Method-Level Bug Prediction: Problems and Promises

SHAIFUL CHOWDHURY*, University of Manitoba, Canada
GIAS UDDIN†, York University, Canada
HADI HEMMATI‡, York University, Canada
REID HOLMES, University of British Columbia, Canada

Fixing software bugs can be colossally expensive, especially if they are discovered in the later phases of the software development life cycle. As such, bug prediction has been a classic problem for the research community. As of now, the Google Scholar site generates ~113,000 hits if searched with the "bug prediction" phrase. Despite this staggering effort by the research community, bug prediction research is criticized for not being decisively adopted in practice. A significant problem of the existing research is the granularity level (i.e., class/file level) at which bug prediction is historically studied. Practitioners find it difficult and time-consuming to locate bugs at the class/file level granularity. Consequently, method-level bug prediction has become popular in the last decade. We ask, *are these method-level bug prediction models ready for industry use?* Unfortunately, the answer is *no*. The reported high accuracies of these models dwindle significantly if we evaluate them in different realistic time-sensitive contexts. It may seem hopeless at first, but encouragingly, we show that future method-level bug prediction can be improved significantly. In general, we show how to reliably evaluate future method-level bug prediction models, and how to improve them by focusing on four different improvement avenues: building noise-free bug data, addressing concept drift, selecting similar training projects, and developing a mixture of models. Our findings are based on three publicly available method-level bug datasets, and a newly built bug dataset of 774,051 Java methods originating from 49 open-source software projects.

CCS Concepts: • **Software and its engineering** → **Empirical software validation**.

Additional Key Words and Phrases: method-level bug prediction, code metrics, maintenance, McCabe, code complexity

ACM Reference Format:
Shaiful Chowdhury, Gias Uddin, Hadi Hemmati, and Reid Holmes. 2023. Method-Level Bug Prediction: Problems and Promises. 1, 1 (January 2023), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Modern software has become more complex and, thus, more bug-prone than ever. While billions of dollars are lost due to software bugs [96], lives are lost too, including the recent incidents of the Boeing 737 Max aircrafts [34]. As such, developers and testers spend significant time finding and fixing software bugs, which may account for 50 to 70% of the whole development cost [115]. The cost of bug fixing, however, is much cheaper if accomplished in the early

*Also with University of Calgary, Alberta, Canada.
†Also with University of Calgary, Alberta, Canada.
‡Also with University of Calgary, Alberta, Canada.

Authors' addresses: Shaiful Chowdhury, shaiful.chowdhury@umanitoba.ca, University of Manitoba, Winnipeg, MB, Canada; Gias Uddin, guddin@yorku.ca, York University, Toronto, ON, Canada; Hadi Hemmati, hemmati@yorku.ca, York University, Toronto, ON, Canada; Reid Holmes, rtholmes@cs.ubc.ca, University of British Columbia, Vancouver, BC, Canada.

Manuscript submitted to ACM

phase of the software development life cycle [15]. Accordingly, bug prediction has been a classic research problem to the community [25, 35, 83, 87, 102, 115, 116]—so much so that it was even labelled as the "prince of empirical software engineering research" [62]. The tragedy is, despite the hype and the effort by the SE research community, bug prediction research lacks impact in industrial practice [62, 109]. This paper aims to reveal some of the root causes of this tragedy, and proposes a set of guidelines to improve future bug prediction research.

A bug prediction model is supposed to detect the code fragments more likely to contain bugs in the future, which should reduce the time and cost of the bug-finding process [54, 112]. Unfortunately, the majority of the bug research (e.g., [9, 37, 122]) focused on predicting bugs at the class/file level source code granularity. Developers find class/file level bug prediction too coarse to be practically useful [75, 83, 99], due to the infeasible time requirement for finding bugs in an arbitrarily large file. Accordingly, *method-level bug prediction* (MLBP) has become one of the holy grails in SE research, leading to significant research in recent years [33, 35, 44, 72, 74, 101]. Given their high prediction accuracy[1] at the practically useful method-level granularity, are not these models ready for industry practice? We do not know until these models are evaluated in practically meaningful scenarios (described later). Unfortunately, all of these MLBP models were evaluated using the time-insensitive k-fold cross validation approach, which is unrealistic in a time-sensitive bug prediction problem [8, 83]. The time-sensitive accuracies of the MLBP models applicable to industry practices are, therefore, unknown.

In this paper, we study the true effectiveness of the existing MLBP models, using scenarios that would be desired by the practitioners. We show that all the existing models perform poorly when evaluated in such scenarios. MLBP thus remains an open research problem, as was also claimed by Pascarella *et al.* [83]. With empirical evidence, we then discuss a quartet of potential avenues that can significantly help future MLBP research. The contribution of this paper is founded on the following five research questions.

To understand the effectiveness of the existing bug prediction models, we answer the following research question.

RQ1: Do the existing method-level bug prediction models perform well when evaluated in realistic scenarios?

Contribution 1: The k-fold cross validation approach forms training data by mixing both past and future data [8]. But in a real-world scenario, future data is unavailable, and that is the whole point for building a bug prediction model. We, therefore, evaluate the prediction accuracy of bug prediction models in three realistic ways (described in Section 4). The common approach in all three scenarios is not to use any future data during training. In all cases, the accuracy (e.g., precision) was significantly worse compared to the cross validation evaluation (e.g., 60% drop in precision).

Leveraging and exploiting the previous bug prediction studies (Section 2), we investigate how MLBP models can be improved in the future. In that vein, we answer the following four research questions.

RQ2: How important is accurate bug labelling for the success of future MLBP models?

Contribution 2: We first identify several drawbacks with the current bug labelling approaches that lead to noisy training data, and thus produce inaccurate prediction models [32]. For example, the keywords that are used to detect bug-fixing commits are often inaccurate: the existence of the word *issue* does not always mean it was a bug-fixing commit. Also, developers often commit unrelated changes together, known as tangled changes [49]. Tangled changes make it difficult to understand which methods in a particular commit are related to bug-fix, and which are not. We show that, due to the noise, the popular bug predictors (e.g., McCabe complexity) perform poorly in distinguishing between buggy and not buggy methods. We then propose an accurate labelling approach to reduce the noise in training data. Results suggest that our more precise labelling approach can help improve future MLBP models significantly.

[1]Unless otherwise stated, accuracy in this paper means precision, recall, and F-score.

RQ3: Can a method's age, as an explanatory variable, be a potential replacement for the expensive model retraining approach?

Contribution 3: A common hypothesis adopted in all the previous MLBP models is that a method with high code complexity and previous code churn history is more likely to have bugs in the future. This hypothesis, unfortunately, does not hold in the long run, due to concept drift in software data [55]. We found that the bug-proneness of a method decays consistently as it ages, regardless of its complexity and change history. This makes a method's age a strong candidate to model its time-varying bug-proneness. Therefore, future MLBP models should consider method ages for accommodating concept drift without the expensive repeated retraining of the models.

RQ4: Should future MLBP research focus on an optimal project set selection?

Contribution 4: A common belief in training machine learning models is, *more data is usually better*. Refuting this presumption, we show that less number of selected projects can often produce more accurate models. This suggests the importance of future research on an optimal project set selection for a given test project, instead of building models with an arbitrarily large number of projects.

RQ5: Should future MLBP research focus on a mixture of models?

Contribution 5: We have found that the accuracy of a bug prediction model highly depends on the method's size in which it is applied. The distribution of bug-prone methods is much more skewed in small methods than in large methods. Consequently, the accuracy is always significantly lower in small methods than in large methods. These two observations suggest that future research should build different models for different method sizes instead of a generic model applied to all.

Replication: To enable replication and extension, we share all the datasets publicly.[2]

1.1 Paper organization

In Section 2, we discuss how the existing research on bug prediction has motivated this study. Section 3 discusses the methodologies that are commonly followed across different research questions. Section 4 presents the accuracy of the existing MLBP approaches in realistic scenarios (RQ1). The other four research questions, RQ2 to RQ5, are answered in Section 5. In Section 6, we summarize our findings and discuss the potential future works along with the threats to validity. Section 8 concludes this paper.

2 RELATED WORK & MOTIVATION

In this section, we discuss the related studies on bug prediction, and how those studies motivated this paper.

2.1 Granularity

Historically, most of the bug prediction models were built for class/file level granularities (e.g., [3, 9, 11, 37, 77, 122, 124]). Unfortunately, practitioners often find it difficult to locate bugs at this coarse level granularity [83, 99]. Finding bugs at the class/file level granularity is inefficient because only around 17% of the methods in a bug-prone file are bug related [74]. Studies have also found that larger files are generally more bug-prone [30, 37, 80], making it even more difficult to find bugs. A reasonable solution is to develop bug prediction models at lower level granularity [57]. The most efficient would be to build line-level bug prediction models. Such models, however, are inaccurate (e.g., [116]), because tracking source code history is often required for building a bug prediction model. This is very difficult at

[2]https://github.com/shaifulcse/dataset-MLBP-2022

the line level because many lines can be similar just by chance [40, 97, 106]. As such, *method-level bug prediction* has become a hot research topic in the community [33, 35, 44, 72, 74, 83, 101]. A method, generally, is much smaller in size than a class, which narrows down the search space. Also, if we can identify the group of more bug-prone methods, our testing process can focus on those methods only (e.g., by adding and improving the unit tests for those methods). This may dramatically optimize the allocation of limited resources during the testing and maintenance phases [66, 104].

2.2 Predictors

Models for bug prediction were built by using source code metrics [9, 33, 37, 78, 124], historical change metrics [38, 76, 102], and developer-related metrics [27]. Zimmerman *et al.* [124] studied the correlation between common complexity metrics and bugs, and found that complex code leads to more bugs. Multiple studies (e.g., [9, 42]) have claimed that the popular C&K metrics [19] are good bug predictors. Unfortunately, the true effectiveness of code metrics has been debated. Research has shown that all the famous code metrics were found to be only as effective as their correlation strength with size [30, 37]. Therefore, these code metrics provide no useful information if their correlation with the size is neutralized. This criticism, however, is valid only at coarse-level granularities, such as class and module, but does not hold for method-level granularity [22, 61].

Historical change metrics include the number of revisions, modification size, and modification type (#added lines, #deleted lines) of a code component to understand its future change- or bug-proneness [22, 74]. Research has claimed that change history can be a better bug predictor than source code metrics [38, 74]. Collecting change history at the method level, however, has always been more challenging than file/class level granularity. Fortunately, the recent state-of-the-art tool, CodeShovel [40], has solved this problem. It can return the complete change history of a given method within a few seconds with high accuracy [39, 40].

2.3 Cross validation

Despite the potential of code metrics and change metrics as good predictors, MLBP is still considered an open research problem [83]. All the MLBP models [33, 35, 44, 72, 74, 101] were evaluated with the time-insensitive k-fold cross-validation approach [92], thus, incorrectly resulting in high accuracy. In a k-fold cross validation, the whole dataset is divided into k different folds. There are k iterations, and in each iteration, a different fold is considered as the test data. Training data is formed by using the remaining $k - 1$ folds. This is unrealistic in bug prediction because the training data would often contain information from the future, known as information leakage [8]. Consider a dataset with 10 consecutive releases of a software project. While the second fold (i.e., release 2) is the test data, the training data will be formed by using release 1 data, and everything from release 3 to release 10. This is impractical because, in a realistic scenario, only the release 1 data will be available for training while predicting bugs for release 2.

Motivation for RQ1. Pascarella *et al.* [83] reproduced the MLBP models of Giger *et al.* [35] with the unrealistic cross validation approach. Similar to the original study, the accuracies of all the models were good. They then evaluated the same models in a time-sensitive realistic scenario, where all the data from release 1 to release $x - 1$ was used for training while predicting bug-proneness at release x. The prediction accuracies of all the models became extremely poor and unacceptable for practice, which led them to call MLBP an open research problem. It is, however, rational to argue that the poor performance of Pascarella *et al.* can be due to the lack of enough training data. Pascarella *et al.* did not consider the potential benefit of enlarging the training dataset by adding external projects' data. Also, their conclusion was based on one single dataset that was built by themselves. Therefore, the findings of Pascarella *et al.*

might not be generalizable. In this research question, we reevaluate the findings of Pascarella *et al.*, but instead of using one dataset and one evaluation scenario, we use three different publicly available datasets and three realistic scenarios.

2.4 Bug labelling

Traditionally, a code component, such as a method, is considered bug-prone if it has been modified in a bug-fixing commit [74, 89, 105]. For identifying the bug-fixing commits, the SSZ approach has been the most adopted [103], which has been further improved in subsequent studies (e.g., [26, 58]). The core of these approaches is to use a set of keywords and bug IDs; if a commit message contains any of these keywords or a bug ID that is also found in a bug report, that commit message is considered to be bug-fixing. The SSZ-like approaches not only find a buggy code component, but also can locate the bug-inducing commit. The accuracy of these SSZ-like approaches, however, has been criticized by a recent study [94]. Considering all the SSZ variants, the highest precision was only 70% in identifying bug-inducing commits.

The keywords that are used to detect bug-fix commit messages differ across studies [37, 89, 105]. These keywords-based systems produce too many false positives and false negatives, and may lead to a noisy dataset [13]. This problem intensifies when unrelated code changes are committed in a single transaction, known as tangled code changes [28, 50, 59, 67]. Therefore, labelling all the modified methods in a bug-fixing commit as buggy methods is inaccurate, and harms bug prediction models significantly [49].

Motivation for RQ2. None of the existing MLBP models attempted to alleviate this bug labelling problem while building the training data. Therefore, the significance of the impact of noisy data on bug prediction accuracy is unclear. Consequently, we are unsure if it is worthwhile for future MLBP research to invest in building a more accurate bug labelling approach.

2.5 Concept drift

"Change is the only constant in software." With the evolution of software, the distribution of their characteristics, such as the distribution of buggy methods, will change. This phenomenon is known as concept drift [12, 29, 55, 113]. Research has shown that bug prediction models should be updated continuously—otherwise, their prediction performance degrades significantly due to the change in data [55, 113]. To our surprise, existing MLBP models have ignored the impact of concept drift. The approaches (e.g., [35, 74]) are to build the model once and to use it forever. Although these approaches can work at the beginning of the software development life cycle, they become less practical over time.

Updating a model with time, however, can be expensive, because it may involve repeatedly selecting similar training data, and then retraining the model [113]. Concept drift in bug prediction was also observed in a study by a group of Google practitioners [65]. Supporting the findings of Rahman *et al.* [86], they have initially observed that if a file was involved with a large number of bug-fixing commits, that file should be flagged as a bug-prone file, and should be inspected more. This theory, however, does not hold for a long time; a bug-prone file, with time and continuous fixing, may become bug-free. Google has later improved the model by considering the age of the bug-fixing commits so that their impact on bug-proneness decays over time. This can potentially capture concept drift, but without repeatedly retraining the model.

Motivation for RQ3. We are interested to know if we can accommodate concept drift in the MLBP models without the expensive repetitive model retraining approach. We can deduce from the literature that a method's bug-proneness should change as it ages, but does it follow a consistent pattern? If so, a method's age can potentially be used to capture its time-varying bug-proneness (i.e., concept drift).

2.6 Project selection

A common hypothesis in machine learning is, *the more the training data, the higher the prediction accuracy*, leading to significant research in artificial data augmentation [53, 118]. This hypothesis, however, does not necessarily hold in software defect prediction. Characteristics of different software can be significantly different from each other [22, 36]. This is problematic because, in most modeling approaches, training and test data must come from similar distributions [68]. Therefore, it is often more useful to select training projects that are similar to a test project, than to arbitrarily augment the training data by adding more and more dissimilar projects [6]. As such, recent bug prediction studies have focused on defining and selecting similar projects [6, 107, 113], and showed that appropriate source project selection can improve bug prediction accuracy very significantly.

Motivation for RQ4. Unfortunately, none of the existing MLBP research has considered similar source project selection before training a model. Therefore, we investigate if future studies should focus on systematic project selection algorithms to help MLBP models.

2.7 Multiple models

A single generic model does not perform well for a dataset that contains clusters of data with different distributions from each other. In such cases, multiple models are built so that each model is associated with one particular cluster of data [48, 108]. The diversity in software data [22] makes building multiple models a potential candidate to improve bug prediction accuracy. Consequently, an array of bug prediction research [48, 63, 108, 114, 120] has focused on this area, specifically in ensemble modeling. In ensemble modeling, a final outcome is generated from the outcomes of multiple base models [53]. For example, a code component would be predicted as buggy only if the majority of the base models predict it as buggy. Reportedly, bug prediction accuracy has improved significantly with different forms of ensemble modeling [3, 7, 114].

Motivation for RQ5. According to a study by Chowdhury *et al.* [22], the characteristics of Java methods vary based on their size. For example, the variability in code metrics distribution is very different between large and small methods. This encourages us to investigate if future MLBP research should focus on building a mixture of models. More specifically, should we build separate models for small and large methods?

3 METHODOLOGY

In this section, we describe the publicly available datasets, the need and the process to make a new dataset, and the analysis approach that we follow to answer our research questions.

3.1 Available dataset

The objective of RQ1 is to evaluate the true effectiveness of the existing MLBP approaches in different realistic scenarios. In particular, we want to know what is the accuracy of MLBP when evaluated in time-sensitive ways, in contrast to the cross validation approaches that were followed previously. Unfortunately, only three of the method-level bug studies have shared their dataset publicly: the dataset of Ferenc *et al.* [33], Shippey *et al.* [101], and Mo *et al.* [74]. The datasets of Ferenc *et al.*, and Shippey *et al.* contain only source code metrics, whereas the dataset of Mo *et al.* contains both the source code and change history metrics. The dataset of Mo *et al.* was incomplete when we first accessed it, but thankfully, the authors fixed it after we contacted them. Similarly, the link to download the dataset of Shippey *et al.* was broken. The authors provided a new working link when we notified them about this problem. All the working links of these three datasets are included with our replication package. Table 1 summarizes these three datasets. These datasets are used to answer three of the five research questions: RQ1, RQ4, and RQ5.

Table 1. Statistics of the three publicly available datasets. All the projects were written in Java.

Dataset	# Projects	# Code metrics	# Change metrics
Ferenc *et al.* [33]	15	37	0
Shippey *et al.* [101]	23	29	0
Mo *et al.* [74]	18	21	19

3.2 A new dataset

In RQ2, we investigate if a more accurate bug labelling approach can help the bug prediction models. Therefore, we need to change how a method is labelled as either buggy or not buggy. Unfortunately, none of the available datasets, described in Section 3.1, facilitates this experiment; they do not provide the raw data that was used to define a method's class: buggy or not buggy. Similarly, none of this datasets is suitable to answer RQ3, where we answer if a method's age and recent change history can be used to capture the concept drift involved in bug prediction models. To answer these research questions, we built a new dataset that we describe as follows.

3.2.1 Project selection. Research based on mining software repositories often relies on aggregated data analysis, where data from multiple projects are collected and merged to produce a single observation (e.g., [37, 74, 83, 105]). Aggregated analysis, however, has been shown to be inaccurate [22], because the observation can be highly influenced by very few large projects, masking observations from the smaller projects. As such, many research has focused on individual project analysis [22, 23, 56, 93, 100, 122]. This approach, however, has its own drawbacks: researchers can intentionally select projects that support their conclusion, which is known as selection or publication bias [37, 85].

We focused on individual project analysis, with an unbiased project selection approach, which was proposed by Chowdhury *et al.* [23]. We selected all the 49 projects that we found after joining the project sets of five different studies [37, 40, 82, 89, 105]. These are all Java projects, similar to the three publicly available dataset, thus alleviates the threats related to analyzing projects with different programming languages [121]. Table 2 shows the 49 projects and their number of methods. Approximately 19% of the collected methods are getters and setters, which may add noise to the analysis [5, 47, 67]. None of the conclusions of this study, however, change if we exclude them from our analysis. Therefore, for simplicity, they were included in our analysis. Our dataset still keeps the getters/setters information for any future experiment that may need them.

3.2.2 Collecting data. We need the change history of a method to understand if the method was involved in a bug fixing commit (RQ2), and if so, how old was the method during the bug fix (RQ3). To collect the change history of our 774,051 Java methods, we have used the state-of-the-art tool, CodeShovel [39, 40]. Unlike other method history collection tools, such as Historage [43] and FinerGit [51], CodeShovel does not require any expensive repository pre-processing. Also, CodeShovel exhibits much better accuracy compared to the leading research (e.g, FinerGit) and industry tools (e.g., IntelliJ / git log). For a given change commit of a method, CodeShovel captures who modified the method, what was modified, when it was modified, and why it was modified (i.e, commit message). It also returns the two source code

Table 2. Description of the 49 projects. In total, 774,051 methods were extracted, and 146,749 of them were simple getters and setters. To enable reproducibility, snapshot SHAs are presented as well.

Repository	# Methods	# Getters-Setters	Snapshot
hadoop	70,081	11,214	4c5cd7
elasticsearch	62,190	8,864	92be38
flink	38,081	6,764	261e72
lucene-solr	37,133	5,597	b457c2
presto	36,715	8,472	bb20eb
docx4j	36,514	19,951	36c378
hbase	36,274	5,743	3bd542
intellij-community	35,950	5,392	cdf2ef
weka	35,639	10,513	a22631
hazelcast	35,265	7,857	a59ad4
spring-framework	26,634	5,719	1984cf
hibernate-orm	24,800	5,647	2c12ca
eclipseJdt	22,124	2,093	475591
guava	20,757	499	e35207
sonarqube	20,627	4,152	6b806e
jclouds	20,358	3,533	7af4d8
wildfly	19,665	3,828	f21f5d
netty	16,908	631	662e0b
cassandra	15,953	1,005	7cdad3
argouml	12,755	1,789	fcbe6c
jetty	10,645	2,651	fc5dd8
voldemort	10,601	2,246	a7dbde
spring-boot	10,374	3,080	199cea
wicket	10,058	2,162	e3f370
ant	9,781	3,072	1ce1cc
jgit	9,548	1,476	855842
mongo-java-driver	9,467	1,576	8ab109
pmd	8,992	1,204	d115ca
xerces2-j	8,153	1,579	cf0c51
RxJava	8,145	59	880eed
openmrs-core	6,066	2,084	c5928a
javaparser	5,862	650	8f25c4
hibernate-search	5,345	912	5b7780
titan	4,590	485	ee226e
facebook-android-sdk	3,759	459	fb1b91
checkstyle	3,340	955	164a75
commons-lang	2,948	214	f69235
lombok	2,684	132	4fdcdd
atmosphere	2,659	328	fadfb0
jna	2,636	354	b8443b
Essentials	2,390	351	d36d80
junit5	2,085	203	be2aa2
hector	1,958	517	a302e6
okhttp	1,953	151	5224f3
mockito	1,498	144	077562
cucumber-jvm	1,146	169	b57b92
commons-io	1,145	85	11f0ab
vraptor4	926	132	593ce9
junit4	874	56	50a285
Total	**774,051**	**146,749**	

versions of a change commit: a method's code before the change and after the change. This facilitates us to calculate different code metrics of a buggy method.

From the raw data provided by CodeShovel, we captured information related to RQ2, and RQ3. For this purpose, we used our own tool—updated for this study—that was tested and used by multiple similar studies [1, 22, 23]. For source code parsing and calculating different code metrics, our tool used the JavaParser library.[3] Since an evolved method can have multiple values for a given code metric, our tool collected and saved data for all the versions of a given method. To make our work verifiable and reproducible, our replication package describes all the fields and their interpretations. In particular, we have collected code metrics focusing on size, testability, readability, dependency, and maintainability. Our objective (RQ2) is to observe how the ability of these code metrics improves in differentiating between buggy and not buggy methods with more accurately labeled training data.

Size. Size of a code component has been reported as the most dominant maintenance predictor by many [23, 30, 37]. We consider size as the source lines of code without comments and blank lines, as was also defined by others [23, 61, 88].

Testability. If a method is difficult to test, the method could be more bug-prone [45]. The popular McCabe [2, 24, 61, 110, 122] is often used as an estimation of the complexity and testability of a method. McCabe [70] is the measurement of the number of independent paths in a method, and the more independent paths a method has, the more difficult it is to test.

Readability. Reading source code is one of the most crucial activities in software development and maintenance [95]. A difficult to read method is believed to be more bug-prone [83]. To measure the readability of source methods, we have used the popular readability tool developed by Buse *et al.* [14].

Dependency. If a method (caller) relies on too many other methods (callees), the method would be more bug-prone, because a bug in any of the callees would automatically propagate to the caller method. This measurement is generally known as fanout [33, 74, 83], that we captured with our tool.

Maintainability. All the above code metrics capture some aspects of maintainability. However, to be more comprehensive, a composite metric, known as maintainability index (MI) [81], is often used. This composite metric is also adopted by different popular industry tools, such as Verifysoft technology [111], and Visual Studio [73].

3.3 Statistical tests

We randomly tested some of the distributions from our different datasets to verify if they are normally distributed. After applying the Anderson-Darling normality test [90], we found that none of them followed a normal distribution. Therefore, we adopted different non-parametric tests for our analysis. For example, to compare if two distributions are statistically different, we use the non-parametric Wilcoxon rank sum test, also known as the Mann Whitney U Test [71]. Similarly, we used the non-parametric Cliff's Delta calculator to calculate how significant is the difference between two given distributions [69]. These two tests are commonly used in software engineering research (e.g., [8, 18, 20, 46, 52, 84]).

4 RESULTS: REVISITING THE PAST

In this section, we conduct an in-depth investigation of the effectiveness of the previous MPBP approaches. In particular, we investigate if the existing MLBP models perform well when evaluated in realistic scenarios (RQ1).

Before evaluating the MLBP models in realistic scenarios on the datasets of Ferenc *et al.* [33], Shippey *et al.* [101], and Mo *et al.* [74], we first need to verify if we are accurately using their dataset similar to their original studies. Both

[3] https://github.com/javaparser/javaparser

Ferenc *et al.* [33] and Mo *et al.* [74] used Weka [117], a Java-based machine learning tool, for reporting their results with the 10-fold cross validation approach. Therefore, with the help of the *python-weka-wrapper*[4] library, we used Weka for reproducing their results. Unfortunately, Shippey *et al.* [101] did not build any model, but only provided a dataset as a benchmark for future MLBP research. Nevertheless, we use their dataset to see the accuracy with the cross validation approach so that we can compare this accuracy with the accuracies in realistic scenarios.

Table 3 compares our results with the original results of Ferenc *et al.* [33]. We have selected the top five machine learning algorithms, as mentioned in the original study. Although our reproduced results are similar to the original study, in most cases the accuracy is a little higher in the reproduced results. We have contacted the authors about the difference, and they found no problem with our approach, replying it could be due to an older Weka version that they used. Unfortunately, the only author who knew about the exact Weka version was unavailable.

Table 4 shows the accuracy of the five machine learning algorithms for the dataset of Shippey *et al.* [101]. Clearly, the observations are similar to the dataset of Ferenc *et al.* Unlike Ferenc *et al.*, Mo *et al.* only used the Random Forrest algorithm, and they reported the accuracy for each project separately. Also, the authors reported accuracy only in *area under roc curve (AUC)*. While we compare our AUC results with the original AUC, we also present results in other accuracy metrics. Table 5 shows the detail. Our reproduced results (AUC score) are almost identical to the original results.

Table 3. Reproducing the results for the dataset of Ferenc *et al.* [33] with 10-fold cross validation. **O** is for original, and **R** is for reproduced.

	Precision		Recall		F-measure	
Algorithm	**O**	**R**	**O**	**R**	**O**	**R**
Random Forest	0.633	0.648	0.632	0.720	0.631	0.682
J48	0.614	0.617	0.613	0.738	0.611	0.672
Random Tree	0.611	0.650	0.611	0.674	0.611	0.662
SimpleLogistic	0.606	0.588	0.604	0.605	0.603	0.597
DecisionTable	0.613	0.583	0.607	0.759	0.598	0.659

Table 4. Results, with 10-fold cross validation, for the dataset of Shippey *et al.* [101]. No comparison can be made because Shippey *et al.* [101] did not provide any accuracy result.

Algorithm	Precision	Recall	F-measure
RandomForest	0.695	0.741	0.717
J48	0.680	0.664	0.672
RandomTree	0.662	0.669	0.665
SimpleLogistic	0.657	0.607	0.631
DecisionTable	0.658	0.625	0.641

As we are now confident that we are using the datasets as intended in the original studies, we now evaluate these approaches and datasets in three realistic scenarios. The realistic scenarios are constructed as follows. When no history data of a test project is available, practitioners can build a cross project defect prediction model. When history data is

[4]https://fracpete.github.io/python-weka-wrapper/

Table 5. Reproducing the results for the dataset of Mo *et al.* [74] with 10-fold cross validation.

Project	Precision	Recall	F1	AUC	
				Original	Reproduced
ActiveMQ	0.73	0.74	0.74	0.87	0.87
Ignite	0.61	0.32	0.42	0.90	0.90
Nutch	0.73	0.58	0.65	0.82	0.82
Camel	0.67	0.38	0.49	0.87	0.87
Flume	0.75	0.86	0.80	0.81	0.81
Struts	0.69	0.69	0.69	0.86	0.85
Maven	0.66	0.57	0.61	0.89	0.89
Kafka	0.6	0.46	0.52	0.83	0.83
Zookeeper	0.73	0.75	0.74	0.81	0.80
Avro	0.74	0.6	0.66	0.83	0.84
Drill	0.72	0.71	0.71	0.82	0.82
Wicket	0.68	0.65	0.66	0.89	0.89
Flink	0.65	0.43	0.52	0.79	0.80
Hbase	0.73	0.79	0.76	0.82	0.82
Calcite	0.64	0.67	0.66	0.74	0.74
CXF	0.66	0.45	0.53	0.88	0.87
Cassandra	0.7	0.75	0.73	0.83	0.82

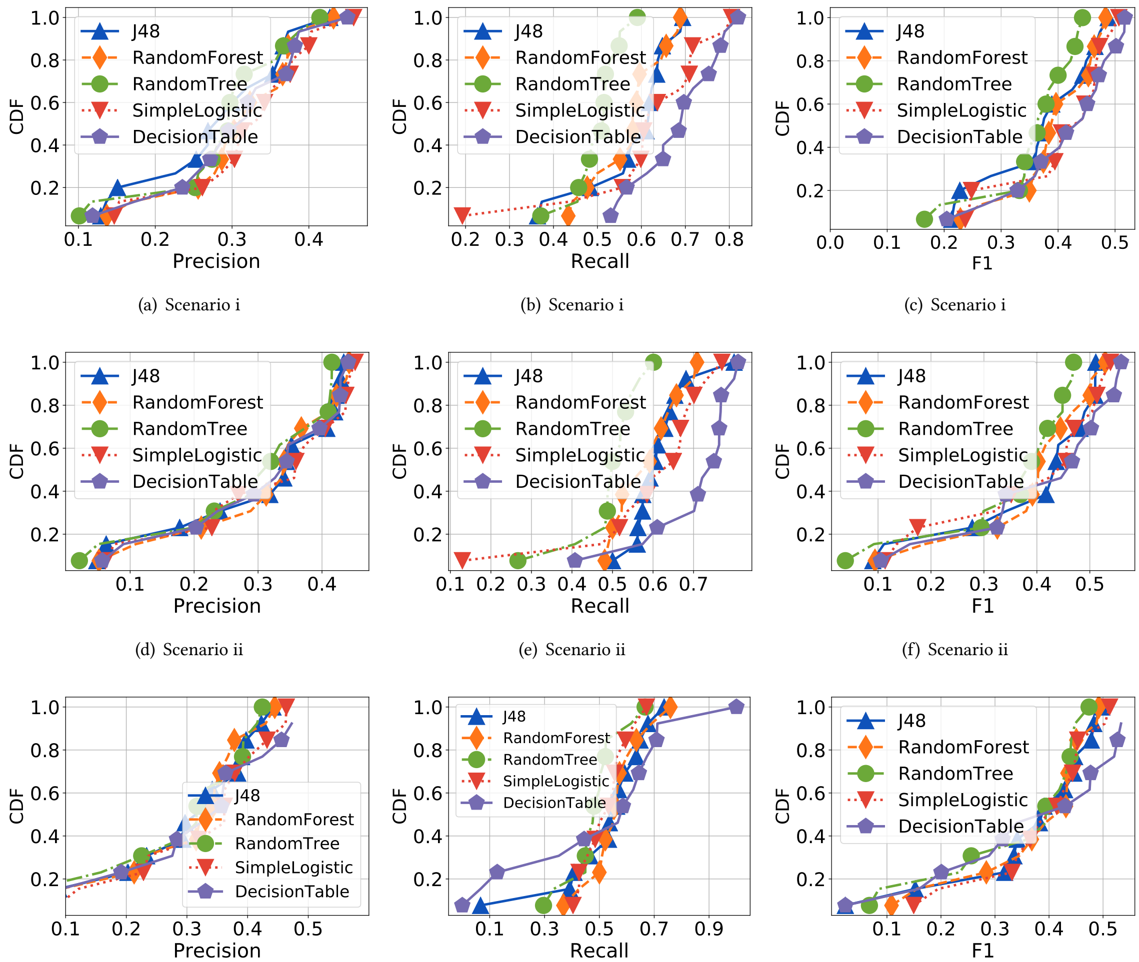
available, practitioners have two more options for building the training data: they can mix the history data of the test project with other projects' data, or they can use only the history data of the test project.

- *scenario i (cross project).* No data from the test project is used in training.
- *scenario ii (cross + past).* Data from all projects and history data of the test project are used for training to predict future bugs of the test project.
- *scenario iii (only past).* This is the same as ii, except only the history data of the test project is used for training.

Figure 1 shows the Cumulative Distribution Function (CDF) of different accuracy metrics for the dataset of Ferenc *et al.* Although the original dataset does not contain the time information required for the accuracy evaluation in scenario *ii (cross + past)* and *iii (only past)*, we have collected this information by a python script that used the SHAs available with the dataset. In all three scenarios, the precisions and the F1 scores are significantly lower than the original results with the cross validation approach (Table 3). For example, for the scenario *i (cross project)* prediction (Figure 1 (a)), precision was within only ∼0.37 for 80% of the projects, considering all the machine learning algorithms. The F1 score was within ∼0.50 (Figure 1 (c)) for 80% of the projects, even with the most accurate Decision Table algorithm. This degraded performance does not improve for the other two scenarios.

Figure 2 shows the results for the dataset of Shippey *et al.* [101]. Unlike the dataset of Ferenc *et al.*, this dataset contains the release numbers. Therefore, for scenarios *ii (cross + past)*, and *iii (only past)*, all the release versions before the release version x were used in training, when the release version x was used for testing. The performance in precision and F1 is now even worse. For example, in *scenario i (cross project)*, the precision and F1 were within ∼0.13 and ∼0.25, respectively. Although the performance improve in scenarios *ii (cross + past)* and *iii (only past)*, they are still low compared to the results with the cross validation approach (Table 4).

Figure 3 shows the results for the dataset of Mo *et al.* Unfortunately, this dataset does not contain any timing information, which obstructed our analysis for scenarios *ii (cross + past)* and *iii (only past)*. The results, at the first glance,



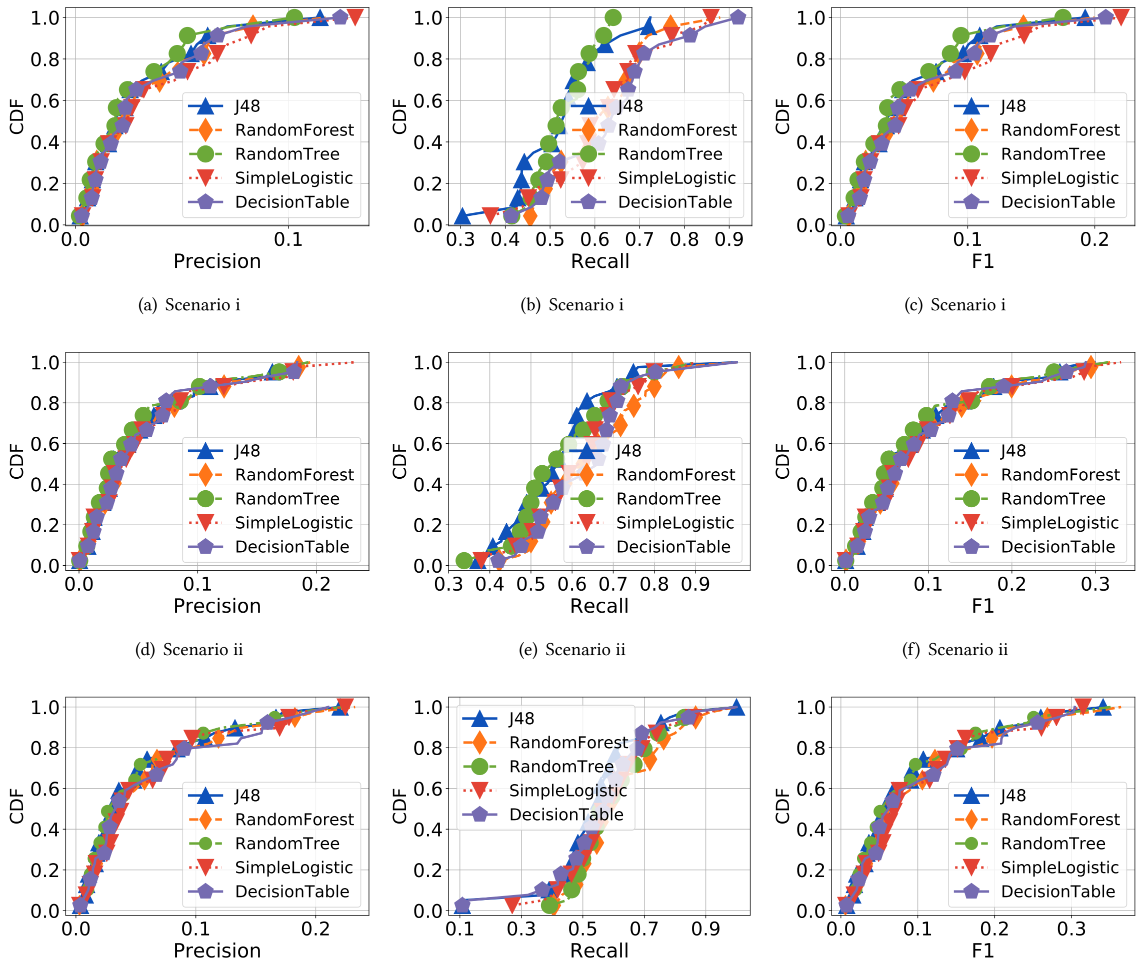
(a) Scenario i (b) Scenario i (c) Scenario i

(d) Scenario ii (e) Scenario ii (f) Scenario ii

(g) Scenario iii (h) Scenario iii (i) Scenario iii

Fig. 1. Results for the dataset of Ferenc *et al.* in three realistic scenarios. For graph readability, the number of markers is kept less than the number of data points. The first row shows the cross project prediction results, where nothing was used from test projects during training (scenario *i*). The second and third rows show the results for scenarios *ii (cross + past)* and *iii (past only)*, respectively. For these two scenarios, data from the first half of the test project's lifetime was used in training, and the last half was used for testing. We tried few other time splits with no noticeable change in our observation.

are encouraging. Although the accuracies are generally lower compared to the cross validation approach (Table 5), this dataset produces significantly better accuracies than the other two datasets. Unfortunately, we found some crucial problems with the construction of the dataset itself, which unduly boosted the accuracy. The authors have collected the change history of a method from its whole lifetime, and then defined the method either as *buggy* or *not buggy*. In the real world, we do not have access to the future change information when we predict the future bug-proneness of a method; we have to predict the future only by using the past. Consider the independent variable *number of changes* used by the study. If a method's *number of changes* is zero in its whole lifetime, then the method was definitely labelled



(a) Scenario i (b) Scenario i (c) Scenario i

(d) Scenario ii (e) Scenario ii (f) Scenario ii

(g) Scenario iii (h) Scenario iii (i) Scenario iii

Fig. 2. Results for the dataset of Shippey *et al.* [101] in three realistic scenarios. The first row shows the results for the scenario *i (cross project)*. The second and third rows show the results for scenarios *ii (cross + past)* and *iii (past only)*, respectively.

as *not buggy* in the dataset, because it was never associated with a change commit, let alone with a bug-fixing commit. Problem is, to make it work, we have to time travel in the future to know if a method will have zero change in its lifetime. Also, building models where a method is labelled as *buggy* for eternity is impractical, because it may become *not buggy* after one or more bug-fixing processes [65]. These kinds of incorrect approaches were reported to be the root cause of the failure of bug prediction models in industrial practice [62].

Summary: Previous MLBP models were evaluated with the unrealistic cross validation approach. In some cases, training data was constructed using information from the future that would be unavailable in any realistic scenario. When evaluated with different practical scenarios, the performance of MLBP is extremely poor. Our conclusion, based on robust analysis, confirms earlier findings [83] that method-level bug prediction is an open research problem.

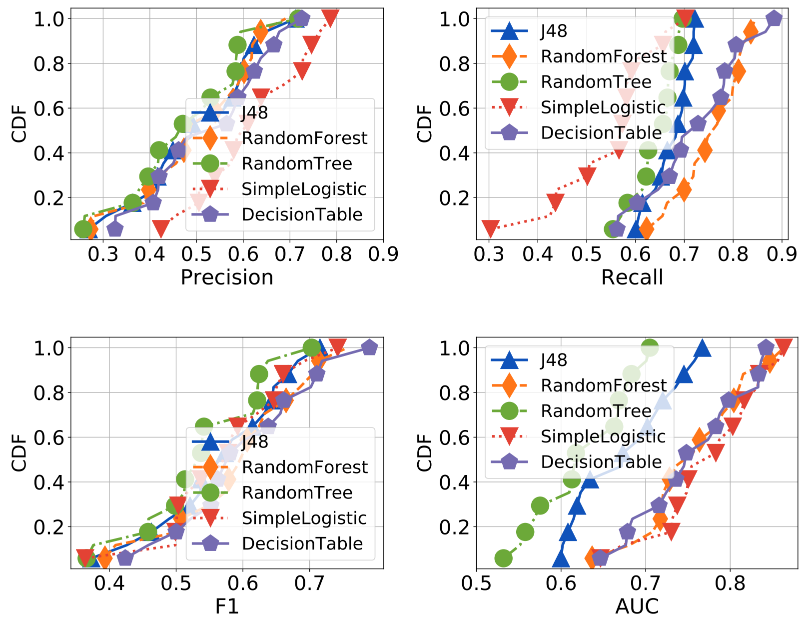


Fig. 3. Results for the dataset of Mo *et al.* for *scenario i*. Result for AUC is also added to be consistent with the original study. This dataset contains no time information, thus preventing us to perform the evaluations for *scenarios ii* and *iii*.

5 RESULTS: POTENTIAL IMPROVEMENT AVENUES

The futility of the existing approaches led us to explore some potential improvement avenues for future MLBP models. In this section, we discuss a quartet of such avenues by answering RQ2 to RQ5.

5.1 RQ2: How important is accurate bug labelling for the success of future MLBP models?

Machine learning models perform poorly when trained on a noisy dataset [41]. While there are several types of noises [123], such as providing inaccurate values for different attributes, noisy data due to mislabelled instances has been the most crucial [41]. Removing mislabelled classes from the training data improves prediction accuracy significantly [91, 123]. As we have mentioned in Section 2.4, bug prediction datasets are susceptible to noise, due to their unconditional credence in finding bug-fix keywords in the commit messages, and due to the developers induced tangled changes.

Problems with Keywords. Let us consider the bug-fix keywords used in the study of Ray *et al.* [89]. The authors have used different forms of nine keywords—*error*, *bug*, *fix*, *issue*, *mistake*, *incorrect*, *fault*, *defect*, and *flaw*—such that if a commit message contains any of these keywords, that commit is considered as a bug-fix commit. Our manual analysis, with 500 randomly selected commits, reveals some problems with this keyword set. For example, the word *issue* in a commit message does not necessarily mean it is a bug-fix commit. It may mean a quality improvement (e.g., jcloud, commit hash 4c83585, commit message: *fixed some quality issues*), or even an enhancement (e.g., hazelcast, commit hash 51675d81, commit message: *issues/13540: Transaction Propagation support implementation (15141)*). As such, all the methods that were changed in these commits will incorrectly be labelled as buggy methods. This problem is common for other keywords as well. For example, although the commit message *Add default /error view for HTML clients* (Spring-boot, commit sha 5211747) contains the word *error*, this commit was for an enhancement, not for a bug fix.

Problems with tangled Changes. From our dataset, described in Section 3.2, we calculate how many methods are modified in each bug-fix commit, and show the cumulative distribution function in Figure 4. In ~40% of the bug-fix commits, only one method was modified, which is good for producing a less noisy bug training dataset; if only one method is modified in a true bug-fix commit, that method is definitely buggy. However, in more than ~15% of the bug-fix commits, at least 10 methods were modified. There are commits that even modified more than 100 methods (or even more than 1000 methods in extreme cases). If a bug-fix commit modifies 100 methods, and only 10 of them were actually buggy, 90 methods would be mislabelled as buggy methods. Our CDF graph shows that in ~60% of the commits, more than one method was modified, and we do not know how many of the methods were actually modified for bug-fix. Therefore, the probability of having too many mislabelled buggy methods in the previously built datasets (e.g., [33, 74, 83]) is high.

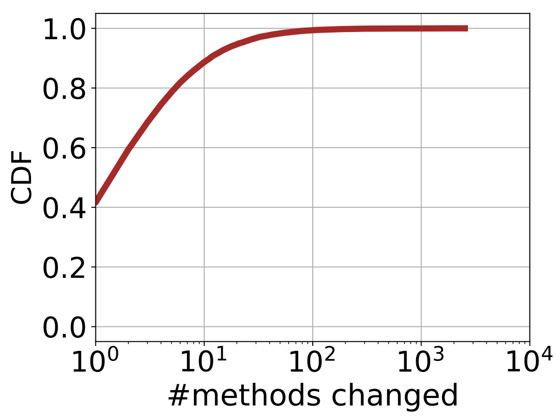


Fig. 4. Cumulative distribution function of the number of methods modified in a bug-fix commit.

Problems with not buggy methods. Traditionally, if a method was never modified with a bug-fix commit, that method is labelled as not buggy [35, 74]. This approach is unreliable in at least two scenarios. i) If a method was introduced just before the data collection process, that method would automatically be labelled as not buggy, although there was no time to observe this method's evolution. ii) Consider that a complex method was committed to a system. However, after just one week, a developer noticed the method, then refactored and improved its quality. Due to the reduced complexity, the method was never associated with a bug-fix commit, therefore would be labelled as not buggy. The problem is, this method now has two versions. It is acceptable to label the refactored version as not buggy, but labelling the first version of the method as not buggy could be problematic. Similar to scenario i, the first version did not have enough time to be labelled as buggy, in case it was indeed a buggy method.

Can these too many sources of noise in bug data explain the poor performance of the previous bug prediction models (**RQ2-1**)? Can a more accurate approach to building bug data improve bug prediction (**RQ2-2**)? To answer these questions, we built two datasets: i) Traditional, and ii) Accurate.

Traditional dataset. Similar to earlier studies, a method is buggy if it was modified in a bug-fix commit. To identify a bug-fix commit, we adopt the keywords used by Ray *et al.* [89]. Likewise, a method is not buggy if it was never modified in a bug-fix commit.

Accurate dataset. To build an accurate dataset, we focused on the precision for labelling a method either as buggy or not buggy. Our developed approach is based on our manual inspection on 500 randomly selected potential bug-fix commits. We label a method as buggy, *if it is the only method that was modified in a commit containing a bug and a fix related keyword in the same sentence of the commit message.* The details are as follows.

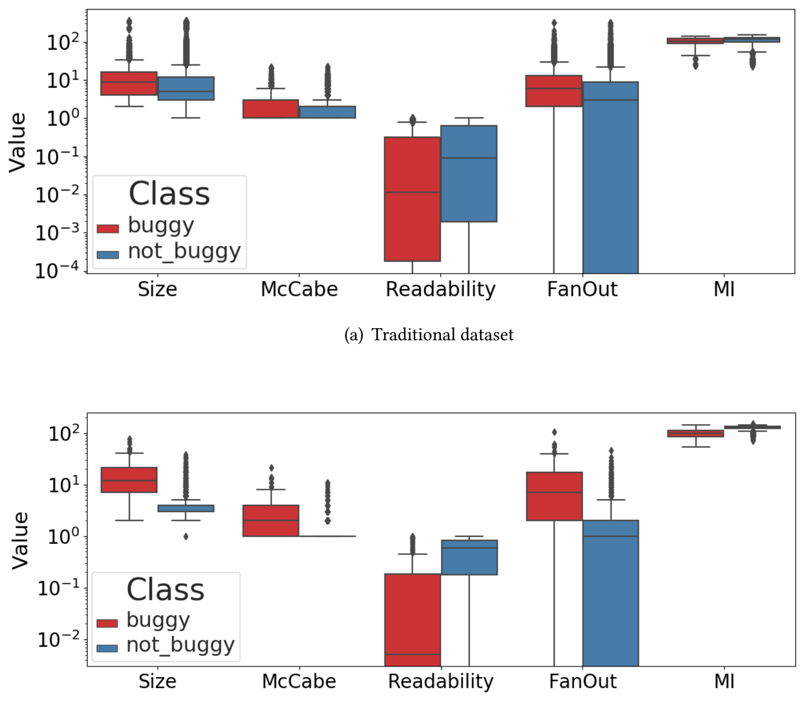
- *Bug words.* We deleted the keyword *issue* from the keyword set of Ray *et al.*, because it produces too many false positives. Instead, we added the word *misfeature* to the list, as suggested by Rosa *et al.* [94].
- *Fix words.* The presence of a bug word alone is not sufficient for detecting bug-fix commits [94]. To achieve high precision, the same sentence containing a bug-related word must contain a fix-related word as well. Leveraging our manual analysis and the existing research, we selected different forms of five fix related words: *fix*, *solve*, *resolve*, *repair*, and *address*.
- If multiple methods were modified in a single commit, we do not know which methods are related to bug-fix, and which are the results of tangled changes. Therefore, we discarded such commits from our dataset.
- If a method has multiple versions, only the modified version in a bug-fix commit is buggy.

A method's version is labelled as not buggy, *if it was unchanged at least for two years, and none of the future versions was associated with a change commit containing any of the bug or fix related keywords we mentioned in this paper.* The rationales are as follows.

- We cannot label a method as not buggy, just because it was never associated with a bug-fix commit. Perhaps, it is a new method, and its bug will be revealed in the future. Therefore, a method has to be unchanged at least for two years. Chowdhury *et al.* [22] showed that if a method is unchanged for two years, the probability of its future change is low.
- If a specific version of a method was not associated with a bug-fix commit, but a later version was, we cannot guarantee the previous version was bug-free, because the bug may have propagated to the later version.
- While the existence of a single bug or fix related keywords in a commit message does not always indicate a bug-fix process, in many cases they do. Therefore, we can not label any method as bug-free that was associated with a commit containing any of these potentially bug or fix-related keywords, including the word *issue*.

Let us now consider the sonarqube project (Table 2) as an example. For this project, Figure 5 shows the distribution of the five code metrics, discussed in Section 3.2.2, after grouping them for the buggy and the not buggy methods. In the traditional noisy dataset (Figure (a)), all the distribution differences are statistically significant, according to the Wilcoxon rank sum test. However, all of these differences have small effect sizes according to the Cliff's delta calculator. For the accurate dataset (Figure (b)), however, all the differences have large effect sizes, not to mention that the differences are statically significant as well. *Do these observations generalize for all 49 projects?* Table 6 shows the results. We had to exclude six projects from our analysis because they did not have enough samples (i.e., at least 10 samples) for the buggy class when calculated for the accurate dataset.

For the traditional noisy dataset, the distribution differences are mostly within negligible or small effect sizes. This means, even the popular explanatory variables struggle to distinguish between the buggy and not buggy methods. This clearly explains the poor performance of the previous MLBP approaches (**RQ2-1**), because this is how their datasets were constructed.



(a) Traditional dataset

(b) Accurate dataset

Fig. 5. Distribution of different code metrics in the buggy and not buggy Java methods from the Sonarqube project. The distributions are significantly more different in the accurate dataset than in the traditional dataset.

In contrast, most of the differences have large effect sizes in the accurate dataset. For example, in the traditional dataset, in 30.61% of the projects (15 projects), the Cliff's delta effect size is negligible between the size distributions of buggy and not buggy methods. This effect size is large for 97.67% of the projects in the accurate dataset. This significantly different behaviour between the two datasets is true for other code metrics as well. This implies that future MLBP models can be improved significantly with our conservative accurate labelling approach (**RQ2-2**).

Summary: The impact of accurate bug labelling on the future MLBP models can be enormous. Previous MLBP models were trained on inaccurate noisy datasets, leading to poor prediction performance. We showed that future bug prediction models can be improved significantly by a more careful bug labelling approach.

5.2 RQ3: Can a method's age capture the inevitable *concept drift* in bug prediction?

Previous studies have observed a positive correlation between change- and bug-proneness [10, 11, 76, 77, 87]. We first investigate if this observation is true at the method-level granularity. To understand the change history of a method, we captured four different change-proneness indicators from the first five years of a method's lifetime. We then captured if the method was buggy or not, in the next five years. The four indicators are, i) number of revisions, ii) diff size, iii)

Table 6. Cliff's Delta Effect sizes for the difference in code metrics distribution between the not buggy and buggy methods. N refers to Negligible, S refers to Small, M refers to Medium, and L refers to Large effect size. For example, 30.61 for N means, in 30.61% of the projects the cliff's delta is negligible.

	Traditional				**Accurate**			
Metrics	N	S	M	L	N	S	M	L
Size	30.61	57.14	12.24	0.0	0.0	2.33	0.0	97.67
Readability	44.9	48.98	4.08	2.04	0.0	2.33	4.65	93.02
McCabe	42.86	48.98	8.16	0.0	0.0	2.33	0.0	97.67
FanOut	32.65	57.14	10.2	0.0	0.0	2.33	0.0	97.67
MI	24.49	61.22	14.29	0.0	0.0	2.33	0.0	97.67

number of added lines only, and iv) Levenshtein edit distance [64]. Chowdhury *et al.* [22] observed that these four change indicators together can provide a comprehensive view of a method's true change history.

Figure 6 shows that the buggy methods are more change-prone than the not-buggy methods. For all four change indicators, the distribution differences are statistically significant according to the Wilcoxon rank sum test, and the effect sizes are large according to the Cliff's delta calculator. Therefore, change history is indeed a good predictor for method-level bug prediction, as was also claimed in other studies [74, 83].

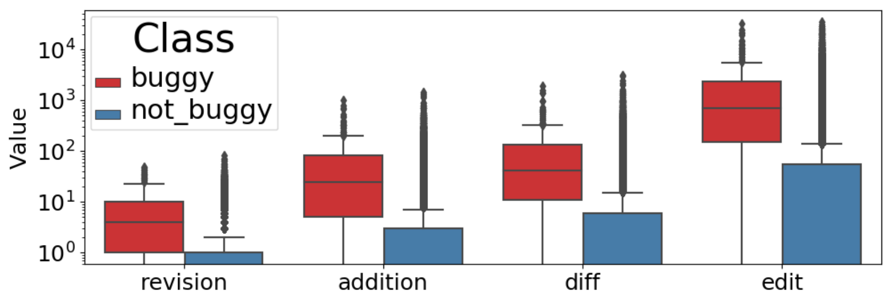


Fig. 6. Distribution of different change metrics in the buggy and not buggy methods in the accurate dataset. Change metrics were captured from the first 5 years, whereas bug information was captured in the next five years. Data was aggregated from all 49 projects. Methods that are not at least 10 years old were excluded from this analysis. Observations are similar for individual project analysis as well.

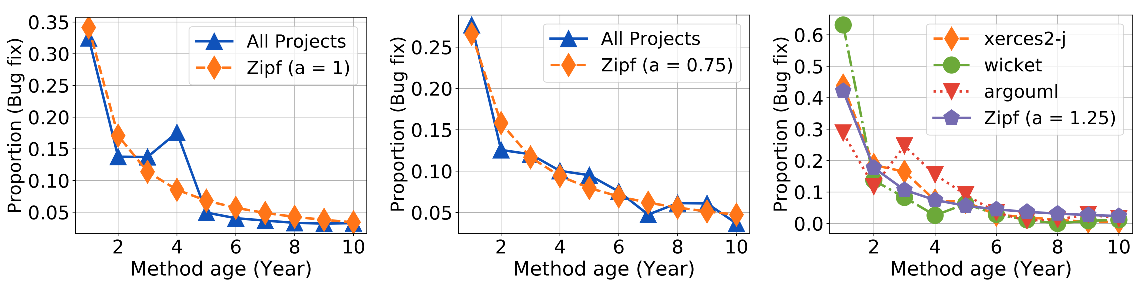
The problem is, due to concept drift [12, 29, 55, 113], the correlation between change- and bug-proneness fluctuates with time [65], inducing unstable bug prediction performances [8]. This problem can be alleviated by retraining the model each time a concept drift occurs [113], which is, of course, laborious and time-consuming. A cheaper alternative would be to use one or more variables that can guide the time-varying prediction of a model. According to a Google study [65], bug-proneness is better modelled with the recent change history than with the complete change history. This makes recent change history a potential candidate to model concept drift. Unfortunately, the differences in Figure 6 between buggy and not buggy methods reduce if we collect the change history from recent times (e.g., from the last three years), instead of collecting it from the whole five years. The Cliff's delta effect sizes become medium from large.

We have also experimented with other scenarios. For example, we have captured bug information between the eighth and the tenth year of a method's life. We then captured the last two years' change history (i.e., changes between years six to eight), and the total change history. For all four change indicators, the effect sizes between the buggy and the not buggy methods were lower with the recent change history compared to the complete change history. *Is it surprising that recent change history was helpful at Google (File level), but not in our study (method level)?* Perhaps not. Previous studies have reported about these contradictory observations between file/class and method-level granularity [22, 61].

Given that recent change history does not help us capture concept drift at the method level, we now focus on method age. What if a method's age can explain its time-varying bug-proneness? Perhaps, a complex old method is less (or more) bug prone than a complex new method. Probably, if a method is old, its bug-proneness is low although it has undergone massive changes in its early life stages. To investigate this, we have captured how old was a method during its involvement with bug-fix commits. If a method was modified with multiple bug-fix commits, we recorded all the corresponding ages. The relevant threats with this analysis are discussed in Section 7.

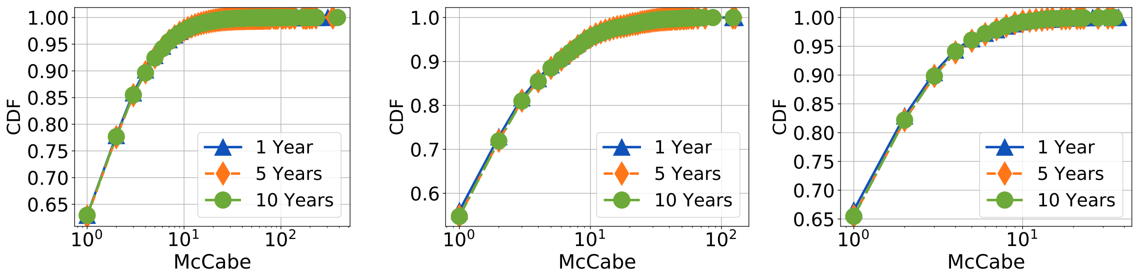
Figure 7 shows that a method's bug-proneness indeed decays over time. The decaying patterns are so consistent that they can be modelled by a power law distribution, such as the Zipf's law [79]. This is encouraging because future MLBP models can consider a method's age to potentially address concept drift, without the laborious and time-consuming model retraining.

A valid skepticism of using method age. The complexity of a method reduces over time, due to the regular perfective and preventive maintenance activities [16, 17]. Since code complexity is a good bug predictor (Section 2.4), the decaying bug-proneness over time is probably due to the reduced code complexity. In that case, method age would not be a good predictor to understand concept drift. A more complex method would be more bug-prone regardless of its age. To verify this, we captured the McCabe complexities of all the methods in three different times: when the methods were 1 year, 5 years, and 10 years old. Figure 8 shows that the complexity distributions did not change over time. According to the Wilcoxon rank sum test, none of these distributions are statistically different from each other. This validates the usefulness of using a method's age in capturing its concept drift.



(a) Accurate dataset (b) Traditional dataset (c) Individual projects

Fig. 7. The bug proneness of a method dwindles as it gets older. Figure (a) is for the accurate dataset, and figure (b) is for the traditional noisy dataset. The little zigzag pattern in Figure (a) is due to the less number of data points than in Figure (b). As bug-related data in each year for each project individually is rare, Figure (a) and (b) show results for the aggregated data. However, in (c), results are shown separately for three individual projects that had much more year-by-year bug data than the other projects. In all cases, the decay patterns can be modelled with the Zipf's Law equation.



(a) All projects (b) Xerces2-j (c) Wicket

Fig. 8. McCabe distribution over time. Figure (a) is for the aggregated data. The results are similar for each of the individual projects. For brevity, we show results for two of them in Figure (b) and Figure (c).

Summary: A method's bug-proneness decays continuously as the method ages. A long-lived buggy method has probably undergone enough bug-fixing processes and become a bug-free method. Therefore, bug-proneness cannot be explained just by using the code complexity and change history of a method. A method's age must be used to capture the concept drift necessary for a realistic and accurate bug prediction model.

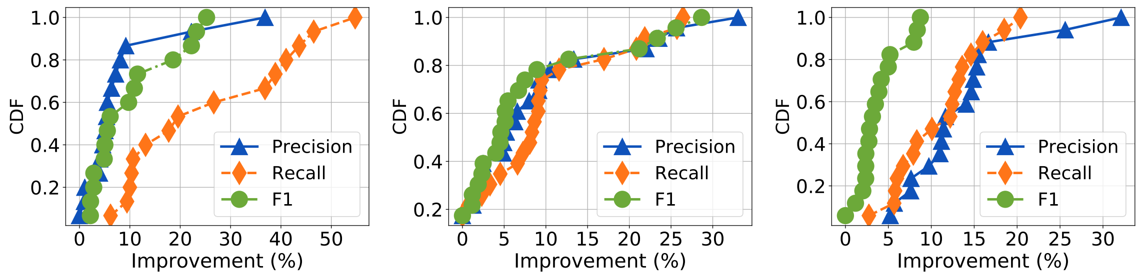
5.3 RQ4: Should future MLBP research focus on an optimal project set selection?

Studies, at higher than the method-level granularity, observed that judicious selection of training projects (or project versions) can significantly improve the accuracy of bug prediction models [6, 107, 113]. This judicious selection is commonly based on some similarity scores between the test project and the training projects. We investigate if this approach can indeed benefit MLBP models so that researchers can build and evaluate such similarity functions for the method-level granularity.

Our hypothesis is that, *if using only a subset of the training projects produces a more accurate prediction model than using all the training projects, then this indicates that optimal project set selection should improve MLBP accuracy.* To better understand our hypothesis, let us consider a test project p_t and two sets of training projects $A = p_1, p_3, p_5$, and $B = p_1, p_2, p_3, p_4, p_5$. Here, A is a subset of B. Now, if the bug prediction accuracy on p_t is higher with A than with B, this means adding p_2 and p_4 in the training set did more harm than good. This clearly implies that adding more projects to the training data does not necessarily improve prediction accuracy, and optimal project selection can indeed help MLBP models.

For our experiments, we used all three public datasets, described in Section 3.1. We followed the cross-project bug prediction approach, where the base model uses all the training projects. For experimenting with the subset projects, we randomly selected x number of training projects, such that $x = 1, 3, 5, 7, 10$. For example, when $x = 5$, we randomly sampled five training projects, without replacement. We then trained a model with these five projects and compared its accuracy with the base model. This experiment was repeated three times so that three different sets of five projects were evaluated. The same approach was followed for all the x values.

For most of the subsets, in all three datasets, the accuracy was significantly lower than the base model. This supports the traditional presumption that *more training projects generally lead to more prediction accuracy.* However, for all the test projects from all three datasets, there was always one or more subsets that had higher accuracy than the base model. This observation is presented in Figure 9. Clearly, all the accuracy scores (precision, recall, and F1) have improved



(a) Ferenc *et al.* (b) Shippey *et al.* (c) Mo *et al.*

Fig. 9. Improvement results for all three datasets. For each test project, there is at least one smaller set of training projects that leads to better accuracy than using all the training projects. When there are multiple subsets with improved accuracy, the maximum improvement was selected. For simplicity, results are shown only for the Random Forest algorithm, but the observations are the same as the other algorithms.

significantly for all the projects. For example, for the dataset of Ferenc *et al.*, the recall has improved by at least 10% for 60% of the projects. For the dataset of Shippey *et al.*, all the scores have improved by at least 10% for 20% of the projects. These improvements can potentially be higher if we test with more subsets. A potential approach to select optimal training projects is discussed in Section 6.

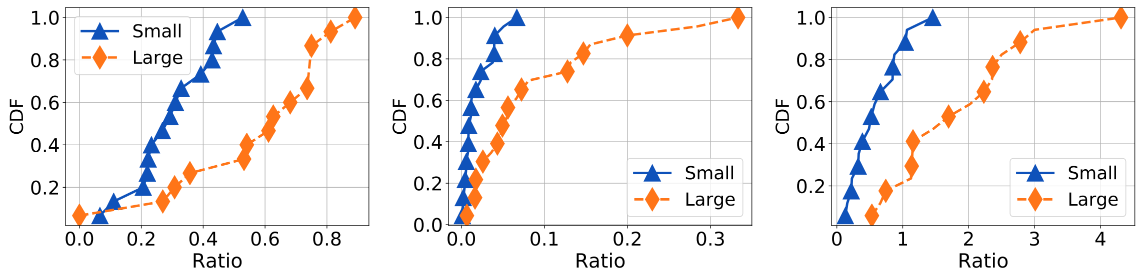
Summary: Future MLBP models should take advantage of optimal project set selection, because this may significantly improve bug prediction performance. However, finding optimal training projects for a given test project is an open research problem for MLBP. In Section 6, we provide a potential layout in this direction.

5.4 RQ5: Should future MLBP research focus on a mixture of models?

As we have discussed in Section 2.7, a mixture of models improves prediction accuracy when there are multiple distinct populations (or clusters) in a single dataset. It is inaccurate to model such a dataset with a single set of parameters. In this RQ, we investigate if future research should model bug proneness separately for small and large methods.

Chowdhury *et al.* [23] followed the six steps procedure of Alves *et al.* [4] for empirically deriving and evaluating the size boundaries of small ($Size \leq 24$), medium ($25 \leq Size \leq 36$), large ($37 \leq Size \leq 63$), and very large methods ($Size \geq 63$). Alves *et al.* [4]'s approach does not depend on intuition or expert opinions (which are generally debatable), and is robust to outlier projects. The three public datasets for our analysis, however, do not have enough buggy methods in each size category. Therefore, we considered any method with $Size \leq 36$ and $Size \geq 37$ as a small and large method, respectively.

We first investigate if the ratios of the buggy and not buggy methods in a project (i.e., # of buggy methods /# not buggy methods) are different in small and large methods. If so, then small and large methods should be treated as two different clusters, and we should not model them together. Figure 10 shows the results for all three datasets. The group of small methods has a much less number of buggy methods than not buggy methods. For example, in the dataset of Ferenc *et al.*, the ratios for the small methods are ≤ 0.4 for ~80% of the projects. But for large methods, the ratios are ≤ 0.4 for only ~20% of the projects. Evidently, the class imbalance issue is much more prevalent in small methods than in large methods. Therefore, applying the over-sampling or the under-sampling approach on the whole dataset, as followed by the previous studies [33, 83], would not solve the problem. Most of the buggy methods would be drawn from the set of large methods, and the not buggy methods would be drawn from the set of small methods.



(a) Ferenc *et al.* (b) Shippey *et al.* (c) Mo *et al.*

Fig. 10. Cumulative distribution functions of the ratios of the buggy and not buggy methods for all three public datasets.

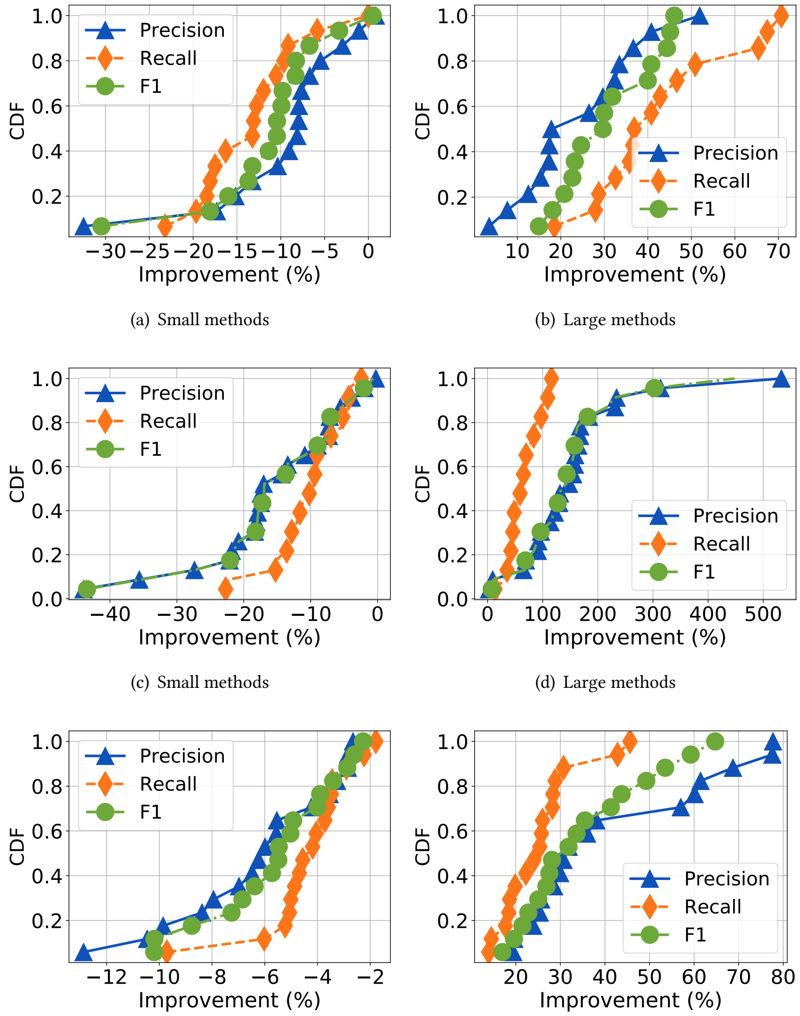
According to our observation, accurate bug prediction should be much more difficult in small methods. To verify this, we captured the prediction accuracy on small methods and large methods separately, and then calculated the accuracy gain (or loss) compared with the prediction accuracy on the whole dataset (base accuracy). For example, if precision is 0.3 on the whole dataset, but 0.5 when evaluated on the large methods only, that is a 66.66% improvement in precision. Figure 11 shows the results. Clearly, compared to the base accuracy, the prediction accuracy is always lower for small methods, and higher for large methods. For example, when evaluated only on the small methods in the dataset of Ferenc *et al.* (Figure 11 (a)), precision and F1 scores dropped more than 10% for 40% of the projects. In contrast, all the scores improved by at least 10% for 80% of the projects when evaluated on large methods only (Figure 11 (b)).

Summary: Characteristics of small and large methods are significantly different from each other: they are different in their code metrics variability [22], and in their bug-proneness distributions. Future MLBP research should group them into two different clusters so that both the pre-processing (e.g., under-sampling) and model training are done separately.

6 DISCUSSION

We have established that the reported high accuracies of previous method-level bug prediction research are inaccurate (RQ1). The accuracies were unduly boosted through information leakage, as future data was used during the model training phase (or even during the data construction phase). In realistic scenarios, the performances of those models and approaches are extremely poor. Method-level bug prediction thus remains an open research problem. We then studied and provided four potential guidelines (RQ2 to RQ5) that can benefit future method-level bug prediction research. The guidelines are as follows.

- Future method-level bug prediction should focus on accurate bug labelling. They can adopt our conservative labelling approach (RQ2), or improve it even further.
- They must adapt to concept drift, and can use a method's age (RQ3) that captures the continually decaying bug-proneness of the method.
- Instead of solely focusing on how to enlarge training data, future research should focus on selecting training projects that are similar to a given test project (RQ4).



(a) Small methods (b) Large methods

(c) Small methods (d) Large methods

(e) Small methods (f) Large methods

Fig. 11. The first row shows the results for the dataset of Ferenc *et al.* The second and the third rows are for the datasets of Shippey *et al.* and Mo *et al.*, respectively. For all the datasets, prediction accuracy is much weaker in small methods. Results are shown only for the Random Forest algorithm, but the observations are similar for all of them.

- Due to their distinct characteristics, small and large methods should be grouped and treated separately (RQ5). For example, instead of one single generic model, at least two different models should be built—one for each size group.

Our conservative bug labelling approach (RQ2), however, reduces the number of samples in both classes. But, for accurate model training, a small accurate dataset is often better than a large noisy dataset [31, 41, 119]. In addition, by leveraging different open-source platforms, such as GitHub, and tools like CodeShovel [40], we can collect as many accurate samples as required. One other challenge to follow our guidelines is not knowing how to select similar projects (RQ4). Recent research has investigated the effectiveness of the bandit algorithms [6], distribution of bug prediction data [113], and collaborative filtering [107] for selecting similar training projects. However, due to the different code granularity, the effectiveness of these approaches is unknown in method-level bug prediction.

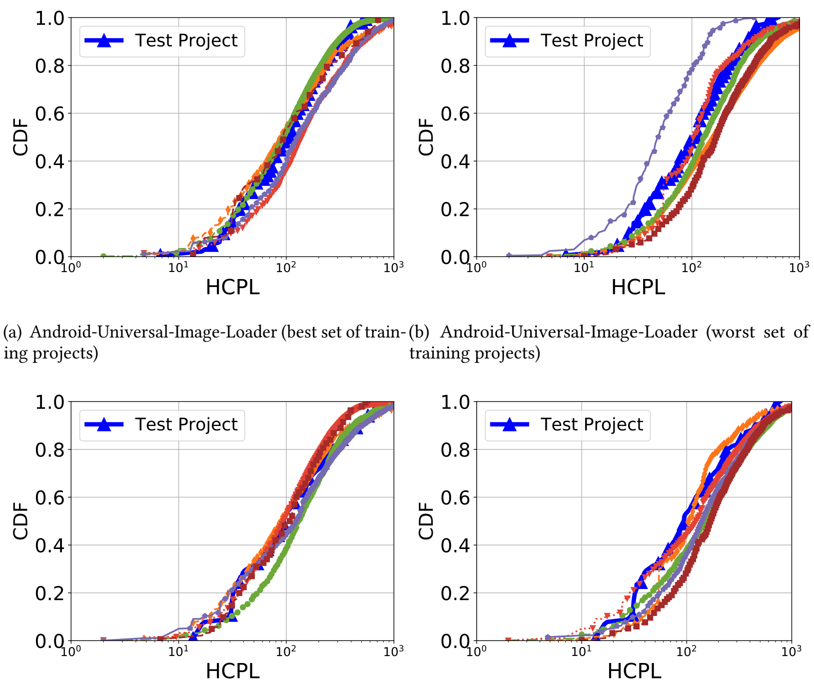
We envision a dedicated future study solely on similar training project selection—selecting a set of training projects that are similar to the test project in a way that produces higher bug prediction accuracy. The challenge is to find a metric (or a set of metrics) that can be used as a proxy for project similarity. We can consider two projects as similar if most of their source methods are similar according to code clone analysis. However, this requires comparing each method of a project with every method from the other project, which would be extremely time-consuming especially when we require a significant number of training projects for building a robust model. Also, even if two projects share many similar methods, the number of bugs and their fix patterns can still be substantially different based on the number and expertise of the contributors of those two projects. Therefore, future studies can investigate if developer-centric information can be used for selecting similar training projects. None of our datasets contains developer-related information, restricting us from such an analysis. Another potential approach to define similarity is to use different code quality indicators—if two projects have similar code quality, perhaps they have similar numbers and types of bugs. To investigate if such an approach can be useful, we consider three source code metrics: LOC (source lines of code), McCabe (cyclomatic complexity), and HCPL (Halstead calculated program length). These three metrics are widely used in software quality and maintenance research [2, 22, 24]. For this analysis, we selected the dataset of Ferenc *et al.*, because its model performs much better than the dataset of Shippey *et al.*, and unlike the dataset of Mo *et al.*, it does not contain any unrealistic future information.

We want to investigate cases where we see substantially different accuracy when two different training project sets are used, although other settings are identical. In that vein, we found that the accuracy in bug prediction for the *Android-Universal-Image-Loader* project is significantly higher with training projects *titan, mct, oryx, hazelcast, and MapDB* than with projects *elasticsearch, JUnit, ceylon-ide-eclipse, antlr4, and mcMMO*, although in both cases the RandomForest algorithm was used. Figure 12 (a) compares the code metric distributions (HCPL) between the test project *Android-Universal-Image-Loader* and the higher accuracy-producing training projects (*titan, mct, oryx, hazelcast, and MapDB*). Figure 12 (b) does the same, except it compares the test project with the worst accuracy-producing training projects. Clearly, the HCPL code metric distribution of the test project is much more similar to the training projects with higher accuracy (Figure a) than the training projects with lower accuracy (Figure b). We also observed a similar case when the *mct* project was the test project—two different training sets produced two subsantially different accuracies for *mct*. Figure 12 (c) and Figure 12 (d) draw the same conclusion—training projects with similar HCPL distribution to the test project produce better accuracy. Encouragingly, this conclusion does not change for the other two code metrics: LOC, and McCabe. This implies that similarity in the distribution of code metrics can be used as a proxy to select similar training projects for improving future MLBP models. We plan to evaluate this more rigorously in the future.

We also observed that all the previous research considered method-level bug prediction as a static problem; a method was either considered buggy or not buggy for its whole lifetime. These models are impractical because they can not adapt to the change of state after a method undergoes one or more bug-fixing processes. Future research, therefore, should consider method-level bug prediction as a time-series problem.

7 THREATS TO VALIDITY

Several threats can harm the validity of our findings.



(a) Android-Universal-Image-Loader (best set of training projects)

(b) Android-Universal-Image-Loader (worst set of training projects)

(c) mct (best set of training projects)

(d) mct (worst set of training projects)

Fig. 12. Comparison between the HCPL metric distributions between the test and the training projects. The first row shows the results for the Android-Universal-Image-Loader project with the best (a) and the worst (b) training projects. The second row shows the same for the mct project. The distribution of code quality indicator HCPL for the test project is more similar to the higher accuracy-producing training projects (Figures (a) and (c)) than the lower accuracy-producing training projects (Figures (b) and (d)). This observation remains the same if we replace HCPL with other quality indicators (LOC, and McCabe).

External validity is impacted by the selection of the three datasets. To the best of our knowledge, these are, unfortunately, the only publicly available datasets designed for method-level bug prediction. We have contacted the first author of [83] asking about the availability of their dataset. Their suggestion was to follow their posted data collection process which may unfortunately require a few months. Our new dataset used 49 open-source Java projects. As such, our results might not generalize for closed-source projects, or for projects written in different programming languages.

Internal validity is hampered by our choice of the two statistical tests: Wilcoxon rank sum, and Cliff's delta. These two tests, however, are widely adopted in software engineering research (e.g., [8, 20, 21, 40, 60, 98]). In addition, we have also analyzed our results with data visualization—e.g., the cumulative distribution function and box-plots.

Construct validity is affected by our selection of bug- and fix-related keywords. Also, for tracing a method's change history, we relied on CodeShovel [40]. CodeShovel uses string similarity to decide if two given methods are similar. This approach can be inaccurate in method overloading, which can be common in Java-based projects. In RQ3, to understand the decay in bug-proneness, we have captured the age of a method during its bug fix. A bug, however, can be much older than its bug-fix time, but exactly when a method became buggy is difficult to capture. However, if we see that bug-fix happens less as a method gets older, it is not an unreasonable assumption that bug-proneness decays over time too.

Conclusion validity of our findings can be impacted by any of the above mentioned threats.

8 CONCLUSION

In this paper, we have established that existing method-level bug prediction models are not suitable for realistic scenarios applicable to industry practices. We have shown three time-sensitive realistic scenarios that future models should be evaluated with. We then discussed four potential research avenues that may improve method-level bug prediction significantly.

Through our findings, the extremely poor performance of the existing models became unsurprising, given that they were trained on noisy datasets. An accurate bug labeling approach, such as the one we have presented, should be used in the future. In addition, the bug-proneness of a method decays over time, due to concept drift. This observation was neglected in the earlier studies, which also partly explains their poor performance. In those models, a complex bug-prone method would always be considered buggy, even if the method has undergone several bug-fixing processes. We have also shown that method-level bug prediction accuracy can be improved by selecting similar training projects and building separate models based on method sizes.

We hope that our findings and guidelines would excite and encourage the research community for producing ever more accurate method-level bug prediction models that are also suitable for industry practice.

9 ACKNOWLEDGMENTS

This research was partly supported by the NSERC Alliance (ALLRP/556396-2020) - Alberta Innovates CASBE Program (Grant #202102242), and Eyes High Postdoctoral Match-Funding Program, while Dr. Chowdhury and Dr Hemmati were at the University of Calgary.

REFERENCES

[1] Syed Ishtiaque Ahmad. 2021. *Investigating the impact of methodological choices on source code maintenance analyses.* Master's thesis. University of British Columbia.

[2] M. Alfadel, A. Kobilica, and J. Hassine. 2017. Evaluation of Halstead and Cyclomatic Complexity Metrics in Measuring Defect Density. In *2017 9th IEEE-GCC Conference and Exhibition*. 1–9.

[3] H. Alsolai, M. Roper, and D. Nassar. 2018. Predicting Software Maintainability in Object-Oriented Systems Using Ensemble Techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution*. 716–721.

[4] T. L. Alves, C. Ypma, and J. Visser. 2010. Deriving metric thresholds from benchmark data. In *IEEE International Conference on Software Maintenance*. 1–10.

[5] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. 2015. Automatic Metric Thresholds Derivation for Code Smell Detection. In *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. 44–53.

[6] Takuya Asano, Masateru Tsunoda, Koji Toda, Amjed Tahir, Kwabena Ebo Bennin, Keitaro Nakasai, Akito Monden, and Kenichi Matsumoto. 2021. Using Bandit Algorithms for Project Selection in Cross-Project Defect Prediction. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 649–653.

[7] Abdullateef O Balogun, Babajide J Odejide, Amos O Bajeh, Zubair O Alanamu, Fatima E Usman-Hamza, Hammid O Adeleke, Modinat A Mabayoje, and Shakirat R Yusuff. 2022. Empirical Analysis of Data Sampling-Based Ensemble Methods in Software Defect Prediction. In *International Conference on Computational Science and Its Applications*. 363–379.

[8] Abdul Ali Bangash, Hareem Sahar, Abram Hindle, and Karim Ali. 2020. On the Time-Based Conclusion Stability of Cross-Project Defect Prediction Models. *Empirical Softw. Engg.* 25, 6 (2020).

[9] V.R. Basili, L.C. Briand, and W.L. Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* 22, 10 (1996), 751–761.

[10] Gabriele Bavota, Mario Linares-Vásquez, Carlos Eduardo Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2015. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering* 41, 4 (2015), 384–407.

[11] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. 2011. Does Measuring Code Change Improve Fault Prediction?. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering* (Banff, Alberta, Canada) *(Promise '11)*. Article 2, 8 pages.

[12] Kwabena E Bennin, Nauman bin Ali, Jürgen Börstler, and Xiao Yu. 2020. Revisiting the Impact of Concept Drift on Just-in-Time Quality Assurance. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. 53–59.

[13] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and Balanced? Bias in Bug-Fix Datasets. 121–130.

[14] Raymond P. L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *IEEE Trans. Softw. Eng.* 36, 4 (July 2010), 546–558.

[15] Celerity. [n.d.]. The True Cost of a Software Bug: Part One. https://www.celerity.com/insights/the-true-cost-of-a-software-bug. [Online; last accessed 01-Sep-2022].

[16] Ned Chapin. 2000. Do we know what preventive maintenance is?. In *International Conference on Software Maintenance*. 15–17.

[17] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering* 26, 6 (2021), 1–44.

[18] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1580–1596.

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

[20] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2019. GreenScaler: training software energy models with automatic test generation. *Empirical software engineering : an international journal* 24, 4 (2019), 1649–1692.

[21] Shaiful Chowdhury, Silvia Di Nardo, Abram Hindle, and Zhen Ming Jack Jiang. 2018. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering* 23, 3 (2018), 1422–1456.

[22] Shaiful Chowdhury, Reid Holmes, Andy Zaidman, and Rick Kazman. 2022. Revisiting the Debate: Are Code Metrics Useful for Measuring Maintenance Effort? *Empirical Software Engineering (EMSE)* 27, 6 (2022), 31 pages.

[23] Shaiful Chowdhury, Gias Uddin, and Reid Holmes. 2022. An Empirical Study on Maintainable Method Size in Java. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 252–264.

[24] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. 1979. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Transactions on Software Engineering* SE-5, 2 (1979), 96–104.

[25] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *2010 7th IEEE working conference on mining software repositories (MSR 2010)*. IEEE, 31–41.

[26] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process* 26, 1 (2014), 107–139.

[27] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. 2018. A Developer Centered Bug Prediction Model. *IEEE Transactions on Software Engineering* 44, 1 (2018), 5–24.

[28] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 341–350.

[29] Jayalath Ekanayake, Jonas Tappolet, Harald C Gall, and Abraham Bernstein. 2009. Tracking concept drift of software projects using defect prediction quality. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. 51–60.

[30] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 27, 7 (2001), 630–650.

[31] Farzaneh S Fard, Paul Hollensen, Stuart Mcilory, and Thomas Trappenberg. 2017. Impact of biased mislabeling on learning with deep networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*. 2652–2657.

[32] Norman E Fenton and Martin Neil. 1999. A critique of software defect prediction models. *IEEE Transactions on software engineering* 25, 5 (1999), 675–689.

[33] Rudolf Ferenc, Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. 2020. An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software* 169 (2020).

[34] Christine Fisher. [n.d.]. Boeing found another software bug on the 737 Max. https://www.engadget.com/2020-02-06-boeing-737-max-software-bug.html. [Online; last accessed 01-Sep-2022].

[35] 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*. 171–180.

[36] Yossi Gil and Gal Lalouche. 2016. When do Software Complexity Metrics Mean Nothing? – When Examined out of Context. *Journal of Object Technology* 15, 1 (Feb. 2016), 2:1–25.

[37] Yossi Gil and Gal Lalouche. 2017. On the Correlation between Size and Metric Validity. *Empirical Software Engineering* 22, 5 (Oct. 2017), 2585–2611.

[38] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. 2000. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering* 26, 7 (2000), 653–661.

[39] Felix Grund, Shaiful Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: A Reusable and Available Tool for Extracting Source Code Histories. In *Proceedings of the International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 221–222.

[40] Felix Grund, Shaiful Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1510–1522.

[41] Shivani Gupta and Atul Gupta. 2019. Dealing with Noise Problem in Machine Learning Data-sets: A Systematic Review. *Procedia Computer Science* 161 (2019), 466–474. The Fifth Information Systems International Conference, 23-24 July 2019, Surabaya, Indonesia.

[42] T. Gyimothy, R. Ferenc, and I. Siket. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering* 31, 10 (2005), 897–910.

[43] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2011. Historage: Fine-grained Version Control System for Java. In *Proc. International Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution*. 96–100.

[44] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug Prediction Based on Fine-Grained Module Histories. 200–210.

[45] Mark Hays and Jane Hayes. 2012. The Effect of Testability on Fault Proneness: A Case Study of the Apache HTTP Server. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*. 153–158.

[46] 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.

[47] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A Practical Model for Measuring Maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*. 30–39.

[48] Steffen Herbold, Alexander Trautsch, and Jens Grabowski. 2017. Global vs. local models for cross-project defect prediction. *Empirical software engineering* 22, 4 (2017), 1866–1902.

[49] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016), 303–336.

[50] K. Herzig and A. Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories*. 121–130.

[51] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. 2020. On tracking Java methods with Git mechanisms. *Journal of Systems and Software* 165 (2020), 110571.

[52] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2017. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering* 45, 2 (2017), 111–147.

[53] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*. Vol. 112.

[54] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, and John Grundy. 2021. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 432–443.

[55] Md Alamgir Kabir, Jacky W Keung, Kwabena E Bennin, and Miao Zhang. 2019. Assessing the significant impact of concept drift in software defect prediction. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 53–58.

[56] D. Kafura and G. R. Reddy. 1987. The Use of Software Complexity Metrics in Software Maintenance. *IEEE Transactions on Software Engineering* SE-13, 3 (1987), 335–343.

[57] Yasutaka Kamei and Emad Shihab. 2016. Defect Prediction: Accomplishments and Future Challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 33–45.

[58] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 81–90.

[59] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! are you committing tangled changes?. In *Proceedings of the 22nd International Conference on Program Comprehension*. 262–265.

[60] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust statistical methods for empirical software engineering. *Empirical Software Engineering* 22, 2 (2017), 579–630.

[61] D. Landman, A. Serebrenik, and J. Vinju. 2014. Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods. In *IEEE International Conference on Software Maintenance and Evolution*. 221–230.

[62] Michele Lanza, Andrea Mocci, and Luca Ponzanelli. 2016. The Tragedy of Defect Prediction, Prince of Empirical Software Engineering Research. *IEEE Software* 33, 6 (2016), 102–105.

[63] Issam H Laradji, Mohammad Alshayeb, and Lahouari Ghouti. 2015. Software defect prediction using ensemble learning on selected features. *Information and Software Technology* 58 (2015), 388–402.

[64] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[65] C. Lewis and R. Ou. [n.d.]. Bug prediction at Google. http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html. [Online; last accessed 01-Sep-2022].

[66] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. 2018. Progress on approaches to software defect prediction. *Iet Software* 12, 3 (2018), 161–175.

[67] Xiaoyu Liu, LiGuo Huang, Chuanyi Li, and Vincent Ng. 2018. Linking Source Code to Untangled Change Intents. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 393–403.

[68] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. 2012. Transfer learning for cross-company software defect prediction. *Information and Software Technology* 54, 3 (2012), 248–256.

[69] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[70] T. J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320.

[71] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010).

[72] T. Menzies, J. Greenwald, and A. Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 1 (2007), 2–13.

[73] Microsoft. 2022. Code Metrics Maintainability Index. https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022. [Online; last accessed 06-Jan-2022].

[74] Ran Mo, Shaozhi Wei, Qiong Feng, and Zengyang Li. 2022. An Exploratory Study of Bug Prediction at the Method Level. *Inf. Softw. Technol.* 144, C (apr 2022).

[75] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. Investigating the Relationship between Evolutionary Coupling and Software Bug-Proneness. 173–182.

[76] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. Analysis of the Reliability of a Subset of Change Metrics for Defect Prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Kaiserslautern, Germany) *(ESEM '08)*. 309–311.

[77] N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering*. 284–292.

[78] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining Metrics to Predict Component Failures. 452–461.

[79] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[80] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance*. 1–10.

[81] P. Oman and J. Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*. 337–344.

[82] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An Exploratory Study on the Relationship between Changes and Refactoring. In *Proceedings of the 25th International Conference on Program Comprehension* (Buenos Aires, Argentina). 176–185.

[83] 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).

[84] Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. [n.d.]. Testing of mobile applications in the wild: A large-scale empirical study on android apps. In *Proceedings of the 28th international conference on program comprehension*. 296–307.

[85] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. Software fault prediction metrics: A systematic literature review. *Information and Software Technology* 55, 8 (2013), 1397 – 1418.

[86] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. BugCache for Inspections: Hit or Miss? 322–331.

[87] Md Saidur Rahman and Chanchal K. Roy. 2017. On the Relationships Between Stability and Bug-Proneness of Code Clones: An Empirical Study. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 131–140.

[88] Paul Ralph and Ewan Tempero. 2018. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand). 13–23.

[89] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. 428–439.

[90] Nornadiah Mohd Razali, Yap Bee Wah, et al. 2011. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics* 2, 1 (2011), 21–33.

[91] Umaa Rebbapragada and Carla E Brodley. 2007. Class noise mitigation through instance weighting. In *European conference on machine learning*. 708–715.

[92] Payam Refaeilzadeh, Lei Tang, and Huan Liu. 2009. Cross-validation. *Encyclopedia of database systems* 5 (2009), 532–538.

[93] D. Romano and M. Pinzger. 2011. Using source code metrics to predict change-prone Java interfaces. In *2011 27th IEEE International Conference on Software Maintenance*. 303–312.

[94] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-Informed Oracle. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain). 436–447.

[95] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. 2016. Improving code readability models with textual features. In *IEEE 24th International Conference on Program Comprehension*. 1–10.

[96] Matteson Scott. [n.d.]. Report: Software failure caused $1.7 trillion in financial losses in 2017. https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/. [Online; last accessed 01-Sep-2022].

[97] Francisco Servant and James A. Jones. 2017. Fuzzy Fine-Grained Code-History Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 746–757.

[98] Martin Shepperd, Michelle Cartwright, and Gada Kadoda. 2000. On building prediction systems for software engineers. *Empirical Software Engineering* 5, 3 (2000), 175–182.

[99] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An Industrial Study on the Risk of Software Changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina).

[100] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787.

[101] Thomas Shippey, Tracy Hall, Steve Counsell, and David Bowes. 2016. So You Need More Method Level Datasets for Your Software Defect Prediction? Voilà! *(ESEM '16)*.

[102] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. 2012. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering* 39, 4 (2012), 552–569.

[103] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[104] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. 2011. A General Software Defect-Proneness Prediction Framework. *IEEE Transactions on Software Engineering* 37, 3 (2011), 356–370.

[105] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution*. 1–12.

[106] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2014. Incremental Origin Analysis of Source Code Files. In *Proceedings Working Conference on Mining Software Repositories (MSR)*. 42—-51.

[107] Zhongbin Sun, Junqi Li, Heli Sun, and Liang He. 2021. CFPS: Collaborative filtering based source projects selection for cross-project defect prediction. *Applied Soft Computing* 99 (2021), 106940.

[108] Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. 2012. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 6 (2012), 1806–1817.

[109] Chakkrit Tantithamthavorn and Ahmed E. Hassan. 2018. An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden). 286–295.

[110] Umesh Tiwari and Santosh Kumar. 2014. Cyclomatic Complexity Metric for Component Based Software. *SIGSOFT Softw. Eng. Notes* 39, 1 (Feb. 2014), 1–6.

[111] VerifySoft. 2022. VerifySoft Maintainability Index. https://verifysoft.com/en_maintainability.html. [Online; last accessed 06-Jan-2022].

[112] Zhiyuan Wan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2020. Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Transactions on Software Engineering* 46, 11 (2020), 1241–1266.

[113] Song Wang, Junjie Wang, Jaechang Nam, and Nachiappan Nagappan. 2021. Continuous Software Bug Prediction. Article 14, 12 pages.

[114] Tiejian Wang, Zhiwu Zhang, Xiaoyuan Jing, and Liqiang Zhang. 2016. Multiple kernel ensemble learning for software defect prediction. *Automated Software Engineering* 23, 4 (2016), 569–590.

[115] Zixu Wang, Weiyuan Tong, Peng Li, Guixin Ye, Hao Chen, Xiaoqing Gong, and Zhanyong Tang. 2022. BugPre: an intelligent software version-to-version bug prediction system using graph convolutional neural networks. *Complex & Intelligent Systems* (2022), 1–21.

[116] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto. 2022. Predicting Defective Lines Using a Model-Agnostic Technique. *IEEE Transactions on Software Engineering* 48, 05 (may 2022), 1480–1496.

[117] Ian H Witten, Eibe Frank, Mark A Hall, Christopher J Pal, and MINING DATA. 2005. Practical machine learning tools and techniques. In *Data Mining*, Vol. 2.

[118] Sebastien C. Wong, Adam Gatt, Victor Stamatescu, and Mark D. McDonnell. 2016. Understanding Data Augmentation for Classification: When to Warp?. In *2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*. 1–6.

[119] Mahama Yahaya, Wenbo Fan, Chuanyun Fu, Xiang Li, Yue Su, and Xinguo Jiang. 2020. A machine-learning method for improving crash injury severity analysis: a case study of work zone crashes in Cairo, Egypt. *International journal of injury control and safety promotion* 27, 3 (2020), 266–275.

[120] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206–220.

[121] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan. 2013. How Does Context Affect the Distribution of Software Maintainability Metrics?. In *IEEE International Conference on Software Maintenance*. 350–359.

[122] Yuming Zhou, Baowen Xu, and Hareton Leung. 2010. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software* 83, 4 (2010), 660 – 674.

[123] Xingquan Zhu, Xindong Wu, and Qijun Chen. 2003. Eliminating class noise in large datasets. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. 920–927.

[124] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. 7 pages.