Explorable Software Defect Prediction: Are We There Yet?

Jiho Shin, Reem Aleithan, Jaechang Nam, Junjie Wang, Song Wang, Member, IEEE

Abstract—Explaining the results of defect prediction models is practical but challenging to achieve. Recently, Jiarpakdee et al. [1] proposed to use two state-of-the-art model-agnostic techniques (i.e., LIME and BreakDown) to explain prediction results. Their study showed that model-agnostic techniques can achieve remarkable performance, and the generated explanations can assist developers to understand the prediction results. However, the fact that they only examined both LIME and BreakDown in a single defect prediction setting calls into question the consistency and reliability of model-agnostic techniques on defect prediction models under various settings.

In this paper, we set out to investigate the reliability and stability of explanation generation approaches based on model-agnostic techniques, i.e., LIME and BreakDown, on defect prediction models under different settings, e.g., data sampling techniques, machine learning classifiers, and prediction scenarios used when building defect prediction models. Specifically, we use both LIME and BreakDown to generate explanations for the same instance under various defect prediction models with different settings and then check the consistency of the generated explanations for the instance. We reused the same defect data from Jiarpakdee et al. in our experiments. The results show that both LIME and BreakDown generate inconsistent explanations under different defect prediction settings for the same test instances. These imply that the model-agnostic techniques are unreliable for practical explanation generation. In addition, our manual analysis shows that none of the generated explanations can reflect the root causes of the predicted defects, which further weakens the usefulness of model-agnostic based explanation generation. Overall, with this study, we urge a revisit of existing model-agnostic based studies in software engineering and call for more research in explainable defect prediction towards achieving reliable and stable explanation generation.

Index Terms—Empirical software engineering, software defect prediction, explanation generation

1 INTRODUCTION

Software Defect Prediction (SDP) models have been actively studied to allocate testing resources efficiently to reduce development costs. Most existing SDP models use various code and development metrics as features to classify a target code fragment as buggy or not. However, a major issue that SDP models face is that they lack actionable messages for the developers to act upon [2], making it very difficult for practical usage.

To address this issue, studies investigating explainable artificial intelligence (XAI) in the domain of defect prediction have been explored recently [3]–[7] but most of these approaches target global explanation does not provide a detailed interpretation of prediction results. Jiarpakdee et al. [1] proposed to use the model-agnostic methods, i.e., LIME [8] and BreakDown [9], [10] to generate instance explanation to explain the prediction of each target code fragment. The explanation is defined as a list of ordered features. Their experiments and use case studies showed that both LIME and BreakDown achieve promising performance and the generated explanations can assist developers by showing actionable guidance for practical usages.

However, in Jiarpakdee et al. [1], LIME and BreakDown were only examined on a single software defect prediction setting which leaves unanswered the more directly relevant question: Are model-agnostic techniques reliable and stable under defect prediction models with different settings? The answer to this question is critical. First, many studies conduct defect prediction under different settings. The explanations generated of model-agnostic techniques are expected to be consistent across different settings to make them reliable and stable. Second, we have seen many studies follow Jiarpakdee et al. [1] to use model-agnostic techniques for other tasks, e.g., defective line prediction [11], online buggy commit prediction [7], and software quality assurance planning [12], understanding the reliability and 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 and BreakDown) on software defect prediction models under different settings.
Specifically, we consider three different settings when building software defect prediction models, i.e., data sampling techniques, machine learning classifiers, and prediction scenarios. Data sampling techniques are used in software defect prediction studies [13–15] to solve the data imbalance issue. In this work, we experiment with five widely used sampling methods (details are in Section 3.3). Various machine learning classifiers, e.g., Logistic Regression (LR), Decision Tree (DT), and Random Forest (RF), etc., have been used to build defect prediction models [16–19]. In this work, we experiment with six common machine learning classifiers (details are in Section 3.4). Defect prediction includes two major scenarios, i.e., cross-version and cross-project defect prediction, in both scenarios, one can choose different versions of historical data to build the models. In this work, we also examine the reliability and stability of LIME and BreakDown on these two scenarios when using different versions of data to build the defect prediction model.

For our analysis, we reuse the same dataset from Jiarpakdee et al. [1], which contains 32 versions of defect data from nine large-scale open-source Java projects. We run both LIME and BreakDown to generate explanations for the same instances under defect prediction models with different settings and then check the consistency of the generated explanations for the instances. Our experimental results show that explanations generated by both LIME and BreakDown are significantly inconsistent when different settings are applied, which makes them unreliable to be used in practice. In addition, our manual analysis shows that none of the generated explanations can reflect the root causes of the predicted defects, which further weakens the usefulness of model-agnostic based explanation generation. Hence, contrary to the claim of Jiarpakdee et al. [1], our study suggests that model-agnostic techniques are neither reliable nor stable to be used for explanation generation for defect prediction. Overall, with this study, we urge to revisit of other explainable software analytics studies that adopt model-agnostic techniques and call for more research in explainable software defect prediction towards achieving consistent explanation generation.

This paper makes the following contributions:

- We perform the first study to analyze the reliability and stability of state-of-the-art model-agnostic based explanation generation techniques, i.e., LIME and BreakDown on software defect prediction.
- We examine the consistency of explanations generated by LIME and BreakDown under software defect prediction models with three typical settings, i.e., data sampling techniques, machine learning classifiers, and prediction scenarios.
- We show neither LIME nor BreakDown can generate consistent explanations and the generated explanations cannot reflect the root causes of the predicted defects. This makes model-agnostic techniques neither reliable nor stable to be used in practice. Thus, we urge a revisit of existing model-agnostic based studies in software engineering and call for more research in building reliable and stable explanation generation for software analytics.
- 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 2 presents the background and motivation of this study. Section 3 shows the experimental setup. Section 4 presents the evaluation results. Section 5 discusses open questions and the threats to the validity of this work. Section 6 presents the related studies. Section 7 concludes this paper.

2 BACKGROUND AND MOTIVATION

This section introduces the background of software defect prediction models and the explanation generation techniques studied in this work and our motivation example.

2.1 File-level Defect Prediction Models

The objective of a file-level defect prediction model is to determine risky files for further software quality assurance activities. [20–26]. A typical release-based file-level defect prediction 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) [22, 27]. 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 machine learning classifiers. Finally, trained models are used to predict files in a later version as buggy or clean.

Following Jiarpakdee et al. [1], this paper also focuses on file-level defect prediction.

2.2 Model-agnostic based 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 on the prediction of an instance according to a trained model [28]. LIME [8] and BreakDown [9], [10] are two state-of-the-art model-agnostic explanation techniques.

LIME [8] mimics a black-box model it aims to explain. To generate an explanation of an instance, LIME follows four major steps. It first 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. BreakDown [9], [10] 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.

1. https://doi.org/10.5281/zenodo.5425868
which works for non-additive models following Jiarpakdee et al. \cite{1}.

Jiarpakdee et al. \cite{1} are the first to leverage model-agnostic explanation techniques to generate instance explanations, which refer to an explanation of the prediction of defect prediction models. The techniques define explanations as a list of ordered features. In this work, we empirically evaluate the reliability and stability of model-agnostic explanation techniques on software defect prediction models with different settings.

2.3 Motivating Example

In this section, we introduce an example to illustrate the problem of explanations generated by a model-agnostic technique, i.e., LIME, which motivates us to further explore the reliability and stability of model-agnostic based explanation generation.

Figure 1 shows the explanations generated by LIME for file “ActiveMQConnection.java” with different software defect prediction models (i.e., LR in Figure 1a and DT in Figure 1b) 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. 1a “COMM” contributes 0.39 buggy-prone because the value is 11, which is over 3. The orange color shows that a feature contributes in predicted as buggy and blue shows it contributes in predicted as clean. Although under both software defect prediction models, the file is predicted as buggy, the generated explanations are significantly different. Specifically, among the ten features selected by LIME on the LR-based defect prediction model, only two were also selected on the DT-based defect prediction model, i.e., “DDEV” and “MaxCylomatic”. However, “DDEV” and “MaxCylomatic” have different ranks. With this much difference, the generated explanation is unreliable and hard to be trusted.

Motivated by this example, in this work we perform a comprehensive assessment and in-depth analysis of the state-of-the-art model-agnostic based explanation generation techniques, i.e., LIME and BreakDown on defect prediction models with different settings. Note that the goal of this study is to evaluate the reliability and stability of a model-agnostic technique against itself under different software defect prediction models, not to evaluate one model-agnostic technique against another.

3 Empirical Study Setup

This section describes our experiment method for evaluating the reliability and stability of model-agnostic based explanation generation techniques, i.e., LIME and BreakDown, on defect prediction models in various settings.

3.1 Research Questions

To achieve the mentioned goal, we have designed experiments to answer the following research questions regarding the reliability and stability of each of the two studied model-agnostic based explanation generation techniques (i.e., LIME and BreakDown):

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

Software defect data are often imbalanced \cite{14}, i.e., the buggy instances are much fewer than the clean ones. Resampling, which changes the distribution between the majority class and the minority class, is an effective way to mitigate the effects of imbalanced data \cite{13, 29}. Thus, applying data sampling techniques is a common step in defect prediction \cite{14}. However, in Jiarpakdee et al. \cite{1}, no data sampling techniques were applied. In RQ1, we investigate whether a model-agnostic technique based explanation tool can generate consistent explanations for the predicted buggy instances under the software defect prediction model with applied different data sampling techniques.

RQ2: Are the generated explanations from the same tool consistent under different machine learning classifiers?

To build accurate software defect prediction models, different machine learning classifiers have been used to build software defect prediction models, e.g., Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), etc. Researchers have examined that these machine learning classifiers achieve the best performance on different datasets or under different prediction scenarios \cite{16, 19}. In RQ2, we examine whether model-agnostic technique based explanation tools can generate consistent explanations for the predicted buggy instances under the software defect prediction model with applied different machine learning classifiers.

RQ3: Are the generated explanations from the same tool consistent under cross-version defect prediction scenarios?

Cross-version defect prediction is one type of within-project defect prediction \cite{30}, which is often used for projects that have sufficient history data, e.g., a project can have multiple releases. One can choose different
TABLE 1: Subjects studied in this work

| Project | #Files | #KLOC | Bug rate | Studied Releases |
|---------|--------|-------|----------|------------------|
| ActiveMQ | 18K-34K | 122-29 | 6%-15% | 2.0,2.1,2.2,3.5-5.8 |
| Camel | 1.5K-8.8K | 75-383 | 2%-18% | 1.4,2.9,2.10,2.11 |
| Derby | 1.9K-2.5K | 412-583 | 14%-33% | 10.2,10.3,10.5 |
| Groovy | 0.9K-0.9K | 74-90 | 3%-8% | 1.3,1.6,b1,1.6.0.b2 |
| Hive | 10K-18K | 246-354 | 20%-26% | 0.93,0.95,0.95.2 |
| Java | 1K-20K | 267-583 | 8%-19% | 0.9,10.0.12 |
| Ruby | 0.5K-10K | 105-238 | 5%-18% | 1.1,1.4,1.5.1,7 |
| Lucene | 0.8K-28K | 101-342 | 5%-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 |

history versions as the training data to build the models with different training data selection approaches [31]. In Jiarpakdee et al. [1], the defect prediction model was trained and tested on data from the same version (i.e., within-version defect prediction). In RQ3, we explore whether a model-agnostic based explanation tool can generate consistent explanations for the predicted buggy instances under the software defect prediction models trained on different history releases from the same project.

**RQ4: Are the generated explanations from the same tool consistent under cross-project prediction scenarios?**

In cross-project defect prediction (CPDP), the training and test datasets are from different projects. CPDP is designed for projects that do not have historical data [26]. Jiarpakdee et al. [1] did not examine the performance of model-agnostic based explanation in a cross-project scenario. In RQ4, we explore whether a model-agnostic based explanation tool can generate consistent explanations for the predicted buggy instances under the software defect prediction model trained on data from different projects.

### 3.2 Experiment Data

In this paper, to avoid potential bias introduced by experiment data, we reuse the same defect data from Jiarpakdee et al. [1], which comprises 32 releases that span 9 open-source software systems. Table 1 shows the statistical information of the dataset.

For building defect prediction models, we also reuse the same software metrics used in Jiarpakdee et al. [1]. In total, 65 software metrics across 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 development process and software quality), and 6 human metrics (describe the relationship between the ownership of instances and software quality). Table 2 shows the metrics used to build defect prediction models in this work. Note that, Jiarpakdee et al. [1] has applied AutoSpearman to remove irrelevant metrics 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 preprocessing.

### 3.3 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** [33]: performs an under-sampling by using centroids as the new majority samples made by k-means clusters.
- **Repeated Edited Nearest Neighbours (RENN)** [34]: applies the nearest-neighbour algorithm to edit the samples by removing instances that are not similar to their neighbours.
- **Random under-sampling (RUS)** [35–37]: randomly picks samples from the majority class to match the minority class.
- **Random over-sampling (ROS)** [36, 38]: over-samples the minority class by picking random samples with replacement.
- **SMOTE** [36, 39]: 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 prediction tasks [29, 40–45]. In this work, we use the implementations of these data sampling techniques from the widely used imbalanced-learn Python library [46].

### 3.4 Studied Defect Prediction Classifiers

Jiarpakdee et al. [1] showed that the model-agnostic techniques can be applied to many machine learning classifiers for explanation generation tasks. In this study, we use the same six machine learning classifiers mentioned in Jiarpakdee et al. [1] to build the defect prediction models. The details of these classifiers are as follows:

- **Logistic Regression**: is the baseline model used in Jiarpakdee et al. [1]. It is a statistical model that uses the logistic function for classifying binary dependent variables. Logistic Regression is still widely used in defect prediction due to its advanced performance despite its simplicity [47].
- **Decision Tree (DT)**: is a model that uses trees to observe which features affect a target class.
- **Random Forest (RF)**: utilizes ensemble learning, which is a technique that combines many classifiers to provide solutions to complex problems. A random forest algorithm consists of many decision trees and makes decisions via majority voting of multiple decision trees.
- **Averaged Neural Network (AVNNet)**: is a neural network model that uses models with different random numbers as seeds. It averages all the resulting models to make a prediction decision.
- **Gradient Boosting Machine (GBM)**: uses an additive model of a forward stage-wise fashion. It uses predictors, such as decision trees, to form an ensemble.
- **Extreme Gradient Boosting Tree (xGBTree)**: follows the gradient boosting principle. However, xGBTree uses a more regularized model to control over-fitting efficiently.

In this work, we used the implementation of the above six machine learning classifiers developed in the scikit-learn library [48] and xgboost [3]. Note that we have also tuned each of the six classifiers with its parameters and use the ones that

2. https://xgboost.readthedocs.io/en/latest/python/index.html
can achieve the best AUC values to build prediction models in our experiments, as suggested in [1].

3.5 Studied Defect Prediction Scenarios

Software defect prediction models can be categorized into within-project and cross-project models based on the source of the training and test datasets. In within-project defect prediction, one can choose different history versions as the training data to build the models, which we call cross-version defect prediction. In this study, we investigate the consistency of explanations generated by the same tool under the two following defect prediction scenarios.

3.5.1 Cross-Version Defect Prediction

The cross-version defect prediction scenario is one of the actively studied scenarios in within-project defect prediction [49]–[51]. In this paper, to perform a cross-version defect prediction 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 defect prediction models respectively.

3.5.2 Cross-project Defect Prediction

The cross-project defect prediction scenario is also another actively studied scenario in the defect prediction field [19], [52], [53]. To perform cross-version defect prediction, we randomly select one version from each project as the test dataset and then we randomly select two different versions from two different projects as the training data to build software defect prediction models, respectively.

For both cross-version and cross-project defect prediction, given a test dataset, we use two different defect models to predict bugs on it, and then we run a model-agnostic based explanation generation technique to generate explanations on files that are predicted as buggy under both models to check whether the generated explanations of the same tool are consistent. We iterate the random selection 10 times to avoid potential bias and report the average of the results.

3.6 Evaluation Measures

In this work, given a model-agnostic based explanation generation technique (i.e., LIME and BreakDown), we use the following two metrics to evaluate the consistency of two explanations generated by it under two different defect prediction models. They are hit_rate and rank_diff.

Hit_rate is the percentage of features that match between the two explanations (i.e., a set of ranked features).

For instance, Jiarpakdee et al. [1] leveraged the top-10 features ranked by model-agnostic techniques as the explanation to interpret the prediction results. If \( N \) \( (N \geq 0 \) and \( N < 10) \) out of the ten features are found in two explanations generated under two different software defect prediction 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 range of the hit_rate value is from 0.0 to 1.0. 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 hit_rate as used in Jiarpakdee et al. [1].

Note that, since hit_rate does not consider the order of features in the explanations, we also introduce rank_diff, 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 \( M \)th and \( H \)th in two different explanations, the ranking difference of it is \( \text{abs}(M - H) \). rank_diff is reported as the 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 better the consistency of an explanation generation technique is.

4 Results and Analysis

This section presents the experimental results and the answers to the research questions regarding the reliability and stability of model-agnostic techniques proposed in Section 3.1.
4.1 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 combine the defect prediction setting (i.e., each of the classifiers listed in Section 3.4 in the within-version defect prediction scenario used in Jiarpakdee et al. [1] with different data sampling techniques to build defect prediction models on each experimental project. We follow Jiarpakdee et al. to use the out-of-sample bootstrap validation technique to create the training and test data on each version of each project listed in Table 1. On the same test dataset, we run both LIME and BreakDown under Jiarpakdee et al.’s defect prediction model and its variant with data sampling techniques to generate explanations for test instances that are predicted as buggy in both models. We use the hit_rate and rank_diff to evaluate the consistency of explanations generated by LIME and BreakDown. 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 and BreakDown. We report the average values of hit_rate and rank_diff of explanations generated by the same model-agnostic technique under defect prediction models with or without different data sampling techniques applied. In this RQ, we examine two typical model-agnostic techniques, i.e., LIME and BreakDown.

Result: Table 3 shows the average hit_rate and rank_diff of explanations generated from the same model-agnostic technique before and after applying each data sampling technique. Figure 2 shows the detailed distribution of hit_rate and rank_diff on each project. Overall, both LIME and BreakDown generate inconsistent explanations on defect prediction models before and after applying different data sampling techniques. Specifically, the average hit_rate values of LIME 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 and BreakDown are different before and after data sampling techniques are applied.

Regarding rank_diff, on average, 5 out of the 10 features in the explanations from LIME and 4 out of 10 features
from BreakDown have different ranks, which implies on average 50% and 40% features in the explanations generated by LIME and BreakDown have a different order under defect prediction models before and after data sampling applied.

In addition, we have also checked that for both LIME and BreakDown, 100% of test instances have different feature orders before and after applying data sampling techniques. From these observations, we can see that explanations generated by LIME and BreakDown are inconsistent when data sampling is applied, which makes them unreliable and unstable.

Both LIME and BreakDown generate inconsistent explanations when data sampling is applied. On average, almost 40% of the features in the explanations generated by LIME and 29% from BreakDown are different when data sampling techniques are applied. In addition, around 50% and 40% of features in the explanations generated by LIME and BreakDown have different orders under any data sampling technique.

### 4.2 RQ2: Explanation Consistency Under Different Classifiers

**Approach:** To investigate the consistency of the generated explanations of the same model-agnostic technique under defect prediction models trained on different machine learning classifiers, we use the six widely-used machine learning classifiers as our experiment subjects (details are in Section 3.4). Note that, to avoid potential bias, we do not apply any data sampling technique in RQ2. For each classifier, we follow the process described in Jiarpakdee et al. [1] to create the training and test data. We use the LR-based software defect prediction model as the baseline as suggested in [1] for the comparison. On the same test dataset, we run a model-agnostic technique on both the baseline (LR-based defect prediction model) and each of the other five examined classifiers, i.e., DT, RF, AVNNet, GBM, and xGBoost, to generate explanations for test instances. When different machine learning models are applied, prediction results of the same instance vary as buggy or clean. So, we only consider instances that are predicted as buggy in both compared machine learning predictors. To measure the consistency, we use hit_rate and rank_diff to evaluate LIME 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. In this RQ, we also examine two model-agnostic techniques, i.e., LIME and BreakDown.

**Result:** Table 4 shows the average hit_rate and rank_diff of the two explanation generation tools on different machine learning classifiers. Overall, both LIME and BreakDown generate inconsistent explanations between different machine learning classifiers. For LIME, 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’s explanations are different when a different machine learning classifier is applied for defect prediction compared to LR based defect prediction model. BreakDown has a slightly higher hit_rate, around 36% of the features in BreakDown’s explanations are different when different machine learning classifiers are applied. In addition, all the rank_diff values of LIME 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 and BreakDown that have different ranks under software defect prediction 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 machine learning classifiers as the base model, and we observe similar findings, which indicates LIME and BreakDown consistently generate unreliable explanations when different classifiers are applied.

Both LIME and BreakDown generate inconsistent explanations under different classifiers. Specifically, on average, 44% of the features in LIME’s explanations and 36% of the features in BreakDown’s explanations are different when different machine learning classifiers are applied. In addition, more than 50% and 40% of the features in the explanations generated by LIME and BreakDown have different orders when different machine learning classifiers are applied.

### 4.3 RQ3: Explanation Consistency Under the Cross-Version Scenario

**Approach:** To investigate the consistency of the generated explanations of a model-agnostic technique under cross-version defect prediction scenario, for each experiment project listed in Table 1 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 software defect prediction models. We run the model-agnostic technique under both models to generate explanations for test instances that are predicted as buggy in both models. 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 3.4) and examine two model-agnostic techniques, i.e., LIME and BreakDown.

**Result:** Table 5 shows the average hit_rate and rank_diff of LIME and BreakDown under the cross-version prediction scenario and Figure 7 shows the detailed distributions of hit_rate and rank_diff. As we can see from the results, the hit_rate values of both LIME and BreakDown are higher than 0.4 on each project. On average, hit_rate is 0.518 across

| Classifier | LIME | BreakDown |
|------------|------|-----------|
|            | hit_rate | rank_diff | hit_rate | rank_diff |
| AVNNet     | 0.515   | 3.325     | 0.681    | 4.359     |
| DT         | 0.515   | 6.185     | 0.667    | 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    | 4.728     |
| Average    | 0.563   | 5.700     | 0.644    | 4.785     |
Fig. 3: The detailed distributions of hit rate and rank diff of LIME and BreakDown on each project under the cross-version defect prediction scenario.

Fig. 4: The detailed distributions of hit rate and rank diff of LIME and BreakDown on each project under the cross-project defect prediction scenario.

| Prediction Scenario | LIME hit rate | LIME rank diff | BreakDown hit rate | BreakDown rank diff |
|---------------------|---------------|----------------|-------------------|--------------------|
| Cross-Version       | 0.518         | 6.172          | 0.591             | 5.213              |
| Cross-Project       | 0.480         | 6.410          | 0.510             | 5.920              |

TABLE 5: Average hit rate and rank diff of the explanations generated by LIME and BreakDown under different defect prediction scenarios.

all the projects for LIME, which means around 50% of the generated explanations of LIME are different under cross-version defect prediction. For BreakDown, we can see that its average hit rate is 0.591, indicating 41% of the generated explanations are different under cross-version defect prediction.

In addition, we can see that the rank diff values both of LIME and BreakDown are around 6 on each project, which indicates around 50% of features in the generated explanations of both LIME and BreakDown have different orders under the cross-project defect prediction scenario.

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

4.4 RQ4: Explanation Consistency Under the Cross-Project Scenario

Approach: To investigate the consistency of the explanations generated by a model-agnostic technique under cross-project defect prediction scenario, we first randomly select one version from each experiment project as the test data, we then randomly select two different versions from two different projects respectively as the training data to build two defect prediction models. We run the model-agnostic technique to generated explanations for test instances that are predicted as buggy in both models. We then use the hit rate and rank diff to evaluate the consistency of explanations generated by the model-agnostic technique. In addition, we use each of the six studied classifiers to run the experiments. We repeat the above process 10 times for avoiding potential data selection bias. Thus, each project has 6*10 experiment runs for examining the consistency of the generated explanations. We examine the studied two model-agnostic techniques, i.e., LIME and BreakDown. Similar to other RQs, we use the average hit rate and rank diff of each run to measure the consistency.

Result: Table 5 shows the average hit rate and rank diff of the two explanation generation tools under the cross-project prediction scenario. Figure 5 presents the detailed distribution of hit rate and rank diff for LIME and BreakDown on each project. As we can see from the figures, the average hit rate of LIME is around 0.48 on each project, which meaning around 52% of features in the generated explanations of LIME are different under the cross-project defect prediction scenario. For BreakDown, the average hit rate is 0.51, indicating 49% features in the generated explanations are different under the cross-project defect prediction scenario.

In addition, we can see that the rank diff values both of LIME and BreakDown are around 6 on each project, which indicates 60% of features in the generated explanations of them have different orders under the cross-project defect prediction scenario.
TABLE 6: Average hit_rate and rank_diff of LIME and BreakDown with different N features selected.

| N   | hit_rate | rank_diff | hit_rate | rank_diff |
|-----|----------|-----------|----------|-----------|
| 5   | 0.722    | 2.974     | 0.681    | 2.279     |
| 8   | 0.571    | 4.146     | 0.686    | 3.446     |
| 10  | 0.606    | 5.277     | 0.713    | 4.096     |
| 15  | 0.713    | 6.788     | 0.776    | 5.229     |

Both LIME and BreakDown generate inconsistent explanations under the cross-project defect prediction scenario. Specifically, around 52% of the features in the generated explanations of LIME and 49% for BreakDown are different under the cross-project defect prediction scenario. In addition, 60% of features in the generated explanations have different orders.

5 Discussion

5.1 Impact of Top-N Features Used in LIME and BreakDown

Following Jiarpakdee et al. [1], in our experiments, we use top 10 features to generate explanations. However, as both the hit_rate and rank_diff can be affected by the number of features used (i.e., N), we further investigate whether our findings of LIME and BreakDown hold when different numbers of features are used.

For our analysis, we take our RQ1 (Section 4.1) as an example to show the impact of different N on the performance of LIME and BreakDown. We follow the same process as described in Section 4.1 to conduct the experiments with different values of N. We experiment N with four values, i.e., 5, 8, 10, and 15. For each project, we combine each of the six examined machine learning classifiers (Section 3.3) with each of the five data sampling techniques (Section 3.3), in total, there are 30 runs, we use the two tools to generate the explanations for models before and after applying a data sampling technique and further calculate the hit_rate and rank_diff values, finally, we average all the hit_rate and rank_diff values on each project for each of the two examined model agnostic tools. Table 6 shows the average hit_rate and rank_diff of LIME and BreakDown with different numbers of top-N features. Figure 5 shows the detailed distribution of hit_rate and rank_diff under a different number of top-N features.

As shown in Table 6, overall, with the increase of N, both the hit_rate and rank_diff increase, this is natural because increasing N will enlarge the search space of LIME and BreakDown, thus more matches will occur. We can see that when N equals to 5, 8, 10, and 15, around 48%, 43%, 39%, and 29% features in the generated explanations of LIME and 35%, 31%, 29%, and 22% of BreakDown are different. We have also revisited other three RQs (RQ2-RQ4) by using LIME and BreakDown with different N and we observe similar results, which indicates LIME and BreakDown always generates inconsistent explanations regardless of the setting of N.

5.2 Consistency Between the Explanations and the Root Causes of Predicted Bugs

Jiarpakdee et al. [1] used a set of ranked features as the explanation for a prediction. Their human-involved case study showed that 65% of the participants agree that model-agnostic techniques can generate the time-contrast explanation to answer the why-questions like Why was file A not classified as defective in version 1.2 but was subsequently classified as defective in version 1.3? However, it's still unknown whether the explanations are consistent with the root causes of the buggy instances. To do so, we use the LR-based defect prediction model from Jiarpakdee et al. [1] to predict buggy files on the latest version of each project, we then randomly select 10 correctly predicted buggy instances from each project, in total 90 instances and their explanations generated by both LIME and BreakDown are collected. For each instance, we trace the data labelling process used in Jiarpakdee et al. [1] to find the linked bug report(s) and we use both a report’s content and its corresponding patch(es) to summarize its root cause. Finally, we manually (independently by the authors) check whether the generated explanations from LIME and BreakDown are consistent.

Our manual analysis shows that none of the explanations generated by these tools can reflect the ground truth root causes of the predicted buggy instances. For example, the file "ActiveMQConnection.java" showed in Figure 1 was labelled as buggy because of two bug reports, i.e., AMQ-1758 and AMQ-1646, which are caused by an incorrect variable usage and an incorrect condition respectively. However, as shown in Figure 1, the generated explanations are the ordered features and their numerical values, which are unrelated to the root causes, e.g., logic errors, missing API usages, syntax errors, functional errors, etc.

This result is natural and expected because features used in Jiarpakdee et al. [1] are all high-level software metrics.
which can only capture the overall statistical characteristics of software programs. Our analysis confirms with these features, both LIME and BreakDown cannot generate explanations that can reflect the root causes of buggy instances, which is unreliable to be used in practice.

5.3 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 defect prediction 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 labelled based on ground truths, the number of the data sets is limited and makes it hard to generalize our results on 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 software bug prediction 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 defect prediction settings, we use top 10 features in the explanations to calculate metrics hit_rate and rank_diff following Jiarpakdee et al. [1]. With different number of features used, the hit_rate and rank_diff of two explanations can be different, which could affect our findings. However, as we shown in Section 5.1, LIME and BreakDown ways generates inconsistent explanations regardless of the number of features used.

6 Related Work on Explainable Defect Prediction

As analytical modelling advances in the software engineering domain, the lack of explainability of analytical models becomes more problematic. Recent studies show the importance and need for such explanations [54]. Even more, as Dam et al., Lewis et al., Menzies and Zimmermann emphasize, these analytical model explanations need to be actionable to provide the most value and practical use to both practitioners and software engineers [2, 6, 54, 55].

Many efforts have been done to build explainable software defect prediction models [1, 8-9, 7, 56, 57]. Jiarpakdee et al. [7] conducted a qualitative survey that investigates developers’ perception of defect prediction goals and their explanations. The results of their experiments showed that majority of the respondents believed that software defect prediction 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 [8] proposed an explainable deep learning defect prediction model which exploits self-attention transformer encoders. By using self-attention transformer encoders, the model can disentangle long-distance dependencies and benefit from its regularizing effect. Also, they can normalize correlations that are inversely proportional to the prediction for more useful data. Jiarpakdee et al. [1] used LIME and BreakDown to generate explanations on file-level defect prediction models that show which metrics are associated with buggy predictions. Khanan et al. [5] 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 Random forest classifier for risk introducing commit prediction and leveraged model-agnostic technique, i.e. LIME, to explain the prediction results. Pornpraisit and Tantithamthavorn [7] 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 on 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. Wattanakriengkrai et al. [4] 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. Lundber and Lee [55] 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. Ribeiro et al. [57] proposed Anchors. Anchor is an extension of LIME generating rule-based explanations using decision rules. These rules are if-then rules (anchors) that have high confidence (at least 95% confidence and highest coverage if more than one rule has the same confidence value). In other words, only the selected features by anchor affect the final prediction.

Recently, Reem [58] conducted the first study to manually check whether the explanations generated by LIME and BreakDown are the same as the root cause of the bugs for change-level defect prediction models. Their results showed that both LIME and BreakDown fail to explain the root causes of predicted buggy changes. In this work, we conduct an empirical study to analyze the reliability and stability of model-agnostic based explanation generation techniques, i.e., LIME and BreakDown on software defect prediction under various settings at file-level defect prediction and we have also conducted the same manual analysis as Reem [58] on file-level defect prediction models and our manual analysis confirms the same finding, i.e., both LIME and BreakDown fail to explain the root causes of predicted buggy instances.
7 Conclusion
In this paper, we investigate the reliability and stability of model-agnostic based explanation generation techniques, i.e., LIME and BreakDown, under different software defect prediction settings. Our experiments on 32 versions of defect prediction data from nine open-source projects show that neither LIME nor BreakDown can generate consistent explanations under different defect prediction settings, thus both are unreliable to be used in practice. In addition, our manual analysis confirms that none of the generated explanations can reflect the root causes of the predicted defects. Thus, contrary to the claim of Jiarpakdee et al. [1], our study suggests that model-agnostic techniques are neither reliable nor stable to be used for explanation generation for defect prediction.

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.

References
[1] 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.
[2] 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.
[3] J. Thumpeys 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.
[4] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, “Predicting defective lines using a model-agnostic technique,” IEEE Transactions on Software Engineering, no. 01, pp. 1–1, sep 5555.
[5] C. Khanan, W. Luiewichana, 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.
[6] 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.
[7] C. Pornprasit and C. Tantithamthavorn, “Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction,” arXiv preprint arXiv:2103.07068, 2021.
[8] 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.
[9] A. Gostievskaya and P. Biecek, “Breakdown: Uncertainty of model explanations for non-additive predictive models,” arXiv preprint arXiv:1903.11420, 2019.
[10] M. Staniak and P. Biecek, “Explanations of model predictions with live and breakdown packages,” arXiv preprint arXiv:1804.01955, 2018.
[11] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, “Predicting defective lines using a model-agnostic technique,” arXiv preprint arXiv:2009.03612, 2020.
[12] D. Rajapaksha, C. Tantithamthavorn, C. Bergmeir, W. Buntine, J. Jiarpakdee, and J. Grundy, “Saqplanner: Generating data-informed software quality improvement plans,” IEEE Transactions on Software Engineering, 2021.
[13] 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.
[14] 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.
[15] T. Jiang, L. Tan, and S. Kim, “Personalized defect prediction,” in 2013 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013, pp. 321–324.
[16] 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.
[17] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework,” IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 485–496, 2008.
[18] 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.
[19] 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.
[20] A. E. Hassan, “Predicting faults using the complexity of code changes,” in ICSE ’09, 2009, pp. 78–88.
[21] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in PROMISE’07, 2007, pp. 9–9.
[22] F. Rahman and P. Devanbu, “How, and why, process metrics are better,” in ICSE’13, 2013, pp. 432–441.
[23] 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.
[24] R. Moser, W. Pedrycz, and C.ucci, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in ICSE’08, 2008, pp. 181–190.
[25] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, “Predicting faults from cached history,” in ICSE’07, 2007, pp. 489–498.
[26] J. Nam, S. J. Pan, and S. Kim, “Transfer defect learning,” in ICSE’13, 2013, pp. 382–391.
[27] 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.
[28] M. T. Ribeiro, S. Singh, and C. Guestrin, “Model-agnostic interpretability of machine learning,” arXiv preprint arXiv:1606.05386, 2016.
[29] S. Wang and X. Yao, “Using class imbalance learning for software defect prediction,” IEEE Transactions on Reliability, vol. 62, no. 2, pp. 432–443, 2013.
[30] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, “On the relative value of cross-company and within-company data for defect prediction,” EMSE’09, vol. 14, no. 5, pp. 540–578, 2009.
[31] S. Wang, J. Wang, J. Nam, and N. Nagappan, “Continuous software bug prediction,” 2021.
[32] 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.
[33] H. Altincay and C. Ergün, “Clustering based under-sampling for improving classifier 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.
[34] I. Tomek et al., “An experiment with the edited nearest-neighbor rule.” 1976.
[35] M. Bach, A. Werner, and M. Palt, “The proposal of undersampling method for learning from imbalanced datasets,” Procedia Computer Science, vol. 159, pp. 125–134, 2019.
[36] 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.
[37] R. S. Wahono, “A systematic literature review of software defect prediction,” Journal of Software Engineering, vol. 1, no. 1, pp. 1–16, 2015.
[38] C. X. Ling and C. Li, “Data mining for direct marketing: Problems and solutions.” in Kdd, vol. 98, 1998, pp. 73–79.
[39] 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.

[40] D. Rodriguez, I. Herrera, 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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

[47] B. Gheatra, S. McIntosh, and A. E. Hassan, “Revisiting the impact of classification techniques on the performance of defect prediction models,” in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1. IEEE, 2015, pp. 789–800.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] J. Guo, M. Rahimi, J. Cleland-Huang, A. Rasin, J. H. Hayes, and M. Vierhauser, “Cold-start software analytics,” in Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 142–153.

[53] A. Radford, R. Jozefowicz, and I. Sutskever, “Learning to generate reviews and discovering sentiment,” arXiv preprint arXiv:1704.01444, 2017.

[54] T. Menzies and T. Zimmermann, “Software analytics: So what?” IEEE Software, vol. 30, no. 4, pp. 31–37, 2013.

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

[56] M. T. Ribeiro, S. Singh, and C. Guestrin, “Anchors: High-precision model-agnostic explanations,” in Proceedings of the AAAI conference on artificial intelligence, vol. 32, no. 1, 2018.

[57] R. Aleethan, “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.