The Journal of Systems & Software 188 (2022) 111245

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Interpretability application of the Just-in-Time software defect prediction model☆

Wei Zheng [a], Tianren Shen [a,*], Xiang Chen [b], Peiran Deng [a]

[a] Northwestern Polytechnical University, School of Software, Xi'an, China
[b] Nan Tong University, School of Information Science and Technology, Nan Tong, China

ARTICLE INFO

Article history:
Received 5 October 2021
Received in revised form 14 December 2021
Accepted 23 January 2022
Available online 3 February 2022

Keywords:
Software metrics
Just-in-Time defect prediction
Interpretability
Interpretation method

ABSTRACT

Software defect prediction is one of the most active fields in software engineering. Recently, some experts have proposed the Just-in-time Defect Prediction Technology. Just-in-time Defect prediction technology has become a hot topic in defect prediction due to its directness and fine granularity. This technique can predict whether a software defect exists in every code change submitted by a developer. In addition, the method has the advantages of high speed and easy tracking. However, the biggest challenge is that the prediction accuracy of Just-in-Time software is affected by the data set category imbalance. In most cases, 20% of defects in software engineering may be in 80% of modules, and code changes that do not cause defects account for a large proportion. Therefore, there is an imbalance in the data set, that is, the imbalance between a few classes and a majority of classes, which will affect the classification prediction effect of the model. Furthermore, because most features do not result in code changes that cause defects, it is not easy to achieve the desired results in practice even though the model is highly predictive. In addition, the features of the data set contain many irrelevant features and redundant features, which are invalid data, which will increase the complexity of the prediction model and reduce the prediction efficiency. To improve the prediction efficiency of Just-in-Time defect prediction technology. We trained a just-in-time defect prediction model using six open source projects from different fields based on random forest classification. LIME Interpretability technique is used to explain the model to a certain extent. By using explicable methods to extract meaningful, relevant features, the experiment can only need 45% of the original work to explain the prediction results of the prediction model and identify critical features through explicable techniques, and only need 96% of the original work to achieve this goal, under the premise of ensuring specific prediction effects. Therefore, the application of interpretable techniques can significantly reduce the workload of developers and improve work efficiency.

© 2022 Elsevier Inc. All rights reserved.

1. Introduction

Software defect prediction technology has been one of the most dynamic contents in software engineering since 1970s. The software has become an essential factor affecting the national economy, military, political, and even social life. Highly reliable and complex software systems depend heavily on the reliability of the software they employ. Software defects are the potential source of related system errors, failures, crashes, and even crashes (Wong et al., 2017). Therefore, defect repair becomes a critical activity in software maintenance, but it also consumes time and resources (Marks et al., 2011).

Defects have an essential impact on software quality and even software economics. For example, the National Institute of Standards and Technology (NIST) estimates that software defects cost as much as the United States $60 billion a year. Thus, identifying and fixing these software flaws could save the United States $22 billion (Newman, 2002).

Statistics show that fixing defects account for 50% to 75% the total cost of software development (LaToza et al., 2006). At the same time, the complexity and difference of defect distribution problems and the deficiency of existing defect prediction technology in solving practical problems are also explained. At present, the existing software defect prediction methods can be divided into static defect prediction methods and dynamic defect prediction methods (Zubrow, 2009). The static defect prediction method is based on the measurement data related to defects to predict the defect tendency, defect density, or defect number of program modules. The dynamic defect prediction method predicts the

☆ Editor: W. Eric Wong.

* Corresponding author.
E-mail addresses: wzheng@nwpu.edu.cn (W. Zheng), trshen@mail.nwpu.edu.cn (T. Shen), xchencs@ntu.edu.cn (X. Chen), dengpeiran@mail.nwpu.edu.cn (P. Deng).

https://doi.org/10.1016/j.jss.2022.111245
0164-1212/© 2022 Elsevier Inc. All rights reserved.

distribution of system defects over time based on the time of defects or failures to discover the distribution law of software defects over time in its life cycle or some stages. This is because, as people develop software, increases the workload, increase or decrease of humans, such as demand, coding work will introduce more defects. Reviews, test work, can be reducing the number of defects, but in general, assumes that process, factors such as the ability of technology are stable, the defects and the scale of software is a proportional relationship. 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 (Eyolfson et al., 2011).

In order to solve the above challenges, researchers in the field of software engineering put forward the Just-in-Time defect prediction technology. Just-in-Time defect prediction technology refers to the technology to predict defects in every code change submitted by developers. In instant defect prediction, the predicted software entity is a code change. The immediacy of Just-in-Time defect prediction technology is reflected in the fact that it can perform defect analysis on a code change after a developer submits it and predict the likelihood that the code change will be defective. Thus, this technology can effectively cope with the challenges faced by traditional defect prediction technology, mainly reflected in the following three aspects:

(1) **Fine-grained.** Code change level prediction focuses more on fine-grained software entities than module or file level defect prediction. As a result, developers can spend less time and effort reviewing code changes predicted to be defective.

(2) **Just-in-Time.** Just-in-Time defect prediction technology can be used to predict defects when a code change is submitted. At this point, developers still have a deep memory of the changed code and do not need to spend time re-understanding their submitted code changes, which helps to fix defects in a more timely manner.

(3) **Easy to trace.** The developer's information is saved in the code changes the developer submits. As a result, the project manager can more easily find the developer who introduced the defect, which facilitates timely analysis of the cause of the defect and helps complete defect allocation (Kamei et al., 2012).

Although the machine learning model has outstanding performance in many fields, such as face recognition, image classification, natural language processing etc, this performance is more dependent on a highly nonlinear model and parameter tuning technology. There is no way to fathom what machine learning models learn from the data and how they make their final decisions. This "end to end" decision-making model results in a machine learning model that is exceptionally unexplanatory. From a human perspective, the decision-making process of the model is incomprehensible. That is, the model is unexplainable. The inexplicability of the machine learning model has many potential dangers, and it is not easy to establish trust between humans and machines. Since an unexplainable model cannot provide more reliable information, its actual deployment in many fields will be minimal. For example, the lack of an 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 developing and applying machine learning in authentic tasks.

Machine learning model interpretability has a wide range of potential applications, including model validation, model diagnosis, auxiliary analysis and knowledge discovery. Interpretability means that we have enough understandable information to solve a problem. Specifically, in artificial intelligence, explainable depth models can provide the decision basis for each prediction result. For example, the example shown in Fig. 1 describes the process of

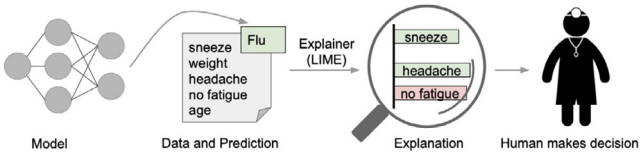

Fig. 1. Explaining individual predictions. A model predicts that a patient has the flu, and LIME highlights which symptoms in the patient's history led to the prediction. Sneeze and headache are sneeze portrayed as contributing to the "flu" prediction, while "no fatigue" is evidence against it. With these, a doctor can make an informed decisiohneaadabcoheut the model's prediction (Ribeiro et al., 2016).

a model used for assisted seeing a doctor prove its credibility to doctors: "the model not only has to give its prediction result (flu), but also provides the result of 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".

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

- RQ1: How efficient is our prediction model?
  In previous studies, Mockus and Weiss only used a large-scale telecommunications system project to evaluate their prediction model (Lessmann et al., 2008), which may result in unreliable results. To better evaluate our prediction model and increase the experimental persuasiveness, we used data sets from six open source projects published by Lessmann et al. (2008). Furthermore, to better identify defects caused by code changes, a new defect prediction model was established based on Kamei's previous work. In the new prediction model, the prediction accuracy of software defects caused by code changes is 68%, and the recall rate is 64%.
- RQ2: What features can be used to judge by interpretability techniques to play a significant role in the prediction?
  Up to now, Just-in-Time defect prediction technology only predicts the possibility of defects in the change. What are the types of defects to be predicted? Where are the software defects located? At present, there is no relevant research on these issues. The defect type describes the cause and characteristics of the defect, and the defect location refers to the module, file, function and even the line of code where the defect is located. Having information about the type of defect and the location of the defect has excellent potential to help developers fix the defect quickly. Although many researchers have proposed some defect classification and location techniques for immediate defects, there is no related research on predicting defect classification and location. This experiment used interpretable models to calculate the number of files (NF), relative loss measures (LA/LF and LT/NF), and the time interval between the last change and the current change. The experiment found that whether changes can repair defects (PD) was the most crucial feature.
- RQ3: After removing unimportant features, what is the performance of the defect model?

At present, Just-in-Time software defect prediction technology still has low efficiency caused by heavy workload when facing substantial software projects. We hope that the most influential features can be screened out through the explanatory model through preliminary screening so that the model can have high predictive power in as little time as possible. Our results show that we can achieve 96% of the predictive model's original capacity at 45% of the original effort.

2. Background and related work

2.1. Just-in-Time Defect prediction technology

During software development and maintenance, code modification is required to remove inherent software defects, improve existing functions, reconstruct existing code, or improve operating performance. However, some code changes may accidentally introduce new defects after completing the modification task (Wong et al., 2010). Therefore, developers want a defect prediction model that can quickly and accurately determine whether committed code change is a Buggy code change (that is, a code change produces a defect) or a clean code change (that is, a code change does not produce a defect). Suppose the code change is predicted to be a Buggy code change. In that case, the developer can quickly locate and remove the defect by designing more unit test cases, conducting code reviews, or looking at similar code changes in the project. At the same time, they are still familiar with the code change. This section refers to this problem as defect prediction based on code modifications, and Kamei et al. refer to this type of software quality assurance activity as Just-in-Time quality assurance (Kamei et al., 2012).

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 granularity and Just-in-Time traceability. Khuat and Le (2020). Empirically evaluated the importance of sampling various classifier sets on unbalanced data in software defect prediction. Combined with sampling technology and an integrated learning model, it played a positive role in data prediction with class imbalance problems. Liu et al. (2018). Used the information gain feature selection algorithm to optimize the original data set and trained and tested the optimized data set with a Bayesian polynomial algorithm. Pascarella et al. (2019). Proposed a novel fine-grained model to predict the defective files contained in the submission and to what extent the model reduces the amount of work required to judge defects based on classification performance (Shihab et al., 2012).

Fig. 2 shows the general process of Just-in-Time 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 Bugzilla 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. Then, when new code changes are submitted, the model will predict the possibility of defects.

2.2. Defect prediction method based on machine learning

At present, most research work is based on machine learning to build a defect prediction model. Among them, common model prediction targets can be roughly divided into two categories: one is to predict the number or density of defects contained in program modules, and the other is to predict the defect tendency of program modules. The selection of defect prediction targets is generally related to the granularity setting of program modules (Qu et al., 2020). Suppose program modules are set to fine granularity (such as class level or file level). In that case, the defect tendency of prediction modules will be the target, and classification methods, including Logistic, are often adopted Regression, Naive Bayes, Decision Tree, etc (Li et al., 2019). Performance evaluation indexes for this kind of method include precision, recall, F-measure or AUC value, etc (Wang et al., 2020).

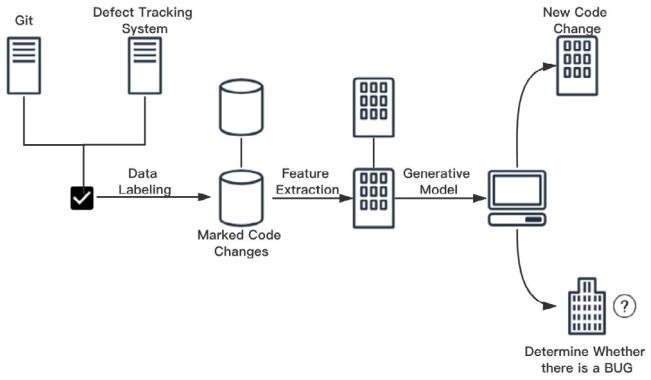

Fig. 2. General framework of Just-in-Time defect prediction.

On the other hand, if set to coarse-grained (such as package level or subsystem level), the goal is to predict the number or density of defects within the module, and regression analysis is often used. Specifically, the metric element is set as the independent variable, and the number of defects or defect density within the module is set as the dependent variable. Regression analysis attempts to build an effective model, which can predict the dependent variable's value according to the independent variable's value. However, there are many factors causing software defects, and any single or simplified prediction algorithm after regression will inevitably idealize the environment in which defects occur, resulting in inaccurate defect estimation. In addition, with the increase of people develop software workload, increase or decrease the defects caused by man-made reasons, such as demand, such as coding work to introduce more defects, and reviews, test work can be reducing the number of defects, but in general, assumes that process, factors such as the ability of technology is stable, the defects and the scale of software is a proportional relationship.

2.3. LIME

Since features in the machine learning model undergo complex changes, and different features may influence each other, it is impossible to establish the relationship between a feature and the output directly. However, to measure the contribution of a feature to the output, the eigenvalue can be changed, and then the importance of the feature can be judged by the change of the output result. · LIME (Local Interpretable Modelagnostic Explanations) is a method for explaining black-box models, i.e. models whose inner logic is hidden and not clearly understandable. The core idea of this method is to learn an interpretable model near the predicted results to achieve local interpretability of the predicted results of the model. LIME method adds disturbance to the input sample to judge the influence of different features on the predicted results according to the change of model output to realize the explainable decision-making process of the black-box model. Then weights are assigned according to the distance between the disturbing data points and the original data, and an interpretable model is learned based on the disturbed samples. Since the decision boundary of the deep learning model is nonlinear, the LIME method interprets sample classification results by learning a local linear model. To ensure that the model's predictions will change, the input sample must have human-understandable perturbations, such as blocking parts of the input image. LIME is a model-independent, interpretable method because it only makes small perturbations around the input values and does not go deep into the model. And LIME method has achieved an excellent explanatory effect in text and image classification, which has

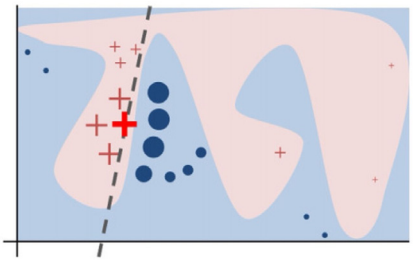

Fig. 3. LIME's modus operandi (Ribeiro et al., 2016).

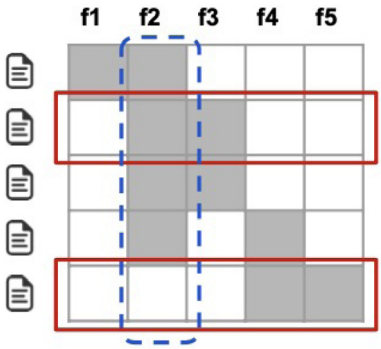

Fig. 4. Toy example W. Rows represent in-Covered Features stances (documents) and columns represent features (words). Feature f2 (dotted blue) has the highest importance. Rows 2 and 5 (in red) would be selected by the pick procedure, covering all but feature f1 (Ribeiro et al., 2016). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

dramatically improved human trust in artificial intelligence. Each part of the name reflects our intention to explain.

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 optimization 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* neighbourhood (Fig. 3).

LIME generates an interpretation of a single prediction by adding random perturbations to the input or selecting certain input features. These interpretable methods are popular because of their ease of operation.

2.4. SP-LIME

The above method is used to evaluate the reliability of the prediction for a single instance or multiple instances if the entire model needs to be evaluated. As a result, we need more samples to help us observe the behaviour of the model, and the SP (Submodular Pick) method is used to select the appropriate instance from a large amount of data. However, it is a high bar for users to select a suitable sample. Therefore, the SP-LIME algorithm is proposed in this paper to search for suitable samples automatically. In Fig. 4, each row represents the interpretation of the sample, and each column represents a feature, namely the weight of the interpretation. Examples of algorithm selection are mainly considered from two aspects. One is the diversity of features; that is, the selected instances should have rich features; The second is the importance of features. The selected features should be included in the decision-making process of the model. Quite right. As in the example above, if the algorithm is sufficiently diverse, only one of the second and third examples will be chosen because their explanations are very similar and do not provide much information to explain the behaviour of the model. In the field of software defect Just-in-Time processing, there is little research on model interpretability. In this paper, through LIME analysis, heuristic feature combination and data processing are carried out, with the ultimate goal of improving the accuracy of the Just-in-Time defect prediction model. Finally, LIME is used to analyse the relationship between the features in the model and the final prediction results to explain the complex Just-in-Time defect prediction model. In the field of immediate software defects, there is less research on model interpretability.

3. Case study design

In this part, we mainly answer the preliminary preparation of the three questions we are studying. We introduce the information about the data set used in the experiment and the pre-processing of the data set.

3.1. Research data

In previous experiments on defect prediction, Mockus and Weiss (2000) and Kim et al. (2008), experiments only examined the risk of code changes in commercial or open-source projects. Our experiment used six different projects, all of which are very well-known 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 have only looked at the risk of change in open-source projects (Kamei et al., 2007) or the risk of change in commercial projects. To increase the generality of our results and produce more concrete results, we used six large, well-known open-source projects (such as Bugzilla, Columba, Mozilla, EclipseJDT, EclipsePlatform, and PostgreSQL). The project used is written in Java. We extracted the information from the project's CVS repository for the case study and combined it with the error report. We collected data from Bugzilla and Mozilla projects using data provided by the MSR2007 MiningChallenge. Eclipse JDT and platform project data from MSR2008 Mining Challenge. For Columba and PostgreSQL, we mirrored the official CVS library.

In Table 1 , 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 code change may cause one or more software defects, the exact number of software defects caused in our experiments 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, the number of files modified each day, and the number of changes made each day. This table also shows the maximum and the average number of developers who made changes to a single

Table 1
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	4620 (36%)	389.8	37.5	2.3	1.5	37.0	8.4
Columba	11/2002–07/2006	4455 (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

file. For example, X file is modified by four developers of A, B, C, D, Y file is modified by B, C developers. The maximum number of development files modified for a single file is 4, and the average is 3.

3.2. Identify defects and introduce changes

To identify defects caused by code changes, we use the SZZ algorithm. To better understand, we use a specific software flaw in Apache project ActiveMQ (AMQ-1381) as an example to detail the four steps of the SZZ algorithm framework.

(1) Identify defect repair changes. Scan all historical data stored in the Git system, that is, all code changes, and identify code changes that contain defect ids in the logs. These code changes are identified as defect fix changes. For example, as shown in Fig. 5, the code change ID for identifying and fixing AMQ-1381 is 645599.

(2) Identify defect code that needs to be fixed. The diff algorithm implemented by the Git system is used to identify code changes and code lines modified to fix defects. These modified lines of code are identified as defective code. As shown in Fig. 5, the code is Change #645599. The modified code contains a function declaration where the Command parameter type is incorrect, and the correct type is Object instead of Command

(3) Identify defects that may introduce change. Use comment commands to trace the commit history of code changes in Git control systems. The first change to the defect code is identified as a possible defect introduction change. For example, as shown in Fig. 5, code change #447068 introduces the faulty 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 defects change should be introduced before the defect is reported (Śliwerski et al., 2005). Therefore, code changes submitted later than the defect report time will be treated as noise in the implementation of SZZ. For example, code change #447608 shown in the figure is submitted at the time of the defect. Before the report was created, 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 approximate algorithm (ASZZ) to identify whether a change is prone to defects because there is no defect identifier quoted in the changelog. In this case, we cannot verify whether the change we determined to be a defect repair is. 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.

3.3. Data processing

Target dimension: This dimension is used to describe the target for committing code changes. Goals for developers to submit changes include fixing defects, implementing new features, refactoring, adding documentation, and so on Herzig et al. (2013). Studies show that changes that modify defects are more critical than other changes and are more likely to introduce defects (Śliwerski et al., 2005). Therefore, in a large number of Just-in-Time defect prediction work, researchers use whether changes repair defects as features (Lessmann et al., 2008; Shihab et al., 2012; Kamei et al., 2016; Fu and Menzies, 2017; McIntosh and Kamei, 2017). In addition, Shihab et al. suggested quantifying code change goals using the number of change-related defect reports (Shihab et al., 2012) (see Table 2). **Code distribution dimension:** This dimension is used to mark the distribution of the modified code in the relevant files. Studies have shown that changes that modify code distributed across multiple files require developers to understand more code, so this more diffuse code change can introduce defects (D'Ambros et al., 2010). Kamei et al. proposed quantifying the change code distribution by using the characteristics of the number of files modified by the change, the number of folders, and the number of subsystems (Fu and Menzies, 2017). Hassan proposed the use of Entropy of change code distribution to predict defects and verify this feature's validity (Hassan, 2009). Therefore, Kamei et al. proposed to use this feature to predict defective changes (Fu and Menzies, 2017).

Scale dimension: This dimension is used to represent the size of the modified code. Moser et al. observed that the larger the size of the code changed, the more likely it was to introduce defects (Moser et al., 2008). In the current work, researchers have used changes to increase or decrease lines of code to quantify the scale of change. In addition, the researchers used other granularity to quantify the size of the code changes. Shihabet al. proposed changes to increase and decrease the number of code segments (chunk) to quantify the change scale (Shihab et al., 2012). Kamei et al. proposed quantifying the change scale using code lines in the change-related files before change submission (Kamei et al., 2012). Meanwhile, Kamei et al. found a high correlation between increasing code lines and decreasing codes of code in a change. In order to avoid the influence of such correlation characteristics on the prediction model, Kamei et al. proposed to use the relative increase of rows and the relative decrease of rows to quantify the change scale. The relative increase lines and the relative decrease lines refer to the ratio of the actual increase or decrease lines of the change to the number of lines of code in the relevant file before the change is submitted (Nagappan and Ball, 2005).

Document modification history dimension: This dimension is used to quantify the change history of the change-related files. Previous empirical studies have shown that the more complex the document modification history (e.g. the document has been modified multiple times, multiple developers have modified the document), the more likely defects are to exist in these cases (Moser et al., 2008; Nagappan and Ball, 2005; Bird et al., 2011). Based on the above research work, researchers of the Just-in-Time Defect Prediction proposed to use the number of changes related to the change files before the change was submitted, and the number of developers who modified these files as characteristics to predict the defective changes with the quantified

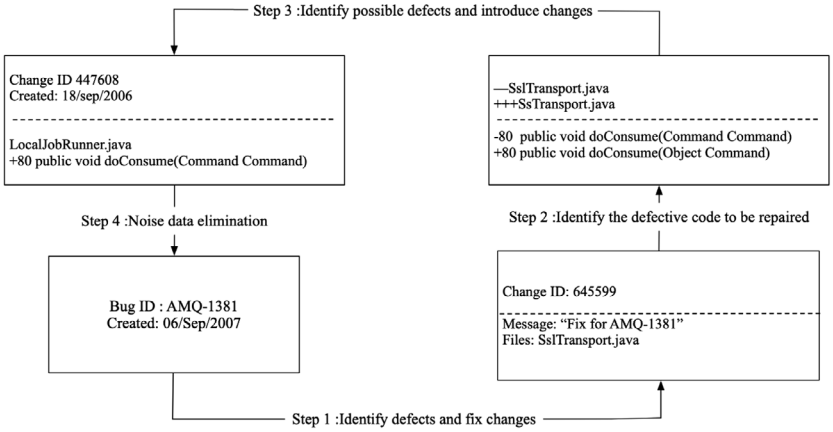

Fig. 5. General framework of the SZZ algorithm.

Table 2
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 and Weiss (2000)
	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. (2012)
	NF	Change the number of files modified	The more the number of modified files, the more likely the change is to introduce defects	Mockus and Weiss (2000)
	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	Shihab et al. (2012)
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 and Weiss (2000)
	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. (2012)
	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 and Weiss (2000)
Purpose	FIX	Whether or not the change is a defect fix	Changes to repair defects are more complex and easier to introduce defects.	Mockus and Weiss (2000)
	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. (2012)
Experience	EXP	The number of changes submitted by the developer	It is not easy for developers with more experience to introduce defects.	Mockus and Weiss (2000)
	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 and Weiss (2000)
	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 and Weiss (2000)
	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 and Kamei (2017)
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. (2012)
	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. (2012)
	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. (2012)

document change history (Kamei et al., 2012; Shihab et al., 2012; Fukushima et al., 2014; Kamei et al., 2016; McIntosh and Kamei, 2017). In addition to these, Shihab et al. propose to quantify the change history of change-related documents by using the number of defect repair changes in changes that modify the history of change-related documents (Shihab et al., 2012). Kamei et al. to quantify the change history of the change file based on the characteristics of the change before the change was submitted and the time difference between the most recent change and the change in the relevant file (Kamei et al., 2012).

Developer experience dimension: This dimension is used to quantify how much experience developers have with code changes. The research shows that the software quality is higher for developers with more experience and lower for developers with fewer experience (Eyolfson et al., 2011). In the Just-in-Time Defect Prediction work, the researchers quantified the developer experience using the number of changes the changing developer had committed. In addition, Kamei et al. proposed to quantify the developer's experience using the number of changes the developer has recently committed and the number of changes before the change was committed, and the number of changes the developer has committed that affect the subsystem associated with that change (Kamei et al., 2012). McIntosh and Kama proposed to quantify the depth of developer experience in developing these subsystems by using the proportion of all changes in the version control system that affect the subsystems related to the change that the developer has committed before the change is committed (McIntosh and Kamei, 2017).

Based on Kamei's previous work, we found that NF is highly correlated with ND, REXP, and EXP (Kamei et al., 2012). Therefore, we excluded ND and REXP from the model. Using NF and DEXP, Nagapan and Ball found that the relative loss rate feature was better than the absolute feature in predicting defect density. So we use LT to normalize LA and LD, similar tonagapan and Ball's approach. We also use NF to standardize LT and NUC, as these characteristics highly correlate with NF (Nagappan and Ball, 2005).

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

We can find that our data is relatively unbalanced through the data set because the number of changes that cause defects is still a tiny number relative to all changes. If mishandled, the prediction model's performance may decrease, leading to inaccurate prediction results (Kamei et al., 2007). 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.

4. Case study results

4.1. RQ1: How efficient is our prediction model?

Overview: To answer RQ1, we use the feature criteria selected in the table to build a software change risk prediction model based on RandomForest. In order to accurately evaluate the performance of the prediction model, we used an open-source project data set for validation.

Validation technique and data used: Before the experiment, we used the 10-Fold cross-validation method for the preliminary processing of the data set (Efron, 1983). Firstly, the data set is randomly selected, and then the data set is divided into 10 equal parts, namely ten times. Then an undersampling method was used to balance the positive and negative samples (Liu et al., 2008). Finally, the prediction model is trained to verify the calculation performance index. We need to perform a total of 10 experiments, and finally average the results of each performance indicators the evaluation result of 10-Fold cross-validation, because a large number of experiments using a large number of data sets and using different learning techniques show that 10 Fold is the appropriate choice for obtaining the best error estimate.

Table 3
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%

Approach: Similar to the previous work (Kamei et al., 2012), we use a Random Forest classification model for prediction. In order to avoid model overfitting, we choose the set of the minimum set as the independent variable of the model. We first manually deleted the features with high similarity and then used the Mallows-based CP criterion to gradually delete the remaining collinear variables and variables that do not influence the model.

In order to evaluate the predictive performance of software defect models, we use different professional indicators to quantify the predictive results of the models. These indicators include accuracy, recall, F1-measure, accuracy and AUC. These performance indicators are widely used to evaluate the performance of various machine learning prediction models. The prediction model has four possibilities for predicting the outcome of code changes: 1. Code changes (true positive, called TP) correctly classified as defect number; 2. Code incorrectly classified as defective is changed to non-defective (false positive, called FP); 3. Code changes correctly classified as defect-free are defect-free (true negative, called TN); 4. Code changes misclassified as not defective (false negatives, called FN). 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 is that the precision, recall, and F1-measure are all for defective code changes.

AUC is also a commonly used model performance indicator in Just-in-Time defect prediction research (Kamei et al., 2012). Area Under Curve (AUC) is defined as the Area enclosed with the coordinate axis Under the ROC Curve. The value of this Area is not greater than 1, and ROC does not depend on the threshold. Therefore, AUC values do not depend on thresholds (Shihab et al., 2012; Śliwerski et al., 2005). The ROC curve is the corresponding point in the ROC space calculated according to the classification results of the model. ROC curve is formed by connecting these points, and the abscissa is False Positive Rate (FPR), and the ordinate is True Positive Rate (TPR). The closer the ROC curve is to the upper left corner, the better the classifier's performance. The ROC curve is usually above the line y=x, so AUC ranges from 0.5 to 1. Lessmann et al. noted that AUC is robust to unbalanced data (Lessmann et al., 2008). AUC calculations automatically take account of imbalances in the data. The closer AUC is to 1, the more accurate the prediction model is. When it is 0.5, the authenticity of the prediction model is the lowest, indicating that the model has no application value.

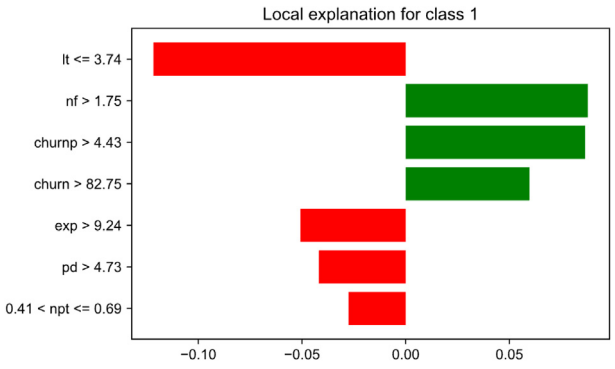

Fig. 6. Columba analysis chart.

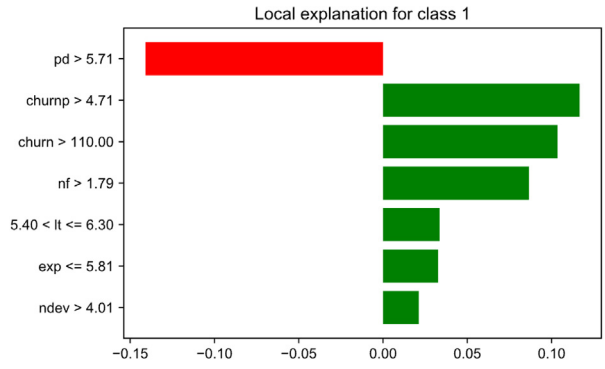

Fig. 7. Mozilla analysis chart.

Results: We used the random forest model we trained for defect prediction, and the results are shown in Table 3. We compared our prediction results with the benchmark method (Random Defect Prediction). The last two columns of Table 3 are the accuracy of our prediction model and the percentage improvement in the effectiveness of the AUC indicator compared to the benchmark method. Experimental results show that in 6 open source projects, the prediction model trained by us achieves 36.8% average accuracy and 68.8% recall rate, which is 90% better than the random prediction model. We note from the table that our prediction model achieves a higher recall rate, which is a significant performance indicator is highly skewed data sets. For example, the average recall rate in the previous study of Kamei et al. (2007). Menzies et al. (2007). The article explains that in many cases, low accuracy and high recall prediction models are still instrumental when there is a significant imbalance in data. This is because, as long as most "bad cases" are avoided, developers can accept the idea of checking some unnecessary files or changed files. The accuracy of the traditional file-level granularity defect prediction model is usually low (for example, the accuracy is 14% Kamei et al., 2007; Menzies et al., 2007).

In open-source projects, we change on average every day. The total number and number of changes resulting in defects were 9.4 and 1.9, respectively. Based on our prediction results, our prediction model labelled an average of 3.5 changes per day as defect induction, of which 2.2 were false-positive results. This result means that developers only need to check for unnecessary changes in 2.2 daily, so we believe our predictive model is helpful in practice.

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

Overview: Just-in-Time defect prediction technology can relatively accurately predict the possibility of defects in code changes, but it is still a black box model (Ljung, 2001), which is not conducive for developers to find the types of defects and possible locations of defects as soon as possible. In this experiment, we hope to use explicable techniques to find the relevant critical features that lead to defects, which will help developers find the type and location of defects more quickly.

Approach: Since the random forest model is also a black-box model to some extent, we choose to use the LIME interpretability method to explain the prediction results of the model. We first train the data set and then randomly select an example for interpretation to judge whether the prediction model is reasonable or not. In order to find out the main characteristics that affect the results of defect prediction, we continue to use the SP-LIME method derived from the LIME method. We select 300 instances

Table 4
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%

in each data set, covering a wide variety of tag types and covering as many cases as possible.

None of the selected instances will be repeated. Since the SP-LIME method is still based on the LIME method, we use the two indicators proposed by Visani et al. (2020), used to evaluate the stability of LIME: VSI (Variable Stability Index) and CSI (Coefficient stability Index). VSI measures the concordance of the variables retrieved, whereas CSI tests the similarity among coefficients for the same variable, in repeated LIME calls. High VSI ensures that different instances display nearly the same features. On the contrary, the lower the VSI value is, the less reliable the model is, and the less reliable the LIME explanation is. For the CSI index, the higher the CSI index value is, the more reliable the influence coefficient of each feature in the LIME model is. Conversely, the lower the CSI value, the more cautious developers will be in evaluating this feature. In six open source databases, we will assess the importance of features in the data set to influence the predicted results and the stability of the model.

Results: Select the analysis graphs of the two data sets, Columbia and Mozilla, for display. Because these two data sets are relatively representative (Figs. 6 7), Columba has the minor total data and a lower class imbalance rate; Mozilla has the most data and 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, and 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 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 critical impact on the two data sets of Columba and Mozilla are NF, IT, EXP, Churn, Churnp and NF, IT, PD, Churn, according to the results shown above.

Table 5
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%

As shown in the table below, we summarize the experimental results in Table 4. 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 weather changes can repair defects (PD) are the most critical 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.

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

Overview: Through the RQ2 study, we extracted the five most essential characteristics corresponding to the six open-source projects in the experimental data set. In RQ3, we want to know what impact the prediction of features extracted by interpretable techniques has on the prediction results and performance of the prediction model.

Approach: The experimental results are shown in Table 5. When the five most essential features are used to train the prediction model, the average accuracy of the model can reach 68.42%. However, 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.

Results: The experimental results are shown in Table 5. The results predicted by the random forest prediction model can be explained by the LIME model, and the five most important features of each project can be screened out. 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.

5. Limitations and threats to validity

Construct validity. A large number of previous work shows that the parameters of Random Forest classification technology impact the performance of defect models (Mitchell, 2011; Mende, 2010; Mende and Koschke, 2009; Tantithamthavorn et al., 2016, 2018). Although the value of n trees we used for the random forest prediction model is the default value of 100, recent studies show that the parameters of the random forest model do not affect our research results (Tantithamthavorn et al., 2016, 2018). However, the effectiveness of this approach poses a threat. Recent studies have pointed out that the selection and quality of data sets may affect the experimental conclusion of the study (Tantithamthavorn et al., 2018; Keung et al., 2013; Yatish et al., 2019). So we selected six open-source data sets and used the undersampling method and detailed cleaning of relevant data to ensure the quality of the data set and reduce the impact of quality on the experimental results of the data set.

External validity. In mining software history archives, noise may be generated in the process of type marking and software measurement of program modules, which will affect the construction of the defect prediction model and have a severe impact on the validity of existing empirical research conclusions. The problem of noise in defect data sets is unavoidable. Kim et al. (2011) manually injected false-positive and false-negative noise randomly and analysed the noise anti-interference ability to exist defects prediction methods. In their empirical study, they found that: (1) when there are enough defect modules in the data set, adding noise does not significantly reduce the prediction performance of existing methods; (2) The anti-interference ability of the existing methods against false harmful noise is more vital; (3) When the total proportion of false-positive and false-negative noise in the data set is minor than 20%–35%, the performance of the existing methods will not be significantly decreased. In addition, they propose a method based on cluster analysis, which can effectively detect and remove noise. In our experiment, we do not think noise has a significant effect on our results. In future work, we will try to reduce the noise as much as possible.

Internal validity. In the research, we choose random forests for our software defect prediction model. There are many methods for classifying prediction models, but recent studies indicate that random forest may be the most suitable machine learning method for software defect prediction (Osman et al., 2017). On the other hand, Catal and Diri (2009), based on NASA data set and taking AUC value as evaluation index, deeply analysed the influence of data set size, measure element and feature subset selection method on the performance of defect prediction model. The results show that the random forest method performs best on large-scale data sets, while the naive Bayes method performs best on small-scale data sets. In terms of explaining technology, we use LIME technology. LIME is a partially explicable approach, but the SP-LIME approach can explain the whole in parts. Admittedly, our experimental results may not apply to all interpretable methods. Nevertheless, the good news is that other interpretable techniques can also be applied to our data set, perhaps with better results, and we will continue to work on this in the future.

6. Conclusion

In this paper, the interpretability model is used to explain and optimize the defect model. We validated our experiment with an in-depth study of six open source projects. Our experimental results show that the random forest model in RQ1 can predict software defects well, with an accuracy of 71.52% and a recall rate of 68.88%. In RQ2, we innovatively used LIME model to explain the software defect prediction model and results. The existing Just-in-Time Defect Prediction technique only predicts the possibility of defects in a change. It does not focus on what defects are predicted by software defects, such as the type of defects. There were no studies on defect types and locations. Defect types can describe the causes and characteristics of defects. Defect location refers to the module, file, function, or even line of code in which the defect resides. Understanding defect types and locations can help developers quickly fix defects. We used the LIME model and derived the SP-LIME interpretability tool to evaluate the characteristics of the most influential in our data, use these characteristics to training model, found that we can use the previous 45% effort to achieve 96% of the original workability, we want to be able to in future engineering applications, pay attention to the cause of the main features of the defects, thus reducing the cost of software defect prediction. Future research focuses on optimizing the distribution of unbalanced data at a deeper level and optimizing the processing of integrated learning algorithms to obtain faster model running performance and higher prediction accuracy.

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.

Acknowledgements

This research was supported by the 2021 Key R&D Program in Shaanxi Province, China (2021GY-041) and the National Natural Science Foundation of China special project capability-based construction method and execution mechanisms for ubiquitous operating systems (62141208).

References

Bejjanki, K.K., Gyani, J., Gugulothu, N., 2020. Class imbalance reduction (cir): a novel approach to software defect prediction in the presence of class imbalance. Symmetry 12 (3), 407.

Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P., 2011. Don't touch my code! examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 4–14.

Catal, C., Diri, B., 2009. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. Inform. Sci. 179 (8), 1040–1058.

Da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E., 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Trans. Softw. Eng. 43 (7), 641–657.

D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, pp. 31–41.

Efron, B., 1983. Estimating the error rate of a prediction rule: improvement on cross-validation. J. Amer. Statist. Assoc. 78 (382), 316–331.

Eyolfson, J., Tan, L., Lam, P., 2011. Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 153–162.

Fu, W., Menzies, T., 2017. Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 72–83.

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., Ubayashi, N., 2014. An empirical study of just-in-time defect prediction using cross-project models. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 172–181.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE, pp. 78–88.

Herzig, K., Just, S., Zeller, A., 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, pp. 392–401.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E., 2016. Studying just-in-time defect prediction using cross-project models. Empir. Softw. Eng. 21 (5), 2072–2106.

Kamei, Y., Monden, A., Matsumoto, S., Kakimoto, T., Matsumoto, K.-i., 2007. The effects of over and under sampling on fault-prone module detection. In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007). IEEE, pp. 196–204.

Kamei, Y., Shihab, E., 2016. Defect prediction: Accomplishments and future challenges. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 5. IEEE, pp. 33–45.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2012. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773.

Keung, J., Kocaguneli, E., Menzies, T., 2013. Finding conclusion stability for selecting the best effort predictor in software effort estimation. Autom. Softw. Eng. 20 (4), 543–567.

Khuat, T.T., Gabrys, B., 2020. A comparative study of general fuzzy min–max neural networks for pattern classification problems. Neurocomputing 386, 110–125.

Khuat, T.T., Le, M.H., 2020. Evaluation of sampling-based ensembles of classifiers on imbalanced data for software defect prediction problems. SN Comput. Sci. 1 (2), 1–16.

Kim, S., Whitehead, E.J., Zhang, Y., 2008. Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34 (2), 181–196.

Kim, S., Zhang, H., Wu, R., Gong, L., 2011. Dealing with noise in defect prediction. In: 2011 33rd International Conference on Software Engineering (ICSE). IEEE, pp. 481–490.

Kim, S., Zimmermann, T., Pan, K., James, Jr., E., et al., 2006. Automatic identification of bug-introducing changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). IEEE, pp. 81–90.

Laradji, I.H., Alshayeb, M., Ghouti, L., 2015. Software defect prediction using ensemble learning on selected features. Inf. Softw. Technol. 58, 388–402.

LaToza, T.D., Venolia, G., DeLine, R., 2006. Maintaining mental models: a study of developer work habits. In: Proceedings of the 28th International Conference on Software Engineering. pp. 492–501.

Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans. Softw. Eng. 34 (4), 485–496.

Li, Y., Wong, W.E., Lee, S.-Y., Wotawa, F., 2019. Using tri-relation networks for effective software fault-proneness prediction. IEEE Access 7, 63066–63080.

Liu, W., Wang, S., Chen, X., Jiang, H., 2018. Predicting the severity of bug reports based on feature selection. Int. J. Softw. Eng. Knowl. Eng. 28 (04), 537–558.

Liu, X.-Y., Wu, J., Zhou, Z.-H., 2008. Exploratory undersampling for class-imbalance learning. IEEE Trans. Syst. Man Cybern. B 39 (2), 539–550.

Ljung, L., 2001. Black-box models from input–output measurements. In: Imtc 2001. Proceedings of the 18th IEEE Instrumentation and Measurement Technology Conference. Rediscovering Measurement in the Age of Informatics (Cat. No. 01ch 37188), Vol. 1. IEEE, pp. 138–146.

Marks, L., Zou, Y., Hassan, A.E., 2011. Studying the fix-time for bugs in large open source projects. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering. pp. 1–8.

Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K.-i., Nakamura, M., 2010. An analysis of developer metrics for fault prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. pp. 1–9.

McIntosh, S., Kamei, Y., 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Trans. Softw. Eng. 44 (5), 412–428.

Mende, T., 2010. Replication of defect prediction studies: problems, pitfalls and recommendations. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. pp. 1–10.

Mende, T., Koschke, R., 2009. Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering. pp. 1–10.

Menzies, T., Dekhtyar, A., Distefano, J., Greenwald, J., 2007. Problems with precision: A response to comments on'data mining static code attributes to learn defect predictors'. IEEE Trans. Softw. Eng. 33 (9), 637–640.

Mitchell, M.W., 2011. Bias of the random forest out-of-bag (oob) error for certain input parameters. Open J. Stat. 1 (03), 205.

Mockus, A., Weiss, D.M., 2000. Predicting risk of software changes. Bell Labs Tech. J. 5 (2), 169–180.

Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th International Conference on Software Engineering. pp. 181–190.

Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th International Conference on Software Engineering. pp. 284–292.

Neto, E.C., Da Costa, D.A., Kulesza, U., 2018. The impact of refactoring changes on the szz algorithm: An empirical study. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 380–390.

Newman, M., 2002. Software Errors Cost Us Economy $59.5 Billion Annually. NIST Assesses Technical Needs of Industry to Improve Software-Testing.

Osman, H., Ghafari, M., Nierstrasz, O., Lungu, M., 2017. 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.

Pascarella, L., Bacchelli, A., 2017. Classifying code comments in java open-source software systems. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, pp. 227–237.

Pascarella, L., Palomba, F., Bacchelli, A., 2019. Fine-grained just-in-time defect prediction. J. Syst. Softw. 150, 22–36.

Qu, Y., Li, F., Chen, X., 2020. Lal: Meta-active learning-based software defect prediction. Int. J. Perform. Eng. 16 (2).

Ribeiro, M.T., Singh, S., Guestrin, C., 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. pp. 1135–1144.

Shihab, E., Hassan, A.E., Adams, B., Jiang, Z.M., 2012. An industrial study on the risk of software changes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11.

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

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2016. Automated parameter optimization of classification techniques for defect prediction models. In: Proceedings of the 38th International Conference on Software Engineering. pp. 321–332.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2018. The impact of automated parameter optimization on defect prediction models. IEEE Trans. Softw. Eng. 45 (7), 683–711.

Visani, G., Bagli, E., Chesani, F., Poluzzi, A., Capuzzo, D., 2020. Statistical stability indices for lime: obtaining reliable explanations for machine learning models. J. Oper. Res. Soc. 1–11.

Wang, S., Li, Y., Mi, W., Liu, Y., 2020. Software defect prediction incremental model using ensemble learning. Int. J. Perform. Eng. 16 (11).

Wong, W.E., Debroy, V., Surampudi, A., Kim, H., Siok, M.F., 2010. Recent catastrophic accidents: Investigating how software was responsible. In: 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. IEEE, pp. 14–22.

Wong, W.E., Li, X., Laplante, P.A., 2017. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. J. Syst. Softw. 133, 68–94.

Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H., 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 157–168.

Yatish, S., Jiarpakdee, J., Thongtanunam, P., Tantithamthavorn, C., 2019. Mining software defects: should we consider affected releases? In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 654–665.

Zubrow, D., 2009. IEEE Standard Classification for Software Anomalies. IEEE Computer Society.