Automatic Identification of High Impact Bug Report by Test Smells of Textual Similar Bug Reports

Jianshu Ding, Guisheng Fan*, Huiqun Yu*, Zijie Huang*
Department of Computer Science and Engineering, East China University of Science and Technology
E-mail: djs@mail.ecust.edu.cn, {gsfan, yhq}@ecust.edu.cn, hzj@mail.ecust.edu.cn

***Abstract*—Bug reports are written by the software stakeholders to track software defects and vulnerabilities. Since Software Quality Assurance (SQA) resources are limited, developers tend to resolve High-Impact Bugs (HIB) in advance. Prior research identified HIBs by analyzing the textual information in bug reports. However, they only consider textual information instead of the root cause of bugs, such as code quality. Since prior study revealed software test smells (i.e., sub-optimal test code implementation) are related to bug proneness, we intend to measure test smell distribution in textual similar bug reports to identify HIB reports. We first construct an effective model, which outperforms the baseline by 29.3% in terms of AUC-ROC. Secondly, we use SHAP to compute the importance of test smell features. Finally, we conduct an empirical survey to discuss the relationship between test smell and HIB reports. Result shows that Assertion Roulette and Conditional Test Logic test smell are important factors in distinguishing the types of bug reports.**

Keywords- high impact bug report; test code smell; software quality assurance; empirical software engineering

I. INTRODUCTION

Software bug triages often need to deal with bugs submitted by developers, testers, and users in the bug tracking system [1]. As software projects evolve, the number of bug reports will inevitably increase since various types of software bugs emerge. In complex systems, it is almost impossible to allocate Software Quality Assurance (SQA) resources to identify and to solve every bug [2]. Therefore, High-Impact Bug (HIB) reports deserve more attention from developers because they may block the core functionality of software. However, manual inspection to discriminate HIBs from all bug reports are time-consuming and inefficient [3][4]. Therefore, in order to reduce the workload of developers and better allocate SQA resources, automatically detecting HIBs from a large number of bug reports is among the most expected features of bug tracking systems from practitioners [5].

In response, HIB reports are actively studied. Ohira et al. [6] manually identified HIBs after reviewing 4002 bug reports in 4 open-source projects. Their research defined 6 types of HIBs including surprise, dormant, blocker, security, performance, and breakage bugs. Based on Ohira et al.'s HIB dataset, HIB predictors were proposed using textual information in description and summary of bug reports.

However, since the root cause of software bugs are erroneous implementations in code, prior studies did not take code related information into consideration, and they derived unpromising prediction performance [7][8][9][10][11].

Recent study [12] revealed test smell (i.e., sub-optimal test code implementation) could help software defect prediction by providing additional predictive power on post-release defects, which indicates potential chances of involving test smells into bug prediction related tasks. To this end, we propose an approach to build machine leaners based on test smells of the test files related to production code in the project with imbalanced learning sampling strategies including Random Over-Sampling (ROS), Random Under-Sampling (RUS), and Synthetic Minority Over-sampling TEchnique (SMOTE) [13][14][15]. Our model outperforms the state-of-the-art [8] by 29.3% in terms of AUC and 41.0% in terms of F-Measure.

The main contributions of the papers are listed as follows.

1) We propose a machine learning based approach using test smell and textual similarity. The prediction accuracy of this classifier is better than the model proposed by prior studies in all indicators, and it is also the first model to use test smell as features to predict HIB.

2) We use SHAP (SHapley Additive exPlanations) [16] 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 and prediction outcomes to reveal the potential relationship between test smells and HIBs.

This paper is organized as follows. In Section II we summarize related literature. Section III presents how we construct our dataset, while Section IV outlines the settings and research questions, as well as the concerned evaluation metrics. In Section V we discuss the results of our experiment, while Section VI overviews the threats to validity and our effort to cope with them. Finally, Section VII concludes the paper and describes future research.

II. RELATED WORK

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

A. High-Impact Bug (HIB)

HIBs can impose impact on a variety of activities in the bug management process and end-users. In this subsection, we introduce 6 types of HIBs outlined in [6] .

(1) 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-

* Corresponding Authors: Guisheng Fan, Huiqun Yu, Zijie Huang

release) and locations (e.g., bugs found in files that are rarely changed in pre-release).

(2) Dormant bugs: A dormant bug [17] 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) [17]. Research [17] found that 33% of the reported bugs in Apache Software Foundation (ASF) projects were dormant bugs.

(3) Blocking bugs: A blocking bug is a bug that blocks other bugs from being fixed [18]. 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.

(4) 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.

(5) Performance bugs: A performance bug [19] 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.

(6) 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 focus on two high-impact bugs, i.e., surprise bugs and breakage bugs. Surprise bugs are bugs which have high impact on developers. These bugs appear in unexpected timing (e.g., in post-release) or locations (e.g., in files that are rarely changed before), and they may disrupt existing plans and workflows of software development. Breakage bugs are the bugs which have high impact on the usability of software since these bugs break pre-existing functionality and harms significantly the user experience [8].

B. HIB Prediction

Developers of software projects rely on bug reports to locate and fix bugs. However, studies revealed that a great number of bug reports are ambiguous or even incorrect. Moreover, compared to other bugs, HIBs poses a greater threat to the operation of the software. Therefore, the automatic prediction of HIBs can help the developers to save efforts and resolve bugs more quickly.

Thung et al. [20] and Valdivia-Garcia et al. [18] first categorized bug reports into bug types and identified blocking bugs respectively, and subsequent studies [7][8][9] found that there exists obvious data imbalance in HIB report datasets. The authors collected the textual information in the description and summary features in the bug report, and they used machine learning classification methods to predict HIBs, which improves the accuracy of prediction. Yang et al. [8] performed HIB prediction using text information (i.e., word frequency vector) in the description and summary in the bug reports. Furthermore, they investigated the solution to the

TABLE I. DESCRIPTIONS OF HIBS

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.

imbalance problem in major HIBs (i.e., breakage and surprise bugs), and they exploited a variety of methods to deal with imbalance problems.

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 [21] in most predicted projects).

(2) Previous studies ignored the original code quality of the software project, which is the root cause of defects. We link smelly test code and related production code to reveal the test code quality of textual similar bug reports. As a result, our model achieves significant better performance.

(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 factors to software bug report prediction by applying an eXplainable AI (XAI) technique called SHAP.

C. Test Smell Detection and Impact Analysis

Test smells occur in poorly designed tests. The definition of Test smell was first proposed by Van Deursen et al. [22]. Moreover, they listed 11 types of test smells including AR and GF, which reflected many aspects of sub-optimal implementation of test code. Recently, Peruma et al. [23] proposed several new test smells including CTL. Their study contains 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 [23]. We list the concerned test smells in Table I.

Since test smell detection is actively studied, little is known about the impact of test smells to production code and software reliability. Kim et al. [12] 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.

III. Test Smell Detection And Textual Similarity Generation

This section describes how we generate our dataset using a test smell detection tool and textual similarity of bug reports.

A. Dataset Processing

The raw HIB dataset was obtained by Ohira et al. [6] through manual review of 4 open-source projects. The dataset contains 4002 issue reports, including Bugs and Improvements.

We exclude the Improvement issue reports. Since we intend to use Java test smell to reflect the code quality, we discard a project that contains little Java source code and has no surprise and breakage bugs. Table II shows the statistics of numbers of HIB reports of the processed dataset, and it also includes the number of breakage and surprise bugs in the remaining 3 projects (i.e., Camel, Derby, Wicket) that we used to construct dataset.

TABLE II. Statistics of Numbers of HIB Reports

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

B. Extracting Textual Similarity Matrix

To capture both textual and code information in bug reports, we combine the original code information with the text information in the bug report. We map the text information into a similarity matrix and combine the test smell detection results to generate a new dataset.

We first process all text information in the same way as Yang et al. [8] did, i.e., removing stop words, numbers, punctuation and transforming words to their root forms (e.g., "reading" and "reads" are reduced to "read"). Then we calculate the term frequency for each stemmed term, and we retrieve the term frequency vector. At the same time, we remove words that only appear once because they introduce noises. Based on the term frequency vector, we calculate the similarity matrix between each term frequency vector data, and we exploit the TsDetect tool to obtain test smell detection result for each test code file. Finally, we assign textual similarity as weights of test smells 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.

C. Detecting Test Smell

Although we know that there exist other test smell detection tools, some of them are not open source. Therefore, we exploit TsDetect [23] as a tool for detecting test smells in Table I. 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 their smell to 0, such circumstance accounts for 7.6% of all data.

IV. Empirical Study Setup

The *goal* of our research is to evaluate to what extent HIBs can be predicted by test quality, with the *purpose* of understanding the relationship between test smells and HIBs. Our experimental setup process is shown in the Figure 1. This section proposes our research questions and methodology.

A. Research Questions and Methodologies

RQ1: How does our proposed model perform compared with the baseline model?

Based on the dataset we processed in Section III, we conduct the Correlation-based Feature Selection (CFS) [24], and we remove features that correlate with each other. Then, we build prediction models using machine learning

classifier with different selections of parameters [25]. To determine the significance of performance improvement of our model, we compare it with a state-of-the-art [8].

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

RQ2 aims at understanding the features' contribution to the prediction performance. We examine the predictive ability of each feature of test smell features by examining their SHAP feature importance to figure out the contribution of each test smell in our prediction model.

RQ3: What is the relationship between the value distribution of the proposed features and the prediction results?

Apart from the model performance and the contribution of features, we also intend to explain the models' outcomes by revealing its behavior. We explain the relationship between the distribution of the values of the proposed features and the emergence of HIBs by exploiting statistical approaches. Furthermore, we intend to provide developers with suggestions based on our findings to help them prevent HIBs from occurring.

B. *RQ1: Defining and Evaluating the Proposed Model in Within-, Cross- and Time-based Scenarios.*

(1) Feature Selection: To remove multicollinearity, we exploit CFS as [21][26] suggested. This process could ensure higher interpretability of XAI results of our model.

(2) Data Balancing: Our dataset is highly imbalanced, which in line with the previous research [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 exploit three strategies to deal with data imbalance including ROS, RUS, and SMOTE [27][28].

(3) Validation Scenarios: We build models separately in both within-project, time-based, and cross-project scenarios. Within-project validation follows a classical 10-Fold Cross-Validation strategy to make sure we have unbiased and stable [29][30] performance data. However, it may cause a violation in time series as we may use future data to predict earlier 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.

(4) Training Classifiers and Threshold Tuning: We apply the Scikit-Learn package [31] from Python to train machine learners using multiple classifiers that have been used in previous studies [26][29][32]. To clarify, we use the original implementation of our baseline to train and evaluate their model. We record data using three imbalance strategies with various choices of its own test smell and different K values representing the number of most similar reports, whose value is selected as {5, 10, 15, 20, 25, 30, 35}.

(5) Performance Assessment: We compute classical performance metrics including precision, recall, F-Measure, and AUC- ROC [33] to pick the best performing model in all 3 scenarios.

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

To answer RQ2, we need to explain the extent of predictive power that each feature contributes to the best-performed model in RQ1. The explanation of AI models is essential for software engineering practitioners [34][35]. Although complex machine learning models are almost black boxes, they could be explained by using interpretable approximation of the original model.

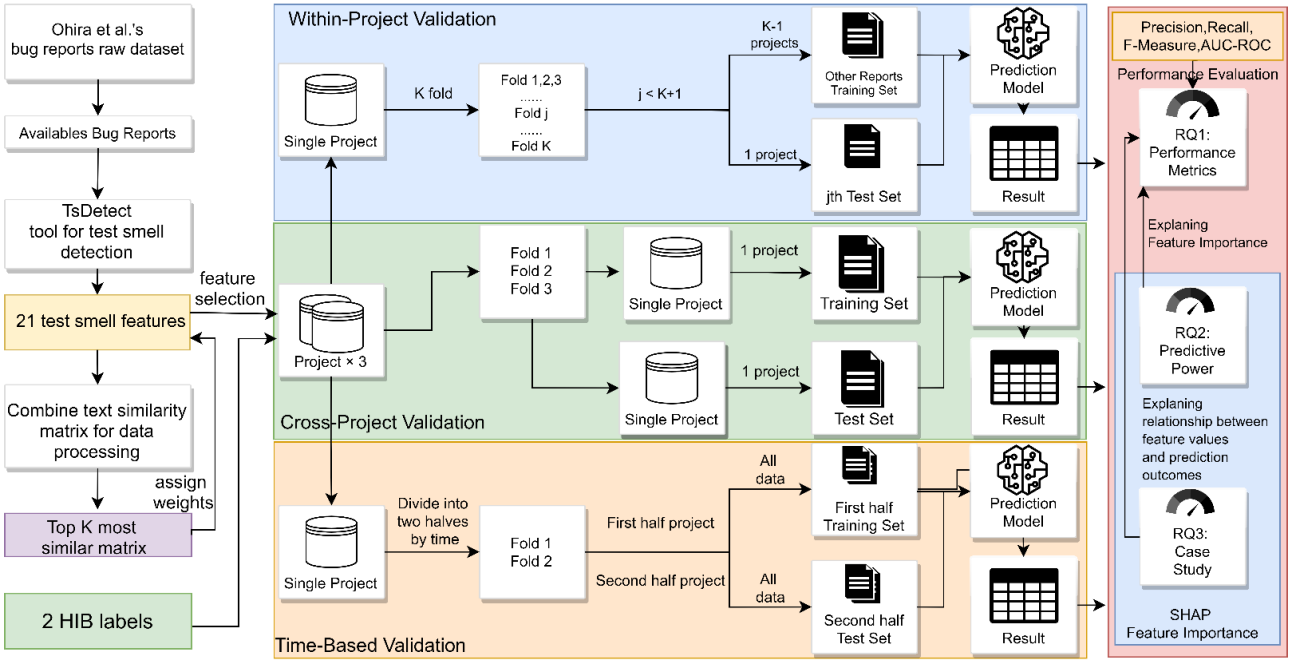


Figure 1. Overview of methodology and experiment process.

We apply the SHAP algorithm which has been studied empirically in recent software engineering paper validating the stability of feature importance methods [21][34]. SHAP uses the game theory based Shapley values [36] to distribute the credit for a classifier's output among its features [21][37][38]. 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 g for explanation defined as a linear function of binary values, more specifically in equation (1):

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

where $\varphi i \in$ 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 $|\varphi_i|$ are feature importance scores that are guaranteed in theory to be locally, consistently, and additively accurate for each data point [21][37]. We use the Python implementation of SHAP [37] 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 [38] 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 [39]. We use the R implementation of Tantithamthavorn et al. [30].

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

D. RQ3: Features' Impact to Prediction Outcomes

In this RQ, we expect to find trends from the feature impact to the prediction outcomes in our dataset. We measure the relationship of the proposed features and the prediction outcomes, which is positive SHAP $|\varphi_i| > 0$ for prediction result of positive (i.e., feature value that makes a sample more likely to be predicted as HIB) and negative SHAP $|\varphi_i|<0$ for negative prediction by providing visualizations including SHAP beeswarm-plot and boxen-plot. Additionally, we also exploit case studies to find explanations for local prediction instances which can reflect the models' behavior in real-world scenarios.

TABLE III. RESULTS OF BREAKAGE BUG PREDICTION

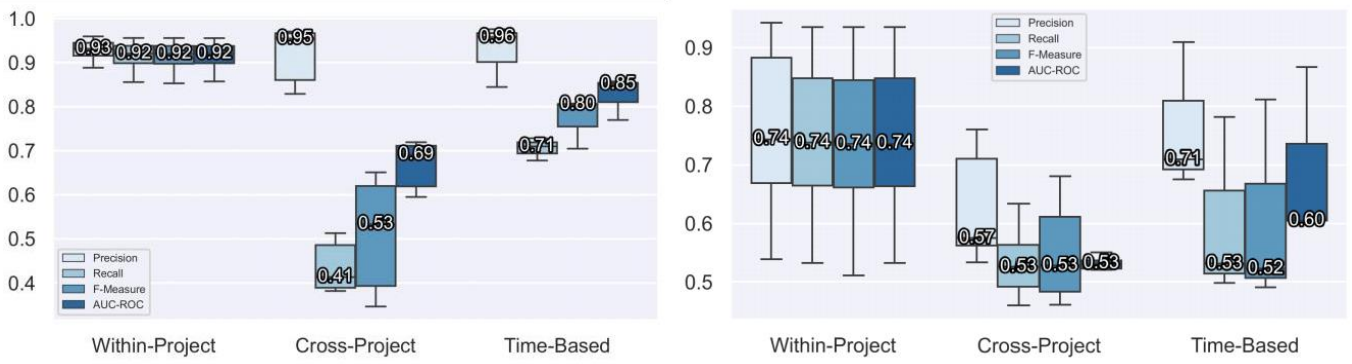
	AUC-ROC		**F-Measure**	
	Proposed	*Baseline*	***Proposed***	*Baseline*
Within-P.	**0.89**	0.52	**0.89**	0.36
Cross-P.	**0.56**	0.55	**0.78**	0.23
Time-B.	**0.83**	0.52	**0.77**	0.38
	Recall		**Precision**	
	Proposed	*Baseline*	***Proposed***	*Baseline*
Within-P.	**0.89**	0.81	**0.91**	0.32
Cross-P.	**0.74**	0.70	**0.85**	0.27
Time-B.	**0.71**	0.68	**0.93**	0.30

TABLE IV. RESULTS OF SURPRISE BUG PREDICTION

	AUC-ROC		**F-Measure**	
	Proposed	*Baseline*	***Proposed***	*Baseline*
Within-P.	**0.75**	0.53	**0.74**	0.45
Cross-P.	**0.51**	0.51	**0.57**	0.43
Time-B.	**0.69**	0.53	**0.61**	0.47
	Recall		**Precision**	
	Proposed	*Baseline*	***Proposed***	*Baseline*
Within-P.	**0.75**	0.62	**0.76**	0.30
Cross-P.	**0.58**	0.49	**0.59**	0.25
Time-B.	**0.60**	0.52	**0.76**	0.31

V. 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



(a) (b)

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

each subsection.

A. RQ1: Model Performance

For feature selection, ECT, SE, EGT, LT, DA, UT, MNT are removed by CFS since they strongly correlate with other test smells. Then, we exploit the classifiers and report the result of the best performed one, i.e., Random Forest, in Figure 2, which depict results for the 3 validation scenarios. We also demonstrate the line chart for the impact of different K selection to the model's performance in Figure 3. We pick 30 as the best K as it derives the most ideal performance. We also present the median of weighted average performance of our models comparing to our baseline in Table III and Table IV. The superior performance is bolded.

Due to the differences in the specification and development among various projects, the standards for writing test codes and production codes are significantly different. For cross-project predictions, the performance of the model is worse than within-project and time-based prediction. However, the median values of the model's performances show that the model still works better than previous studies in most cases.

Finding 1. Our proposed prediction model is significantly superior to the baseline model. In terms of cross-project prediction, our model reaches mean F-Measures of 78% and 57% for breakage and surprise bugs. For within-project and time-based prediction, the model achieves F-Measures of 89%, 77%, and 74%, 61%, respectively.

B. RQ2: Features' Predictive Power

Figures 4-6 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 test smells contribute to our model. However, the contribution of each feature to the model is not the same. In all cases, AR and CTL are always the top two features in the importance rank.

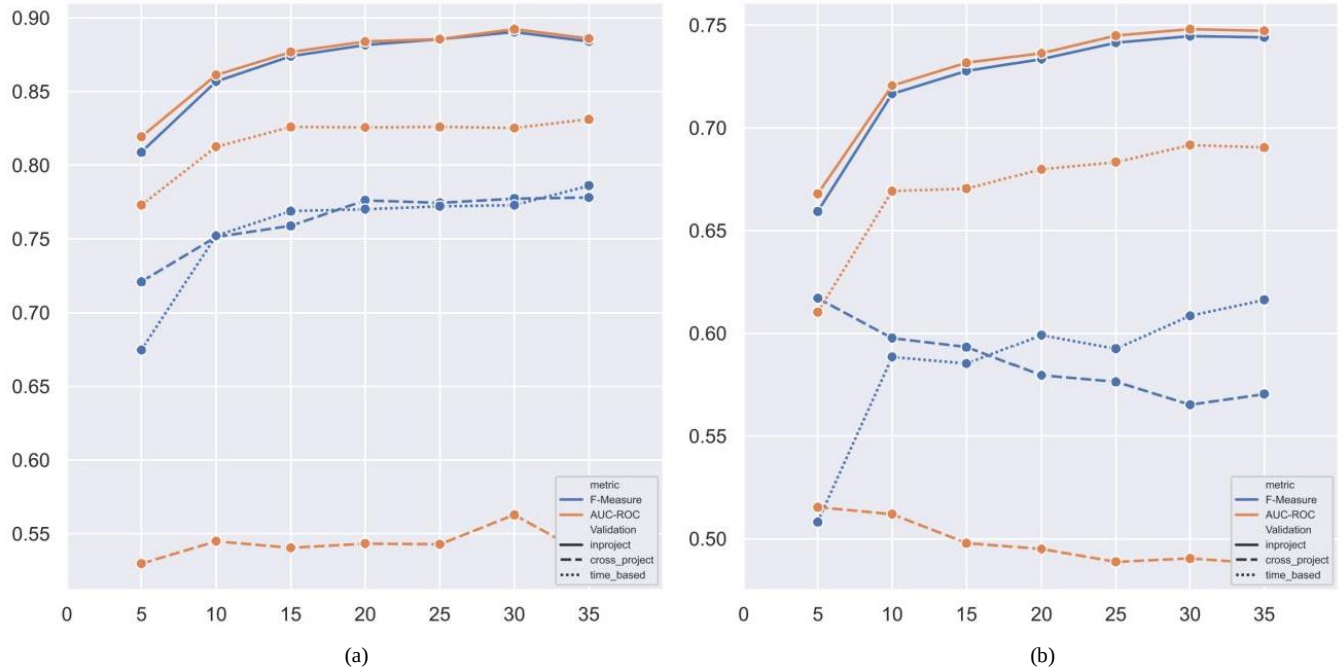
In particular, AR is the strongest factor among all the features in terms of predictive power, showing that more assertion statements without explanations may affect the reliability of software system. Meanwhile, CTL is also the strongest contributing factor among all the features, reflecting that too many control flow statements may cause loop errors when the program is running, which leads to the appearance of HIB.

These results show that there exists a strong correlation between the occurrence of HIB and the sub-optimal implementation in the test codes of software project, and further details and case studied will be discussed in RQ3.

Finding 2. AR and CTL have the most significant contribution to the prediction performance, indicating that redundant code and poor specification are related to the appearance of HIB. At the same time, every feature contributes to the correct prediction of HIB, which shows that it is feasible to take advantage of test smells to predict HIB.

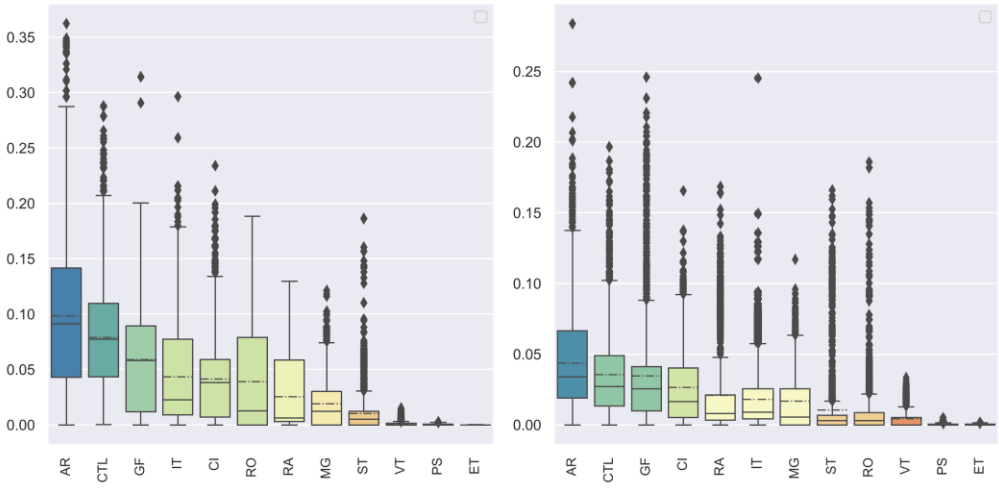
C. RQ3: Model Behavior

Figures 7-8 depict SHAP beeswarm-plots displaying the



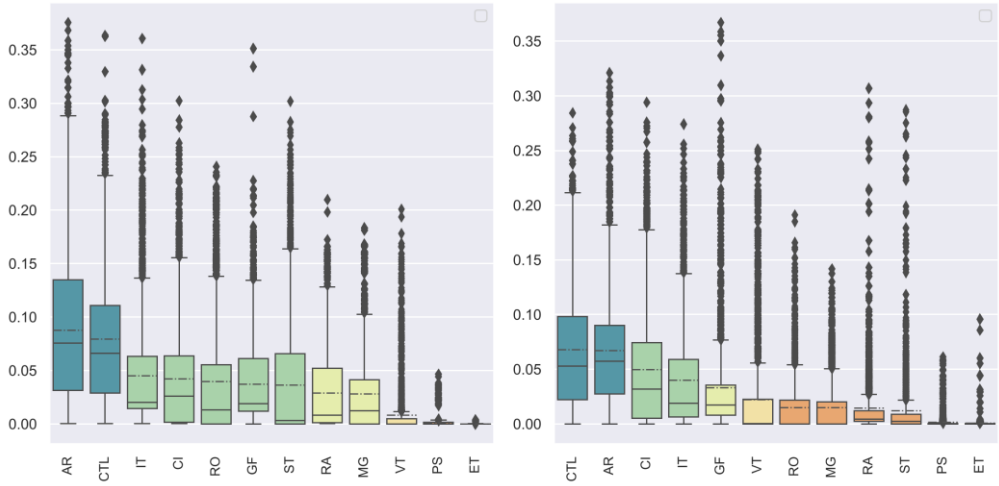
(a) (b)

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



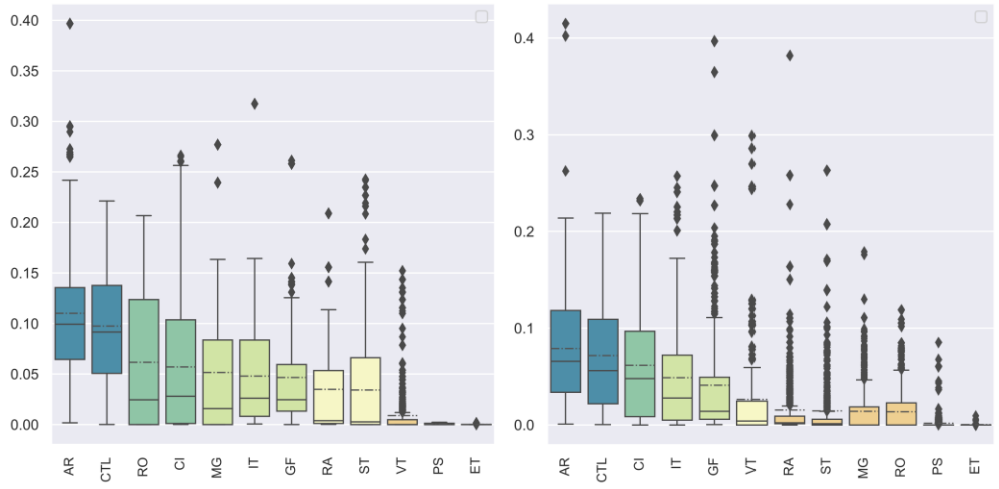
(a) (b)

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



(a) (b)

Figure 5. Within-project prediction feature importance classified by SK-ESD. (a) breakage bugs. (b) surprise bugs.



(a) (b)

Figure 6. Time-based prediction feature importance classified by SK-ESD. (a) breakage bugs. (b) surprise bugs.

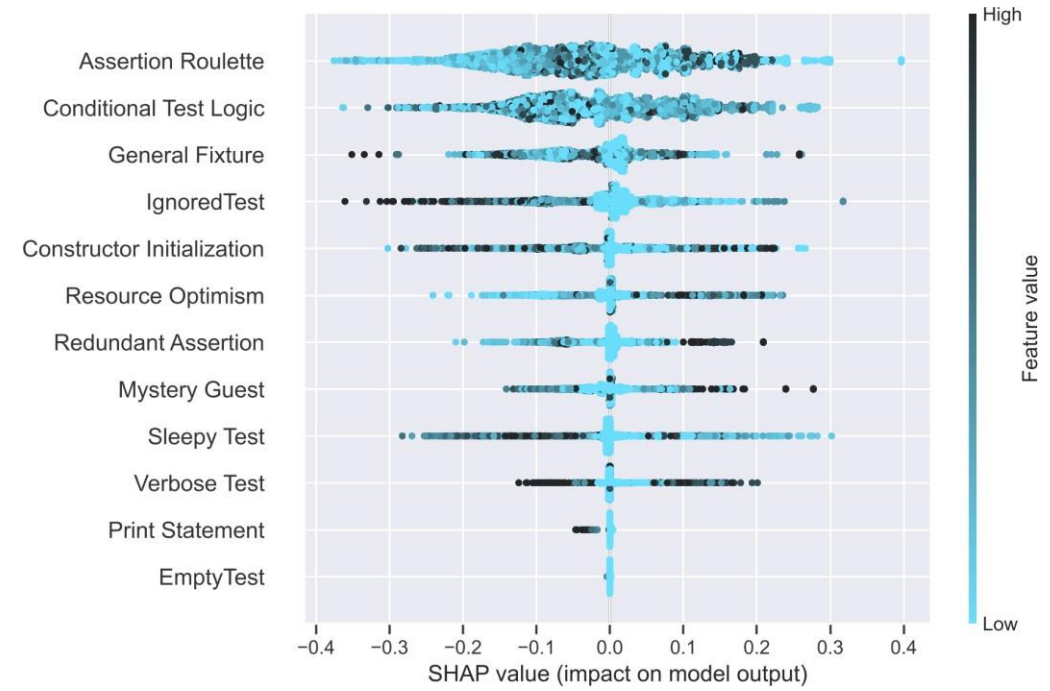


Figure 7. Feature values' impact to the correct prediction cases of the breakage bug model.

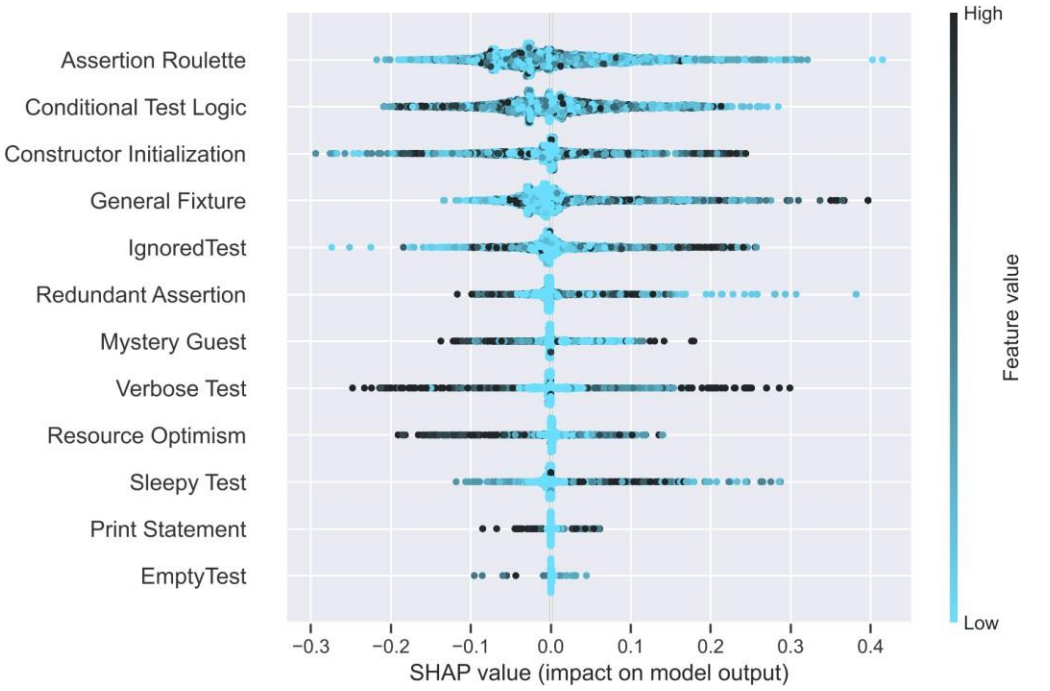


Figure 8. Feature values' impact to the correct prediction cases of the surprise bug model.

Breakage and Surprise Bug. Darker (lighter) points represent higher (lower) feature values. Meanwhile, data points in the right (left) represent higher (lower) SHAP $|\varphi i|$ values that lead to the prediction outcome of HIB (normal bugs) [34].

Figure 9 depicts the distribution of test smell features using an enhanced version of boxplot called boxen-plot [40]. Boxen-plot cuts data into more quantiles, and it makes it more feasible to display tails of large-sampled data.

Since predictive power could reflect the impact of features on the prediction outcomes [26][29]. We reveal AR, CTL and CI may have high impacts on HIB, which would be analyzed and discussed further in case studies.

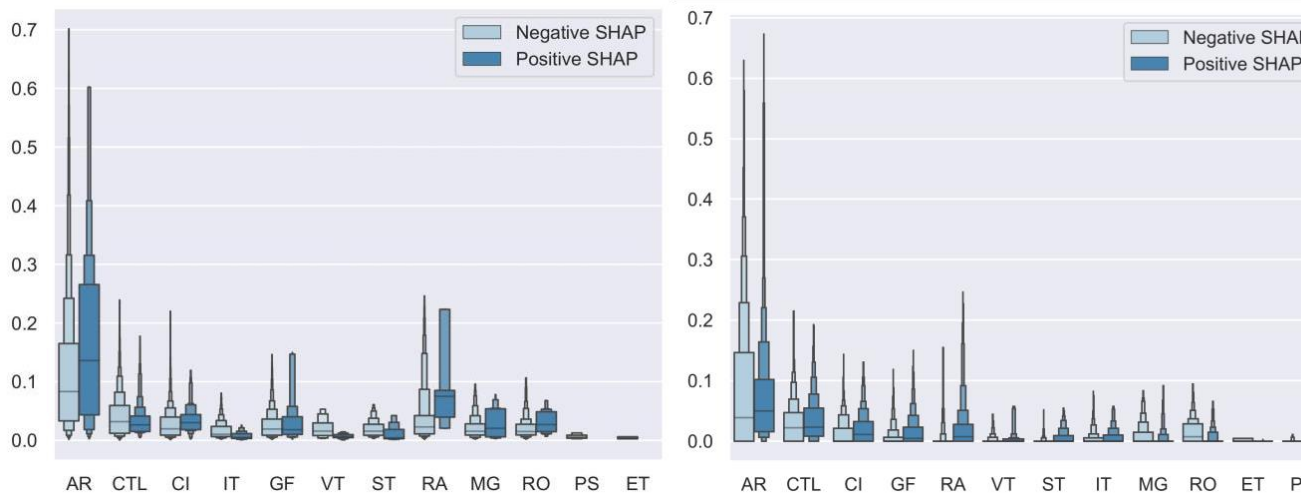


Figure 9. Distribution of the features. (a) breakage bugs. (b) surprise bugs.

Description:
This hampers restart of services by example the overhauled JMX in CAMEL-1933.When a service was restarted 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());
}
```

Figure 10 Example of AR for breakage and surprise bugs.

The AR feature is top-ranked in terms of predictive power. In order to verify whether our model works in practical scenarios, we exploit case studies by manually investigating data related to breakage and surprise bugs in the project, i.e., we intend to figure out whether the AR smell is really connected with the description provided by the developer in the bug report. For example, the bolded statement in Figure 10 shows a bug caused by multiple confusing assertion in test code, which indicates that AR is indeed one of the causes of HIB.

In terms of CTL and CI, we also find that there exist other cases in the dataset that can support our findings. Due to the limitation of space, we cannot present all details of case studies. However, we provide an online appendix[1] to demonstrate the bug reports related to CTL and CI . Therefore, according to both case study and the XAI algorithm, we can prove that test smells is related to certain types of HIBs.

Despite the trend that most test smell features have a significant impact on predicting HIB for breakage and surprise bugs, there still exists an exception. The CTL column in Figure 9 shows the opposite trend when predicting Breakage bugs, and it is difficult to explain why less occurrence of certain smells are related to bugs with less impact. However, we still have two inferences for these circumstances. First, these smells may be related to other HIBs. Second, they may be only related to trivial bugs. More empirical studies should be conducted to further investigate the interaction between these test smells and HIBs.

Practically, we recommend that developers should write better tests to avoid the occurrence of HIBs, especially to avoid AR, CTL, CI in implementation.

Finding 3. Our prediction results indicate that (1) complex and inaccessible codes are more likely to trigger HIBs, (2) AR, CTL, CI and other test smells are strongly related with the appearance of HIB. In order to reduce the emergence of HIB, we suggest that developers should write better tests to avoid the occurrence of test smells, especially AR, CTL, and CI.

VI. Threats To Validity

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

A. Construct Validity

The major threat to construct validity is the reliability of our datasets. We combine 2 sources of information, i.e., test smell detection results and bug report dataset.

For test smells, we exploit a detection tool called TsDetect. The author designed this tool according to the definition of different types of problems in test code, and they conducted manual verification to prove that the tool is correct and effective. 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.

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.

B. 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., Cliff's Delta effect size and 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 [41], these solutions may be imperfect.

C. External Validity

The programming specifications of different projects, the coding styles of different developers, and the diversity of different software projects are all inevitable threats to external validity. This problem already hinders the cross-project prediction performance, and it should be addressed in future work. However, the cross-project prediction of our model has already been improved over the compared baseline model.

VII. Conclusion

This paper investigates whether it is possible to predict the occurrence of HIB by using test smell occurrence of textually similar bug reports. In addition, it analyzes the features' predictive power and the relationship between the proposed features and the prediction results using statistical approaches and case studies. We exploit a tool called TsDetect to detect test smells, and we generate a word frequency vector similarity matrix based on the text information in the bug reports. Then, we select similar bug reports based on the test smell data to construct the dataset. Afterwards, we assess the predictive power of all the proposed features. Finally, we evaluate the correlation and significance of the difference in distributions of the features that lead to the occurrence of HIB.

Result shows that our model outperforms the state-of-the-art baseline model in within-project and time-based prediction, and it achieves mean F-Measures of 89%, 77% and 74%, 61% respectively for 2 types of HIBs concerned. In cross-project prediction, our model achieves worse performance, but it still outperforms the compared baseline.

Additionally, every proposed feature has some contribution to the model, while 2 test smell features are the strongest predictors among them. Unexpectedly, CTL occurrence indicated less HIB occurrence in the Breakage bug prediction, which needs further empirical interpretation through case studies. In conclusion, we recommend developers paying attention to test code maintainability to reduce the occurrence of HIB, and AR, CTL, CI test smells should be especially avoided.

Future work includes: (1) the integration of more effort- and process-aware metrics, (2) involving other kinds of HIB such as security bugs, (3) manually construct a dataset of test related bug report to further investigate the relationship between maintainability and reliability of test code.

Acknowledgment

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," 12th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2007, pp. 34-43, doi:10.1145/1321631.1321639.

[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 (JCST), vol. 29, 2014, pp. 713–723, doi: 10.1007/s11390-014-1461-6.

[3] M. R. Karim, A. Ihara, X. Yang, H. Iida, and K. Matsumoto, "Understanding key features of high-impact bug reports," 8th International Workshop on Empirical Software Engineering in Practice (IWESEP), 2017, pp. 53-58, doi: 10.1109/IWESEP.2017.17.

[4] H. Li, G. Gao, R. Chen, X. Ge, and S. Guo, "The influence ranking for testers in bug tracking systems," International Journal of Software Engineering and Knowledge Engineering (IJSEKE), vol. 29, no.1, 2019, pp. 93-113, doi: 10.1142/S0218194019500050.

[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 (TSE), vol. 46, no. 8, pp. 836–862, 2020.

[6] M. Ohira et al., "A dataset of high impact bugs: Manually-classified issue reports," 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), 2015, pp. 518-521, doi: 10.1109/MSR.2015.78.

[7] X. L. Yang, D. Lo, Q. Huang, X. Xia and J. L. Sun, "Automated identification of high impact bug reports leveraging imbalanced learning strategies," 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), 2016, pp. 227-232, doi: 10.1109/COMPSAC.2016.67.

[8] X. L. Yang, D. Lo, X. Xia, Q. Huang, and J. L. Sun, "High impact bug report identification with imbalanced learning strategies," Journal of Computer Science and Technology (JCST), vol. 32, no.1, 2017, pp. 181-198, doi: 10.1007/s11390-017-1713-3.

[9] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," 2010 7th IEEE Working Conference on Mining Software Repositories (MSR), 2010, pp. 11-20, doi: 10.1109/MSR.2010.5463340.

[10] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High impact defects: A study of breakage and surprise defects," 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE), 2011, pp. 300-310, doi: 10.1145/2025113.2025155.

[11] H. Li, Y. Qu, S. Guo, G. Gao, R. Chen, G. Chen, and S. Mobayen. "Surprise bug report prediction utilizing optimized integration with imbalanced learning strategy," Complexity, vol. 2020, p.8509821, 2020, doi:10.1155/2020/8509821

[12] D.J. 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 (EMSE), vol. 26, no. 4, p. 100, 2021. doi: 10.1007/s10664-021-09969-1

[13] J. Han, M. Kamber, "Data Mining: Concepts and Techniques," Morgan Kaufmann, 2006, doi: 10.1016/C2009-0-61819-5.

[14] H. He and E. A. Garcia, "Learning from Imbalanced Data," in IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 21, no. 9, pp. 1263-1284, 2009, doi: 10.1109/TKDE.2008.239.

[15] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," Journal of Artificial Intelligence Research (JAIR), vol. 16, pp. 321-357, 2002, doi: 10. 1613/jair. 953.

[16] C. S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in 31st International Conference on Neural Information Processing Systems (NIPS), pp. 4768 – 4777, 2017.

[17] T. H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," 11th Working Conference on Mining Software Repositories (MSR), 2014, pp. 82-91, doi: 10.1145/2597073.2597108.

[18] H. Valdivia-Garcia, E. Shihab, and M. Nagappan, "Characterizing and predicting blocking bugs in open source projects," Journal of Systems and Software (JSS), vol. 143, 2018, pp. 44-58, doi: 10.1016/j.jss.2018.03.053

[19] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, pp. 237-246, doi: 10.1109/MSR.2013.6624035.

[20] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," 2012 19th Working Conference on Reverse Engineering (WCRE), 2012, pp. 205-214, doi: 10.1109/WCRE.2012.30.

[21] G. K. Rajbahadur, S. Wang, G. Ansaldi, Y. Kamei and A. E. Hassan, "The impact of feature importance methods on the interpretation of defect classifiers," IEEE Transactions on Software Engineering (TSE), 2021, p. Early Access, doi: 10.1109/TSE.2021.3056941.

[22] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP), 2001, pp. 92-95

[23] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," ESEC/FSE 2020 –28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2020, pp. 1650-1654, doi: 10.1145/3368089.3417921

[24] M. A. Hall. "Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning," 17th International Conference on Machine Learning (ICML), 2000, pp. 359-366, doi: 10.5555/645529.657793.

[25] Tian, Y., Ali, N., Lo, D.et al. "On the unreliability of bug severity data," Empirical Software Engineering (EMSE), vol.21, no.6, pp. 2298-2323, 2016, doi: 10.1007/s10664-015-9409-1

[26] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-driven code smell prioritization," IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), 2020, pp. 220-231, doi: 10.1145/3379597.3387457.

[27] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," IEEE Transactions on Software Engineering (TSE), vol. 46, pp. 1200-1219, 2020, doi: 10.1109/TSE.2018.2876537.

[28] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based code smells prioritization for prefactoring," IEEE 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1-10, doi: 10.1109/ICPC.2016.7503705.

[29] F. Palomba and D. A. Tamburri, "Predicting the emergence of community smells using socio-technical metrics: A machine-learning approach," Journal of Systems and Software (JSS), vol. 171, p. 110847, 2021, doi:10.1016/J.JSS.2020.110847.

[30] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," IEEE Transactions on Software Engineering (TSE), vol. 43, 2017, pp. 1-18, doi: 10.1109/TSE.2016.2584050.

[31] F. Pedregosa et al.,"Scikit-learn: Machine learning in Python," Journal of Machine Learning Research (JMLR), vol. 12, 2011, pp. 2825-2830.

[32] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," IEEE Transactions on Software Engineering (TSE), vol. 45, no. 2, 2019, pp. 194-218, doi: 10.1109/TSE.2017.2770122.

[33] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," Pattern Recognition, vol. 30, no. 7, 1997, pp. 1145-1159, doi: 10.1016/S0031-3203(96)00142-2.

[34] G. Esteves, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani. "Understanding machine learning software defect predictions," Automated Software Engineering (ASEJ), vol. 27, no. 3, 2020, pp. 369–392, doi: 10.1007/s10515-020-00277-4.

[35] G. E. Santos, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani. "Predicting software defects with explainable machine learning," 19th Brazilian Symposium on Software Quality (SBQS), 2020, pp.1-10, doi: 10.1145/3439961.3439979.

[36] Shapley L S. A value for n-person games. Contributions to the Theory of Games, vol. 2, no. 28, 1953, pp. 307-317.

[37] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee. "From local explanations to global understanding with explainable AI for trees," Nature Machine Intelligence, vol. 2, no. 1, 2020, pp. 56-67.

[38] Scott A J, Knott M. A cluster analysis method for grouping means in the analysis of variance. Biometrics, vol. 30, no. 3, 1974, pp. 507-512.

[39] X. Yang, H. Yu, G. Fan, K. Yang. "DEJIT: A differential evolution algorithm for effort-aware just-intime software defect prediction," International Journal of Software Engineering and Knowledge Engineering (IJSEKE), vol. 31, no. 3, 2021, pp. 289-310.

[40] H. Hofmann, H. Wickham, and K. Kafadar, "Letter-value plots: Boxplots for large data," Journal of Computational and Graphical

Statistics (JCGS), vol. 26, no. 3, 2017, pp. 469–477, doi: 10.1080/10618600.2017.1305277.

[41] Jiarpakdee J, Tantithamthavorn C, Grundy J. Practitioners perceptions of the goals and visual explanations of defect prediction models. In Proc. IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 432–443. doi: 10.1109/MSR52588.2021.00055