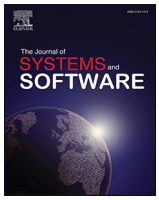
The Journal of Systems and Software 218 (2024) 112182

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss



CfExplainer: Explainable just-in-time defect prediction based on counterfactuals

Fengyu Yang [a,b], Guangdong Zeng [a,b,*], Fa Zhong [a,b], Peng Xiao [a,b], Wei Zheng [a,b], Fuxing Qiu [a,b]

[a] School of software, Nanchang Hangkong University, Nanchang 330063, PR China
[b] Software Testing and Evaluation Center, Nanchang Hangkong University, Nanchang 330063, PR China

ARTICLE INFO

Keywords:
Explainable just-in-time defect prediction
Counterfactual
Weighted class association rule
Explanation effectiveness
Explanation reliability

ABSTRACT

Just-in-time (JIT) defect prediction helps rationally allocate testing resources and reduce testing costs. However, most JIT defect prediction models lack explainability, which significantly affects their credibility. Recently, the local interpretable model-agnostic explanations (LIME) method has been used widely in model-explainable research, and many improved LIME-based methods have been proposed. However, problems with respect to explanation effectiveness and reliability remain, which seriously affects the practical use of LIME. To address this problem, CfExplainer, a local rule-based model-agnostic approach, is proposed. The approach first applies counterfactuals to generate synthetic instances. It then mines weighted class association rules based on synthetic instances, and it optimises the process of generating, ranking, pruning, and predicting the class association rules. Next, it employs the rules with the highest priority to explain the prediction results of the model. Experiments were conducted using the public datasets employed in related studies. Compared to other state-of-the-art methods, in terms of explanation effectiveness, CfExplainer's instance similarity improves by 26.5 %-31.2 %, and local model fittness improves by 2.0 %-3.5 %, 2.3 %-3 %, and 0.7 %-7.5 % on the AUC, F1-score, and Popt metrics, respectively. In terms of the reliability of the explanation, explanations that are 2.6 %-4.7 % more unique and 2.5 %-5.9 % more consistent with the actual characteristics of defect-introducing commits than other state-of-the-art methods. Thus, the explanations of the proposed approach can enhance the model credibility and help guide developers in fixing defects and reducing the risk of introducing them.

1. Introduction

In existing high-complexity software systems, it is difficult for limited software quality assurance resources to cover all test cases. Thus, software testing is not capable of finding all defects. To address this issue, software defect prediction (SDP) techniques have been developed to search for more efficient software testing methods. SDP techniques can help testers more effectively allocate test resources and improve test efficiency (Giray et al., 2023).

Software defect prediction is classified into three categories: within-project (WP) (Uddin et al., 2022), cross-project (CP) (Khatri and Singh, 2022), and just-in-time (JIT) defect prediction (Zhuang et al., 2022). Of these, JIT defect prediction predominates. It utilizes change data as the basis for making accurate predictions. However, most JIT defect prediction models are black-box models that lack explainability, which hinders the practical use of JIT defect prediction models (Zheng et al., 2021). Therefore, researchers have proposed an explainable SDP technique to help developers understand the causes of incurred defects and provide guidance for fixing them. Recently, the local interpretable model-agnostic explanations (LIME) approach has been proposed (Ribeiro et al., 2016). It is a method for explaining model predictions. When given an instance to be explained, LIME first generates synthetic instances similar to the instance to be explained using the random perturbation method. It then predicts synthetic instances using the black-box model and applies the prediction results as class labels of synthetic instances. Finally, it simulates the predictive behaviours of the black-box model through local models to generate specific explanations of instances to be explained. The above process is shown in Fig. 1.

It should be noted, however, that LIME-based explanations face the following problems. (1) LIME-generated explanations are heavily dependent on synthetic instances. Moreover, the use of randomized perturbation methods may generate synthetic instances that are not

* Corresponding author.
E-mail address: 2117421644@qq.com (G. Zeng).

https://doi.org/10.1016/j.jss.2024.112182
Received 20 February 2024; Received in revised form 6 July 2024; Accepted 4 August 2024
Available online 5 August 2024

similar to the instance to be explained, thereby failing to generate explanations that are specific to the instance to be explained. (2) The low fitness of local models leads to explanations that cannot effectively explain the decision-making mechanisms of black-box models, which limits the effectiveness of the explanations. (3) Explanations generated by LIME without considering associations between features, and generating only single-feature rules—i.e. only considering the degree of contribution of single features to prediction results—do not well explain the prediction results and lack operability. The above problems seriously affect the effectiveness of LIME explanations. To address the lack of operability of LIME, (Rajapaksha et al., 2021) proposed SQAPlanner, which uses cross-mutation to generate synthetic instances, explains the predictions of black-box models based on rules and generates four types of guidance and risk thresholds, experimental results show that the rule explanations generated by SQAPlainer are effective, stable and operational. However, SQAPlanner constructs the local model by treating the features as equally important and ignoring the degree of contribution of the features to the defects, which will seriously affect the quality of the mined rules, which will in turn affect the fittness of the local model, and ultimately lead to the lack of effectiveness of the explanations generated by the SQAPlanner, as well as the explanations generated by the SQAPlanner fail to provide the degree of importance of the rules, i.e., they do not know the contribution of the features to the defects, which will make the generated explanations difficult to be applied in software development practice. In addition, to address the effectiveness of LIME explanations and the limitations of SQAPlanner, (Pornprasit et al., 2021) proposed PyExplainer, which adopts the method of cross-mutation (Ibrahim et al., 2021) for generating synthetic instances, and it constructs the local model of RuleFit. Experimental results showed that the explanations generated by PyExplainer outperformed those of other baseline methods. Although LIME and improved methods based on LIME (e.g. SQAPlanner, PyExplainer methods) have achieved better results, random perturbation or cross-mutation methods for generating synthetic instances ignore feature constraints, which could seriously affect the truthfulness of the synthetic instances. For example, when generating synthetic instances, instances may be generated that defy real-world laws (e.g. developers=1.5), synthetic instances lack truthfulness, which will seriously affect the reliability of the explanation, the generated explanation will not provide developers with reliable guidance for correcting defects. In addition, when explaining the same instance, the synthetic instances generated by the improved methods of LIME and LIME using random perturbation or cross mutation have some randomness, which is due to the fact that both of them have random factors in the perturbation of the features, which makes the generated synthetic instances with randomness, and the generated synthetic instances stock in different iteration processes may have large differences, which results in lacking of uniqueness in the generated explanations, i.e. The generated explanation is not specific to the instance to be explained, which will seriously affect the reliability of the explanation. In summary, current explainable software defect prediction faces the following two problems: 1) lack of truthfulness of synthetic instances generated by random perturbation or cross mutation; 2) lack of uniqueness of generated explanations.

To investigate a model's explainability, researchers have proposed the counterfactual explanation (CE) (Wachter et al., 2017). The CE obtains the prediction results desired by users with minimal input changes by using counterfactual instances, thereby elucidating the decision-making behaviour of the models. To address the lack of explainability of the models, (Cito et al., 2022) investigated the counterfactual explanations of source code models by making minimal changes to the source code for changing the model predictions. Further, they explored the factors affecting the generation of realistic and trustworthy counterfactual explanations, hence verifying the validity of the approach on three different source code models. Meanwhile, (Cheng et al., 2020) designed an interactive visualisation system called the decision explorer with counterfactual explanations (DECE) to provide decision support to users, to help users understand the decision-making mechanisms of the model, provide decision support to users, and also introduce a series of interactive functions that enable users to customise the generation of counterfactual explanations to find actionable explanations that better meet their needs.

Inspired by CEs that generate counterfactual instances by minimising changes, we propose CfExplainer, a model-agnostic approach based on local rules, to explain the prediction results of JIT defect prediction models. CfExplainer draws on the thinking of counterfactual explanation to generate counterfactual instances by minimising changes, while the generated counterfactual instances are stable during different iterations, using counterfactuals to generate synthetic instances, and mining class association rules based on synthetic instances, finally explaining the prediction results of the black-box model based on class association rules. Our scripts are available online(https://github.com/software-defetc-prediction-research/CfExplainer). The main contributions of this study are as follows:

(1) CfExplainer's instance similarity is higher than that of other methods and provides more accurate explanations for the model's prediction results.
(2) CfExplainer has a better local model of fittness than the other methods, generating explanations that are closer to the instances to be explained.
(3) CfExplainer explanations are reliable and outperform other methods on *%Unique* and *%Consistency* metrics.

The remainder of this paper is organised as follows. Section II describes related work, Section III presents CfExplainer, Section IV outlines the experimental design, Section V presents the analysis of the experimental results, Section VI provides discussion, Section VII explains the validity threat analysis, and Section VIII presents conclusions.

2. Related work

Research surveys have shown that developers often doubt the results of defect prediction (Zheng et al., 2022), a sentiment that stems from the unexplainability of the model. Therefore, the explainability of models is as critical as their predictive performance. This section focuses on LIME-based explainable SDP and their counterfactual.

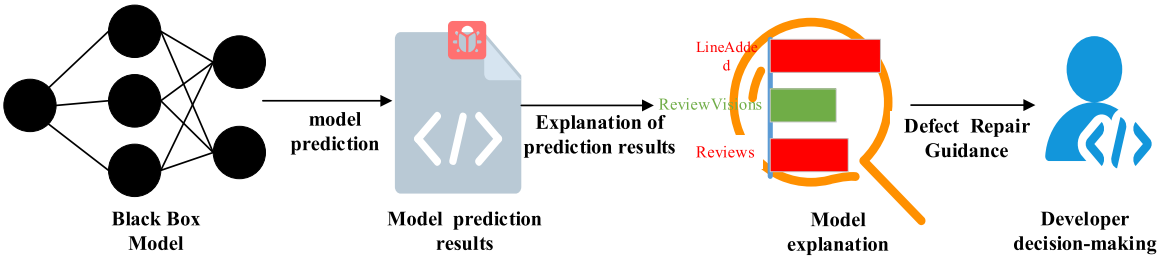


Fig. 1. Explanation of the proposed model.

2.1. Explainable software defect prediction based on lime

LIME explains the prediction of a model by constructing an explainable local model, which is a model-level explanation. It explains the prediction results of the model using the following steps: It (1) selects the instances to be explained and generates synthetic instances by random perturbation; (2) predicts synthetic instances; and (3) constructs explainable local models based on synthetic instances and explains the prediction results of black box models by fitting the predictions of black box models through local models.

Although LIME has been widely used in explainable studies of models, it still has some limitations. For example, the synthetic instances generated by LIME through random perturbations may not be similar to the instances to be explained, and the explanations generated by LIME are heavily reliant on the synthetic instances (Jia et al., 2019), which will seriously affect the effectiveness of the explanations. To address the above limitations, (Jiarpakdee et al., 2020) empirically evaluated three model-agnostic methods, (Gosiewska and IBreakDown, 2019), LIME, and hpo-LIME, where hpo-LIME is a hyper-parameter optimisation of LIME. The experimental results showed that the instance explanations generated by hpo-LIME had a high overlap with the global explanations of defect prediction models, and the generated explanations were reliable. Furthermore, Zafar et al. (Zafar and Khan, 2021) proposed DLIME, which first groups the training data using hierarchical aggregation and then selects the relevant clusters of the instances to be explained using K-nearest neighbours instead of generating synthetic instances using the random perturbation method. Finally, they construct explainable local models based on the relevant clustered instances of the instances to be explained to generate explanations. The experimental results showed that DLIME can generate more reliable explanations. To solve the problem of the effectiveness of the LIME explanation, Pornprasit et al. (Pornprasit et al., 2021) proposed PyExplainer, which uses the cross-mutation method to generate instances and construct local rule models. Their experimental results showed that the effectiveness of the PyExplainer explanation outperforms that of other methods.

However, most current LIME-based-model explainability studies generate synthetic instances mainly by using random perturbation or cross-mutation, ignoring the constraints of the instances. That is, the generated instances may not conform to real-world laws, which leads to the lack of reliability of the generated explanations. Therefore, the generation of effective and reliable explanations based on LIME requires further research.

2.2. Counterfactual

Counterfactual explanation is a method of explaining the predicted results of models. It generates counterfactual instances by constructing hypothetical conditions that change the model's outputs contrary to the actual conditions to explain the model's decision-making mechanism, i.e., the predicted results of the model are explained by the generated counterfactual instances, which is an instance-level explanation method. The main steps in generating counterfactual explanations include: (1) selecting an instance to be explained, generating counterfactual instances based on the instance to be explained, and modifying the feature values of the instance to be explained by minimizing perturbations; (2) inputting counterfactual instances into the model for prediction results that differ from the original prediction. To this end, it compares the differences between the original and counterfactual instances to explain the decision-making mechanisms for model prediction.

To address the issue of model explainability, (Mothilal et al., 2020) proposed a counterfactual explanation that generates diversified counterfactuals that are well approximated to the decision boundaries. It generates counterfactual instances of a quality superior to that of previous methods. Hanmer and Mendiratta (2022) utilized counterfactual explanation in the field of SDP. Their prediction results from a model based on the counterfactual explanation improved the model transparency, thereby providing guidance to developers for defect correction. Furthermore, Temraz and Keane (2022) proposed the Counterfactual Augmentation (CFA) method to solve the data imbalance problem, CFA generates counterfactual instances in most classes of data to reduce the data imbalance rate compared with other data sampling techniques (e.g., SMOTE (Joloudari et al., 2023)). However, constructing effective counterfactual instances is a key challenge in counterfactual explanation. Maragno et al. (2022) proposed an optimal learning framework for generating counterfactual explanations with constraints that make the generated counterfactual explanations more reliable.

In summary, counterfactual data augmentation is achieved by generating counterfactual instances to reduce the class imbalance rate, while counterfactual explanation is achieved by generating counterfactual instances to explain the prediction results of the model, both via counterfactual instances. Although counterfactual explanations can be used to explain the model's prediction results, counterfactual explanations are designed for defect prediction for continuous releases of software, which requires multiple versions of the software to be trained and evaluated, whereas just-in-time defect prediction is based on developer-submitted data rather than being trained and evaluated based on multiple training versions. Therefore, Pornprasit et al. pointed out in their discussion that counterfactual explanation is not applicable to JIT defect prediction (Pornprasit et al., 2021).

Currently, LIME and LIME-improved explainable defect prediction face a lack of truthfulness of synthetic instances and lack of uniqueness of generated explanations. Inspired by counterfactuals and the fact that there is no research on generating synthetic instances using counterfactuals, these issues motivate us to propose a method for generating synthetic instances based on counterfactuals. The core idea of the counterfactuals approach for generating synthetic instances is to improve the reliability of the explanations by minimally changing the eigenvalues and artificially constraining the range of perturbations.

2.3. Motivating example

We illustrate these problems through an example. As shown in Fig. 2, the LIME random perturbation approach may generate instances that are not similar to those to be explained and which defy the laws of real-world data. Furthermore, the low fitness of the local model cannot effectively simulate the decision-making mechanism of the black box model, which affects the effectiveness of the explanation.

Explanations 1 and 2 in Fig. 2 are the explanations generated by LIME in the first and second iterations, respectively. On the left is the explanation generated by LIME, where the horizontal axis indicates the degree of the feature contribution to the model prediction result. On the right is the actual feature value of the Java file. Orange indicates that the feature supports the prediction of defects; blue indicates that the feature supports the prediction of non-defects. The contribution of "ns" to the prediction of defects is 0.054 since the actual feature value of "ns" is 2, according to the explanation generated by LIME. When "ns$>1''$, "ns" is considered to have a supporting effect on predicting defects, and its contribution of 0.054 is the weight coefficient of the local model. Under different iterations, there are large differences in explanations generated by both, which indicates the lack of reliability of the LIME-generated explanations.

As mentioned previously, explainable software defect prediction based on LIME and improved methods of LIME faces the following problems: 1) lack of truthfulness of generated synthetic instances, i.e., synthetic instances do not have truthfulness; and 2) lack of uniqueness of generated explanations. To address the above problems, we propose CfExplainer.

3. Methodology

This section describes CfExplainer in detail. Its general framework is shown in Fig. 3.

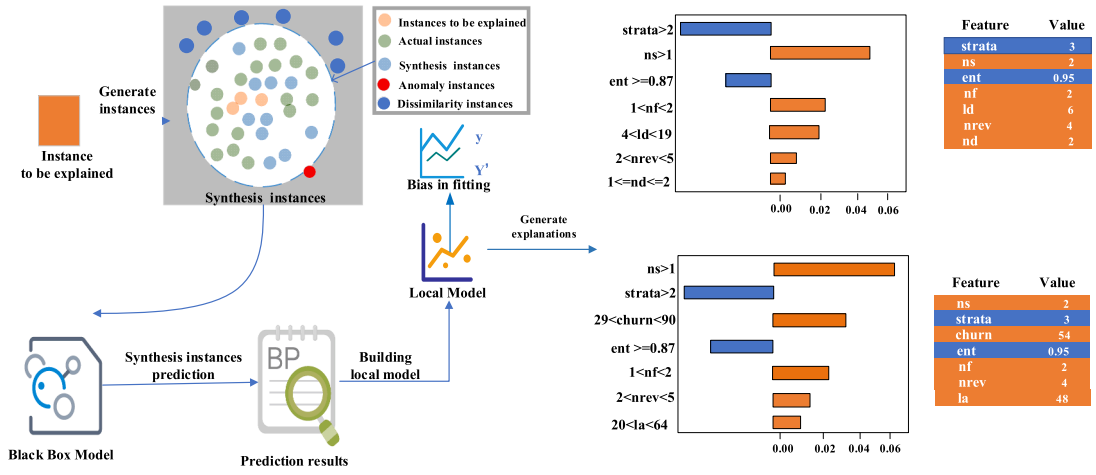


Fig. 2. Explanation of LIME generation.

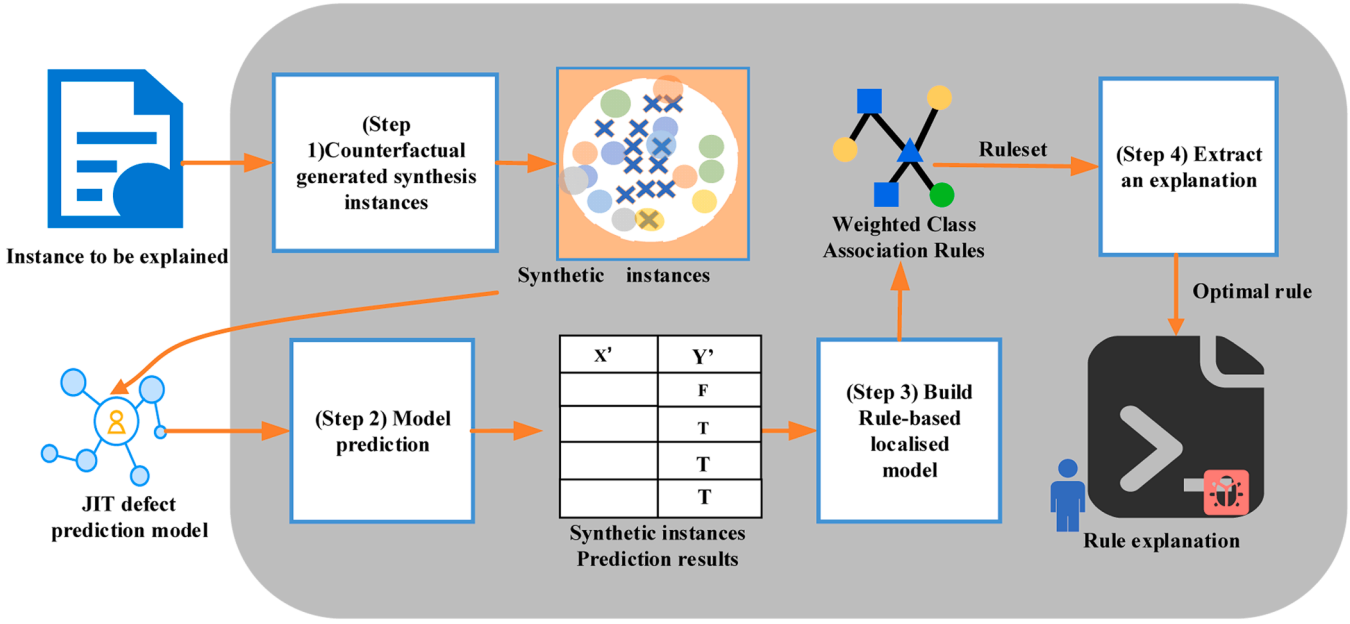


Fig. 3. CfExplainer framework diagram.

CfExplainer generates explanations through the following three steps: (1) generating synthetic instances, (2) constructing local models, and (3) generating explanations.

3.1. Generating synthetic instances

To ensure instance similarity, we selected the top N instances that were the most similar to the instance to be explained from the training data, as used by Rajapaksha et al. (2021), and we set N to 40. However, there may have been an insufficient number of instances, resulting in inadequate mining of the characteristics of the defects introduced by the instances to be explained, which would have affected the reliability of the explanation. Therefore, counterfactuals were introduced to generate synthetic instances.

Similar Instance Selection: select N instances with high similarity from the neighbourhood of the instance to be explained, N instances selection consists of the following three steps.

1. Feature normalisation: To eliminate the effects of different feature scales on instance selection, we performed Z-score normalisation on the training sets.

2. Calculate the instance similarity score: We first calculated the Euclidean distance between instances in each category in the training set and the instance to be explained, i.e., the Euclidean distance between each instance in the defective and non-defective classes and the instances to be explained, and then sorted them in descending order according to the Euclidean distance, and selected the top N instances in each class based on the similarity scores, and the similarity scores are calculated as shown in Eq. (1).

$$K(i_k, i_e) = \exp\left(-\frac{dist(i_k, i_e)^2}{2w^2}\right) \tag{1}$$

where $dist(i_k, i_e)$ is the Euclidean distance between instances i_k and i_e, and w is the kernel width advocated by Ribeiro et al. (2016), that is, the product of 0.75 and the number of instance features.

3. Select top-N similar instances: based on the calculated instance similarity scores, we sort the similarity scores of the instances of each class in descending order, and then we select the top-N instances of each class from the sorted instances.

Counterfactual generation of synthetic instances: Counterfactual

is a method used to change the model output results by modifying the feature values of instance x but minimising the modification of the feature values. Meanwhile, the loss function is as small as possible; therefore, the counterfactual sample should be close to the original instances. The loss function is used to measure the distance between the prediction result of the counterfactual instance and the original instance prediction result, as well as the distance between the counterfactual and the instance of interest, as shown in Eq. (2):

$$L(x, x', y', \lambda) = \lambda \cdot (f(x') - y')^2 + d(x, x') \tag{2}$$

where λ is a tuning parameter, x' is the counterfactual of x, $f(x')$ is the predicted output of the counterfactual instance x', y' is the expected output, and $d(x, x')$ is the distance between x' and x, which is used to measure the similarity between the counterfactual instances and the original instances, calculated as shown in Eq. (3).

$$d(x, x') = \sum_{j=1}^{p} \frac{\left| x_j - x'_j \right|}{MAD_j} \tag{3}$$

where MAD is the absolute median difference, x_j denotes the jth feature of instance x, x'_j denotes the jth feature of counterfactual x', and p is the number of features.

3.2. Synthetic instance prediction

In the process of generating synthetic instances, we retain the feature values of the synthetic instances, while the class labels of the synthetic instances are not retained, this is because the model-agnostic approach is to explain the predictions of the model by simulating the behaviour of the global model through the local model. Therefore, CfExplainer uses the global model to obtain a prediction of the synthetic instance (i.e., whether the synthetic instance is defective or non-defective), uses the prediction result of the global model for the synthetic instance as the class label of the synthetic instance, and then we fit the prediction result of the global model by constructing an explainable local model, allowing the local model to learn the behaviour of the global model. Finally, the explainability of the model is improved by explaining the prediction results of the global model from the explainable local model.

3.3. Building local model

To build a high-accuracy local model, we explained its prediction results by mining weighted class association rules that included (1) rule generation, (2) rule ranking, (3) rule pruning, and (4) rule prediction.

3.3.1. Rule generation

JIT defect prediction dataset faces data imbalance problem, traditional class association rule mining methods often have difficulty in mining the defect rules owing to the low support of the defect rules (A=>bug). Moreover, traditional class association rule methods ignore the contribution of the features and regard the features as equally important, which will seriously affect the quality of the rules. where A on the left side of the defect rule is the antecedent, i.e., the set of features, and bug on the right side is the target class.

Therefore, we firstly used the classical Apriori (Shankar et al., 2022) algorithm to mine the class association rules; secondly, we considered the contribution of features to defects, recalculated support of rules based on the contribution of features, and then obtained the weighted support of rules. The weighted support not only considers feature frequency, but also considers the contribution of features to defects, which increases the attention to defective classes and avoids defective rules being filtered owing to imbalanced data. The weighted support was calculated as shown in Eq. (5).

$$support(r) = support(A => C) = P(A \cup C) \tag{4}$$

$$W(r) = \prod_{i=1}^{n} W(item)_i * support(r) \tag{5}$$

where support(r) is the support of rule r, $item$ is the feature, $W(item)_i$ is the feature contribution, that is, the spearman correlation coefficient, and n is the number of features.

3.3.2. Rule ranking

In traditional class association rules, rule ranking is based on the support of rules, and high support means high priority. However, in scenarios with imbalanced data, defective rules tend to be ignored owing to low support. Therefore, weighted support is used in this study to rank the rules. When two rules, r_1 and r_2, are given, r_1 has a high priority if (1) $W(r_1) > W(r_2)$; (2) $W(r_1) = W(r_2)$. However, the length of the rule for r_1 is smaller than that for r_2, and r_1 has a high priority. (3) $W(r_1) = W(r_2)$, and the rule length is the same; the earlier the rule is generated, the higher the priority. After getting the mined rule set, we divide the rule set into defective rule (DR) and non-defective rule (NDR), and then sort DR, NDR according to the above method, the detailed algorithm is shown in Table 1.

3.3.3. Rule pruning

To improve the rule prediction accuracy, we prune rules based on their weighted support by adopting pruning strategies to remove redundant and conflicting rules.

Remove redundant rules: If rule $r_1 : f_1 => defect(W(r_1) = v_1)$, $r_2 : f_1, f_2 => clean((W(r_2) = v_2)$, where $W(r_1) > W(r_2)$, this indicates that feature f_2 in r_2 does not contribute to the prediction result, and thus r_2 is removed, and vice versa, the rule is retained.

Remove conflicting rules: If rules $r_1 : f_1, f_2 => defect$, $r_2 : f_1, f_2 => clean$, with the same antecedents, we predict opposite results; such rules do not provide developers with guidance on defect fixing. Therefore, we remove all conflicting rules. The detailed algorithm is shown in Table 2.

3.3.4. Multi-Rule prediction

When predicting an instance, the instance may simultaneously satisfy multiple rules, as shown in Fig. 4.

In data imbalance scenarios, it is unreasonable to select a single rule with high priority for classification, and it is easy to misclassify defective classes as nondefective classes. Therefore, we classify the instances based on the sum of the weighted support of the rules and sum the weighted support of the defective rules satisfied by the given instances. We thereby obtain the defective probability Sum_D, and the non-defective probability Sum_{ND}. If $Sum_D > Sum_{ND}$, the instance is a defective class; otherwise, it is a nondefective class.

$$Sum_D = W(r_1) + W(r_2) + \ldots + W(r_n) \tag{6}$$

$$Sum_{ND} = W(r_1) + W(r_2) + \ldots + W(r_m) \tag{7}$$

where n and m are the numbers of defective rules and defect-free rules satisfied by an instance, respectively.

3.4. Rules explanation

CfExplainer explains the predictions of a model based on weighted class association rules. Compared to LIME explanations, the explanations generated by CfExplainer account for the association between features, which makes generated explanations more actionable and reduces the risk of introducing defects. To ensure the reliability of the explanation, we select the best rule (i.e. the rule with the highest weighted support) from the set of association rules to explain the prediction results of the model. The following key information is contained in the rule explanation. (1) decision features: the features related to defects; (2) feature importance scores: the contribution of features to

Table 1
Algorithm for rule ranking.

Algorithm 1: Pseudo-code for rule ranking
Inputs: *DR, NDR;* Outputs: rules 1: DR_time= DR.index 2: NDR_time= NDR.index 3: DR'=DR.sort_values(by=["support", "length","DR_time"], ascending=False) 4: NDR' = NDR.sort_values(by=["support", "length","NDR_time"], ascending=False) 5: return rules ={DR',NDR'};

Table 2
Rule pruning algorithm.

Algorithm 2: Pseudo-code for rule pruning.
Inputs : DR, NDR; *Outputs* : *rules* 1 : *if* $\forall r_1 : X_i \rightarrow C(Wsupp = s1)$ *and* $\forall r_2 : X_i \cdot X_j \rightarrow C(Wsupp = s2)$ *then* 2 : *if* $(s2 > s1)$ *then* 3 : $DR' = DR' - \{r1\}$; 4 : $NDR' = NDR' - \{r1\}$; 5 : *else* 6 : $DR' = DR' - \{r2\}$; 7 : $NDR' = NDR' - \{r2\}$; 8 : *if* $\forall r' : X_i \rightarrow$ *Defect and* $\forall r'' : X_i \rightarrow$ *No Defect then* 9 : $DR' = DR' - \{r'\}$; 10 : $ND = NDR' - \{r''\}$; 11 : *return rules* $= \{DR', NDR'\}$;

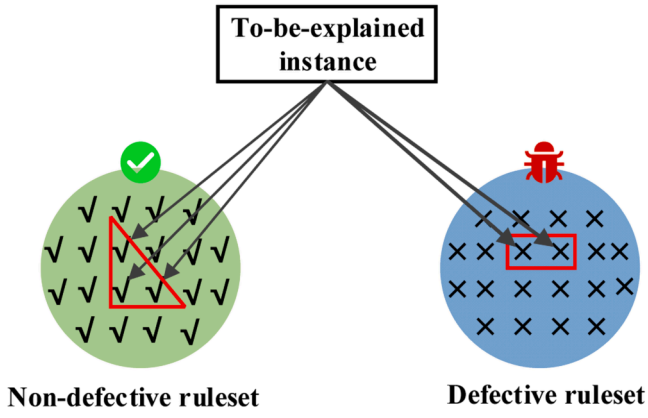


Fig. 4. Multi-rule prediction.

defects; (3) relationships between features and the model's predicted results; i.e. whether features play a supporting or opposing role in model prediction results; and (4) risk probability, or the risk probability of prediction of defects, that is, weighted support.

4. Experimental design

Our experiments were conducted on a computer with a GPU (NVIDIA GTX 3060), CPU (i7–11,700 K, 16 cores), Windows 10 operating system, and Python 3.9 programming language. To reduce the randomness of the conclusions, we repeated the experiment five times, taking the median of the results. This section presents the research dataset and the details of the experimental design, the experimental design is shown in Fig. 5.

4.1. Research questions

To verify the efficacy of CfExplainer, we proposed three research questions from the perspectives of both effectiveness and reliability of CfExplainer explanation, in which RQ1 and RQ2 are to verify the effectiveness of CfExplainer explanation, and RQ3 is to verify the reliability of CfExplainer explanation, and the research questions are specified as follows.

RQ1: How similar are the synthetic instances generated by CfExplainer to the instance to be explained?

Motivation: The validity of an explanation depends heavily on the quality of the synthetic instances (Pornprasit et al., 2021). Therefore, if the synthetic instances are not similar to the to-be-explained instances, the local model may not provide accurate insights about the logical reasoning of the global model. Therefore, the generated synthetic instances should be similar to the to-be-explained instances, whereas LIME using a random perturbation approach may generate synthetic instances that are not similar to the to-be-explained instances, which can seriously affect the effectiveness of the explanation. For this reason, we need to evaluate the effectiveness of CfExplainer's explanations, and the fittness of CfExplainer's local models will be studied.

RQ2: Does our CfExplainer have higher local model fitness than other baseline methods?

Motivation: the key to model agnostic methods is to build local models with good fittness to simulate the predictive behaviour of the global model. Goodness of fit is often used as a measure of how well a local model approximates the predictions of the global model (Ribeiro et al., 2016), with a higher fit indicating that the local model is able to better simulate the predictive behaviour of the global model. Therefore. local models should have good fittness, however, LIME's local models often fail to fit the global model effectively, i.e., they have low fittedness, which can seriously affect the effectiveness of the explanation. Therefore, to make the CfExplainer explanation effective, it is necessary to construct a high fit local model, i.e., a local model that can accurately fit the predictions of the global model. For this reason, we need to evaluate the effectiveness of the CfExplainer explanation by studying the fittness of the local model of CfExplainer.

RQ3: How reliable is the CfExplainer explanation?

Motivation: different instance synthesis methods generate different instances, which in turn leads to different explanations generated by model-agnostic methods. Currently, model-agnostic methods (LIME, PyExplainer) often use random perturbation or cross-mutation methods to generate synthetic instances, which have a certain degree of

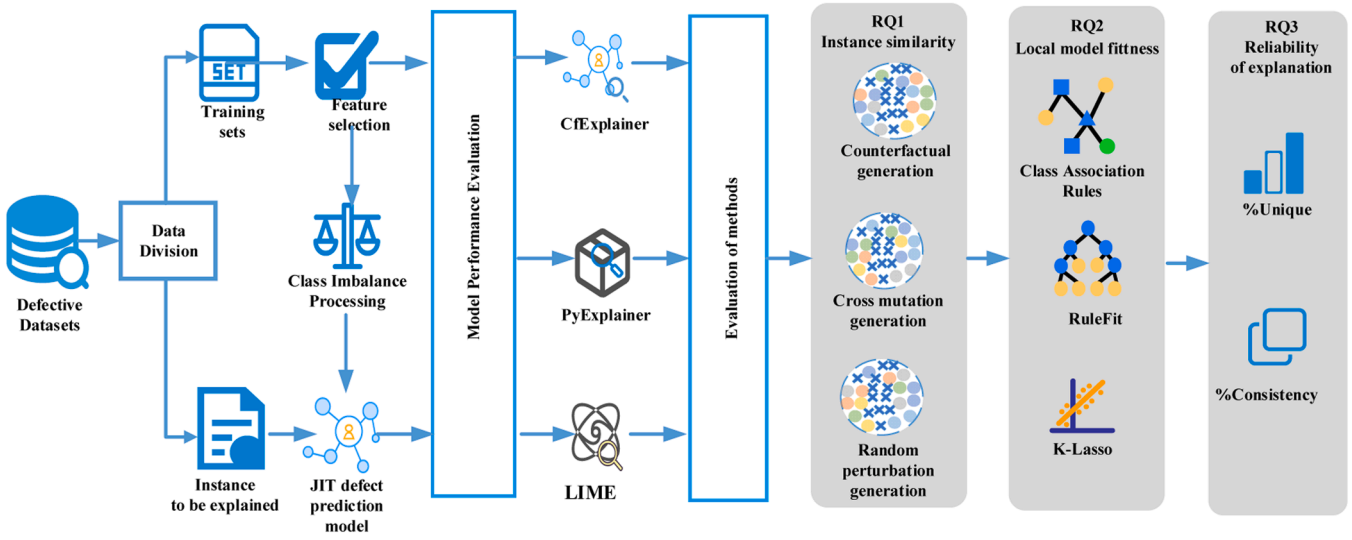


Fig. 5. Experimental design diagram.

randomness, i.e., the difference of instances generated by multiple iterations will produce different explanations of the rules, which in turn affects the reliability of the explanations. In addition, random perturbation or cross mutation methods generate synthetic instances that do not satisfy real-world laws, and the generated synthetic instances lack truthfulness, which can seriously affect the reliability of the explanation. Therefore, we will study the reliability of explanations and evaluate the reliability of explanations generated by the CfExplainer method through metrics.

4.2. Datasets

Mcintosh and Kamei (2018) proposed three important criteria to be satisfied for just-in-time defect dataset selection: traceability, fast iteration, and code review policy, and whether or not a just-in-time defect dataset satisfies the criteria of traceability, fast iteration, and code review policy could seriously affect the effectiveness of the method. Therefore, we selected Openstack and Qt from the open source software projects provided by McIntosh and Kamei, which both datasets satisfy the above three criteria. In addition, we also chose the Openstack and Qt datasets because these two datasets (1) are often used as benchmarks for defect prediction studies (Pornprasit and JITLine, 2021; Gesi et al., 2021; Zhang et al., 2023); and (2) are able to manually validate the effectiveness of the SZZ algorithm (Śliwerski et al., 2005) in order to reduce the number of false positives and false negatives. Where Openstack is an open source software for cloud infrastructure services and Qt is a cross-platform application development framework. Table 3 lists the details of the datasets, where each dataset is categorized into five types of commit-level features, i.e., size (e.g. lines of code added, lines of code deleted), diffusion (e.g. #modified files), history (#developers), experience, and code review activities. The features are show in Table 4.

Table 4
Description of Characteristics.

	Property	Description
Size	Lines added	The number of lines added by a change.
	Lines deleted	The number of lines deleted by a change.
Diffusion	Subsystems	The number of modified subsystems.
	Directories	The number of modified directories.
	Files	The number of modified files.
	Entropy	The spread of modified lines across file.
History	Unique changes	The number of prior changes to the modified files.
	Developers	The number of developers who have changed the modified files in the past.
	Age	The time interval between the last and current changes.
Author/Rev. Experience	Prior changes	The number of prior changes that an actor has participated in.
	Recent changes	The number of prior changes that an actor has participated in weighted by the age of the changes (older changes are given less weight than recent ones).
	Subsystem changes	The number of prior changes to the modified subsystem(s) that an actor has participated in.
	Awareness	The proportion of the prior changes to the modified subsystem(s) that an actor has participated in.
Review	Iterations	Number of times that a change was revised prior to integration.
	Reviewers	Number of reviewers who have voted on whether a change should be integrated or abandoned.
	Comments	The number of non-automated, non-owner comments posted during the review of a change.
	Review window	The length of time between the creation of a review request and its final approval for integration.

Table 3
Statistics on Dataset Information.

Project	Training Sets				Testing Sets			
	Start Date	End Date	# Commits	# Defective Commits	Start Date	End Date	#Commits	# Defective Commits
OpenStack	11/30/2011	08/13/2013	9246	980 (11 %)	08/13/2013	0228/2014	3.963	646(16 %)
Qt	06/18/2011	05/08/2013	19,312	1577 (8 %)	05/08/2013	03/18/2014	8.277	476(6 %)

4.3. Model Construction and model explanation

Data division: To ensure that the evaluation of our just-in-time defect predictions reflects a real-world scenario, we sort the dataset in ascending order by the date of record of the code commits (Pornprasit and JITLine, 2021), and divide the dataset into training (70 %) and testing (30 %) sets using time-wise hold-out validation technique techniques (such as those used by Pornprasit et al. (2021)), where time-wise hold-out validation technique can ensure that later-appearing commits will not be used for model training and commits that appear earlier will not be used in model evaluation.

Feature selection: To reduce interference from redundant features, the features must be filtered. Feature selection is mainly classified into filter-based and wrapper-based methods. Wrapper-based methods obtain the best feature subset through learner feedback interaction, while filter-based methods evaluate the feature subset on the training set independently, and the computational complexity of wrapper-based methods is high compared to filter-based methods. Therefore, we choose the filter-based feature selection method, i.e., feature selection using AutoSpearman, which is chosen because it is widely used in software defect prediction (Pornprasit et al., 2021; Roman et al., 2023). In addition, just-in-time defect data sets tend to have nonlinear relationships between features, and the method is suitable for data sets that have nonlinear relationships between features, and can be better applied to this scenario. We perform feature selection in the training set by automatically selecting a feature from the highly correlated features that has the least correlation with other features not in the set, and then applying the selected feature to the test set. We used the value suggested by Kraemer et al. (2003), i.e., a threshold of 0.7 to indicate strong correlation between features, and also for fair comparison with the baseline method, we finally selected seven features.

Class imbalance processing: SDP datasets tend to be highly imbalanced, which significantly affects the predictive performance of the model. Therefore, to ensure the prediction performance of the JIT defect prediction model, we conducted class imbalance processing on data based on counterfactuals (Wang et al., 2023). Accordingly, we minimally perturbed different feature values of non-defective classes to change the prediction results of the model. The constraints of features were also considered while the feature values were perturbed, the feature constraints here refer to the real range of values of the features, and the range of values of the features is limited by human prior knowledge, which makes the generated counterfactual samples have truthfulness. For example, the line of code (LOC) is an integer value greater than 0 in the real scenario, and we set the value of LOC to an integer value greater than 0 when generating synthetic instances, so that the generated counterfactual samples have truthfulness. The above process can generate diverse and truthful counterfactual samples, and the generated counterfactual samples are defective samples, which increases the number of defective samples and makes the defective and non-defective data balanced with each other. In the process of generating the defective class data, we adopted the suggestion of Vuttipittayamongkol et al. (2021) that model performance is best when the ratio of defective to non-defective classes is 1:1. Therefore, we conducted the data augmentation with the final ratio of defective classes to non-defective classes to be 1:1.

JIT defect prediction model build: To build a high-performance JIT defect prediction model, we chose two classical classification models, that is, random forest (RF) and logistic regression (LR). Previous studies have shown that the prediction performance of random forest and logistic regression generally outperforms that of other classification models (Balaram and Vasundra, 2022; Mehmood et al., 2023).

Model Evaluation: SDP data are often highly unbalanced, and accuracy does not accurately measure the performance of the model in identifying defects. Therefore, we use F1-score and area under the ROC curve (AUC) metrics (Chen et al., 2022) to evaluate the prediction performance of the model, since these metrics perform better in evaluating the model performance in identifying defects in class imbalance scenarios. The F1-score is calculated as shown in Eq. (8).

$$F1 - score = \frac{2 * Recall * Precision}{Precision + Recall} \tag{8}$$

Where *Recall* is the recall rate, which indicates the ratio of the number of samples correctly predicted as defective to the number of true defective samples in all samples, as shown in Eq. (9), and *Precision* is the precision rate, which indicates the ratio of the number of samples correctly predicted as defective to the number of all samples predicted as defective, as shown in Eq. (10).

$$Recall = \frac{TP}{TP + FN} \tag{9}$$

$$Precisin = \frac{TP}{TP + FP} \tag{10}$$

Where TP denotes true positive, FN denotes false negative and FP is false positive.

In addition, we introduce the work-perception metric Popt (Liu et al., 2022) to evaluate the code review effort. Popt denotes the percentage of detected defects versus all defects after the code inspection effort. Popt is based on the concept of Alberg diagrams as shown in Fig. 6. Popt is defined as follows.

$$Popt(m) = 1 - \frac{Area(Optimal) - Area(m)}{Area(Optimal) - Area(Worst)} \tag{11}$$

Where Area(Optimal) and Area(Worst) represent the optimal and worst models respectively, Area(m) denotes the predictive model and m denotes the percentage of lines of code (loc). In the optimal model, all modules are sorted in descending order of actual defect density, while in the worst model, all modules are sorted in ascending order of actual defect density.

JIT defect prediction model performance: To ensure that the generated instance explanations originated from accurate defect prediction models, we evaluated the prediction performance of the classifiers. RF and LR defect prediction models were constructed on OpenStack and Qt datasets. On the OpenStack dataset, the RF and LR classifiers were 0.864 and 0.841 for the Popt metrics, respectively. On the Qt dataset, the RF and LR classifiers were 0.959 and 0.963 for the Popt metrics, respectively. The experimental results are shown in Table 5. It is shown that RF and LR had good prediction performances.

Model explanation: for the model prediction results of the JIT defect model, we apply CfExplainer, PyExplainer and LIME to generate rule explanations of the model prediction results. We choose PyExplainer, LIME as a baseline comparison because LIME has been widely used in software engineering research (Rajapaksha et al., 2021;

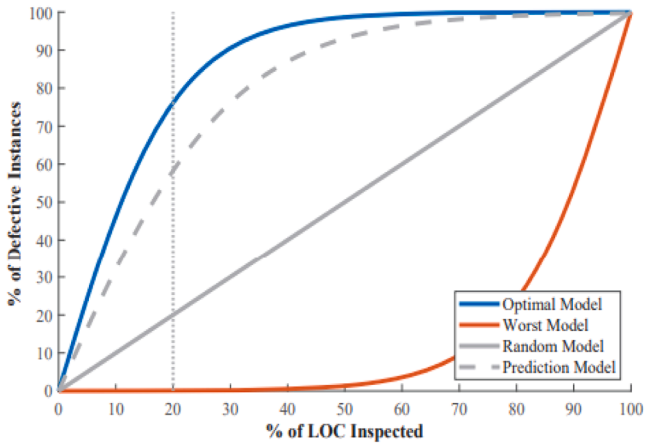


Fig. 6. Popt metrics graph (Liu et al., 2022).

Table 5
Predictive performance of model.

Datasets	RF			LR		
	Popt	AUC	F1-score	Popt	AUC	F1-score
OpenStack	0.864	0.831	0.443	0.841	0.704	0.427
Qt	0.959	0.784	0.395	0.963	0.679	0.336

Pornprasit et al., 2021; Peng and Menzies, 2021), PyExplainer has better explanation results, is one of the state-of-the-art explainable JIT defect prediction methods, and is similar to CfExplainer. CfExplainer, PyExplainer and LIME explanation generation consists of four steps: (1) generating synthetic instances; (2) global model prediction of synthetic instances; (3) constructing a local model based on synthetic instances; and (4) generating explanations based on the local model.The differences between CfExplainer, PyExplainer and LIME are as follows : CfExplainer, PyExplainer and LIME generate synthetic instances through counterfactuals, cross-mutations, and random perturbations, respectively; (2) local models are different: CfExplainer, PyExplainer and LIME construct weighted class association rules, RuleFit (Kato et al., 2022), and K-Lasso (Pornprasit and JITLine, 2021) local models, respectively .

Explanation evaluation: To verify the validity of CfExplainer, we evaluated it from two perspectives: (1) the effectiveness of the CfExplainer explanation. Since the effectiveness of the CfExplainer explanation depends on the instance similarity and local model fitness, the effectiveness of the CfExplainer explanation was evaluated herein according to the instance similarity and local model fitness. We used the Euclidean distance to evaluate the instance similarity, and we employed AUC, F1-score, Popt metrics to evaluate the local model fitness. (2) Reliability of CfExplainer explanation: We used two metrics, *%Unique* and *%Consistency* (Pornprasit et al., 2021), which evaluated the reliability of the explanations.

%Unique: Measure of the percentage of unique explanations over all generated explanations. A higher uniqueness indicates that the model-agnostic approach effectively generates a more specific (i.e. less repetitive) explanation.

%Consistency: Measures the percentage of the defect-introducing commits in the testing data that have characteristics satisfying the rule feature in the generated explanation. A higher *%Consistency* indicates that the model-agnostic methods effectively generate explanations that are consistent with the actual characteristics of the introduced defects.

4.4. Statistical testing

To verify whether CfExplainer is significantly different compared to other methods, two statistical tests (i.e., Wilcoxon singed-rank test and Cliff's |δ| effect size) were used to determine whether there is a significant difference between the methods. Where Wilcoxon singed-rank test is a nonparametric test that measures the difference in distribution between two population, difference exists when p-value is <0.05; Cliff's delta |δ| test is a non-parametric effect size test that measures significant differences between two methods. The Wilcoxon singed-rank test and Cliff's delta |δ| effect size test are often used in the SDP field to test whether there is a significant difference between methods (Rajapaksha et al., 2021; Pornprasit et al., 2021) . Therefore, we used the Wilcoxon singed-rank test and Cliff's delta |δ| to test for differences between the methods. Cliff's delta |δ| is divided into four levels, as shown in Table 6; larger levels of validity are associated with significant differences.

Table 6
Cliff's delta (|δ|) effect size.

Cliff's delta (|δ|)	Effect size level
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.330 \leq \|\delta\| < 0.474$	Medium
$0.474 \leq \|\delta\|$	Large

5. Analysis of results

5.1. RQ1: how similar are the synthetic instances generated by CfExplainer to the instances to be explained?

To evaluate the effectiveness of the CfExplainer explanation, we measured it in terms of instance similarity and local model fitness. To measure the instance similarity, we used the Euclidean distance measure, calculated using Eq. (12).

$$d(i_1, i_2) = \sqrt{\sum_{j=1}^{n} \left(i_{1j} - i_{2j}\right)^2} \tag{12}$$

where $d(i_1, i_2)$ are the Euclidean distances between instances i_1 and i_2; the smaller the distance, the higher the similarity, n is the number of features.

We used the following equation to calculate the relative percentage difference in Euclidean distance: $\%diff = \frac{\text{CfExplainer} - \text{Baseline}}{\text{Baseline}} \times 100$.

Analysis of instance similarity results: The similarity of synthetic instances generated by CfExplainer with to be explained instances is improved by 26.5 %−34.2 %, 61.3 %−81.3 % compared to PyExplainer, LIME.

On the OpenStack dataset, the median Euclidean distance between synthetic instances generated by the CfExplainer (counterfactual generation), PyExplainer (cross-mutation generation), and LIME (random perturbation generation) and instances to be explained were 318, 483, and 821, respectively. CfExplainer provided improvement of 34.2 % and 61.3 % compared to PyExplainer and LIME, respectively. On the Qt dataset, the median Euclidean distances between synthetic instances and instances to be explained generated by CfExplainer, PyExplainer, and LIME were 342, 465, and 1831, respectively. CfExplainer provided improvement of 26.5 % and 81.3 % compared to PyExplainer and LIME, respectively. The experimental results are shown in Table 7. On both OpenStack and Qt datasets, the p-values of CfExplainer versus PyExplainer and LIME on the instance similarity metric are 0.018, 0.004, respectively, and Cliff's |δ| values are all 1, which indicates that CfExplainer has significant differences in instance similarity compared to PyExplainer and LIME.

The experimental results show that the synthetic instances generated by our CfExplainer are closer to the instances to be explained compared to PyExplainer and LIME. CfExplainer, PyExplainer and LIME use counterfactuals, cross-mutation, and random perturbation to generate synthetic instances, respectively, and the different methods of synthetic instance generation will generate different synthetic instances, and there will be large differences in instance similarity, and instance similarity will seriously affect the effectiveness of the explanation. Since LIME adopts random perturbation method to generate synthetic instances, the random perturbation method generates synthetic instances by adding a normally distributed value to the eigenvalues of the instances to be explained randomly, but this perturbation method is not applicable to sparse and high-dimensional data like the JIT dataset, and it does not take into account the characteristics of the actual JIT dataset. In addition, PyExplainer first selects a set of initial instances and then generates synthetic instances by performing crossover and mutation operations on the feature values. The crossover and mutation operations are shown in

Table 7
Instance similarity metrics.

Model-agnostic methods	OpenStack	Qt
CfExplainer	318	342
PyExplainer	483	465
LIME	821	1831

Eqs. (13) and 14.

$$I_{crossover} = i_1 + (i_2 - i_1) * \alpha \tag{13}$$

$$I_{\text{mutation}} = i_1 + (i_2 - i_3) * u \tag{14}$$

where i_1, i_2, i_3 are randomly selected instances, α is a random number between 0 and 1, and u is a random number between 0.5 and 1.

Although PyExplainer is able to generate more similar synthetic instances compared to the LIME method, the selection of initial instances can seriously affect the results of synthetic instance generation. In addition, the synthetic instances generated by the random numbers α and u during the process of cross-mutation have a certain degree of randomness, and the values of the features may be changed significantly during the process of cross-mutation, which can all seriously affect the similarity of the instances. However, CfExplainer generates synthetic instances by minimally perturbing the features through counterfactuals, which ensures the similarity with the original instances. Therefore, the similarity between the synthetic instances generated by our CfExplainer and the instances to be explained is higher than PyExplainer, LIME. In conclusion, the similarity between the synthetic instances generated based on counterfactuals and the instances to be explained is higher than that of random perturbations and cross mutations, and the generation of synthetic instances based on counterfactuals can provide more accurate explanations of the model's prediction results. Therefore, the explanations generated by CfExplainer are closer to the instances to be explained than those generated by LIME and PyExplainer.

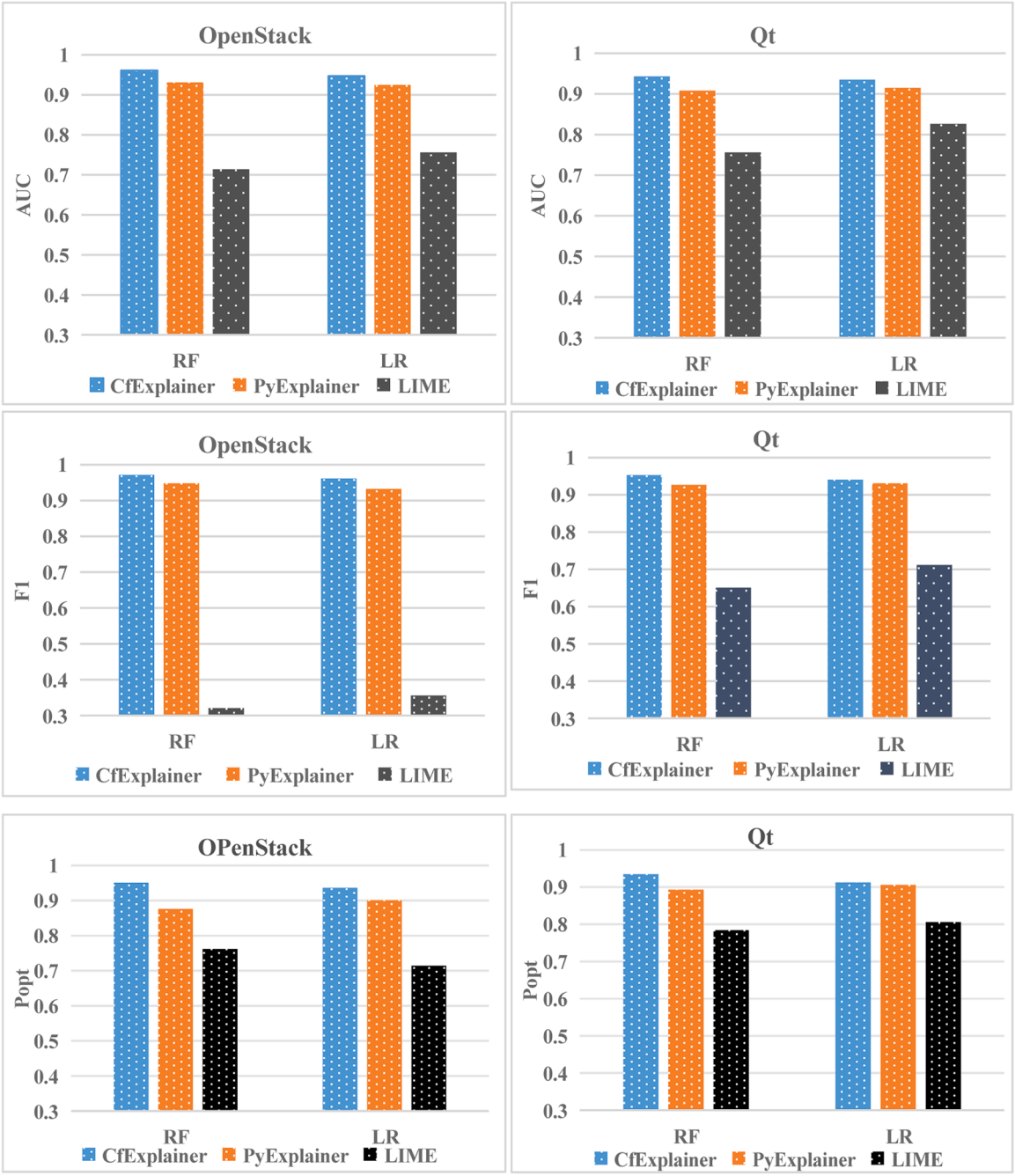


Fig. 7. Plot of the fitness of local model.

5.2. RQ2: does our CfExplainer have higher local model fitness than other baseline methods?

To evaluate whether the local model fit of CfExplainer outperforms other methods, we compared the fitness of the local models of CfExplainer (Association Rule), PyExplainer (RuleFit), and LIME (K-Lasso) using AUC, F1-score, and Popt metrics.

Analysis of local model fittness results: Compared with PyExplainer and LIME, CfExplainer improves 2.0 %−3.5 % and 10.9 %−24.7 % in AUC metrics, 2.3 %−3.0 % and 56.2 %−66.3 % in F1-score metrics, and 0.7 %−7.5 % and 10.7 %−22.2 % in Popt metrics. Meanwhile, CfExplainer has significant improvement in local model fittness in AUC, F1-score, and Popt metrics compared to PyExplainer and LIME.

On the AUC metric, CfExplainer was 0.961 and 0.943 for the OpenStack and Qt datasets, respectively. On the F1-score metric, CfExplainer achieved 0.971 and 0.953 on the OpensStack and Qt datasets, respectively. On the Popt metric, CfExplainer was 0.951 and 0.935 on OpenStack and Qt datasets, respectively. The experimental results are shown in Fig. 7. Furthermore, on the AUC metric, the p-values of CfExplainer versus PyExplainer and LIME are 0.125 and 0.005, respectively; on the F1-score metric, the p-values of CfExplainer versus PyExplainer and LIME are 0.109 and 0.008, respectively; on the Popt metric on the Popt metric, the p-value of CfExplainer versus PyExplainer and LIME is 0.125, 0.006, respectively, the p-value values are limited by the number of datasets, but the Cliff's |δ| values of CfExplainer versus PyExplainer are 1.0, 0.875, and 1, respectively. The Cliff's |δ| values of CfExplainer and LIME are 1, this suggests that the CfExplainer approach has large effect sizes and practical significance.

The experimental results show that our local model of CfExplainer outperforms PyExplainer and LIME, which shows that the local model of CfExplainer is able to learn the prediction behaviour of the global model better. The local model fittness is an important factor affecting the explanation effectiveness of CfExplainer. From the experimental results of RQ1, it can be seen that the similarity between the synthetic instances generated by CfExplainer and the to-be-explained instances is better than that of PyExplainer and LIME, and the higher instance similarity of CfExplainer means that the generated synthetic instances are more similar to the distribution of the to-be-explained instances' data in the test set, which makes CfExplainer's local model able to learn the characteristics of the to-be-explained instances better. In addition, CfExplainer's local model-like association rules have better prediction performance in SDP scenarios, and the contribution of features to defect prediction is taken into account, and the process of generating, sorting, pruning, and predicting rules is optimised by the new rule metric weighted support, which improves the fittness of the local model. Therefore, the local model fittness of CfExplainer is better than that of PyExplainer and LIME.

5.3. RQ3: how reliable is the CfExplainer explanation?

Model-agnostic methods often use random perturbation or cross mutation methods to generate synthetic instances, and the generated synthetic instances are random, and different synthetic instances will be generated during different iterations, leading to the generation of different rule explanations for the local model, which seriously affects the uniqueness of the explanations, i.e., the generated explanations are not able to be specific to the to-be-explained instances. Therefore, we investigate the uniqueness of the explanation by measuring whether the generated explanation can be specific to the to-be-explained instance through the *%Unique* metric.

Analysis of *%Unique* results: The explanations generated by CfExplainer are improved by 2.6 %−4.7 % and 28.2 %−35.8 % in *%Unique* metrics compared to PyExplainer and LIME. Meanwhile.

On the OpenStack dataset, the median *%Unique* of CfExplainer is 0.971, which increases by 2.7 %−4.7 % and 33 %−34.9 % compared with PyExplainer and LIME methods, respectively. On Qt dataset, the median *%Unique* of CfExplainer method is 0.953, which increases by 2.6 %−4.0 % and 28.2 %−35.8 % compared with PyExplainer and LIME methods, respectively. The experimental results are shown in Fig. 8. On OpenStack and Qt datasets, The p-values of CfExplainer versus PyExplainer and LIME on the %Unique metric are 0.062 and 0.005, respectively, and Cliff's |δ| values are all 1, this suggests that the CfExplainer approach has large effect sizes and practical significance

The experimental results show that the uniqueness of our CfExplainer explanations is better than PyExplainer and LIME, which suggests that CfExplainer is able to generate explanations with higher uniqueness.PyExplainer and LIME generate synthetic instances by using cross-mutation and random perturbation, respectively. There is a certain degree of randomness in the synthetic instances, and the synthetic instances generated by both of them in different iteration processes will have large differences, which seriously affects the uniqueness of the explanation, while CfExplainer generates synthetic instances based on the counterfactuals in different iteration processes, which are made by minimally perturbing the features so as to make the global model prediction results change, and by performing conditional constraints on the perturbation of the feature values, which make the synthetic instances generated in different iterative processes are stable and not random. Therefore, CfExplainer generates synthetic instances that are stable and generate more unique explanations than LIME and PyExplainer.

Model-agnostic methods use random perturbation or cross-mutation methods to generate synthetic instances, and when features are perturbed, instances that do not conform to real-world laws (anomalous instances) are generated, and these anomalous instances can seriously affect the consistency of the explanation. CfExplainer mines the weighted class association rules through the synthetic instances, and when the instances to be explained are predicted to be defective, the rules with the highest weighted support in the set of defective rules are selected to explain the prediction results of the global model, and conversely, the rule with the highest weighted support in the set of non-defective rules is selected to explain the prediction of the global model. Therefore, we studied the consistency of the explanations, measured by the *%Consistency* metric, whether CfExplainer can efficiently generate explanations that are consistent with the introduced defects.

Analysis of *%Consistency* Results: The explanations generated by CfExplainer are improved by 2.5 %−5.9 % and 9.3 %−42.6 % in *%Consistency* metrics compared to PyExplainer and LIME.

On the OpenStack dataset, the median *%Consistency* of CfExplainer was 0.806. Compared to PyExplainer and LIME, CfExplainer yielded improvement by 3.6 %−3.9 % and 24.5 %−24.9 %, respectively. On the Qt dataset, the median *%Consistency* of CfExplainer was 0.784. Compared to PyExplainer and LIME, CfExplainer yielded improvement by 2.5 %−5.9 % and 9.3 %−42.6 %, respectively. The experimental results are shown in Fig. 9. Furthermore, the p-values of CfExplainer versus PyExplainer and LIME on the %Consistency metric are 0.109, 0.008, respectively, and Cliff's |δ| values are all 1, this suggests that the CfExplainer approach has large effect sizes and practical significance

The experimental results show that the consistency of our CfExplainer explanations is better than PyExplainer and LIME, which suggests that CfExplainer is able to generate higher consistency explanations.PyExplainer and LIME use cross-mutation and random perturbation to generate synthetic instances, respectively, which have randomness, and both may generate synthetic instances (anomalous instances) that deviate from the real-world laws, i.e., the synthetic instances lack truthfulness, and the local model is constructed based on synthetic instances, which in turn leads to the knowledge learnt by the local model is not in line with the real world, i.e., the mined rules lack reliability. However, CfExplainer generates synthetic instances based on counterfactuals, sets artificial constraints when perturbing features, and generates synthetic instances with truthfulness, which enables CfExplainer's local model to learn knowledge that conforms to the real-world laws, i.e., it can correctly explain the real reasons for introducing defects. In addition, CfExplainer constructs local models by mining

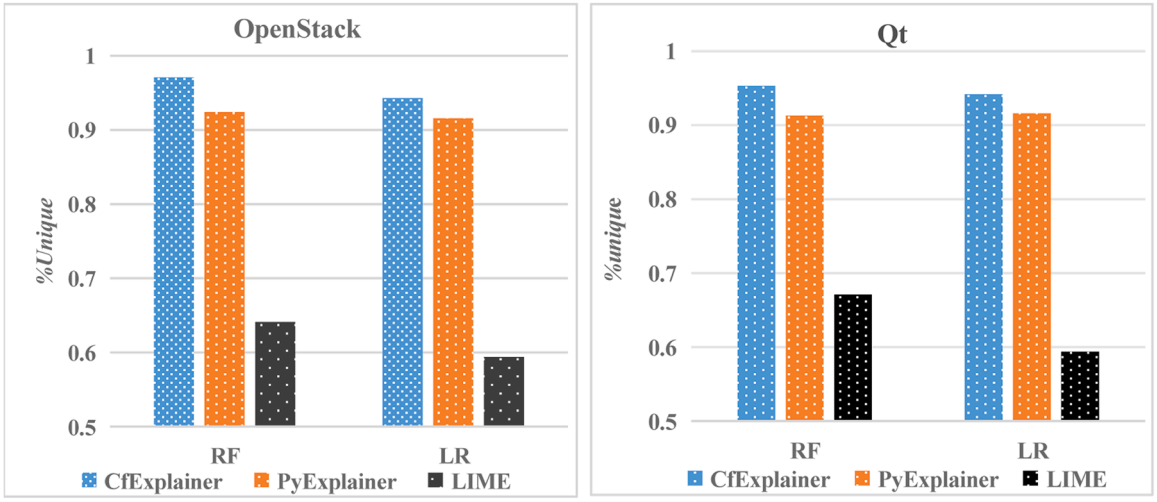


Fig. 8. Explanation of *%Unique*.

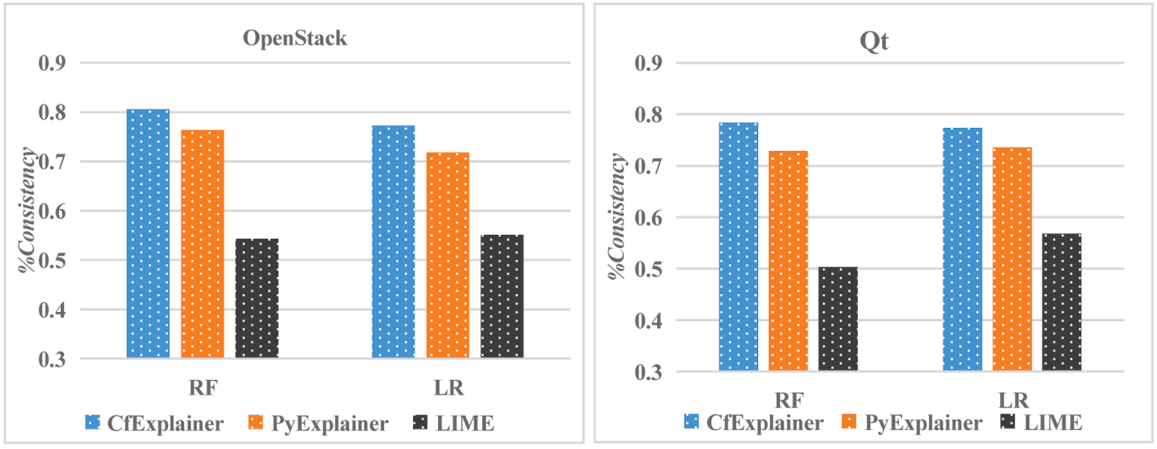


Fig. 9. Explanation of *%Consistency*.

weighted class association rules, considers the contribution of features to defects, uses weighted support measure to measure the importance of rules, and optimises the process of generating, ranking, pruning, and prediction of rules, and does filtering of redundant and conflicting rules, which improves the quality of the rule set, and thus improves the reliability of explanations generated by the local models. Therefore, the explanations generated by CfExplainer have higher consistency than LIME and PyExplainer.

5.4. Parameter setting

To reduce the influence of the parameters on the experiments, we explored the influence of the minimum support (minsupport) threshold on the fitness of the weighted class association rules. Minsupport thresholds for class association rules on different datasets tend to be different, while different minsupport thresholds affect the number of rules generated; the smaller the minsupport, the greater the number of rules generated, and vice versa. This in turn leads to different rule ranking results, which ultimately affects the quality of rule mining and the accuracy of prediction. Therefore, we studied the effect of different minsupport values on the fitness of the weighted class association rules. To set the appropriate minsupport threshold, we determined the best minsupport based on a grid search.

As shown in Fig. 10, when minsupport is 0.05–0.15, the fitness of the weighted class association rules increases as minsupport increases.

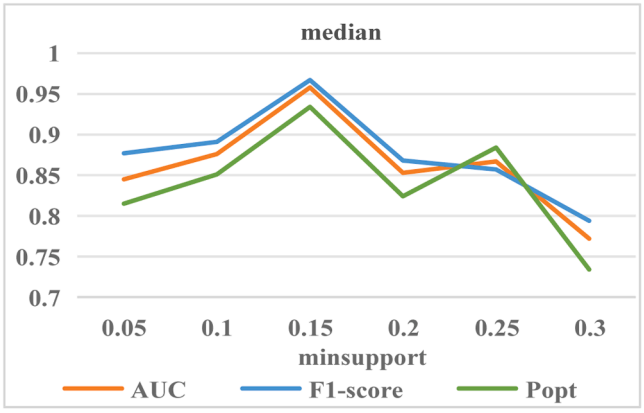


Fig. 10. Effect of minsupport on goodness of fit.

When minsupport is 0.15–0.2, the fitness of the weighted class association rules decreases. When minsupport is 0.2–0.25, the fitness slowly increases and then decreases again after 0.25.

The prediction performance of the model has an impact on the effectiveness of the CfExplainer method to a certain extent. Therefore, we studied the impact of just-in-time defect prediction model parameters on the interpretation performance of CfExplainer by selecting

representative parameters of random forests and logistic regression, i.e., n_estimator and the regularization parameter penalty (Contreras et al., 2021; Li et al., 2022) .

As shown in Fig. 11, the impact of n_estimators and penalty on method performance is demonstrated. On the %Unique and %Consistency metrics, %Unique and %Consistency increased when n_estimators increased from 50 to 150, but then %Unique and %Consistency gradually decreased as n_estimators continued to increase.

In logistic regression models, the regularization parameter penalty has a significant impact on the performance of the method. L1 regularization, L2 regularization, and Elasticnet are three commonly used regularization methods that prevent overfitting by adding penalties to the loss function associated with the model parameters. When the logistic regression model uses L1 regularization, the *%Unique* and *%Consistency* metrics are 0.926 and 0.786, respectively. When the logistic regression model uses L2 regularization, the *%Unique* and *%Consistency* metrics are 0.854 and 0.741, respectively. When the logistic regression model uses Elasticnet regularization, the *%Unique* and *%Consistency* metrics are 0.883 and 0.762, respectively.

6. Threats to validity

6.1. Internal threats

Minsupport must be set manually when mining weighted association rules, which rely on prior knowledge. Minsupport thresholds for class association rules on different datasets usually differ, and setting minsupport usually affects the number of rules generated; the smaller minsupport, the more rules are generated, and vice versa. This leads to different results in ranking rules and ultimately affects the rule-mining quality and prediction accuracy. To reduce the randomness of the conclusions, the number of synthetic instances generated by CfExplainer is set to 2000.

6.2. External threats

Because of the limited datasets that meet traceability, rapid iteration, and code review policies, consistent with other approaches in this field, we only selected OpenStack and Qt datasets. In future research, to further validate CfExplainer's generalisability, we will validate it on more standard datasets.

7. Conclusion

We herein proposed CfExplainer, a model-agnostic approach based on local rules. CfExplainer uses counterfactuals to generate synthetic instances, mines weighted class association rules based on synthetic instances, optimises the processes of generating, ranking, pruning, and predicting class association rules, and uses the highest-priority rules to explain the predictions of the model. CfExplainer explanations outperformed those of other methods in terms of effectiveness and reliability. The explanations generated by CfExplainer improved the transparency of JIT defect prediction models. CfExplainer can thus help developers analyse the causes of defects, provide developers with guidance on defect fixes, and reduce the risks of introducing defects.

Some current model interpretation methods are designed for in-project software defect prediction based on successive versions of software, but this requires multiple successive versions of software for training and evaluation. Therefore, they are not applicable to JIT defect prediction models, while CfExplainer is performed in an just-in-time defect prediction scenario, which is suitable for open scenarios with rapid software iteration, and the generated explanations can improve the transparency of just-in-time defect prediction models, help developers analyze the causes of defects, and provide developers with guidance on defect fixing to reduce the risk of introducing defects. Developers understand the model's prediction results through the explanations generated by CfExplainer. For example, CfExplainer provides rule interpretations for instances to be interpreted (e.g., 29<churn<100 and ns>1⇒Defect), which are predicted to be defective because churn is greater than 29 and <90, and ns is greater than 1. The rule interpretations help testers to focus on the aspects that are most relevant to the risk of defects, rather than focusing on other, less important aspects, and thus improve the efficiency of software fixes. In order to improve the usefulness of CfExplainer in explaining SDP models, more dimensional features will be mined through feature engineering techniques in the future to enhance the model's explaining capability.

The minsupport settings depend on prior knowledge. Therefore, to reduce the potential effects of prior knowledge, we will consider adaptive minsupport in future studies. Furthermore, the quality of synthetic instances generated based on counterfactuals has significant effects on the effectiveness and reliability of the explanations. Therefore, we will further study optimisation methods for the counterfactual generation of synthetic instances.

Author contributions

All authors are contributed equally.

CRediT authorship contribution statement

Fengyu Yang: Supervision, Funding acquisition. **Guangdong Zeng:** Writing – original draft, Validation, Software, Methodology. **Fa Zhong:**

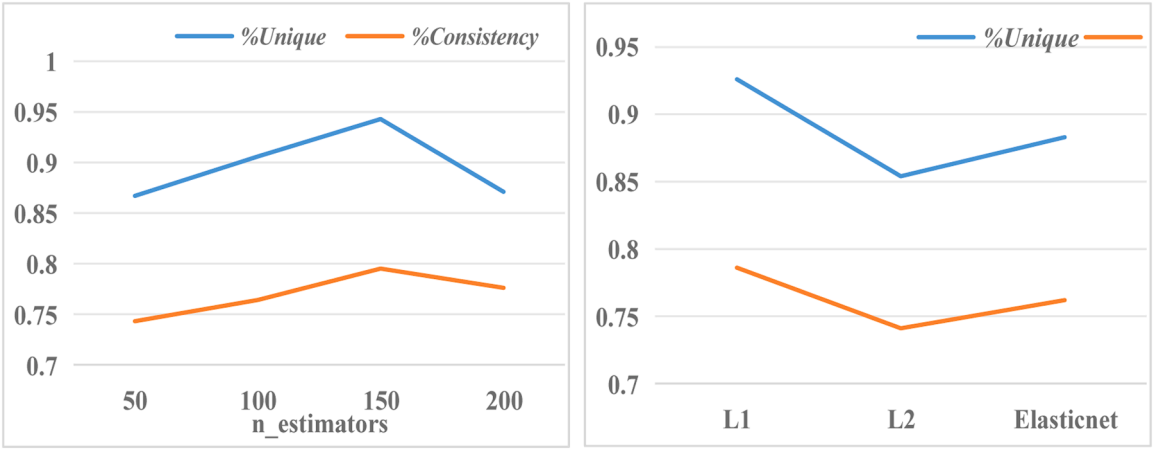


Fig. 11. Effect of model classification parameters on CfExplainer.

Visualization, Conceptualization. **Peng Xiao:** Supervision, Conceptualization. **Wei Zheng:** Supervision, Conceptualization. **Fuxing Qiu:** Supervision, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the link to my data/code at the manuscript.

Funding

This paper is supported by the National Natural Science Foundation Project (61762067).

Availability of data and materials

The dataset used in this paper at https://github.com/software-rebels/JITMovingTarget. To foster the replication of our study, we will publish the implementation of our method at https://github.com/software-defetc-prediction-research/CfExplainer.

Acknowledgment

We are grateful for the generous support of this work by the National Natural Science Foundation Project[grant numbers 61762067]. We would like to thank Editage (www.editage.cn) for English language editing.

References

Balaram, A., Vasundra, S., 2022. Prediction of software fault-prone classes using ensemble random forest with adaptive synthetic sampling algorithm. Autom. Softw. Eng. 29 (1), 6.

Chen, H., Jing, X.Y., Zhou, Y., et al., 2022. Aligned metric representation based balanced multiset ensemble learning for heterogeneous defect prediction. Inf. Softw. Technol. 147, 106892.

Cheng, F., Ming, Y., Qu, H., 2020. Dece: decision explorer with counterfactual explanations for machine learning models. IEEE Trans. Vis. Comput. Graph. 27 (2), 1438–1447.

Cito J., Dillig I., Murali V., et al. Counterfactual explanations for models of code//Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. 2022: 125–134.

Contreras, P., Orellana-Alvear, J., Muñoz, P., et al., 2021. Influence of random forest hyperparameterization on short-term runoff forecasting in an andean mountain catchment. Atmosphere (Basel) 12 (2), 238.

Gesi J., Li J., Ahmed I. An empirical examination of the impact of bias on just-in-time defect prediction//Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2021: 1–12.

Giray, G., Bennin, K.E., Köksal, Ö., et al., 2023. On the use of deep learning in software defect prediction. J. Syst. Softw. 195, 111537.

Gosiewska, A., Biecek, P., 2019. IBreakDown: uncertainty of model explanations for non-additive predictive models. arXiv preprint arXiv:1903.11420.

Hanmer, R., Mendiratta, V., 2022. Data Analytics: Predicting Software Bugs in Industrial Products[M]//System Dependability and Analytics: Approaching System Dependability from Data, System and Analytics Perspectives. Springer International Publishing, Cham, pp. 39–53.

Ibrahim, M.F., Nurhakiki, F.R., Utama, D.M., et al., 2021. Optimised genetic algorithm crossover and mutation stage for vehicle routing problem pick-up and delivery with time windows. In: IOP Conference Series: Materials Science and Engineering, 1071. IOP Publishing, 012025.

Jia Y., Bailey J., Ramamohanarao K., et al. Improving the quality of explanations with local embedding perturbations//Proceedings of the 25th ACM SIGKDD International conference on knowledge discovery & Data Mining. 2019: 875–884.

Jiarpakdee, J., Tantithamthavorn, C.K., Dam, H.K., et al., 2020. An empirical study of model-agnostic techniques for defect prediction models. IEEE Trans. Softw. Eng. 48 (1), 166–185.

Joloudari, J.H., Marefat, A., Nematollahi, M.A., et al., 2023. Effective class-imbalance learning based on SMOTE and convolutional neural networks. Appl. Sci. 13 (6), 4006.

Kato, H., Hanada, H., Takeuchi, I., 2022. Safe rulefit: learning optimal sparse rule model by meta safe screening. IEEE Trans. Pattern. Anal. Mach. Intell. 45 (2), 2330–2343.

Khatri, Y., Singh, S.K, 2022. Towards building a pragmatic cross-project defect prediction model combining non-effort based and effort-based performance measures for a balanced evaluation. Inf. Softw. Technol. 150, 106980.

Kraemer, H.C., Morgan, G.A., Leech, N.L., et al., 2003. Measures of clinical significance. J. Am. Acad. Child Adolescent Psychiatry 42 (12), 1524–1529.

Li, J., Liang, K., Song, X., 2022. Logistic regression with adaptive sparse group lasso penalty and its application in acute leukemia diagnosis. Comput. Biol. Med. 141, 105154.

Liu, X., Li, Z., Zou, J., et al., 2022. An empirical study on multi-source cross-project defect prediction models. //. In: 2022 29th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 318–327.

Maragno, D., Röber, T.E., Birbil, I., 2022. Counterfactual explanations using optimization with constraint learning. arXiv preprint arXiv:2209.10997.

Mcintosh, S., Kamei, Y., 2018. Are Fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Trans. Softw. Eng. 412–428. https://doi.org/10.1109/TSE.2017.2693980.

Mehmood, I., Shahid, S., Hussain, H., et al., 2023. A novel approach to improve software defect prediction accuracy using machine learning. IEEE Access.

Mothilal R.K., Sharma A., Tan C. Explaining machine learning classifiers through diverse counterfactual explanations.//Proceedings of the 2020 conference on fairness, accountability, and transparency. 2020: 607–617.

Peng, K., Menzies, T., 2021. Defect reduction planning (using timelime). IEEE Trans. Softw. Eng. 48 (7), 2510–2525.

Pornprasit, C., Tantithamthavorn, C., Jiarpakdee, J., et al., 2021. Pyexplainer: explaining the predictions of just-in-time defect models. //. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 407–418.

Pornprasit, C., Tantithamthavorn, C.K., 2021. JITLine: a simpler, better, faster, finer-grained just-in-time defect prediction[C]. //. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, pp. 369–379.

Rajapaksha, D., Tantithamthavorn, C., Jiarpakdee, J., et al., 2021. SQAPlanner: generating data-informed software quality improvement plans. IEEE Trans. Softw. Eng. 48 (8), 2814–2835.

Ribeiro, M.T., Singh, S., Guestrin, C., 2016. Why should I trust you?" Explaining the predictions of any classifier[C]. //. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, pp. 1135–1144.

Roman A., Brozek R., Hryszko J. Predictive power of two data flow metrics in software defect prediction[C]//ENASE. 2023: 114–125.

Shankar, S.P., Naresh, E., Agrawal, H, 2022. Optimization of association rules using hybrid data mining technique. Innov. Syst. Softw. Eng. 18 (2), 251–261.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? ACM Sigsoft Softw. Eng. Notes 30 (4), 1–5.

Temraz, M., Keane, M.T, 2022. Solving the class imbalance problem using a counterfactual method for data augmentation. Mach. Learn. Appl. 9, 100375.

Uddin, M.N., Li, B., Ali, Z., et al., 2022. Software defect prediction employing BiLSTM and BERT-based semantic feature. Soft. Comput. 26 (16), 7877–7891.

Vuttipittayamongkol, P., Elyan, E., Petrovski, A., 2021. On the class overlap problem in imbalanced data classification. Knowl. Based. Syst. 212, 106631.

Wachter, S., Mittelstadt, B., Russell, C., 2017. Counterfactual explanations without opening the black box: automated decisions and the GDPR. Harv. JL & Tech. 31, 841.

Wang, S., Luo, H., Huang, S., et al., 2023. Counterfactual-based minority oversampling for imbalanced classification. Eng. Appl. Artif. Intell. 122, 106024.

Zafar, M.R., Khan, N., 2021. Deterministic local interpretable model-agnostic explanations for stable explainability. Mach. Learn. Knowl. Extract. 3 (3), 525–541.

Zhang, H., Kuang, L., Wu, A., et al., 2023. Just-in-time defect prediction enhanced by the joint method of line label fusion and file filtering. IET Softw.

Zheng, W., Shen, T., Chen, X., 2021. Just-in-time defect prediction technology based on interpretability technology[C]. //. In: 2021 8th International Conference on Dependable Systems and Their Applications (DSA). IEEE, pp. 78–89.

Zheng, W., Shen, T., Chen, X., et al., 2022. Interpretability application of the Just-in-Time software defect prediction model. J. Syst. Softw. 188, 111245.

Zhuang, W., Wang, H., Zhang, X., 2022. Just-in-time defect prediction based on AST change embedding. Knowl. Based. Syst. 248, 108852.

Fengyu Yang received the master degree from Zhejiang University of Technology, in 2006. He is currently an Associate Professor and a Master Supervisor with the School of Software, Nanchang Hangkong University. His research interests include software testing and big data analytics.

Guangdong Zeng is currently pursuing the M.S. degree with the School of Software, Nanchang Hangkong University. His research interests include software defect prediction and association rules.

Fa Zhong is currently pursuing the M.S. degree with the School of Software, Nanchang Hangkong University. His research interests include software testing and software defect prediction.

Peng Xiao received the Ph.D. degree from Beihang University, in 2018. He is currently a Lecturer and a Master Supervisor with the School of Software, Nanchang Hangkong University. His research interest include software testing and software reliability.

Wei Zheng received the Ph.D. degree from Xidian University, in 2010. He is currently a Professor and a Master Supervisor with the School of Software, Nanchang Hangkong University. His research interests include complex networks and software reliability.