Automatic Identification of High Impact Bug Report by Product and Test Code Quality

Jianshu Ding

Department of Computer Science and Engineering,
East China University of Science and Technology,
Shanghai, 200237, P. R. China
djs@mail.ecust.edu.cn,

Guisheng Fan

Department of Computer Science and Engineering,
East China University of Science and Technology,
Shanghai, 200237, P. R. China
gsfan@ecust.edu.cn,

Huiqun Yu

Department of Computer Science and Engineering,
East China University of Science and Technology,
Shanghai, 200237, P. R. China
yhq@ecust.edu.cn,

Zijie Huang

Department of Computer Science and Engineering,
East China University of Science and Technology,
Shanghai, 200237, P. R. China
hzj@mail.ecust.edu.cn,

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Bug reports are submitted by the software stakeholders to foster the location and elimination of bugs. However, in large-scale software systems, it may be impossible to track and solve every bug, and thus developers should pay more attention to High-Impact Bugs (HIB). Previous studies analyzed textual descriptions to automatically identify HIBs, but they ignored the quality of code, which may also indicate the cause of HIBs. To address this issue, we integrate the features reflecting the quality of production (i.e., CK metrics) and test code (i.e., test smells) into our textual similarity based model to identify HIBs. Our model outperform the compared baseline by up to 39% in terms of AUC-ROC and 64% in terms of F-Measure. Then, we explain the behaviour of our model by using SHAP to calculate the importance of each feature, and we apply case studies to empirically demonstrate the relationship between the most important features and HIB. The results show that several test smells (e.g., Assertion Roulette, Conditional Test Logic, Duplicate Assert, Sleepy Test) and product metrics (e.g., NOC, LCC, PF,

and ProF) have important contributions to HIB identification.

Keywords: high impact bug report; test code smell; product metric; empirical software engineering.

1. Introduction

To report and track the process of bug elimination, software bug reports could be submitted and updated by developers, testers, and users in online Issue Tracking Systems. The severity and impact of bug reports are manually assigned by software bug triagers [1]. In software systems with growing complexity, since the emergence of new bugs are inevitable, the triage and analysis of them could become very challenging, and it is almost impossible to allocate software quality assurance (SQA) resources to identify and resolve all of them [2–4]. Therefore, developers should pay more attention to high impact bug (HIB) reports to prevent HIBs from affecting core functionality of software projects. Thus, the ability to automatically identify HIBs from all available bug reports has become one of the most expected features of software tracking system from practitioners [5].

Researchers have been focusing on identifying HIBs. For example, Ohira et al. [6] manually identified HIBs after reviewing 4002 bug reports in 4 open-source projects, and they defined 6 types of HIBs namely Surprise, Dormant, Blocker, Security, Performance, and Breakage bugs. Based on the previous HIB dataset, many predictors have been proposed to automatically detect HIBs using the text information in the report. However, since the root cause of software bugs are erroneous and sub-optimal implementations in code, prior studies did not take code-related information into consideration, and they derived unpromising prediction performance [7–11].

Software code quality could be measured by product metrics such as the CK (Chidamber and Keremer) metrics [12]. CK metrics evaluate the extent of coupling, cohesion, and complexity of object oriented design, and they were proved effective as the backbone for object-oriented design metric [13]. Various studies indicated the effectiveness of CK metrics in empirical studies of software engineering [14, 15], and they were frequently used in software defect prediction studies [16]. Apart from the CK metrics, the quality of software testing may also impact software reliability. Testing is a crucial SQA activity which aims at revealing potential bugs. Recent study [17] revealed test smell (i.e., sub-optimal test code implementation) could help software defect prediction by providing additional predictive power on post-release defects.

To foster HIB prediction using code quality information, we build machine learners based on test smells related to tests of production code as well as product metrics with imbalanced learning sampling strategies including Random Over-Sampling (ROS), Random Under-Sampling (RUS), and Synthetic Minority Over-sampling TEchnique (SMOTE). [18–20].

The main contribution of our work are listed as follows.

1. We propose a machine learning based HIB prediction model approach using

product metrics, test smell features, and textual similarity. The performance is better than the model proposed by prior studies in all indicators, and it is also the first model to use product metrics and test smells as features to predict HIB.
2. We use SHAP (SHapley Additive exPlanations) [21] to explain the reason why our model derives better results in prediction by revealing feature importance and exploiting case studies.
3. We discuss empirically the relationship between test smells, product metrics, and prediction outcomes to reveal the relationship between code quality and HIBs.

The extension and improvement with respect to the preliminary conference paper [22] includes:

1. We extend the features by adding 49 product metrics. The model proposed in this paper outperforms the model of the preliminary conference paper [22] by up to 2% in terms of AUC and 4% in terms of F-Measure.
2. We extend the discussion of features' predictive power and model behavior in terms of the impact of product metrics to HIB prediction.
3. We correct some issues in our dataset and experiments, e.g., some data in the conference version are redundant since they do not contain Java code.

This paper is organized as follows. In Section 2 we summarize related literature. Section 3 presents how we construct our dataset, while Section 4 outlines the settings and research questions, as well as the concerned evaluation metrics. In Section 5 we discuss the results of our experiment, while Section 6 overviews the threats to validity and our effort to cope with them. Finally, Section 7 concludes the paper and describes future research.

2. Related Work

This section introduces research related to three major aspects of this paper, i.e., HIBs, test smells, and product metrics.

2.1. *High-Impact Bugs*

Various activities in the bug management process and end-users may be impacted by HIBs. In this subsection, we introduce 6 types of HIBs outlined in [6] .

Surprise bugs. A surprise bug [10] can disturb the workflow and/or task scheduling of developers, since it appears at unexpected timing (e.g., bugs detected in post-release) and locations (e.g., bugs found in files that are rarely changed in pre-release). Shihab et al. showed that surprise bugs exist in only 2% of all files [9]. However, surprise bugs may disturb developers' task scheduling greatly.

Dormant bugs. A dormant bug [23] is defined as a bug that was introduced in one version (e.g., Version 1.1) of a system, yet it is not reported until after the next immediate version (i.e., a bug is reported against Version 1.2 or later) [23].

Research [23] found that 33% of the reported bugs in Apache Software Foundation (ASF) projects were dormant bugs.

Blocking bugs. A blocking bug is a bug that blocks other bugs from being fixed [24]. It often happens if a dependency relationship exists among software components. Since a blocking bug inhibit developers from fixing other dependent bugs, it has a high impact on developers' task scheduling. Due to this reason, blocking bugs have to be fixed early to not prevent other bugs from getting fixed.

Security bugs. A security bug [9] can raise a serious problem which often impacts on uses of software products directly. It exploits to gain unauthorized access or privileges in the systems. In general, security bugs are supposed to be fixed as soon as possible.Since Internet devices (e.g., smartphones) and their users are increasing every year, security issues of software products should be of interest to many people. In general, security bugs are supposed to be fixed as soon as possible.

Performance bugs. A performance bug [25] is defined as programming defects that cause significant performance degradation. The performance degradation contains poor user experience, lazy application responsiveness, lower system throughput, and needles waste of computational resources.

Breakage bugs. A breakage bug [10] is a functional bug which is introduced into a product because the source code is modified to add new features or to fix existing bugs. A breakage bug can cause usable functions in old versions unusable after releasing new versions.

In this paper, we mainly identify two high-impact bugs, i.e., surprise and breakage bugs. Surprise Bugs may appear at unexpected timing (e.g., bugs detected in post-release) and locations (e.g., bugs found in files that are rarely changed in pre-release), which may cause high impact on software. Breakage bugs are the bugs which can have impact on functionality of whole software system, leading to the error of system usability [8].

2.2. *HIB Identification*

Developers rely on bug reports to locate and fix bugs. However, researches revealed that bug reports could be very ambiguous or even incorrect in meaning, which is very unfavorable for developers to focus on the most important bug through manually inspecting bug reports. Therefore, researchers focused on helping developers by automaticly identify HIBs.

Thung et al. [26] and Valdivia-Garcia et al. [24] first categorized bug reports into bug types and identified bugs respectively, and subsequent studies [7–9] found that there exists obvious data imbalance in HIB report datasets. The authors proposed an approach to use machine learning classification methods to identify HIBs based on the textual information from the description and summary features in the bug report, which improves the performance of prediction. Yang et al. [8] converted the text information in the bug report into word frequency vectors after basic processing (i.e., removing the stop words, numbers and punctuation marks, using Iterated

Lovins Stemmer to transform them to their root forms). The machine learning model was built based on the word frequency vector. Furthermore, they also tried to address the data imbalance problem in HIB datasets by exploiting data sampling.

Compared with previous studies, the major improvements and contributions of our work are:

1. Since HIB datasets are highly imbalanced, the up-mentioned research were not reporting reliable prediction performance evaluation metrics which are insensitive to imbalanced data (e.g., AUC). Meanwhile, their performances are not promising (e.g., F-Measure < 0.7 [27] in most predicted projects).
2. Prior studies ignored the production and test code quality of the software project, which could be the root cause of defects.
3. We explain our model from a different and more detailed aspect comparing to prior studies, i.e., we focus on the impact of test-related features to software bug report prediction by applying an explainable AI (XAI) technique called SHAP.

2.3. *Test Smell Detection and Impact Analysis*

Test smells will be more likely to present in poorly designed tests. Van Deursen et al. [28] first proposed the definition of various test smells including Assertion Roulette and General Fixture, which reflected many aspects of sub-optimal implementation of test code. Later, Peruma et al. [29] extended prior studies and also proposed several new test smell including Conditional Test Logic, indicting other multiple aspects including quality, reliability, and maintainability. In this paper, we focus on the most frequently occurred test smells which could be detected by a reliable evaluation tool, i.e., TsDetect [29].

Although test smell detection is actively studied, little is known about the impact of test smells to test code and software reliability. Kim et al. [17] proposed that test smell metrics can provide additional explanatory power on post-release defects. Therefore, we believe that we could use test smells to capture code quality related information and improve the performance of HIBs prediction.

2.4. *CK Product Metrics*

The original metric set proposed by Chidamber and Kemerer [12] contains 6 OOP (Object-oriented Programming) design metrics, which measures the inheritance, coupling, and cohesion of code. Such metrics were proved related to software defects [30]. We use an active open-source implementation of CK metrics designed by Aniche [31], which has been proved reliable in related work [32, 33]. The implementation also extended the original metrics by involving more method-level metrics since they could provide design information at a smaller granularity.

Table 1: Statistics of the HIB report dataset

Project	Total	Java	Surprise	Breakage
Camel	579	507	200	41
Derby	731	634	94	165
Wicket	663	581	213	50

3. Dataset Generation

This section describes how we generate our dataset using TsDetect, CK metrics, and textual similarity of bug reports.

3.1. *HIB Dataset Pre-Processing*

The original HIB dataset was obtained by Ohira et al. [6] by manually inspecting 4 open-source projects and selecting issue reports containing two categories of BUGS and IMPROVEMENTS.

Since we mainly focus on bugs, we first excluded IMPROVEMENT reports. The TsDetect tool currently only supports test smell detection for Java projects. Therefore, we discarded one project (i.e., Ambrari) with little Java code, and we consider 3 projects, namely Camel, Derby, and Wicket. Table. 1 shows the statistics of numbers of HIB reports of the processed dataset.

3.2. *Extracting Textual Similarity Matrix*

The textual pattern of a bug report could be used to identify other similar reports. We map the title and description of various bug reports into textual similarity matrix. Then, we combine the similarities of the most similar bug reports (i.e., using them as weight) with code quality features (i.e., test smell and CK metric) to summarize the overall quality of code related to similar bug reports. Therefore, we can generate features for HIB identification.

In terms of the processing of texts, we pre-processed the descriptions of bug reports using a process similar to Yang et al.'s work [8] , i.e., removing stop words, numbers, punctuation and stemming words to their root forms (e.g., "reading" and "reads" are reduced to "read"). Then, we calculate the word frequency vector for each stemmed term and retrieve the term frequency vector. Meanwhile, we remove words that only appear once, because they do not have much information and may even introduce noise. Based on the word frequency vector that has been processed, we can calculate the degree of similarity of each item, and build a similarity matrix based on the result. Finally, we assign textual similarity as weights of test smells and product metric features to each test class.

In terms of bug report similarity calculation, we should specify the most similar K bug reports for generating the similarity matrix. The K parameter will be tuned in the subsequent experiment section.

Table 2: Description of Test Smells

Test smell	Abbr.	Description
Assertion Roulette	AR	A test method that contains more than one assertion statement without an explanation/message (parameter in the assertion method).
Conditional Test Logic	CTL	A test method that contains one or more control statements (i.e, if, switch, conditional expression, for, foreach, and while statement).
Constructor Initialization	CI	A test class that contains a constructor declaration. This may introduce side effects when the test class inherits another class, i.e., the parent class's constructor will still be invoked.
Default Test	DT	A test class named either 'ExampleUnitTest' or 'ExampleInstrumentedTest'.
Duplicate Assert	DA	A test method that contains more than one assertion statement with the same parameters.
Eager Test	EGT	A test method that contains multiple calls to multiple production methods which is difficult in maintenance.
Empty Test	ET	A test method that does not contain a single executable statement.
Exception Catching Throwing	ECT	A test method that contains either a throw statement or a catch clause.
General Fixture	GF	Not all fields instantiated within the setUp method of a test class are used by all test methods in the test class.
Ignored Test	IT	A test method or class that contains the @Ignore annotation.
Lazy Test	LT	Multiple test methods calling the same production method.
Magic Number Test	MNT	A test method that contains unexplained and undocumented numeric literals as parameters or identifiers, which increases difficulty in maintenance.
Mystery Guest	MG	A test method containing object instances of files and databases classes.
Redundant Print	RP	A test method that invokes either the print or println or printf or write method of the System class.
Redundant Assertion	RA	A test method that contains an assertion statement in which the expected and actual parameters are the same.
Resource Optimism	RO	A test method utilizes an instance of a File class without calling its exists(), isFile() or notExists() methods.
Sensitive Equality	SE	A test method that invokes the toString() method of an object.
Sleepy Test	ST	A test method that invokes the Thread.sleep() method.
Unknown Test	UT	A test method that does not contain a single assertion statement and @Test(expected) annotation parameter.
Verbose Test	VT	A test method that is too complicated or cumbersome
Print Statement	PS	Print statements in unit tests are redundant as unit tests are executed as part of an automated script and do not affect the failing or passing of test cases.

3.3. *Detecting Test Smell*

We exploit the state-of-the-art TsDetect [29] tool for detecting 21 test smells in Table. 2. Since this tool is only available for Java projects, we discard non-Java codes. Then, we evaluate test smells of test code in the three open-source projects. In case of the absence of test code related to the production code, we set the value of such features to 0, such circumstance accounts for 7.6% of all data.

3.4. *Evaluating CK Metrics*

We use the implementation of the Aniche CK [12] which is used in various studies. The advantage of this implementation is that it uses static analysis (i.e., no need for code compilation), and it calculates both class-level and method-level code metrics. An obvious improvement brought by CK metrics is that data for code components without test code could be populated by CK metrics, which could improve the performance of the predictor. Since there are 49 CK metrics, we list their abbreviations and short descriptions in Table. 3. The full description of CK metris is available in

Table 3: Description of CK metrics

CK Metric	Abbr.
Coupling between objects/Modified	CBO/CBO Modified
FAN-IN/FAN-OUT	FAN-IN/FAN-OUT
Depth Inheritance Tree	DIT
Number of Children	NOC
Number of Fields	TM/SM/PM/PriM/ProM/DM/VM/AM/FM/SynM
Number of Methods	TF/SF/PF/PriF/ProF/DF/VF/AF/FF/SynF
Number of Visible Methods	TvM/SvM/PvM/PrivM/ProvM/DvM/VvM/AvM/FvM/SynvM
Number of Static Invocations	NOSI
Response/Weight Method for a Class	RFC/WMC
Lines of Code	LOC
Lack of Cohesion of Methods	LCOM/LCOM*
Tight/Loose Class Cohesion	TCC/LCC
Quantity of Returns/Loops/Comparisons	RQT/LQT/CQT
Quantity of Try-catches/Parenthesized Expressions	TQT/PQT
String Literals	SL
Quantity of Number/Math Operations/Variables	NQT/MQT/VQT
Max Nested Blocks	MNB
Quantity of Anonymous classes, Inner Classes, and Lambda Expressions	AQT
Number of Unique Words/Log Statements	UQT/LSQT
Has Javadoc	HJ
Modifiers	MOD
Usage of Each Variable/Field	UEV/UEF
Method Invocations	MI

our online appendix [a].

4. Empirical Study Setup

The goal of our study is to improve the performance of HIB identification, with the purpose of evaluating to what extent the code quality can help predict HIB. To

[a]1

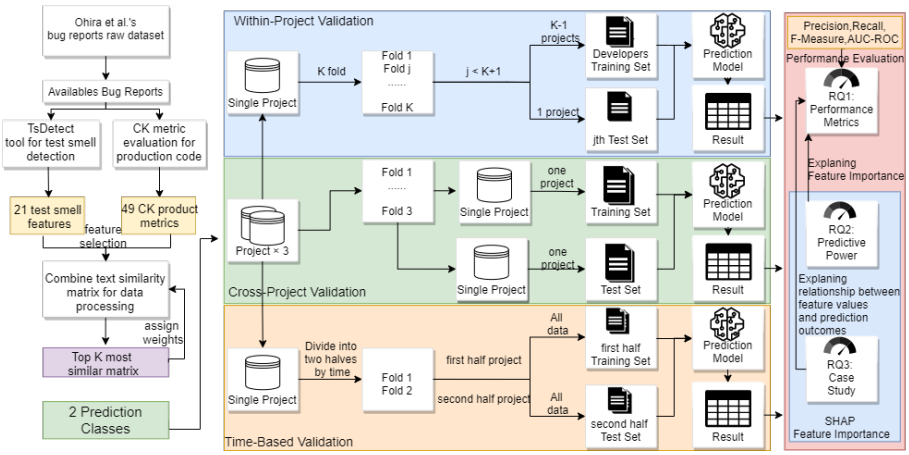


Fig. 1: Overview of methodology and experiment process.

these ends, we propose the following research questions. Moreover, our experimental setup process is shown in the Fig. 1.

• **RQ1.** How does our proposed model perform compared with the baseline models?

Based on the dataset we processed in Section 3, we conduct feature selection to remove the features that correlate with each other. Then, we build prediction model using machine learning classifier with different selections of parameters [34]. To determine the significance of performance improvement of our model, we compare it with a state-of-the-art [8] and the model proposed in our preliminary conference version [22].

• **RQ2.** What is the predictive power of the test smell features to the occurrence of high-impact bugs?

RQ2 aims at explaining the contribution of each feature to the predictor's prediction performance. We calculate the SHAP feature importance of each feature to illustrate their predictive power for identifying the likelihood of the HIB presence to figure out the contribution of each feature in our prediction model.

• **RQ3.** What is the relationship between the value distribution of the proposed features and the prediction results?

In addition to the contribution of model performance and features, we intend to explain the results of the model by revealing its behavior. We also include case studies to empirically demonstrate the impact of the important features to the presence of HIBs. Threfore, we may provide suggestions to developers based on our findings to help them prevent HIB from occurring.

4.1. *RQ1: Defining and Evaluating the Proposed Model*

Feature Selection. High correlation among features could negatively impact model performance and hinder the interpretability of models [35]. Thus, we apply a new feature selection method called Autospearman [36] which can reduce the multicollinearity and correlation while preserving most features.

Data Balancing. As reported by previous studies, the dataset in the HIB report is highly imbalanced. [8]. Breakage and Surprise bugs only account for 15% and 29% of all the bugs, which may affect the performance of the model. Therefore, we adopt ROS, RUS, and SMOTE to deal with data imbalance to improve prediction performance [37, 38].

Validation Scenarios. We build models separately in within-project, time-based, and cross-project scenarios. In order to ensure that our experimental results are credible and verifiable, we set within-project validations follow the classical 10-Fold Cross-validation strategy so that we can have unbiased and stable performance [39, 40] data. However, it may cause a violation in time series since we may violate the principle of time-set-based schemes, leading to the use of subsequent time data to predict previous data. Thus, we introduce time-based validation similar to our baseline did [8], which ordered the data chronically and used the first half of the data as training set to predict the second half of the data as test set. At the same

time, in order to verify the potential in terms of generalization of our model, we also included cross-project experimental settings into the scope of investigation.

Training Classifiers and Threshold Tuning. We use the Scikit-Learn package in Python [41] to train machine learners using multiple classifiers, which have been used in previous studies [39, 42, 43]. For clarification, we used the original implementation of the baseline to train and evaluate their model. We use three unbalanced strategies to record data. There are many choices for its own test smell. Different K values represent the number of most similar reports. The value is selected as $\{5, 10, 15, 20, 25, 30, 35\}$.

Performance Assessment. We use four commonly used measurement indicators including precision, recall, F-Measure, and AUC-ROC [44]. We rely on these indicators to select the best predictive model from all three scenarios.

4.2. *RQ2: Explaining the Predictive Power of Features*

We need to explain the predictive power of each feature in the model with the best predictive performance in RQ1 to answer RQ2. The explanation of AI models is essential for software engineering practitioners [45, 46]. Although complex machine learning models are almost black boxes, they could be explained by using interpretable approximation of the original model.

We apply the SHAP algorithm which has been studied empirically in recent software engineering paper validating the stability of feature importance methods [27, 45]. SHAP uses the game theory based Shapley values [47] to distribute the credit for a classifier's output among its features [27, 48, 49]. For each data point in the training set, SHAP transforms features into a space of simplified binary features as input. Afterwards, SHAP builds the model for explanation defined as a linear function of binary values, more specifically in Eq. 1:

$$g(z) = \phi_0 + \sum_{i=1}^{M} \phi_i z_i \ , . \tag{1}$$

where $\phi_i \in R$ for $i = 0, 1, ..., M$ are Shapley values. M is the number of simplified input features, and $z_i = \{z_1, z_2, ..., z_M\}$ are binary vectors in simplified input space where $z \in [0, 1]^M$. Note that $|\phi_i|$ are feature importance scores that are guaranteed in theory to be locally, consistently, and additively accurate for each data point [27, 48]. We use the Python implementation of SHAP [48] in our study.

In order to rank the features' importance with effect size awareness, we also involve the Scott-Knott Effect Size Difference (SK-ESD) test. Scott-Knott test [49] uses hierarchical clustering methods to group the means of assessment metrics of multiple models, which is a statistical metric for comparing and differentiating model performance. Scott-Knott test assumes the distribution of input data to be normally distributed. The SK-ESD test is an enhanced version of the original Scott-Knott test that corrects the non-normal distribution of the input. Meanwhile, it uses Cliff's Delta as an effect size measure to merge groups which have negligible

effect sizes. Scott-Knott and SK-ESD tests have already been applied in software defect prediction [50]. We use the R implementation of Tantithamthavorn et al. [40].

Both SHAP and SK-ESD algorithms are executed on each prediction class independently.

4.3. *RQ3: Case Study of Prediction Outcomes*

In this RQ, we expect to exploit case studies to find explanations for local prediction instances which can reflect the models' behavior in real-world scenarios. We manually search the projects marked with the occurrence of HIBs from the original dataset and then compare the descriptions in bug reports proposed by the developers with the code information in the corresponding projects to determine whether the features selected in our model are indeed inline with problems which exist HIBs in real-world scenarios.

5. Result and Discussion

In this section, we answer the proposed research questions by demonstrating and discussing the results of the experiment. We also outline our findings at the bottom of each subsection.

5.1. *RQ1: Model Definition and Performance*

In terms of feature selection, we select 13 CK metrics and 16 test smells, and the full names and descriptions are available in our online appendix [b].

We apply a variety of classifiers to find the best-performed one. We use Random Forest as the classifier and report the median of weighted average performance in Table. 4 and Table. 5. Meanwhile, we also include the performance of the baseline work [8] and the model of our preliminary conference paper [22]. The superior performance is bolded. Fig. 2 also depict the best results among the three scenarios. Moreover, We also show the effects of different K values on the prediction performance of the model in Fig. 3, and we pick 30 as the best K as it derives the most ideal performance.

Due to the differences in the specification and development among various projects, the standards for writing test codes and production codes are significantly different. Therefore, for cross-project predictions, the performance of the model is worse than within-project and time-based prediction.

The proposed model significantly outperformed the baseline model [8] (using only textual similarity) by up to 39% in terms of AUC-ROC and 64% in terms of F-Measure. Meanwhile, it also improved the model designed in our preliminary conference paper [22] (using textual similartiy and test smell features), the improvement brought by additional product metrics is up to 2% in AUC-ROC and 4% in

[b]1

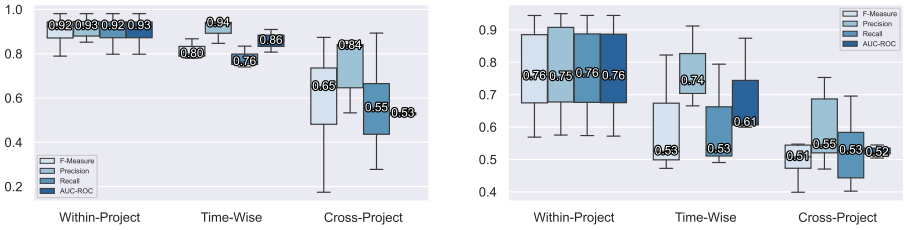


Fig. 2: Prediction performance. (a) breakage bugs. (b) surprise bug

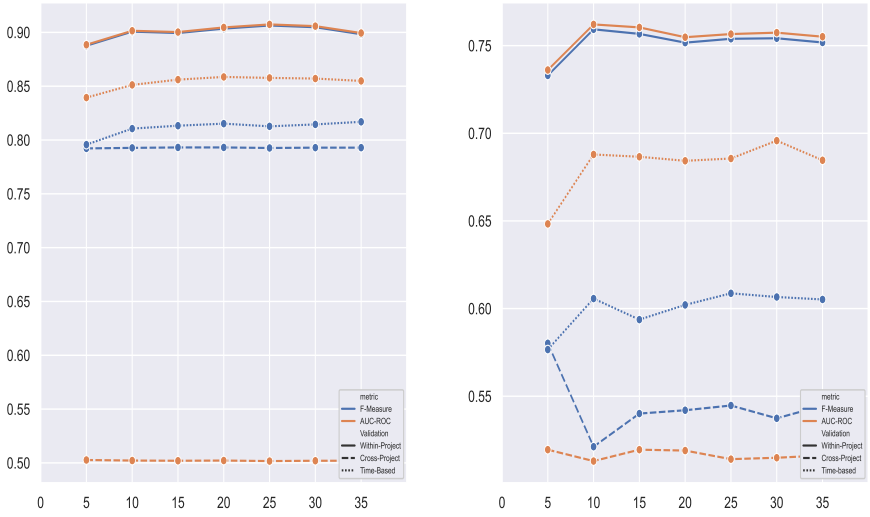


Fig. 3: Performance variations of different K selections. (a) breakage bugs. (b) surprise bugs.

F-Measure. We believe that production code quality boosts the performance of the proposed model, especially in the code that test smells rarely present.

Finding 1. Our proposed model is significantly superior to the compared baseline study, and the newly integrated production code features are also improving the performance based on the model built in our preliminary conference paper by up to 2% in AUC-ROC and 4% in F-Measure.

Table 4: Results of Breakage Prediction

	AUC-ROC			F-Measure		
	Proposed	Conference	Baseline	**Proposed**	Conference	Baseline
Within-P.	**0.91**	0.89	0.52	**0.91**	0.89	0.36
Cross-P.	**0.53**	**0.53**	0.52	**0.65**	0.61	0.23
Time-B.	**0.86**	0.83	0.52	**0.80**	0.77	0.52
	Recall			Precsion		
	Proposed	Conference	Baseline	**Proposed**	Conference	Baseline
Within-P.	**0.91**	0.89	0.81	**0.92**	0.91	0.32
Cross-P.	**0.55**	0.52	0.51	**0.84**	0.80	0.27
Time-B.	**0.76**	0.71	0.68	**0.94**	0.93	0.30

Table 5: Results of Surprise Prediction

	AUC-ROC			F-Measure		
	Proposed	Conference	Baseline	**Proposed**	Conference	Baseline
Within-P.	**0.76**	0.75	0.53	**0.76**	0.74	0.45
Cross-P.	**0.52**	0.51	0.51	**0.51**	**0.51**	0.43
Time-B.	**0.61**	0.60	0.53	**0.53**	0.52	0.47
	Recall			Precsion		
	Proposed	Conference	Baseline	**Proposed**	Conference	Baseline
Within-P.	**0.75**	**0.75**	0.62	**0.76**	**0.75**	0.30
Cross-P.	**0.53**	0.51	0.49	**0.55**	0.54	0.29
Time-B.	**0.53**	0.51	0.50	**0.74**	0.72	0.31

5.2. *RQ2: Features' Predictive Power*

Fig. 4-Fig. 6 depict the mean (in dashed lines), median (in solid lines), as well as the rank of each feature's importance. Features given the same rank are marked in the same color. The importance presented in this section can be used to describe the contribution of features to the performance of our model.

All the features we apply have contribution to our model. However, the contribution differs from features. In all scenarios, ST, CTL, and AR are always the top three that contribute to the model, while DA, ProF, LCC, PF, and NOC are also ranked in the top three contributing features except for the time-based prediction of surprise bugs.

In particular, Sleep Test (ST) is the feature with the best predictive performance among all the features, indicating that when the behaviour of program is less predictable, e.g., does not know the duration of pausing, it is more likely to cause HIBs. Consistent with our previous research, Assertion Roulette (AR) and Conditional Test Logic (CTL) are still ranked in the top three predictive power, indicating that more unexplained assertion statements may affect the reliability of the software system and excessive control flow statements may cause difficulty in implementing correct code, which leads to the appearance of HIBs.

At the same time, we also discover some new features with high predictive contribution. The high contribution of NOC indicates that when a class itself has subclasses, the complexity of the structure will have a certain impact on HIB.

The appearance of Loose Class Cohesion (LCC) shows that the degree of cohesion between classes will also have a certain impact on the model. PF and ProF reflect that the number of the field members of a class may also affect the model. The interpretation of high importance of Duplicate Assert (DA) is similar to AR.

These results show that our model reveals potential relationship between the code quality and the HIBs in the software project. Further details of such relationship and case studies will be presented in RQ3.

Finding 2. Certain features of test code and production code quality are related to the prediction of HIB occurrence. At the same time, each feature contributes to the prediction of the model, indicating that it is feasible to combine test smell and CK metrics to predict HIB.

5.3. *RQ3: Case Study*

Fig. 7-Fig. 8 depict examples we found from the project that can confirm the features of our proposed model. The content in the Description comes from the bug reports submitted by the developers in the problem summary, and the Code is found from the source code according to the project number, which can show the problem referred to in the description. In order to reduce unnecessary and invalid information in the article, the useless information in Description and Code is omitted, and the corresponding parts are displayed in bold.

We find that ST, CTL, AR, DA, ProF, LCC, PF, and NOC may have a great influence on HIB prediction, and ST, AR, CTL features are the top three in terms of predictive ability.

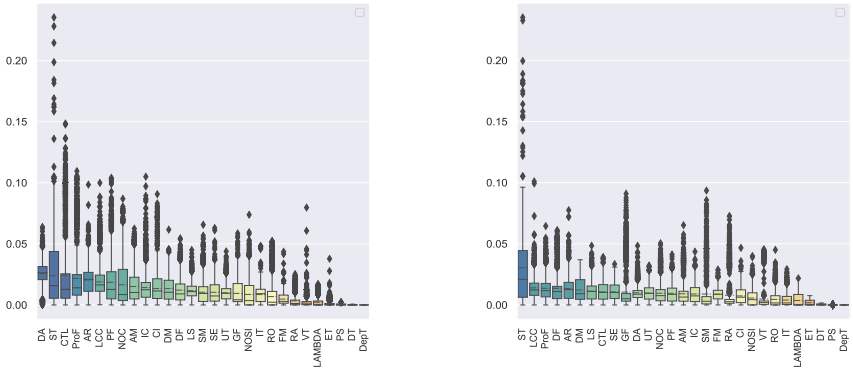
In order to verify whether and how our model reveal the relationship between code quality and HIB in more realistic scenarios, we exploit case studies by manually investigating data related to HIBs in the dataset, i.e., we intend to figure out whether the ST, AR, CTL smell is really connected with the description provided by the developer in the bug reports. At the same time, whether product metrics such as DA, NOC, LCC, PF, ProF are closely related to the emergence of HIB is also worth investigating.

5.3.1. *Case Study of Prediction Using Test Smell*

For example in Fig. 7, the bolded statement in the figure shows the error caused by multiple obfuscated assertions in the test code showing that repeated assertions can lead to confusion within the project, which is in line with the definition of AR and DA. From this case study, we can find that AR is indeed one of the causes of HIB.

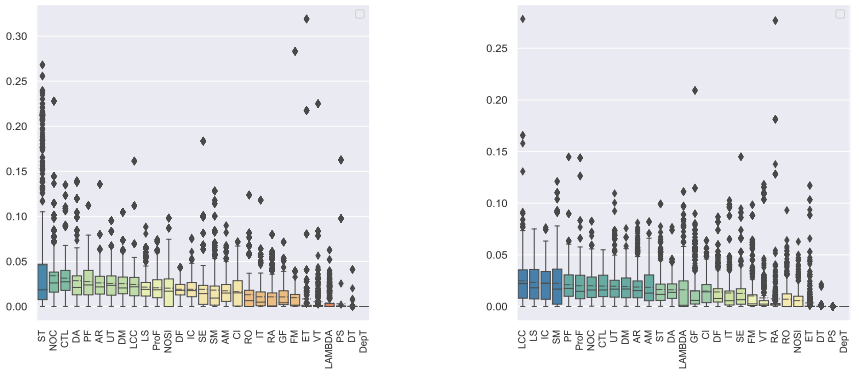
In terms of CTL and ST, we also find that there exist other cases in the dataset that can support our findings. We do not intend to present all evidence in the paper since it may be too redundant. However, we provide an online appendix[c] to

[c]http:



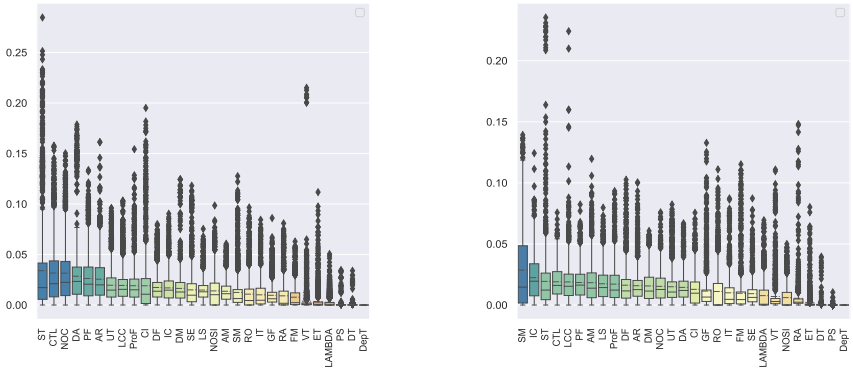
(a) (b)

Fig. 4: Cross-project prediction feature importance classified by SK-ESD. (a) breakage bugs. (b) surprise bugs.



(a) (b)

Fig. 5: Time-based prediction feature importance classified by SK-ESD. (a) breakage bugs. (b) surprise bugs.



(a) (b)

Fig. 6: Within-project prediction feature importance classified by SK-ESD. (a) breakage bugs. (b) surprise bugs.

Description:
..... it had the following incorrect state: **started = true starting = false stopped = true stopping = false** The stopped should have been changed to false as its started.

Code:

```
public void testServiceSupport() throws Exception {
MyService service = new MyService();
service.start();
assertEquals(true, service.isStarted());
assertEquals(false, service.isStarting());
assertEquals(false, service.isStopped());
assertEquals(false, service.isStopping());
service.stop();
assertEquals(true, service.isStopped());
assertEquals(false, service.isStopping());
assertEquals(false, service.isStarted());
assertEquals(false, service.isStarting());}
```

Fig. 7: Example of an HIB related to Assertion Roulette.

demonstrate the bug reports related to CTL and ST. Therefore, according to both case study , we can prove that test smells and product metrics are related to certain types of HIBs.

5.3.2. *Case Study of Prediction Using CK Metrics*

The appearance of the product metrics in the top three of the contribution rankings shows that the production code quality is also important for predicting HIB.

From example in Fig. 8, we find that NOC can lead to positive prediction of HIB occurrence, indicating that complex code structure is more difficult to understand, and it is easier to cause errors. At the same time, the contribution of LCC also shows that if the purpose and responsibility of the code is confusing, it is more likely to cause HIB.

Finding 3. Our model results show that (i) complex redundant and difficult to understand production code and test cases are more likely to cause HIBs, (ii) ST, AR, CTL and other test smells are related to the appearance of HIB, and (iii) information in production code is also meaningful for predicting HIB. We recommend that developers should not only pay attention to the writing of test cases, but also pay attention to the readability and maintainability of the production code itself, especially to avoid the relevant test smell, complexity, and cohesion issues we mentioned above.

Description:
.....When running this code, the output results are messed up. When I delete some unnecessary classes and associated code, the code runs correctly. ..... The Language resolveLanguage is not the relevant class that needs to be used in this project. ..... Some public classes can be merged. Unable to resolve scripting languages in OSGi environment The OsgiLanguageResolver.java created since CAMEL-1221 does not resolve scripting languages by using the default resolver declared in camel-script. .....

Code:

```
public class    OsgiLanguageResolver implements LanguageResolver {
      private static final Logger LOG = LoggerFactory.getLogger(OsgiLanguageResolver.class);

      private final BundleContext bundleContext;

      public OsgiLanguageResolver(BundleContext bundleContext) {
            .....(278 lines of code)
      }

      public Language resolveLanguage(String name, CamelContext context) {
            ..... (136 lines of code)
      }

      protected Language getLanguage(String name, CamelContext context) {
            .....(114 lines of code)
      }

     protected LanguageResolver getLanguageResolver(String name, CamelContext context) {
            .....(57 lines of code)
      }

}
```

Fig. 8: Example of an HIB related to high NOC and LCC.

6. Threats to Validity

This section introduces threats to validity and the way we address them.

6.1. *Construct Validity*

The major threat to construct validity is the reliability of our datasets. Our datasets come from three different sources, i.e., test smell detection results, CK metric results and bug report dataset.

In order to get the test result of test smell, we exploit a detection tool called TsDetect. For detecting various possible problems in the test code, the authors of the article designed this tool, and they also manually reviewed different data sets to

verify the correctness and effectiveness of the generated tool. In addition, we strictly follow the installation, configuration, and execution guides of the detection tool. As for the software project code, we download the specified version of the dataset from GitHub.

Relating to CK metric, we also use a mature testing tool, which was born to just calculate the classLevelMetrics. CK metric is an active open-source project led by senior reaseraker, which has been used many times by other former work [13,15,51]. Using CK metric to get defect information in production code can help improve prediction performance. however, The correctness of the test results has not yet been proven.

The bug report dataset we used was created by Ohira et al. through manual review of 4 open-source projects with 4002 issue reports included. Since they are obtained by manual review, it is inevitable that there might be subjective problems or errors in data labeling.

6.2. *Conclusion Validity*

In terms of the reliability of model settings, we employ three strategies to deal with the data imbalance of the dataset. We also report results of classical evaluation metrics, i.e., precision, recall, F-Measure, and AUC-ROC. Furthermore, we apply statistical tests, e.g., SK-ESD, to validate the significance of our conclusions. The reliability of feature importance algorithm is also a threat to conclusion validity. Since model explanation is still a new topic in software analytics [52], these solutions may be imperfect.

6.3. *External Validity*

External validity includes the programming specifications of different projects, the coding styles of different developers, and the diversity of different software projects. We have already discussed this issue in the cross-project prediction scenario, and we should address it by involving more projects in future work.

7. Conclusion

This paper investigated whether it is possible to use test smell features and product metrics of textual similar bug reports to identify HIB. Experimental results showed that our proposed model is better than the model without product metrics in the preliminary conference paper and the compared baseline.

In addition, each proposed feature has a certain contribution to the prediction performance of the model. Meanwhile, we also managed to discover empirical evidence of such relationship in case studies. In conclusion, we recommended that developers pay attention to the quality of production code and test code, especially to avoid some of the test smells and design problems we mentioned that may lead to the positive prediction of HIB occurrence.

Future work includes: (1) manually construct a dataset of projects using other programming languages, (2) involving developer-related metrics, and (3) evaluating empirically the usability of our model in real-world scenarios.

Acknowledgements

This work was partially supported by the National Natural Science Foundation of China under Grant No. 61772200, and the Natural Science Foundation of Shanghai under Grant No. 21ZR1416300.

References

[1] P. Hooimeijer and W. Weimer, Modeling bug report quality, in *ACM/IEEE International Conference on Automated Software Engineering(ASE'07)*, (ACM, 2007), pp. 34–43.

[2] T. Xie, L. Zhang, X. Xiao, Y.-F. Xiong and D. Hao, Cooperative software testing and analysis: Advances and challenges, *Journal of Computer Science and Technology* **29**(4) (2014) 713–723.

[3] M. Karim, A. Ihara, X. Yang, H. Iida and K. Matsumoto, Understanding key features of high-impact bug reports, in *Proceedings - 8th IEEE International Workshop on Empirical Software Engineering in Practice, IWESEP 2017*, 2017, pp. 53–58.

[4] H. Li, G. Gao, R. Chen, X. Ge, S. Guo and L.-Y. Hao, The influence ranking for testers in bug tracking systems, *International Journal of Software Engineering and Knowledge Engineering* **29**(1) (2019) 93–113.

[5] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng and B. Xu, How practitioners perceive automated bug report management techniques, *IEEE Transactions on Software Engineering* **46**(8) (2020) 836–862.

[6] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara and K. Matsumoto, A dataset of high impact bugs: Manually-classified issue reports, in *IEEE International Working Conference on Mining Software Repositories*, 2015, pp. 518–521.

[7] X. Yang, D. Lo, Q. Huang, X. Xia and J. Sun, Automated identification of high impact bug reports leveraging imbalanced learning strategies, *Proceedings - International Computer Software and Applications Conference* **1** (2016) 227–232.

[8] X. Yang, D. Lo, Q. Huang, X. Xia and J. Sun, High impact bug report identification with imbalanced learning strategies, *Journal of Computer Science and Technology (JCST)* **32** (2017) 181–198.

[9] M. Gegick, P. Rotella and T. Xie, Identifying security bug reports via text mining: An industrial case study, in *Proceedings - International Conference on Software Engineering*, 2010, pp. 11–20.

[10] E. Shihab, A. Mockus, Y. Kamei, B. Adams and A. Hassan, High-impact defects: A study of breakage and surprise defects, in *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering(SIGSOFT/FSE)*, 2011, pp. 300–310.

[11] H. Li, Y. Qu, S. Guo, G. Gao, R. Chen and G. Chen, Surprise bug report prediction utilizing optimized integration with imbalanced learning strategy, *Complexity* **2020** (2020) 33–48.

[12] S. Chidamber and C. Kemerer, A metrics suite for object oriented design, *IEEE Transactions on Software Engineering* **20**(6) (1994) 476–493.

[13] B. Suri and S. Singhal, Investigating the oo characteristics of software using ckjm metrics *2015 4th International Conference on Reliability, Infocom Technologies and Optimization: Trends and Future Directions, ICRITO 2015* 2015, pp. 44–51.
[14] D. Shawky and S. Abd-El-Hafiz, Characterizing software development method using metrics, *Journal of Software: Evolution and Process* **28**(2) (2016) 82–96.
[15] R. Malhotra and A. Bansal, Fault prediction considering threshold effects of object-oriented metrics, *Expert Systems* **32**(2) (2015) 203–219.
[16] D. Radjenović, M. Heričko, R. Torkar and A. Živkovič, Software fault prediction metrics: A systematic literature review, *Information and Software Technology* **55**(8) (2013) 1397–1418.
[17] D. Kim, T.-H. Chen and J. Yang, The secret life of test smells - an empirical study on test smell evolution and maintenance, *Empirical Software Engineering* **26**(5) (2021) 47–59.
[18] R. Nayak, L. Buys and J. Lovie-Kitchin, Data mining in conceptualising active ageing, *Conferences in Research and Practice in Information Technology Series* **61** (2006) 39–45.
[19] A. Ghanem, S. Venkatesh and G. West, Learning in imbalanced relational data, in *Proceedings - International Conference on Pattern Recognition*, 2008, pp. 167–175.
[20] N. Chawla, K. Bowyer, L. Hall and W. Kegelmeyer, Smote: Synthetic minority over-sampling technique, *Journal of Artificial Intelligence Research* **16** (2002) 321–357.
[21] S. Lundberg and S.-I. Lee, A unified approach to interpreting model predictions, *Advances in Neural Information Processing Systems* **78** (2017) 4766–4775.
[22] D. Jianshu, F. Guisheng, Y. Huiqun and H. Zijie, Automatic identification of high impact bug report by test smells of textual similar bug reports, in *2021 The 21st International Conference on Software Quality, Reliability and Security(QRS)*, 2021, pp. 121–134.
[23] T.-H. Chen, M. Nagappan, E. Shihab and A. Hassan, An empirical study of dormant bugs, in *11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings*, 2014, pp. 82–91.
[24] H. Valdivia-Garcia, E. Shihab and M. Nagappan, Characterizing and predicting blocking bugs in open source projects, *Journal of Systems and Software* **143** (2018) 44–58.
[25] A. Nistor, T. Jiang and L. Tan, Discovering, reporting, and fixing performance bugs, in *IEEE International Working Conference on Mining Software Repositories*, 2013, pp. 237–246.
[26] F. Thung, D. Lo and L. Jiang, Automatic defect categorization, in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2012, pp. 205–214.
[27] G. Rajbahadur, S. Wang, G. Ansaldi, Y. Kamei and A. Hassan, The impact of feature importance methods on the interpretation of defect classifiers, in *IEEE Transactions on Software Engineering*, 2021, pp. 11–25.
[28] V. Deursen and L. Moonen, Refactoring test code, in *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.
[29] A. Peruma, K. Almalki, C. Newman, M. Mkaouer, A. Ouni and F. Palomba, Tsdetect: An open source test smells detection tool, in *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1650–1654.
[30] R. Subramanyam and M. Krishnan, Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects, *IEEE Transactions on Software Engineering* **29**(4) (2003) 297–310.

[31] M. Aniche, *Java code metrics calculator (CK)*, (2015). Available in https://github.com/mauricioaniche/ck/.

[32] M. Iammarino, F. Zampetti, L. Aversano and M. Di Penta, An empirical study on the co-occurrence between refactoring actions and self-admitted technical debt removal, *Journal of Systems and Software* **178** (2021) p. 110976.

[33] A. Mori, G. Vale, M. Viggiato, J. Oliveira, E. Figueiredo, E. Cirilo, P. Jamshidi and C. Kastner, Evaluating domain-specific metric thresholds: An empirical study, in *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2018, pp. 41–50.

[34] Y. Tian, N. Ali, D. Lo and A. Hassan, On the unreliability of bug severity data, *Empirical Software Engineering* **21**(6) (2016) 2298–2323.

[35] J. Jiarpakdee, C. Tantithamthavorn and A. E. Hassan, The impact of correlated metrics on the interpretation of defect models, *IEEE Transactions on Software Engineering* **47**(2) (2021) 320–331.

[36] J. Jiarpakdee, C. Tantithamthavorn and C. Treude, Autospearman: Automatically mitigating correlated software metrics for interpreting defect models, in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 92–103.

[37] C. Tantithamthavorn, A. Hassan and K. Matsumoto, The impact of class rebalancing techniques on the performance and interpretation of defect prediction models, in *IEEE Transactions on Software Engineering*, 2018, pp. 1109–1121.

[38] N. Sae-Lim, S. Hayashi and M. Saeki, Context-based code smells prioritization for prefactoring, *IEEE International Conference on Program Comprehension* **16** (2016) 56–78.

[39] F. Palomba and D. Tamburri, Predicting the emergence of community smells using socio-technical metrics: A machine-learning approach, *Journal of Systems and Software* **171** (2021) 574–581.

[40] C. Tantithamthavorn, S. McIntosh, A. Hassan and K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, *IEEE Transactions on Software Engineering* **43**(1) (2017) 1–18.

[41] F. Pedregosa and G. Varoquaux, Scikit-learn: Machine learning in python, *Journal of Machine Learning Research* **12** (2011) 2825–2830.

[42] F. Pecorelli, F. Palomba, F. Khomh and A. De Lucia, Developer-driven code smell prioritization, in *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories(MSR 2020)*, 2020, pp. 220–231.

[43] F. Palomba, M. Zanoni, F. Fontana, A. De Lucia and R. Oliveto, Toward a smell-aware bug prediction model, *IEEE Transactions on Software Engineering* **45**(2) (2019) 194–218.

[44] A. Bradley, The use of the area under the roc curve in the evaluation of machine learning algorithms, *Pattern Recognition* **30**(7) (1997) 1145–1159.

[45] G. Esteves, E. Figueiredo, A. Veloso, M. Viggiato and N. Ziviani, Understanding machine learning software defect predictions, *Automated Software Engineering* **27**(3-4) (2020) 369–392.

[46] G. Santos, E. Figueiredo, A. Veloso, M. Viggiato and N. Ziviani, Predicting software defects with explainable machine learning, in *PervasiveHealth: Pervasive Computing Technologies for Healthcare*, 2020, pp. 56–74.

[47] S. L. S., A value for n-person games, *Contributions to the Theory of Games* **2**(28) (1953) 307–317.

[48] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal and LeeS.-I., From local explanations to global understand-

ing with explainable ai for trees, *Nature Machine Intelligence* **2**(1) (2020) 56–67.

[49] A. Scott and M. Knott, A cluster analysis method for grouping means in the analysis of variance, *Biometrics* **30**(3) (1974) 507–512.

[50] X. Yang, H. Yu, G. Fan and Y. K., Dejit: A differential evolution algorithm for effort-aware just-intime software defect prediction, *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* **31**(3) (2021) 289–310.

[51] R. C. d. Cruz and M. M. Eler, Using a cluster analysis method for grouping classes according to their inferred testability: An investigation of ck metrics, code coverage and mutation score, in *2017 36th International Conference of the Chilean Computer Science Society (SCCC)*, 2017, pp. 1–11.

[52] H. Hofmann, H. Wickham and K. Kafadar, Letter-value plots: Boxplots for large data, *Journal of Computational and Graphical Statistics* **26**(3) (2017) 469–477.