An Empirical Study of Model-Agnostic Interpretation Technique for Just-in-Time Software Defect Prediction

Xingguang Yang[1,2], Huiqun Yu[1,3(✉)], Guisheng Fan[1(✉)], Zijie Huang[1], Kang Yang[1], and Ziyi Zhou[1]

[1] Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China
{yhq,gsfan}@ecust.edu.cn

[2] Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai 201112, China

[3] Shanghai Engineering Research Center of Smart Energy, Shanghai, China

Abstract. Just-in-time software defect prediction (JIT-SDP) is an effective method of software quality assurance, whose objective is to use machine learning methods to identify defective code changes. However, the existing research only focuses on the predictive power of the JIT-SDP model and ignores the interpretability of the model. The need for the interpretability of the JIT-SDP model mainly comes from two reasons: (1) developers expect to understand the decision-making process of the JIT-SDP model and obtain guidance and insights; (2) the prediction results of the JIT-SDP model will have an impact on the interests of developers. According to privacy protection laws, prediction models need to provide explanations. To this end, we introduced three classifier-agnostic (CA) technologies, LIME, BreakDown, and SHAP for JIT-SDP models, and conducted a large-scale empirical study on six open source projects. The empirical results show that: (1) Different instances have different explanations. On average, the feature ranking difference of two random instances is 3; (2) For a given system, the feature lists and top-1 feature generated by different CA technologies have strong agreement; However, CA technologies have small agreement on the top-3 features in the feature ranking lists. In the actual software development process, we suggest using CA technologies to help developers understand the prediction results of the model.

Keywords: Software defect prediction · Just-in-time · Classifier-agnostic interpretation · Model interpretation

1 Introduction

Software defects will cause errors and unexpected results of software system [28]. Although software quality assurance activities (such as code review, software

H. Gao and X. Wang (Eds.): CollaborateCom 2021, LNICST 406, pp. 420–438, 2021.
https://doi.org/10.1007/978-3-030-92635-9_25

testing, etc.) can effectively find and repair defects in software system. However, under the constraints of software testing resources and software release time, it is unrealistic for software developers to implement comprehensive quality assurance activities for software systems [43].

Software defect prediction is an effective method to improve the efficiency of SQA activities [21]. By accurately predicting the defect probability of the program modules in the software system, developers can design a reasonable test resource allocation plan, thereby reducing the cost of software development and improving the quality of the software.

Just-in-time software defect prediction (JIT-SDP) is a fine-grained defect prediction method, and its prediction entity is fine-grained code changes. JIT-SDP technology has the following three advantages [19]: (1) Fine granularity. Compared with coarse-grained file, function, or package, code changes contain fewer lines of code. Therefore, once a change is predicted to be defect-prone, developers can efficiently check the code of the change; (2) Immediately. Once the developer submits a code change, the defect prediction model can immediately predict the defect probability of the change. Therefore, developers still clearly remember the process of software development when checking the code for changes. (3) Traceability. There is usually only one developer for a code change. Therefore, project managers can easily find suitable developers to check and fix defect changes. Due to the importance of JIT-SDP for improving software quality, many companies have adopted JIT-SDP technology, such as Avaya [27], Blackberry [35], and Cisco [40].

Currently, JIT-SDP has received extensive research and attention. Most of the research focuses on the prediction performance of the JIT-SDP model [48], data annotation [8], feature extraction [23] and so on. However, few studies have focused on the interpretability of the JIT-SDP model. The interpretability of models is very critical in the field of software engineering [7]. In the research of JIT-SDP, the need for interpretability of prediction models mainly comes from two aspects: (1) Lack of trust. Developers may ask "Why the code change I submitted is predicted to be buggy". In a software analysis system, if the prediction model cannot provide an explanation for the prediction results, the developer may not be able to believe the prediction results of the model [7], which will hinder the application of the JIT-SDP model in actual software development. (2) Data privacy protection. In the application of the JIT-SDP model, if code changes submitted by a developer are often predicted as buggy by the model, the developer's job opportunities, salary, bonus, etc. may be affected. According to the Article 22 of the European Union's General Data Protection Regulation (GDPR) [31], if the data used in the decision-making model affects individuals or organizations, the decision-making model needs to provide an explanation.

Classifier-specific (CS) techniques can provide explanations by analyzing the internal structure of the model. However CS methods are not universal to all prediction models. Many complex classification models such as SVM, deep neural network and other black box models do not have specific model interpretation techniques [5,30]. In the JIT-SDP model, many black-box models have

high prediction performance, such as random forest [17], deep neural work [10], etc. Therefore, we propose to use three classifier-agnostic (CA) interpretation techniques to interpret the JIT-SDP model, namely Local *Local Interpretable Model-agnostic Explanations* (LIME) [32], *BreakDown* [9,37] and *SHapley Additive ExPlanations* (SHAP) [24]. These three CA technologies can provide explanations for each instance. The Sect. 3 of this paper describes the basic principles of these three CA technologies. The main contributions of this article are as follows:

1. We introduced three CA technologies LIME, BreakDown and SHAP to explain the prediction results of the JIT-SDP model.
2. We conducted an a large empirical research on six open source projects, and evaluated the distribution of the ranking differences of each feature in different instances;
3. For a given project, we evaluate the agreement of the feature ranking lists generated by the three CA technologies.

The organization structure of the rest of this paper is as follows: Sect. 2 introduces the work related to JIT-SDP and the interpretability of software defect prediction; Sect. 3 briefly introduces the basic principles of the three CA methods; Sect. 4 explains the experimental settings; Sect. 5 introduces the results and analysis of the experiment; Sect. 6 summarizes the research of this paper and explains the future work.

2 Related Work

2.1 Just-in-time Software Defect Prediction

Mockus and Weiss [27] first proposed the JIT-SDP technology. They use the attributes of the code change, such as the number of lines added or deleted, and the number of code modified subsystems, to predict the defect probability of the change, and they apply the prediction model to a 5ESS switch system software. Recently, Kamei et al. [19] conducted large-scale empirical research on six open source projects and five commercial projects. They use 14 change metrics to measure code changes. The results show that their prediction model can achieve 68% accuracy and 64% recall, and can identify 35% of buggy changes when investing 20% of the code inspection effort. Subsequently, Kamei et al. [17] studied the prediction performance of the JIT-SDP model in the cross-project scenario. They used random forest to build models and conducted research on 10 projects. The results show that the method of data merging and ensemble learning can improve the prediction performance of cross-project models. Yang et al. [48] compared the prediction performance of simple unsupervised models and supervised models. They found that in the effort-aware JIT-SDP, the simple unsupervised models are better than the state-of-the-art supervised models. Inspired by the work of Yang et al. [48], Huang et al. [11,12] further compared the prediction performance of supervised and unsupervised models in JIT-SDP.

They found that the unsupervised models have many context switches and high false alarms. To this end, they proposed a supervised model called CBS, which combines the EALR model proposed by Kamei et al. [19], and the LT model proposed by Yang et al. [48]. The results show that the proposed method is significantly better than other benchmark models in precision and F1 indicators. Besides, to improve the performance of the JIT-SDP model, researchers have also proposed many methods, such as multi-objective optimization algorithms [6,44], differential evolution algorithm [45,46], ensemble learning [47], deep learning [10], online learning [3], etc.

2.2 Explainability in Software Defect Prediction

The interpretability of software defect prediction models is a relatively new research area. Jiarpakdee et al. [15] first studied the participants' perceptions of the goal of the software defect prediction model. Research results show that 82% to 84% respondents believe that software defect prediction is useful in the three goals of optimizing test resource allocation, understanding defect-associated characteristics, and understanding the prediction results of prediction models. In addition, the results showed that LIME was the most popular model interpretation technique. Subsequently, Jiarpakdee et al. [14] empirically evaluated three classifier-agnostic interpretation techniques on 32 defect data sets, namely LIME-HPO, LIME, and BreakDown. The results show that it is necessary to use model-agnostic technology to explain the software defect prediction model, and the instance explanations have a high degree of overlap with the global explanations. Wattanakriengkrai et al. [42] proposed a framework called LINE-DP to identify defective lines of code. LINE-DP first uses the code token feature to build a file-level defect prediction model. Then for a file that is predicted to be defective, LINE-DP uses LIME to identify risky tokens. Finally, the code lines with risky tokens are predicted as defective code lines. Rajbahadur et al. [30] analyzed the impact of different model interpretation techniques on defect classifiers. They studied six CS technologies and two CA technologies, and conducted research on 18 software projects. The results show that the feature importance rank list generated by CA and CS methods do not always have a strong agreement. In addition, different CA methods have strong argeement in indicators top-1 overlap and top-3 overlap. To improve the agreement of feature ranks among CS methods, they recommend using the CFS method to remove correlated features. There are also some studies that analyze the impact of imbalance techniques [39] and the correlation of features [16] on model interpretation. Currently, there are relatively few studies on the interpretability of the JIT-SDP model. Pornprasit et al. [29] proposed a JIT-SDP method called JITLine. The method first extracts the tokens features from code changes and builds a commit-level prediction model; then, for each test instance, JITLine uses the model interpretation technology LIME to calculate the feature importance score of each token. Finally, the code lines in the commit are sorted according to the feature importance score.

3 Classifier-Agnostic Interpretation Technique

The interpretability or explainability of a prediction model refers to the degree to which human observers can understand the reasons why the model makes predictions [7,26]. There are two main ways to obtain the interpretability of a machine learning model [22]:

1. **Make the decision-making process of the machine learning model transparent and easy to understand.** This kind of model interpretation method requires the internal structure of the model to be transparent and easy to understand, so it is also called a classifier-specific (CS) interpretation technique. The white box model has a simple internal structure, so it can make the prediction process of the model transparent. For example, the decision-making process of a decision tree can be displayed in a visual tree structure. The path from the root node to the leaf node of the decision tree provides an explanation of the prediction result. Although the white-box model has high interpretability, it has poorer prediction performance than the black-box model.
2. **Provide an explanation for each prediction result of models.** This kind of model interpretation technology can interpret any machine learning algorithm, including complex black-box models (such as random forest, SVM, deep learning, etc.), so it is also called classifier-agnostic (CA) interpretation technology.

Just-in-time software defect prediction is a binary classification task. Most of the current researches use complex black-box algorithms (such as random forest [17], deep learning [10], etc.) to build classification models and obtain high prediction performance. To this end, this paper introduces three state-of-the-art CA technologies LIME [32], BreakDown [9,37], and SHAP [24] to explain the prediction results of the JIT-SDP model. This section details the specific processes of the three CA technologies.

3.1 LIME

LIME is a local surrogate model proposed by Ribeiro et al. [32]. The model uses an interpretable machine learning model (i.e., linear regression, decision tree etc.) to locally mimics the predictions of the black-box model. Therefore, a local surrogate model can be used to explain an individual instance. Assuming that $\hat{f}$ is a black-box model and x is an instance that needs to be explained, the loss function of the LIME method is shown in Eq. 1.

In the Eq. 1, the minimized loss function L reflects the closeness of the prediction results between the black-box model $\hat{f}$ and the proxy model g. G represents the set of interpretable machine learning algorithms; π defines the domain range when sampling around instance x; $Omega(g)$ represents the model complexity of the interpretable model g.

$$explanation(x) = \arg\min_{g \in G} L(\hat{f}, g, \pi_x) + \Omega(g) \tag{1}$$

The key steps of training the local surrogate model are as follows: (1) Perturb the target instance to generate new points; (2) Use the black-box model to predict the new points; (3) According to the closeness of new points to the target instance, set a weight for each point; (4) Use an interpretable machine learning algorithm to train a local surrogate model on the data set with weights. The local surrogate model trained in our experiment is ridge regression. Therefore, the coefficients of the regression model are used to measure the feature importance scores of a change.

3.2 BreakDown

BreakDown is a CA technology. For a target instance, BreakDown uses a greedy strategy to sequentially measure the importance scores of features. Assuming that x is a target instance with m features, $\hat{f}$ is a black box model, and the feature importance score generated by BreakDown satisfies Eq. 2, where $v(j, x)$ represents the feature importance score calculated by BreakDown for the instance on the j-th feature, $\hat{f}(x)$ represents the prediction result of the black-box model on the instance x, and v_0 represents the average value of the prediction results of the black-box model in the training set.

$$\hat{f}(x) = v_0 + \sum_{j=1}^{m} v(j, x) \tag{2}$$

The calculation process of the BreakDown method mainly includes the following four steps: (1) Use the black-box model to predict all the training data, and calculate the average of the prediction results; (2) For each feature, sequentially replace the value of a feature in the training data with the value of the target instance to construct new training data; (3) Use the black box model to predict the new training set. By analyzing the differences in the prediction results on the new data set, the most important features are identified, and the differences in the prediction results are used to measure the importance scores of the features; (4) Repeat the third step until the importance scores of all the features are calculated.

3.3 SHAP

SHAP calculates the contribution of each feature to the predicted result according to the coalition game theory. The model interpretation of SHAP is shown in Eq. 3, where g represents an interpretable model, m represents the size of the coalition, $z' \subseteq \{0, 1\}^m$ and Φ_j represents the Shapley value of the j-th feature, which is used to evaluate the importance score of the feature. In the study of JIT-SDP, since all the features of the target instance x exist, the coalition vector is a vector with all values of 1, so Eq. 3 can be simplified to Eq. 4.

$$g(z') = \Phi_0 + \sum_{j=1}^{m} \Phi_j z'_j \tag{3}$$

$$g(x') = \Phi_0 + \sum_{j=1}^{m} \Phi_j \tag{4}$$

4 Experimental Setup

This section introduces the experimental setup from three aspects: data sets, building classification models, and evaluation metrics. We aim to answer the following two research questions through experiments:

- RQ: What are the interpretation results of classifier-agnostic techniques on the JIT-SDP model?
- RQ2: For a given project, what is the difference in the feature importance rank lists generated by different CA methods?

The experiment is run on a workstation, and its configuration is *Intel(R) Xeon(R) Gold 5118 CPU @ 2.30 GHz*; *RAM 251 GB*.

4.1 Data Sets

The experiment uses public data sets widely used in JIT-SDP research [3,38]. The data set contains the defect data of ten open source projects, which have a development cycle of more than five years, rich historical data and a good defect-inducing changes rate. To extract the metrics values of changes in the projects and identify defective code changes, Cabral et al. [3] used *Commit Guru* [33] to collect the change-level defect data set of projects. Commit Guru is an automated defect data collection tool that can extract code changes from the *Git Hub* and use the SZZ algorithm [36] to identify buggy changes. Table 1 shows the basic information of ten projects, including the name of the project, the number of changes, the number of defective changes, the proportion of buggy changes, the development period of the project, and the programming language. Each instance in the data sets contains 14 metrics, which can be effectively used to build the JIT-SDP model [48]. Table 2 describes the names and definitions of 14 metrics. These metrics can be divided into five dimensions including diffusion, size, purpose, history, and experience. Due to the limitation of the length of the paper, the detailed introduction can refer to Kamei et al. [19]'s study.

4.2 Building Classification Models

Before building the classification model, we use a cross-validation method to divide the data sets of each project into a training set and a test set. K-fold cross-validation [25] randomly divides the data sets into k folds with the same size, each of which has roughly the same proportion of defect changes. Then each fold of data is uses as the test set, and the remaining k-1 fold data is used as the training set. Finally, the average value of k prediction results is used as the final prediction performance. The challenge of k-fold cross-validation is to choose an

Table 1. The Basic Information of Data Sets

Project	#Changes	#Defective changes	%Defective rate	Period	Language
Fabric8	15591	5177	33.2	12/2011–12/2017	Java
JGroups	21468	6305	29.4	09/2003–12/2017	Java
Camel	36753	12488	34.0	03/2007–12/2017	Java
Tomcat	24043	10372	43.1	03/2006–12/2017	Java
Brackets	21347	8082	37.9	12/2011–12/2017	JavaScript
Neutron	24057	9214	38.3	12/2010–12/2017	Python
Spring-Integration	10999	4640	42.2	11/2007–01/2018	Java
Broadleaf	17429	5046	29.0	11/2008–12/2017	Java
Nova	61364	24857	40.5	08/2010-01–2018	Python
NPM	9258	2773	30.0	09/2009–11/2017	JavaScript

Table 2. The description of metrics

Dimension	Metric	Description
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix
History	NDEV	Number of developers that changed the files
	AGE	Average time interval between the last and the current change
	NUC	Number of unique last changes to the files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

appropriate value of k. We use the widely used 10-fold cross-validation technique to split training and test set [41].

After dividing the training set and the test set, we use the random forest algorithm [2] to build a classification model. Random forest is an ensemble learning algorithm based on bagging. It uses decision trees as the base learners and introduces a random attribute selection scheme. We chose random forest to build the classifier mainly considering the following two reasons: First, compared with traditional machine learning models, random forest has higher robustness, prediction accuracy, and stability [13]; Secondly, the research of Kamei et al. [18] shows that random forest has better prediction performance than other modeling

techniques in software defect prediction research. Then they introduced random forest to the research of JIT-SDP [17].

4.3 Evaluation Metrics

This paper adopts three CA methods to explain the JIT-SDP model. To evaluate the differences in the feature importance rank lists generated by different model interpretation methods, similar to the research of Rajbahadur et al. [30], we introduced four evaluation indicators: Kendall's Tao coefficient, Kendall's W coefficient, top-3 overlap, and top-1 overlap. The specific introduction is as follows.

Kendall's Tau coefficient (τ) [20] is a non-parametric statistical coefficient used to evaluate the correlation between two sequences. If the two sequences are completely the same, then $\tau = 1$; if the two sequences are completely opposite, then $\tau = -1$. This paper uses τ to evaluate the agreement of the feature importance rank lists generated by the two CA methods. According to the Akoglu's suggestion [1], the relationship between the results of the τ test and the interpretation of agreement is shown in Eq. 5.

$$Kendall's\ \tau\ Agreement = \begin{cases} weak, & \text{if } |\tau| \leq 0.3 \\ moderate, & \text{if } 0.3 < |\tau| \leq 0.6 \\ strong, & \text{if } 0.6 < |\tau| \leq 1 \end{cases} \tag{5}$$

Kendall's W coefficient is a statistic used to evaluate the agreement of multiple sequences, and its range is between 0 and 1. The larger the value of Kendall's W, the more consistent the feature importance rank lists generated by the three CA methods. The agreement explanation of Kendal's W coefficient is shown in Eq. 6 [4].

$$Kendall's\ W\ Agreement = \begin{cases} weak, & \text{if } W \leq 0.3 \\ moderate, & \text{if } 0.3 < W \leq 0.5 \\ good, & \text{if } 0.5 < W \leq 0.7 \\ strong, & \text{if } 0.7 < W \leq 1 \end{cases} \tag{6}$$

Top-k overlap [30] is used to evaluate the agreement of multiple sequences on top k values. This paper uses the top-k indicator to evaluate the overlap of the feature importance rank lists generated by LIME, BreakDown, and SHAP on the top-k features. The calculation process of top-k is shown in Eq. 7, where p represents the number of ranking lists. In our experiment, p is set to 3, and k is set to 1 and 3.

$$Top-k\ overlap = \frac{|\bigcap_{i=1}^{p} features\ at\ top\ k\ ranks\ in\ list_i|}{|\bigcup_{i=1}^{p} features\ at\ top\ k\ ranks\ in\ list_i|} \tag{7}$$

Eqs. 8 and 9 give interpretations for the results of top-3 overlap and top-1 overlap, respectively.

$$Top-3\ Agreement = \begin{cases} negligible, & \text{if } 0.00 < top-3\ overlap \leq 0.25 \\ small, & \text{if } 0.25 < top-3\ overlap \leq 0.50 \\ medium, & \text{if } 0.50 < top-3\ overlap \leq 0.75 \\ large, & \text{if } 0.75 < top-3\ overlap \leq 1.00 \end{cases} \tag{8}$$

$$Top-1\ Agreement = \begin{cases} low, & \text{if } 0.00 < top-1\ overlap \leq 0.50 \\ high, & \text{if } 0.50 < top-1\ overlap \leq 1.00 \end{cases} \tag{9}$$

For example, if the top-3 feature sets generated by the three CA methods are as follows: LIME={LA, NF, Entropy }, BreakDown={NUC, LA, EXP }, SHAP={LA, SEXP, ND}, then top-3 overlap=1/7. Therefore, the top-3 agreement among three CA methods is negligible. If the feature of top-1 generated by three CA methods are: LIME = {LA}, BreakDown = {LA}, and SHAP = {LA}, then top-1 overlap = 1. Therefore, the top-1 agreement among three CA methods is high.

5 Experimental Results and Analysis

In this section, according to the experimental results, we analyze the two research questions proposed in Sect. 4.

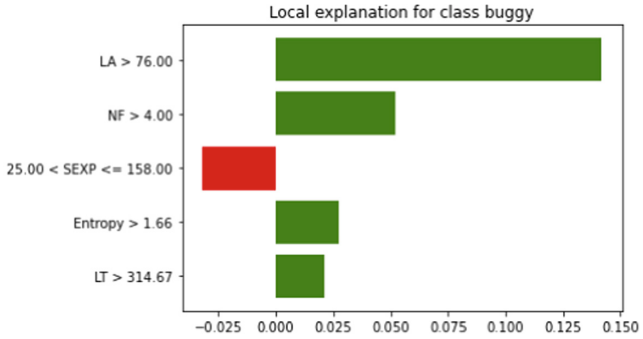
5.1 Analysis for RQ1

Motivation. Currently, most of the JIT-SDP researches pay attention to the prediction performance of classifiers while ignoring the interpretability of model when building classification models, which hinders the application of JIT-SDP technology in actual software development. The traditional CS interpretation methods interpret prediction models according to the internal structure of models. However, the CS method cannot explain an individual instance, and cannot explain the black box model. To this end, we introduced three CA interpretation technologies, LIME, BreakDown, and SHAP, to interpret JIT-SDP models.

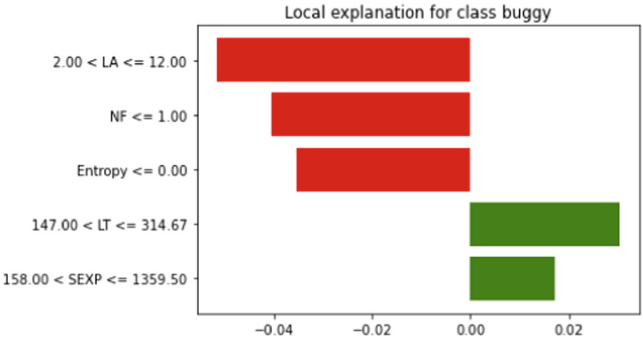
Approach. To answer RQ1, we counted the differences in the feature importance ranks of the instance explanations generated by the three CA technologies. Specifically, for each project, we first use a 10-fold cross-validation method to divide the data set into a training set and a test set, and then use random forest to build a classification model on the training set. To calculate the difference in the instance explanations in the test set, we first use the CA technology to calculate the feature importance scores for each instance. Then, according to the feature importance scores, we sort the features of each instance. Finally, we counted the differences in feature importance ranks among different instance interpretations, and used box plots to show the distribution of the differences in

feature importance ranks. For example, if the feature importance score of one instance c_1 is [LA = 0.108, EXP = 0.049, NUC = 0.259], the feature importance score of another instance $c2$ is [LA = 0.305, EXP = 0.014, NUC = 0.205], the feature importance rank of c_1 is $[1st = NUC, 2nd = LA, 3rd = EXP]$, and the feature importance rank of c_2 is $[1st = LA, 2nd = NUC, 3rd = EXP]$. Therefore, the ranking difference between c_1 and c_2 on feature LA is $|2 - 1| = 1$. We apply the calculation process to all instances that are accurately predicted as buggy changes in the test set.

Results



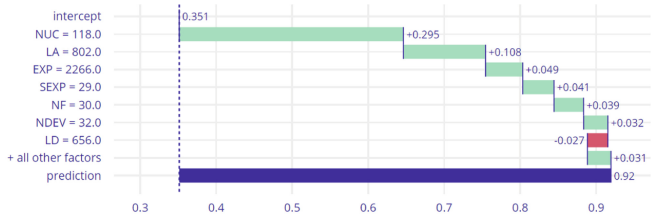
(a) Change 1



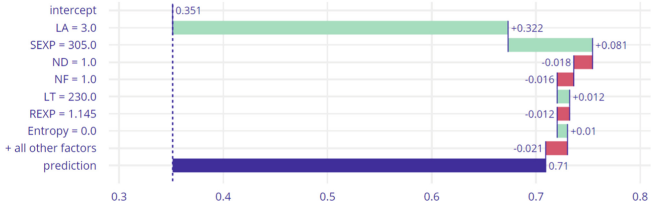
(b) Change 2

Fig. 1. An example of instance explanation of LIME

CA technologies can explain the prediction results of the JIT-SDP model and provide visual explains. We selected two instances Change 1 and Change 2 from the project Fabric8. These two instances are accurately predicted as buggy changes by the prediction model, and have a defect probability of 92% and 71%, respectively. Figure 1, Fig. 2, and Fig. 3 show the explanations about Change 1 and Change 2 provided by LIME, BreakDown, and SHAP, respectively.



(a) Change 1



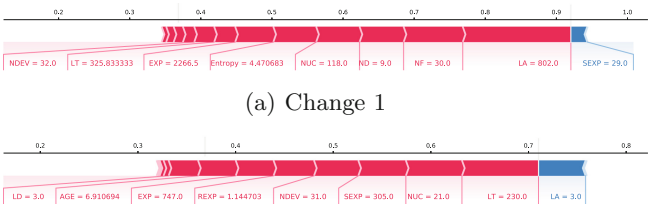
(b) Change 2

Fig. 2. An example of instance explanation of BreakDown

Figure 1 shows the visual interpretation generated by LIME for Change 1 and Change 2. The green bar indicates the score that supports the change to become buggy, and the red bar indicates the score that supports the change to be clean. It can be seen from Fig. 1 that the three most important factors for Change 1 to be predicted as buggy are $\{LA > 76\}$, $\{NF > 4\}$ and $\{Entropy > 1.66\}$; In the remaining 8% probability, the condition $\{25 < SEXP \leq 158\}$ is the main factor for Change 1 to have a clean probability; For Change 2, condition $\{147 < LT \leq 314.67\}$ and condition $\{158 < SEXP \leq 1359.5\}$ are the main factors that change 2 is predicted to be buggy; In the remaining 29% probability, the condition $\{2 < LA \leq 12\}$ is the most important factor for Change 2 to have a clean probability.

Figure 2 shows the explanations of the BreakDown technology on the prediction results of Change 1 and Change 2, in which the green bar and the red bar respectively indicate the feature importance scores that support and oppose the change to be buggy. It can be seen from Fig. 2 that for Change 1, the feature importance scores of the conditions $\{NUC = 118\}$ and $\{LA = 802\}$ are 0.295 and 0.108, which are the factors that accounts for the largest proportion of the defect probability; The importance score of the condition $\{LD = 656\}$ is -0.027, which is the largest factor in the probability of clean in Change 1. For change 2, the feature importance score of condition $\{LA = 3\}$ is 0.322, which is the largest factor of defect probability; The scores of the conditions $\{ND = 1\}$ and $\{NF = 1\}$ are -0.018 and -0.016, which are the largest factors for the existence of clean probability in Change 2.

Figure 3 shows instance explanations generated by the SHAP technology, in which the red and blue areas represent the feature importance scores of supporting and opposing changes that are predicted to be buggy, respectively. The



(a) Change 1

(b) Change 2

Fig. 3. An example of instance explanation of SHAP

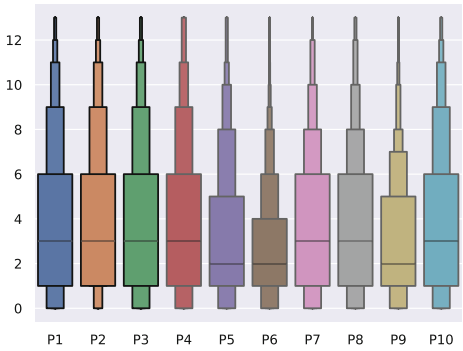


Fig. 4. The distribution of rank differences of each metric generated by LIME

longer the red or blue area is, the larger the absolute value of the corresponding feature importance score is. As can be seen from Fig. 3, for Change 1, LA, NF, and ND are the three most important features that cause Change 1 to be predicted as buggy; SEXP is the most important feature that Change 1 is predicted to be clean. For Change 2, LT, NUC, and SEXP are the three most important features that cause Change 2 to be predicted as buggy; and LA is the main feature of Change 2 with clean probability.

It can be seen from Fig. 1, Fig. 2, Fig. 3 that Change 1 and Change 2 have different instance explanations. To this end, we counted the differences in the feature importance ranks of different instances for each project. Figure 4, Fig. 5, and Fig. 6 respectively show the experimental results of LIME, BreakDown and SHAP technologies on ten projects, where P1, P2,... P10 represents the abbreviations of the names of the ten projects in Sect. 4, and the vertical axis represents the difference value of the feature ranks. As can be seen from Fig. 4, Fig. 5, and Fig. 6, the median of difference of the feature ranks generated by LIME, BreakDown and SHAP is 3 in 7, 10, and 10 projects. Therefore, the most important feature of one instance may rank fourth in another instance. Therefore, there are great differences among instance explanations, which further verifies the importance of introducing CA technology into JIT-SDP model.

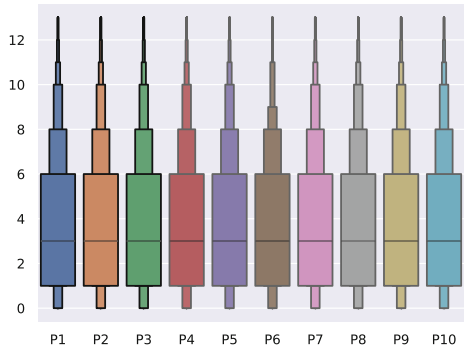


Fig. 5. The distribution of rank differences of each metric generated by BreakDown

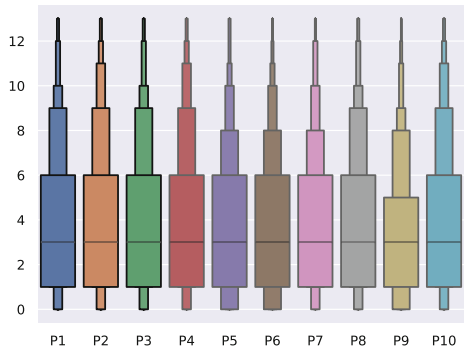


Fig. 6. The distribution of rank differences of each metric generated by SHAP

5.2 Analysis for RQ2

Motivation. The feature importance rank lists generated by CA technologies can provide valuable insights and guide developers to design code changes with low probability of defects. However, the principles and calculation processes of different CA methods are different. Therefore, we further studied the degree of difference in the feature importance rank lists generated by different CA methods.

Approach. For each project, we first use the 10-fold cross-validation method to divide the data set into a training set and a test set. Subsequently, we used LIME, BreakDown, and SHAP to calculate the feature importance scores of each instance in the test set. Finally, the features of the instances are sorted according to feature importance scores. According to the feature importance ranks of all instances in the data set, we use the Scott-Knott effect size difference (ESD) test [41] to calculate feature ranking lists of an project. Scott-Knott ESD test is a variant of Scott-Knott test [34], which uses hierarchical cluster analysis to divide the features of a project into several statistically different groups. Compared with

Scottp-Knott test, Scott-Knott ESD test does not require the input data to obey normal distribution, and can combine two groups with negligible effect size into a group. We use Kendall's Tao coefficient, Kendall's W coefficient, top-3 overlap, and top-1 overlap to evaluate the agreement of feature ranking lists of different CA methods.

Results. Table 3 shows the values of Kendall's Tau indicator for any two CA methods of the three CA technologies. The values marked in red in the table indicate that the τ coefficients are greater than 0.6, and the values marked in blue indicate that the τ coefficients are greater than 0.3 and less than or equal to 0.6. As can be seen from the Table 3, the feature ranking lists generated by LIME and BreakDown have strong agreement on five projects, and there is moderate agreement on the other five projects; The feature ranking lists generated by LIME and BreakDown have strong agreement on ten projects; The ranking lists generated by BreakDown and SHAP have strong agreement on nine projects and have a moderate agreement on one project. The 'Average' row counts the average value of the τ coefficient on ten projects. It shows that the feature ranking lists generated by any two CA methods have strong agreement.

Table 4 shows the results in the top-1 overlap, top-3 overlap and Kendall's W indicators of the feature ranking lists generated by the three CA methods. It can be seen from Table 4 that the top-1 overlap values of the feature ranking lists generated by the three CA methods are all equal to 1. Therefore, in the feature ranking lists, the three CA methods have high agreement on top-1 feature; In addition, the values marked in red indicate that the values of the top-3 overlap indicator are greater than 0.25 and less than or equal to 0.5. The values marked in blue indicate that the values of the top-3 overlap indicator are less than or equal to 0.25. It can be seen that among the top-3 features of the feature ranking lists, the three CA methods have small agreement on 7 projects and negligible agreement on three projects. The Kendall's W coefficient shows that the feature ranking lists generated by the three CA methods have values greater than 0.7 on ten projects, so they have a strong agreement. The 'Average' row counts

Table 3. Kendall's Tau values between two CA methods

Project	τ(LIME, BreakDown)	τ(LIME, SHAP)	τ(BreakDown, SHAP)
Fabric8	0.745	0.711	0.887
JGroups	0.679	0.774	0.748
Camel	0.559	0.701	0.856
Tomcat	0.594	0.671	0.823
Brackets	0.649	0.678	0.894
Neutron	0.561	0.761	0.667
Spring-Integration	0.691	0.810	0.770
Broadleaf	0.607	0.887	0.565
Nova	0.557	0.750	0.758
NPM	0.386	0.724	0.742
Average	0.603	0.747	0.771

Table 4. Top-1 overlap, top-3 overlap, and Kendall's W among three CA methods

Project	Top-1 overlap	Top-3 overlap	Kendall's W
Fabric8	1.000	0.400	0.846
JGroups	1.000	0.400	0.856
Camel	1.000	0.500	0.821
Tomcat	1.000	0.400	0.799
Brackets	1.000	0.500	0.819
Neutron	1.000	0.200	0.851
Spring-Integration	1.000	0.200	0.833
Broadleaf	1.000	0.500	0.828
Nova	1.000	0.500	0.853
NPM	1.000	0.200	0.734
Average	1.000	0.380	0.824

the average of the three evaluation indicators on ten projects. As can be seen from the 'Average' row, the feature ranking lists of the three CA methods have strong agreement and high agreement on top-1 feature; However, they have small agreement on top-3 features.

6 Conclusion and Future Work

We introduced three CA interpretation technologies LIME, BreakDown, and SHAP to interpret the prediction results of the JIT-SDP model. The experiment conducted large-scale empirical research on six open source projects. The empirical results show that: (1) The CA technology has different explanations for different instances. On average, the difference of feature ranks for two instances is 3. (2) The feature ranking lists and the top-1 feature generated by different CA technologies have a strong agreement. However, the top-3 features in the feature lists of different CA technologies have a small agreement. Therefore, in the actual software development process, we recommend using CA technology to provide practical explanations for the prediction results of the JIT-SDP model.

Our research provides knowledge about the JIT-SDP model interpretation. In the future, we will further carry out the following research: (1) This paper uses three CA technologies to explain the prediction results of the JIT-SDP model. In the future, we will study the agreement of the explanations between CA technologies and CS technologies. (2) Different CA technologies have different principles and calculation processes. Therefore, we will explore the agreement of explanations generated by different CA technologies for a given instance. (3) The running time of the interpretation technology is one of the factors that determine whether the participants will adopt the interpretation technology. Therefore, we will further analyze the running time of different CA technologies.

In order to ensure the reproducibility of experimental results, we provide all experimental data and experimental codes, which could be download at https://github.com/yangxingguang/jit_explain

Acknowledgment. This work was supported by the National Natural Science Foundation of China (No. 61772200), the Project Supported by Shanghai Natural Science Foundation (No. 21ZR1416300).

References

1. Akoglu, H.: User's guide to correlation coefficients. Turk. J. Emerg. Med. **18**(3), 91–93 (2018)
2. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
3. Cabral, G.G., Minku, L.L., Shihab, E., Mujahid, S.: Class imbalance evolution and verification latency in just-in-time software defect prediction. In: Proceedings of the 41st International Conference on Software Engineering, pp. 666–676 (2019)
4. Cafiso, S., Di Graziano, A., Pappalardo, G.: Using the Delphi method to evaluate opinions of public transport managers on bus safety. Saf. Sci. **57**, 254–263 (2013)
5. Chakraborty, S., et al.: Interpretability of deep learning models: a survey of results. In: IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation, pp. 1–6 (2017)
6. Chen, X., Zhao, Y., Wang, Q., Yuan, Z.: MULTI: multi-objective effort-aware just-in-time software defect prediction. Inf. Softw. Technol. **93**, 1–13 (2018)
7. Dam, H.K., Tran, T., Ghose, A.: Explainable software analytics. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, pp. 53–56 (2018)
8. Fan, Y., Xia, X., Da Costa, D.A., Lo, D., Hassan, A.E., Li, S.: The impact of changes mislabeled by SZZ on just-in-time defect prediction. IEEE Trans. Softw. Eng. **47**(8), 1559–1586 (2019)
9. Gosiewska, A., Biecek, P.: iBreakDown: Uncertainty of model explanations for nonadditive predictive models. arXiv preprint arXiv:1903.11420 (2019)
10. Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N.: Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. In: Proceedings of the 16th International Conference on Mining Software Repositories, pp. 34–45 (2019)
11. Huang, Q., Xia, X., Lo, D.: Supervised vs unsupervised models: a holistic look at effort-aware just-in-time defect prediction. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME, pp. 159–170. IEEE Computer Society (2017)
12. Huang, Q., Xia, X., Lo, D.: Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empirical Softw. Eng. **24**(5), 2823–2862 (2019)
13. Jiang, Y., Cukic, B., Menzies, T.: Can data transformation help in the detection of fault-prone modules?. In: Proceedings of the Workshop on Defects in Large Software Systems, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 16–20 (2008)
14. Jiarpakdee, J., Tantithamthavorn, C., Dam, H.K., Grundy, J.: An empirical study of model-agnostic techniques for defect prediction models. IEEE Trans. Softw. Eng. 1 (2020)

15. Jiarpakdee, J., Tantithamthavorn, C., Grundy, J.C.: Practitioners' perceptions of the goals and visual explanations of defect prediction models. In: 18th IEEE/ACM International Conference on Mining Software Repositories, pp. 432–443 (2021)
16. Jiarpakdee, J., Tantithamthavorn, C., Hassan, A.E.: The impact of correlated metrics on the interpretation of defect models. IEEE Trans. Softw. Eng. **47**(2), 320–331 (2021)
17. Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E.: Studying just-in-time defect prediction using cross-project models. Empirical Soft. Eng. **21**(5), 2072–2106 (2016)
18. Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K., Adams, B., Hassan, A.E.: Revisiting common bug prediction findings using effort-aware models. In: 26th IEEE International Conference on Software Maintenance, pp. 1–10 (2010)
19. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. **39**(6), 757–773 (2013)
20. Forthofer, R.N., Lehnen, R.G.: Rank correlation methods. In: Public Program Analysis. Springer, Boston, MA (1981). https://doi.org/10.1007/978-1-4684-6683-6_9
21. Li, Z., Jing, X., Zhu, X.: Progress on approaches to software defect prediction. IET Softw. **12**(3), 161–175 (2018)
22. Lipton, Z.C.: The mythos of model interpretability. Commun. ACM **61**(10), 36–43 (2018)
23. Liu, J., Zhou, Y., Yang, Y., Lu, H., Xu, B.: Code churn: a neglected metric in effort-aware just-in-time defect prediction. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 11–19 (2017)
24. Lundberg, S.M., Lee, S.: A unified approach to interpreting model predictions. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, pp. 4765–4774 (2017)
25. Mervyn, S.: Cross-validatory choice and assessment of statistical predictions. J. Roy. Stat. Soc.: Ser. B (Methodol.) **36**(2), 111–133 (1974)
26. Miller, T.: Explanation in artificial intelligence: insights from the social sciences. Artif. Intell. **267**, 1–38 (2019)
27. Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Tech. J. **5**(2), 169–180 (2000)
28. Naik, K., Tripathy, P.: Software testing and quality assurance: theory and practice. John Wiley & Sons (2011)
29. Pornprasit, C., Tantithamthavorn, C.: Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In: 18th International Conference on Mining Software Repositories, pp. 1–11 (2021)
30. Rajbahadur, G.K., Wang, S., Ansaldi, G., Kamei, Y., Hassan, A.E.: The impact of feature importance methods on the interpretation of defect classifiers. IEEE Trans. Softw. Eng. 1 (2021)
31. Regulation, G.D.P.: Regulation eu 2016/679 of the european parliament and of the council of 27 April 2016. Official Journal of the European Union (2016)
32. Ribeiro, M.T., Singh, S., Guestrin, C.: 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, pp. 1135–1144 (2016)
33. Rosen, C., Grawi, B., Shihab, E.: Commit guru: Analytics and risk prediction of software commits. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, pp. 966–969 (2015)

34. Scott, A.J., Knott, M.: A cluster analysis method for grouping means in the analysis of variance. Biometrics, pp. 507–512 (1974)
35. Shihab, E., Hassan, A.E., Adams, B., Jiang, Z.M.: An industrial study on the risk of software changes. In: 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, p. 62 (2012)
36. Sliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories (2005)
37. Staniak, M., Biecek, P.: Explanations of model predictions with lime and breakdown packages 10(2), 395 (2018). arXiv preprint arXiv:1804.01955
38. Tabassum, S., Minku, L.L., Feng, D., Cabral, G.G., Song, L.: An investigation of cross-project learning in online just-in-time software defect prediction. In: IEEE/ACM 42nd International Conference on Software Engineering, pp. 554–565 (2020)
39. Tantithamthavorn, C., Hassan, A.E., Matsumoto, K.: The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans. Softw. Eng. **46**(11), 1200–1219 (2020)
40. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Ihara, A., Matsumoto, K.: The impact of mislabelling on the performance and interpretation of defect prediction models. In: 37th IEEE/ACM International Conference on Software Engineering, pp. 812–823 (2015)
41. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. **43**(1), 1–18 (2017)
42. Wattanakriengkrai, S., Thongtanunam, P., Tantithamthavorn, C., Hata, H., Matsumoto, K.: Predicting defective lines using a model-agnostic technique. IEEE Trans. Softw. Eng. (2021). https://doi.org/10.1109/TSE.2020.3023177
43. Yang, X., Yu, H., Fan, G., Shi, K., Chen, L.: Local versus global models for just-in-time software defect prediction. Sci. Program. 2384706:1–2384706:13 (2019)
44. Yang, X., Yu, H., Fan, G., Yang, K.: An empirical studies on optimal solutions selection strategies for effort-aware just-in-time software defect prediction. In: The 31st International Conference on Software Engineering and Knowledge Engineering, pp. 319–424 (2019)
45. Yang, X., Yu, H., Fan, G., Yang, K.: A differential evolution-based approach for effort-aware just-in-time software defect prediction. In: Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages, pp. 13–16 (2020)
46. Yang, X., Yu, H., Fan, G., Yang, K.: DEJIT: a differential evolution algorithm for effort-aware just-in-time software defect prediction. Int. J. Softw. Eng. Knowl. Eng. **31**(3), 289–310 (2021)
47. Yang, X., Lo, D., Xia, X., Sun, J.: TLEL: a two-layer ensemble learning approach for just-in-time defect prediction. Information & Software Technology **87**, 206–220 (2017)
48. Yang, Y., et al.: Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 157–168 (2016)