *BreakDown* have different orders when different ML classifiers are applied.



(a) hit_rate of *LIME-HPO* (b) $rank\_diff$ of *LIME-HPO*

(c) hit_rate of *BreakDown* (d) $rank\_diff$ of *BreakDown*

Fig. 3: The detailed distributions of hit_rate and $rank\_diff$ of *LIME-HPO* and *BreakDown* on each project under the cross-version SDP scenario.

C. RQ3: Explanation Consistency Under the Cross-Version Scenario

Approach: To investigate the consistency of the generated explanations of a model-agnostic technique under a cross-version SDP scenario, for each experiment project listed in Table I, we use its latest version as the test data, and we then randomly select two different versions from the same project as the training data to train two different SDP models. We run a model-agnostic technique under both models to generate explanations for test instances. We use the hit_rate and $rank\_diff$ to evaluate the consistency of explanations generated by the model-agnostic technique. Note that, in this study, we use six different classifiers (details are in Section III-D) and examine two model-agnostic techniques, i.e., *LIME-HPO* and *BreakDown*.

Result: Table IV shows the average hit_rate and $rank\_diff$ of *LIME-HPO* and *BreakDown* under the cross-version prediction scenario. Figure 3 shows the detailed distributions of hit_rate and $rank\_diff$. As we can see from the results, the hit_rate values of both *LIME-HPO* and *BreakDown* are higher than 0.4 on each project. On average, the hit_rate is 0.518 across all the projects for *LIME-HPO*, which means around 50% of the generated explanations of *LIME-HPO* are different under cross-version SDP. For *BreakDown*, we can see that its average hit_rate is 0.591, indicating that 41% of the generated explanations are different under cross-version SDP. In addition, we have further checked that the $rank\_diff$ of both *LIME-HPO* and *BreakDown* on most projects is higher than 5, which indicates around 50% of features in the generated explanations of both *LIME-HPO* and *BreakDown* have different orders under SDP models built on different versions.

TABLE IV: Average hit_rate and $rank\_diff$ of the explanations generated by *LIME-HPO* and *BreakDown* under different SDP scenario.

Prediction Scenario	LIME-HPO		BreakDown	
	hit_rate	$rank\_diff$	hit_rate	$rank\_diff$
Cross-Version	0.518	6.172	0.591	5.213

TABLE V: Studied JIT SDP data used in.

Project	Training Data		Testing Data	
	#commits	buggy rate	#commits	buggy rate
Openstack	9,246	11%	3,963	16%
Qt	19,312	8%	8,277	6%

Both *LIME-HPO* and *BreakDown* generate inconsistent explanations under cross-version SDP scenarios. Overall, 50% of features in the generated explanations of *LIME-HPO* and 41% of *BreakDown* are different. In addition, around 50% of features in the generated explanations of *LIME-HPO* and *BreakDown* have different orders under the cross-version SDP scenario.

V. Discussion

A. Stability of Explanations for Just-In-Time Defect Prediction

Recently, Jiarpakdee et al. [10] proposed *PyExplainer*, i.e., a local rule-based model-agnostic technique for explaining the prediction result of Just-In-Time (JIT) SDP models. *PyExplainer* has been proven to be more effective than *LIME* on data from two open-source projects, which is shown in Table V. In this paper, we have also conducted experiments to examine the stability of explanations generated by *PyExplainer*. To remove potential bias, we used the replication package provided by *PyExplainer* to run our experiments. Note that, since *PyExplainer* was only evaluated on code commits from two open-source projects, some of the SDP settings we examined for file-level SDP do not apply to *PyExplainer*, i.e., the cross-version SDP scenario. Thus, we only examine the stability of the generated explanations of *PyExplainer* under two SDP settings, i.e., different data sampling techniques (details are in Section V-A1) and different prediction classifiers (details are in Section V-A2).

Our experiment shows that *PyExplainer* achieves a considerable hit_rate in their generated explanations, however, suffers to maintain the stability in the ranking order of the generated explanations, i.e. more than 4 ranking differences in the top-10 generated explanations. Details are as follows.

1) Explanation Consistency under Different Data Sampling Techniques: To investigate the consistency of the generated explanations of *PyExplainer* under different data sampling approaches, we rebuild the two types of SDP models used in Jiarpakdee et al. [10], i.e., RF-based and LR-based SDP models, by applying different data sampling techniques on the training data. On the same test dataset, we run *PyExplainer* on SDP models with and without applying data sampling techniques to generate explanations for each test instance. In this work, we examine the five data sampling techniques listed in Section III-C. We use the hit_rate and $rank\_diff$ to evaluate the consistency of explanations generated by a *PyExplainer*. In total, we have 20 runs on each project, i.e., 2 classifiers * 5 data sampling * 2 options (with or without sampling), for *PyExplainer*. We report the average values of hit_rate and $rank\_diff$ of explanations generated by *PyExplainer* under SDP models with or without different data sampling techniques applied.

TABLE VI: Average hit_rate and $rank\_diff$ of the explanations generated by *PyExplainer* under SDP models with and without applying data sampling techniques.

Data sampling techniques	*PyExplainer*	
	hit_rate	$rank\_diff$
Cluster Centroids	0.791	4.253
Repeated Edited NN	0.807	4.063
Random under-sampling	0.799	4.228
Random over-sampling	0.795	4.328
SMOTE	0.810	4.102
Average	0.800	4.195

Table VI shows the average hit_rate and $rank\_diff$ of explanations generated from *PyExplainer* with and without applying each data sampling technique. Overall, the hit_rate and $rank\_diff$ values of *PyExplainer* under different data sampling techniques are around 0.8 and 4.2 respectively. Compared to *LIME* and *BreakDown*, *PyExplainer* is more stable regarding hit_rate, one of the possible reasons is the number of features in JIT SDP data is less than that of file-level SDP data, i.e., after applying AutoSpearman, the feature reduces from 28 to 15 on the JIT SDP dataset, and from 65 to 27 on the file-level SDP dataset respectively.

PyExplainer generates inconsistent explanations when data sampling is applied. On average, almost 20% of the features in the explanations are different when data sampling techniques are applied. In addition, around 42% of features in the explanations have different orders under any data sampling technique.

2) Explanation Consistency under Different Classifiers: Jiarpakdee et al. [10] examined two classifiers, i.e., RF and LR, when evaluating the performance of *PyExplainer*. To avoid potential bias, we reuse the same two classifiers for the replication package of *PyExplainer* in this experiment. Specifically, we use the LR-based SDP model as the baseline for the comparison. On the same test dataset, we run *PyExplainer* on both the baseline (LR-based SDP model) and RF-based SDP model, to generate explanations for test instances. We consider instances that have the same predicted results in both compared SDP models for a fair comparison. To measure the consistency, we use hit_rate and $rank\_diff$ to evaluate*PyExplainer* on these two different classifiers. We report the average values of hit_rate and $rank\_diff$ across the two experiment projects when comparing classifiers.

Table VII shows the average hit_rate and $rank\_diff$ of explanations generated from *PyExplainer* with LR-based and

TABLE VII: Average hit_rate and $rank\_diff$ of the explanations generated by *PyExplainer* under different SDP models.

Classifier	*PyExplainer*	
	hit_rate	$rank\_diff$
RF	0.776	4.503

RF-based SDP models. The hit_rate and $rank\_diff$ are 0.776 and 4.503 respectively suggesting around 22% feature difference and 45% feature ranking difference in the explanations generated by *PyExplainer* under the two different classifiers. Similar to the results shown in Section V-A1, compared to *LIME-HPO* and *BreakDown*, *PyExplainer* has better performance under different classifiers, which can be caused by the significantly different number of features in the JIT SDP dataset and file-level SDP dataset.

PyExplainer generates inconsistent explanations under different classifiers. On average, almost 22% of the features in the explanations are different and around 45% of the features in the explanations have different orders.

VI. User Case Study

Our experiment results in Section IV show that model-agnostic techniques generate different explanations for the same instance under different SDP scenarios. Yet, little is known about whether the difference in the generated explanations can affect developers' perception of the prediction of an instance, i.e., the reason for the prediction.

Specifically, we conducted a survey study of 14 practitioners (i.e., four PhD students and ten Master students) to investigate their perceptions of instance explanations generated by model-agnostic techniques under different SDP scenarios that are consistent. The years of their experience in software development based on Java varied from two to five years. Following [6], we conducted this survey as follows:

Survey objectives: The survey aimed to investigate whether instance explanations generated by model-agnostic techniques under different SDP scenarios and answer the why-questions (i.e., Property-contrast, Objective-contrast, and Time-contrast) affect developers' perceptions of prediction.

Survey design: We follow [6] to use instances from the releases 2.10.0 and 2.11.0 of the Apache Camel project as the experimental subjects to generate instance explanations. In addition, we also investigate three types of explanations examined in [6], i.e., Property-contrast explanation (e.g., the reason that a file was predicted as defective), Object-contrast explanation (e.g., why file A was predicted as buggy, while file B was predicted as clean?), and Time-contrast explanation (e.g., why a file was predicted differently in two subsequent versions?). Note that, for our case study, we randomly selected 10 files from both releases 2.10.0 and 2.11.0 for each of the three objectives. For each file, we randomly pick one explanation from each of the four SDP scenarios in Section IV. We also collect the default explanations generated in [6]. Thus, in total, each file has five different explanations. The survey consists of three sets of questions with respect to the three objectives of the study. All questions are closed-ended questions. For each file, we present its five different explanations and ask participants to label the level of impact caused by the difference between an explanation from four SDP scenarios (see Section IV) and the original explanation from [6]. Given a pair of explanations, participants were asked to answer the question: **to what degree do you think your understanding of the prediction is the same based on the two different explanations?** We labeled each level of the ordinal scales with words as suggested in [46], i.e., strongly disagree, disagree, neutral, agree, and strongly agree. The survey takes approximately 1 hour to complete and is anonymous.

Result analysis: To analyze the data, we first checked the completeness of the collected data (i.e., whether all questions were appropriately answered). We then summarised and presented key statistical results. **Our analysis shows that 85.7%, 78.5%, and 71.4% of the participants disagree that their understanding of a prediction of a file based on the explanations generated by model-agnostic techniques under different SDP scenarios is consistent regarding Property-contrast explanation, Objective-contrast explanation, and Time-contrast explanation.** We conclude that model-agnostic techniques can dramatically impact users' understanding of the generated explanations when different scenarios are applied to SDP models.

VII. Threats to Validity

Internal Validity. The main internal threat of our study is the limited number of model-agnostic techniques (i.e., *LIME* and *BreakDown*) that we explored. Due to this limitation, we can't generalize our results to all model-agnostic techniques in the file-level SDP discipline. However, in our future studies, we will explore more techniques and compare the results to *LIME* and *BreakDown*. Furthermore, in this paper, we described a detailed methodology and setup of the experiment and the data set used, allowing other researchers to contribute to our study or further explore the other unexplored techniques.

External Validity. Even though the data sets used in this work are well labeled based on ground truths, the number of the data sets is limited and makes it hard to generalize our results to other data sets and domains. Future work needs to further investigate the study on other data sets. Besides, all the experiment projects are Java projects, although they are popular projects and widely used in existing SDP studies, our findings may not be generalizable to commercial projects.

Construct Validity. To measure the consistency of explanations generated by the same model agnostic technique (i.e., *LIME* and *BreakDown*) under different SDP settings, we use the top-10 features in the explanations to calculate metrics hit_rate and $rank\_diff$ following Jiarpakdee et al. [6]. With a different number of features used, the hit_rate and $rank\_diff$ of the two explanations can be different, which could affect our findings. However, we find that *LIME* and *BreakDown*

generate inconsistent explanations regardless of the number of features used.

VIII. Related Work

Many efforts have been made to build explainable SDP models [2]–[4], [6], [47], [48]. Jiarpakdee et al. [5] conducted a qualitative survey that investigates developers' perceptions of SDP goals and their explanations. The results of their experiments showed that the majority of the respondents believed that SDP is very important and useful and *LIME* and *BreakDown* are ranked as the top two approaches among a list of explanation generation approaches, in terms of the usefulness, quality, and insightfulness of the explanations. Humphreys and Dam [2] proposed an explainable deep learning SDP model that exploits self-attention transformer encoders. By using self-attention transformer encoders, the model can disentangle long-distance dependencies and benefit from its regularizing effect. Jiarpakdee et al. [6] used *LIME* and *BreakDown* to generate explanations on file-level SDP models that show which metrics are associated with buggy predictions. Khanan et al. [4] proposed an explainable JIT-DP framework, *JITBot*, that automatically generates feedback for developers by providing risks, and explaining the mitigation plan of each commit. They used a random forest classifier for risk-introducing commit prediction and leveraged model-agnostic technique, i.e., *LIME*, to explain the prediction results. Pornprasit and Tantithamthavorn [47] proposed *JITLine*, which ranks defective lines in a commit for finer granularity. With *JITLine*, they are able to predict both defect-introducing commits and identify lines that are associated with the commit. They exploit Bag-of-Token features extracted from repositories and apply them to machine learning classifiers to calculate the defect density of each commit. Then, they use defect density scores to rank different lines of the commit as risky. Recently, Jiarpakdee et al. [10] proposed *PyExplainer*, i.e., a local rule-based model-agnostic technique for explaining the prediction result of Just-In-Time SDP models. Wattanakriengkrai et al. [3] proposed a framework called *LINE-DP*, which applies *LIME* on a file-level prediction model trained with code token features. The explanation generated from *LIME* will show which code tokens are introducing bugs in the file. Then they use these explanations to identify a line buggy if the line contains bug-prone tokens. Lundberg and Lee [48] proposed SHAP which is a model-agnostic technique that works similarly to *BreakDown*, however instead of using the greedy strategy, it uses game theory to calculate the contribution probability of each feature to the final prediction of the prediction model.

Reem [49] conducted the first study to manually check whether the explanations generated by *LIME* and *BreakDown* are the same as the root causes of the bugs for change-level SDP models. Their results showed that both *LIME* and *BreakDown* fail to explain the root causes of predicted buggy changes. Roy et al. [50] investigated and compared the disagreement of the explanations generated from *LIME* and *SHAP*. They investigated 10 different ML predictors when comparing the explanation techniques. In this work, we conduct an empirical study to analyze the reliability and stability of model-agnostic explanation generation techniques, i.e., *LIME* and *BreakDown* on SDP under various settings at file-level SDP. Note that our work doesn't focus on comparing generations from different explanation models such as [50], but compares the generated results that use different SDP settings, i.e., ML classifiers, data sampling, and prediction scenarios.

IX. Conclusion

In this paper, we investigate the reliability and stability of model-agnostic explanation generation techniques, i.e., *LIME-HPO* and *BreakDown*, under different SDP settings. Our experiments on 32 versions of SDP data from nine open-source projects show that neither *LIME-HPO* nor *BreakDown* can generate consistent explanations under different SDP settings. Our user case study further confirms that inconsistent explanations can significantly affect developers' understanding of the prediction results, which implies that the model-agnostic techniques can be unreliable for practical explanation generation under different SDP settings. Overall, with this study, we urge a revisit of existing model-agnostic techniques in software engineering and call for more research in explainable SDP toward achieving stable explanation generation.

In the future, we plan to examine the reliability and stability of model-agnostic techniques used in other software engineering tasks and explore more reliable explanation generation techniques for prediction tasks in the software engineering domain.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper.

References

[1] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 372–381.

[2] J. Humphreys and H. K. Dam, "An explainable deep model for defect prediction," in *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE, 2019, pp. 49–55.

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

[4] C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, and T. Sunetnanta, "Jitbot: an explainable just-in-time defect prediction bot," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1336–1339.

[5] J. Jiarpakdee, C. Tantithamthavorn, and J. C. Grundy, "Practitioners' perceptions of the goals and visual explanations of defect prediction models," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 2021, pp. 432–443. [Online]. Available: https://doi.org/10.1109/MSR52588.2021.00055

[6] 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*, 2020.

[7] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[8] A. Gosiewska and P. Biecek, "Ibreakdown: Uncertainty of model explanations for non-additive predictive models," *arXiv preprint arXiv:1903.11420*, 2019.

[9] M. Staniak and P. Biecek, "Explanations of model predictions with live and breakdown packages," *arXiv preprint arXiv:1804.01955*, 2018.

[10] C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam, "Pyexplainer: Explaining the predictions of just-in-time defect models," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 407–418.

[11] D. Rajapaksha, C. Tantithamthavorn, C. Bergmeir, W. Buntine, J. Jiarpakdee, and J. Grundy, "Sqaplanner: Generating data-informed software quality improvement plans," *IEEE Transactions on Software Engineering*, 2021.

[12] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.

[13] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 99–108.

[14] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 2013, pp. 279–289.

[15] 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," *TSE'12*, vol. 39, no. 6, pp. 757–773, 2012.

[16] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[17] Y. Gao and C. Yang, "Software defect prediction based on adaboost algorithm under imbalance distribution," in *Proceedings of the 2016 4th International Conference on Sensors, Mechatronics and Automation*, 2016.

[18] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 91–100.

[19] S. Wang, J. Wang, J. Nam, and N. Nagappan, "Continuous software bug prediction," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–12.

[20] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE'09*, 2009, pp. 78–88.

[21] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *ICSE'13*, 2013, pp. 432–441.

[22] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *FSE'11*, 2011, pp. 311–321.

[23] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE'08*, 2008, pp. 181–190.

[24] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *ICSE'07*, 2007, pp. 489–498.

[25] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *ICSE'13*, 2013, pp. 382–391.

[26] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised vs. supervised models," in *ESEM'17*, 2017, pp. 344–353.

[27] M. T. Ribeiro, S. Singh, and C. Guestrin, "Model-agnostic interpretability of machine learning," *arXiv preprint arXiv:1606.05386*, 2016.

[28] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "Autospearman: Automatically mitigating correlated software metrics for interpreting defect models," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 2018, pp. 92–103.

[29] H. Altınçay and C. Ergün, "Clustering based under-sampling for improving speaker verification decisions using adaboost," in *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, 2004, pp. 698–706.

[30] I. Tomek, "An experiment with the edited nearest-neighbor rule," *IEEE Trans. Syst., Man, Cybern.*, vol. 6, p. 448—452, 1976.

[31] R. S. Wahono, "A systematic literature review of software defect prediction," *Journal of Software Engineering*, vol. 1, no. 1, pp. 1–16, 2015.

[32] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Information and Software Technology*, vol. 58, pp. 388–402, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584914001591

[33] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[34] D. Rodriguez, I. Herraiz, R. Harrison, J. Dolado, and J. C. Riquelme, "Preliminary comparison of techniques for dealing with imbalance in software defect prediction," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, 2014, pp. 1–10.

[35] C. Pak, T. T. Wang, and X. H. Su, "An empirical study on software defect prediction using over-sampling by smote," *International Journal of Software Engineering and Knowledge Engineering*, vol. 28, no. 06, pp. 811–830, 2018.

[36] S. Feng, J. Keung, X. Yu, Y. Xiao, and M. Zhang, "Investigation on the stability of smote-based oversampling techniques in software defect prediction," *Information and Software Technology*, p. 106662, 2021.

[37] L. Chen, B. Fang, Z. Shang, and Y. Tang, "Tackling class overlap and imbalance problems in software defect prediction," *Software Quality Journal*, vol. 26, no. 1, pp. 97–125, 2018.

[38] F. Wang, J. Huang, and Y. Ma, "A top-k learning to rank approach to cross-project software defect prediction," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 335–344.

[39] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[40] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no. 2, pp. 201–230, 2012.

[41] 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*, vol. 18, no. 17, pp. 1–5, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-365.html

[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[43] Z. Xu, J. Liu, X. Luo, and T. Zhang, "Cross-version defect prediction via hybrid active learning with kernel principal component analysis," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 209–220.

[44] X. Yang and W. Wen, "Ridge and lasso regression models for cross-version defect prediction," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 885–896, 2018.

[45] S. Amasaki, "Cross-version defect prediction: use historical data, cross-project data, or both?" *EmSE'20*, pp. 1–23, 2020.

[46] A. G. Koru and H. Liu, "An investigation of the effect of module size on defect prediction using static measures," in *Proceedings of the 2005 workshop on Predictor models in software engineering*, 2005, pp. 1–5.

[47] C. Pornprasit and C. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," *arXiv preprint arXiv:2103.07068*, 2021.

[48] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st international conference on neural information processing systems*, 2017, pp. 4768–4777.

[49] R. Aleithan, "Explainable just-in-time bug prediction: Are we there yet?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 129–131.

[50] S. Roy, G. Laberge, B. Roy, F. Khomh, A. Nikanjam, and S. Mondal, "Why don't xai techniques agree? characterizing the disagreements between post-hoc explanations of defect predictions," in *ICSME NIER Track*, 2022.