Just-in-Time Defect Prediction Technology based on Interpretability Technology

Wei Zheng
Northwestern Polytechnical University
School of Software
Xi'an, China.
wzheng@nwpu.edu.cn

Tianren Shen
Northwestern Polytechnical University
School of Software
Xi'an, China.
834562113@qq.com

Xiang Chen
Nan Tong University
School of Information Science and Technology
Nan Tong, China
xchencs@ntu.edu.cn

***Abstract*—In recent years, in the field of software defect prediction, researchers have proposed the Just-in-Time defect prediction technology, which can predict whether there are defects in each code change submitted by developers. This method is instant and easy to trace. However, the accuracy of Just-in-Time defect prediction is affected by the imbalance of data set categories. 20% of the defects in the software engineering field may exist in 80% of the modules. In most cases, code changes that do not cause defects account for a larger proportion. Therefore, there is an imbalance rate in the data set, that is, the imbalance between the minority and majority categories, which will affect the classification prediction effect of the model. Most types, that is, code changes that will not produce defects will make the model have an artificially high prediction accuracy, and it is difficult to obtain the expected results in practical applications. Moreover, the data set features contain many irrelevant features and redundant features, which will also increase the complexity of the prediction model. In order to improve the prediction efficiency of just in time defect prediction. Improve the interpretability and transparency of the model and establish the trust relationship between users and decision-making model. For this reason, we have established a RandomForest defect prediction model, using multiple different types of change features to study 6 open source projects from different fields. The model is explained to a certain extent using LIME interpretability technology . Using interpretability methods to extract features and trying to reduce the developer's workload as much as possible. Our research results show that through the interpretability of the defect prediction model and identifying key features, 45% of the original workload can be used, and 96% of the original work effect can be achieved.**

***Index Terms*—software metrics, Just-in-Time defect prediction,interpretability, interpretation method**

I. Introduction

Software defect prediction has always been the most important research field in software engineering research. Due to the complex nature of the software system, software defects cannot be avoided. Software defects refer to defects and problems that exist in software products that cause the product to fail to meet the software requirements and its specifications, and need to be repaired [1], the existence of software defects, extremely.

This greatly restricts the application and development of software and brings great economic losses. According to the estimation of the National Institute of Standards and Technology (NIST), it is a year caused by software defects in the United States. The economic loss is as high as 60 billion U.S. Dollars. Through further research, NIST discovered, identified and repaired these software defects, which can help the United States save 22 billion U.S. Dollars [2]. Therefore, repairing defects has become a key activity in software maintenance, but it also consumes a lot of time and resources [3]. The data shows that the cost of repairing defects accounts for 50% to 75% of the total cost of software development [4]. Finding defects in time can help reduce the cost of repairing defects and improve software quality.

Initially, in order to cope with this thorny problem, researchers in the field of software engineering proposed software defect prediction technology. However, in traditional defect prediction, the main purpose is to predict coarse-grained software entities, such as files, modules, and packages. The software defect prediction model may predict that a huge file has defects, but for developers, checking the entire file will consume a lot of time and effort, and the file may have been modified by multiple people. Find a suitable developer It is not simple. Although these prediction models have certain advantages, in some cases they can find defects in time, but in the actual application of software development environment, the shortcomings of coarse-grained prediction methods are undoubtedly exposed. In the face of huge software development packages, it is obvious that coarse-grained prediction methods are exposed. The software defect prediction technology of the company has been unable to meet the timely discovery of software defects, and there is an obvious problem of inefficiency [5].

In order to cope with the above-mentioned challenges, we propose the just-in-time defect prediction technology, which refers to the technology that predicts whether there are defects in each code change submitted by the developer. In Just-in-Time defect prediction, the predicted software entity is a code change. The Just-in-Time defect prediction technology is instantaneous, which is embodied in the fact that this prediction technology can change the code after the developer submits a code change Carry out defect analysis to predict the possibility of defects. This technology can effectively deal with the challenges faced by traditional defect prediction technologies, which are mainly reflected in the following three aspects:

2767-6684/21/$31.00 ©2021 IEEE
DOI 10.1109/DSA52907.2021.00017

(1) **Fine-grained.** Compared with module or file-level defect prediction, change-level prediction focuses on more fine-grained software entities. Developers can spend less time and effort to review code changes predicted to be defective.

(2) **Just-in-Time.** Just-in-Time defect prediction technology can predict when the code change is submitted. At this time, the developer still has a vivid memory of the changed code, and does not need to spend time to re-understand the code change submitted by himself, which is helpful for more Fix defects in time.

(3) **Easy to trace.** The developer's information is saved in the code changes submitted by the developer. Therefore, the project manager can more easily find the developer who introduced the defect, which helps to analyze the cause of the defect in time and help complete the defect assignment [6].

In previous related studies, a batch of instant defect prediction research results have been produced, but there are also some limitations. Among them, Mockus and Weiss first proposed defect prediction at the code change level [7], In their prediction technology,the predicted software entity is a code change combination composed of multiple code change submissions, but they only analyzed a large-scale telecommunication's system. Kim et al. first proposed defect prediction for each code change in TSE in 2008 [8]. Kamei et al. called this defect prediction technology for the first time in TSE in 2013 [9]. Early research did not consider the work required for comprehensive quality assurance, such as testing and code review. Early research only considered indicators in a specific environment. In recent years, instant defect prediction technology has become a research hotspot in the field of defect prediction due to its fine-grained, instantaneous and traceable advantages. In a large amount of work, researchers have focused on data annotation, feature extraction, model construction, model evaluation, etc. A large number of valuable theories and technologies have been put forward. In addition, instant defect prediction has also attracted attention from the industry. For example, Shihab et al. conducted an empirical study on 60 teams of a large software company [10], and Kamei et al. conducted an empirical study on five company projects [9], demonstrating the practicality of instant defect prediction technology. It can be seen that: instant defect prediction technology has attracted the attention of software engineering academia and industry, and has produced a number of excellent research results, but also faces the lack of unified modeling Challenges of technology and evaluation indicators. Whatever, there is currently no research work to sort out and summarize the current research progress in this field. In view of this, this article intends to focus on the current research progress of instant defect technology, from data annotation, feature extraction, model construction and model evaluation and other aspects are sorted out, conducted and summarized, and the main problems and future development directions in the field are summarized. In this paper, we use the data set of six open source projects published by Kamei [9], including There are 220,000 changes, and different types of change features are used to better conduct our experimental research.

Although machine learning outperforms humans in many meaningful tasks, its performance and application are also questioned due to the lack of interpretability. For ordinary users, the machine learning model is like a black box. We give it an input and feed back a decision result. No one can know the decision basis behind it and whether its decision is reliable. The lack of interpretability may pose a serious threat to many practical applications based on machine learning in practical tasks, especially in security sensitive tasks. For example, the lack of interpretable automatic medical diagnosis model may bring wrong treatment plans to patients, and even seriously threaten the life safety of patients. Therefore, the lack of interpretability has become one of the main obstacles to the further development and application of machine learning in real tasks.

How should "explain a model" itself be understood? For humans, it refers to providing visual text or images so that people can intuitively understand the prediction results based on the model. More generally speaking, it is the process of the model "self-proving innocence".

For example, the example shown in the Fig.1 describes the process of a model used for "assisted seeing a doctor" to prove its credibility to doctors: the model not only has to give its prediction result (flu), but also provides the result of this The basis of the conclusion-sneeze, headache and no fatigue (counter-example). Only by doing so can doctors have reason to believe that its diagnosis is justified and well-founded to avoid the tragedy of "frustrated life".

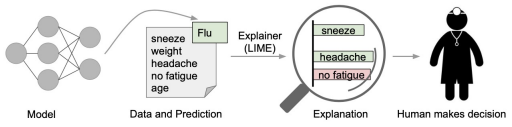

Fig. 1. **Explaining individual predictions. A model predicts that a patient has the flu, and LIME highlights which symptoms in the patient's history led to the prediction. Sneeze and headache are sneeze portrayed as contributing to the "flu" prediction, while "no fatigue" is evidence against it. With these, a doctor can make an informed decisiohneaadabcoheut the model's prediction. [33]**

We illustrate our research in the form of three research questions:

- RQ1: How efficient is our prediction model?
  Mockus and Weiss only used a large-scale telecommunication's system project to evaluate their predictive model [9]. In order to better evaluate the efficiency of our model, we use the data set of six open source projects published by Kamei [9]. In order to better identify the defects caused by code changes, according to a series of new ones proposed by Kamei The characteristics of the code changes are used to build a new defect prediction model. We can predict the defects caused by the entire code change with 68% accuracy and 64% recall.
- RQ2: What features can be used to determine which features play a significant role in prediction through

interpretability techniques?
The existing Just-in-Time defect prediction technology only predicts the possibility of defects in the change, and there is currently no relevant research on what the predicted defect is, such as the type and location of the defect. The defect type describes the cause and characteristics of the defect, the defect location refers to the module, file, function and even code line where the defect is located. Knowing the defect type and location can help developers to quickly repair the defect. Although researchers have proposed many defect classification and defect location techniques for immediate defects There is no relevant research on the predicted defect classification and defect location. We use the interpretable model to find out the NF (number of files), the relative loss measure (LA/LF and LT/NF), and the time interval between the last change and the current change. (PD) is the most important risk factor.

- RQ3: After removing unimportant features, what is the performance of the defect model?
At this stage, Just-in-Time software defect prediction still has the problem of heavy workload and low work efficiency. We hope that through preliminary screening, the most influential features can be screened out through interpretable models. Our research results show that we can only spend 45% of the original work and achieve 96% of the original work capacity.

II. BACKGROUND AND RELATED WORK

A. Just-in-Time defect prediction technology

Fig.2 shows the general process of instant defect prediction technology, which mainly includes three stages: data annotation, feature extraction, and model construction. Among them, the data annotation stage mainly relies on version control systems (such as git) and defect tracking systems (such as bugzlla or jira), the code changes are marked as defect changes (buggy) or non-defect changes (clean); the feature extraction stage mainly expresses code changes by extracting features of different dimensions; the model construction stage mainly relies on machine learning technology to build a predictive model. When new code changes are submitted, the model will predict the possibility of defects.

In recent years, Just-in-Time defect prediction technology has become a research hotspot in the field of defect prediction due to its advantages of fine-grained and instant traceability. Khuat TT et al [35]. empirically evaluated the importance of sampling for various classifier sets on unbalanced data in the software defect prediction problem. Combining sampling technology and integrated learning models, it is positive for data prediction with imbalance problems. The role of. Liu et al. used the information gain feature selection algorithm to optimize the features of the original data set, and combined with the polynomial Bayes algorithm to train and test the optimized data set. L Pascarella et al [36]. proposed a novel fine-grained model to predict the defective documents contained in the submission, and reduce the workload required to judge defects according to the classification performance and the degree to which the model. In the class weight learning stage, Hu et al. obtain the optimal weights of different classes through adaptive learning of class weights; then, in the training stage, use the optimal weights obtained in the previous step to train 3 base classifiers, and pass the soft The integrated method combines three base classifiers; finally, in the decision-making stage, a decision is made according to the threshold shift model. KK Bejjanki et al [37]. proposed class imbalance reduction (CIR). By considering the distribution characteristics of the data set, they proposed an algorithm to establish symmetry between defective records and non-defective records in an unbalanced data set. The combination of feature selection and ensemble learning is also a research hotspot in classification prediction. Wang et al. combined Relief feature selection algorithm and heterogeneous ensemble learning algorithm to identify three different cryptosystems.

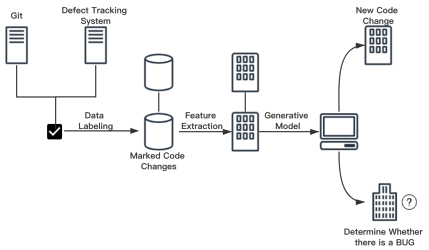

Fig. 2. **General framework of just-in-time defect prediction.**

B. LIME

LIME (Local Interpretable Model-agnostic Explanations) abbreviation. It can be seen from the name that the model is a partially interpretable model and an interpretable method that has nothing to do with the model itself. Use the trained local proxy model to interpret a single sample. Suppose that for the black box model that needs to be explained, take the sample of the concerned instance, generate new sample points by perturbing nearby, and obtain the predicted value of the black box model, and use the new data set to train an interpretable model (such as Linear Regression, Decision Tree) to get a good local approximation to the black box model. Each part of the name reflects our intention to explain. Its name also reflects its characteristics very well:

Local: Build a local linear model or other proxy model based on the predicted value you want to explain and the nearby samples;

Interpretable: The explanation made by LIME is easily understood by humans. Use a locally interpretable model to interpret the prediction results of the black box model,and construct the relationship between the local sample features and the prediction results;

Model-Agnostic: The algorithm explained by LIME has nothing to do with the model. Whether it is using various complex models such as Random Forest, SVM or XGBoost,

the prediction results obtained can be explained by the LIME method;

Explanations: LIME is an afterthought method;

The author puts forward four conditions that the interpreter needs to meet:

Interpretability: There are requirements for both models and features. Decision trees, linear regression and naive Bayes are all interpretable models, provided that the features are also easy to interpret. Otherwise, it is like Word Embedding, even a simple linear regression cannot be interpretable. And interpretability also depends on the target group, such as explaining these models to business people who don't understand the models. In contrast, linear models are easier to understand than simple Bayes.

Local fidelity: Now that we have used interpretable models and features, it is impossible to expect simple interpretable models to be equivalent to complex models (such as the original CNN classifier) in effect. Therefore, the interpreter does not need to achieve the effect of a complex model globally, but at least the effect must be close locally, and the local part here represents the surroundings of the sample we want to observe.

Model-independent: Any other model, such as SVM or neural network, the interpreter can work.

Global perspective: Accuracy, AUC, etc. are sometimes not a good indicator, we need to explain the model. The job of the interpreter is to provide an explanation of the sample to help people trust the model. **General Idea:** LIME aims to approximate the black-box model f with a simple function g around the point of interest x. g is required to lie into the class of explainable models G.

$$f : R^P \rightarrow R, black - box\, model$$
$$g : R^p \rightarrow R, explainable\, model$$

where P is the number of features employed by the black-box model, to make predictions about the response variable. The explainable model g uses only p of the original P variables, in order to reduce the complexity. Solving the following optimisation problem, we obtain the function g most similar to f in the neighbourhood of x.

$$\underset{g \in G}{argmin} L(f, g, \pi_x) + \Omega(g)$$
$$\Omega(g) : complexity\, of\, g$$
$$L : loss\, function$$
$$\pi_x : weight\, assigned\, according\, to\, x\, proximity$$

Chosen a given individual x, LIME returns a local explainable model g, which in turn provides the most important variables to predict the points in the x neighbour- hood.(Fig.3)

C. SP-LIME

Partially, we only explained the behavior of the model on one example, but this cannot see the behavior of the model as a whole. Therefore, we need more samples to help us observe the behavior of the model. However, choosing a suitable sample is a high threshold for users. Therefore, the SP

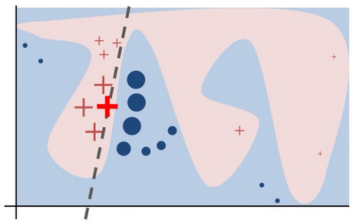

Fig. 3. **LIME's modus operandi. [33]**

(submodular-pick) LIME algorithm is proposed in the article to automatically search for suitable samples.

In the Fig.4, each row represents the interpretation of a sample, and each column represents a feature, that is, a weight in the interpretation. The algorithm selection examples are mainly considered from two aspects. The first is the diversity of features, that is, the selected examples should have rich features, and the second is the importance of features, that is, the selected features should be included in the decision-making process of the model. Quite a right to speak. As an example in the above figure, if the algorithm can fully consider the diversity, only one of the second and third examples will be selected because their explanations are very similar and cannot provide more information to explain the behavior of the model. In the field of immediate software defects,

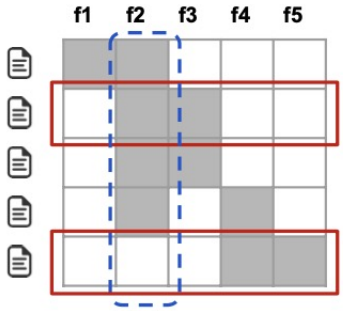

Fig. 4. **Toy example W. Rows represent in- Covered Features stances (documents) and columns represent features (words). Feature f2 (dotted blue) has the highest importance. Rows 2 and 5 (in red) would be selected by the pick procedure, covering all but feature f1 [33].**

there is less research on model interpretability. In this article, through LIME analysis, this article carries out heuristic feature combination and data processing, and the ultimate goal is to improve the accuracy of the Just-in-Time defect prediction model. Finally, LIME is used to analyze the relationship between the features in the model and the final prediction result,to explain the complex Just-in-Time defect prediction model.

III. Case Study Design

In this part, we mainly answer the preliminary preparations for the three questions we are studying. Here, we introduce

in detail the relevant information of the data set used in the experiment and the related processing of the data set.

A. Research Data

In previous experiments on defect prediction, the experiments of Mockus and Weiss and Kim et al. [7] [8]only examined the risk of code changes in a commercial or open source project. In our experiment, we used 6 different projects, which are very famous open source projects (Bugzilla, Columba, Mozilla, Eclipse JDT, Eclipse Platform, and PostgreSQL). The projects used are all written in JAVA. Previous studies on change risk only examined the risk of open source project changes [26], or only the risk of commercial project changes. To increase the generality of our results and produce more specific results, we used 11 different items. Six are large, well-known open source projects (ie Bugzilla, Columba, Mozilla, EclipseJDT, EclipsePlatform, and PostgreSQL). The project used is written in java. To conduct our case study, we extracted information from the project's CVS repository and combined it with the bug report. We use the data provided by the MSR2007 Mining Challenge to collect data from the Bugzilla and Mozilla projects. Data for Eclipse JDT and platform projects are collected from the MSR 2008 Mining Challenge. For Columba and PostgreSQL, we mirrored the official CVS repository.

In Table I, the statistical data related to all projects is summarized. The total number of code changes for all projects in this table, in parentheses next to it, shows the percentage of changes that will lead to defects in the total changes. Although a code change may cause one or more software defects, in our experiments, the exact number of software defects caused is not very important for our prediction model and experimental results. The table shows the average of the LOC at the file level and the change level, as well as the number of files modified each day, and the number of changes made each day. This table also shows the maximum and average number of developers who made changes to a single file. For example, X file is modified by four developers of A, B, C, D, Y file is modified by B, C developers, then the maximum number of development files modified for a single file is 4 and the average is 3.

B. Identify defects and introduce changes

In order to identify the defects caused by code changes, we use the SZZ algorithm. For a better understanding, we use a defect in the Apache project ActiveMQ (AMQ-1381) as a specific example to describe the four steps of the algorithm framework in detail.

(1) Identify defect repair changes. Scan all historical data stored in the version control system, that is, all code changes, and identify code changes containing defect IDs in the log. These code changes are identified as defect repair changes. As shown in Fig.4, The code change ID for identifying and repairing AMQ-1381 is 645599.

(2) Identify the defective code to be repaired. Use the diff algorithm implemented by the version control system to identify the lines of code that are changed and modified by the code to repair the defects. These modified lines of code are identified as defective code. As shown in the Fig.5, the code is Change #645599 The modified code contains a function declaration, where the Command parameter type is wrong, and its correct type is Object instead of Command.

(3) Identify possible defect introduction changes. Use the annotate command in the code version control system to trace back the code change submission history. The first change to the defect code is identified as a possible defect introduction change. As shown in Fig.5, the code change #447068 introduces the incorrect function declaration found in step 2.

(4) Noise data removal. Remove possible noise data from possible defect introduction changes. Noisy data refers to changes that have been mistakenly marked as introducing defects but do not actually introduce defects (false positive). Sliwerski et al. proposed that the introduction of defects Changes should be submitted before the defect is reported [11]. Therefore, code changes that are submitted later than the defect report time will be treated as noise in the implementation of SZZ. The code change #447608 shown in the figure is submitted at the time of the defect. Before the report was created, therefore, this code change was finally confirmed to be a code change that introduced defect AMQ-1381. In the Columba and PostgreSQL examples, we use an

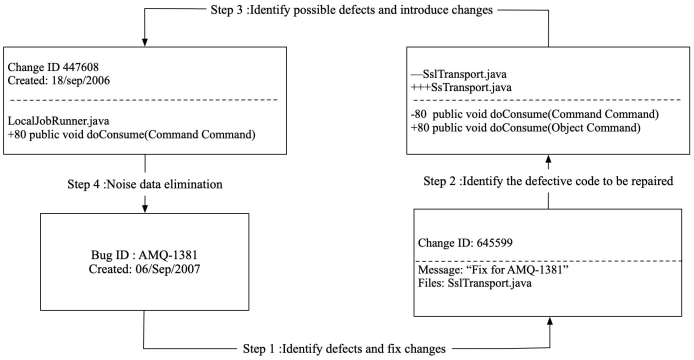

Fig. 5. **General framework of the SZZ algorithm.**

approximate algorithm (ASZZ) to identify whether a change is prone to defects, because there is no defect identifier quoted in the change log. In this case, we cannot verify whether the change that we determined to be a defect repair is really a defect repair. The algorithm only needs to find the keywords associated with the defect repair change (for example, "Fixed" or "Bug"), and assume that the change fixes the defect.

C. Data Processing

Target dimension: This dimension is used to characterize the goal of submitting code changes. The goals of developers submitting changes include repairing defects, implementing new features, refactoring, adding documents, etc. [16]. Research shows that changes to modify defects are more important than other types of changes Complex, more likely to

TABLE I
STATISTICS OF THE STUDIED PROJECTS

	Period	The total number of changes	Average LOC		# of modified files	# of changes	# dev.per file	
			File	Change	per change	per day	Max	Avg
Bugzila	08/1998 - 12/2006	4,620 (36%)	389.8	37.5	2.3	1.5	37.0	8.4
Columba	11/2002 - 07/2006	4,455(31%)	125.0	149.4	6.2	3.3	10.0	1.6
Eclipse JDT	05/2001 - 12/2007	35,386(14%)	260.1	71.4	4.3	14.7	19.0	4.0
Eclipse Platform	05/2001 - 12/2007	64,250(14%)	231.6	72.2	4.3	26.7	28.0	2.8
Mozilla	01/2000 - 12/2006	98,275(5%)	360.2	106.5	5.3	38.9	155.0	6.4
PostgreSQL	07/1996 - 05/2010	20,431(25%)	563.0	101.3	4.5	4.0	20.0	4.0
OSS-Median	-	27,909(20%)	310.1	86.7	4.4	9.4	24.0	4.0

introduce defects [11]. Therefore, in a large number of Just-in-Time defect prediction work, researchers use whether the change repairs the defect as a feature [9], [10], [18]–[20]. In addition, Shihab et al. also proposed to use Change the number of related defect reports to quantify the code change target [10].

Code distribution dimension: This dimension is used to characterize the distribution of change and modification code in related files. Research shows that for changes in which the modification code is distributed in multiple files, the developer needs to understand more code. Therefore, this more decentralized code Changes may introduce defects [31]. Kamei et al. proposed to use features such as the number of files, the number of folders, and the number of subsystems to be modified to quantify the distribution of changed codes [19]. Hassan proposed the use of entropy prediction of the distribution of changed codes Defects, and verified the effectiveness of this feature [34]. Therefore, Kamei et al. proposed to use this feature to predict defective changes [19].

Scale dimension: This dimension is used to characterize the scale of change and modification code. Moser et al. observed: The larger the scale of the change and modification code, the more likely it is to introduce defects [21]. In existing work, researchers have used changes to increase, Reduce the number of lines of code to quantify the size of the change. In addition, researchers use other granularities to quantify the size of the changed code. Shihab et al. proposed using changes to increase and decrease the number of code segments (chunks) to quantify the size of the change [10]. Kamei et al proposed to change the code line of the relevant document before the change is submitted to quantify the scale of the change [6]. At the same time, Kamei et al. found that the increase in the number of lines and the reduction in the number of lines of code are highly correlated. In order to avoid this correlation from affecting the prediction model , Kamei et al. proposed to use the relative increase in the number of lines and the relative decrease in the number of lines to quantify the scale of the change. The relative increase in the number of lines and the relative decrease in the number of lines respectively refer to the ratio of the actual increase and decrease in the number of lines to the number of code lines in the relevant file before the change is submitted [22].

Document modification history dimension: This dimension is used to quantify the modification history of related documents. Empirical research shows that the more complex the document modification history (e.g., it has been modified multiple times, modified by multiple developers, etc.), the more likely it is to have defects [21], [23], [24]. In view of this, the Just-in-Time defect prediction researchers proposed to use the number of revisions of related files before the change is submitted, and the number of developers who modify these files as a feature to quantify the revision history of the files to predict defective changes [6], [10], [17], [18], [20]. In addition, Shihab et al. proposed to use the number of defect repair changes in the historical changes of the modified change-related documents to quantify the modification history of the change-related documents [10]. Kamei et al. proposed to modify the related changes before submitting the changes The time difference between the recent change of the file and the change is used as the modification history of the feature quantified change file [6].

Developer experience dimension: This dimension is used to quantify the developer experience of code changes. Research has shown that developer experience will affect software quality [5]. In Just-in-Time defect prediction work, researchers use the number of changes submitted by developers, Quantify developer experience. In addition, Kamei et al. proposed to use the number of recent changes submitted by the developer and the number of changes that affect the relevant subsystems of the changes submitted by the developer before the change is submitted to quantify the developer experience [6]. McIntosh et al. It is proposed that before the use of change submission, the changes that affect the relevant subsystems of the changes submitted by the developer account for the proportion of all changes that affect these subsystems in the version control system, to quantify the developer's development experience of these subsystems [20].

According to previous work by Kamei [6], it is found that NF is highly correlated with ND, REXP and EXP. Therefore, we exclude ND and REXP from the model, and use NF and EXP to further find that LA and LD are highly correlated. Nagapan and Ball [32] reported that relative churn features

TABLE II
FEATURE SUMMARY TABLE

Feature dimension	Feature name	Definition	Rationale	Related Work
Diffusion	NS	Change the number of modified subsystems	The more changes that modify the subsystem, the more likely it is that defects will be introduced	Mockus, et al. [7]
	ND	Change the number of modified code directories	The more changes you modify the code directory, the more likely it is to introduce defects	Kamei, et al. [6]
	NF	Change the number of files modified	The more the number of modified files, the more likely the change is to introduce defects	Mockus, et al. [7]
	Entropy	Modify the distribution of the code in the change-related files (use Information entropy calculation)	The greater the information entropy, the more scattered the changed code is in related files, and the more code developers need to understand, the more likely it is to introduce defects	Hasssan [10]
Size	LA/LD	Added lines and deleted lines	The more lines of code increase and decrease, the more likely it is that defects will be introduced	Mockus, et al. [7]
	CA/CD	Added chunks and deleted chunks	The more code segments are added or reduced, the greater the impact on the software code, the more likely it is to introduce defects.	Shihab, et al. [10]
	LT	Lines of code of files touched by the change	The larger the file, the more likely it is that the modification of the file will introduce defects.	Mockus, et al. [7]
Purpose	FIX	Whether or not the change is a defect fix	Changes to repair defects are more complex and easier to introduce defects.	Mockus, et al. [7]
	NBR	Number of defect reports related to the change	The more related defect reports, the more code changes need to be modified, and the more likely it is to introduce defects.	Shihab, et al. [10]
Experience	EXP	The number of changes submitted by the developer	It is not easy for developers with more experience to introduce defects.	Mockus, et al. [7]
	REXP	The number of recent changes submitted by developers	Developers who frequently modify code recently are more familiar with project development and are not easy to introduce defects.	Mockus, et al. [7]
	SEXP	The number of subsystems that the developer has submitted changes that affect the change	Developers modify familiar subsystems and it is not easy to introduce defects.	Mockus, et al. [7]
	Awareness	Among the changes submitted by the developer, the changes that affect related subsystems account for the proportion of all changes that affect these sub-systems	The more modifications to the subsystem, the more familiar the developer is with the subsystem, and the less likely it is to introduce defects.	McIntosh, et al. [20]
History	NDEV	The number of developers who have modified the files related to this change	The more developers who have modified the file, the more likely the changes to modify the file will introduce defects.	Shihab, et al. [10]
	NUC	The number of changes made to the relevant documents of the change	The more times a file is modified, the more code developers need to understand when modifying the file, and the more likely it is to introduce defects when modifying the file.	Shihab, et al. [10]
	AGE	The average time difference between the most recent change and the change related to the document that has been modified	Recently submitted changes are more likely to introduce defects.	Kamei, et al. [6]

are better than absolute features in predicting defect density. Therefore, using LT to normalize LA and LD is similar to the method of nagapan and Ball. We also use NF to standardize LT and NUC, because these features have a high correlation with NF.

Because most features are biased in distribution, we use logarithmic transformation to reduce this bias. We have performed standard log transformation for each feature. Because the value of FIX is a Boolean variable, it is excluded.

Through the data set, we can find that our data is relatively unbalanced, because the number of changes that cause defects is still a small number relative to all changes. If handled incorrectly, the performance of the prediction model may decrease, leading to inaccurate prediction results [26]. In order to solve the problem of data imbalance, we undersampling the training data, that is, randomly delete most non-defect-causing change instances, making it equal to the number of defect-causing change instances. Here we are only under-sampling the training data, and will not perform special processing on the test data.

IV. Case Study Results

A. RQ1: How efficient is our prediction model?

Overview: In order to answer RQ1, we use the characteristics in the table to establish a software change risk prediction model, and then use the open source project data set to evaluate the performance of the model.

Validation technique and data used: We used the 10-fold cross-validation method on the data set [27]. First, the data set was randomly shuffled, and then the data set was divided into 10 equal parts, namely 10 folds. After that, each fold was used as a test set, and the other 10 folds passed The under-sampling method is then used as a training set to train the prediction model and verify the calculation performance indicators. A total of 10 experiments are run, and finally the performance indicators obtained each time are averaged as the evaluation result of 10-fold cross-validation.

Approach: Similar to the previous work [6], we use the random forest classification model to make predictions. In order to avoid overfitting our model, we choose a set of smallest sets as the independent variables of the model. We first manually deleted the highly correlated features, and then used the Mallows-based CP criterion to gradually delete the remaining collinear variables and variables that have no effect on the model.

In order to evaluate the prediction performance of the software defects of the model, we use different indicators to quantify the prediction results of the model. These indicators include precision rate, recall rate, F1-measure, correct rate and AUC. These performance indicators can all be calculated. The predictive model has four possibilities for predicting the results of a code change: 1. predicting a defective code change as a true positive (TP); 2. predicting a defective code change as having no defect (false positive, referred to as FP); 3. Predict a code change without defects as no defect (true negative, referred to as TN); 4. Predict a code change without defects as a defect (false negative, referred to as FN) according to The prediction model can calculate the accuracy, recall, correctness, and F1-measure of the four prediction results in the test set. Since Just-in-Time defect prediction research pays more attention to the prediction effect of defective changes, the description in this article The precision, recall, and F1-measure are all for defective code changes.

In Just-in-Time defect prediction research, AUC is also a commonly used performance indicator [6]. AUC stands for Area Under the Curve of receiver operating characteristic, which refers to the area under the receiver operating characteristic curve (ROC). The ROC curve is The TP ratio (true positive rate, TPR) is a function curve with the FP ratio (false positive rate, FPR) as a variable on all thresholds (threshold). The prediction model needs to use a threshold to perform the labeling of code changes. Judgment threshold ranges from 0 to 1. When the predictive model predicts a code change, the model will calculate the probability value of the code change including the defect. In order to get the prediction result (the code change is defective or not defective), the The model compares the probability value with the threshold: if the probability value is greater than the threshold, the model predicts the code change as defective; otherwise, it predicts the code change as no defect. In this way, the TP, FP, TN, and FN can be calculated Therefore, the calculation of precision, recall, F1-measure and correct rate all depend on the threshold of the prediction model [10], [11].

Since ROC is a function curve of TPR with FPR as a variable on all thresholds, ROC does not depend on the threshold. And AUC is the area under the ROC curve, so the AUC value does not depend on the threshold [10,11]. Lessmann et al. pointed out , AUC is robust to unbalanced data [9]. The calculation of AUC automatically takes into account the imbalances in the data. AUC has a statistical explanation [10]. In the context of instant defect prediction, AUC can be evaluated the predictive model has a higher probability of calculating the defective probability of a randomly selected defective change than a randomly selected code change without defects. In practical applications, the output of the predictive model is used to schedule the work. AUC is suitable for evaluating this kind of scheduling.

Results: We made our predictions, and the results are given in the Table III. We compare our prediction results with the benchmark method (random prediction defect). The last two columns in the Table III use our prediction model in precision Compared with the AUC indicator, the improvement (percentage) of the random prediction model. For six open source projects, our prediction model achieved an average precision rate of 36.8% and a recall rate of 68.8%, which shows that our performance is 90% higher than the random prediction model. We noticed from the table that our predictive model achieves a higher recall rate, which is a more important performance indicator in highly skewed data sets. For example, the average recall rate in the previous study by Kim [26] et al. was Menzies et al. [29] explained in the article that, in many cases, when the data has great imbalance, the prediction

TABLE III
RQ1 EXPERIMENT RESULT TABLE

	Acc	Prec	Recall	F1	AUC	Improved Prec	Improved AUC
BUZ	71.4%	59.5%	67.3%	63.0%	77.5%	64.0%	55.0%
COL	70.5%	50.8%	68.0%	58.0%	72.7%	63.7%	46.3%
JDT	69.7%	26.6%	65.2%	37.4%	73.2%	91.9%	45.4%
MOZ	75.4%	14.1%	74.6%	23.6%	81.7%	191.1%	64.4%
PLA	68.7%	26.6%	67.6%	37.7%	74.3%	86.5%	48.9%
POS	73.4%	47.0%	70.6%	56.3%	76.5%	89.0%	52.9%
Avg	71.5%	37.4%	68.9%	46.0%	76.0%	97.7%	52.2%
Med	71.0%	36.8%	67.8%	47.0%	75.4%	87.8%	50.9%

model with low precision and high recall is still very useful. This is because, as long as most "bad situations" are avoided, developers can accept the idea of checking a little more than needed files or changed files. The precision of traditional defect prediction models with file-level granularity is usually low (for example, the precision is 14% [29], [30]. As shown in the data set detail table, in open source projects, we change on average every day The total number of changes and the number of changes that caused defects are 9.4 and 1.9, respectively. Based on our prediction results, our prediction model marks an average of 3.5 changes per day as defect induction, of which 2.2 are false positive results. This result means that developers every day Only need to repeatedly check 2.2 unnecessary changes, so we believe that our predictive model is useful in practical applications.

B. RQ2: What features can be used to judge by interpretability techniques to play a significant role in the prediction?

Overview: The existing technical knowledge of Just-in-Time defect prediction predicts the possibility of defects in the change, and does not explain the predictive defect model, which helps developers to find out the type and location of defects more quickly. We hope to be able to use interpretable techniques to find the key features that cause defects.

Approach: Because the random forest is also a black box model to a certain extent, we use the LIME method to interpret the prediction results. We first float the training set and the test set, and then randomly select a single instance to interpret it. In order to be able to find the main characteristics that affect the defect prediction results, we use the SP-LIME method in LIME. We select 300 instances in each data set. The selected instances can cover various label types and cover as many cases as possible. , And not redundant (not duplicated). Because the SP-LIME method is still based on the partial LIME method, we use the two indicators mentioned by Visani et al [38]. to evaluate the stability of LIME: VSI index (variable stability index) and CSI (coefficient stability index), high The VSI value guarantees that the characteristics presented in different LIMEs are almost the same. On the contrary, a lower VSI value indicates that the LIME model

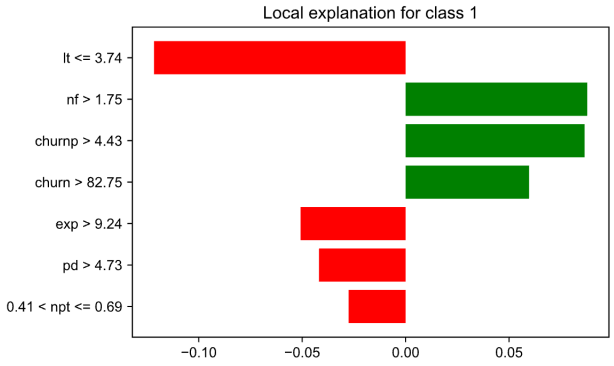

Fig. 6. Columba Analysis Chart.

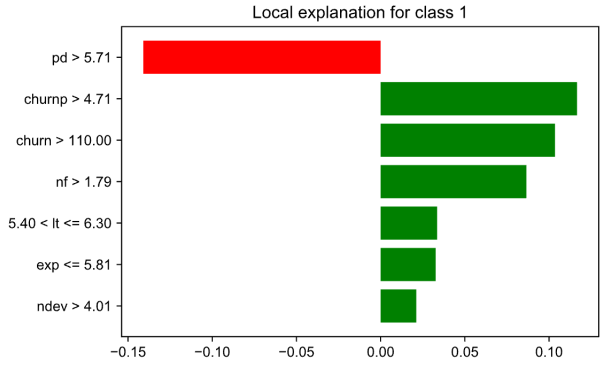

Fig. 7. Mozilla Analysis Chart.

is unreliable. For the CSI index, the higher the CSI index value, it means that the influence coefficient of each feature in the LIME model is very reliable, and the lower value will make developers evaluate this feature very cautiously. We will evaluate the importance of the influence of their characteristics and the stability of the model on six open source databases.

Results: Select the analysis graphs of the two data sets Columba and Mozilla for display. Because these two data

TABLE IV
FEATURE EXTRACTION SUMMARY TABLE

	ns	nf	entropy	lt	ndev	pd	npt	exp	sexp	churn	churnp	CSI	VSI
BUZ		-		-		+				+	+	94%	90%
COL		+		-				-		+	+	92%	84%
JDT		-				-			-	+	+	95%	88%
MOZ		+		+		-				+	+	96%	86%
PLA		+		-		+				+	+	97%	89%
POS		-		+		-				+	+	95%	86%

sets are relatively representative(Fig.6 Fig.7), Columba has the least amount of total data and has a lower class imbalance rate; Mozilla has the most data and has the highest class imbalance rate. The summary_plots of the two original data sets are shown in the figure. According to the two pictures, the following rules of software defect data distribution are summarized: (1) The more subsystems, function libraries, and files defined in a program, the more defects are likely to occur; (2) The operation of adding, deleting or modifying the code The more, the more likely it is to cause defects; (3) The more lines of code in the file, the more likely it is to cause defects; (4) The location where the defect is repaired is more likely to have other defects; (5) The more developers have contacted (6) Code changes, the shorter the interval, the more likely it is to have defects; (7) The more the code is changed last time, the greater the probability of defects; (8) Related procedures The more experience the staff has, the smaller the probability of bugs caused by changes. At the same time, the top five features that have the most important impact on the two data sets of Columba and Mozilla are NF, IT, EXP, Churn, Churnp and NF, IT, PD, Churn, Churnp according to the results shown above.

As shown in the table below, we summarize the experimental results into a table IV. We use **+ -** to indicate whether the feature predicts a software defect as a defect is positive or negative. We can see that NF (number of files), relative loss metrics (LA/LF and LT/NF), and whether changes can repair defects (PD) are the most important risk factors. At the same time, the average values of our CSI index and VSI index reached 95% and 87%, respectively, indicating that our LIME has considerable stability.

C. RQ3: After removing unimportant features, what is the performance of the defect model?

Overview: Through the research on RQ2, we have extracted the top five most important features corresponding to the six open source projects. In RQ3, we want to know what the features we extracted through interpretability technology have for the performance of the defect prediction model influences.

Approach: Based on the results of RQ2, we have calculated the top five characteristics of each data set corresponding to the influence degree. We use the random forest model in RQ1 to make predictions. Similar to RQ1, we still use the 10-fold cross-validation method. The five-bit features are trained and predicted, and compared with the results in RQ1.

TABLE V
COMPARISON TABLE OF PREDICTION EFFICIENCY FOR EXCLUDING LOW-IMPACT VALUE FEATURES

	Using 11 features	Retain the five most influential features
BUZ	71.4%	68.3%
COL	70.5%	68.9%
JDT	69.7%	64.3%
MOZ	75.4%	72.1%
PLA	68.7%	65.2%
POS	73.4%	71.7%

Results: The experimental results are shown in Table V ,When the five most important features are used to train the prediction model, the average accuracy of the model can reach 68.42%. Compared with the prediction accuracy using all the features, the accuracy is reduced by 3.1%. Explain that if we filter out the most representative features in advance, we can use 45% of the workload to achieve 96% of the original work efficiency.

V. LIMITATIONS AND THREATS TO VALIDITY

Construct validity.A lot of previous work has shown that the parameters of the classification technology have an impact on the performance of the defect model [39]–[42]. Although the value of ntree we use for the random forest is the default 100, recent studies have shown that the parameters of the random forest model will not affect our research [41], [42]. The effectiveness of this poses a threat. Recent studies have pointed out that the choice and quality of data sets may affect the conclusions of a study [42]–[44], so we chose six open source data sets, and under-sampling and related data cleaning to ensure the quality of the data sets , To reduce the impact of the quality of the data set on the experimental results.

External validity. Although we used data sets from 6 open source projects, there are, for example, some commercial projects. Therefore, our project data set may not represent the verification results of all project data sets, but our data set is more comprehensive and larger in scale than the data set of previous research work [7], [8], [26]. I believe that our research

work is useful in the field of defect prediction. The explanatory direction has a certain contribution. Of course, we still have some obvious problems. In the field of defect prediction, there are many types of features. We may not measure other features. Other features may also affect the possibility of defects. If possible, the features are more comprehensive. May further improve our forecast results.

Internal validity. In the research, we chose Random Forest as our software defect prediction model. Recent studies have shown that Random Forest may be the most suitable machine learning method for software defect prediction [45]. Using different forecasting methods, the final conclusion may be slightly biased, but the overall trend should be the same. In terms of interpretable technology, we use LIME technology. LIME is actually a locally interpretable method, but it can interpret the whole from the part. Therefore, our experimental conclusions may not be applicable to all interpretable methods. Nevertheless, other interpretability techniques can also be applied to our data set, which can be explored in future work.

VI. Conclusion

In this article, we use the interpretability model to explain and optimize the defect model. We validate our experiments through extensive research on six open source projects. Our research results show that the random forest model in RQ1 can perform defect prediction very well, reaching an accuracy of 71.52% and a recall rate of 68.88%. In RQ2, we pioneered the use of the LIME model to interpret the prediction model and results. The existing instant defect prediction technology only predicts the possibility of defects in the change, and focuses on what the predicted defect is, such as the type of defect. There is no relevant research on the defect type and location. The defect type describes the reason and characteristics of the defect. The defect location refers to the module, file, function or even code line where the defect is located. Knowing the defect type and location can help developers quickly repair the defect. We use the LIME model to evaluate the most influential features, use these features to train the prediction model, and find that we can use 45% of the previous workload to achieve 96% of the original work capacity, and we hope to be able to minimize chemical engineering in the future. Pay attention to the risk of defects, thereby reducing the cost of software defect prediction. Future research will focus on how to optimize the distribution of unbalanced data at a deeper level, and how to optimize the processing process of the integrated learning algorithm, so as to obtain faster model running time and higher prediction accuracy.

VII.

References

[1] Zubrow D. IEEE Standard Classification for Software Anomalies. IEEE Std, 2009.1–23.

[2] Newman M. Software errors cost us economy 59.5 billion annually: NIST assesses technical needs of industry to improve software-testing. 2002.

[3] Marks L, Zou Y, Hassan AE. Studying the fix-time for bugs in large open source projects. In: Proc. of the 7th Int'l Conf. on Predictive Models in Software Engineering. New York: ACM Press, 2011. No.11.

[4] LaToza TD, Venolia G, DeLine R. Maintaining mental models: A study of developer work habits. In: Proc. of the 28th Int'l Conf. on Software Engineering. New York: ACM Press, 2006. 492–501.

[5] Eyolfson J, Tan L, Lam P. Do time of day and developer experience affect commit bugginess? In: Proc. of the 8th Working Conf.on Mining Software Repositories. New York: ACM Press, 2011. 153–162.

[6] Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. on Software Engineering, 2013,39(6):757-773.

[7] Mockus A, Weiss DM. Predicting risk of software changes. Bell Labs Technical Journal, 2000,5(2):169-180.

[8] Kim S, Jr. Whitehead EJ, Zhang Y. Classifying software changes: Clean or buggy? IEEE Trans. on Software Engineering, 2008, 34(2):181-196.

[9] Lessmann S, Baesens B, Mues C, Pietsch S. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans. on Software Engineering, 2008,34(4):485-496.

[10] Shihab E, Hassan AE, Adams B, Jiang ZM. An industrial study on the risk of software changes. In: Proc. of the 20th Int'l Symp. on the Foundations of Software Engineering. New York: ACM Press, 2012. No.62.

[11] Sliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? In: Proc. of the 2nd Working Conf. on Mining Software Repositories. New York: ACM Press, 2005. 24-28.

[12] Kamei Y, Shihab E. Defect prediction: Accomplishments and future challenges. In: Proc. of the 23rd Int'l Conf. on Software Analysis, Evolution, and Reengineering. Washington: IEEE, 2016. 33-45.

[13] Kim SH, Zimmermann T, Pan K, Jr. Whitehead EJ. Automatic identification of bug-introducing changes. In: Proc. of the 21st Int'lConf. on Automated Software Engineering. Washington: IEEE, 2006. 81-90.

[14] Da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans. on Software Engineering, 2017,43(7):641-657.

[15] Neto EC, Da Costa DA, Kulesza U. The impact of refactoring changes on the SZZ algorithm: An empirical study. In: Proc. of the 25th Int'l Conf. on Software Analysis, Evolution and Reengineering. Washington: IEEE, 2018. 380-390.

[16] Herzig K, Just S, Zeller A. It's not a bug, it's a feature: How misclassification impacts bug prediction. In: Proc. of the 35th Int'l Conf. on Software Engineering. Washington: IEEE, 2013. 392-401

[17] Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N. An empirical study of Just-in-Time defect prediction using cross- project models. In: Proc. of the 11th Working Conf. on Mining Software Repositories. New York: ACM Press, 2014. 172-181.

[18] Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE. Studying Just-in-Time defect predictionusing cross- project models. Empirical Software Engineering, 2016,21(5):2072-2106.

[19] Fu W, Menzies T. Revisiting unsupervised learning for defect prediction. In: Proc. of the 25th Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2017. 72-83.

[20] McIntosh S, Kamei Y. Are fix-inducing changes a moving target? A longitudinal case study of Just-in-Time defect prediction. IEEE Trans. on Software Engineering, 2018,44(5):412-428.

[21] Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proc. of the 30th Int'l Conf. on Software Engineering. New York: ACM Press, 2008. 181-190.

[22] Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. In: Proc. of the 27th Int'l Conf. on Software Engineering. New York: ACM Press, 2005. 284-292.

[23] Matsumoto S, Kamei Y, Monden A, Matsumoto KI, Nakamura M. An analysis of developer metrics for fault prediction. In: Proc. of the 6th Int'l Conf. on Predictive Models in Software Engineering. New York: ACM Press, 2010. 18.

[24] Bird C, Nagappan N, Murphy B, Gall H, Devanbu P. Don't touch my code! Examining the effects of ownership on software quality.In: Proc. of the 19th Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2011. 4-14.

[25] Yang Y, Zhou Y, Liu J, Zhao Y, Lu H, Xu L, Xu B, Leung H. Effort-aware Just-in-Time defect prediction: Simple unsupervised models could be better than supervised models. In: Proc. of the 24th Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2016. 157-168.

[26] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto, "The Effects of Over and Under Sampling on Fault- Prone Module Detection," Proc. Int'l Symp. Empirical Software Eng. and Measurement, pp. 196- 204, 2007.
[27] B. Efron, "Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation," J. Am. Statistical Assoc., vol. 78, no. 382, pp. 316-331, 1983.
[28] S. Kim, E.J. Whitehead Jr., and Y. Zhang, "Classifying Software Changes: Clean or Buggy?" IEEE Trans. Software Eng., vol. 34, no. 2, pp. 181-196, Mar. 2008.
[29] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'"," IEEE Trans. Software Eng., vol. 33, no. 9, pp. 637-640, Sept. 2007.
[30] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto, "The Effects of Over and Under Sampling on Fault- Prone Module Detection," Proc. Int'l Symp. Empirical Software Eng. and Measurement, pp. 196-204, 2007.
[31] D'Ambros M, Lanza M, Robbes R. An extensive comparison of bug prediction approaches. In: Proc. of the 7th Working Conf. on Mining Software Repositories. Washington: IEEE, 2010. 31-41.
[32] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," Proc. Int'l Conf. Software Eng., pp. 284-292, 2005.
[33] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "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 (KDD '16). Association for Computing Machinery, New York, NY, USA, 1135–1144.
[34] Hassan AE. Predicting faults using the complexity of code changes. In: Proc. of the 31st Int'l Conf. on Software Engineering. Washington: IEEE, 2009. 78-88.
[35] Thanh Tung Khuat, Bogdan Gabrys,A comparative study of general fuzzy min-max neural networks for pattern classification problems,Neurocomputing,Volume 386,2020,Pages 110-125,ISSN 0925-2312
[36] L. Pascarella and A. Bacchelli, "Classifying Code Comments in Java Open-Source Software Systems," 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017, pp. 227-237, doi: 10.1109/MSR.2017.63.
[37] Bejjanki KK, Gyani J, Gugulothu N. Class Imbalance Reduction (CIR): A Novel Approach to Software Defect Prediction in the Presence of Class Imbalance. Symmetry. 2020; 12(3):407. https://doi.org/10.3390/sym12030407
[38] Visani, G., Bagli, E., Chesani, F., Poluzzi, A., Capuzzo, D. (2020). Statistical stability indices for LIME: obtaining reliable explanations for Machine Learning models. arXiv preprint arXiv:2001.11757.
[39] T. Mende, "Replication of Defect Prediction Studies: Problems, Pitfalls and Recommendations," in Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE), 2010, pp. 1–10.
[40] T. Mende and R. Koschke, "Revisiting the Evaluation of Defect Prediction Models," Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE), pp. 7–16, 2009.
[41] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Mat- sumoto, "Automated Parameter Optimization of Classification Techniques for Defect Prediction Models," in Proceedings of the International Conference on Software Engineering (ICSE), 2016, pp. 321–332.
[42] "The Impact of Automated Parameter Optimization on De- fect Prediction Models," Transactions on Software Engineering (TSE), p. In Press, 2018.
[43] J. Keung, E. Kocaguneli, and T. Menzies, "Finding Conclusion Stability for Selecting the Best Effort Predictor in Software Effort Estimation," Automated Software Engineering, vol. 20, no. 4, pp. 543– 567, 2013.
[44] S. Yathish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamtha- vorn, "Mining Software Defects: Should We Consider Affected Re- leases?" in In Proceedings of the International Conference on Software Engineering (ICSE), 2019, p. To Appear.
[45] Osman, Haidar, Mohammad Ghafari, Oscar Nierstrasz, and Mircea Lungu. "An extensive analysis of efficient bug prediction configurations." In Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 107-116. ACM, 2017.