Impact of Methodological Choices on the Analysis of Code Metrics and Maintenance

Syed Ishtiaque Ahmad[a], Shaiful Chowdhury[b] and Reid Holmes[a]

[a]Department of Computer Science, University of British Columbia, Vancouver, Canada
[b]Department of Computer Science, University of Manitoba, Winnipeg, Canada

ARTICLE INFO

Keywords:
Software evolution
Change-proneness
Bug-proneness
Empirical studies
Software data science
Code metrics

ABSTRACT

Many statistical analyses and prediction models rely on past data about how a system evolves to learn and anticipate the number of changes and bugs it will have in the future. As a software engineer or data scientist creates these models, they need to make several methodological choices such as deciding on size measurements, whether size should be controlled, from what time range metrics should be obtained, etc. In this work, we demonstrate how different methodological decisions can cause practitioners to reach conclusions that are significantly and meaningfully different. For example, when measuring SLOC from evolving source code of a method, one could decide to use the initial, median, average, final, or a per-change measure of method size. These decisions matter; for instance, some prior studies observed better performance of code metrics for bug prediction in general, while other studies found negative results when performance was evaluated through a time-based approach. Understanding the impact of these different methodological decisions is especially important given the increasing significance of approaches that use these large datasets for software analysis tasks. This paper can impact both practitioners and researchers by helping them understand which of the methodological choices underpinning their analyses are important, and which are not; this can lead to more consistency among research studies and improved decision-making for deployed analyses.

1. Introduction

Software maintenance is a challenging [1] and costly task [2] and researchers are investigating ways to reduce this cost by understanding the correlations between different indicators and maintenance cost and by building predictive models [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. These analyses and models generally focus on identifying the highly maintenance-prone components such as change- [13, 14, 15] and bug-prone [16, 17, 18] source code components. Researchers have investigated static code metrics (e.g., McCabe [19], C&K [20]), process metrics (e.g., code churn [21], change burstiness [22], number of developers [23]), and combinations of metrics [24, 25] as these are often associated with bug and change proneness. Unfortunately, there have been disagreements among research communities related to the true effectiveness of these metrics [26, 27, 28].

While some studies argue that code metrics are better for predicting pre-release bugs [29, 30], others claim metrics perform well for post-release [31] as well. Different studies have also had disagreements at different granularities (e.g., module-level [29, 32, 33, 34] vs. method-level [35]), or using different aggregation schemes (e.g., mean vs. entropy [36]). What are the causes of these disagreements? Are observations about code metrics and maintenance impacted by different ways a metric can be measured (e.g., SLOC measurement) or by other factors such as the time frame from which metrics are obtained or by the way data is aggregated across projects?

siahmad@cs.ubc.ca (S.I. Ahmad); shaiful.chowdhury@umanitoba.ca (S. Chowdhury); rtholmes@cs.ubc.ca (R. Holmes)
ORCID(s): 0000-0003-4213-494X (R. Holmes)

Creating useful, high-quality predictive models requires strong methodological rigor to avoid inconsistent or misleading predictions. In this work, we explore several of the key methodological decisions made by researchers and practitioners as they design their data analysis pipelines for software evolution analyses. Specifically, based on contradictions reported by prior work (e.g., [37, 38, 18, 27, 39, 26]), we examine the impact of five common methodological choices on the relationship between code metrics and maintenance. These five choices must be made when designing predictive maintenance models. These decisions relate to size measures, size normalization, timeframe selection, change analysis, and result aggregation schemes which impact change and bug proneness prediction models. Our analysis reveals that these choices can lead to substantially different results, and explain some of the previous contradictory findings from prior software maintenance prediction efforts.

We performed an analysis of these methodological decisions at the method level granularity. This granularity was chosen because method level granularity is commonly desired by the research community [35, 40] and industrial developers [41]. To do this, we extracted the complete method level history of 1,598,592 Java methods from 53 open-source projects. To the best of our knowledge, our analysis of these large number of projects and methods itself is very unique when compared to other similar prior studies (e.g., [35, 42, 40, 38, 43]). With this significantly large dataset, we then examined the impact of the most commonly used variations of five different methodological choices on the relationship between code metrics and maintenance.

The main contribution of this paper is a demonstration of the impact that different methodological decisions,

commonly used in data collection and analysis of software evolution data, can have on the concrete findings that would be derived from these data. We show how these decisions explain many contradictions present in prior work so future analyses and deployments of these models can be better designed and interpreted. Our observations are particularly important for researchers and practitioners, leveraging the vast trove of historical data available from past system evolution, so they can conduct statistical analysis and build their models using high-quality data to make accurate and meaningful conclusions and predictions. To aid in a more accurate analysis of code metrics and maintenance, we provide a set of recommendations that future studies can follow.

To enable replication and extension, we share our dataset publicly.[1]

1.1. Paper Organization

In Section 2, we begin by explaining the rationale behind choosing size as the representative metric for code metrics and the decision to conduct our analysis at the method-level code granularity. Following this, we outline the motivations for the five research questions we selected, which correspond to the five methodological choices we made. Section 3 describes the methodology of this paper. Results and analysis of the five research questions are presented in Section 4. Section 5 presents the recommendations we make along with the threats that can impact our results. We discuss the related works in Section 6. Section 7 concludes this paper.

2. Background and Research Questions

Given the expense associated with software maintenance, engineers and researchers have long sought to predict future software evolution trends in a bid to reduce the number of future bugs and decrease future maintenance costs. These predictions typically use historical data, either about the system under analysis or other prior systems, to identify code units or changes that could be prone to future changes. Future changes are typically used as a proxy for both the expensive tasks of fixing bugs and evolving existing source code. Two of the most common kinds of software evolution predictions are change-proneness and bug-proneness; these models share similar methodological steps, as would any analysis that takes past historical project data as input.

Given a desired analysis unit granularity (usually a subsystem, file, or method) from one or more projects, all practitioners wishing to perform change- or bug-proneness predictions need to make some common methodological choices to build their analysis pipelines. Based on our observations from prior studies (e.g., [42, 40, 44, 45, 46, 26, 47, 48, 49, 50, 51, 52]), we identified the five most common choices where the variation of these choices may lead to contradictory findings. In this work, we investigate each of these five choices. In our analysis procedure, we fix our granularity unit to the method level, for a total of 1,598,592 methods from the historical evolution of 53 projects, and use SLOC (source lines of code without comments and blank lines, also known as size) as the selected code metric as it is widely used in both industrial and research settings.

2.1. Code Metrics for Evolution

Previous work examined several code metrics to predict bug and change proneness. However, the researchers could not establish a common ground on the usefulness of code metrics [26, 27, 28, 44]. Some studies found favourable outcomes for code metrics [38, 37, 53, 54, 18, 55], while others discovered negative results [35, 27, 39]. Some argue that process metrics [56, 57] perform better than static code metrics [58, 31] for fault prediction in highly iterative post-release software systems, while others found that the combination of process and static code metrics perform well in terms of accuracy [49]. Despite the continuous debate about code metrics, previous studies unanimously agree that size is the effective code metric to estimate software maintenance [26, 59, 60, 61]. *Therefore, we use size (or SLOC) as the selected code metric to understand how different methodological choices can impact the analysis of code metrics and maintenance*.

2.2. Code Granularity

The relationship between code metrics and software maintenance can be analyzed at different levels of source code granularity: module/component level [29, 50], class/file level [26, 62, 59, 63, 52], method level [42, 45, 61, 40], and line level [64]. Unfortunately, practitioners find it difficult to work with coarse granularity levels, such as module and class levels [42, 41, 65]. For example, it is difficult to find a bug from a class, as a class can often be more than 500 or 1000 (sometimes even more than 5000) lines of code [66]. Also, research has found that only a fraction of methods in a class are actually bug-prone [45]. Line-level maintenance analyses, on the other hand, are often difficult and infeasible. For example, in this study, we will need to collect the change history of different code components, which is easy at the file/class or method level, but difficult at the line level. Line level change history suffers from many false positives and false negatives because many lines are similar just by chance [41, 67, 68].

Considering these factors, method-level maintenance analysis has gained traction in the recent past, and rightly so [69, 41, 61, 42, 45, 40, 44]. *Following the recent studies, we also focus on size at the method-level granularity to understand its varied impact on maintenance based on the methodological choices a researcher or a practitioner can make.*

2.3. The Five Methodological Choices

We now discuss the five methodological choices that may impact the analysis of code metrics and maintenance. As we describe next, we found these methodological choices emerged frequently in software maintenance studies.

[1] https://zenodo.org/records/12905296

2.3.1. (RQ1) Size measurement

Let us consider a scenario where a researcher aims to find what types of source code methods are maintenance-prone. More specifically, the researcher is interested in finding if there are patterns in code metrics distributions that make a source code method more or less maintenance-prone. The researcher, for example, can group highly change- and bug-prone methods in one bucket, and the less change- and bug-prone methods into another bucket. Distributions of different code metrics between these two buckets can then be compared. If significantly different distributions are observed, practitioners can be advised about the patterns of maintainable source code methods (e.g., about the maintainable method size [61]). The question is, what code metric value of a method should be considered while performing this analysis? This is important because, with the evolution of a source method, its code metrics' values can change (sometimes very significantly) [70, 44].

In our dataset, there is a method `processFiltered` from the `checkstyle`[2] project which has been revised 73 times. This method was also involved with bug-fix commits seven times. This is definitely a highly maintenance-prone method. The initial size of this method was 31 (a medium-sized method [61]). After much evolution, the size became 108 (a very large-sized very high-risk method [61]), and after refactoring and optimization, the size finally became only 22 (a small-sized low-risk method [61]). Now, to learn the code metrics patterns of these types of methods, which code metric value (i.e., size value) should represent this method? The first value (31), the last value (22), the largest value (108), the mean (55), the median (44), the mode (39), or the version-specific value (described later) proposed by Chowdhury et al [44, 61]? We provide details about calculating the version-specific values in Section 3.5.3.

We found that many prior relevant studies had to make this decision (which metric value to use) while analyzing maintenance or developing maintenance prediction models (e.g., [71, 45, 44, 72, 46, 15]). For example, to calculate the code metrics of classes and interfaces, Romano et al. [71] considered the current code version from the versioning repositories. This means, the authors considered the last code metric value of an evolved class or interface (or the first value for the class or interface that was not evolved after their introduction).

While all these measurements can be rationally selected by individual studies, can these variations in code metric measurement approaches explain the contradictory findings in earlier studies about the relationship between code metrics and maintenance (RQ1)?

2.3.2. (RQ2) Size normalization for bug and change analysis

The research community almost unanimously agrees that a larger code component is more likely to have more code smells, and thus, is more maintenance prone [42, 69, 61, 46, 26, 73]. Chowdhury et al. [69, 61] found that larger Java methods go through more revisions and more bug fixes than smaller methods. Khomh et al. [46] claimed that size and structural changes are connected to anti-patterns of classes, and classes containing anti-patterns are more maintenance-prone. Gil et al. [26] found that size is the most influential of all code metrics while explaining software maintenance efforts. According to Gil et al., size is the only useful code metric to understand software maintenance, because all other code metrics become useless when their correlation with size is normalized. This observation was also common in many other prior studies [59, 60, 74]. The conclusion of these studies is simple: the larger the size, the more change- and bug-prone a code component is. This implies that practitioners should spend more time on large code components for maintenance-related tasks.

In contrast to the previously mentioned studies, a very different conclusion was made by Olbrich et al. [75]. The authors found that large classes (e.g., God and Brain classes) are less maintenance-prone if change frequency and bug rate are calculated with normalized SLOC (i.e., per line of code). This, to some extent surprising, results were also found in the study of Yamashita et al. [76].

Considering these somewhat contradictory findings, should practitioners focus on larger code components while focusing on change- and bug-proneness, and smaller code components while focusing on change and bug-density? *In other words, is size correlated positively with total change and bugs, but negatively with change and bug per line of code? This is what we want to confirm with our large and robust dataset at the method-level source code granularity (RQ2).*

2.3.3. (RQ3) Age normalization and timeframe selection

A common potential problem with earlier code metrics-related work was ignoring the impact of the age of a code component (module, class, method, etc.) while analyzing its maintainability. Generally, researchers take a particular project snapshot to retrieve the source code and then to measure different code metrics of different code components [75]. This can create both internal and external threats to validity. For example, although a research study, in general, would use one particular snapshot of a particular project, the code components in that project can still be of different ages: some were just introduced to the system, while others have evolved for a long time [41]. Similarly, even when the same set of software projects are used in different research studies, they do not necessarily use the same projects' snapshots. For example, the `eclipse` project was used by the code smell study of Khomh et al. in 2012 [46] and Palomba et al. in 2018 [73]—same project, but very different project snapshots.

The age differences across different code components can be problematic: from prior work, we know that different chunks of data within one project could lead to different prediction models [77], making just-in-time prediction models popular in recent times. A natural question is if two

[2] https://github.com/checkstyle/checkstyle

code components have very similar code metric distribution but very different ages, do they still have similar change- and bug-proneness? What if, despite the similarly high code complexity, the older code component has become less maintenance-prone over time due to the different corrective and perfective changes it has already gone through? If the performance of code metrics can change significantly as a particular code component ages [78, 69], can age be one of the factors contributing to the contradictory findings about code metrics in earlier studies [44, 26, 45, 42]?

We, therefore, investigate if age should be normalized while analyzing code metrics' effectiveness to understand maintenance. If so, what timeframe explains the relationship between code metrics and maintenance the most (RQ3)?

2.3.4. (RQ4) Change analysis

Prior studies vary in how they consider the change-proneness of a code component. A common approach is to consider the number of revisions of a class/file or method as its change proneness [46, 55, 47, 44, 14]. As a result, software quality reporting tools depend on the number of revisions or commit frequency as a measure of change hotspot [79, 80, 81]. However, commit patterns are dependent on developers' coding styles, expertise, and experience [82, 83]. This observation led some other studies to use different change size metrics as the change-proneness indicators including diff size [84, 85], new additions [85], and edit distance [86, 39, 84].

Kawrykow and Robillard [87] analyzed the type of transformations applied to methods and found that a large fraction (15.5%) of method changes can be classified as non-essential. They reported that most changes involve removing or adding *this* keyword or are induced by rename refactoring. Additionally, Ray et al. [88] observed that most changes introduced by developers in multiple revisions are non-unique (i.e., follow a specific repetitive pattern). Therefore, if we do not account for the type of changes applied to a method, we can produce inconsistent findings for code metric analysis.

In this work, we adopt the differentiation technique of essential and non-essential changes by Kawrykow and Robillard [87] to demonstrate the impact of different types of changes at the method level. *Intuitively, some changes will have a greater evolutionary impact than others. For example, changing a code comment is less likely to directly introduce a future bug than modifying a key algorithm. We therefore ask, can the sizes and types of the changes made to a code unit produce contradictory results while analyzing code metrics and maintenance (RQ4)?*

2.3.5. (RQ5) Data aggregation

To analyze code metrics and maintenance, researchers often rely on mining software repositories (MSR). Generally, a researcher selects a set of software project repositories to mine data about changes and bugs and to extract source code metrics. With this approach, a researcher has to select one of the two approaches: aggregated analysis, and individual project analysis.

For example, to observe the relationship between size and bug-proneness, a researcher can aggregate data from all the projects [26, 42, 18, 73], and apply a statistical test to produce one single value, such as Kendall's τ correlation coefficient or bug-prediction accuracy. Project aggregation can hinder accurate analysis: external factors, such as code review policy, developers' commit habits, etc., can impact different projects differently. For example, Gil et al. [28] confirmed that code metrics distributions are often significantly different across software projects, which are difficult to normalize even after applying different transportation approaches [44]. Also, with project aggregation, the outcome of any analysis can be significantly impacted by a few large outlier projects.

An obvious alternative to aggregated analysis is to apply individual project analysis [85, 89, 90, 71, 46], where the outcome is presented for each project separately. This can confirm if a particular observation (e.g., a high correlation coefficient between cyclomatic complexity and bug-proneness) is common across different projects. This approach, however, can suffer from selection/publication bias—one can select a particular set of projects (while ignoring other projects) that support their publication need [91]. *We ask, can the inaccurate application of aggregated or individual project analysis be one of the reasons for previous contradictory findings in code metrics and maintenance analysis research (RQ5)?*

3. Methodology

To accurately examine the impact of design alternatives for the five methodological decision points, we collected a corpora of method-level historical data to facilitate a deeper analysis of how those methods evolved and what change-proneness and bug-proneness approaches would have predicted for them.

3.1. Project Selection

Over the last decade, GitHub has gained popularity as a rich source of open-source projects. In our work, we used the SEART GitHub Search Tool [92] to identify candidate systems for analysis. Our search criteria included Java projects (excluding forks), that have $\geq$ 1,000 commits, $\geq$ 200 stars, and $\geq$ 30 contributors. Generally, the number of stars is considered a proxy for popularity [93]. We only selected Java projects as the number of bugs and metrics values varies according to the programming language used [94, 95].

We then sorted the resulting system by repository file size and manually filtered out systems less than 2,000 kilobytes to try to remove toy projects from the 1,204 Java projects. Finally, we applied purposive sampling [96] to select a wide range of actively maintained projects. We considered projects as *actively maintained* that release at least two updates per year and merge a minimum of three pull requests per month, demonstrating continuous integration of new code contributions. Additionally, we require active community engagement, including responding to issues within six months of their creation and participating in discussions. To

verify this, we reviewed some latest open pull requests for each of the 53 projects, ensuring that the response time from the creation of the pull request was within six months. This approach allows us to select projects that are both up-to-date and well-maintained. These are the choices we made after having a discussion among the authors.

We discarded unpopular projects (e.g., projects with fewer stars and contributors) because we wanted to avoid projects that were personal toy projects, as they don't represent the complexity of real-world projects. Unpopular projects may not go through proper maintenance and thus their change-proneness and bug-proneness would automatically be low. These projects can contain bugs and still may seem harmless due to no bug-fixing activities. We believe that our project selection approach mitigated the perils of using GitHub repositories for software engineering research [97].

The final set of 53 projects, presented in Table 1, had a median age of 11 years and a median of 35, 207 changes. We believe this diverse set of projects had enough change history to have undergone sufficient historical changes to observe temporal effects. Furthermore, the selected projects had a large number of contributors (median 157) to capture different developers' coding styles, although for projects with strict coding conventions this may be less of a concern.

The resulting 53 projects is a larger sample size than related method-level studies, which have used 13 [35], 21 [40] and 20 [41] projects. Additionally, subsets of our selected projects have also been used in previous studies [18, 98, 99]. The selected projects serve a diverse set of domains, including code analysis (checkstyle), mocking libraries (mockito), HTTP clients (okhttp), and JSON parsing (fastjson, gson).

3.2. Data Collection

We used *CodeShovel* [41] to extract the complete set of change commits for a given source code method. CodeShovel does this by walking backward through a project's commit history starting from the reference commit listed in Table 1. It can detect cross-file changes (file move and file rename), method signature changes (method rename, return type change, exception changes, parameter changes, parameter type changes), metadata changes (documentation and annotation changes), and method body changes. CodeShovel has been shown to accurately uncover 97% of all method changes. CodeShovel also collects meta-information about each commit such as commit message, authors name, `commit SHA`, and commit date. For each project, we extracted history of all methods (including methods that were later removed) by iterating backwards from the reference commit on the main branch.

After using *CodeShovel* to extract the complete change history for the 1,598,592 methods in our 53 project corpus, we computed key metrics for each method at each revision using a tool that we have implemented for this purpose. The tool uses the `javaparser`[3] library and parses the source code of each method at each changed commit to compute required metrics. To sanity check our metrics calculations, two authors randomly selected 1000 Java methods, finding no inconsistencies in our expected values for each metric on each method. We represented the change in metric value at each commit for a given method as an array of metric values.

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

3.3. Dataset Description

The raw dataset contains 1,598,592 [4] Java methods with a total of 3,171,244 revisions. It contains meta information of each method such as the author's name, commit timestamp, commit message, and SHA. The dataset also includes the actual source code, file path, and metric values that reflect the state of the method at each commit. To facilitate our research objectives, we extracted and structured the data from the raw dataset in a systematic manner. Our resultant dataset encompasses a collection of 53 JSON files, each belonging to an individual Java project. Each JSON file consists of a JSON object that includes a key representing each method within a given repository. These keys serve as unique method identifiers and allow us to link the data with the bug and raw datasets.

The metric values for each commit are represented in an array, where the first index always represents the state when the method was first introduced into the repository. For example, a method with SLOC (source lines of code) values [10, 20, 30] had a SLOC value of 10 when it was first introduced. The total length of the array (excluding the introduction commit) represents the number of changes the method underwent. The meta-information contained within the dataset also includes details such as the authors responsible for committing changes, the type of changes made, and the diff sizes. The bug dataset, contained within the *repo-bug-data* folder, similarly encompasses 53 JSON files, each corresponding to a specific repository, and includes the method ID to facilitate linking to the relevant dataset.

3.4. Maintenance Indicator Selection

This evaluation uses change-proneness and bug-proneness as exemplar analyses against which different methodological decisions can be evaluated.

3.4.1. Change-proneness

The number of revisions to a code unit has been used widely by the research community as an indication of maintenance effort [85, 55, 100]. A common software design idiom is that "code should be open to extension, but closed to modification". In terms of methods, this means that well-designed methods should not need many revisions, as a given method should mainly need to be changed to fix bugs, not add new features. To measure the change-proneness of the method we count the ***#Revisions***, the total number of times a method was changed regardless of the type of modification applied. For RQ4, we also considered the nature (essential vs. non-essential) and size of changes in addition to the ***#Revisions***. The CodeShovel tool, for a given method,

[4] 1,598,592 Java methods refers to all the methods that were present in a specific reference commit, except for those that were removed before that commit

Table 1
Project details for 53 open-source projects sorted by #methods in descending order. The table also shows the popularity of each project (e.g., number of stars, and contributors), in addition to the size of each project (e.g., number of files, and methods).

Repo	Stars	#Cont	Ref. Commit	Years	#Methods	#Files	TtlSLOC	TtlRevisions
sonarqube	5,937	137	39abd3de7	10.8	130,333	7,293	517,951	249,504
flink	16,764	908	6a8b03011	10.2	118,815	11,282	1,259,298	227,872
wildfly	2,593	343	a9e061675	10.9	112,098	10,081	616,780	153,071
hibernate-orm	4,678	420	cfc7b9725	13.6	95,135	10,260	779,669	179,706
spring-framework	43,641	544	42061d27b	12.3	90,138	7,512	696,598	243,270
docx4j	1,620	37	26aaa13fe	13.2	68,568	3,897	321,657	96,542
RxJava	44,922	278	82f489e1d	9.1	68,428	1,870	312,499	96,018
guava	41,775	264	eec5e6d76	11.7	66,468	1,979	377,142	68,277
soot	1,860	109	beb5a98be	24.4	66,256	3,684	391,457	103,727
jetty.project	3,160	159	4c67b886a	11.9	51,534	2,948	429,856	109,289
spring-boot	56,378	828	7a3bd6d44	8.3	49,025	5,674	330,089	117,460
assertj-core	1,981	259	f8730b34c	11.0	48,611	4,466	192,774	61,390
drill	1,562	168	771c81194	9.3	48,539	4,289	576,427	56,317
wicket	570	83	e8cdd56db	16.5	46,598	3,256	216,748	147,121
hibernate-search	390	58	994cb9e82	13.5	42,596	4,044	263,523	136,040
netty	27,185	526	6724786dc	12.7	38,016	2,700	300,383	68,104
ant	303	58	5f8c6370a	21.1	33,616	1,317	145,017	78,089
pmd	3,482	229	848ec2e7b	18.6	31,604	2,832	142,831	108,589
mongo-java-driver	2,422	151	38117f16f	11.9	27,991	1,648	133,717	74,750
antlr4	10,306	259	dc6678847	11.3	27,909	652	56,952	22,107
DSpace	573	146	3c1b90d60	19.2	26,394	2,551	261,003	61,574
openmrs-core	1,017	371	ffcc19911	15.2	24,382	1,149	126,645	63,931
fastjson	23,594	173	9b8b18677	9.7	20,768	2,982	179,371	9,460
javaparser	3,771	157	1ce7d85c2	9.3	20,309	1,651	175,026	72,545
logging-log4j2	1,230	122	e24e2e890	11.1	19,997	2,262	173,812	46,656
jgit	983	140	4560bdf7e	10.8	19,010	1,587	232,145	30,665
titan	5,135	34	ee226e524	3.8	16,796	904	74,510	27,617
checkstyle	6,115	287	3d8b43445	19.6	16,350	2,702	239,197	51,991
commons-lang	2,138	160	a1495290b	18.6	13,944	397	84,443	39,937
voldemort	2,475	65	a7dbdea58	6.4	12,157	997	175,868	19,722
cucumber-jvm	2,291	226	bb8f6bb1e	12.4	10,678	777	43,999	25,250
junit5	4,691	167	da2dee2bf	5.8	9,928	1,190	80,768	35,207
swagger-core	6,796	202	567bb88c9	9.8	9,653	761	50,583	13,836
mockito	12,043	221	43facffb6	13.2	8,991	944	55,569	27,011
mybatis-3	15,965	182	b791f4476	11.1	8,850	1,238	60,825	15,166
lombok	10,391	114	5120abe47	11.7	7,789	1,503	84,609	15,509
flyway	5,987	2,821	03bec1cc4	11.2	7,697	445	24,948	12,298
atmosphere	3,524	111	75f568553	10.7	7,466	417	41,605	13,463
okhttp	40,428	236	edf477cb4	9.7	7,387	158	37,132	39,574
vraptor4	343	48	593ce9ad6	5.6	6,988	551	26,725	9,049
astyanax	1,021	55	dadde734e	9.8	6,391	618	55,399	9,732
Essentials	1,086	215	2c68d1b86	10.1	6,381	442	41,610	19,943
Hystrix	21,751	109	3cb215898	6.7	5,890	411	50,510	7,231
Twitter4J	2,563	132	8376fade8	11.2	5,546	411	31,577	20,667
jna	6,655	143	a0641dd92	23.2	4,787	564	87,353	13,726
junit4	8,160	151	02aaa01b	20.9	4,762	471	31,242	13,060
hawtio	1,233	102	bbd681905	8.5	4,642	204	15,259	4,140
truth	2,296	83	60dcd0931	10.0	4,609	185	34,461	13,154
commons-io	787	76	29b70e15	19.0	4,207	360	39,836	16,667
rabbitmq-java-client	996	52	cc7c86773	13.2	4,161	367	32,052	10,092
hector	648	71	a302e68ca	10.8	3,682	459	31,365	6,123
gson	19,809	109	ceae88bd	12.4	3,633	207	25,280	5,781
spark	30,409	1,694	54079b0f9	10.7	2,089	986	79,958	3,224
Total	518,433	14,793	-	653.0	1,598,592	122,535	10,846,053	3,171,244
Mean	9,782	279	-	12.0	30,162	2,312	204,643	59,835
Median	3,482	157	-	11.0	16,796	1,238	126,645	35,207
Min	303	34	-	4.0	2,089	158	15,259	3,224
Max	56,378	2,821	-	24.0	130,333	11,282	1,259,298	249,504

provides a `JSON` file containing all the commit SHAs for commits that modified the method. Counting the number of commits for a method automatically provides the ***#Revisions*** that the method underwent. Each commit also contains the current source code of the method, in addition to the `git diff` which contains all the newly added lines and deleted lines between the current method version and the previous method version. With this source code information, we have calculated different change sizes and types (e.g., diff size, addition only, and edit distance) of a source code method that we use in RQ4. For example, just calculating how many lines in the `git diff` record start with a `+` gives how many new lines were added in a particular commit for a given method. If we take the sum of all the values that we get for each of the change commits, we get the total newlines that were added to a method during the study period. To capture the edit distance, we have used the *getLevenshteinDistance ()* method from the Java `StringUtils` class.

3.4.2. Bug-proneness

The CodeShovel tool capture the commit messages of each of the change commits for a given method. The bug-proneness of any method is defined as the number of its change commits that are bug-fix commits. Previous studies have used different sets of bug-related keywords to identify if a commit is a bug-fix or not: a commit is a bug-fix commit if its message contains any of those keywords. Zhang and Hassan [51] used *'bug'* ,*'fix'*, *'error'*, *'issue'*, *'crash'*, *'problem'*, *'fail'*, *'defect'*, and *'patch'* while Ray et al. [101] used *'error'*, *'bug'*, *'fix'*, *'issue'*, *'mistake'*, *'incorrect'*, *'fault'*, *'defect'*, *'flaw'*, and *'type'* as the bug-related keywords. We considered the subset from the union of all keywords used in earlier studies [101, 36, 102] resulting in a list of 10 keywords: *error, bug, fixes, fixing, fixed, mistake, incorrect, fault, defect,* and *flaw*. From the previous works, we have excluded the *'issue'* and *'type'* keywords, because our manual inspection suggested that they produce too many false positives. We searched for exact matches of these keywords from the commit message, partial matches were ignored.

3.5. Statistical Tests

Throughout this analysis, we use a consistent set of tests.

3.5.1. Correlation Analysis

Correlation analysis is usually the first step for building a bug prediction model [103, 52, 21, 51]. These analyses measure the strength of association and the direction of the relationship between two variables. If the two variables move in the same direction, those variables are said to have a positive correlation. If they move in opposite directions, then they have a negative correlation. The strength of correlation ranges from -1 to $+1$, where -1 indicates a strong negative and $+1$ indicates a strong positive correlation. A value of 0 indicates the two variables are completely independent [51, 103]. Correlation analysis is important as it helps to select predictors to build models [26, 104, 51]. A strong association between the outcome and the independent variable implies the variable is a good candidate for bug prediction models [51]. Furthermore, a strong correlation between multiple predictors can help researchers remove unnecessary predictors to reduce model complexity.

There are several types of correlation coefficient statistics, including *Pearson r*, *Kendall* τ, and *Spearman* ρ [105]. The selection of a particular analysis depends on the distribution of data and the presence of outliers. *Pearson r* is a parametric test and assumes the data are normally distributed. We applied the Anderson-darling normality test [106] for several projects and found that their code metrics' distributions are not normally distributed.

Spearman ρ and *Kendall* τ are non-parametric tests that work with rank-ordered variables [105]. For our analysis, we selected *Kendall* τ because it is more robust to outliers, and commonly adopted in many SE research studies [107, 44, 26].

3.5.2. Statistical tests

In cases when we needed to test if two given distributions are statistically significantly different, we used the two-sided Mann-Whitney U [105] test with a confidence interval of 95 percent (i.e., $\alpha = 0.05$). This is a non-parametric test and does not assume any distribution of the data.

To quantify the size of the difference between two distributions, we used Cliff's δ [108]. For effect sizes, Cliff's δ (a non-parametric test) represents the degree of overlap between two samples. This is more accurate and robust than Cohen's d [109]. The δ value ranges between -1 and $+1$ where a negative value implies that the second sample values are greater than the first sample and a positive value indicates the opposite. We used prior work mapping of Cliff's δ value to label the degree of effect size [51, 110]: *negligible* ($0 \leq |\delta| < 0.147$), *small* ($0.147 \leq |\delta| < 0.330$), *medium* ($0.330 \leq |\delta| < 0.474$) and *large* ($0.474 \leq |\delta| \leq 1$). Unless otherwise stated, all the presented results in this paper are statistically significant ($P < 0.05$).

3.5.3. Versioning technique for code metrics

As we mentioned earlier, a method can have different SLOC values while it evolves. Consider a method that was revised 5 times with SLOC values 10 (r0), 20 (r1), 10 (r2), 20 (r3), and 50 (r4). Here the value of the dependent variable is 4 (i.e., #Revisions after the method's introduction r0), but what would be the value of the independent variable? Should the SLOC be the introduction value (10), the most recent (50), or the mean (22)? Although using the mean is a popular choice among the research community [35, 111, 7, 51], it is sensitive to skewness [112]. Zhang and Hassan also investigated dispersion, such as standard deviation, as an aggregation scheme for method-level to file-level data to determine its impact on bug prediction models [51].

While all these can be valid decisions for training change or bug prediction models, these SLOC measurements do not represent the actual number of changes or bugs triggered by a particular SLOC value. For example, if the initial SLOC value represents this method, we have to associate SLOC 10 with four revisions, which would be inaccurate, because

SLOC 10 is responsible for two revisions only. Chowdhury et al. [44, 61] claimed that the versioning technique is more accurate where the above method will have three versions that map each SLOC responsible for inducing a change or bug-fix commit. With the versioning technique, the method in the above example has the following SLOC values: SLOC 10, SLOC 20, and SLOC 50. SLOC 10 maps to 2 revisions, SLOC 20 maps to 2 revisions; SLOC 50 will map to 0 revisions because it did not induce any change. The versioning technique is frequently applied in bug-prediction research where code metrics are calculated from the version where a code component was involved in a bug-fix commit (e.g., [42, 40, 5]).

The following pseudo-code represents the algorithm for calculating the versioned source lines of code (SLOC) for a single method.

```
result ← { }
last_sloc ← SLOC[SLOC.length − 1]
if SLOC.length > 1 then
    // SLOC.length-2 skips the last element in the array
    for i ← 0 to SLOC.length − 2 do
        if SLOC[i] present in result then
            result[SLOC[i]] ← result[SLOC[i]] + 1
        else
            result[SLOC[i]] ← 1
        end if
    end for
    if last_sloc is not present in result then
        result[last_sloc] ← 0
    end if
else
    result[last_sloc] ← 0
end if
```

4. Results: Methodological Decision Impact

In this section we detail the resulting impact of reasonable and commonly used alternatives for the five methodological decision points researchers and practitioners must reason about when deploying analyses based on historical software evolution data.

4.1. RQ1: Does the SLOC selection methodology impact evolutionary analyses?

Can the common scenario encountered by prior relevant studies [71, 45, 44, 72, 46, 15] while calculating a code metric value induce contradictory findings about a code metric's effectiveness? Figure 1a shows the cumulative distribution function (CDF) of the correlation coefficients between SLOC and #Revisions for all 53 projects. According to the figure, all measurement techniques (except versioned SLOC) exhibit similar performance where Cliff's δ effect size is negligible in most cases. This is not surprising because a large number of the methods (41%) were never modified after their introduction, having identical values for their introduction, last, mean, and median since they all measured the same single commit. Our finding about the large number of unmodified methods complements earlier studies [70, 113]. However, when considering versioned SLOC we observe that for ∼50% of the projects, the correlation values are higher compared to the other measurement techniques.

For better insights, we also reevaluated the correlation values by considering methods that were revised at least once (Figure 1b). Now, the correlation values of different SLOC measures with #Revisions are significantly higher than the values obtained using versioned SLOC. With SLOC versioning, we observe that the strength of the correlation between SLOC and maintenance is not as strong as has been previously claimed (e.g., [26, 90, 27, 38]). This observation emphasizes the importance of methodological choices, since various SLOC aggregation approaches, such as using the mean SLOC for a method, can overstate results if the versioning technique is considered more accurate.

Similarly, considering methods that change at least once, the correlation of SLOC with #Bugs (Figure 1c) also shows that other SLOC measures exhibit higher correlation than versioned SLOC for most projects.

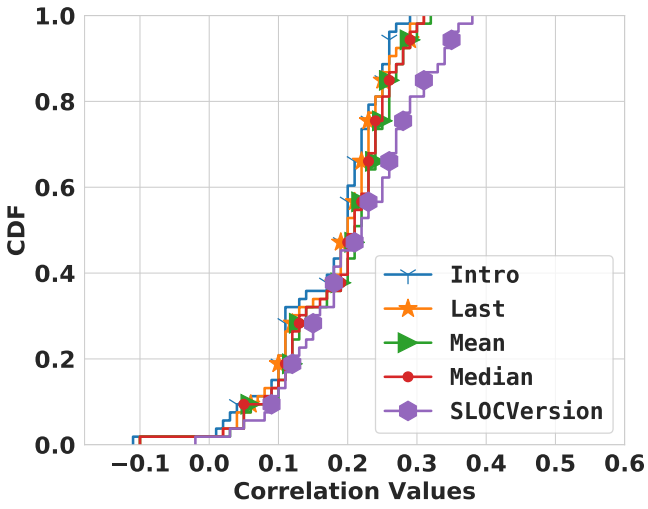
RQ1 Summary

Most of the SLOC measurement techniques, such as using only the introduction, the last, or the mean value, show similar performance while estimating change- and bug-proneness. These approaches, however, can produce very different observations when compared to the versioning techniques which is also common in the literature [42, 40, 5, 61, 69, 44].

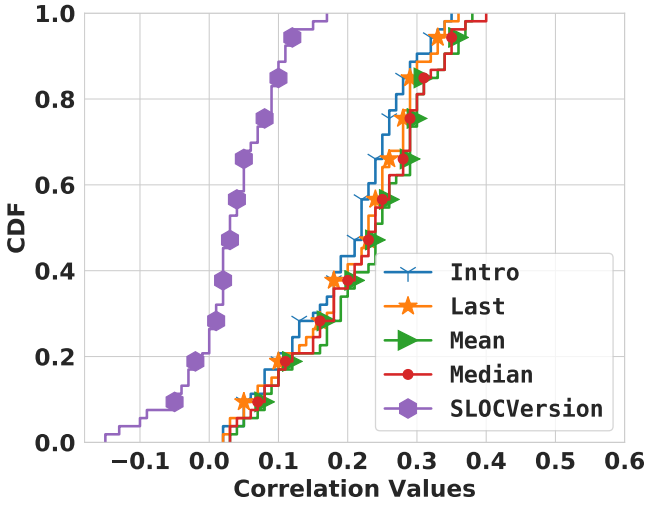
4.2. RQ2: Does normalized size produce different results?

Consider a method with SLOC $s1$ that was associated with n revisions, and a second method with SLOC $s2$ and m revisions. Usually [26, 61], $s1$ and $s2$ are treated as the values of the independent variable SLOC, whereas n and m are treated as the values of the dependent variable #Revisions. This is also true while performing bug-proneness analysis. However, the number of revisions per line of code (change-density) or the number of bugs per line of code (bug-density) can also be considered as the dependent variable [75]. In that case, n and m will be replaced by $n/SLOC$ and $m/SLOC$, respectively. Do these two different approaches produce different results?

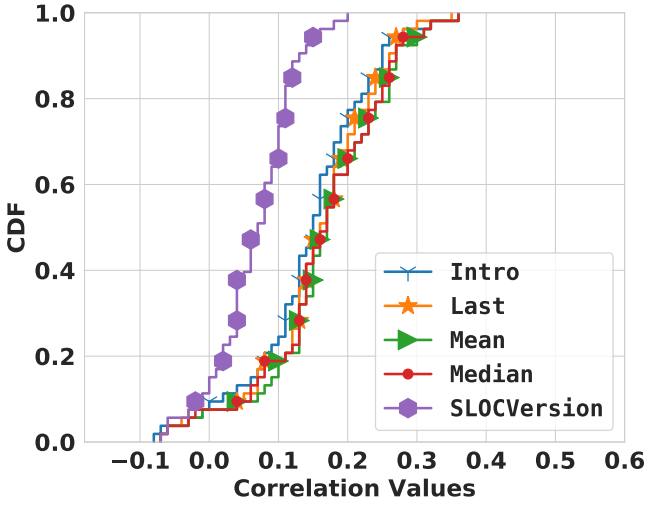
Figure 2a shows that for all 53 projects, the correlations between SLOC and #Bugs (and #Revisions) are positive, complementing previous studies [114, 18, 29, 30]. However, when density is considered, the outcome becomes less clear. For the revision density, the correlations are negative for ∼70% of the projects. This clearly suggests that small methods have a higher density in the number of revisions. This would be in contradiction with previous studies (e.g., [114, 18, 29, 30]) if the density in the number of revisions and the total number of revisions are used interchangeably as an indication of change-proneness.



(a) SLOC with #Revisions (all)

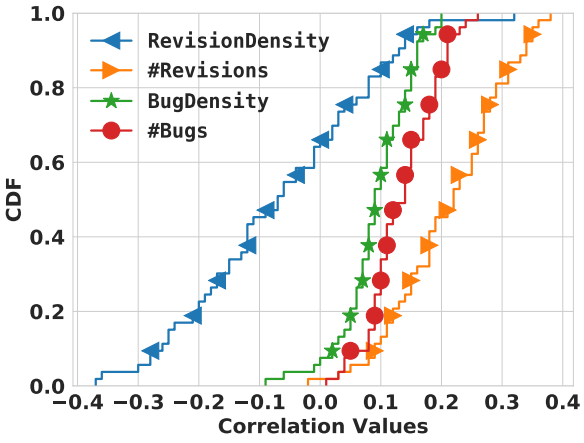


(b) SLOC with #Revisions (at least one revision)

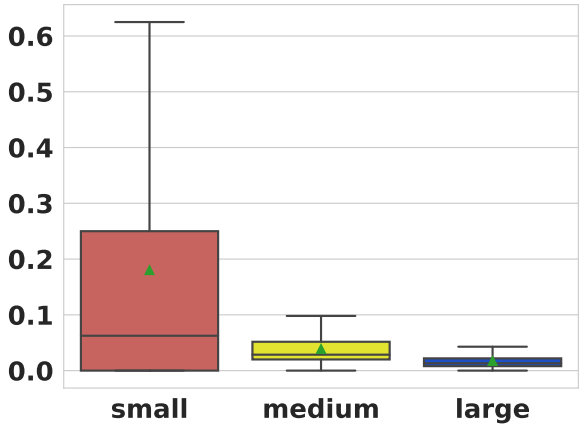


(c) SLOC with #Bug (at least one revision)

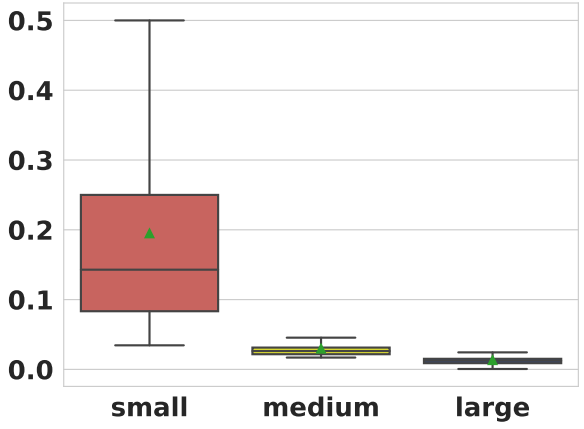
Figure 1: Distribution of correlation coefficients between different SLOC measurements with number of revisions and bugs for all 53 projects. For graph readability, we marked after every 5 points only.



(a) SLOC density with #Revisions/#Bugs



(b) Revisions per SLOC distribution



(c) Bugs per SLOC distribution

Figure 2: (a) compares the correlation distribution of #Revisions and #Bugs with their respective densities. (b) and (c) show their density distributions by grouping methods into small, medium, and large. In (c), we considered methods that have at least one bug.

Bug density, however, does not show the same trend. For ∼90% of the projects the correlations are still positive, which is quite surprising because change-proneness and bug-proneness are supposed to be correlated [85, 35], thus should not exhibit two very different correlations with SLOC.

To have a deeper insight, we further explore the distribution of these density values by grouping methods in different SLOC categories based on Spadini et al.'s [18] work: small ($SLOC < 30$), medium ($60 > SLOC > 30$), and large ($SLOC > 60$). Figure 2b shows that methods in the small group have a higher revision density than the

methods in medium and larger groups, consistent with our observation from Figure 2a. When we draw the same graphs for bug density, the graph becomes unreadable, because most methods do not have a bug associated with them (83%). This extremely skewed bug distribution could explain the inconsistent result we observe in Figure 2a. Indeed, when we consider methods with at least one bug (Figure 2c) we now clearly see that even bug density is higher in smaller methods.

A natural question is whether `getter` and `setter` methods impacted these results. We, therefore, repeated the experiments after excluding them but observed similar results. One may also argue that the correlation with density becomes negative due to the inverse relation between $SLOC$ and ($\#Revisions/SLOC$) or ($\#Bugs/SLOC$) [115]. This argument, however, is invalid because #Revisions and #Bugs are not constant across methods with different sizes [61].

RQ2 Summary

Considering SLOC as an example of code metrics, we found that maintenance analysis can be contradictory if the evaluation context is not considered. Large methods are more maintenance-prone when the total number of revisions or bugs is considered. When density is considered, smaller methods are, surprisingly, more maintenance-prone than larger methods.

4.3. RQ3: Do different ages of methods and temporal choices induce inconsistent results for change/bug proneness?

Previous studies on code maintenance either did not control for method age or were not explicit about this control [29, 30, 116, 40]. This oversight is significant because the age of a code component is relevant when estimating its future maintenance burdens [69, 44]. While calculating the correlation between age and the number of revisions (or bugs) reveals a weak relationship—partly because many methods in our datasets have never been changed—considering the last change date of a method as an indicator of its age shows a stronger correlation. Specifically, the correlation between method age and the number of bugs is 0.36, and with the number of revisions, it is 0.73.

These significant correlation coefficients indicate that age is an important factor that should be experimentally controlled—e.g., we should not compare a newly introduced method with a five-year-old method.

To get more insight into the impact of age normalization, we followed two steps. In step 1, we removed all the methods that are less than x years of age. In the filtered samples, all the methods will be at least x years of age (x can be any value that we show later). This makes sure no methods in our dataset have an age less than x years. Unfortunately, we may still compare an x year-old method with an $(x + i)$ year-old method. Therefore, in step 2, we removed all the revisions and bug-related data that occurred after x years of a method's life.

Figure 3a compares the correlation between SLOC and #Revisions with no age-normalization and age normalization for different values of x. We observe that without age control, the correlation of SLOC with #Revisions is underestimated for most projects when compared with age-normalized methods. This is particularly true when the value of `x` is 0.5 years (Figure 3a). As a result, without age normalization, the interpretation of comparing correlation values is inaccurate. The performance difference between 0.5 years with no age control is statistically significant (Mann-Whitney U test) with a medium effect size (Cliff's δ).

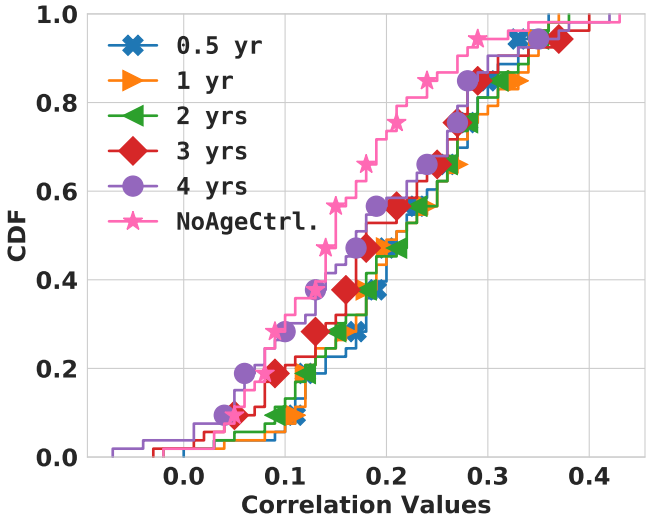
In contrast, when correlating SLOC values with #Bugs, we did not observe any significant results when comparing the different time ranges with the no age control group. For example, the performance difference between 0.5 years and no age control group is not statistically significant ($p > 0.05$).

We also evaluated if the choice of intervals, for capturing change and bug information, impacts code metric performance. For an interval between year `x` and year `y`, we removed all the methods with less than `y` years of age. Next, from the filtered samples, we consider changes and bugs that occur only between year `x` and year `y`. Consistent with our earlier observation, the performance of SLOC is much better (higher correlation coefficients with #Revisions) for 0-0.5 year interval than the other intervals (Figure 3b). Although to a lesser extent, this observation is true for bugs as well (Figure 3c).

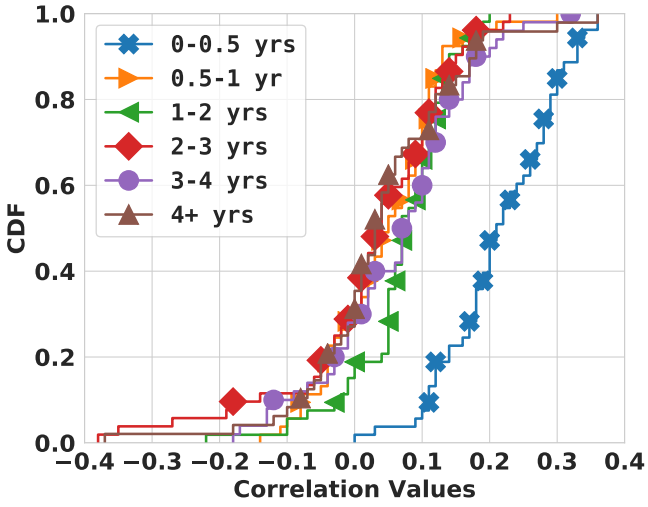
As age normalization is important for accurate observations, for all the other RQs we have used two years of age normalization with which we were able to retain 88% of the methods and 65% of the change history that those methods underwent. Had we considered less than two years, we would have significantly decreased our method histories potentially biasing our results. With more than two years of age normalization, on the other hand, would reduce the number of methods significantly.

RQ3 Summary

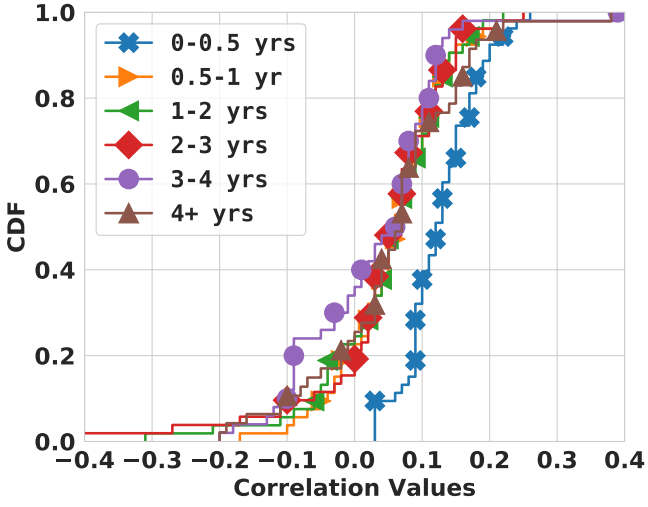
Age normalization is important for understanding the true maintenance impact of code metrics (e.g., SLOC). The amount of history analyzed for a method influences the predictive power of the model. Therefore, using different projects' snapshots in different research studies hinders making a generalized observation about code metrics' impact on software maintenance. This may also explain the contradictory findings in earlier studies [44] about code metrics' effectiveness. We found that code metrics, such as SLOC, show the highest correlation with maintenance in the first six months of a method history.



(a) SLOC with #Revisions



(b) SLOC with #Revisions



(c) SLOC with #Bugs

Figure 3: (a) compares different correlation coefficients of SLOC with #Revisions with no age normalization for 53 projects. (b) and (c) is the distribution of correlation coefficients at different intervals between SLOC with #Revisions and #Bugs for 53 projects. For graph readability, we marked after every 5 points only.

4.4. RQ4: Can the size and type of changes impact maintenance analysis?

Many code metric studies look at the total number of revisions without classifying the size of changes and use it as a maintenance indicator [85, 55, 100]. Some studies, however, focused on change sizes, such as the diff size [84, 85]—the number of added and deleted lines; new additions [85]—the number of added lines only; and edit distance [86, 39, 84]—the number of characters that need to be added, deleted, or updated to convert one source code version into another. For a specific method version, we have added all of its diff sizes from all of its change commits while calculating the total diff size for that method version. We repeated this for other change size indicators as well. We investigate if these variations in change-proneness measurements produce different results while associating with code metrics. Figure 4 shows that the three change size indicators perform very similarly while analyzing their correlation with SLOC. However, when only the #Revisions are considered, the performance of SLOC dwindles significantly. This indicates that studies that considered #Revisions as the only change-proneness indicators were underestimating the predictive power of source code metrics.

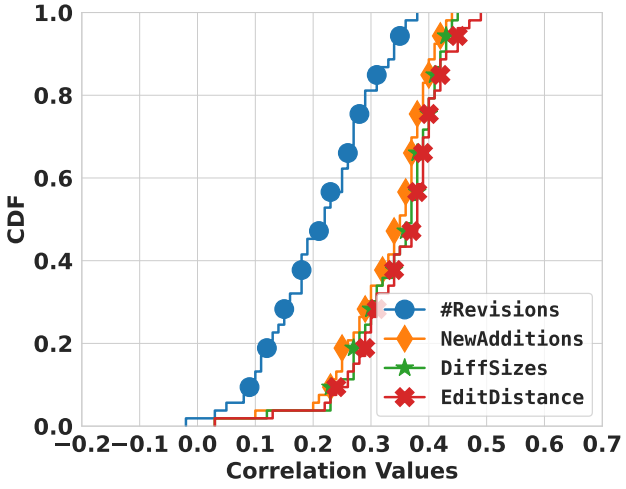


Figure 4: Cumulative distribution function (CDF) of the correlation coefficients between SLOC and different change sizes. For graph readability, we marked at every five points only.

A method can go through many changes without significantly changing its behavior. For example, through code refactoring, a method can be moved to a new file, or its container file can be renamed. Changes can also happen to a method's documentation. CodeShovel, the tool we used to trace a method's change history, captured all of these changes. It is possible that practitioners when they aim to predict maintenance-prone methods, may like to ignore these types of changes from their analysis. For example, Kawrykow and Robillard found that out of all changes applied to a method, 15.5% were trivial [87]. As a result, the total number of changes performed on a method might exaggerate the number of important changes performed on it.

To facilitate analysis considering different types of changes, we further decompose the number of revisions according to the type of change and examine the correlation of different kinds of changes with SLOC to understand the change-proneness of a method. CodeShovel provides these raw change categories as it traverses the method history. We consider the following four types of changes, with some overlaps, to observe if types of changes can impact code metrics and maintenance analysis. And if so, how much is the impact?

1. **#EssentialChanges:** All changes that modify the content of a method's signature or its body. Method signature changes include modifier changes (e.g., from `public` to `private`), parameter changes (either in type or name), exception changes, parameter meta-information change (e.g., `final` keyword added), method rename, and return type changes.

2. **#BodyChangesOnly:** Body change refers to a change made to the implementation of a method. This can include modifications to the statements or expressions that make up the method's body, as well as changes to the structure or control flow of the method. In our work, all changes made to the body of a method, with the exception of cosmetic changes such as formatting or whitespace, are categorized as *BodyChange*.

3. **#NonEssentialChanges:** All changes related to formatting, annotation changes (e.g., `@test`, `@suppress`), and documentation changes. Also includes cross-file changes such as renaming the file or moving methods from one file to another. Formatting changes include both white space and indentation changes.

4. **#Revisions:** Any revision of a method, including non-essential changes.

Figure 5 shows that significantly different outcomes in correlation values can be obtained by accounting for the type of changes. We found that the correlations of SLOC with #BodyChanges and #EssentialChanges are much higher compared with #NonEssentialChanges. For example, if #Revisions is calculated with #EssentialChanges only, the correlation coefficient is $\geq$ 0.3 for 60% of our projects, but no project has such correlation coefficient if #Revisions is calculated with #NonEssentialChanges only. Clearly, the performance with the #Revisions is significantly impacted because of #NonEssentialChanges.

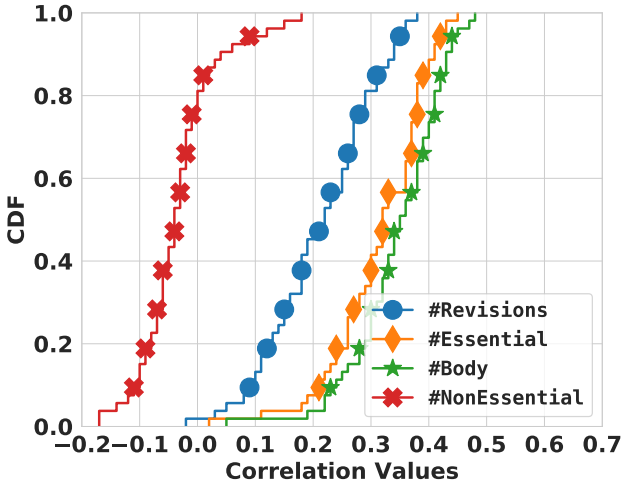


Figure 5: Distribution of correlation coefficients of SLOC with different types of changes for 53 projects. For graph readability, we marked after every 5 points only.

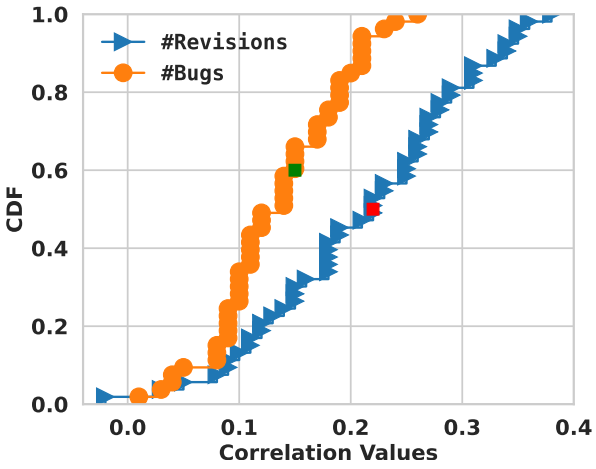


Figure 6: Distribution of correlation coefficients of SLOC with #Revisions and #Bugs for 53 projects. Red and green square denotes the aggregated correlation coefficients between SLOC and #Revisions, and SLOC and #Bugs, respectively.

RQ4 Summary

Prior studies have differed in how they calculated change-proneness (both in size and type). Our results show that these differences can be one of the many root causes behind the contradictory findings about code metrics and maintenance. Non-essential changes, such as file renaming, and method move can degrade the predictive power of code metrics.

4.5. RQ5: Do aggregate project analyses meaningfully reflect the correlation for individual project analyses?

Prior work is broadly divided into two categories for analyzing software projects related to code metrics and maintenance studies. While some studies aggregated all the code metric data from their selected projects to produce a single statistical value to understand maintenance [18, 26, 35], others opted for individual project analysis [90, 85, 22, 88]. Can these two approaches produce significantly different results?

We explore the differences between aggregated and individual project analyses further with our dataset. Combining all projects, we found that the overall correlation coefficient for SLOC with #Revisions is 0.22 and is 0.15 for #Bugs (Figure 6). We observe that almost half of the projects are within ± 0.05 of these aggregated values for SLOC with #Revisions (∼40%) and #Bugs (∼52%). We selected 0.05 as this is $1/20^{th}$ of the max correlation value (1.0) and is small enough to capture all projects close to aggregated value. Therefore, aggregated project analysis seems to represent correlation accurately only for ∼50% of the projects. Given the spread across the correlation values, the chances of an aggregate value overestimating or underestimating the #Revisions and #Bugs is high. At the same time, individual project analysis can be misleading if the number of selected projects is very small. From Figure 6, we observe that for some projects the correlations are close to 0, whereas for some others the correlations are close to 0.4—two very extreme observations. As a result, code metrics research should rely on individual project analysis, but with a significantly large number of projects.

RQ5 Summary

Per-project correlations differ meaningfully from aggregate analyses for maintenance indicator prediction. Correlations should be computed on individual projects, as aggregated analysis can be impacted by a few outliers. To produce reliable conclusions, a large number of projects should be analyzed.

5. Discussion

In this section, we discuss the implications of our findings along with the threats to the validity of our analysis.

5.1. Guidelines for Future Studies

When preparing a data collection pipeline it is crucial that the data be collected and pre-processed in a methodologically sound manner. In this work, we demonstrated the impact of different decisions on large historical datasets that can lead to inconsistent conclusions depending on methodological choices.

Since the SLOC values of most methods do not change [70, 113], the correlation values for a method's introduction, its last commit, and median provide consistent results with #Revisions and #Bugs. One may argue that, out of these available options, practitioners can use them interchangeably to train models on past historical data and predict future changes and bugs without much contradiction in the outcome. However,

these measurements overestimate the association of SLOC with maintenance indicators and are not truly effective as one might presume when compared with a more accurate representation of SLOC to their respective changes and bugs (e.g., SLOC versioning). This is probably one of the reasons for unsuccessful maintenance prediction models when evaluated with realistic scenarios [26, 42, 69].

Recommendation

We recommend that future studies should use the SLOC versioning technique for more consistent outcomes, because other aggregated techniques, such as mean, do not accurately map a code metric to its corresponding maintenance.

Once a representation for SLOC is chosen, researchers and practitioners should decide (and report) how they calculate change- and bug-proneness: are they considering the total number of changes and bugs or the change or bug density? These decisions are important because the outcomes are completely opposite based on the prediction variables.

Recommendation

For replicability, extension, and interpretation, future studies should report and justify their rationale for selecting total changes (or bugs) instead of change or bug density, and vice versa.

The next decision is deciding on a timeframe for model training. Our results indicate that unstable conclusions and predictive models can be generated if different subsets are used from different timeframes. This is mainly due to concept drift [117, 69, 118]. A previous study [69] reported that even a complex code component may become less bug-prone as it has probably gone through enough corrective, and perfective changes. Our result indicates that the first six months of method history show the strongest correlation between code metrics and different maintenance indicators.

Recommendation

To understand the accurate impact of code metrics, researchers should select multiple snapshots of the same project. The impact of code metrics on maintenance reduces over time. Therefore, researchers should explain and justify their selection of a particular project snapshot if they use only one.

A method can experience a large number of changes in its lifetime. While each of these changes has a purpose, a big fraction of these changes are unrelated to the actual content of a given method and are often not harmful. Therefore, change classification should be performed so that models do not overestimate or underestimate future changes.

Recommendation

Change analysis should be performed only for important changes, discarding non-essential changes from the dataset, such as method move and file rename. These non-essential changes reduce the correlation strength between code metrics and maintenance.

Finally, our results indicate that aggregated project analysis can be misleading, especially if there are outlier projects in the dataset. Similarly, individual project analysis can produce inaccurate conclusions if very few projects are analyzed.

Recommendation

Researchers should analyze a large number of projects individually to understand when and why code metrics are good maintenance indicators (context is king). In case when aggregated analysis is necessary, they should consider the impact of outlier projects, and can probably discard them from their analysis.

A pertinent question may arise regarding the relevance of our findings in the context of recent studies that have explored embedding-based models to enhance bug prediction. Code metrics have been essential for the past four decades and continue to be highly relevant today, as demonstrated by recent research such as that conducted by Aladics *et al.* [3]. Their study demonstrated that bug prediction models achieve optimal performance when combining embeddings with code metrics. In some instances, such as with models using the RandomForest algorithm, code metrics alone can outperform embedding-based models. Additionally, Mashhadi *et al.* [4] showed that incorporating code metrics alongside code representation (e.g., token embeddings) can enhance the performance of LLM-based bug prediction models, like CodeBert. This underscores the continued relevance of code metrics.

Our recommendations are also applicable to maintenance studies that concentrate solely on embedding features. For example, RQ4 explores how the selection of dependent variables (e.g., change sizes and types) can significantly influence study outcomes, while RQ5 examines the impact of analyzing individual versus aggregated projects. These insights are crucial for embedding-based research, as the variability in embeddings across different projects can significantly affect model performance.

5.2. Threats to Validity

As with any complex analysis, this work has several threats:

Conclusion validity: The results of complex software analyses rely on both the correctness of the analysis and the underlying data preparation methodologies. In this work, we

have examined the impact of specific methodological questions to help analysis designers better prepare their data for analysis. To help guard against threats to conclusion validity we verified the underlying data distribution before applying non-parametric tests and used the more conservative *Kendall* τ to evaluate association strengths instead of *Spearman* ρ. To reduce the threat that method heterogeneity could impact our findings, we examined small, medium, and large methods independently when it was necessary.

We have employed *size* as the representative of other code metrics based on previous research indicating that size exhibits a strong correlation with various metrics [26, 59]. Consequently, our conclusions could be challenged. Future studies should explore whether our findings remain valid when considering alternative code metrics beyond just size.

Internal validity: The precision of our keyword-based bug labeling approach can be impacted due to the tangled code changes [82]. Identifying the unrelated code changes in a bug-fix commit, unfortunately, is an open and active research problem [119, 69]. The findings in this paper rely on the method histories being accurate and complete. While the *CodeShovel* tool has been shown to have high accuracy when finding the complete history for 90% of methods, including 97% of historical changes, any automated history-tracking approach is bound to miss some changes. While SLOC is the most commonly used metric used for predicting change- and bug-proneness, other metrics that may not correlate with SLOC could also be appropriate for future investigation. This paper investigates a limited number of aggregation techniques at method-level; other measures like Gini [120], Atkinson [121], and Shannon entropy [122] might produce different results.

Construct validity: Our tool for computing metrics for each method at each revision can induce measurement error since the process is fully automated. To reduce this threat, two authors verified the correctness of our tool by randomly selecting and validating 1000 Java methods for all collected metrics. Also, while evaluating the impact of the methodological choices for a given code metric, we kept the choice for other code metrics the same. For example, in RQ1, while we were investigating the impact of different ways of SLOC measurements, we used #Revisions as the indicator of change-proneness. If we use all other change-proneness indicators, we have to evaluate at least eight more scenarios for each of the size measurement choices. For instance, for the SLOC measurement with the versioning technique alone, we have to experiment four times for four different revision sizes (Figure 4), and four times for four different revision types (Figure 5). This large number of combinations is true for all research questions. For each research question, if we do not keep other factors constant while evaluating a given factor, the number of figures (and their analysis) will be too many which may impact the presentation and readability of the paper. This, however, can be considered a threat to our observations. We mitigated this threat by experimenting with a few of the possible different experimentation scenarios. The observations are still the same: the choice we make significantly impact the outcome of code metric studies.

External validity: The most significant threat to the generalizability of our results stems from our project selection. All projects were open source systems written in Java. Closed-source systems may exhibit different correlation between SLOC with number of revisions and bugs than open-source systems, as might non-Java systems. Finally, while the age range of the systems was reasonable (with 20 having histories of over 12 years), examining additional systems with different characteristics could also increase the generalizability of our findings. The external validity of our findings could also be further improved by demonstrating the impact of differing choices on the predictive outcomes of already-existing models.

6. Related Work

A variety of inconsistencies has arisen in prior work, driven at least partially by the five methodological decisions examined in this paper.

6.1. Measurement Selection

When performing coarse-grained analyses (e.g., file, module, or system), researchers and practitioners are forced to select an aggregation scheme that combines multiple method-level metrics to a coarse-grained level. One of the most popular choices among the research community is to use the average value of a metric [35, 111, 7, 51]. Zhang et al. [7] built a universal cross-project bug prediction model and aggregated method-level data to file level using max, average, and summation scheme and clustered projects based on the similar distributions of 6 metrics (e.g., lines of code, number of files, number of commits, number of developers, etc.). Similarly, in another work, Zhang and Hassan [51] experimented with different aggregation techniques of software metrics and observed their impact on bug prediction models. They also aggregated method-level data to file level and examined 11 aggregation schemes such as sum, mean, median, standard deviation, Gini [120], Hoover [123], Atkinson [121], Shannon entropy [122], etc., and compared the correlation of these aggregated metrics with SLOC.

While these aggregation schemes are some rational choices a researcher could make, they can interfere with the correlation among metrics [124, 38, 104] and produce inconsistent results. Vasilescu et al. [124] found that values aggregated using mean from method level data to package level leads to inconsistent correlation results of SLOC with bugs for some projects. As these aggregation schemes can also be applied to method [35] having different SLOC values at each revision, contradictory results could also be based on which schemes were used.

6.2. SLOC Control

Most practitioners agree that a file with more lines of code is more bug-prone [125]. Size as a metric has been studied extensively in the literature at various granularity

(e.g., class [111], system [126], packages [29, 30, 116]) and in different contexts. For example, prior work studied the relationship of size such as a large number of test methods per class with the density of test smells [127] and code coverage [107]. Although Greiler et al. [127] found that the number of test methods per class is related to the higher density of test smell, having a higher number of test methods is not indicative of effective test suite [107].

Similarly, several works [59, 74, 26] have analyzed the impact of class size on maintenance prediction models. Kore et al. [128] applied a tree-based classification technique to detect change prone-class and observed that large classes tend to change more than smaller classes. Although the results obtained from these models are insightful, these results are not transferable to other count-based bug prediction models that rely on the number of bugs [29]. A possible cause of this inconsistency could be that the number of bugs in such a system is not being normalized by SLOC appropriately.

Lefever et al. [81] found that after normalizing size, there are substantial inconsistencies in the software quality reporting tools (e.g., SonarQube, DV8, Structure 101) when identifying the most problematic files. Although these disagreements are perceived at a higher level of granularity (e.g., system level [126], classes [111], files [129] or packages [29, 30, 116]), they also exist at method-level, as our study has shown.

6.3. Age Control

Many studies have used time-based sampling techniques to understand software evolution or to build bug prediction models at coarse granularity (e.g., system level [126], classes [111], file, packages [29, 30, 116]). For example, at the system level with a time interval of 3 months, Osman et al. [130] compared the evolution of different types of exception handing in code (e.g., standard exceptions, custom exceptions) between the application and third-party libraries. Additionally, for bug prediction, time-based sampling such as 3 months or 6 months [131] and all data (e.g., [111, 129, 126]) have been explored.

In addition, Zhang et al. [29] examined the pre-and post-release ranking ability of SLOC at the package level and found that 20% of the largest modules were responsible for the majority of the bugs (51-63%). Fenton and Ohlsson [30] observed a similar connection between size metric (SLOC) and the number of faults but in pre-release data. Although these coarse-grained analyses are a popular choice among the research community, the inconsistent results of these studies have been reported as well. Andersson and Runeson [116] replicated the study of [30, 29] and did not observe any ranking ability of SLOC for some projects. Moreover, a negative result was observed in the method-level study [35] at more realistic settings (such as release-by-release validation models) that contradict the findings of previous studies [40].

Menzies et al. [132] studied bug prediction results from 28 recent studies and reported that findings from these studies contradict each other about what influences software bugs. They found that for different chunks of data, a completely different model [77] is learned even from the same project. One possible cause for inconsistent conclusions reported in previous studies could be accounted for the age of code components not being explicitly controlled.

6.4. Change Analysis

Several studies have used the number of revisions as a maintenance indicator [85, 55, 100] by accounting for the type of transformation applied to a given file or method. For example, Koru et al. [128] evaluated change count by looking at the revision history to identify the most change-prone classes. Additionally, Farago et al. [133] observed that files that are committed at a higher frequency (i.e., with longer revisions history) have a negative impact on maintainability than the files with lower commit frequency. Shrikanth et al. [134] observed that the majority of the bugs are localized in the early 50 commits of a project. As commit patterns are usually impacted by practitioners' habits and organizations' culture, some studies [44, 87, 86, 39, 84, 85] considered change size and types as the indicators of change-proneness, instead of solely relying on the number of revisions.

6.5. Data Aggregation

Practitioners working with code metrics to understand software maintenance need to decide whether to analyze projects separately [90, 85, 22, 88] or combine them to obtain a single statistical value for a research question under analysis [18, 26, 35]. Pascarella et al. [35] aggregated 13 projects' bug data together to build a single bug prediction model although the number of bugs in each system under analysis varied substantially. Similarly, Gil and Lalouche [26] combined 26 projects to investigate the confounding effect of SLOC on other metrics, such as depth of inheritance. Spadini et al. [18] combined 10 open-source projects to study the change- and bug proneness of test code.

While aggregated metrics of several projects to obtain a single value is useful, it obscures the effect of outlier projects having a highly skewed distribution of metrics [135]. These outlier projects are the result of software evolution that is impacted by several external factors such as developer expertise, commit pattern [82, 83], time, and budget. Therefore, one might opt for individual analysis. Nagappan et al. [22] studied the ability of change burst to predict the number of bugs and were able to achieve a high accuracy of 90% for Windows and 71% for Eclipse projects by studying them separately. Shin et al. [85] analyzed the Mozilla Firefox web browser and Red Hat Enterprise Linux Kernel separately and investigated the ability of code complexity, churn, and developer activities to discriminate between neutral and vulnerable files and predict vulnerabilities. Although individual project analysis overcomes the problem of aggregated project analysis, it suffers from selection or publication bias [26]. As there is a lack of randomization in selecting these projects, this often results in missing out on projects with unique characteristics.

7. Conclusion

Software developers and researchers make decisions using metrics they have gathered from their project histories. Collecting and analyzing this historical data requires some important methodological choices. In this paper, we demonstrated how common decisions for these choices can meaningfully impact analysis outcomes. We showed how alternate methodological choices for these decisions account for many differences found in the literature. Going forward, we would like to replicate this study with a wider variety of metrics beyond SLOC and build alternative models following the provided guidelines. We hope that the guidelines presented in this work can help improve both the quality of decisions made by practitioners and the consistency of results for research approaches that use historical evolutionary data.

References

[1] L. J. Arthur, Software evolution: the software maintenance challenge, Wiley-Interscience, 1988.

[2] R. C. Seacord, D. Plakosh, G. A. Lewis, Modernizing legacy systems: software technologies, engineering processes, and business practices, 2003.

[3] T. Aladics, J. Jász, R. Ferenc, Bug prediction using source code embedding based on doc2vec, in: International Conference on Computational Science and Its Applications, 2021, pp. 382–397.

[4] E. Mashhadi, H. Ahmadvand, H. Hemmati, Method-level bug severity prediction using source code metrics and llms, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), 2023, pp. 635–646.

[5] R. Ferenc, D. Bán, T. Grósz, T. Gyimóthy, Deep learning in static, metric-based bug prediction, Array 6.

[6] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, Transactions on Software Engineering (TSE) 33 (1) (2006) 2–13.

[7] F. Zhang, A. Mockus, I. Keivanloo, Y. Zou, Towards building a universal defect prediction model with rank transformed predictors, Empirical Software Engineering 21 (5) (2016) 2107–2145.

[8] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, N. Ubayashi, Deepjit: An end-to-end deep learning framework for just-in-time defect prediction, in: International Conference on Mining Software Repositories (MSR), 2019, pp. 34–45.

[9] S. Wang, T. Liu, J. Nam, L. Tan, Deep semantic feature learning for software defect prediction, Transactions on Software Engineering (TSE) 46 (12) (2020) 1267–1293.

[10] E. Arisholm, L. C. Briand, A. Foyen, Dynamic coupling measurement for object-oriented software, Transactions on Software Engineering (TSE) 30 (8) (2004) 491–506.

[11] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, A. De Lucia, A developer centered bug prediction model, Transactions on Software Engineering (TSE) 44 (1) (2017) 5–24.

[12] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, E. J. Whitehead, Does bug prediction support human developers? findings from a google case study, in: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 372–381.

[13] M. Viggiato, J. Oliveira, E. Figueiredo, P. Jamshidi, C. Kästner, How do code changes evolve in different platforms? a mining-based investigation, in: International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 218–222.

[14] A. R. Sharafat, L. Tahvildari, Change prediction in object-oriented software systems: A probabilistic approach., J. Softw. 3 (5) (2008) 26–39.

[15] Y. Zhou, H. Leung, B. Xu, Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness, Transactions on Software Engineering (TSE) 35 (5) (2009) 607–623.

[16] Q. Song, Z. Jia, M. Shepperd, S. Ying, J. Liu, A general software defect-proneness prediction framework, Transactions on Software Engineering (TSE) 37 (3) (2010) 356–370.

[17] A. G. Koru, D. Zhang, H. Liu, Modeling the effect of size on defect proneness for open-source software, in: International Workshop on Predictor Models in Software Engineering, 2007, pp. 10–10.

[18] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, A. Bacchelli, On the relation of test smells to software code quality, in: International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 1–12.

[19] T. J. McCabe, A complexity measure, Transactions on Software Engineering (TSE) (4) (1976) 308–320.

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

[21] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: International conference on Software engineering, 2005, pp. 284–292.

[22] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, B. Murphy, Change bursts as defect predictors, in: International symposium on software reliability engineering, 2010, pp. 309–318.

[23] E. J. Weyuker, T. J. Ostrand, R. M. Bell, Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models, Empirical Software Engineering 13 (5) (2008) 539–559.

[24] L. Madeyski, M. Jureczko, Which process metrics can significantly improve defect prediction models? an empirical study, Software Quality Journal 23 (3) (2015) 393–422.

[25] Y. Shin, L. Williams, Can traditional fault prediction models be used for vulnerability prediction?, Empirical Software Engineering 18 (1) (2013) 25–59.

[26] Y. Gil, G. Lalouche, On the correlation between size and metric validity, Empirical Software Engineering 22 (5) (2017) 2585–2611.

[27] M. Shepperd, A critique of cyclomatic complexity as a software metric, Software Engineering Journal 3 (2) (1988) 30–36.

[28] J. Y. Gil, G. Lalouche, When do software complexity metrics mean nothing?-when examined out of context., J. Object Technol. 15 (1) (2016) 2–1.

[29] H. Zhang, An investigation of the relationships between lines of code and defects, in: International Conference on Software Maintenance, 2009, pp. 274–283.

[30] N. E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, Transactions on Software Engineering (TSE) 26 (8) (2000) 797–814.

[31] R. Shatnawi, W. Li, The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process, Journal of systems and software 81 (11) (2008) 1868–1882.

[32] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, Transactions on Software Engineering (TSE) 31 (4) (2005) 340–355.

[33] A. Schröter, T. Zimmermann, A. Zeller, Predicting component failures at design time, in: International symposium on Empirical software engineering, 2006, pp. 18–27.

[34] S. S. Rathore, S. Kumar, Predicting number of faults in software system using genetic programming, Procedia Computer Science 62 (2015) 303–311.

[35] L. Pascarella, F. Palomba, A. Bacchelli, Re-evaluating method-level bug prediction, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 592–601.

[36] F. Zhang, A. E. Hassan, S. McIntosh, Y. Zou, The use of summation to aggregate software metrics hinders the performance of defect prediction models, Transactions on Software Engineering (TSE) 43 (5) (2016) 476–491.

[37] J. Johnson, S. Lubo, N. Yedla, J. Aponte, B. Sharif, An empirical study assessing source code readability in comprehension, in: International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 513–523.

[38] D. Landman, A. Serebrenik, J. Vinju, Empirical analysis of the relationship between cc and sloc in a large corpus of java methods, in: International Conference on Software Maintenance and Evolution, 2014, pp. 221–230.

[39] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, R. Oliveto, Automatically assessing code understandability: How far are we?, in: International Conference on Automated Software Engineering (ASE), 2017, pp. 417–427.

[40] E. Giger, M. D'Ambros, M. Pinzger, H. C. Gall, Method-level bug prediction, in: International Symposium on Empirical Software Engineering and Measurement, 2012, pp. 171–180.

[41] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, R. Holmes, Codeshovel: Constructing method-level source code histories, in: International Conference on Software Engineering (ICSE), 2021, pp. 1510–1522.

[42] L. Pascarella, F. Palomba, A. Bacchelli, On the performance of method-level bug prediction: A negative result, Journal of Systems and Software 161.

[43] T. Shippey, T. Hall, S. Counsell, D. Bowes, So you need more method level datasets for your software defect prediction? voilà!, ESEM '16, 2016.

[44] S. Chowdhury, R. Holmes, A. Zaidman, R. Kazman, Revisiting the debate: Are code metrics useful for measuring maintenance effort?, Empirical Software Engineering 27 (6) (2022) 1–31.

[45] R. Mo, S. Wei, Q. Feng, Z. Li, An exploratory study of bug prediction at the method level, Inf. Softw. Technol. 144 (C).

[46] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical software engineering : an international journal 17 (3) (2012) 243–275.

[47] R. M. Bell, T. J. Ostrand, E. J. Weyuker, Does measuring code change improve fault prediction?, in: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Promise '11, 2011.

[48] R. Moser, W. Pedrycz, G. Succi, 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, ESEM '08, 2008, p. 309–311.

[49] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, C. Catal, Empirical analysis of change metrics for software fault prediction, Computers & Electrical Engineering 67 (2018) 15–24.

[50] A. G. Koru, D. Zhang, K. El Emam, H. Liu, An investigation into the functional form of the size-defect relationship for software modules, Transactions on Software Engineering (TSE) 35 (2) (2008) 293–304.

[51] F. Zhang, A. E. Hassan, S. McIntosh, Y. Zou, The use of summation to aggregate software metrics hinders the performance of defect prediction models, Transactions on Software Engineering (TSE) 43 (5) (2017) 476–491.

[52] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: Third International Workshop on Predictor Models in Software Engineering, 2007, pp. 9–9.

[53] D. Landman, A. Serebrenik, E. Bouwers, J. J. Vinju, Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions, Journal of Software: Evolution and Process 28 (7) (2016) 589–618.

[54] R. K. Bandi, V. K. Vaishnavi, D. E. Turk, Predicting maintenance performance using object-oriented design complexity metrics, Transactions on Software Engineering (TSE) 29 (1) (2003) 77–87.

[55] V. Antinyan, M. Staron, W. Meding, P. Österström, E. Wikstrom, J. Wranker, A. Henriksson, J. Hansson, Identifying risky areas of software code in agile/lean software development: An industrial experience report, in: Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 154–163.

[56] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: International conference on Software engineering, 2008, pp. 181–190.

[57] B. Caglayan, A. Bener, S. Koch, Merits of using repository metrics in defect prediction for open source projects, in: Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development, 2009, pp. 31–36.

[58] M. Alshayeb, W. Li, An empirical validation of object-oriented metrics in two different iterative software processes, Transactions on Software Engineering (TSE) 29 (11) (2003) 1043–1049.

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

[60] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, T. Dybå, Quantifying the effect of code smells on maintenance effort, Transactions on Software Engineering (TSE) 39 (8) (2012) 1144–1156.

[61] S. A. Chowdhury, G. Uddin, R. Holmes, An empirical study on maintainable method size in java, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 252–264.

[62] H. Alsolai, M. Roper, D. Nassar, Predicting software maintainability in object-oriented systems using ensemble techniques, in: 2018 IEEE International Conference on Software Maintenance and Evolution, 2018, pp. 716–721.

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

[64] C. Pornprasit, C. K. Tantithamthavorn, Deeplinedp: Towards a deep learning approach for line-level defect prediction, IEEE Transactions on Software Engineering 49 (1) (2023) 84–98.

[65] E. Shihab, A. E. Hassan, B. Adams, Z. M. Jiang, An industrial study on the risk of software changes, in: International Symposium on the Foundations of Software Engineering, 2012, pp. 1–11.

[66] H. Zhang, H. B. K. Tan, An empirical study of class sizes for large java systems, in: 14th Asia-Pacific Software Engineering Conference (APSEC'07), IEEE, 2007, pp. 230–237.

[67] F. Servant, J. A. Jones, Fuzzy fine-grained code-history analysis, in: International Conference on Software Engineering (ICSE), 2017, pp. 746–757.

[68] D. Steidl, B. Hummel, E. Juergens, Incremental origin analysis of source code files, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 42–51.

[69] S. Chowdhury, G. Uddin, H. Hemmati, R. Holmes, Method-level bug prediction: Problems and promises, ACM Trans. Softw. Eng. Methodol.Just Accepted.

[70] D. Steidl, F. Deissenboeck, How do java methods grow?, in: International Working Conference on Source Code Analysis and Manipulation (SCAM), 2015, pp. 151–160.

[71] D. Romano, M. Pinzger, Using source code metrics to predict change-prone java interfaces, in: 2011 27th IEEE international conference on software maintenance (ICSM), 2011, pp. 303–312.

[72] R. Abbas, F. A. Albalooshi, M. Hammad, Software change proneness prediction using machine learning, in: 2020 international conference on innovation and intelligence for informatics, computing and technologies (3ICT), 2020, pp. 1–7.

[73] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 482–482.

[74] Y. Zhou, B. Xu, H. Leung, L. Chen, An in-depth study of the potentially confounding effect of class size in fault prediction, Transactions on Software Engineering and Methodology 23 (1) (2014) 1–51.

[75] S. M. Olbrich, D. S. Cruzes, D. I. Sjøberg, 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, IEEE, 2010, pp. 1–10.

[76] K. Yamashita, C. Huang, M. Nagappan, Y. Kamei, A. Mockus, A. E. Hassan, N. Ubayashi, Thresholds for size and complexity metrics: A case study from the perspective of defect density, in: 2016 IEEE

international conference on software quality, reliability and security (QRS), 2016, pp. 191–201.

[77] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, D. Cok, Local vs. global models for effort estimation and defect prediction, in: International Conference on Automated Software Engineering (ASE), 2011, pp. 343–351.

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

[79] S. Banitaan, K. Daimi, Y. Wang, M. Akour, Test case selection using software complexity and volume metrics, in: Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE)., 2015, pp. 12–14.

[80] H. Cervantes, R. Kazman, Software archinaut: a tool to understand architecture, identify technical debt hotspots and manage evolution, in: Proceedings of the 3rd International Conference on Technical Debt, 2020, pp. 115–119.

[81] J. Lefever, Y. Cai, H. Cervantes, R. Kazman, H. Fang, On the lack of consensus among technical debt detection tools, in: International Conference on Software Engineering: Software Engineering in Practice, 2021, pp. 121–130.

[82] K. Herzig, A. Zeller, The impact of tangled code changes, in: Working Conference on Mining Software Repositories (MSR), 2013, pp. 121–130.

[83] D. Matter, A. Kuhn, O. Nierstrasz, Assigning bug reports using a vocabulary-based expertise model of developers, in: International working conference on mining software repositories, 2009, pp. 131–140.

[84] I. Scholtes, P. Mavrodiev, F. Schweitzer, From aristotle to ringelmann: a large-scale analysis of team productivity and coordination in open source software projects, Empirical software engineering : an international journal 21 (2) (2016) 642–683.

[85] Y. Shin, A. Meneely, L. Williams, J. A. Osborne, Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, Transactions on Software Engineering (TSE) 37 (6) (2010) 772–787.

[86] D. Ståhl, A. Martini, T. Mårtensson, Big bangs and small pops: On critical cyclomatic complexity and developer integration behavior, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: (ICSE-SEIP), 2019, pp. 81–90.

[87] D. Kawrykow, M. P. Robillard, Non-essential changes in version histories, in: International Conference on Software Engineering (ICSE), 2011, pp. 351–360.

[88] B. Ray, M. Nagappan, C. Bird, N. Nagappan, T. Zimmermann, The uniqueness of changes: Characteristics and applications, in: Working Conference on Mining Software Repositories, 2015, pp. 34–44.

[89] Y. Zhou, B. Xu, H. Leung, 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.

[90] D. Kafura, G. R. Reddy, The use of software complexity metrics in software maintenance, Transactions on Software Engineering (TSE) (3) (1987) 335–343.

[91] D. Radjenović, M. Heričko, R. Torkar, A. Živkovič, Software fault prediction metrics: A systematic literature review, Information and Software Technology 55 (8) (2013) 1397 – 1418.

[92] O. Dabic, E. Aghajani, G. Bavota, Sampling projects in GitHub for MSR studies, in: International Conference on Mining Software Repositories (MSR), 2021, p. To appear.
URL https://arxiv.org/abs/2103.04682

[93] H. Borges, A. Hora, M. T. Valente, Understanding the factors that impact the popularity of GitHub repositories, in: International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 334–344.

[94] B. Ray, D. Posnett, V. Filkov, P. Devanbu, A large scale study of programming languages and code quality in github, in: International Symposium on Foundations of Software Engineering, 2014, pp. 155–165.

[95] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, J. Vitek, On the impact of programming languages on code quality: a reproduction study, Transactions on Programming Languages and Systems (TOPLAS) 41 (4) (2019) 1–24.

[96] I. Etikan, S. A. Musa, R. S. Alkassim, Comparison of convenience sampling and purposive sampling, American journal of theoretical and applied statistics 5 (1) (2016) 1–4.

[97] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, The promises and perils of mining github, in: Proceedings of the 11th working conference on mining software repositories, 2014, pp. 92–101.

[98] A. Zaidman, B. Van Rompaey, S. Demeyer, A. Van Deursen, Mining software repositories to study co-evolution of production & test code, in: International Conference on Software Testing, Verification, and Validation (ICST), 2008, pp. 220–229.

[99] L. S. Pinto, S. Sinha, A. Orso, Understanding myths and realities of test-suite evolution, in: International Symposium on the Foundations of Software Engineering (ESEC/FSE), 2012, pp. 1–11.

[100] V. Antinyan, M. Staron, J. Derehag, M. Runsten, E. Wikström, W. Meding, A. Henriksson, J. Hansson, Identifying complex functions: By investigating various aspects of code complexity, in: Science and Information Conference (SAI), 2015, pp. 879–888.

[101] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, P. Devanbu, On the" naturalness" of buggy code, in: International Conference on Software Engineering (ICSE), 2016, pp. 428–439.

[102] A. Mockus, L. G. Votta, Identifying reasons for software changes using historic databases., in: icsm, 2000, pp. 120–130.

[103] T. Zimmermann, N. Nagappan, Predicting defects using network analysis on dependency graphs, in: International conference on Software engineering, 2008, pp. 531–540.

[104] J. Jiarpakdee, C. Tantithamthavorn, A. E. Hassan, The impact of correlated metrics on defect models, arXiv preprint arXiv:1801.10271.

[105] D. J. Sheskin, Handbook of parametric and nonparametric statistical procedures, Chapman and Hall/CRC, 2003.

[106] The anderson-darling test for normality, Journal of quality technology 30 (3) (1998) 298–299.

[107] L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness, in: International conference on software engineering, 2014, pp. 435–445.

[108] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys, in: Annual meeting of the Florida Association of Institutional Research, Vol. 13, 2006.

[109] J. Cohen, Statistical power analysis for the behavioral sciences, Academic press, 2013.

[110] M. Hess, J. Kromrey, Robust confidence intervals for effect sizes: A comparative study of cohen's d and cliff's delta under non-normality and heterogeneous variances, 2004.

[111] A. Agrawal, T. Menzies, Is" better data" better than" better data miners"?, in: International Conference on Software Engineering (ICSE), 2018, pp. 1050–1061.

[112] D. P. Doane, L. E. Seward, Measuring skewness: a forgotten statistic?, Journal of statistics education 19 (2).

[113] G. Robles, I. Herraiz, D. M. Germán, D. Izquierdo-Cortázar, Modification and developer metrics at the function level: Metrics for the study of the evolution of a software project, in: International Workshop on Emerging Trends in Software Metrics (WETSoM), 2012, pp. 49–55.

[114] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, in: International Conference on Automated Software Engineering, 2016, pp. 4–15.

[115] J. Rosenberg, Some misconceptions about lines of code, in: Proceedings international software metrics symposium, 1997, pp. 137–142.

[116] C. Andersson, P. Runeson, A replicated quantitative analysis of fault distributions in complex software systems, Transactions on Software Engineering (TSE) 33 (5) (2007) 273–286.

[117] M. A. Kabir, J. W. Keung, K. E. Bennin, M. Zhang, Assessing the significant impact of concept drift in software defect prediction, in: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Vol. 1, 2019, pp. 53–58.
[118] S. Wang, J. Wang, J. Nam, N. Nagappan, Continuous software bug prediction, 2021.
[119] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi, et al., A fine-grained data set and analysis of tangling in bug fixing commits, Empirical Software Engineering 27 (6).
[120] C. Gini, Measurement of inequality of incomes, The Economic Journal 31 (121) (1921) 124–126.
[121] A. B. Atkinson, et al., On the measurement of inequality, Journal of economic theory 2 (3) (1970) 244–263.
[122] C. E. Shannon, A mathematical theory of communication, Mobile computing and communications review 5 (1) (2001) 3–55.
[123] E. M. Hoover, The measurement of industrial localization, The Review of Economic Statistics (1936) 162–171.
[124] B. Vasilescu, A. Serebrenik, M. Van den Brand, By no means: A study on aggregating software metrics, in: Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics, 2011, pp. 23–26.
[125] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, X. Yang, Perceptions, expectations, and challenges in defect prediction, Transactions on Software Engineering (TSE) 46 (11) (2018) 1241–1266.
[126] M. Kondo, D. M. German, O. Mizuno, E.-H. Choi, The impact of context metrics on just-in-time defect prediction, Empirical Software Engineering 25 (1) (2020) 890–939.
[127] M. Greiler, A. Zaidman, A. Van Deursen, M.-A. Storey, Strategies for avoiding text fixture smells during software evolution, in: Working Conference on Mining Software Repositories (MSR), 2013, pp. 387–396.
[128] A. G. Koru, H. Liu, Identifying and characterizing change-prone classes in two large-scale open-source products, Journal of Systems and Software 80 (1) (2007) 63–73.
[129] S. Wang, T. Liu, J. Nam, L. Tan, Deep semantic feature learning for software defect prediction, Transactions on Software Engineering (TSE) 46 (12) (2018) 1267–1293.
[130] H. Osman, A. Chis, C. Corrodi, M. Ghafari, O. Nierstrasz, Exception evolution in long-lived Java systems, in: Proceedings of the International Conference on Mining Software Repositories, 2017, pp. 302–311.
[131] S. McIntosh, Y. Kamei, Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction, Transactions on Software Engineering (TSE) 44 (5) (2017) 412–428.
[132] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, T. Zimmermann, Local versus global lessons for defect prediction and effort estimation, Transactions on Software Engineering (TSE) 39 (6) (2012) 822–834.
[133] C. Faragó, P. Hegedűs, R. Ferenc, Cumulative code churn: Impact on maintainability, in: International Working Conference on Source Code Analysis and Manipulation (SCAM), 2015, pp. 141–150.
[134] S. N.C., S. Majumder, T. Menzies, Early life cycle software defect prediction. why? how?, in: International Conference on Software Engineering (ICSE), 2021, pp. 448–459.
[135] F. Zhang, A. Mockus, Y. Zou, F. Khomh, A. E. Hassan, How does context affect the distribution of software maintainability metrics?, in: International Conference on Software Maintenance, 2013, pp. 350–359.