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 the 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 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.

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

https://github.com/shinjh0849/stability_of_XDP.git
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 a buggy file “ActiveMQConnection.java” in activemq-5.0.0 on two models.

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 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 classifiers (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

<table>
<thead>
<tr>
<th>Project</th>
<th>#Files</th>
<th>#KLOC</th>
<th>Bug rate</th>
<th>Studied Releases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveMQ</td>
<td>1.8K-3.4K</td>
<td>142-259</td>
<td>6%-15%</td>
<td>5.0,5.1,5.2,5.3,5.5,5.8</td>
</tr>
<tr>
<td>Camel</td>
<td>1.5K-8.8K</td>
<td>75-383</td>
<td>2%-18%</td>
<td>1.4,2.9,2.10,2.11</td>
</tr>
<tr>
<td>Derby</td>
<td>1.9K-2.7K</td>
<td>412-533</td>
<td>14%-33%</td>
<td>10.2,10.3,10.5</td>
</tr>
<tr>
<td>Groovy</td>
<td>0.7K-0.9K</td>
<td>74-99</td>
<td>3%-8%</td>
<td>1.5,7,16.0,16.1,6.0,16.2</td>
</tr>
<tr>
<td>HBase</td>
<td>10K-18K</td>
<td>246-534</td>
<td>20%-26%</td>
<td>0.94.0,0.950.0,0.952</td>
</tr>
<tr>
<td>Hive</td>
<td>14K-27K</td>
<td>287-563</td>
<td>8%-19%</td>
<td>0.9,0.10,0.12</td>
</tr>
<tr>
<td>JRuby</td>
<td>0.7K-16K</td>
<td>105-238</td>
<td>5%-18%</td>
<td>1.1,1.4,1.5,1.7</td>
</tr>
<tr>
<td>Lucene</td>
<td>0.8K-28K</td>
<td>101-342</td>
<td>3%-24%</td>
<td>2.3,2.9,3.0,3.1</td>
</tr>
<tr>
<td>Wicket</td>
<td>16K-28K</td>
<td>109-165</td>
<td>4%-7%</td>
<td>3.1,3.2,3.3,3.5,3.5</td>
</tr>
</tbody>
</table>

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 (AVNet), 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 [3] 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 dataset. 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. 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].

\[ \text{Hit}_\text{rate} = \frac{N}{M} \]

\[ \text{Hit}_\text{rate} \] indicates how similar the two explanations are without considering the ranking orders of features in the explanations. The higher the \text{hit}_\text{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 \text{hit}_\text{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 \text{M}th and \text{H}th in two different explanations, the ranking difference of it is abs(M - H). rank\_diff is reported as the
TABLE II: Average hit_rate and rank_diff of techniques with and without applying data sampling techniques.

<table>
<thead>
<tr>
<th>Data sampling techniques</th>
<th>LIME-HPO</th>
<th>BreakDown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>hit_rate</td>
<td>rank_diff</td>
</tr>
<tr>
<td>Cluster Centroids</td>
<td>0.577</td>
<td>5.564</td>
</tr>
<tr>
<td>Repeated Edited NN</td>
<td>0.608</td>
<td>5.328</td>
</tr>
<tr>
<td>Random under-sampling</td>
<td>0.574</td>
<td>5.555</td>
</tr>
<tr>
<td>Random over-sampling</td>
<td>0.641</td>
<td>4.892</td>
</tr>
<tr>
<td>SMOTE</td>
<td>0.632</td>
<td>4.948</td>
</tr>
<tr>
<td>Average</td>
<td>0.606</td>
<td>5.277</td>
</tr>
</tbody>
</table>

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.

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,
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.

Table III: Average hit_rate and rank_diff of the explanations generated by LIME-HPO and BreakDown with SDP models with different classifiers.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>LIME-HPO</th>
<th>BreakDown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>hit_rate</td>
<td>rank_diff</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>AVNNet</td>
<td>0.613</td>
<td>3.325</td>
</tr>
<tr>
<td>DT</td>
<td>0.515</td>
<td>6.185</td>
</tr>
<tr>
<td>GBM</td>
<td>0.559</td>
<td>5.714</td>
</tr>
<tr>
<td>RF</td>
<td>0.557</td>
<td>5.712</td>
</tr>
<tr>
<td>XGB</td>
<td>0.570</td>
<td>5.564</td>
</tr>
<tr>
<td>Average</td>
<td>0.563</td>
<td>5.700</td>
</tr>
</tbody>
</table>

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 4 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.
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 IV. 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 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 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.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>PyExplainer hit_rate</th>
<th>rank_diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF</td>
<td>0.776</td>
<td>4.503</td>
</tr>
</tbody>
</table>

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 [40], 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], [3], [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. Pomprasit 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


