Code-line-level Bugginess Identification: How Far have We Come, and How Far have We Yet to Go?

ZHAOQIANG GUO, State Key Laboratory for Novel Software Technology, Nanjing University and Huawei Technologies Co., Ltd
SHIRAN LIU, XUTONG LIU, WEI LAI, and MINGLIANG MA, State Key Laboratory for Novel Software Technology, Nanjing University
XU ZHANG, Beijing Bytedance Network Technology Co., Ltd
CHAO NI, Zhejiang University
YIBIAO YANG, YANHUI LI, and LIN CHEN, State Key Laboratory for Novel Software Technology, Nanjing University
GUOQIANG ZHOU, Nanjing University of Posts and Telecommunications
YUMING ZHOU, State Key Laboratory for Novel Software Technology, Nanjing University

Background. Code-line-level bugginess identification (CLBI) is a vital technique that can facilitate developers to identify buggy lines without expending a large amount of human effort. Most of the existing studies tried to mine the characteristics of source codes to train supervised prediction models, which have been reported to be able to discriminate buggy code lines amongst others in a target program. **Problem.** However, several simple and clear code characteristics, such as complexity of code lines, have been disregarded in the current literature. Such characteristics can be acquired and applied easily in an unsupervised way to conduct more accurate CLBI, which also can decrease the application cost of existing CLBI approaches by a large margin. **Objective.** We aim at investigating the status quo in the field of CLBI from the perspective of (1) how far we have really come in the literature, and (2) how far we have yet to go in the industry, by analyzing the performance of state-of-the-art (SOTA) CLBI approaches and tools, respectively. **Method.** We propose a simple heuristic baseline solution GLANCE (aiminG at controL- ANd ComplEx-statements) with three implementations (i.e., GLANCE-MD, GLANCE-EA, and GLANCE-LR). GLANCE is a two-stage CLBI framework: first, use a simple model to predict the potentially defective files; second, leverage simple code characteristics to identify buggy code lines in the predicted defective files. We use GLANCE as the baseline to investigate the effectiveness of the SOTA CLBI approaches, including natural language processing (NLP) based, model interpretation techniques (MIT) based, and popular static analysis tools (SAT). **Result.** Based on

This work is partially supported by the National Natural Science Foundation of China (62172205, 62172202, 62272221, 62072194) and by the National Program on Key Basic Research Project (2020YFA0713600).
Authors' addresses: Z. Guo, State Key Laboratory for Novel Software Technology, Nanjing University, Jiangsu Province, China and Huawei Technologies Co., Ltd, Hangzhou, China; emails: gzq@smail.nju.edu.cn, guozhaoqiang@huawei.com; S. Liu, X. Liu, W. Lai, M. Ma, Y. Yang (corresponding author), Y. Li, L. Chen, and Y. Zhou (corresponding author), State Key Laboratory for Novel Software Technology, Nanjing University, Jiangsu Province, China; emails: {shiranliu,xryu,DZ20330011,dg21330022}@smail.nju.edu.cn, {yangyibiao,yanhuili,lchen,zhouyuming}@nju.edu.cn; X. Zhang, Beijing Bytedance Network Technology Co., Ltd., China; email: zhangxu.314@bytedance.com; C. Ni, Zhejiang University, Ningbo, Zhejiang Province, China; email: chaoni@zju.edu.cn; G. Zhou (corresponding author), Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu Province, China; email: zhougq@njupt.edu.cn.

1049-331X/2023/05-ART102 $15.00
https://doi.org/10.1145/3582572

19 open-source projects with 142 different releases, the experimental results show that GLANCE framework has a prediction performance comparable or even superior to the existing SOTA CLBI approaches and tools in terms of 8 different performance indicators. **Conclusion.** The results caution us that, if the identification performance is the goal, the real progress in CLBI is not being achieved as it might have been envisaged in the literature and there is still a long way to go to really promote the effectiveness of static analysis tools in industry. In addition, we suggest using GLANCE as a baseline in future studies to demonstrate the usefulness of any newly proposed CLBI approach.

CCS Concepts: • **Software and its engineering → Software maintenance tools;**

Additional Key Words and Phrases: Code line, bugginess, defect prediction, quality assurance, static analysis tool

ACM Reference format:
Zhaoqiang Guo, Shiran Liu, Xutong Liu, Wei Lai, Mingliang Ma, Xu Zhang, Chao Ni, Yibiao Yang, Yanhui Li, Lin Chen, Guoqiang Zhou, and Yuming Zhou. 2023. Code-line-level Bugginess Identification: How Far have We Come, and How Far have We Yet to Go?. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 102 (May 2023), 55 pages.
https://doi.org/10.1145/3582572

1 INTRODUCTION

Bugginess identification aims at distinguishing the potential buggy source codes in a target software product, which has been one hot research topic in the domain of **software quality assurance** (**SQA**) [41, 57]. As stated in [3], SQA activities usually spend on 23% of total effort in the whole process of software development. Meanwhile, most of the SQA effort concentrate on identifying the real defective parts of the source codes. To alleviate the review pressure of SQA teams and ensure effective SQA activities, researchers hence have proposed various automatic bugginess identification techniques [14, 23, 30–32, 55, 56, 77, 79, 81] to prioritize SQA effort [52]. However, most of these studies identify potentially buggy portion of source codes in a coarse-granularity (e.g., file [77], method [14], and change [81]). Although these approaches provide a great convenience for code review, there is still a vital problem worthy of attention. That is, the size of potentially buggy codes recommended by these approaches are usually large so that practitioners still need to spend a lot of effort to check the real buggy locations. According to Wattanakriengkrai et al.'s study [73], there are less than 3% of code lines in a defective source file that contain bugs.

Fortunately, researchers [27, 60, 73] have attempted to conduct studies on **code-line-level bugginess identification** (**CLBI** for short), a fine granularity, to generate more accurate and meticulous identification results. Overall, the current CLBI studies in the literature can be divided into three main categories. The first category uses automatic Static Analysis Tool to find potentially buggy lines (SATs for short). A SAT (e.g., PMD [66]) will first scan a target software product (e.g., source code files) and detect whether codes match predefined warning (or bug) patterns. Then, all code lines captured by any warning patterns will be reported as potentially buggy lines. The second category uses **Natural Language Processing techniques** (**NLPs** for short) to identify potentially buggy lines. This kind of approaches [68] usually constructs a statistical language model (e.g., *n-gram*) to measure the "*naturalness*" (i.e., the suspicious score) of source codes. The intuition behind this is that, the less "naturalness" the code, the more likely it is to contain defects. The third category utilizes Model Interpretation Techniques to predict buggy lines (MITs for short). This kind of approaches implements code-line-level prediction by interpreting the contributions of tokens on the corresponding coarse-grained (i.e., file-level) bugginess identification model. As a result, the lines contain risky tokens are considered as potentially buggy lines. Recently, Wattanakriengkrai

et al., [73] proposed a **state-of-the-art** (**SOTA**) MIT based approach LineDP that leverages a model-agnostic technique LIME [62] as the core interpreter to identify buggy code lines. All the above approaches have been reported to be able to discriminate buggy code lines to a certain extent.

Yet, we find that there are two main problems that the existing SOTA approaches cannot deal with well. On the one hand, SAT has a very low accuracy. The reason is that the patterns defined by a SAT are not always strongly related to bugs, some of which just point out the nonstandard code writing styles. Therefore, a SAT usually reports a large number of false positives. On the other hand, NLP and MIT based approaches need to maintain a large amount of model data or train a supervised model, which can consume a large amount of computational cost and are limited to the quality of the training data set. In particular, for MIT, it is time-consuming to interpret the risk of tokens, which may lead to a poor user experience. In practice, the above problems may hinder the application of the existing SOTA CLBI approaches.

In this study, we are dedicated to explore an accurate and efficient CLBI solution to alleviate the above-mentioned two problems. To this end, we first manually analyze buggy code lines from the dataset collected by Wattanakriengkrai et al [73]. We have the following two observations: (1) buggy code lines are usually have a high complexity measured by the code size and the number of function calls. As stated by Zhou et al., [86], the larger the size of a source code file, the more likely it contains bugs. This finding is supported by many other related studies [12, 35, 74]. In addition, recent studies [76, 80] pointed out that complex function calls could increase the risk of introducing defects into a program; and (2) many buggy code lines are correlated with control structures (e.g., `for`, `if`, and `while`). As reported in [70, 71], control structures play important roles in bug detection. To the best of our knowledge, the existing CLBI techniques have disregarded these simple and clear code characteristics. Such characteristics can be acquired and applied easily in an unsupervised way to facilitate the CLBI task, whose application cost is very low compared with the existing SOTA approaches by a large margin. Based on the above observations, we propose a simple heuristic baseline CLBI solution **aiminG at controL- ANd ComplEx-statements** (**GLANCE**). This name informs that buggy lines can be predicted by taking a quick glance at the control structure and complexity metrics measured in code lines. Specifically, GLANCE is a two-stage CLBI framework. At the first stage, a file-level classifier is built to predict the defective files in a target project. At the second stage, we measure the buggy-proneness of a code line by utilizing the above-mentioned code characteristics and then output a recommended rank of all code lines from each defective file predicted in the first stage. Based on the selection of different file-level classifiers, we analyze and compare three GLANCE implementations: GLANCE-MD (Classifier: ManualDown [87]), GLANCE-EA (Classifier: Effort Aware ManualDown), and GLANCE-LR (Classifier: Logistic Regression), respectively.

Based on the simple baseline solution GLANCE, we attempt to investigate the status quo in the field of CLBI. More specifically, we want to know: (1) how far have we really come by comparing the existing SOTA NLPs and MITs in the literature with GLANCE? and (2) how far have we yet to go by comparing the popular **static analysis tools** (**SATs**) in industry and GLANCE? To obtain reliable comparison results, we first collect a large-scale code-line-level defect dataset following and improving the existing data collection process [60, 64, 73]. After that, we run and evaluate each approach under cross-release experimental scenario, which means that the data of previous release is used to construct a model (only for supervised) and the data of next release is used to examine the effectiveness of the model. Our results show: (1) GLANCE exhibits a competitive classification and ranking performance compared with various existing CLBI approaches; (2) compared with two popular SATs (PMD and CheckStyle), the simple GLANCE exhibits great advantages in terms of eight indicators. The above results indicate that, the current research progress is not satisfactory in the literature in the field of CLBI and there is still a long way to go to really

promote the effectiveness of SATs in the industry. Based on the above results, it would be better to apply GLANCE to predict buggy code lines in practice. Note that, although GLANCE (especially for GLANCE-LR) performs better than SOTA approaches, it still suffers from higher false alarms, which urges researchers pay more attention on the CLBI task. If possible, it is suggested to use GLANCE as an easy-to-implement baseline in future studies to demonstrate the usefulness of any newly proposed CLBI approach.

In summary, we make the following contributions in this article:

(1) We collect a large-scale code-line-level defect dataset, including 19 popular ASF projects with 142 distinct releases, which allows for the other researchers to conduct more comprehensive studies in this field. To ensure that other researchers can access it, we make the complete dataset and the corresponding collection scripts publicly available [8].
(2) We propose a simple baseline CLBI solution GLANCE with three specific implementations (GLANCE-MD, GLANCE-EA, and GLANCE-LR, respectively) to predict buggy lines. Note that, GLANCE is very easy to be applied to a real-world project.
(3) We conduct a comprehensive experimental comparison between the SOTA CLBI techniques and GLANCE to investigate the status quo in CLBI. Surprisingly, we find that GLANCE shows a great competitiveness compared with the existing approaches and tools.
(4) We conduct additional discussions to give a more comprehensive understanding on the characteristics of GLANCE, which can facilitate researchers and practitioners to utilize and improve GLANCE in the future.
(5) We have made the entire replication kit (all code scripts and additional experimental results) of this study available in [11], which can be conveniently reproduced in future CLBI studies.

The rest of this article is organized as follows. Section 2 introduces the background of code-line-level bugginess identification. Section 3 presents the motivating analysis and Section 4 presents the process of constructing a simple baseline approach. Later, Section 5 describes the experimental setup and Section 6 reports the experimental results. After that, we discuss additional results in Section 7. In Section 8, we summarize the implications inspired by this study. Section 9 analyzes the threats to the validity of our study. Section 10 introduces the related works. Finally, Section 11 concludes the article and outlines the direction for our future work.

2 BACKGROUND ON CODE-LINE-LEVEL BUGGINESS IDENTIFICATION

This section sheds a light on the status quo in **code-line-level bugginess identification** (**CLBI**). First, we review the representative SOTA techniques in the field. Then, we highlight the challenges that still need to be tackled.

2.1 State-of-the-Art (SOTA) Techniques

In the literature, researchers have leveraged various techniques to predict buggy lines. Specifically, the existing SOTA CLBI techniques can be classified into the following three categories.

2.1.1 Static Analysis Tool-based Approach (SATs). Automatic **static analysis tools** (**SATs**) play an important role in the workflow of **software quality assurance** (**SQA**) activities in both usefulness and convenience [2, 16, 24, 25, 29, 65, 69]. First, plenty of underlying issues in software artifacts, such as coding defects [26], vulnerabilities [84], and code style violations [66], can be identified by SATs. Therefore, more reasonable SQA resources, including human cost and test suites, can be assigned to improve software quality based on the detection results of SATs. Second, SATs provide a simple and convenient way to detect software issues without dynamically executing a target program. Instead, each SAT retrieves a set of pre-defined common bug patterns

(summarized by software experts) in the target program and then reports the information of all problematic code matched by the bug patterns. Specifically, most of them are designed as light-weight tools, which can be used by either independent command line tools or built-in components of popular IDEs such as Eclipse and IntelliJ IDEA.[1]

In the software development community, many popular SATs, such as FindBugs,[2] PMD,[3] and CheckStyle,[4] have been widely used to detect potential bugs [67] [5]. Based on the actual development experience of software analysis experts, the tool providers of different SATs manually predefined a set of distinct bug patterns to indicate specific bugs. For example, there are 9 different bug categories (e.g., `Correctness`) with a total of 424 patterns in the latest version of FindBugs while 8 different bug categories with a total of 310 patterns in PMD. Note that, different SATs may concern the same vital bug categories. The categories of `Performance` and `Security`, for instance, have been provided in both FindBugs and PMD.

Recently, Wattanakriengkrai et al., [73] used SATs to conduct a CLBI study. Given a software project, they used SATs to scan the code files and detect the distribution of predefined bug patterns in the target software artifact (i.e., source code files). When a bug pattern was found in the artifact, SATs reported a warning line instance that contained the detailed bug information (e.g., buggy file name, buggy lines, and bug priority). After collecting all warning line instances, they ranked these lines according to the importance of their corresponding bug priority. Finally, the ranked buggy lines were recommended to developers for manual inspection.

2.1.2 Natural Language Processing-based Approach (NLPs). Natural language processing technique has been widely applied for transforming natural language texts into computable values in software engineering [68]. The language model is a typical NLP technique that can be used to estimate the occurrence probability of a token or sentence based on a given training corpus [22] [9]. The n-gram language model is the most popular one that has been widely used in the research field of software engineering (e.g., self-admitted technical debt identification [72] and API recommendation [61]). Given a token sequence $S =< t_1, t_2, \ldots, t_n >$ of length n, one n-gram model estimates its occurrence probability based on the product of a series of conditional probabilities (see the first Formula). Here, $P(t_i \mid t_1, \ldots, t_{i-1})$ denotes the probability of token $t_i$following the prefix $< t_1, \ldots, t_{i-1} >$ (h for short). Indeed, there are many possible prefixes so that the above probability is impractical to estimate. To reduce the difficulty of calculation, researchers usually estimate the probability of $P(t_i \mid t_1, \ldots, t_{i-1})$ by making the Markov assumption (i.e., the conditional probability of a token t_i only depends on the preceding n-1 tokens) (see the second Formula).

$$P(S) = P(t_1) \prod_{i=2}^{N} P(t_i \mid t_1, \ldots, t_{i-1}) . \tag{1}$$

$$P(t_i \mid h) = P(t_i \mid t_1, \ldots, t_{i-1}) \approx P(t_i \mid t_{i-n+1}, \ldots, t_{i-1}) . \tag{2}$$

The intuition behind using n-gram model to predict buggy code is based on the "naturalness" of source code [60]. That is, *unnatural code is more likely to be incorrect.* More concretely, the unnatural code denotes the source code that rarely appears in corpora while the natural code denotes the highly repetitive ones. In this sense, Ray et al., [60] calculate the cross-entropy [22, 68] (see Formula (3)) of each code line to represent its "naturalness" and recommend the developers

[1] https://www.jetbrains.com/opensource/idea/.
[2] http://findbugs.sourceforge.net.
[3] https://pmd.github.io/.
[4] https://checkstyle.sourceforge.io/.

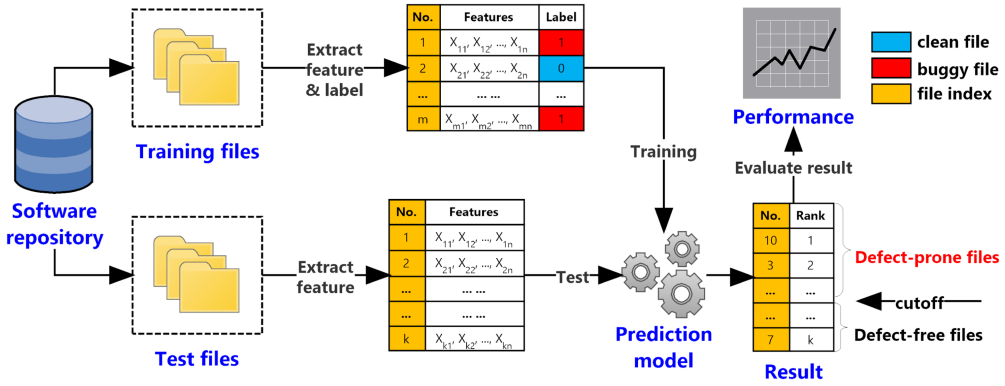

Fig. 1. General process of file-level defect prediction.

to inspect the unnatural code (with higher entropy) lines preferentially.

$$H_M(\mathrm{s}) = -\frac{1}{n}\sum_{i=1}^{n}\log P\left(t_i \mid t_{i-n+1}, \ldots, t_{i-1}\right). \tag{3}$$

2.1.3 Model Interpretation Technique-based Approach (MITs). Recently, model interpretation technique is used to predict the buggy lines by interpreting the risky code tokens in a defect prediction model [27, 73]. In the framework of MITs, the entire process of defection prediction is divided into two stages (i.e., file-level and line-level prediction). At the file-level prediction stage, a MITs will train a classifier based on the features (or metrics) extracted from a training project to predict the defective source files in the test project (follows the process shown in Figure 1) [19, 38, 42, 53, 80]. At the line-level prediction stage, for each defective file, interpret the reason why a file was predicted as defective by analyzing the risky value (e.g., coefficient) of each feature (i.e., code token) in the prediction model. A code line is deemed as buggy if any risky tokens occur in the line, otherwise, it is considered clean. For the convenience of checking code, assign each predicted buggy line a risk score with the number of risky tokens occurring in the line, and rank the lines in descending order from the most to the least risk scores. Consequently, it is suggested to check lines according to the order in the ranked list.

At present, there are mainly three kinds of interpretation strategies. For the models trained by linear-based classifiers (e.g., Linear Regression), the tokens with positive coefficient values in these models will be considered as the risky tokens. For the models trained by tree-based classifiers (e.g., Random Forest), the feature importance value (e.g., information gain) will be deemed as the risky score of each candidate token. In addition to the above two strategies, an independent third-party interpreter can also be used to make explanations for a trained classifier. Specifically, Wattanakriengkrai et al., [73] propose a state-of-the-art MITs called LineDP. They use the state-of-the-art model-agnostic technique LIME (i.e., Local Interpretable Model-agnostic Explanations) [28, 62] as the interpreter to identify the possible risky tokens based on a trained Logistic Regression model. After that, LineDP considers the top 20 tokens with the highest positive scores as the bug indicators to calculate the defect-proneness of code lines in a target source file.

2.2 Challenges in CLBI

Although the above CLBI techniques can be used to identify potentially buggy code lines, there still exist two main challenges that may hinder their usability in practice.

(1) ***C1: Low accuracy (for SATs).*** This challenge denotes the fact that most of existing CLBI techniques cannot perform a satisfactory accuracy in identifying buggy code lines, especially for SATs. SATs usually report many false positives because the patterns defined by a SAT are not always strongly related to bugs, some of which just point out the nonstandard code writing styles. For instance, the empirical study by Imtiaz et al., [25] showed that there were only 36.7% of warnings reported by SATs were regarded as true defects by developers. In this situation, CLBI approaches play a small role so that developers still need to review many code snippets.

(2) ***C2: High application cost (for NLPs and MITs).*** This challenge highlights the fact that the SOTA approaches may consume a large amount of application costs when applying them in practice. On the one hand, many training data are needed to build a reliable bugginess identification model. To this end, one needs to collect many high-quality training data. On the other hand, it is necessary to prepare enough time to build and apply a model, and enough hardware to store the model. For instance, for MITs, it is time-consuming to interpret the risk of tokens, which may lead to a poor user experience. Meanwhile, practitioners need to store a great number of *n-gram* pairs to build an accurate NLP model.

3 MOTIVATING OBSERVATIONS

Given the above challenges within the current SOTA approaches, is it possible to provide an effective solution with high accuracy and low application cost to identify buggy code lines? To answer this question, we review and analyze buggy lines in ActiveMQ project used in [73] to exploit the characteristics of bugs at the code line level. As a result, we observed that two important factors that may be closely related to buggy lines. In this section, First, we present several intuitive motivating examples to illustrate our new observations and propose two preliminary hypotheses. Then, we conduct a case study in ActiveMQ-5.0.0 to investigate and validate the universality of our hypotheses.

3.1 Code Example

In the literature, two types (control elements and code complexity metrics) of code characteristics have attracted the attention of researchers [33, 88]. First, Kapur et al., [33] pointed out that programming constructs (e.g., control elements) can serve as useful features to represent source code and hence be used to estimate the defectiveness of source code. The **control elements** (**CEs**) are the keywords (e.g., `if`, `for`, `while`) that declare an executive condition, the status (i.e., `true` or `false`) of which determines the execution flow of the program. Second, code complexity metrics (e.g., the number of tokens and the number of functions) are another most concerned defect indicators, which have been shown to correlate with defect density in several studies [4, 37, 52, 88]. The intuition behind it is that, the more complex the software entity, the more likely it is to contain defects.

Based on the above prior knowledge, we investigate two such characteristics i.e., control elements and code complexity at code-line level. Figure 2 depicts three examples of code change patches (including buggy lines and fixed lines) in the ActiveMQ project for fixing bugs AMQ-2327, AMQ-4529, and AMQ-4952, respectively. Here, the rows of source files and methods are marked as gray, the buggy (or fixed) lines in each source file are marked as red (or green), and the clean lines are shown in white. The first column denotes the line number of each statement. The second column shows the content of source code lines. In the last column, the specific CE occurring in each buggy line, the number of function calls (#Func), and the number of code tokens (#Tokens) are summarized, respectively.

From the first example (#1), we can see that the buggy lines contain a control element, i.e., the line-67 that contains a control element if is a buggy one, whose condition has been modified in

#1 AMQ-2327		activemq-broker/src/main/java/org/apache/activemq/network/ConduitBridge.java @@ -64,7 +64,7 @@ protected boolean addToAlreadyInterestedConsumers(ConsumerInfo info) {	CE
65		for (DemandSubscription ds : subscriptionMapByLocalId.values()) {	
66		DestinationFilter filter = DestinationFilter.parseFilter(ds.getLocalInfo().getDestination());	
67	-	if (!ds.getRemoteInfo().isNetworkSubscription() && filter.matches(info.getDestination())) {	if
67	+	if (canConduit(ds) && filter.matches(info.getDestination())) {	
68		LOG.debug("{} {} with ids {} matched (add interest) {}", new Object[]{	
69		configuration.getBrokerName(), info, info.getNetworkConsumerIds(), ds });	
#2 AMQ-4529		activemq-client/src/main/java/org/apache/activemq/command/MessageId.java @@ -153,7 +153,7 @@ public MessageId copy() {	#Func
153		MessageId copy = new MessageId(producerId, producerSequenceId);	1
154		copy.key = key;	0
155		copy.brokerSequenceId = brokerSequenceId;	0
156	-	copy.dataLocator = new AtomicReference<Object>(dataLocator != null ? dataLocator.get() : null);	2
156	+	copy.dataLocator = dataLocator;	
157		copy.entryLocator = entryLocator;	0
158		copy.plistLocator = plistLocator;	0
159		return copy;	0
#3 AMQ-4925		activemq-unit-tests/src/test/java/org/apache/activemq/bugs/AMQ4952Test.java @@ -296,7 +296,7 @@ protected BrokerService createProducerBroker() throws Exception	#Token
296		if (networkProps != null) {	3
297		IntrospectionSupport.setProperties(nc, networkProps);	4
298		}	0
299	-	nc.setStaticallyIncludedDestinations(Arrays.asList(new ActiveMQQueue[]{QUEUE_NAME}));	7
299	+	nc.setStaticallyIncludedDestinations(Arrays.<ActiveMQDestination>asList(new ActiveMQQueue[]{QUEUE_NAME}));	
300		}	0

Fig. 2. Examples of buggy lines (red color) and fixed lines (green color) in ActiveMQ.

its fixing version. The second and third example (#2 and #3) show that the complexity of buggy code lines is usually larger than that of clean code lines. Specifically, on the one hand, there are many functions calls in buggy lines. The buggy line (i.e., line-156) contains two function calls while other clean lines (i.e., line-153–155, 157–159) in the same code context only contain at most one function call. On the second hand, buggy lines usually contain more code tokens (i.e., individual words) than clean lines. For instance, the number of code tokens of buggy line-299 is 7, which is higher than the other four clean lines (i.e., line-296–298, and 300).

Based on the above examples, we propose the following two hypotheses:

(1) *Control elements could be used as bug indicators to signify the code-line-level bugs.*
(2) *Code complexity metrics could be used as bug indicators to signify the code-line-level bugs.*

3.2 Case Study

To obtain the quantitative results for the two proposed hypotheses, we conduct two case studies in the project ActiveMQ-5.0.0, which has been investigated in [73]. More specifically, to validate the first hypothesis, we calculate the percentages of CE lines (i.e., the line contain control elements) among buggy lines and clean lines of each defective source file, respectively, as well as the percentage of buggy CE lines among all CE lines, to investigate the relationship between buggy lines and control elements. To validate the second hypothesis, we calculate #Func (i.e., the number of function calls) and #Token (i.e., the number of code tokens) in lines of each defective source file, as well as the percentage of buggy lines among all lines, to investigate the relationship between buggy lines and code complexity.

Figure 3 reports the percentage distributions of the first case study in ActiveMQ-5.0.0. On average, 23.8% of buggy code lines in each defective source file contain control elements. This indicates that these bugs may be caused by the errors of control statements. Intuitively, it is helpful to recommend these control elements related lines to developers. According to the second boxplot in Figure 3, we can find that there are 13.8% clean lines that contain control elements on average.

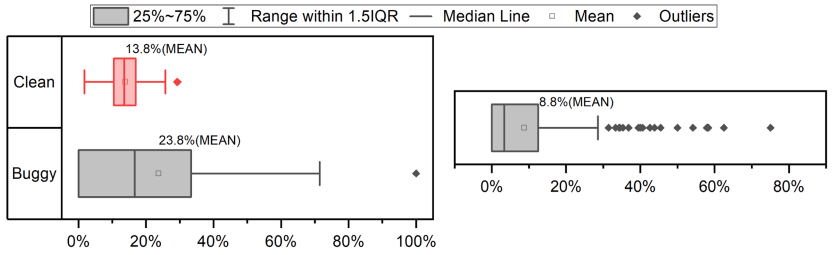

Fig. 3. The distribution of the ratios of CE lines (control elements) in all defective files of ActiveMQ-5.0.0.

Additionally, the percentages of CE lines in the majority of clean files (i.e., more than 75%) are lower than 20%. This shows that, if the control lines in the defective files are ranked before other lines, developers can find most buggy lines caused by control elements in the case of spending only 20% effort (i.e., LOC). Moreover, on average, 8.8% of CE lines contains bugs. The above results show that control elements can be regarded as a bug indicators. It is possible to adjust the position of a control line in the ranking list to improve the effectiveness of recommendations.

Figure 4 reports the distributions of the second case study in ActiveMQ-5.0.0. in terms of #Func and #Token, respectively. Considering the number of function calls, as can be seen that, on average, there are 0.37 function calls in each clean line while 1.21 function calls in each buggy line. In addition, the number of function calls of more than 75% buggy lines are higher than that of all clean lines. This means that, the number of function calls may be an effective indicator that can be used to distinguish the buggy lines. According to the number of code tokens, we can also find that the buggy lines contain more tokens (i.e., 5.09) than clean lines (i.e., 3.29) on average. Moreover, on average, 14% and 8.2% of code lines contain bugs when checking the code lines whose corresponding #Func >=1 and #Token >=3, respectively. In common sense, the more the number of function calls and the more the number of code tokens, the more complicated a code line. Finally, we can conclude that the buggy lines are more complex than clean lines according to the Figure 4.

Inspired by the examples and the results of case studies, we realize that it may have a good ability to rank code lines by leveraging the two important factors: control elements and code complexity metrics. As such, the following question naturally raises: how to construct a simple yet effective prediction model to provide a better ranking list for the lines in a predicted defective file? We answer the question in the next section.

4 GLANCE: AIMING AT CONTROL- AND COMPLEX-STATEMENTS

In this section, we introduce a simple baseline approach GLANCE to rank the potentially buggy lines. First, we present the overall framework of GLANCE. Then, we elaborate on the technical details, including predicting defective files and identifying buggy lines.

4.1 Approach Overview

Figure 5 presents an overview of GLANCE. On the whole, the application of GLANCE to a set of test files in practice involves the following two stages: file-level identification and line-level identification. At the former stage (Figure 5, Left), we leverage a specific file-level classifier to identify defective files so that a large number of clean lines in bug free source files can be eliminated directly. At the latter stage (Figure 5, Right), we first extract two types of code-line-level code metrics (i.e., control elements and code complexity) from the predicted defective files. Then, we compute the defect-proneness score of each predicted defective file based on the above

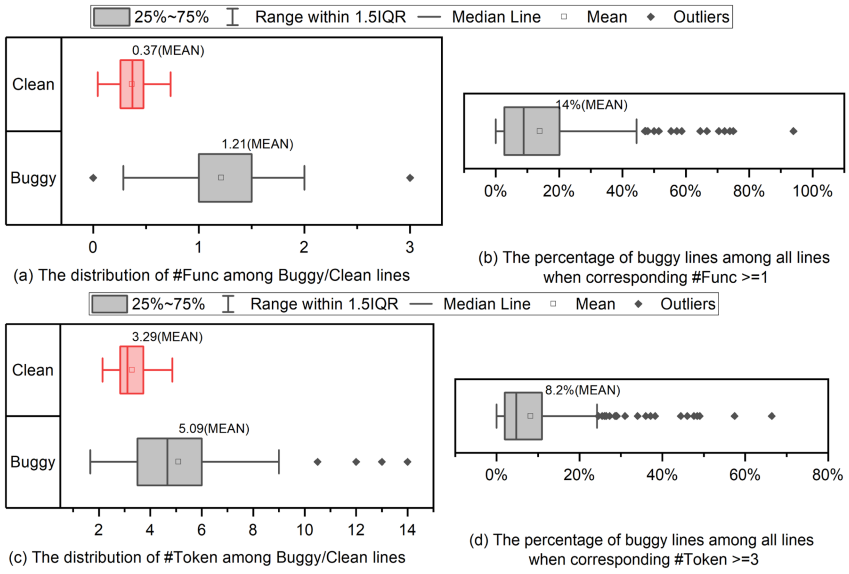

Fig. 4. The distribution of the number of function calls and code tokens in all defective files of ActiveMQ-5.0.0.

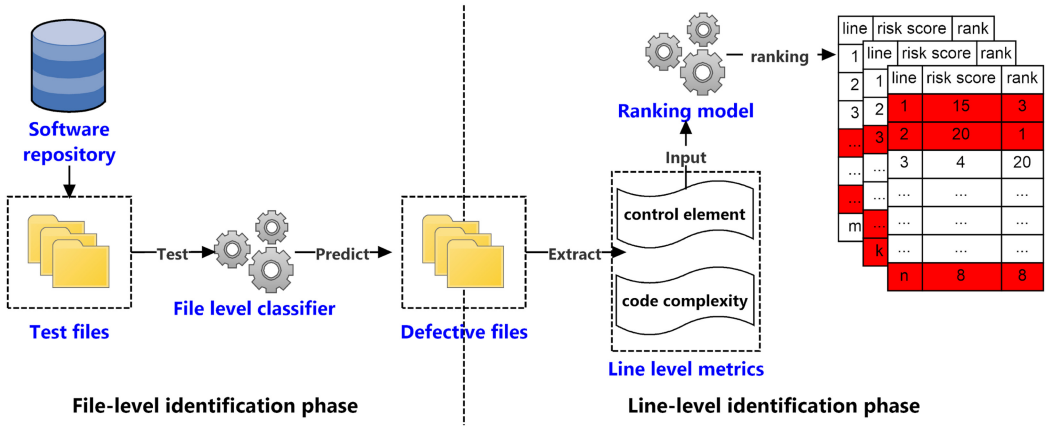

Fig. 5. The workflow of GLANCE.

metrics and input the score into a ranking model. Finally, GLANCE will output both classification results and ranking results. Note that, as mentioned in Section 2.1.3, the latest LineDP [73] ranks buggy lines based on LIME scores from the explanations for the file-level classifier. Comparatively, the novelty of GLANCE is that it uses a simpler way to evaluate the suspicious score of each code line.

4.2 Predicting Defective Files

At the first stage, the main purpose of GLANCE is to eliminate the clean lines in non-defective files of the test set so that a large proportion of effort can be saved directly. Meanwhile, to avoid missing defect codes as much as possible, the file-level classifier of GLANCE should maintain a

high recall rate. To exploit an effective file-level classification performance, we use the following three file-level classifiers:

- ***ManualDown (MD for short).*** In a recent examination of cross-project defect prediction (CPDP for short) [87], the simple module size model (i.e., ManualDown) exhibits very competitive or even superior prediction performances compared with the existing CPDP models. ManualDown is an easy-to-use unsupervised prediction model, which only considers the simple module size metric (i.e., SLOC, source line of code) as the defect-proneness indicator of the source file in the target program. The intuition is that larger files tend to have more defects [47]. More specifically, for each source file m in the target project, ManualDown first predicts its risky score as SLOC(m). After that, all source files are ranked in descending order from the most to the least risky scores. Finally, the top x% number of files in the rank list will be classified as defect-proneness. According to Zhou et al., [87], ManualDown can perform well when the threshold value is set to 50 and hence the same value is used in this study. As can be seen, ManualDown has great advantages in building cost, application cost, and scalability [87], which are important factors that need to be considered when building a prediction model, as argued by Monden et al., [48].
- ***Effort Aware ManualDown (EA for short).*** Although ManualDown performs well in coarse-granularity (e.g., file) defect prediction, the human effort for reviewing the defect-prone files predicted by ManualDown is very large since half of all files with larger size (i.e., SLOC) are predicted to be defective. In the context of CLBI, ManualDown cannot satisfy the requirement of saving reviewing effort because developers need to check many obviously clean lines within large-scale clean files. To reduce the effort of SQA activities, we design a revised model called Effort Aware ManualDown (EA) to predict defective files based on a preset effort threshold. Instead of counting the top x% files, the files occupying the top x% of review effort (i.e., SLOC) in the rank list will be classified as defect-prone. As shown in Algorithm 1, EA calculates the accumulative review effort of each file and predicts the file as defective in turn based on the ranking order of all files until half of all effort is cumulated. Based on the findings of Zhou et al., [87], 50% is a commonly used and effective default value and hence it is also used as the effort threshold of EA. Compared with MD, fewer files will be predicted as defective by EA, so it will consume less developers' effort. For instance, suppose there are six files with a total effort of 24 in the rank list, and the effort of each file is 9, 5, 4, 3, 2, and 1 in turn. MD will predict the top three files as defective, which needs to spend an effort of 18 (9+5+4). Meanwhile, under the effort threshold = 0.5, EA will predict the top files as defective until the accumulative effort has exceeded 12 (24*0.5). As a result, EA only needs to spend an effort of 14 (9+5>12), which is small than that of MD (i.e., 18).
- ***Logistic Regression (LR for short).*** In addition to the above unsupervised classifiers, the supervised classifiers built on a training set often have a better prediction performance. According to the latest study [73], Logistic Regression is proved to be a simple yet effective supervised classifier that can be used easily to identify buggy lines in the bugginess identification field. Based on this cognition, we also include LR as a candidate file-level classifier for GLANCE. Specifically, for each file, we first discard the non-alphanumeric characters (e.g., "=", ";") from the entire text because these characters are of no help to predict defects. After that, the code tokens are extracted from each file. Note that no extra data cleaning process (i.e., lowercase, stemming, and lemmatization) will be performed so that the original form of each token can be preserved. The reason for this is that, the studied projects are written by Java, in which the tokens are case-sensitive so that the real information can be destroyed if applying the above data clean process [73]. As a result, each file will be represented as a

Table 1. The Frequently Occurred Characteristics of Buggy Code Statements

Category	Type	Description
Control line	Branch statements	A code line containing one of branch control elements {**if, else, switch**}
	Loop statements	A code line containing one of loop control elements {**for, while, do**}
	Jump statements	A code line containing one of jump control elements {**break, continue, return**}
Complex line	Call statements	A code line containing many function calls
	Long statements	A code line containing many code tokens

ALGORITHM 1: Pseudocode for $EA(M, k)$ // Predicting the label (buggy or clean) of a source file

Require: **M** is the set of source files in the target project; **k** is a percentage threshold of effort, k ∈ [0,100%], k=50% by default
Ensure: M_b is the set of predicted buggy files
  $M_b \leftarrow \emptyset$ // Set of buggy files predicted by EA
  $M_{dos} \leftarrow$ sort M in descorder of SLOC // Same as ManualDown
  $E_{all} \leftarrow$ all effort of M, $E_{acc} \leftarrow 0$ // All effort and accumulative effort
  for $m \in M_{dos}$ **do**
    if $E_{acc} < k \times E_{all}$ **then**
      $M_b \leftarrow M_b \cup m$
      $E_{acc} \leftarrow E_{acc} + E_m$ // E_m, the effort of m
    else
      break // Effort is exhausted
    end if
  end for

feature vector with the form of a **bag-of-word** (**BOW**). Finally, a LR model will be trained based on the feature vectors extracted from the source files.

4.3 Identifying Buggy Lines

Based on the observations found in Section 3, we conjecture that there are two kinds of code lines that are most likely to contain defects (shown in Table 1). The first is the control line that contains one of the control elements at least. Note that the control elements can be further divided into three categories (i.e., branch control, loop control, and jump control) according to their effects. The second is the complex line that has a complex structure. These lines can be measured in two dimensions: function calls and code tokens. Our purpose is to improve the defect-proneness scores of the two kinds of lines so that more buggy lines can be ranked before the clean ones. To this end, we design the following discriminative line-level metrics to identify the most suspicious lines that could contain defects.

Control Elements (CE). CE is a binary metric that measures the weights for control lines and other lines, respectively. For a code line l, it will be assigned 1 if the code line contains one of control statements shown in Table 1. The intuition behind designing CE is that, the line containing control elements is more likely to be defect-prone. Indeed, similar binary metrics - like Crash Component (CC for short) - are used in prior study [6, 75].

$$CE(l) = \begin{cases} 1, & \text{if } l \text{ contains control elements} \\ 0, & \text{otherwise} \end{cases}. \tag{4}$$

Number of Tokens (NT). NT measures the complexity of a code line in terms of the number of code tokens. The intuition behind designing NT is that, the more code tokens the line, the more likely to be defect-prone it is. Similar metrics can be found in [36].

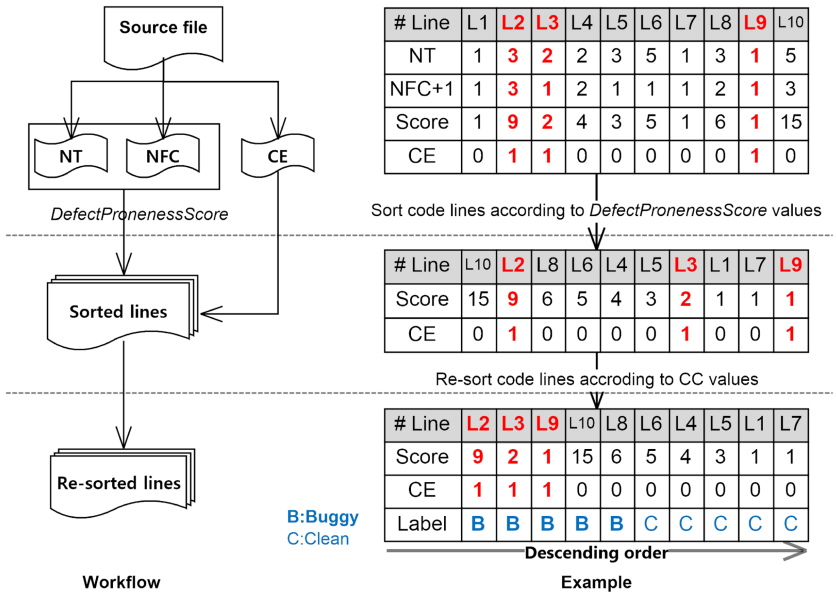

Fig. 6. An example for sorting lines in a defective file (The red number denotes that the line contains a control element, i.e., CE = 1).

Number of Function Calls (NFC). NFC measures the complexity of a code line in terms of the number of function calls. The intuition behind designing NFC is that, the more function the line, the more likely to be defect-prone it is. Similar metrics are used in [49, 88]. After obtaining the above three metrics, we combine NT and NFC to calculate the defect-proneness score of a code line l, in a similar way used in [76]. Note that not all buggy lines contain function calls. To avoid the final scores of these lines being set to 0, we add 1 to all NFCs. As a result, for each defective file predicted by the line-level prediction model (see section 4.2), we will calculate an integral defect score for each line in the file. As can be seen, it is very easy to compute the score in practice so that the application cost is very low.

$$DefectPronenessScore(l) = NT(l) * (NFC(l) + 1). \tag{5}$$

Figure 6 shows an example for illustrating how to sort the potentially buggy lines in a defective file. First, GLANCE ranks all the code lines in the file according to their corresponding `DefectPronenessScore` in descending order. Then, all lines that contain control elements (i.e., CE = 1) will be moved to the front of the ranking list. Finally, the ranking list is recommended to developers. Note that, it is very likely that the predicted suspicious values of several code lines are the same (i.e., they have a tied rank) since they are calculated as integers. In this situation, GLANCE cannot distinguish the ranking of these code lines. To simulate developers' habit of reviewing code from front to back, the predicted buggy code lines with tied ranks will be resorted according to their orders (i.e., line number) in the file. Meanwhile, we set a line-level threshold k to classify the lines in the top k percent (k = 50 by default) as buggy lines explicitly. For example, five lines (i.e., [L_2, L_3, L_9, L_{10}, L_8]) are predicted as buggy code lines in Figure 6.

After obtaining the rank of buggy lines in each defective file, we sort all defective files according to the corresponding predicted file-level score. In this case, the code lines contained in all files are predicted to be a uniform rank recommended to developers. For example, suppose there are three files that are predicted to be defective at file-level prediction stage, file A (score = 0.6), file B

(score = 0.8), and file C (score = 0.7), respectively. Meanwhile, the lines [L_A1, L_A3, L_A5] in file A are identified as buggy lines, the lines [L_B4, L_B2, L_B1] in file B are identified as buggy lines, and the lines [L_c5, L_C2] in file C are identified as buggy lines. Then, a rank of eight code lines (i.e., [L_B4, L_B2, L_B1, L_C5, L_C2, L_A1, L_A3, L_A5]) would be recommended to developers.

Note that, we provide both classification and ranking results for the following considerations. When the available budget is enough, we suggest the SQA teams to inspect all buggy lines predicted by GLANCE to identify as many buggy lines as possible. Note that the cutoff ratio can be easily customized by the project manager according to their actual need. Otherwise, it is recommended to inspect each line in turn according to the predicted ranking list until the budget has been run out. Since we examine three file-level classifiers: MD, EA, and LR in this study, we have three implementations for GLANCE, namely GLANCE-MD, GLANCE-EA, and GLANCE-LR, respectively. In the following, GLANCE refers to all baseline approaches represented by the above three implementations.

5 EXPERIMENTAL SETUP

This section introduces the details of the experimental setup. First, we present the research questions (Section 5.1) our experiments would like to answer. Then, we describe the detailed data collection process (Section 5.2) and the statistics information of the collected dataset (Section 5.3). Afterward, we expound on the prediction setting in our experiment: (Section 5.4). Next, we shed light on the performance indicators (Section 5.5), and describe how to acquire the results of SOTA approaches (Section 5.6). Finally, we interpret the statistical analysis method (Section 5.7).

5.1 Research Questions

To study how far have we come and how far yet to go in the field of code-line-level bugginess identification, we investigate the following research questions:

5.1.1 RQ1: (Effectiveness of GLANCE) How Well does GLANCE Perform in Identifying Defect-prone Code Lines? The purpose of RQ1 is to investigate the effectiveness of GLANCE in terms of both classification and ranking performances. As shown in Sections 3 and 4, GLANCE is proposed according to intuitive analyses, including motivating examples and case studies, on the ActiveMQ-5.0.0 project from [73]. Essentially, GLANCE is a baseline approach with a very simple structure. Therefore, the first question we are eager to know is whether GLANCE has an acceptable ability to identify defect-prone code lines in terms of both the classification and ranking performance. The answer to this RQ would shed light on the usefulness of GLANCE and provide us with its real performance at code-line level bugginess identification.

5.1.2 RQ2: (How Far have We Come in the Literature) How Effective do the SOTA Approaches Perform Compared against GLANCE in Identifying Defect-prone Lines? The purpose of RQ2 is to investigate the status quo of CLBI approaches in the literature by carrying out comparative experiments between existing CLBI approaches and GLANCE. It can be seen that, all of the existing CLBI approaches (i.e., NLPs and MITs) in the literature are exquisitely designed to improve the accuracy of buggy code lines identification, while the design of GLANCE is trivial relatively. Given its simplicity, however, a problem naturally arises: is GLANCE more effective than the existing approaches in identifying defect-prone code lines? If the answer is yes, this means that GLANCE is more practical for developers. More importantly, the comparison results of RQ2 can reveal whether the existing studies [60, 73] have made an outstanding progress in the field of CLBI.

5.1.3 RQ3: (How Far Yet to Go in the Industry) Have Popular Static Analysis Tools Achieved a Satisfactory Performance in Identifying Buggy Code Lines? The ultimate goal of CLBI study is to

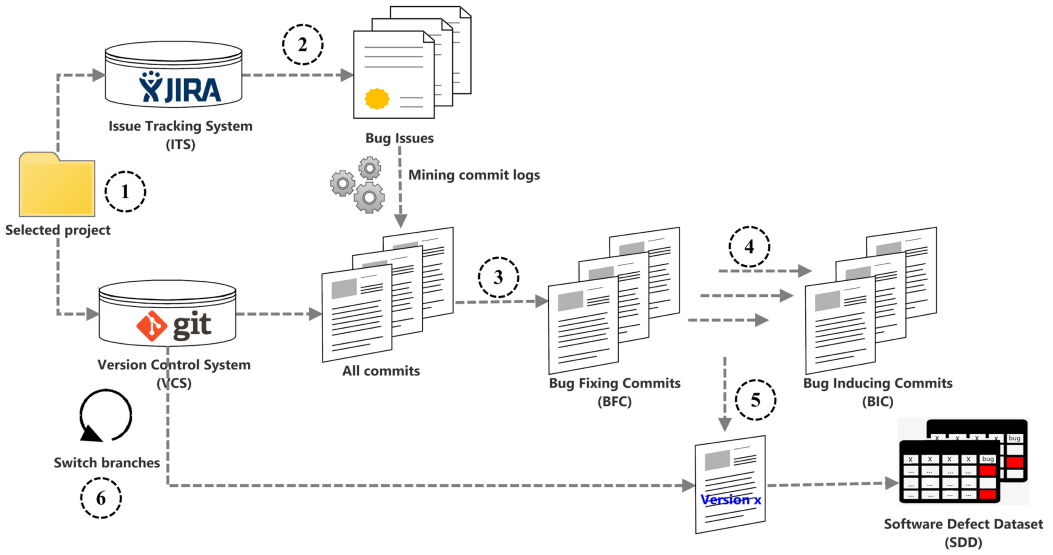

Fig. 7. The process of defect data collection for version x.

improve the effectiveness of practical defect detection applications (i.e., SATs). For this purpose, we first investigate whether the popular SATs in the industry have achieved a satisfactory performance in detecting buggy code lines by comparing them with GLANCE. Next, we analyze the difference between SATs and GLANCE based on the true positive buggy lines identified by them. The answer to this RQ would inform the real performance of popular SATs and exhibit how far yet to go in the industry.

5.2 Studied Dataset

Wattanakriengkrai et al., [73] have evaluated the effectiveness of the SOTA approaches in the dataset provided by Yatish et al., [82]. To the best of our knowledge, it is the latest dataset that can be used for code-line level bugginess identification. However, based on our observations, there are two problems in the above dataset that will affect its reliability. First, many test files are labeled as buggy in their dataset. According to the latest study [39], however, test files are not proper to be used for building in SQA models since they are designed to check the correctness of a project rather than deemed as any functional components of the project. Second, some buggy lines fixed in multiple branches are left out because their collection process does not consider different branches. Following the existing defect dataset collection methods [73] [60, 64], we collect a new large line-level defect dataset with the following steps (also shown in Figure 7). To enable subsequent researchers to collect more data from other Apache projects, we have made our data collection python scripts publicly available [8].

5.2.1 *Data Collection Process.*

(1) **Selecting studied projects. Apache software foundation**[5] **(ASF)** is a famous non-profit organization dedicated to supporting open-source software projects, many of which are favored by both researchers and developers [73]. The ASF projects deploy their code

[5]https://www.apache.org/.

repositories on GitHub,[6] in which researchers can easily obtain the entire code evolution and change process. In this work, we execute the query link[7] to search the desired projects. More concretely, we first sort the ASF Java projects in the descending order according to their star values. Then, we select the top projects whose star value is greater than 1,000 in the ranked list as candidates.

(2) **Collecting bug issues from JIRA.** Issue tracking system (e.g., JIRA) is used to record the issues (e.g., submit a bug or add a new feature) during the software development process. Note that, there are many fields (i.e., content) that indicate useful information in each issue. For example, the field description explains what the issue is through natural language and the field `issuetype` describes the category (e.g., bug or new feature) of each issue. At this step, we collect all available bugs by retrieving all issues whose `issuetype` is bug, `status` is resolved or closed, and `resolution` is fixed or resolved.[8] To conduct representative experiments, the projects whose number of bug issues is lower than 100 are filtered out at this step.

(3) **Identifying bug-fixing commits.** Bug-fixing commits (BFC for short) record the patch information that was used to fix a specific bug. For an ASF project (e.g., Camel[9]), developers are requested to provide the fix patches and to include the project name and JIRA ticket number (e.g., *Camel-9999: Some message goes here*) in the commit message. Based on such a constraint, we identify BFCs by mining commit log messages of a specific code repository. More specifically, we identify BFCs whose commit messages are matched by the regular expression "`project-\d+`". Meanwhile, the type of an issue id in BFC messages should be marked as bug in JIRA platform. Note that, there are some BFCs that do not satisfy the constraint, which is identified by detecting both the bug issue number and "fix" token in the commit message.

(4) **Tracking bug-inducing commits.** In contrast to BFC, bug-inducing commits (BIC for short) record when the buggy lines were introduced, which can be used to measure the life cycle of bugs. The intuition behinds mining BIC based on the cognition of "*defective lines are those lines that were changed by bug-fixing commits*" [59, 60]. To acquire BIC, we first need to determine the buggy files and buggy lines from BFC. According to prior works [60, 64, 73], the files modified by a BFC are considered as buggy files. Further, the deleted lines (or changed) lines in a buggy file are deemed as buggy lines. Based on this cognition, it is easy to determine the buggy lines by parsing the patches provided by BFCs. Note that there are many different types (e.g., Java source files, resource files, and configuration files) of files in a project. In the literature, only Java source files will be considered as defective files because they contain the actual program code that needs to be fixed. In other words, the consensus is that only errors in Java source files are considered as bugs. In addition, for a given version of a project, it is not proper to consider the issues in the test file as bugs of the version [39]. In our study, therefore, only the non-test type of Java files can be regarded as buggy files. Once the buggy files and the corresponding buggy lines are identified, BICs can be easily tracked using the git blame command.

(5) **Mapping buggy lines to versions.** Having obtained both BFC and BIC of each buggy line, we are able to identify its corresponding location (i.e., line number) in the file of a specific project version (or release). In detail, we first calculate the life period of buggy lines according to BIC and BFC. Then, we collect the publish time of the specific version and filter out the buggy lines whose life period cannot cover the publish time. In other words, a buggy line

[6] https://github.com/apache.

[7] https://github.com/search?l=Java&o=desc&q=apache&s=stars&type=Repositories.

[8] project = [project name] AND issuetype = Bug AND status in (Resolved, Closed) AND resolution in (Fixed, Resolved) ORDER BY key ASC.

[9] https://camel.apache.org/manual/latest/contributing.html#pull-request-at-github.

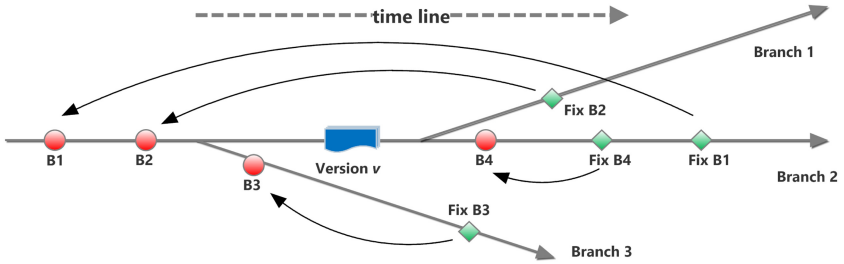

Fig. 8. Labeling the versions that are affected by a bug.

belongs to a version if it satisfies the constraint: `BIC time <= publish time < BFC time`. Finally, we identify the location (i.e., line number in the target version) of selected buggy lines by parsing the diff file between the code snapshots of BFC and the target version.

(6) **Combining bugs from different branches.** As is known, there are many different branches (i.e., development flow) in a git repository. The release of a project version and the bug-fixing actions can occur in different branches. As shown in Figure 8, the red circle denotes a BIC action, and the green diamond denotes a BFC action. There are three branches Branch 1, Branch 2, and Branch 3, and four bugs `B1`, `B2`, `B3`, and `B4`. As can be seen, the target version v exists both in Branch 1 and Branch 2. Considering the Branch 1, we can identify buggy lines of `B2` for the target version v according to the above five steps. Similarly, the buggy lines of `B1` can also be identified for the target version v in terms of Branch 2. The example indicates that some buggy lines can be missed if we only consider the single default (or main) branch. Therefore, we switch different branches to collect the corresponding buggy lines and combine them to obtain a complete defect dataset.

5.2.2 Dataset Summary. Following the above data collection process, we collect a large dataset that contains both file- and line-level ground truths. Specifically, the former contains the file name, the defect label, and the corresponding content of each source file, while the latter contains the list of defective files and their corresponding buggy line numbers.

In summary, the dataset consists of 19 popular open-source Java systems from Apache Software Foundation with 142 different releases. As shown in Table 2, the first and second columns report the name and application domain of each project, respectively. The third and fourth columns report the number (7 on average) of selected releases and the number of bugs (i.e., #Bugs, 217 on average) of each project, respectively. Note that, the #Bugs is a new statistic compared with SOTA work [73]. Since a bug can be fixed in a BFC, there is a corresponding relationship between a BFC and a fixed bug. In our context, therefore, we consider the number of BFC as #Bugs, which can be used to evaluate the performance of a CLBI approach in the perspective of bug level. The fifth to seventh columns report the file-level information for each project, including the number of all files (1507 on average), the number of buggy files (191 on average), and the proportion of buggy files (12.67% on average) of each project, respectively. The eighth to tenth columns report the line-level information for each project, including the number of all lines (262.5 KLOC on average), the number of buggy lines (2.7 KLOC on average), and the proportion of buggy lines (1.03% on average), respectively. For the ease of observation, we only report the average value for each data item of both file- and line-level information. As can be seen, these studied systems vary in application domains, size of systems, and the proportion of buggy files. In this sense, they are representative and hence can be used to evaluate the effectiveness of the investigated CLBI approaches adequately. The more detailed information for each release is publicly available in our replication kit [8].

Table 2. The Detailed Statistical Information of All Releases in Each Studied Project

Project	Domain	#Rel	#bugs	File-level			Line-level (KLOC)		
				Avg. #Files	Avg. #Buggy	Avg. %Buggy	Avg. #Lines	Avg. #Buggy	Avg. %Buggy
ambari	Hadoop management tool	7	62~489	2,307	313	13.57%	366.3	5.4	1.47%
activemq	Messaging and integration patterns	14	97~845	1,835	226	12.32%	292.8	3.3	1.13%
bookeeper	Scalable storage service	3	31~157	240	88	0.3667	57.7	2.2	0.0381
calcite	Dynamic data management framework	8	301~554	1,430	622	43.50%	304.5	12.8	4.20%
cassandra	Highly-scalable partitioned row store	6	138~407	583	156	26.76%	155.7	1.9	1.22%
flink	Open source stream processing framework	2	55~78	3,583	105	2.93%	564.1	1.1	0.20%
groovy	Java-syntax-compatible OOP	14	36~641	914	73	7.99%	194.1	1.2	0.62%
hbase	Distributed scalable data store	5	239~493	1,076	263	24.44%	487	4.6	0.94%
hive	Data warehouse system for Hadoop	4	56~321	3,263	214	6.56%	953.1	2.8	0.29%
ignite	Horizontally scalable computing platform	3	824~859	2,658	826	31.08%	612	10.3	1.68%
log4j2	Log information system	11	215~255	798	180	22.56%	109	1.7	1.56%
mahout	Machine learning applications environment	6	60~117	933	164	17.58%	134.7	2.2	1.63%
maven	Software project management tool	6	32~124	663	69	10.41%	94.6	0.6	0.63%
nifi	Distribute data processing system	4	323~545	2,809	513	18.26%	432.1	5.9	1.37%
nutch	Highly scalable web crawler	13	54~158	332	64	19.28%	56.7	0.6	1.06%
storm	Distributed real-time computation system	5	5~183	997	44	4.41%	166.4	0.4	0.24%
tika	Metadata and structured text detection tool	12	105~179	393	92	23.41%	66.2	1.4	2.11%
struts	Java web applications framework	15	94~227	1,087	120	11.04%	150.8	1.3	0.86%
zookeeper	Centralized service system	4	40~80	339	72	21.24%	63.7	0.8	1.26%
Average	-	7	217	1,507	191	12.67%	262.5	2.7	1.03%

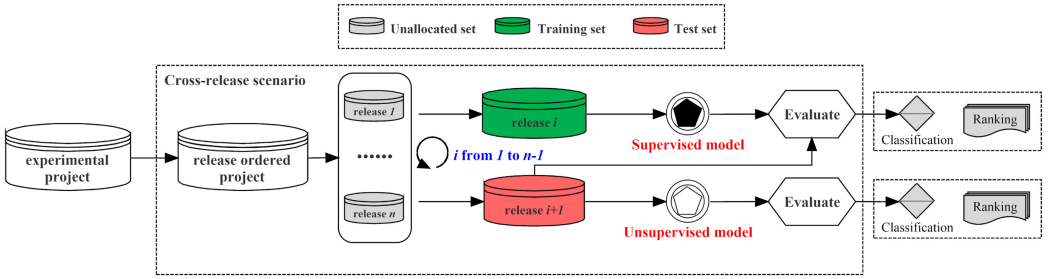

Fig. 9. Overview of the cross-release-prediction scenario.

5.3 Prediction Setting

Based on the above dataset, we apply the *cross-release-prediction setting* (CR) to conduct our experiments and to acquire the classification and ranking results of the studied projects.

Motivation. The CR prediction setting is applied based on the two reasons. First, the CR prediction setting mimics a practical model application scenario. As the above dataset we collected, generally, there are multiple releases in a software project, and the number of releases will continue to increase over time. Naturally, one can leverage previous release to build model and apply it in the next new release. Second, this setting has been used in many previous studies [73] [87], which means that it is extensively accepted by the current researchers.

Approach. Figure 9 shows an overview of the CR prediction setting. Under the setting, we study the identification performance of a CLBI approach by building and testing it based on the data from different releases in a project. More specifically, supposing there are n releases in a project, three steps are carried out as follows:

(1) First, we rank the releases in chronological order according to the release date;
(2) Second, we build a CLBI model based on release `i` (`i` ∈ [1, n-1]) and evaluate it on release `i+1`. Note that, an unsupervised model will be evaluated on `i+1` directly without a building

Table 3. The Performance Indicators used for Code-line-level Defect Prediction Evaluation

Scenario	Name	Definition	Meaning	Better	Range
Classification	Recall	$\|TP\|/(\|TP\|+\|FN\|)$	The proportion of buggy lines a CLBI model can identify	High	[0, 1]
	FAR	$\|FP\|/(\|FP\|+\|TN\|)$	The proportion of clean lines are incorrectly predicted as buggy ones by a CLBI model	Low	[0, 1]
	D2H	$\sqrt{[(1-\text{ Recall })^2+(0-FAR)^2]}/2$	A combination of recall and FAR	Low	[0, 1]
	CoF	$\|FN\|/(\|FN\|+\|TN\|)$	The proportion of buggy lines in all clean lines predicted by a CLBI model	Low	[0, 1]
	MCC	$\frac{\|TP\|\times\|TN\|-\|FP\|\times\|FN\|}{\sqrt{(\|TP\|+\|FP\|)(\|TP\|+\|FN\|)(\|TN\|+\|FP\|)(\|TN\|+\|FN\|)}}$	The correlation coefficient between actual and predicted outcomes	High	[-1, 1]
Ranking	R@20%	$\frac{\#Buggy lines in top 20\% of a rank}{\#All buggy lines}$	The proportion of buggy lines a CLBI model can identify when only the top 20% code lines of a rank are inspected	High	[0, 1]
	IFA	k	The initial number of clean lines need to be inspected before finding the first buggy line	Low	[0, N]
Bug level	HOB	*#Hit bugs/#All bugs*	The ratio of bugs hit by a model	High	[0, 1]

process. As a result, we can collect n-1 groups of identification results for each project. For example, there are two groups of results for project bookeeper (i.e., r-4.0.0 → r-4.2.0, r-4.2.0 → r-4.4.0). In total, we can obtain 123 (i.e., 142-19) predicted pairs in this study.

(3) Finally, we evaluate each result individually and compute the average value to estimate the performance of a CLBI approach on a project.

5.4 Performance Indicators

Under the above prediction scenario, A CLBI approach can acquire two types of prediction results: classification and ranking results. To conduct a comprehensive evaluation for CLBI approaches, following previous works [59, 73, 85], we select 8 popular performance indicators in our experiments, all of which are presented in Table 3. In this table, the first column reports the scenario to which a specific indicator belongs. The second to fourth columns show the name, the mathematical definition, and the meaning of a specific indicator, respectively. The last two columns present the expected direction for a better performance value and the value range of each indicator.

5.4.1 Classification Performance Indicators. Under the classification scenario, an approach will assign a label (true or false) for each line to indicate whether the line is defective or not. According to the predicted labels and the actual labels, the results of one release can be divided into four categories (or sets): TP (the set of lines correctly classified as defective), TN (the set of lines correctly classified as not defective), FP (the set of lines incorrectly classified as defective), and FN (the set of lines incorrectly classified as not defective). As shown in Table 3, all classification performance indicators are based on the size of TP, TN, FP, and FN directly or indirectly.

— **Recall** measures the proportion of buggy lines correctly identified by a prediction model. The goal of a prediction model is to identify buggy lines as many as possible. Therefore, the higher is the Recall value, the better is the model.

— **FAR** (i.e., False Alarm Rate [73]) measures the proportion of the number of clean lines incorrectly predicted as buggy ones. A higher FAR value indicates that developers need to spend more effort to inspect defective lines incorrectly identified by a prediction model. Therefore, the lower is the FAR value, the better is the model.

— **D2H** (i.e., Distance-to-Heaven [1, 73]) is a combination of Recall and FAR. Following the definition (see Table 3), D2H will equal to 0 if a model achieves the best Recall (=1) and the

best FAR (=0), and equal to 1 if it achieves the worst Recall (=0) and the worst FAR (=1). Therefore, the lower is the D2H value, the better is the model.

— **CoF** [85] (i.e., Cost of Failures) is the number of true buggy lines to the number of lines predicted as clean. According to Zhang et al., [85], the cost of software failures will arise from the defects that are missed by a prediction model. They derived CoF from "the intuitions that a defect prediction based SQA strategy should at least cost less than the strategy of inspecting all modules and the strategy of inspecting randomly sampled modules". This requires that a model should have a CoF lower than $\min(C_i/C_{fn}, \%D)$, where C_i is the average cost for inspecting an instance, C_{fn} is the average cost of missing a defective instance, and D is the percentage of defective instances in the project. In practice, the lower is the CoF value, the better is the model.

— **MCC** (i.e., Matthews Correlation Coefficients [73]) measures the correlation coefficient between actual and predicted outcomes. According to [73], an MCC value of 1 indicates a perfect prediction and vice versa. Therefore, the higher is the MCC value, the better is the model.

5.4.2 Ranking Performance Indicators. Under the ranking scenario, an approach will rank the lines in a target release in descending order from the most to the least risky scores. Developers can inspect each line in turn based on the list of recommendations until they find out (all) the buggy lines. In the literature [40, 59, 60, 83], researchers usually leverage the following two ranking indicators.

— **R@20%** [60, 83] measures the proportions of buggy lines in the top 20% recommended lines ranked by a prediction model. This indicator leverages a cutoff point to split the ranked list into two parts: buggy lines (i.e., the lines ranked before the cutoff) need to be inspected and clean lines (i.e., the lines ranked after the cutoff). By default, the threshold is set to 20%. After splitting the ranked list, the calculation process of R@20% is the same as the classification indicator Recall. Like the meaning of Recall, the higher is the R@20% value, the better is the model.

— **IFA** (i.e., Initial False Alarm [83]) is the number of lines that need to be inspected before finding the initial buggy line. According to Parnin et al., [54], developers would like to stop inspecting when they could not find the buggy lines in the top recommended lines. Therefore, the lower is the IFA value, the better is the model.

5.4.3 Bug Level Indicators. In the field of file-level bug localization [39, 76], a bug is considered as a hit by a model if any relevant buggy files are identified (a bug is related to multiple buggy files). Similarly, a bug is also associated with a set of buggy code lines. Based on this cognition, we hence use the following indicator to evaluate the bug level performance of a CLBI model.

— **HOB** (i.e., Hit of Bugs [59]) measures the ratio of bugs hit in all bugs in a target release. According to Rahman et al., [59], HOB can be used to measure the ability to identify bugs of a model. Specifically, a bug is considered as hit by a model if even a single identified line overlaps the set of buggy lines relevant to the bug. The identification of the single buggy line means that there is a strong possibility that the bug would be noticed during the code review process. Therefore, developers can find and fix a bug more effectively if some of the associated buggy code lines are identified by a CLBI model. In practice, the higher is the HOB value, the better is the model.

5.5 Studied CLBI Approaches

As shown in Table 4, our study examines the effectiveness of four categories of CLBI models. Specifically, the first to last columns report the category, the name abbreviation, and the brief

Table 4. The Investigated CLBI Approaches in This Study

Objects	Category	Approach	Description
Models	GLANCE	GLANCE-MD	GLANCE based on ManualDown classifier
		GLANCE-EA	GLANCE based on Effort-aware ManualDown classifier
		GLANCE-LR	GLANCE based on Logistic Regression classifier
	NLPs	NGram	Ranking lines by the average entropy values of n-grams
		NGram-C	Ranking lines by the average entropy values of cache n-grams
	MITs	TMI-LR	TMI based on Logistic Regression classifier
		TMI-MNB	TMI based on Multinomial Naive Bayes classifier
		TMI-SVM	TMI based on Support Vector Machine classifier
		TMI-DT	TMI based on Decision Tree classifier
		TMI-RF	TMI based on Random Forest classifier
		LineDP	LIME based on Logistic Regression classifier
Tools	SATs	PMD	A source code analyzer
		CheckStyle	A development tool to check coding standard

description, respectively. In the following, we introduce how to obtain the identification results of the approaches in different categories.

GLANCE. This category contains three simple baseline approaches: GLANCE-MD, GLANCE-EA, and GLANCE-LR, the detail of GLANCE has been explained in Section 4 and hence will not be repeated here.

As aforementioned in Section 2.1, there are three categories of approaches that can be used to predict defects at the code line level. To conduct an all-around exploration, we investigate all these approaches in this study.

Natural Language Processing based approaches (NLPs). This category contains two approaches: NGram and NGram-C, both of which leverage the entropy value of code tokens to represent the bugginess score of each code line. In particular, an entropy value of NGram is estimated by a normal n-gram model, while an entropy value of NGram-C is estimated by a cache-based n-gram model [60] that considers both global and local contexts. Specifically, NLPs first compute the entropy for each code token in source files based on the probability estimated by n-gram models. Then, they consider the average entropy value of tokens in the same code line as the bugginess score. Finally, they predict the lines whose entropy values are greater than a threshold (i.e., 0.7 in [73]) as defect-prone lines. Note that, the NGram and NGram-C in this study are built based on the n-gram implementation of Hellendoorn et al., [20], which has been used in previous study [73].

Model Interpretation based approaches (MITs). This category contains six approaches: TMI-LR, TMI-MNB, TMI-SVM, TMI-DT, TMI-RF, and LineDP. The first five belong to the traditional model interpretation approach (marked by prefix TMI), which explains top 20 possible risky tokens based on the trained classifiers Logistic Regression, Multinomial Naïve Bayes, Support Vector Machine, Decision Tree, and Random Forest models, respectively. The last one is the latest MIT-based approach that uses a model-agnostic technique LIME to make an explanation for a classifier [73]. The MITs are all two-stage approaches. For a given project, MITs first predict the defective files by a file-level classifier built on the tokens in the training data. The BoT (Bag of tokens) model is used to transform the source code into numerical features for building file-level classifiers. Then, they predict a bugginess score for each code line in each predicted defective file by explaining the file-level classifier. Specifically, to interpret a trained classifier, the top 20 features (i.e., risky code tokens) that make the largest positive contributions are extracted as the bug indicators. As such, the bugginess score of each code line is computed by counting the number of occurrences of all bug indicators.

All the above MITs are built based on the popular data mining tool Python Scikit-Learn package,[10] which has provided a robust implementation for each classifier. Following the previous work [73], we also leverage the default parameter values to build each classifier. Additionally, the coefficient values (`coef_` in Scikit-Learn), importance of features (`feature_importances_` in Scikit-Learn), and LIME scores (LIME package) are used to identify risky code tokens in linear-based models (TMI-LR, TMI-MNB, and TMI-SVM), tree-based models (TMI-DT and TMI-RF), and LineDP, respectively.

Static Analysis Tool based approaches (SATs). This category contains two approaches: PMD and Checkstyle, which are both popular static Java analysis tools. By specifying a set of rules, SATs will detect and generate a warnings report (that contains all detected buggy lines and their corresponding severity) for a given project. In this study, four sets of important rules that are associated with defect (i.e., design, errorprone, multithreading, and security) in PMD are used, while the two built-in rule sets (`google` and `sun`) of CheckStyle are used. The severity of each buggy line is transformed as a bugginess score by taking the reciprocal. For example, the severity "2" will be transformed to 0.5 (i.e., 1/2). Finally, a recommended rank of reported buggy code lines will be provided based on their corresponding bugginess scores. Note that we do not investigate the popular SAT FindBugs. The reason is that FindBugs can only analyze the compiled classes or jar files, which are not easy to acquire in a specific release of the code repository. Often, the compilation process of source code in the whole repository will encounter various errors. In addition, according to [60], there is no significant difference in performance between FindBugs and PMD.

5.6 Statistical Analysis

After calculating the values of all performance indicators, we need to leverage statistical analysis to measure the differences between GLANCE and the SOTA approaches. In our context, the statistical analysis is performed as follows: We employ BH-corrected p-values [6] from the Wilcoxon signed-rank test, a practical and powerful adjustment approach of p-values to control the false discovery rates for independent test statistics, to examine whether there is a statistically significant difference at the significance level of 0.05. Furthermore, we use the Cliff's delta (δ) to quantify the effect size of the difference. This enables us to know whether the magnitude of the difference in performances between two approaches is important from the viewpoint of practical application. By convention, the magnitude of the difference is considered negligible (N, $|\delta| < 0.147$), small (S, $0.147 \leq |\delta| < 0.33$), median (M, $0.33 \leq |\delta| < 0.474$), or large (L, $|\delta| > 0.474$).

6 EXPERIMENTAL RESULTS

In this section, we conduct a large experiment based on the above setup and analyze the experimental results using the aforementioned performance indicators to answer the corresponding research questions.

6.1 RQ1: Effectiveness of GLANCE

Figure 10 reports the performance distribution comparison among GLANCE-MD (marked gray), GLANCE-EA (marked red), and GLANCE-LR (marked blue) based on 123 prediction pairs. From Figure 10, we make the following observations. First, GLANCE has a strong ability to identify buggy codes, especially for GLANCE-MD and GLANCE-LR. More specifically, about half of the prediction pairs (see median line) can identify more than 50% of the buggy code lines (Recall) or reveal more than 60% of the bugs (HOB). Meanwhile, 40% of buggy code lines (R@20%) can be identified by half of the prediction pairs when only spending 20% of the total review effort. Second,

[10] https://scikit-learn.org/stable/.

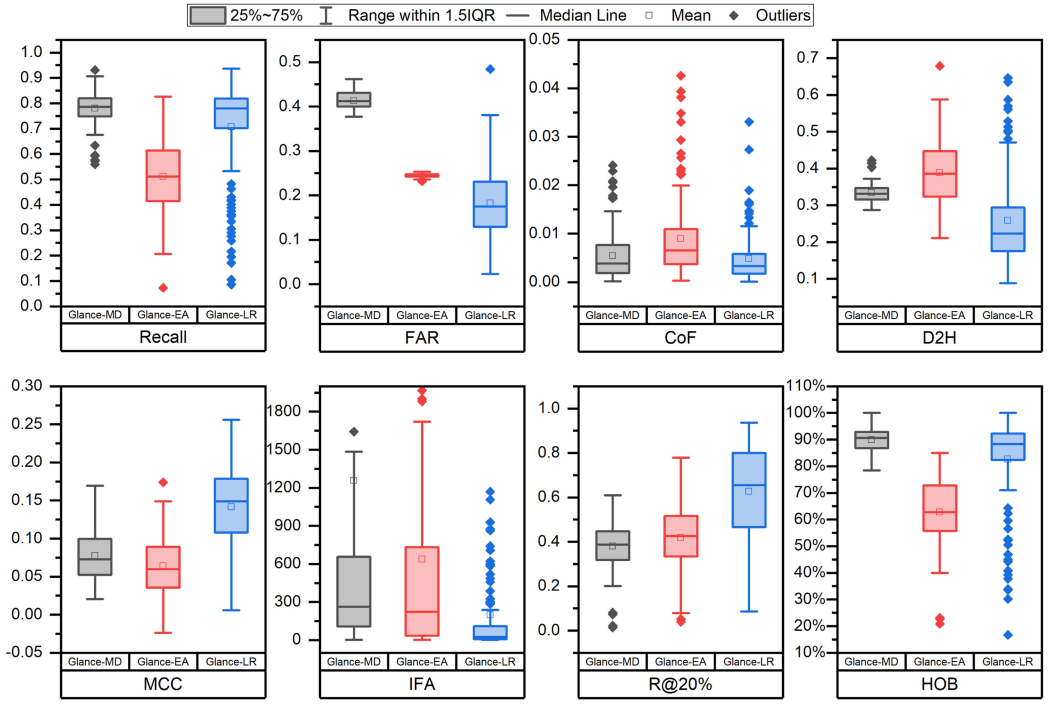

Fig. 10. Performance comparison among GLANCE-MD, GLANCE-EA, and GLANCE-LR.

the performance of GLANCs is obviously stronger than that of a random guess model. In terms of the classification indicators, most prediction pairs can obtain an MCC value greater than 0, a D2H, and a FAR lower than 0.5 regardless of whether the GLANCE-MD, GLANCE-EA, or GLANCE-LR is considered. However, it should be noted that the false alarm rate is still very high from a practical point of view. In addition, there is only a small ratio of buggy instances in the predicted clean lines, i.e., CoF achieves a very small value (< = 0.02) on average, which means a low cost caused by missing defects. Therefore, GLANCE can be considered as a cost-effective prediction model. Considering the ranking indicators (i.e., IFA), about half of the prediction pairs of GLANCE can identify the first buggy line within the first 300 lines of code to be reviewed. Note that, the average number of lines of code for a project is 260 KLOC, which is much higher than 300. Under this circumstance, this is an excellent IFA value. Third, GLANCE-LR has the best identification performance among the three GLANCE based baseline approaches, regardless of which indicator is considered. Since the difference between the three approaches lies in the use of different file-level classifiers, **Logistic Regression** (**LR**) is proved to be the most suitable file-level classifier for GLANCE in this study.

According to the above preliminary observations, we further investigate the concrete prediction performance of GLANCE in detail. Table 5 summarizes the average classification, ranking, and bug level performance of GLANCE-LR based on 19 individual projects, respectively. Given the similarity of the analysis and the length of the article, the complete experimental results for GLANCE-MD and GLANCE-EA can be viewed in online resources [11]. In terms of classification performance, on median, GLANCE-LR can achieve a value of 0.703, 0.191, 0.005, 0.286, and 0.124 in terms of Recall, FAR, CoF, D2H, and MCC, respectively. The Recall value shows that a large proportion (i.e., 70.3%) of buggy lines in the files of a target release can be identified by GLANCE-LR.

Table 5. The Performance of GLANCE-LR under Cross-release-prediction Scenario (The Best Performance Value of Each Indicator is Marked in **Bold** While the Worst Performance Value is Marked in Underlined)

Project	Classification					Ranking		Bug level
	Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
ambari	0.587	0.191	0.007	0.341	0.094	654	0.509	75%
activemq	0.768	0.142	0.003	0.202	0.171	44	0.759	86%
bookeeper	0.677	0.369	0.005	0.371	0.062	261	0.383	86%
calcite	0.772	0.338	0.014	0.289	0.178	27	0.45	**93%**
cassandra	0.769	0.238	0.004	0.243	0.124	84	0.57	85%
flink	0.086	**0.03**	**0.001**	0.647	0.012	621	0.086	17%
groovy	0.588	0.145	0.003	0.326	0.083	374	0.572	72%
hbase	0.703	0.254	0.004	0.286	0.094	87	0.49	83%
hive	0.298	0.065	0.002	0.499	0.045	2143	0.298	51%
ignite	0.782	0.283	0.005	0.253	0.14	64	0.601	88%
log4j2	0.773	0.195	0.005	0.217	0.181	**14**	0.693	89%
mahout	0.67	0.155	0.006	0.262	0.159	118	0.641	80%
maven	**0.859**	0.141	**0.001**	**0.142**	0.142	245	**0.859**	91%
nifi	0.65	0.212	0.005	0.29	0.111	20	0.555	84%
nutch	0.79	0.19	0.003	0.201	0.152	113	0.727	85%
storm	0.505	0.039	0.002	0.351	0.098	234	0.505	64%
tika	0.788	0.223	0.007	0.219	**0.198**	59	0.644	87%
struts	0.742	0.126	0.005	0.208	0.17	86	0.735	89%
zookeeper	0.654	0.229	0.007	0.305	0.097	61	0.542	90%
Median	0.703	0.191	0.005	0.286	0.124	87	0.57	85%
Mean	0.656	0.188	0.005	0.297	0.122	279	0.559	79%

This also indicates that there is a certain correlation between buggy lines and our designed code-line-level metrics. Although it is unsatisfying that GLANCE-LR has a slightly higher FAR (0.191), GLANCE can achieve an excellent (low) value (0.286) when considering the D2H, which is a composite indicator that is combined by Recall and FAR. As for the ranking performance, on median, GLANCE-LR can achieve a value of 0.570 in terms of R@20%. That is, GLANCE-LR can identify 57.0% buggy lines when inspecting only the top 20% code lines in its recommendation list. This indicates that, the ranks of code lines provided by GLANCE-LR are superior to a random ranking model (whose R@20% is 20% theoretically). Meanwhile, GLANCE-LR achieves a lower median IFA value (i.e., 87), which indicates that developers can find the first buggy line if they review 87 code lines in the rank list. Regarding the ratio of hit bugs, on median, 85% of bugs can be revealed by GLANCE-LR. In other words, lines of code identified by GLANCE-LR are associated with 85% of specific bugs. Considering the performance results of individual projects, GLANCE-LR performs best for the project maven (obtains the optimal values in four indicators Recall, CoF, D2H, and R@20%) but performs the worst for the project flink (obtains the worst values in three indicators MCC, R@20%, and HOB).

Conclusion. *Overall, in terms of performance indicators alone, the results indicate that, GLANCE based baseline approaches can achieve acceptable performance values regardless of whether classification, ranking, or bug level indicators are considered. This is especially true for GLANCE-LR. Meanwhile, a drawback is that GLANCE still can produce high false alarms.*

Table 6. The Mean Value and Median Value of Different CLBI Approaches (The Best Performance Value of Each Indicator is Marked in **Bold** While the Worst Performance Value is Marked in Underlined)

	Approaches	Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
Mean	NGram	0.552	0.302	0.009	0.385	0.058	693	0.43	80%
	NGram-C	0.275	0.173	0.012	0.528	0.029	1199	0.267	57%
	TMI-LR	0.093	0.022	0.012	0.642	0.052	**82**	0.093	24%
	TMI-MNB	0.501	0.189	0.008	0.381	0.082	514	0.466	71%
	TMI-SVM	0.074	**0.018**	0.013	0.655	0.047	104	0.074	20%
	TMI-DT	0.206	0.060	0.011	0.564	0.064	119	0.205	46%
	TMI-RF	0.383	0.107	0.009	0.447	0.094	145	0.375	62%
	LineDP	0.562	0.152	0.007	0.334	0.118	266	0.536	75%
	GLANCE-MD	**0.779**	0.413	0.005	0.334	0.077	1256	0.38	**90%**
	GLANCE-EA	0.512	0.245	0.009	0.388	0.064	637	0.418	63%
	GLANCE-LR	0.707	0.182	**0.005**	**0.259**	**0.141**	197	**0.627**	83%
Median	NGram	0.555	0.308	0.007	0.38	0.053	233	0.432	81%
	NGram-C	0.278	0.170	0.009	0.527	0.027	288	0.267	58%
	TMI-LR	0.059	**0.012**	0.010	0.666	0.045	24	0.059	21%
	TMI-MNB	0.527	0.188	0.006	0.367	0.086	143	0.487	72%
	TMI-SVM	0.042	0.007	0.010	0.677	0.033	**18**	0.042	16%
	TMI-DT	0.201	0.049	0.009	0.566	0.063	58	0.201	49%
	TMI-RF	0.439	0.099	0.007	0.406	0.093	53	0.429	67%
	LineDP	0.611	0.154	0.005	0.301	0.127	79	0.594	80%
	GLANCE-MD	**0.786**	0.413	0.004	0.331	0.073	264	0.387	**91%**
	GLANCE-EA	0.512	0.246	0.007	0.386	0.060	224	0.425	63%
	GLANCE-LR	0.781	0.175	**0.003**	**0.223**	**0.149**	26	**0.655**	88%

6.2 RQ2: How Far have We Come in the Literature

This section answers the question of, in the literature, how far have we come for CLBI task by comparing the SOTA approaches with simple GLANCE based approaches. More specifically, we present the detailed analysis for performances (Section 6.2.1) and differences (Section 6.2.2) of the prediction results obtained from different approaches, respectively.

6.2.1 Performance Comparison. Table 6 makes a summary in terms of the median and mean values for all indicators. The best performance of each indicator is marked in **bold** while the worst performance is marked in underlined. Meanwhile, Figure 11 reports the performance comparison of three important indicators D2H, R@20%, and HOB between existing SOTA approaches (NLPs and MITs) and GLANCE based approaches for each project. Note that, a blue cell denotes a better performance value while a red cell denotes a worse performance value. The detailed result for each prediction pair is provided online [11]. According to the comparison results based on eight indicators, we make the following observations.

— ***NLPs vs. GLANCE.*** Overall, it is obvious that NGram performs a performance superior to NGram-C under most indicators. One possible reason is that, the "naturalness" of a source file is increased by NGram-C due to the file is weighted by both the local (a.k.a., cache) and global (considers all files in a project) components of NGram-C. According to [60], the more "naturalness" the codes, the lower entropy values, and the less likely it is to contain defects. Therefore, compared with NGram, the entropy values computed by NGram-C are lower so that more code lines are predicted as clean (i.e., entropy value < 0.7). A lower Recall

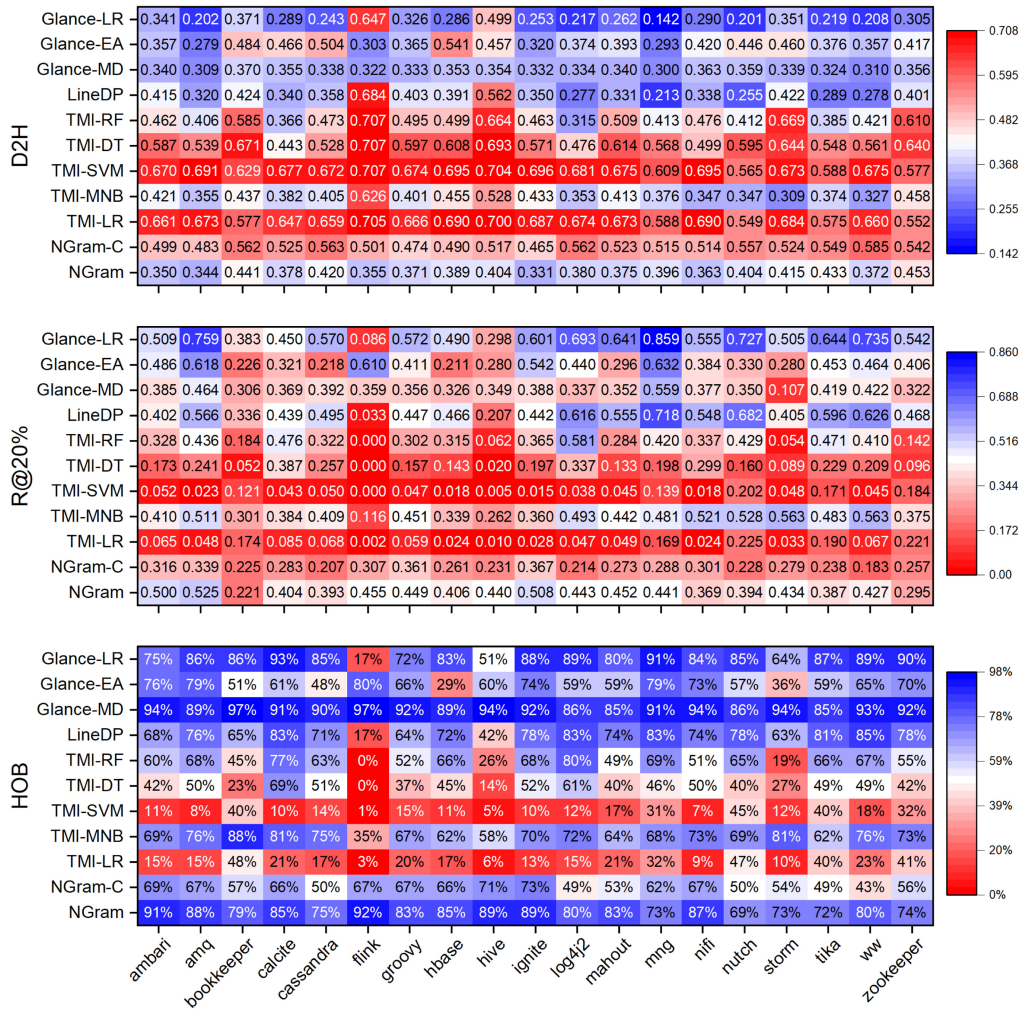

Fig. 11. Performance (D2H, R@20%, and HOB) comparison between NLPs, MITs, and GLANCE for each project.

value leads to overall performance degradation for NGram-C. More specifically, on median, NGram can identify 55.5% buggy lines with a median D2H value of 0.380. Meanwhile, on median, 81% bugs can be revealed by NGram approach. Therefore, we regard NGram as the representative NLP approach and compare it with GLANCE. Considering a single project, NGram achieves the best D2H value of 0.344 in project amq while worst D2H value of 0.453 in project zookeeper. We can see that, NLPs do not show a performance superior to GLANCE. In particular, compared with NGram, GLANCE-EA has a very similar performance under most indicators. However, both GLANCE-MD and GLANCE-LR perform better than NGram regardless of whether Recall, CoF, D2H, MCC, or HOB is considered. The above results indicate that, when identifying buggy code lines, the performance of NLPs is no better than that of GLANCE.

— ***MITs vs. GLANCE.*** Among all MITs, LineDP performs the best in identifying buggy code lines. More specifically, on median, LineDP can identify 61.1% buggy lines with a median

Table 7. The Results from Wilcoxon-signed Test and Cliff's Delta When Comparing GLANCE-LR with the SOTA Approaches (Bold Fonts Denote that the p-value is Less than 0.001) (Abbr. N: Negligible; S: Small; M: Medium; L: Large)

	Approaches	Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
BH-corrected p-values	NGram	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
	NGram-C	**0.000**	0.173	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
	TMI-LR	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.033	**0.000**	**0.000**
	TMI-MNB	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.043	**0.000**	**0.000**
	TMI-SVM	**0.000**	0.073	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
	TMI-DT	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.562	**0.000**	**0.000**
	TMI-RF	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.343	**0.000**	**0.000**
	LineDP	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.007	**0.000**	**0.000**
Cliff's delta	NGram	0.648 (L)	-0.795 (L)	-0.397 (M)	-0.694 (L)	0.819 (L)	-0.582 (L)	0.607 (L)	0.415 (M)
	NGram-C	0.884 (L)	0.046 (N)	-0.556 (L)	-0.897 (L)	0.933 (L)	-0.655 (L)	0.880 (L)	0.784 (L)
	TMI-LR	0.977 (L)	0.958 (L)	-0.583 (L)	-0.976 (L)	0.796 (L)	0.092 (N)	0.975 (L)	0.961 (L)
	TMI-MNB	0.983 (L)	0.969 (L)	-0.589 (L)	-0.983 (L)	0.824 (L)	0.168 (S)	0.982 (L)	0.969 (L)
	TMI-SVM	0.689 (L)	-0.092 (N)	-0.369 (M)	-0.678 (L)	0.661 (L)	-0.464 (M)	0.521 (L)	0.670 (L)
	TMI-DT	0.919 (L)	0.829 (L)	-0.540 (L)	-0.918 (L)	0.755 (L)	-0.121 (N)	0.907 (L)	0.851 (L)
	TMI-RF	0.771 (L)	0.539 (L)	-0.431 (M)	-0.755 (L)	0.493 (L)	-0.169 (S)	0.644 (L)	0.742 (L)
	LineDP	0.624 (L)	0.209 (S)	-0.269 (S)	-0.486 (L)	0.276 (S)	-0.234 (S)	0.308 (S)	0.528 (L)

D2H value of 0.301. Meanwhile, on median, 80% bugs can be revealed by LineDP approach. Therefore, we regard LineDP as the representative MIT approach and compare it with GLANCE. Considering a single project, LineDP achieves the best D2H value of 0.213 in project mng while worst D2H value of 0.684 in project flink. We can see that, LineDP does not always show a performance superior to GLANCE. Note that, although the performances of GLANCE-MD and GLANCE-EA are lower than that of LineDP, they perform better than the other majority MITs. Surprisingly, however, GLANCE-LR performs better than LineDP in almost all performance indicators. For example, GLANCE-LR has a median improvement of 27.8%, 10.3%, and 10% in terms of Recall, R@20%, and HOB, respectively. The above results indicate that, GLANCE-LR is superior to LineDP, and GLANCE-MD and GLANCE-EA are superior to the other MITs when identifying buggy code lines.

Table 7 reports the 64 (8 SOTA approaches × 8 indicators) sets of test results from the statistical analysis under cross-release prediction scenario between GLANCE-LR and the SOTA approaches. For the ease of observation, we annotate the p-values (<0.001) and the cliff's delta (>0.474). In the view of the statistical test, we make the following observations. GLANCE-LR is significantly better than (i.e., p-value<0.001) NLPs and MITs under most indicators. More specifically, about 89% (57/64) of tests show a significant difference and the effect size of 77% (49/64) tests are large ($|\delta| > 0.474$).

6.2.2 Difference Analysis. Following previous work [15], we use Hit and Over to evaluate the degree of differences of each set of prediction results. More specifically, Hit measures the percentage of true positive instances of a SOTA approach that can also be correctly identified by GLANCE based approaches. Over measures the ratio (in percentage) of the number of buggy lines only correctly identified by GLANCE based approaches to the number of TP buggy lines identified by the SOTA approaches.

$$\text{Hit} = \frac{|TP_{\text{GLANCE}-LR} \cap TP_{\text{SOTA}}|}{|TP_{\text{SOTA}}|}. \tag{6}$$

$$\text{Over} = \frac{|TP_{\text{GLANCE-LR}} - TP_{\text{SOTA}}|}{|TP_{\text{SOTA}}|}. \tag{7}$$

Table 8. The Classification Difference between GLANCE-LR and SOTA Approaches in the Perspective of TP Buggy Lines (Hit = $|TP_{\text{GLANCE}-LR} \cap TP_{\text{SOTA}}| / |TP_{\text{SOTA}}|$; Over = $|TP_{\text{GLANCE-LR}} - TP_{\text{SOTA}}| / |TP_{\text{SOTA}}|$)

	NGram						**LineDP**					
	GLANCE-MD		GLANCE-EA		GLANCE-LR		GLANCE-MD		GLANCE-EA		GLANCE-LR	
	Hit	Over	Hit	Over	Hit	Over	Hit	Over	Hit	Over	Hit	Over
Mean	78%	66%	51%	43%	71%	60%	80%	106%	51%	73%	86%	43%
Median	79%	62%	51%	39%	79%	56%	81%	47%	50%	35%	87%	39%

Table 8 reports the six sets (3 GLANCE based approaches × 2 SOTA approaches) of classification difference between GLANCE based approaches and the representative SOTA approaches (NGram and LineDP) in the perspective of TP buggy line instances. For each set of results, the first and second columns report Hit and Over, respectively. From Table 8, we make the following important observations. First, in terms of Hit, the buggy lines are correctly identified by existing SOTA approaches are highly overlapped with those identified by GLANCE based approaches. For example, on median, 79%, 51%, and 79% of buggy lines detected by NGram can be identified by GLANCE-MD, GLANCE-EA, and GLANCE-LR, respectively. This indicates that, GLANCE based approaches can replace SOTA approaches to some extent. Second, considering Over, GLANCE can identify many new buggy lines that cannot be detected by SOTA approaches. For instance, 47%, 35%, and 39% of new distinct buggy lines missed by LineDP can be identified by GLANCE-MD, GLANCE-EA, and GLANCE-LR, respectively.

Furthermore, Figure 12 reports the distributions of buggy lines identification difference between GLANCE-LR with NGram and LineDP in an individual project, respectively. Compared with NGram and LineDP, GLANCE-LR can identify more than 60% and 75% of the same buggy lines in all releases of 11/19 and 18/19 projects, respectively. Meanwhile, more new buggy lines of code are found by GLANCE-LR to varying degrees.

Conclusion. *In summary, GLANCE based approaches exhibit a competitive classification and ranking performance compared with the existing SOTA CLBI approaches. This is especially true for GLANCE-LR. This means that, the current progress in predicting buggy code lines are not being achieved as it might have been envisaged, if the prediction performance is the ultimate goal.*

6.3 RQ3: How Far Yet to Go in Industry

This section answers the question of, in industry, how far yet to go for CLBI task by comparing the SOTA static analysis tools with simple GLANCE based approaches. More specifically, we present the detailed analysis for performances (Section 6.3.1) and differences (Section 6.3.2) of the prediction results obtained from different approaches, respectively.

6.3.1 Performance Comparison. Table 9 makes a summary in terms of the median and mean values for all indicators. The best performance value of each indicator is marked in **bold** while the worst performance value is marked in underlined. Meanwhile, Figure 13 reports the performance comparison of D2H, R@20%, and HOB between existing SOTA SATs and GLANCE based approaches for each project. The detailed results for each prediction pair are provided online [11]. According to the comparison results based on eight indicators, we make the following observations.

SATs vs. GLANCE. As can be seen, overall, both PMD and CheckStyle achieve a poor performance regardless of whether classification, ranking, or bug level indicators are considered.

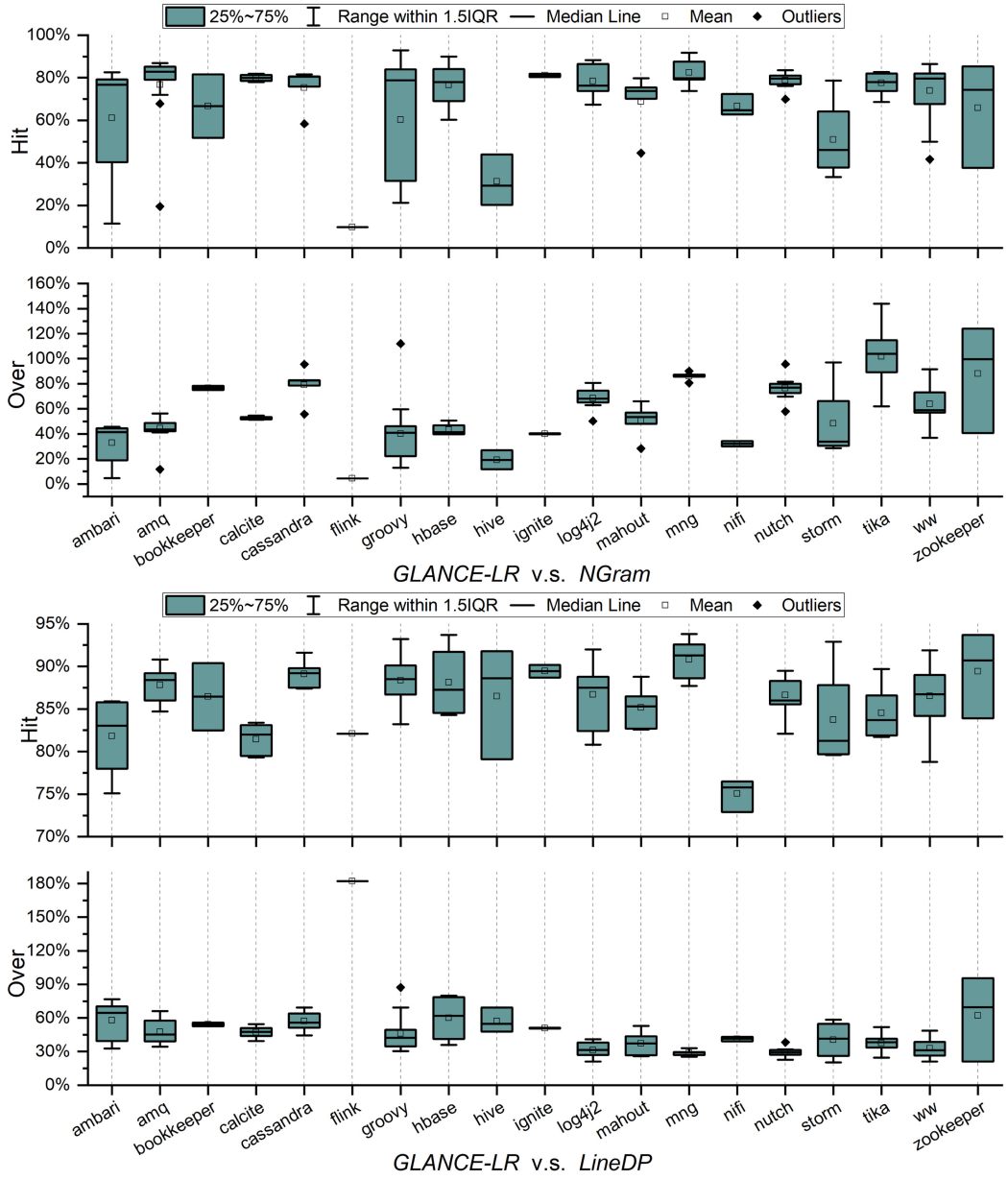

Fig. 12. The difference of buggy lines identification between NGram, LineDP, and GLANCE-LR in individual projects.

Specifically, PMD and CheckStyle can only achieve a low recall rate of 0.185 and 0.289, respectively. In particular, the warning rules of PMD are more conservative, and only the most likely buggy lines of code are reported. However, CheckStyle tends to detect more suspicious buggy lines of code for review and hence it recalls more buggy lines. Note that, regarding the composite indicator D2H, CheckStyle shows a slight advantage. Considering a single project, CheckStyle achieves the best D2H value of 0.395 in project mng while worst D2H value of 0.647 in project flink. The D2H

Table 9. The Mean Value and Median Value of Different CLBI Tools (The Best Performance Value of Each Indicator is Marked in Bold While the Worst Performance Value is Marked in Underlined)

	Approaches	Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
Mean	PMD	0.184	**0.064**	0.012	0.579	0.050	523	0.184	51%
	CheckStyle	0.286	0.188	0.012	0.525	0.026	1123	0.269	62%
	GLANCE-MD	**0.779**	0.413	0.005	0.334	0.077	1256	0.380	**90%**
	GLANCE-EA	0.512	0.245	0.009	0.388	0.064	637	0.418	63%
	GLANCE-LR	0.707	0.182	**0.005**	**0.259**	**0.141**	**197**	**0.627**	83%
Median	PMD	0.185	0.064	0.009	0.578	0.048	111	0.185	51%
	CheckStyle	0.289	0.182	0.010	0.524	0.026	547	0.281	65%
	GLANCE-MD	**0.786**	0.413	0.004	0.331	0.073	264	0.387	**91%**
	GLANCE-EA	0.512	0.246	0.007	0.386	0.060	224	0.425	63%
	GLANCE-LR	0.781	0.175	**0.003**	**0.223**	**0.149**	**26**	**0.655**	88%

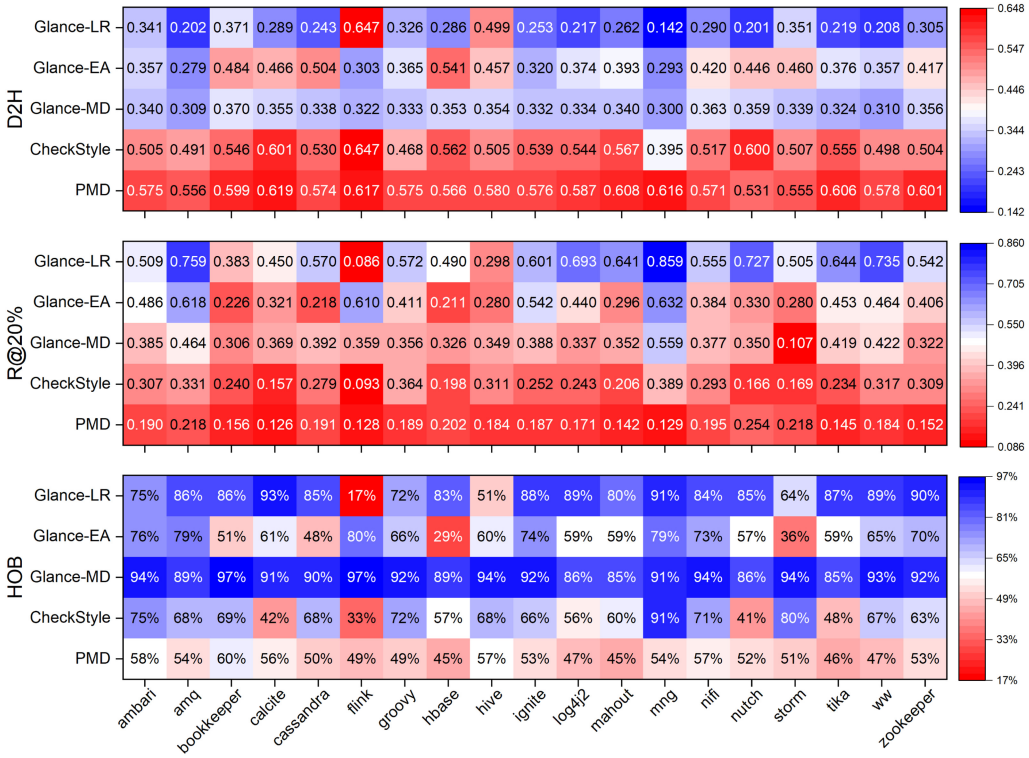

Fig. 13. Performance (D2H, R@20%, and HOB) comparison between SATs and GLANCE for each project.

values of CheckStyle on each project is not very different. It can be easily seen that, both PMD and CheckStyle do not show a performance superior to GLANCE regardless of whether Recall, CoF, D2H, MCC, R@20%, or HOB is considered. Note that, although the FAR values of PMD and CheckStyle are better than GLANCE, it comes at the expense of extremely low recall rates.

Table 10 reports the 16 (2 SOTA tools × 8 indicators) sets of test results from statistical analysis under the cross-release prediction scenario between GLANCE-LR and the SOTA tools. In the view

Table 10. The Results from Wilcoxon-signed Test and Cliff's Delta When Comparing GLANCE-LR with the State-of-the-art Approaches

	Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
BH-corrected	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
p-value	**0.000**	0.481	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
Cliff's delta	0.947 (L)	0.848 (L)	-0.556 (L)	-0.946 (L)	0.863 (L)	-0.343 (M)	0.946 (L)	0.822 (L)
	0.876 (L)	-0.065 (N)	-0.544 (L)	-0.890 (L)	0.942 (L)	-0.680 (L)	0.877 (L)	0.708 (L)

(bold fonts denote that the p-value is less than 0.001) (Abbr. N: Negligible; S: Small; M: Medium; L: Large).

Table 11. The Classification Difference between GLANCE-LR and SATs in the Perspective of TP Buggy Lines (Hit $= |TP_{\text{GLANCE}-LR} \cap TP_{\text{SATs}}| / |TP_{\text{SATs}}|$; Over $= |TP_{\text{GLANCE-LR}} - TP_{\text{SATs}}| / |TP_{\text{SATs}}|$)

	PMD						CheckStyle					
	GLANCE-MD		GLANCE-EA		GLANCE-LR		GLANCE-MD		GLANCE-EA		GLANCE-LR	
	Hit	Over	Hit	Over	Hit	Over	Hit	Over	Hit	Over	Hit	Over
Mean	84%	360%	56%	236%	75%	328%	82%	219%	54%	140%	74%	202%
Median	85%	348%	56%	240%	84%	313%	85%	186%	57%	129%	80%	176%

of the statistical test, we make the following observations. GLANCE-LR is significantly better than (i.e., p-value<0.001) SATs under almost all indicators but FAR. More specifically, about 94% (15/16) of tests show a significant difference and the effect size of 88% (14/16) of tests are large ($|\delta| > 0.474$).

6.3.2 Difference Analysis. Table 11 reports the classification difference between GLANCE based approaches and the representative static analysis tools (PMD and CheckStyle) in the perspective of TP buggy line instances. Figure 14 reports the distributions of buggy lines identification difference between GLANCE-LR with PMD and CheckStyle in individual projects, respectively.

From Table 11, we make the following important observations. First, in terms of Hit, the buggy lines correctly identified by the existing SOTA tools are highly overlapped with those identified by GLANCE based approaches. For example, on median, 85%, 56%, and 84% of buggy lines detected by PMD can be identified by GLANCE-MD, GLANCE-EA, and GLANCE-LR, respectively. This indicates that, GLANCE based approaches can replace the SOTA tools to some extent. Second, considering Over, GLANCE can identify quite a lot of new buggy lines that cannot be detected by the SOTA tools. For instance, 186%, 129%, and 176% new distinct buggy lines missed by CheckStyle can be identified by GLANCE-MD, GLANCE-EA, and GLANCE-LR, respectively. This shows that SATs cannot correctly detect many TP buggy lines, especially for PMD.

Furthermore, Figure 14 reports the distributions difference of buggy lines identified between GLANCE-LR with PMD and CheckStyle in individual projects, respectively. Compared with PMD and CheckStyle, GLANCE-LR can identify more than 60% and 60% of the same buggy lines in all releases of 11/19 and 12/19 projects, respectively. Meanwhile, a considerable ratio of new distinct buggy lines is found by GLANCE-LR to varying degrees.

Conclusion. *In summary, compared with the two popular static analysis tools PMD and CheckStyle, the simple GLANCE approach exhibits great advantages in terms of both classifications, ranking, and bug level performance. This means that, SATs performs badly in detecting buggy code lines, and there is still a long way to go to really promote the effectiveness of SATs in industry, if the prediction performance is the ultimate goal.*

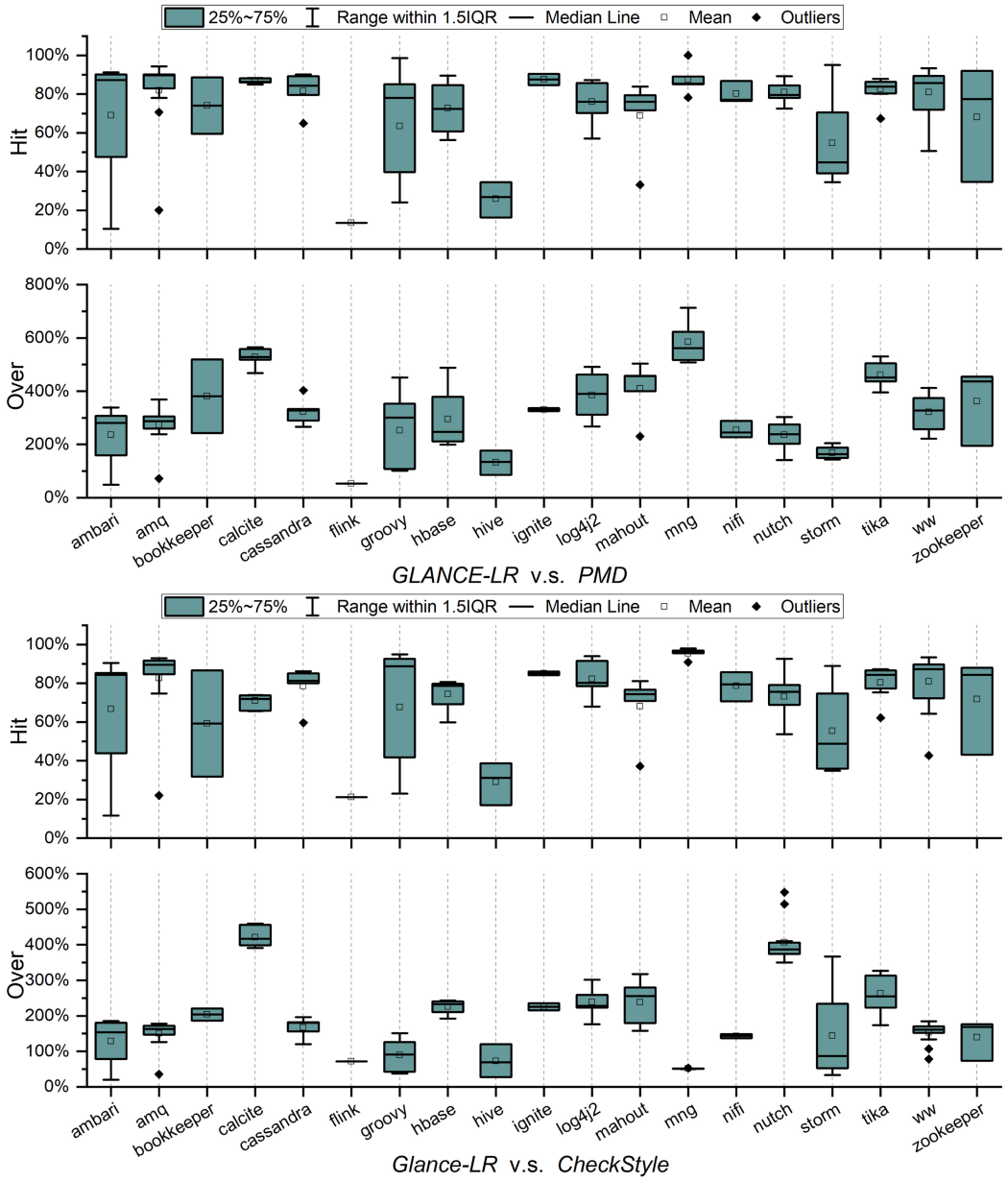

Fig. 14. The difference of buggy lines identification between PMD/CheckStyle and GLANCE-LR in individual projects.

7 DISCUSSION

In Section 6, we compare GLANCE with the existing SOTA approaches and tools to gain an understanding of the real progress in code-line-level bugginess identification. The used GLANCE is very simple but enough for our purpose in this study. In this section, we conduct additional experiments to provide a more comprehensive understanding of the characteristics of GLANCE, thus facilitating researchers and practitioners to improve and utilize GLANCE in the future.

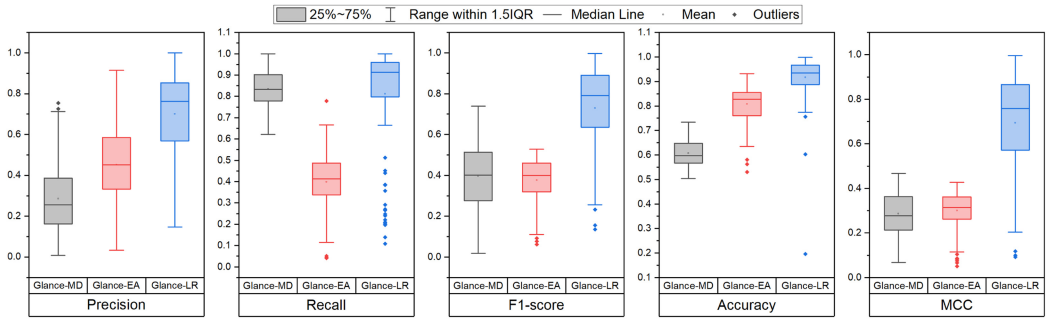

Fig. 15. The classification performance comparison between different file-level classifiers ($Precision = TP/(TP + FP)$, $Recall = TP/(TP + FN)$, F1-Score = $2 \times Precision \times Recall/(Precision + Recall)$, $Accuracy = (TP + TN)/(TP + TN + FP + FN)$).

7.1 How do the Configuration Factors Affect the Performance of GLANCE?

According to Section 4, there are three important factors in GLANCE that play a role in identifying buggy lines, namely file-level classifier, effort-aware threshold, and code line level metrics (i.e., NT, NFC, and CE). In this section, we will compare and discuss the influences of different factors on the performance of GLANCE.

7.1.1 The Influence of File-level Classifiers. Figure 15 reports the representative classification performance (see ref. [87]) comparison between different file-level classifiers. As can be seen, GLANCE with Logistic Regression classifier achieves the best classification performance, followed by GLANCE with EA classifier. More specifically, GLANCE-LR has an average value of 0.73 and 0.69 in terms of F1-Score and MCC, respectively. Note that, since EA classifier only correctly recalls 40% defective files on average, the overall Recall (Figure 15) of GLANCE-EA is lower than the other two baselines. Overall, Logistic Regression is the best classifier in terms of classification performance.

As is known, one main goal of the research on defect prediction is to reduce the code reviewing effort so that SQA teams can locate the buggy code snippets earlier. To this end, Figure 16 reports the reviewing effort comparison between different file-level classifiers. On average, in the rank lists provided by GLANCE-MD, GLANCE-EA, and GLANCE-LR, developers need to review about 40%, 25%, and 18.9% of the total code lines in a project, respectively. Note that, the review effort of the two unsupervised classifiers are relatively stable, while that of LR classifier varies greatly in different projects (2.3%–48.7%). According to existing studies [31, 83], the huge amount of reviewing effort will reduce the user experience of developers, and even make them give up using such a model in practice. Therefore, considering the reviewing effort, GLANCE-LR is recommended when the training data is available. Otherwise, GLANCE-EA is the best choice as the baseline.

7.1.2 The Influence of Line-level Threshold. Figure 17 reports the detailed distributions of three representative indicators D2H, R@20%, and HOB of GLANCE-LR under different threshold values ranging from 5%–100%. For the ease of observation, we connect the obtained median and mean value under each threshold by blue and red lines, respectively. As can be seen, based on the ranks of GLANCE-LR, more and more buggy can be revealed with the increase of the threshold value. Meanwhile, the D2H value is getting better and better. The average Recall increase greatly before the scale of threshold value reaches 50%. However, when the threshold value exceeds 50%, the performance improvements slow down and even stop, which is very uneconomical. In this case,

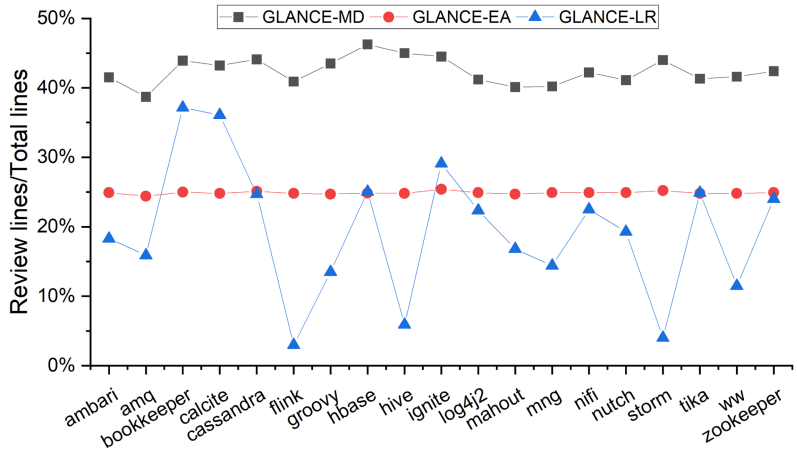

Fig. 16. The review effort (measured by SLOC) comparison between different file-level classifiers.

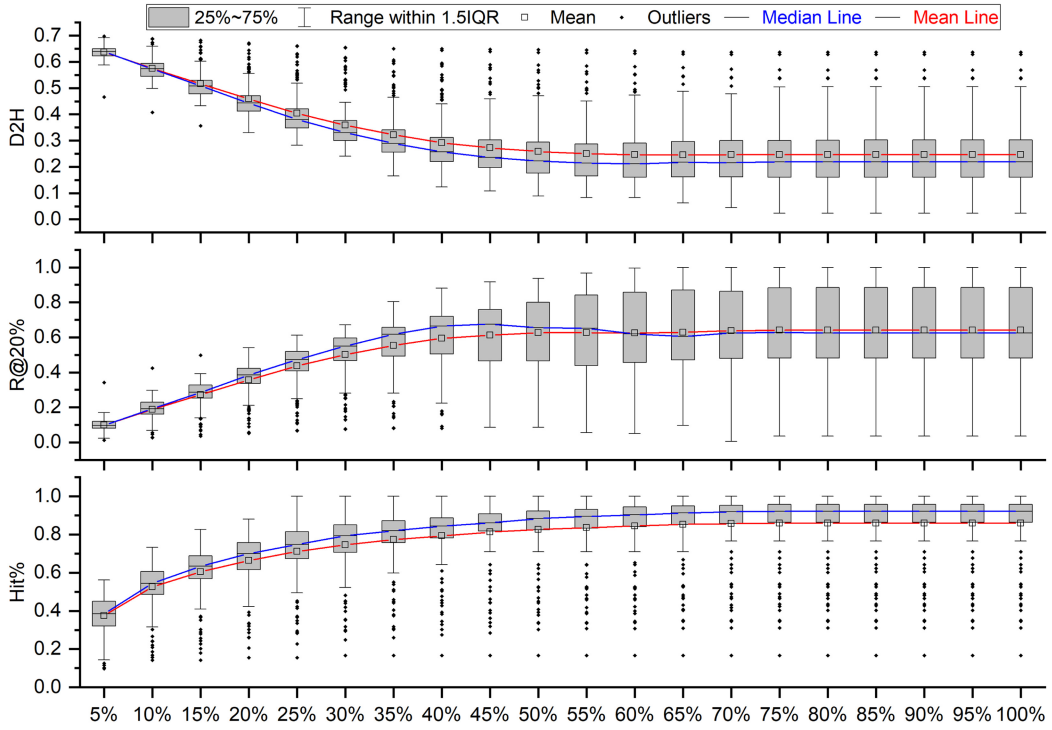

Fig. 17. The distributions of performance trend of GLANCE-LR under different threshold values in terms of D2H, R@20%, and HOB, respectively.

much effort will be spent if developers continue to inspect the rest of the codes. Further, Table 12 summarizes the performance trend of GLANCE-LR under different threshold values ranging from 5%–50% in terms of all indicators. In most cases, a threshold value of 40%–50% can make GLANCE play a better performance. In summary, the experimental results indicate that, 50% is a proper value to cut off the process of reviewing code lines.

Table 12. The Summary of Performance Trend of GLANCE-LR under Different Threshold Values in Terms of All Indicators

Line-level threshold		5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
Median	Recall	0.099	0.188	0.273	0.357	0.438	0.507	0.569	0.624	0.667	**0.707**
	FAR	**0.017**	0.035	0.053	0.071	0.09	0.108	0.126	0.145	0.163	0.182
	CoF	0.012	0.011	0.01	0.009	0.008	0.008	0.007	0.006	**0.005**	**0.005**
	D2H	0.637	0.575	0.516	0.458	0.404	0.359	0.322	0.292	0.273	**0.259**
	MCC	0.061	0.082	0.099	0.113	0.125	0.133	0.138	**0.141**	**0.141**	**0.141**
	IFA	82	**51**	96	125	75	104	197	179	202	198
	R@20%	0.099	0.188	0.273	0.357	0.438	0.502	0.553	0.595	0.612	**0.627**
	HOB	38%	53%	61%	66%	71%	75%	77%	79%	81%	**83%**
Mean	Recall	0.097	0.192	0.286	0.384	0.472	0.558	0.628	0.691	0.736	**0.781**
	FAR	**0.017**	0.034	0.051	0.069	0.086	0.104	0.122	0.139	0.157	0.175
	CoF	0.01	0.009	0.008	0.007	0.006	0.005	0.005	0.004	0.004	**0.003**
	D2H	0.639	0.573	0.508	0.443	0.38	0.33	0.29	0.257	0.236	**0.223**
	MCC	0.058	0.083	0.105	0.118	0.136	0.143	0.147	**0.152**	0.15	0.149
	IFA	**21**	**21**	27	29	24	36	53	33	41	26
	R@20%	0.097	0.192	0.286	0.384	0.472	0.55	0.617	0.664	0.675	**0.655**
	HOB	38%	54%	63%	70%	75%	79%	82%	84%	86%	**88%**

Table 13. The Summary of Performance Comparison of GLANCE-LR with NT and NFC

Indicators		Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
Mean	GLANCE-LR	**0.707**	**0.182**	**0.005**	**0.259**	**0.141**	**197**	**0.627**	**83%**
	NT	0.623	0.183	0.007	0.306	0.117	312	0.553	79%
	NFC	0.655	0.183	0.006	0.287	0.126	234	0.583	81%
Median	GLANCE-LR	**0.781**	**0.175**	**0.003**	**0.223**	**0.149**	**26**	**0.655**	**88%**
	NT	0.657	**0.175**	0.004	0.288	0.122	58	0.586	84%
	NFC	0.721	**0.175**	0.004	0.253	0.131	**26**	0.603	86%

7.1.3 The Influence of Code Line Level Metrics. To investigate the influence of NT and NFC metrics, we calculate NT and NFC as DefectProne ssScore separately to rank code lines. Table 13 reports the summary of performance comparison of GLANCE-LR with NT and NFC. As can be seen, regardless of mean or median, the metric NFC has a better identification performance compared with the metric NT in terms of all indicators. For instance, on median, the model built only on NFC can recall 9.74% more buggy lines than that of models built on metric NT. This indicates that many defects in the code are indeed related to function calls. In addition, compared with models leveraging only one metric, GLANCE-LR produces a more effective classification result. More specifically, on median, GLANCE-LR has a median improvement of 18.9% and 8.3% in terms of Recall. This means that, both metrics NT and NFC promote each other in ranking buggy lines. Figure 18 reports the influence of CE metric. As can be seen, when leveraging the CE metric to re-sort the ranked list, on average, there is a slight improvement regardless of whether GLANCE-LR, NT, or NFC is considered. Although the improvement is insignificant, it does not mean that the metric CE is useless. In fact, GLANCE only uses metric CE in a simple way. There is still room to explore the usage of this metric. In addition, considering that the cost of re-sort operation based on CE is very low, it is still reasonable to incorporate it to the GLANCE approach.

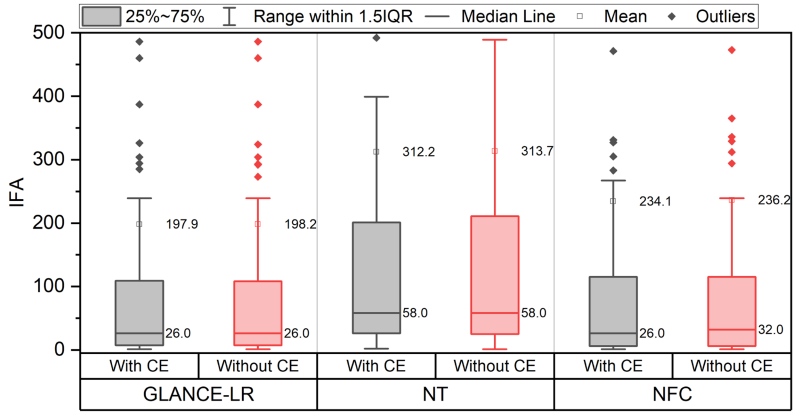

Fig. 18. The distributions of IFA values of GLANCE-LR, NT, and NFC with and without considering metric CE, respectively.

Conclusion. *In summary, to reduce a considerable amount of review effort and improve the precision when identifying defective files, the Logistic Regression is the preferred classifier for GLANCE. It is appropriate to use 50% as the default line level threshold value for GLANCE. Meanwhile, all three code line level metrics should be used to build GLANCE in practice.*

7.2 Which should Researchers Focus on Most If They Want to Boost CLBI based on GLANCE?

As an easy-to-use baseline for CLBI tasks, the performance of GLANCE can be considered the lower performance bound of CLBI approaches to a certain extent when identifying buggy code lines. According to the results of RQ1 in Section 6, as can be seen, although GLANCE can recall a large proportion of buggy lines (e.g., on median, the Recall of GLANCE-LR is 70.3%), the values of classification indicator FAR and ranking indicator IFA are both not satisfactory. This means that, there are many clean lines are mistakenly predicted as buggy ones so that developers still need waste much effort to review these false positives. In this discussion, we will analyze the reason and point out a feasible research focus that motivates how to improve the performance of CLBI tasks based on GLANCE.

Generally, GLANCE goes through two stages: file-level prediction and line-level prediction. In this framework, the performance of GLANCE will be affected by both two stages. For example, Figure 19 reports the distributions of the real buggy files (TP) and the clean files (FP) in the set of defective files predicted by the file-level prediction model EAMD in GLANCE. As can be seen, in most projects (12 out of 19), more than 50% of clean files are mistakenly predicted as buggy. In particular, 91% defective files in the Storm project identified by EA are false positives (FP). The existence of these FP files will increase the values of IFA greatly. For example, if a clean file with 500 lines of code is mistakenly predicted as defective and the lines in the file are ranked at the top of the total list, the developers need to review 250 (the default threshold is 50%) code lines. In this situation, the value of IFA is at least 250. However, this IFA value should be reduced by at least 250 if the clean file is correctly classified because there is no need for developers to review any lines in the file. Based on the above analysis, we assume that the performance of GLANCE is competitive when the target files can be confirmed to contain defects. To verify this conjecture, we first design a theoretically optimal GLANCE based model, called GLANCE-BEST, whose file-level

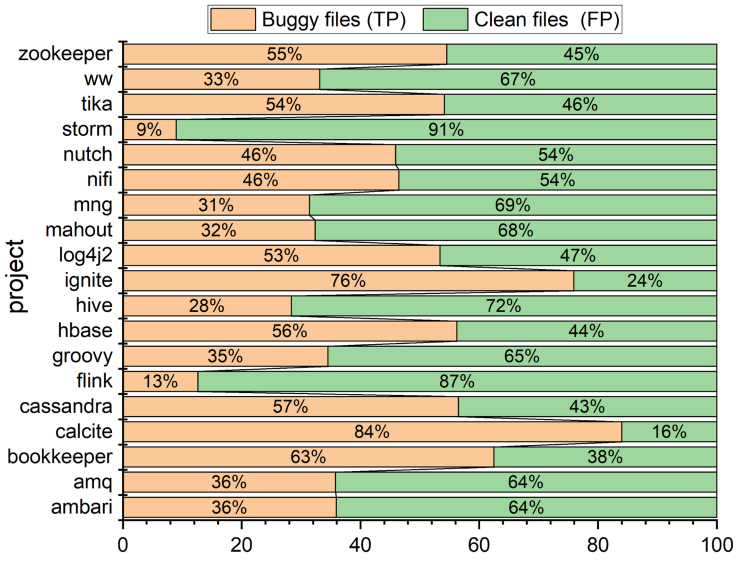

Fig. 19. The distributions of the real buggy files (TP) and the clean files (FP) in the set of defective files predicted by the file-level prediction model in GLANCE-EA.

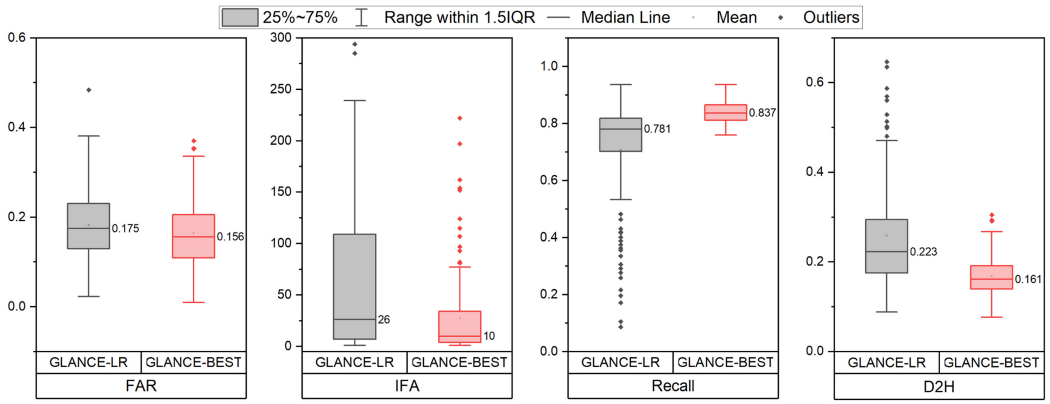

Fig. 20. The main performance comparison between GLANCE-LR and GLANCE-BEST.

classification results are entirely correct. In other words, we assign a correct label (known from the collected dataset) for each file in the test release directly. Obviously, all false positive (FP) files are eliminated in GLANCE-BEST. Then, we evaluate and compare the performance between GLANCE-LR (the optimal model in this study) and GLANCE-BEST (the optimal model in theory), and report the comparison in Figure 20. As can be seen, there is a non-negligible space for improving the performance of GLANCE based approaches. More specifically, after removing the impact of mis-identified clean files (i.e., FP defective files), GLANCE based approaches can be greatly improved regardless of which indicator is considered. For example, on median, GLANCE-BEST has a decrease of 10.9% in terms of FAR. Meanwhile, half of IFA values are less than 10 lines of code, which is acceptable in the process of code review. In addition, the other important classification indicators such as Recall and B2H reach a better level.

Table 14. The Median Performance Comparison of SOTA Approaches based on Default Setting and Same Defective Files

Approach	NGram		NGram-C		PMD		CheckStyle		GLANCE-LR
Setting	*Default*	*Same*	*Default*	*Same*	*Default*	*Same*	*Default*	*Same*	-
Recall	**0.555**	0.492	**0.278**	0.230	**0.185**	0.157	**0.289**	0.225	0.781
FAR	0.308	**0.121**	0.170	**0.074**	0.064	**0.026**	0.182	**0.066**	0.175
CE	0.007	**0.006**	0.009	0.009	0.009	0.009	0.010	**0.009**	0.003
D2H	0.380	**0.370**	**0.527**	0.549	**0.578**	0.597	**0.524**	0.551	0.223
MCC	0.053	**0.109**	0.027	**0.058**	0.048	**0.078**	0.026	**0.065**	0.149
IFA	233	**62**	288	**96**	111	**49**	547	**89**	26
R@20%	0.432	**0.482**	**0.267**	0.23	**0.185**	0.157	**0.281**	0.225	0.655
HOB	**81%**	73%	**58%**	49%	**51%**	46%	**65%**	55%	88%

Conclusion. *As can be seen, the underlying performances of GLANCE are hindered by the of the file-level prediction results. Therefore, in future studies, to make GLANCE more effective, it is highly recommended to build more accurate file-level defect prediction models so that as many FP buggy files as possible are eliminated at the file-level prediction stage.*

7.3 Does GLANCE-LR still Maintain Advantages When Evaluating It with the SOTA Approaches on the Exact Same Defective Files?

According to Section 6, GLANCE-LR shows more advantages than SOTA approaches. However, considering details of different approach, the buggy lines identified by each approach are from different defective file set (except for TMI-LR and LineDP). It is unknown whether GLANCE-LR can maintain the advantage when evaluating it and SOTA approaches on the exact same defective file set. Note that, other four MITs cannot produce exact same defective files as GLANCE-LR due to different file-level classifiers. Therefore, we only compare GLANCE-LR with NLPs and SATs in this discussion.

To this end, we consider the defective files predicted by GLANCE-LR as the exact same defective files. Based on the predicted buggy lines in these files, we re-evaluate the performance of NLPs and SATs. Table 14 reports the median performance comparisons results. In particular, Default denotes the original performance of SOTA approaches (reported in Section 6) while Same denotes the corresponding re-evaluated performance. The better value between Default and Same groups are marked as bold. As can be seen, for all SOTA approaches, the Recall and HOB values of Same group have decrease slightly. The reason is that some true buggy lines not in the exact same files are missed. Meanwhile, the results of the Same group (e.g., NGram-C) have an obvious lower FAR (i.e., 0.074) value than that (i.e., 0.170) of Default group. Similar phenomena can be observed for IFA values. For example, on median, a developer needs to review 111 and 49 code lines before finding the first buggy lines based on the ranks provided by PMD evaluated on Default and Same groups, respectively. This means that a large proportion of clean lines are excluded with the help of LR classifier. In terms of median D2H, the results of the Same group do not show obvious advantages, or even worse than that of Default group for most approaches. Overall, more clean lines can be excluded by Same group results of all SOTA approaches. Yet, it is worth noting that, compared with SOTA approaches re-evaluated on the exact same defective files, GLANCE-LR still achieve better performance values in term of most indicators (except for FAR). In other words, the advantages of GLANCE-LR still exist when comparing it with SOTA approaches on the exact same defective files.

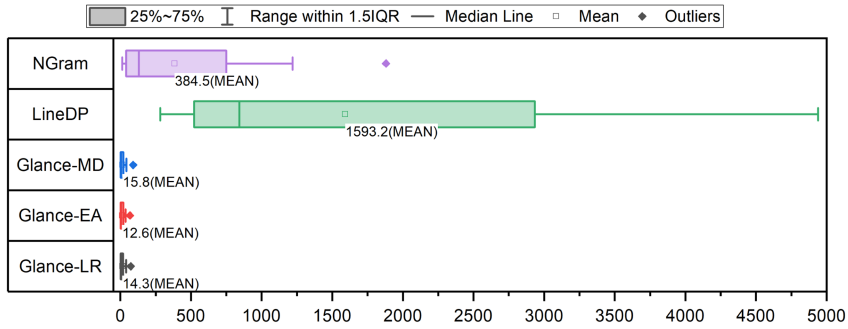

Fig. 21. The comparison of overall (building + prediction) time (Unit: second) between NGram, LineDP, and GLANCE.

Conclusion. *In summary, when evaluating four SOTA approaches based on exact the same files that identified by GLANCE-LR, more clean lines (refer to FAR and IFA) are excluded from the prediction results. Meanwhile, GLANCEN-LR still performs better than SOTA approaches considering most indicators.*

7.4 Is GLANCE a Cost-effective Approach Compared with the SOTA Approaches when Identifying Buggy Lines?

Following the result of Section 6.2, although GLANCE shows a competitive classification and ranking performance compared with the existing SOTA CLBI approaches (i.e., NLPs and MITs), it is unknown whether GLANCE is more cost-effective. As stated in Section 2.2, the main challenge (C2) for SOTA approaches is their high application cost (refers to the execution time cost), which may lead to a poor user experience. In this discussion, we will empirically investigate the degree to which how much execution time cost is necessary to complete the CLBI task for different approaches. In particular, the NGram and LineDP is selected because they are the most effective approaches in their respective categories.

The execution time is collected based on a standard computing machine with an Intel Core i7-6700 3.4GHz and 16GB of RAM. Under CR setting applied in our work, we calculate a total of 142 pairs of time cost for each approach. Figure 21 presents the distribution of overall (including model building and prediction) execution time for each project. On average, in order to complete the identification task on a project, NGram need take 384.5 seconds and LineDP need take 1,593.2 seconds, which are both much higher than the cost of GLANCE (lower than 20 seconds). Table 15 reports detailed execution time in terms of model building and prediction time, respectively. For the ease of presentation, we calculate the average time for each project and then get 19 groups of results. As can be seen, NGram need take a lot of time (381.6 seconds on average) to build the language model corpus while it can quickly output prediction results. Instead, because LineDP need first make explanations and then rank buggy lines for each test file, it takes more time (1,583.5 seconds on average) in the prediction stage, which is cost-ineffective when applying it in practice.

Conclusion. *Overall, GLANCE based approaches are cost-effective compared with SOTA CLBI approaches when identifying buggy code lines. In particular, NGram (belongs to NLPs) need take much to build models while LineDP (belongs to MITs) is cost-ineffective when applying it to make predictions.*

Table 15. The Details of Building and Prediction Time (Unit: Second) between NGram, LineDP, and GLANCE

Project	Model building time (Unit: s)			Model prediction time (Unit: s)		
	NGram	LineDP	GLANCE-LR	NGram	LineDP	GLANCE-LR
ambari	745.8	17.9	14.6	4.4	2,915.7	3.8
amq	274.1	8.7	8.5	2.5	1,293.9	2.6
bookkeeper	13.6	0.3	0.5	0.3	842.2	0.9
calcite	269.5	7.1	8.1	2.2	3,303.9	4.3
cassandra	68.7	2.0	2.5	0.7	1,172.2	1.9
flink	1,211.2	30.9	36.3	7.9	642.8	5.1
groovy	142.9	3.7	3.8	1.2	1,002.4	2.4
hbase	295	8.3	10.3	4.2	4,164.2	6.3
hive	1,867.7	51.6	64.9	12.7	1,979.6	8.6
ignite	899.8	24.9	29.3	6.7	4,914.2	7.7
log4j2	58.9	1.5	1.8	0.6	769	1.5
mahout	92.4	2.1	2.3	0.8	765.3	1.7
mng	40.8	1.1	1.3	0.5	353.4	1.0
nifi	960.7	16.7	19.4	7.9	3,629.5	6.3
nutch	16.5	0.4	0.4	0.2	360.4	0.7
storm	130.8	3.2	3.8	1.2	279.3	1.4
tika	27.2	0.6	0.7	0.3	521.5	0.9
ww	115.3	2.9	3.0	0.9	675.7	1.6
zookeeper	19.5	0.7	0.7	0.3	500.6	0.9
Average	381.6	9.7	11.2	2.9	1,583.5	3.1

7.5 Will the Noisy Label in Our Dataset have a Substantial Impact on Our Experimental Results?

As admitted in the previous works [7, 60, 73], some wrongly labeled code lines (i.e., the noisy label) may be generated by the data collection process introduced in Section 5.2.1. This can reduce the quality of the studied dataset in Section 5.2.2 and hence may affect the experimental results reported in Section 6. In the following, we will investigate the degree to which noisy label may affect our experimental results via three steps: noisy label analysis, sensitivity analysis, and extended analysis.

7.5.1 Noisy Label Analysis. In the first step, we summarize the main categories of the potential noisy labels in our dataset, which is shown in Table 16. The first and second column respectively report the categories and their corresponding concrete noisy types. The third column reports whether there is a sound solution to reduce a certain type of noisy label in the current literature. The last column reports the solution taken in this article. Specifically, they can be summarized as the following two categories:

(1) ***False negative*** *refers to the actual buggy lines that are wrongly labeled as clean.* This kind of noise is caused by the missed bugs. First, it is possible that some code lines that cause the bugs but not changed by the BFC are mistakenly considered as clean. In the literature, there is no good solution for the time being. Second, the false negatives happen with the bugs not reported, which is unavoidable in all data mining-based studies because defects/bugs in any software project are discovered in a continuous process. We have extracted all available bug data in the project for research in the development period. Therefore, the impact of the two kinds of false negatives

Table 16. The Summary of Noisy Labels in Our Dataset

Category	Type	Can it be reduced in the literature?	How to reduce them in this study?
False negatives	The buggy lines not changed by BFCs	No	-
	The buggy lines not reported by developers/users	No	-
	The buggy lines not mined by scripts	Yes	Select proper projects
False positives	The clean lines belong to comment code lines and blank lines	Yes	Apply regular expression
	The clean lines belong to refactoring lines	Yes	Matching keywords in commit log

cannot be measured in our study. Third, the false negatives happen with bugs not mined. In our study, we select the popular open-source **apache software foundation** (**ASF**) projects whose evolution process is well organized.[11] These projects maintain a clear link between BFC and the issue report so that all issues can be mined once a BFC commit log message contains the issue id. Therefore, this kind of false negatives should not have a large impact on our dataset.

(2) ***False positive*** *refers to the actual clean lines that are wrongly labeled as buggy.* This kind of noise is mainly caused by irrelevant lines and refactoring lines. Specifically, the former refers to the comment code lines and blank lines that are unrelated to defects. Following prior works [7, 60, 73], these noisy code lines can be easily filtered out from the set of buggy lines by applying simple regular expression, which has also been done for our dataset. The latter refers to the code lines that aim at adjusting the structure of source code instead of fixing the software bugs. Note that, the refactoring code lines mainly exist in the refactoring BFCs whose commit log message is usually marked by the keywords of "Refactor XXX", which has been eliminated by checking the log messages of studied BFCs, just as prior works [7, 60, 73].

In summary, three out of five types of noisy labels in our dataset have been removed in proper ways. In this sense, we have reduced the proportion of noisy labels in the dataset as much as possible. Note that, the reality is that the noise in terms of both false positives and false negatives can be avoided only by manually checking all data instances, which is usually unrealistic due to huge effort and unfamiliar context for specific software project. Therefore, following to the prior works [60, 64, 73], our study applies the same compromise scheme (i.e., the process described in Section 5.2.1) to collect the defect dataset and to analyze the experimental results.

7.5.2 Sensitivity Analysis. Although we put a lot of effort to reduce noisy labels, our dataset cannot be guaranteed to be completely free of noise. More specifically, other small parts of refactoring code lines may also appear in the non-refactoring BFCs. Very recently, Silva et al., [63] developed a state-of-the-art multi-language refactoring detection tool RefDiff, which mines refactoring code in the commit history of git repositories. To check the degree of this kind of refactoring lines, we leverage RefDiff [63] to detect the ratio of refactoring lines among BFC lines in each studied project. As shown in Figure 22, the proportion of refactoring code lines in our dataset is very low. On average, only 2% of buggy lines belong to refactoring code lines, and the proportion shall not exceed 5%. The low degree of the refactoring lines in the non-refactoring BFCs means that it is unlikely to have a significant impact on the experimental results.

We next conduct a sensitivity analysis to measure the degree of impact from refactoring lines quantitatively. Specifically, we first apply RefDiff tool [63] to detect all refactoring lines in the set of buggy code lines. After that, we correct the label of these code lines (i.e., false positives) from "buggy" to "clean", and we can obtain a new dataset with less noise. Then, we re-conduct experiments in Sections 6.2 and 6.3 and acquire new identification performance of all CLBI approaches

[11] https://camel.apache.org/community/contributing/.

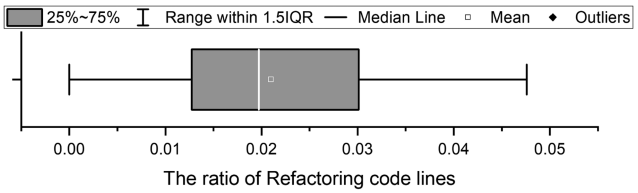

Fig. 22. The distribution of ratios of refactoring code lines detected by RefDiff among all BFC lines.

that are evaluated based on the new dataset. Finally, we subtract the original performance (shown in Table 6 or Table 9) from the new performance to investigate the sensitivity of each approach to the noisy labels in studied dataset. Table 17 reports the mean and median performance difference for each CLBI approaches. As can be seen, most of the difference values are lower than 0, indicating that the performance of all approaches on the noiseless dataset has slightly decreased compared with the original performance. In other words, the performance of all approaches measured on the original dataset is slightly exaggerated. Nevertheless, considering the most indicators, GLANCE based approaches still exhibit a competitive classification and ranking performance compared with the existing SOTA CLBI approaches.[12] This means that the noisy labels in our dataset do not have a large impact on our experimental results.

7.5.3 Extended Analysis. Recently, to fill the gap of the low-quality dataset in the just-in-time defect prediction (short for JIT-DP) and defect localization (short for JIT-DL) community, Ni et al., [51] built a large-scale manually labeled line-level defect dataset called JIT-Defects4J based on the LLTC4J, a dataset collected by Herbold et al [21]. Although the granularity of the high-level labels of Ni et al.'s dataset [51] is at the commit-level, not at the file-level as our dataset, the two datasets (Ni et al.'s and ours) are used to study a very similar application scenario. That is, first to predict high-level buggy modules, and then to identify fine-grained buggy lines from large modules. We next conduct an extended experiment on the JIT-Defects4J dataset to further investigate to what extent our conclusion (i.e., GLANCE framework is a strong baseline) is impacted by the noisy labels. Table 18 reports the statistical information of the selected 9 projects from JIT-Defects4J dataset.[13] Note that, there is no version information for each project in JIT-Defects4J. We apply the cross-project defect prediction (short for CPDP) setting [87] to simulate a real application scenario. Specifically, one target project is used as the test set, and the rest 8 projects are used in turn as the training set to build a model for prediction. In this experiment, we calculate the average value of 8 prediction results for each target project to evaluate each approach.

As shown in Section 6, LineDP is the SOTA CLBI approach in the literature. Figure 23 reports the performance comparison between LineDP and three GLANCE approaches in terms of the comprehensive indicator D2H (the lower the better) based on JIT-Defect4J dataset. As can be seen, the D2H performance value of LineDP is greater than 0.5 on all projects, with an average value of 0.642, which means that LineDP can hardly work on the JIT-Defects4J dataset. Meanwhile, GLANCE-MD, GLANCE-EA, and GLANCE-LR can achieve an average D2H value of 0.468, 0.580, and 0.668, respectively. Compared with LineDP, GLANCE-MD, and GLANCE-EA perform better while GLANCE-LR performs worse. By analyzing the prediction results, we found that the LR classifier used by LineDP and GLANCE-LR has poor classification performance at the file-level prediction phase on the new dataset, resulting in the two approaches cannot effectively identify defective lines. Note

[12]The detailed experimental results can be found in the replication package.

[13]To obtain more reliable experimental results, we only select the projects whose number of buggy commits is larger than 100 from the Ni et al.'s dataset [51].

Table 17. The Mean and Median Performance Difference for Each CLBI Approach (Difference Value = New Performance Values - Performance Values in Table 6 or Table 9)

	Approaches	Recall	FAR	CoF	D2H	MCC	IFA	R@20%	HOB
Mean	NGram	-0.003	-0.003	0	0	0	32	0.003	0
	NGram-C	-0.001	-0.002	0	0.001	0	0	0	0
	TMI-LR	0.003	0.001	0	-0.002	0.001	6	0.002	0.007
	TMI-MNB	-0.006	0	0.001	0.004	-0.002	-24	-0.006	0
	TMI-SVM	-0.003	0	0	0.002	-0.003	0	-0.003	0.001
	TMI-DT	0.002	0.002	0	-0.001	0	9	0.003	0.005
	TMI-RF	0.004	0	0	-0.003	0.001	20	0.005	0.005
	LineDP	-0.005	0	0	0.003	-0.001	21	-0.007	0.001
	PMD	-0.03	-0.036	0	0.02	0.027	-430	-0.03	-0.063
	CheckStyle	-0.047	-0.117	-0.001	0.016	0.039	-797	-0.03	-0.082
	GLANCE-MD	-0.004	0	0.001	0.001	0	-3	-0.001	0
	GLANCE-EA	-0.003	0	0	0.002	0	-12	-0.002	0
	GLANCE-LR	-0.004	0	0	0.002	0	-31	-0.004	0.002
Medain	NGram	-0.006	-0.004	0	0.004	0.001	24	0.002	0
	NGram-C	0.002	-0.007	0	0.001	0.001	-1	0	0
	TMI-LR	0.002	-0.001	-0.001	-0.002	-0.003	-3	0.002	0.006
	TMI-MNB	-0.005	0	0	0.001	-0.004	-10	-0.004	-0.001
	TMI-SVM	0.001	0.001	0	0	-0.001	0	0.001	0.006
	TMI-DT	0	0.006	0	-0.001	0.002	-10	0	0
	TMI-RF	0.01	-0.003	0	-0.007	0.008	3	0.006	0.005
	LineDP	0.003	-0.005	0	0.002	-0.001	0	-0.007	0.001
	PMD	-0.032	-0.038	0	0.021	0.028	-62	-0.032	-0.048
	CheckStyle	-0.066	-0.117	-0.001	0.027	0.041	-421	-0.058	-0.101
	GLANCE-MD	-0.006	-0.001	0	0.001	-0.001	-12	-0.005	0
	GLANCE-EA	-0.006	0	-0.001	0.004	-0.001	0	-0.003	0
	GLANCE-LR	-0.002	-0.004	0	0.001	-0.002	6	0.008	0.003

Table 18. The Statistical Information of JIT-Defects4J Dataset

Project	Commit-level			Line-level (LOC)		
	#Files	#Buggy	%Buggy	#Lines	#Buggy	%Buggy
ant-ivy	5,042	490	9.72%	60,279	1,650	9.72%
commons-compress	4,338	209	4.82%	46,219	627	4.82%
commons-configuration	4,752	182	3.83%	66,767	650	3.83%
commons-lang	8,088	158	1.95%	95,315	563	1.95%
commons-math	16,506	432	2.62%	213,191	3,046	2.62%
commons-net	3,747	150	4.00%	29,278	441	4.00%
commons-vfs	4,297	158	3.68%	32,493	384	3.68%
giraph	4,753	359	7.55%	79,039	1,904	7.55%
parquet-mr	4,693	233	4.96%	68,023	729	4.96%
Average	6,246	263	4.79%	76,733	1,110	1.47%

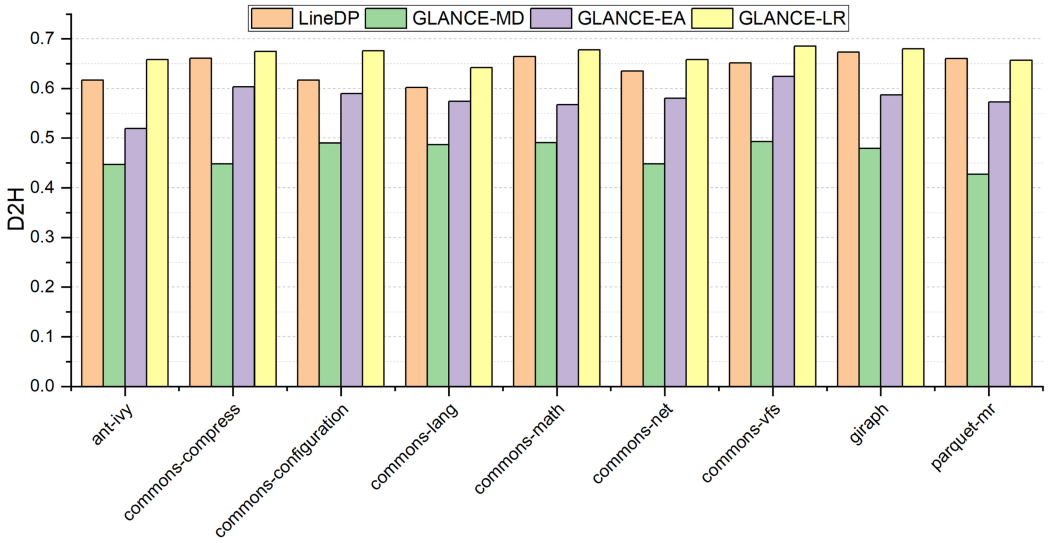

Fig. 23. The performance comparison between LineDP and GLANCE approaches in terms of D2H on JIT-Defects4J.

that the unsupervised file-level models (i.e., MD and EA) are relatively less affected by the extended dataset, especially for MD. On the whole, the CLBI performance of both the SOTA approach (i.e., LineDP) and the GLANCE based approaches are affected by the dataset label. However, it is worth mentioning that the proposed GLANCE framework still performs better than the SOTA approach.

> ***Conclusion.*** *In summary, the noisy label analysis shows that there are five types of noisy labels in our dataset, three of which have been reduced in our study in proper ways. The sensitivity analysis results show that the remaining noisy labels in our dataset do not have a large impact on our experimental results. The extended analysis results indicate that GLANCE still shows advantages compared with the SOTA approaches (i.e., LineDP) based on the latest manually labeled JIT-Defect4J dataset.*

8 IMPLICATIONS FOR RESEARCHERS, PRACTITIONERS, AND TOOL PROVIDERS

This study has important implications for researchers, practitioners, and tool providers.

8.1 For Researchers

We contribute a set of simple yet strong baseline GLANCE in the field of CLBI. In the future, if a new CLBI approach is proposed, it should be compared against GLANCE to demonstrate its practical value. This will help our community develop more effective approaches for identifying buggy code lines. The detailed implications are as follows:

(1) **Our work elaborates the importance of simple code-line-level metrics in CLBI tasks.** As can be seen, although GLANCE only leverages several code-line-level metrics to calculate the defect-proneness of each code line, it shows a competitive performance compared with the SOTA approaches. This means that these code-line level metrics are useful for indicating the defect-proneness of code lines. Therefore, it is recommended for researchers to pay attention to these metrics when building new effective CLBI prediction models. In addition, this study is just a preliminary exploration of the effectiveness of code-line level metrics in the CLBI task. In fact, this

study only focuses on the control statements and complex statements in a source file. Many other code-line level characteristics can also be mined to boost the performance of GLANCE in future studies.

(2) **Our work highlights an important software engineering research viewpoint that simple solutions may be better than sophisticated ones.** Actually, various studies in the field of software engineering [13, 20, 40, 41, 43, 81, 83, 87] have already stressed a similar standpoint. For instance, our previous work [87] in cross-project defect prediction indicated that, Manual-Down, a very simple module size model, performed similarly to or even better than almost all the state-of-the-art (complex) supervised models. In addition, the latest SATD identification approach, MAT [15], also showed that a simple approach could be both effective and practical. Liu et al [41] and Majumder et al., [43] reported that simple models could run hundreds of times faster than complicated models. All the above facts reveal a reasonable research process: researchers should first seek simple rather than intricate and complex solutions when faced with a software engineering problem. This is also consistent with the idea of âĂIJless, but betterâĂİ advocated by Menzies,[14] which aims at using simple approaches to solve complex problems. As pointed out by Majumder et al., [43], comparing new sophisticated approaches against the simpler alternatives before researchers release research results will assist avoid wasting research effort and find more useful and practical solutions.

8.2 For Practitioners

In practice, GLANCE can be easily leveraged by practitioners for predicting potential buggy code lines without huge application costs. The detailed implications are as follows:

(1) **This work reminds practitioners to recognize the value of a simple baseline approach in the field of CLBI again.** In the literature, several sets of automatic approaches (NLPs and MITs) have been proposed to identify the defective code lines. However, they are often supervised, and they have higher application costs so that it is not very convenient for practitioners to use them in practice. Unlike them, the proposed GLANCE has great advantages in both prediction performance and actual application cost. Therefore, paying attention to GLANCE will benefit the prediction of code-line-level defects in practice. Note that, two implementations of GLANCE (GLANCE-MD and GLANCE-EA) are training-free approaches, and practitioners can easily use them without collecting a lot of training data. This will save practitioners' time and effort greatly. Additionally, the rationales of GLANCE provide a more intuitive explanation for prediction results. For practitioners, therefore, it is easy to understand why some code lines are predicted to be defective. Furthermore, with the help of GLANCE, practitioners will gradually realize which kind of code lines should be checked first in their daily routine.

(2) **The more meaningful enlightenment to practitioners is that good coding habits will reduce the risk of code lines containing defects.** According to the results of this empirical study, it can be seen that there are some correlations between software defects and coding habits. That is, the code lines contain complex statements, and control statements are more likely to contain defects. For this reason, it is strongly recommended that practitioners should keep good programming habits and avoid writing buggy prone codes. In particular, it is an effective way to simplify the function of a single statement instead of gathering multiple functions in one statement. Although the latter is more compact in the code structure, it is very easy to introduce defects due to the complex logic structure. Therefore, splitting a complex statement into several simple statements is more helpful for developers to find the potential problems and hence repair them quickly. In addition, the code logic consists of a sequence of simple statements that is more conducive than

[14] http://menzies.us/.

an extremely complex statement to the communication between different participants. If possible, making standard specifications for programming in the same team can not only improve the robustness of software artifacts but also facilitate the process of CLBI in later.

8.3 For Tool Providers

This work provides a simple baseline solution GLANCE for identifying buggy code lines, which can be easily integrated in popular tools (SATs and IDEs) by tool providers to contribute more useful code optimization suggestions for users. The detailed implications are as follows:

(1) **GLANCE can be integrated into automatic static analysis tools (SATs) to prioritize static warnings to reduce the high false positive rates.** Although existing SATs (e.g., PMD) can analyze and report the potential defects (e.g., warnings) in the target software artifacts quickly without any dynamic execution process, it has been pointed out that most warnings are actually false positives [29, 34]. In this sense, developers always need to waste a lot of effort to eliminate these trivial warnings. Therefore, researchers [16, 26, 65] have been working on mitigating the false positive rates of SATs. As shown in Section 4, GLANCE can quickly assess the possibility that each line of code contains defects, which can be considered as a priority score if the code line is reported as a warning by SATs. Then, all warnings are ranked from the most to the least priority scores. According to the effectiveness of GLANCE, true warnings will be ranked at first so that they can be reviewed as soon as possible, while false positives will be ranked at last.

(2) **GLANCE can be used in the integrated development environments (IDEs) to motivate tool providers to enhance the code checker component.** As is known, IDEs (e.g., Eclipse[15] for Java, Visual Studio[16] for C++/C#, and PyCharm[17] for Python) are popular among programmers since these tools provide a great convenience for developing programs. Notably, like the SATs, the code checker component in these IDEs plays an important role in checking the quality of source codes before committing them to the code repository. For example, the function of âĂIJInspect CodeâĂİ in PyCharm can scan the source codes in the specified inspection scope and report all possible problems (e.g., incorrect call arguments and unused local). Removing these problems can greatly improve the robustness of the code. In this sense, the usefulness the of code checker component can be enhanced by GLANCE by reporting more potential risk of code. For example, the line of code with more than 20 tokens, or the line of code with more than 5 function calls, need to be highlighted by the code checker component, requiring a careful review by developers. These highlights will be removed if the developer confirms that there are no errors.

9 THREATS TO VALIDITY

The most important threats of this study are reflected in construct, internal, and external validity. Construct validity is the degree to which the dependent and independent variables accurately measure the concept they purport to measure. Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. External validity is the degree to which the results of the research can be generalized to the population under study and other research settings.

9.1 Construct Validity

In this study, the used dependent variable is a binary variable that indicates whether a file or a code line is defective. These labels were collected following the existing heuristic defect dataset

[15] https://www.eclipse.org/.
[16] https://visualstudio.microsoft.com/.
[17] https://www.jetbrains.com/pycharm/.

collection methods [60, 64, 73]. It is possible that there are some wrongly labeled code lines, which has been admitted in previous works. However, the reality is that false positives and false negatives can be avoided only by manually checking all data instances, which is usually unrealistic due to huge effort and unfamiliar context for specific software project. Therefore, current researchers [59, 60, 73] can only adopt a compromise scheme to collect defect data sets. In our study, we make some effort to alleviate the two types of noises. Specifically, for false positives, we have filtered out the comment code lines and blank lines that are unrelated to defects from the set of buggy lines. In this way, only those changes that modified the real function related source code will be tracked as BIC. Meanwhile, for false negatives, we found that some defect labels might be missed in the existing collection process by observing the development characteristics of real projects. That is, the code repository of a real project usually contains more than one development branch (or flow). The fixing information of some buggy lines that exists in multiple branches will be missed by existing collection methods since only one single default (or main) branch is considered. Therefore, we switch different branches to collect the corresponding buggy lines and combine them to obtain a complete defect dataset as soon as possible. In this sense, the threat to the construct validity has also been reduced in this study in the aspect of completeness of defect label collection.

Another threat is the possible wrongly marked buggy line since some code changes in some BFCs may be code refactoring operations instead of real bug fixing actions. For the refactoring code lines in the refactoring BFCs with a commit log message of âĂIJRefactor XXXâĂİ, we eliminate them by checking the log messages. For the refactoring code lines in the non-refactoring BFCs, we examine the ratio of refactoring code lines among all BFC lines. The low degree (2% on average) indicates that our experimental results cannot be influenced significantly.

9.2 Internal Validity

There are two main internal threats in this article. The first internal threat is from the selection of existing CLBI approaches. Because this article is devoted to the current research progress of CLBI, it is important to select the most representative works. To this end, the approaches used in this article are all from the top international conferences and journals. To the best of our knowledge, most bugginess identification studies focus on coarse-granularity instead of fine-grained code-line-level. In this sense, these selected approaches can be regarded as representative in the field of CLBI. Therefore, in our opinion, this threat has been minimized.

The second internal threat is from the comparison between GLANCE and the SOTA approaches. Because the implementation of these SOTA approaches are not open source, all approaches investigated in this study have been re-implemented by ourselves. To alleviate the deviation of the replicated approaches as much as possible, we carefully read the original articles and implement these approaches step by step. The implementation details have been set to public so that other researchers can acquire them easily [11]. Due to the subjectivity of any implementation code, it still needs to be validated in future work.

9.3 External Validity

The most important external threat is that our findings may not be generalized to other projects (e.g., non-Java projects or commercial projects). In our experiment, due to the lack of appropriate data sets, we only use a dataset consisting of 19 popular open-source ASF Java projects with 142 distinct releases, which are across a wide range of application domains and project scales. Based on this dataset, we obtain quite consistent results of performance comparison in most projects. In addition, the statistically meaningful conclusion can be drawn on large enough target releases. In this sense, we believe that this external threat is within the acceptable range. Nonetheless, we do not announce that our results can be generalized to all projects because the characteristics of

projects in reality are stochastic and changeable. To mitigate this threat further, there is a need to replicate our study across a wider variety of non-Java and commercial projects in the future.

10 RELATED WORKS

10.1 Identifying Buggy Lines under Just-in-time Scenario

In addition to the three types of solutions (i.e., NLPs, MITs, and SATs) studied in this article that are devoted to identifying buggy lines in a target software release, recently, several researchers [57, 78] aimed at conducting line level bugginess identification under **just-in-time** (**JIT**) scenario. Typically, this kind of approaches apply a two-stage process. Specifically, they first predict if a code change (i.e., commit) is defective at check-in time to help developers prioritize their limited SQA effort on the riskiest commits during the software development process. After that, the buggy lines in defect-introducing changes are furtherly identified so that much SQA effort can be reduced.

Yan et al., [78] first proposed a two-stage framework for just-in-time defect identification and localization. In the first stage, they trained a Logistic Regression model based on 14 popular change-level features (e.g., LT, Lines of code in a file before the current change). In the second stage, motivated by the software naturalness of historical source code, they built a source code language model based on n-gram modeling technique to locate the buggy lines. Later, Pornprasit et al., [57] also proposed a machine learning based approach JITLine to predict buggy lines in defect-introducing changes. First, JITLine leveraged Bag-of-Tokens features and commit-level features from McIntosh et al., [45] to build Random Forest commit-level classifier. Second, similarly to LineDP [73], JITLine also applied the LIME score [62] to calculate the suspicious of code lines in a defective change.

Obviously, the above approaches cannot be migrated to the context of GLANCE proposed by this article. Under the scenario of cross-release bugginess identification, due to the lack of the commit-based historical change information, practitioners or researchers cannot extract change-level features used by either [78] or [57] based on a given code release.

10.2 Static Software Metrics for Bugginess Identification

Over the years, many studies [4, 36, 46, 49, 50, 58] have mined various static code metrics as the defect features from different dimensions to facilitate the accuracy of their models. Static code metrics have attracted much attention due to the following two advantages. First, owing to their simple and clear definitions, static metrics are one of the few measures researchers and developers can collect in a consistent manner without ambiguity in the terminology of the metrics [47]. As a result, they are easy to collect and use, which reduces the cost of a prediction model [36]. Second, the models built with static code metrics perform well in terms of prediction performance and model generality in many studies [18, 49, 57].

Code complexity metrics are the most concerned defect indicators, which have been shown to correlate with defect density in several studies [4, 37, 49, 52, 88]. The intuition behind it is that, the more complex the software entity, the more likely it is to contain defects. The earliest code complexity metrics are McCabe metrics [44] that measure the complexity of pathways between module symbols, and Halstead metrics [17] that estimate the reading complexity. According to Ostrand et al.'s study [52], the size of a file is the most significant factor associated with the number of software faults. Zimmermann et al., [88] provided a popular benchmark with common complexity metrics. According to Nagappan et al.'s study [49], there was no single set of metrics that fits all projects. Therefore, they leveraged principal component analysis to determine the most suitable metrics combination and then used them to build accurate defect predictors.

Process-related metrics are another kind of defect indicator that researchers follow with interest [18, 36, 40, 50]. The process-related metrics mine the historical data (e.g., code churn [40])

from the process of software development to build defect predictors. Nagappan et al.'s empirical study [50] showed that organizational complexity metrics were effective predictors of defect-prone binaries in Windows Vista. Hassan et al., [18] proposed extended change (i.e., entropy) complexity metrics and showed that the models based on change complexity models were better predictors of future faults. Kondo et al., [36] proposed context metrics that measured the number of words/keywords around the changed lines of a commit as defect indicators for just-in-time defect prediction. The intuition behind context metrics is that, the more words the context contains, the more complex the context is, and the more likely it is to contain defects.

As can be seen, the above static metrics have been widely explored and applied for coarse-granularity bugginess identification, which show their usefulness. Furtherly, compared with prior works, our work investigates and leverages two types (complexity and control) of static metrics properly in GLANCE to predict code-line-level software defects. The study further confirms the value of static code metrics to a certain extent.

10.3 Simple Baseline Models in Software Engineering

Like the idea in our study, there have been many studies on simple baseline models in different fields (e.g., defect prediction [40, 81, 87], self-admitted technical debt identification [15, 83], search-based models [10]) of **software engineering** (**SE**) [13, 15]. The most important advantage is that the computational cost of these simple baseline models usually is low compared with sophisticated models, which motivates researchers to study simple baseline models in the field of SE.

In the field of defect prediction, Yang et al., [81] investigated the predictive power of simple unsupervised models in effort-aware just-in-time defect prediction and compared them with the SOTA supervised models. Their results showed that the unsupervised model (i.e., LT) performed better than supervised models. Later, Liu et al., [40] built a **code churn based unsupervised model** (**CCUM**), which exhibited a more superior performance. Zhou et al., [87] conducted a comprehensive study to investigate how far researchers had really progressed in **cross-project defect prediction** (**CPDP**). They found that the performance of simple unsupervised module size models (i.e., ManualDown) had a comparable or even superior prediction performance compared with the existing SOTA CPDP models.

In other fields of SE, Chen et al., [10] designed âĂIJSamplingâĂİ as a baseline optimizer for search-based SE models, which is competitive with corresponding SOTA evolutionary algorithms while requiring orders of magnitude fewer evaluations. Fu et al., [13] conducted a case study on deep learning and proposed to use a simple unsupervised model to solve the problem of which questions in the Stack Overflow programmer discussion forum can be linked together. According to their results, their model can achieve similar (or even better) results compared with the investigated supervised deep learning model. Meanwhile, their model can reduce the cost of the total time of the investigated deep learning model by 84 times. Guo et al., [15] found that there is a high correlation between task tags (e.g., TODO) and self-admitted technical debt, and hence proposed a baseline model MAT based on task tags. Their experiments in two large scale datasets indicate that MAT is competitive to the SOTA deep learning-based models in identifying self-admitted technical debt.

Compared with existing simple baseline models, the proposed GLANCE furtherly confirmed the usability of the idea of âĂIJeasy over hardâĂİ [13] in the field of software engineering. In particular, GLANCE can provide a more convenient and efficient solution when one wants to identify buggy code lines.

11 CONCLUSION AND FUTURE WORK

In this article, we conduct a comprehensive empirical study to investigate the status quo in the field of CLBI. We compare three categories of SOTA CLBI approaches with the simple baseline

solution GLANCE in terms of classification and ranking performance. GLANCE emphatically considers both control statements and complex statements in a file to identify buggy lines. However, the existing CLBI approaches cannot capture this relationship with buggy lines from the labeled training data properly. We use 19 different open-source ASF Java projects to conduct the comparison experiment. Our experimental results surprisingly show that, GLANCE is very competitive or even superior to all the studied SOTA CLBI approaches and tools, regardless of whether classification and ranking performance indicators are considered. This result indicates that, if the identification performance is the goal, the real progress in CLBI is not being achieved as it might have been envisaged in the literature and there is still a long way to go to really promote the effectiveness of static analysis tools in industry. Note that, since GLANCE can still report high false alarms, the limitations of SOTA CLBI approaches have not been completely resolved. Meanwhile, due to a low computation cost and a low memory requirement, GLANCE can be efficiently applied in practice. Therefore, we strongly recommend that future CLBI studies consider GLANCE as an easy-to-implement baseline. In practice, using GLANCE as a new baseline will enable us to determine whether a new identification approach is practically useful.

In the future, we plan to investigate the more characteristics of buggy code lines for an accurate CLBI. Specifically, we will investigate developers' habit of writing programs to mine more information about defect-prone codes and provide a more effective CLBI approach.

ACKNOWLEDGMENTS

We are very grateful to S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto for sharing their file-level and line-level data sets online. This motivates us to find the characteristics of buggy code lines and hence to propose the simple baseline CLBI solution (i.e., GLANCE).

REFERENCES

[1] Amritanshu Agrawal, Wei Fu, Di Chen, Xipeng Shen, and Tim Menzies. 2021. How to "DODGE" complex software analytics. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2182–2194. DOI: https://doi.org/10.1109/TSE.2019.2945020

[2] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2019. A survival analysis-based prioritization of code checker warning: A case study using PMD. In *Proceedings of the Big Data, Cloud Computing, and Data Science Engineering, Selected Papers from The 4th IEEE/ACIS International Conference on Big Data, Cloud Computing, Data Science and Engineering.* Roger Y. Lee (Ed.), Vol. 844, Springer, 69–83. DOI: https://doi.org/10.1007/978-3-030-24405-7_5

[3] Ömer Faruk Arar and Kürsat Ayan. 2015. Software defect prediction using cost-sensitive neural network. *Applied Soft Computing* 33 (2015), 263–277. DOI: https://doi.org/10.1016/j.asoc.2015.04.045

[4] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* 22, 10 (1996), 751–761. DOI: https://doi.org/10.1109/32.544352

[5] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE Computer Society, 470–481. DOI: https://doi.org/10.1109/SANER.2016.105

[6] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society Series b-methodological* 57, 1 (1995), 289–300.

[7] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: Discovering negative association rules from code for bug detection. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.), ACM, 411–422. DOI: https://doi.org/10.1145/3236024.3236032

[8] BugDet 2022. Dataset and scripts. (2022). Retrieved from https://github.com/Naplues/BugDet. Accessed November 8, 2022.

[9] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories.* Premkumar T. Devanbu, Sung Kim, and Martin Pinzger (Eds.), ACM, 252–261. DOI: https://doi.org/10.1145/2597073.2597102

[10] Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. 2019. "Sampling" as a baseline optimizer for search-based software engineering. *IEEE Trans. Software Eng.* 45, 6 (2019), 597–614. https://doi.org/10.1109/TSE.2018.2790925

[11] CLBI 2022. Replication kit. (2022). Retrieved from https://github.com/Naplues/CLBI. Accessed November 8, 2022.

[12] Norman E. Fenton and Niclas Ohlsson. 2000. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering* 26, 8 (2000), 797–814. DOI: https://doi.org/10.1109/32.879815

[13] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.), ACM, 49–60. DOI: https://doi.org/10.1145/3106237.3106256

[14] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. 2012. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. Per Runeson, Martin Höst, Emilia Mendes, Anneliese Amschler Andrews, and Rachel Harrison (Eds.), ACM, 171–180. DOI: https://doi.org/10.1145/2372251.2372285

[15] Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. 2021. How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study. *ACM Transactions on Software Engineering and Methodology* 30, 4 (2021), 45:1–45:56. DOI: https://doi.org/10.1145/3447247

[16] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? A study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.), ACM, 317–328. DOI: https://doi.org/10.1145/3238147.3238213

[17] Maurice H. Halstead. 1977. Elements of software science (Operating and Programming Systems Series). Elsevier Science Inc. DOI: https://dl.acm.org/doi/10.5555/540137

[18] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 78–88. DOI: https://doi.org/10.1109/ICSE.2009.5070510

[19] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. 2015. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology* 59 (2015), 170–190. DOI: https://doi.org/10.1016/j.infsof.2014.11.006

[20] Vincent J. Hellendoorn and Premkumar T. Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.), ACM, 763–773. DOI: https://doi.org/10.1145/3106237.3106290

[21] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristof Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matús Sulír, Fatemeh H. Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. 2022. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering* 27, 6 (2022), 125. DOI: https://doi.org/10.1007/s10664-021-10083-5

[22] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*. Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.), IEEE Computer Society, 837–847. DOI: https://doi.org/10.1109/ICSE.2012.6227135

[23] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories*. Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.), IEEE / ACM, 34–45. DOI: https://doi.org/10.1109/MSR.2019.00016

[24] David Hovemeyer and William W. Pugh. 2007. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Manuvir Das and Dan Grossman (Eds.), ACM, 9–14. DOI: https://doi.org/10.1145/1251535.1251537

[25] Nasif Imtiaz, Brendan Murphy, and Laurie A. Williams. 2019. How do developers act on static analysis alerts? An empirical study of coverity usage. In *Proceedings of the 30th IEEE International Symposium on Software Reliability Engineering*. Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro (Eds.), IEEE, 323–333. DOI: https://doi.org/10.1109/ISSRE.2019.00040

[26] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie A. Williams. 2019. Challenges with responding to static analysis tool alerts. In *Proceedings of the 16th International Conference on Mining Software Repositories*. Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.), IEEE / ACM, 245–249. DOI: https://doi.org/10.1109/MSR.2019.00049

[27] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. 2020. The impact of automated feature selection techniques on the interpretation of defect models. *Empirical Software Engineering* 25, 5 (2020), 3590–3638. DOI: https://doi.org/10.1007/s10664-020-09848-1

[28] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, Hoa Khanh Dam, and John C. Grundy. 2022. An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering* 48, 2 (2022), 166–185. DOI: https://doi.org/10.1109/TSE.2020.2982385

[29] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 35th International Conference on Software Engineering*. David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.), IEEE Computer Society, 672–681. DOI: https://doi.org/10.1109/ICSE.2013.6606613

[30] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106. DOI: https://doi.org/10.1007/s10664-015-9400-x

[31] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. 2010. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*. Radu Marinescu, Michele Lanza, and Andrian Marcus (Eds.), IEEE Computer Society, 1–10. DOI: https://doi.org/10.1109/ICSM.2010.5609530

[32] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773. DOI: https://doi.org/10.1109/TSE.2012.70

[33] Ritu Kapur and Balwinder Sodhi. 2020. A defect estimator for source code: Linking defect reports with programming constructs usage metrics. *ACM Transactions on Software Engineering and Methodology* 29, 2 (2020), 12:1–12:35. DOI: https://doi.org/10.1145/3384517

[34] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should I fix first?. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Ivica Crnkovic and Antonia Bertolino (Eds.), ACM, 45–54. DOI: https://doi.org/10.1145/1287624.1287633

[35] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 489–498. DOI: https://doi.org/10.1109/ICSE.2007.66

[36] Masanari Kondo, Daniel M. Germán, Osamu Mizuno, and Eun-Hye Choi. 2020. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering* 25, 1 (2020), 890–939. DOI: https://doi.org/10.1007/s10664-019-09736-3

[37] Akif Günes Koru and Hongfang Liu. 2005. An investigation of the effect of module size on defect prediction using static measures. *Proceedings of the 2005 Workshop on Predictor Models in Software Engineering* 30, 4 (2005), 1–5. DOI: https://doi.org/10.1145/1082983.1083172

[38] Lov Kumar, Sanjay Misra, and Santanu Ku. Rath. 2017. An empirical analysis of the effectiveness of software metrics and fault prediction model for identifying faulty classes. *Computer Standards and Interfaces* 53 (2017), 1–32. DOI: https://doi.org/10.1016/j.csi.2017.02.003

[39] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4BL: Reproducibility study on the performance of IR-based bug localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Frank Tip and Eric Bodden (Eds.), ACM, 61–72. DOI: https://doi.org/10.1145/3213846.3213856

[40] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Ayse Bener, Burak Turhan, and Stefan Biffl (Eds.), IEEE Computer Society, 11–19. DOI: https://doi.org/10.1109/ESEM.2017.8

[41] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: How far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.), ACM, 373–384. DOI: https://doi.org/10.1145/3238147.3238190

[42] Amirabbas Majd, Mojtaba Vahidi-Asl, Alireza Khalilian, Pooria Poorsarvi-Tehrani, and Hassan Haghighi. 2020. SLDeep: Statement-level software defect prediction using deep-learning model on static code features. *Expert Systems with Applications* 147 (2020), 113156. DOI: https://doi.org/10.1016/j.eswa.2019.113156

[43] Suvodeep Majumder, Nikhila Balaji, Katie Brey, Wei Fu, and Tim Menzies. 2018. 500+ times faster than deep learning: A case study exploring faster methods for text mining stackoverflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*. Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.), ACM, 554–563. DOI: https://doi.org/10.1145/3196398.3196424

[44] Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering* 2, 4 (1976), 308–320. DOI: https://doi.org/10.1109/TSE.1976.233837

[45] Shane McIntosh and Yasutaka Kamei. 2018. Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* 44, 5 (2018), 412–428. DOI: https://doi.org/10.1109/TSE.2017.2693980

[46] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering* 33, 1 (2007), 2–13. DOI: https://doi.org/10.1109/TSE.2007.256941

[47] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayse Basar Bener. 2010. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (2010), 375–407. DOI: https://doi.org/10.1007/s10515-010-0069-5

[48] Akito Monden, Takuma Hayashi, Shoji Shinoda, Kumiko Shirai, Junichi Yoshida, Mike Barker, and Ken-ichi Matsumoto. 2013. Assessing the cost effectiveness of fault prediction in acceptance testing. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1345–1357. DOI: https://doi.org/10.1109/TSE.2013.21

[49] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*. Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.), ACM, 452–461. DOI: https://doi.org/10.1145/1134285.1134349

[50] Nachiappan Nagappan, Brendan Murphy, and Victor R. Basili. 2008. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*. Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.), ACM, 521–530. DOI: https://doi.org/10.1145/1368088.1368160

[51] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: Integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.), ACM, 672–683. DOI: https://doi.org/10.1145/3540250.3549165

[52] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. 2004. Where the bugs are. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*. George S. Avrunin and Gregg Rothermel (Eds.), ACM, 86–96. DOI: https://doi.org/10.1145/1007512.1007524

[53] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering* 31, 4 (2005), 340–355. DOI: https://doi.org/10.1109/TSE.2005.49

[54] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 20th International Symposium on Software Testing and Analysis*. Matthew B. Dwyer and Frank Tip (Eds.), ACM, 199–209. DOI: https://doi.org/10.1145/2001420.2001445

[55] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2018. Re-evaluating method-level bug prediction. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.), IEEE Computer Society, 592–601. DOI: https://doi.org/10.1109/SANER.2018.8330264

[56] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2020. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software* 161 (2020), 1–15. DOI: https://doi.org/10.1016/j.jss.2019.110493

[57] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction. In *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 369–379. DOI: https://doi.org/10.1109/MSR52588.2021.00049

[58] Danijel Radjenovic, Marjan Hericko, Richard Torkar, and Ales Zivkovic. 2013. Software fault prediction metrics: A systematic literature review. *Information and Software Technology* 55, 8 (2013), 1397–1418. DOI: https://doi.org/10.1016/j.infsof.2013.02.009

[59] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar T. Devanbu. 2014. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*. Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.), ACM, 424–434. DOI: https://doi.org/10.1145/2568225.2568269

[60] Baishakhi Ray, Vincent J. Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar T. Devanbu. 2016. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*. Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.), ACM, 428–439. DOI: https://doi.org/10.1145/2884781.2884848

[61] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Michael F. P. O'Boyle and Keshav Pingali (Eds.), ACM, 419–428. DOI: https://doi.org/10.1145/2594291.2594321

[62] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data*

Mining, San Francisco. Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.), ACM, 1135–1144. DOI: https://doi.org/10.1145/2939672.2939778

[63] Danilo Silva, João Paulo da Silva, Gustavo Jansen de Souza Santos, Ricardo Terra, and Marco Túlio Valente. 2021. RefDiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2786–2802. DOI: https://doi.org/10.1109/TSE.2020.2968072

[64] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*. ACM. DOI: https://doi.org/10.1145/1083142.1083147

[65] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 283–294. DOI: https://doi.org/10.1109/ASE.2009.72

[66] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects. *Empirical Software Engineering* 25, 6 (2020), 5137–5192. DOI: https://doi.org/10.1007/s10664-020-09880-1

[67] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. IEEE, 127–138. DOI: https://doi.org/10.1109/ICSME46990.2020.00022

[68] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.), ACM, 269–280. DOI: https://doi.org/10.1145/2635868.2635875

[69] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25, 2 (2020), 1419–1457. DOI: https://doi.org/10.1007/s10664-019-09750-5

[70] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. David Lo, Sven Apel, and Sarfraz Khurshid (Eds.), ACM, 708–719. DOI: https://doi.org/10.1145/2970276.2970341

[71] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Ivica Crnkovic and Antonia Bertolino (Eds.), ACM, 35–44. DOI: https://doi.org/10.1145/1287624.1287632

[72] Supatsara Wattanakriengkrai, Napat Srisermphoak, Sahawat Sintoplertchaikul, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Hideaki Hata, and Kenichi Matsumoto. 2019. Automatic classifying self-admitted technical debt using N-Gram IDF. In *Proceedings of the 26th Asia-Pacific Software Engineering Conference*. IEEE, 316–322. DOI: https://doi.org/10.1109/APSEC48747.2019.00050

[73] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2022. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering* 48, 5 (2022), 1480–1496. DOI: https://doi.org/10.1109/TSE.2020.3023177

[74] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 181–190. DOI: https://doi.org/10.1109/ICSME.2014.40

[75] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: Locate crash-inducing changes based on crash reports. *Empirical Software Engineering* 23, 5 (2018), 2866–2900. DOI: https://doi.org/10.1007/s10664-017-9567-4

[76] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: Locating crashing faults based on crash stacks. In *Proceedings of the International Symposium on Software Testing and Analysis*. Corina S. Pasareanu and Darko Marinov (Eds.), ACM, 204–214. DOI: https://doi.org/10.1145/2610384.2610386

[77] Meng Yan, Yicheng Fang, David Lo, Xin Xia, and Xiaohong Zhang. 2017. File-level defect prediction: Unsupervised vs. supervised models. In *Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Ayse Bener, Burak Turhan, and Stefan Biffl (Eds.), IEEE Computer Society, 344–353. DOI: https://doi.org/10.1109/ESEM.2017.48

[78] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E. Hassan, David Lo, and Shanping Li. 2022. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering* 48, 2 (2022), 82–101. DOI: https://doi.org/10.1109/TSE.2020.2978819

[79] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26. DOI: https://doi.org/10.1109/QRS.2015.14

[80] Yibiao Yang, Mark Harman, Jens Krinke, Syed S. Islam, David W. Binkley, Yuming Zhou, and Baowen Xu. 2016. An empirical study on dependence clusters for effort-aware fault-proneness prediction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. David Lo, Sven Apel, and Sarfraz Khurshid (Eds.), ACM, 296–307. DOI: https://doi.org/10.1145/2970276.2970353

[81] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.), ACM, 157–168. DOI: https://doi.org/10.1145/2950290.2950353

[82] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining software defects: Should we consider affected releases?. In *Proceedings of the 41st International Conference on Software Engineering*. Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.), IEEE / ACM, 654–665. DOI: https://doi.org/10.1109/ICSE.2019.00075

[83] Zhe Yu, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies. 2022. Identifying self-admitted technical debts with jitterbug: A two-step approach. *IEEE Trans. Software Eng.* 48, 5 (2022), 1676–1691. https://doi.org/10.1109/TSE.2020.3031401

[84] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*. Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.), IEEE Computer Society, 334–344. DOI: https://doi.org/10.1109/MSR.2017.2

[85] Hongyu Zhang and S. C. Cheung. 2013. A cost-effectiveness criterion for applying software defect prediction models. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*., Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.), ACM, 643–646. DOI: https://doi.org/10.1145/2491411.2494581

[86] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *3Proceedings of the 4th International Conference on Software Engineering*. Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.), IEEE Computer Society, 14–24. DOI: https://doi.org/10.1109/ICSE.2012.6227210

[87] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. 2018. How far we have progressed in the journey? An examination of cross-project defect prediction. *ACM Transactions on Software Engineering and Methodology* 27, 1 (2018), 1:1–1:51. DOI: https://doi.org/10.1145/3183339

[88] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*. IEEE Computer Society, 9. DOI: https://doi.org/10.1109/PROMISE.2007.10

Received 6 January 2022; revised 8 November 2022; accepted 14 December 2022