Is Lines of Code a Good Measure of Effort in Effort-Aware Models?

Emad Shihab[*], Yasutaka Kamei[+], Bram Adams[#] and Ahmed E. Hassan[@]

Department of Software Engineering[*], Rochester Institute of Technology, Rochester, NY, USA, emad.shihab@rit.edu

Graduate School and Faculty of Information Science and Electrical Engineering[+], Kyushu University, Fukuoka, Japan, kamei@ait.kyushu-u.ac.jp

Lab on Maintenance, Construction and Intelligence of Software (MCIS)[#], École Polytechnique de Montréal, Canada, bram.adams@polymtl.ca

Software Analysis and Intelligence Lab (SAIL)[@], Queen's University, Kingston, ON, Canada, ahmed@cs.queensu.ca

Abstract

Context: Effort-aware models, e.g., effort-aware bug prediction models aim to help practitioners identify and prioritize buggy software locations according to the effort involved with fixing the bugs. Since the effort of current bugs is not yet known and the effort of past bugs is typically not explicitly recorded, effort-aware bug prediction models are forced to use approximations, such as the number of lines of code (LOC) of the predicted files.

Objective: Although the choice of these approximations is critical for the performance of the prediction models, there is no empirical evidence on whether LOC is actually a good approximation. Therefore, in this paper, we investigate the question: *is LOC a good measure of effort for use in effort-aware models?*

Method: We perform an empirical study on four open source projects, for which we obtain explicitly-recorded effort data, and compare the use of LOC to various complexity, size and churn metrics as measures of effort.

Results: We find that using a combination of complexity, size and churn metrics are a better measure of effort than using LOC alone. Furthermore, we examine the impact of our findings on previous effort-aware bug prediction work and find that using LOC as a measure for effort does not significantly affect the list of files being flagged, however, using LOC under-estimates the amount of effort required compared to our best effort predictor by approximately 66%.

Conclusion: Studies using effort-aware models should not assume that LOC is a good measure of effort. For the case of effort-aware bug prediction, using LOC provides results that are similar to combining complexity, churn, size and LOC as a proxy for effort when prioritizing the most risky files. However, we find that for the purpose of effort-estimation, using LOC may under-estimate the amount of effort required.

Keywords: Effort-aware Prediction; Prediction Models; Defect Prediction

---

1. Introduction

A large amount of research work has focused on predicting where bugs may occur in software systems [37]. The main goal of this line of work is to prioritize quality assurance efforts to make effective use of quality assurance resources.

However, the adoption of software bug prediction research in practice remains low [16, 13]. One of the main reasons for this low adoption is the fact that, for a long time, these bug prediction models did not take into consideration the effort needed to act on the recommended artifact [16]. In this context, effort means the amount of effort required to address (i.e., unit test or code review) the recommended artifact (i.e., file or folder). This notion of effort is different from the work on the next-release problem (i.e., given a set of parameters for a project how much effort is needed for the next release) [1, 12] and from the work that focused at predicting the amount of time required to fix a bug (e.g., [36]).

To incorporate effort into bug prediction, recent work (e.g., [22, 3, 23, 19]) studied the performance of effort-aware bug prediction models. These models typically use lines of code (LOC) as a de facto measure of the effort required to address bugs, since explicit effort is rarely recorded by development projects. The authors showed that when effort is considered in the prediction models, the usefulness of some prediction models (e.g., models that use size to prioritize buggy locations) may be affected. However, they point out that perhaps there might exist better measures of effort than LOC [22].

To the best of our knowledge, previous work did not address the important question of what is the best way to measure effort in effort-aware models? Yet, finding a good estimator of effort is critical to the accuracy, and hence success of, effort-aware models in practice.

In this paper, we perform an empirical study on four Open Source Software (OSS) projects for which we obtain explicit effort data – namely JBoss, Spring,

Hibernate and Mule and compare the use of LOC to other code and process metrics as measures of effort. First, we perform correlation analysis to examine the relationship between the various code and process metrics and effort. Then, we examine which of process metrics, code metrics or the commonly used LOC measure best predict effort. Lastly, we examine the impact of our findings on prior research findings related to effort-aware bug prediction. We aim to answer the following research questions:

RQ1 Does LOC correlate better with effort than other code and process metrics?

We find that LOC has a low correlation with effort in all four studied projects. Code metrics, in particular complexity metrics, have the highest correlation with effort. However, the overall correlation values for both LOC and complexity metrics remain relatively low.

RQ2 Is LOC a better predictor of effort than other code and process metrics?

We find that code metrics, in particular complexity metrics, are better predictors of effort than LOC in all four studied projects. Moreover, combining LOC, complexity, churn and size metrics yields the best prediction results.

RQ3 How do our findings impact previous effort-aware bug prediction findings?

Revisiting prior effort-aware bug prediction work shows that for the purpose of ranking the most buggy files, using LOC as a measure of effort performs similar to our best effort predictor. However, for the purpose of effort-estimation, using LOC under-estimates the amount of effort required compared to our best effort predictor.

To facilitate future studies, we plan on making our explicit effort data set available to the software maintenance research community[1].

The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 describes our methodology. Section 4 presents an exploratory analysis of the used effort data. Section 5 presents the findings of our study. Section 6 discusses our findings and their implications. Section 7 lists the threats to validity and Section 8 concludes the paper.

[1]We are currently in the process of adding our data set to the PROMISE repository.

2. Related Work

In this section, we highlight the previous work and contrast with our work. The majority of the related work comes from the areas of software bug prediction and effort estimation.

2.1. Bug Prediction

Standard bug prediction models. A large number of prior work uses code metrics to predict defective or fault-prone modules (e.g., files). The majority of this prior work shows that McCabe's cyclomatic complexity metric [21] and the Chidamber and Kemerer (CK) metrics suite [9] can be used to successfully predict pre- and post-release bugs with high accuracy on both, commercial and open source systems [29, 5, 35, 15, 11, 8, 28, 37].

Other work uses process metrics, such as the number of prior bugs or prior changes to predict fault-prone modules. This work shows that the number of prior changes to a file is a better predictor of bugs than code metrics in general [14, 2, 20]. More recently, Moser *et al.* [25] show that process metrics in general perform better than code metrics when predicting post-release bugs in Eclipse. Hassan [17] shows that using the complexity of code changes is a good predictor of bugs.

Although it has been shown that standard bug prediction can be beneficial to improve the quality of software systems, its adoption remains low [13, 16]. Therefore, a number of recent studies have argued that effort needs to be considered to improve the adoption of standard bug prediction research.

Effort-aware bug prediction models. Recent work examined the performance of software prediction models when effort is considered (i.e., effort-aware bug prediction) [22, 19, 3, 24]. In all of these prior studies, LOC was used as the de facto measure of effort. The authors showed that considering effort yields different results from standard bug prediction (i.e., the performance of models built using size metrics becomes comparable to that of a random predictor).

Our study picks up from where this work left off by examining the important question: what is a good measure of effort in effort-aware models? Although LOC was used as a de facto measure, we would like to examine whether there exist other metrics that are better predictors of effort to use in effort-aware models, as hinted at by previous studies [22].

2.2. Effort Estimation

There has been a healthy number of studies on effort estimation as well. Most of these studies examine the next-release problem: given a set of project param-

eters (e.g., the number of available developers) how much effort will my next release require? Other work attempts to estimate the effort required to fix a bug.

Estimating Effort of Next Release. A number of studies focuses on estimating the effort required for a future release in order to assist in project scheduling or external contract bidding [7, 31, 4, 23].

These studies use metrics based on the features of past project releases (i.e., functional size measures, manager's skill level and software development environment) and project requirements (i.e., how many developers and testers are available to work on the next release). For example, Shepperd and Schofield[31] propose the Estimation by Analogy (EbA) technique that derives the effort from one or more past analogous projects that are similar to the current project. They show that EbA outperforms other estimation methods that use linear regression.

Estimating the Time to Fix a Bug. Other studies related to effort estimation focus on estimating the time it takes to fix a bug [1, 12, 18, 34, 36]. Song *et al.* [34] uses association rule mining to estimate the effort required to fix a bug. They use bug type data (e.g., whether the bug concerns an initialization error or an interface error) to predict the effort required to fix the bug. Weiß *et al.* [36] use the text in the title and description fields of an issue report to predict the effort required to address the issue. Giger *et al.* [12] and Hewett and Kijsanayothin [18] estimate the time to fix a bug based on metrics extracted from the bug report itself (e.g., the reporter's name and the bug severity and priority). More recently, Ahsan *et al.* [1] analyze the correlation between the time it takes to address an issue and source code metrics (e.g., LOC and Cyclomatic complexity). They find low correlation values for LOC and Cyclomatic complexity for the Mozilla project.

There are some key differences between the prior work on effort estimation and our work. First, our goal is *not to estimate the time it takes to fix a bug*, rather our goal is to *determine what is a good measure of the effort for use in effort-aware models*. Second, we map and perform our effort analysis to the file level, not the issue or bug level. We map effort to the file level in order to use and investigate the impact of our findings on previous bug prediction work (which is mostly performed at the file level), which we examine in more detail in Section 5.

Third, one very important strength of our work is the used data set. The majority of the previous studies use the time between when a bug was opening until its closure as a measure of effort. These effort values are noisy since they may contain the off-time, where a developer is busy on other tasks or interrupted by other unrelated issues. In contrast, our effort data is input by the actual *practitioner who made the change* and considers the effort spent only on addressing the specific issue. Weiß *et al.* [36] used explicit effort data but they only consider one

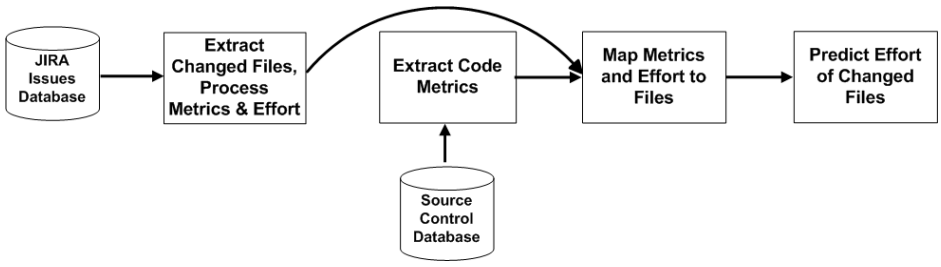


Figure 1: Approach Overview

Table 1: Number of Issues in Each Project

	Hibernate	**JBoss**	**Mule**	**Spring**
# issues before 2011-01-01	10,666	92,179	6,543	23,504
∩ **Non-empty effort data**	242	2,220	125	1,852
∩ **Type: Bug, Defect, New Feature, Feature Request, Task, Sub-task, Improvement, Patch**	241	2,178	125	1,833
∩ **Status: Closed, Resolved**	231	2,131	122	1,713
∩ **Resolution: Done, Fixed, Complete**	225	1,979	113	1,496
∩ **Priority: not Trivial**	218	1,966	113	1,025
∩ **Total issues that modify source code**	**171**	**506**	**40**	**297**
Total number of files	**1,147**	**1,421**	**286**	**1,409**

of the projects we consider, and in contrast, predict the time to fix a bug.

3. Methodology

Figure 1 provides an overview of our approach, which we detail in this section. In a nutshell, we first extract issues that have effort records from the JIRA issues database. These issues are parsed and the changed files, their churn (i.e., lines added and deleted) and the effort associated with the issues is recorded. Then, we extract the source code of the changed files for each issue and analyze the source code to extract various code (i.e., complexity and size) and process (i.e., churn) metrics. We map all of the metrics to the file level and use this information to examine the correlations with, and predict effort.

Table 2: Summary of Metrics

Dim.	ID	Name	Definition	Level
	LOC	Total Lines of Code	The total number of lines.	File
Complexity	CDEN	Cyclomatic Complexity Density	The density of cyclomatic complexity (VG_sum/LOC).	File
	CODEN	Comment Density	The density of comments (TCOLOC/LOC).	
	CC	Cyclomatic Complexity	The cyclomatic complexity for all nested functions or methods. Three measures are provided for CC: avg, max and total.	
	CCM	Cyclomatic Complexity Modified	The modified cyclomatic complexity for all nested functions or methods. The CCM expects that a multi-decision structure statement (e.g., switch) is counted as 1. Three measures are provided for CCM: avg, max and total.	
	CCS	Cyclomatic Complexity Strict	The strict cyclomatic complexity for all nested functions or methods. The CCS expects that logical conjunction and conditional expressions add 1 to the complexity for each of their occurrences. Three measures are provided for CCS: avg, max and total.	
	CCE	Cyclomatic Complexity Essential	The essential cyclomatic complexity for all nested functions or methods. The CCE iteratively replaces all structured programming primitives to a single statement, then calculates CC. Three measures are provided for CCE: avg, max and total.	
	CBO	Coupling Between Objects	The number of other classes to which this class is coupled. Three measures are provided for CBO: avg, max and total.	Class
	NOC	Number Of Children	The number of immediate subclasses. Three measures are provided for NOC: avg, max and total.	
	RFC	Response For a Class	The number of methods, including inherited ones. Three measures are provided for RFC: avg, max and total.	
	DIT	Depth of Inheritance Tree	The depth of a class within the inheritance hierarchy. Three measures are provided for DIT: avg, max and total.	
	LCOM	Lack of Cohesion in Methods	100% - average cohesion for class data members. Three measures are provided for LCOM: avg, max and total.	
	WMC	Weighted Methods per Class	The number of weighted methods per class. Three measures are provided for WMC: avg, max and total.	
	FOUT	Fan out	The number of method calls. Three measures are provided for FOUT: avg, max and total.	Method
	NBD	Nested Block Depth	The nested block depth of the methods. Three measures are provided for NBD: avg, max and total.	
	PAR	Number of Parameters	The number of parameters of the methods. Three measures are provided for PAR: avg, max and total.	
	VG	McCabe Cyclomatic Complexity	The McCabe cyclomatic complexity of the methods in a file. Three measures are provided for VG: avg, max and total.	
Size	TBLOC	Total Blank Lines of Code	The number of blank lines.	File
	TCLOC	Total Code Lines of Code	The number of lines that contain source code.	
	TCOLOC	Total Comment Lines of Code	The number of lines that contain comments.	
	TNOS	Total Number of Semicolons	The number of semicolons.	
	TNOST	Total Number of Statements	The number of declarative or executable statements.	
	TNODST	Total Number of Declarative Statements	The number of declarative statements.	
	TNOEST	Total Number of Executable Statements	The number of executable statements.	
	TNOT	Total Number of Classes	The number of classes.	
	TNOM	Total Number of Methods	The number of methods.	
	NOF	Number of Fields	The number of fields of the classes. Three measures are provided for NOF: avg, max and total.	Class
	NOM	Number of Methods	The number of methods of the classes. Three measures are provided for NOM: avg, max and total.	
	NSF	Number of Static Fields	The number of static fields of the classes. Three measures are provided for NSF: avg, max and total.	
	NSM	Number of Static Methods	The number of static methods of the classes. Three measures are provided for NSM: avg, max and total.	
	MLOC	Method Lines of Code	The number of method lines of code. Three measures are provided for MLOC: avg, max and total.	Method
Churn	CHURN	Total Churn	The sum of the number of added lines of code and the number of deleted lines of code.	File
	ADD	Total Added Lines	The number of lines of code added to a file.	
	DEL	Total Deleted Lines	The number of lines of code deleted in a file.	

3.1. JIRA Issues Database

JIRA is a popular issue tracking system. In addition to having the standard features of an issue tracking system (e.g., recording bug descriptions, states and open/closure dates), JIRA has a very unique feature that enables users to input the value of effort that an issue takes to address.

The first step of our work was to extract these issues from the JIRA databases of four Java projects: Hibernate, JBoss, Mule and Spring. The main reason for choosing to use these projects is the fact that they used JIRA as their issue tracking system and they recorded effort for their issues.

3.2. Extract Issues

Similar to previous studies [36], we focused on issues that had a non-empty effort field, had a type field as Bug, Defect, New Feature, Feature Request, Task, Sub-task, Improvement or Patch, had a status of closed or resolved, had resolution as Done, Fixed or Complete and had a priority that was non-trivial. In addition, we ignored issues that did not modify any source code files. For issues that modify code files and other types of files (e.g., XML files), we mapped their effort to the source code files only, since bug prediction work focuses on source code files.

Table 1 shows the number of used issues and files from each project in our study. We note that only a small number of issues contains effort information and an even smaller number meets all of our criteria to be considered in the study.

At first glance, the number of issues in our data set seems to be low. However, our data set contains approximately double the amount of issues compared to the study by Weiß *et al.* [36] (which uses 567 issues). To the best of our knowledge, that study is considered the state of the art study for this kind of data. In addition, our data set covers four different projects, whereas Weiß *et al.* leveraged data from only one project.

3.3. Extract Changed Files, Process Metrics and Effort

After downloading all of the issues in html format, we wrote scripts that parsed all of the downloaded issues to determine the following: 1) the names of those files that were changed to address the issue, 2) the process metrics (i.e., the number of churned lines) for each file and 3) the amount of effort required to address the issue.

The effort value extracted from the issues is logged by the person who addressed the issue. The effort value is measured in time and includes the time needed to perform all activities that the practitioner needed to complete the assigned issue. It includes the time needed to identify which files need to be changed,

Table 3: Change Level Effort (measured in hours)

Project	Mean	SD	Min	Max	Skew	Kurtosis
Hibernate	4.84	8.85	0.02	78.00	4.46	27.99
JBoss	12.71	21.95	0.00	280.00	5.31	47.78
Mule	4.74	4.44	0.00	18.00	1.21	0.50
Spring	4.42	6.97	0.08	44.00	3.10	10.86

the time it took to inspect these files and decide where to make the changes and the time it took to make the actual changes. Having explicit effort data makes using this effort value ideal for our study, especially since this data is free of noise (i.e., it does not include things such as triage time and testing delays).

At the end of this step, we have a linkage between the effort an issue required, files changed to address the issue and the amount of churn done to these files.

3.4. Extract Source Code and Code Metrics

To obtain code metrics (e.g., code complexity metrics), we require the source code of the changed files. We use the list of file names and corresponding revision numbers in each issue to download the corresponding revisions of files from the source control database.

We use the `Understand 2.0` [30] tool to analyze each file and extract their code metrics (i.e., complexity and size). Table 2 lists all of the metrics used in our study. A total of 47 complexity metrics and 26 size metrics were extracted for each files. In addition, we annotate this metric set with the churn metrics for each file. In total, we ended up with an extensive set of 76 code and process metrics for each file.

We use the code and process metrics in our study because they are widely available for most software projects and their relationship to quality has been studied extensively in the past (e.g., [26, 37]). Furthermore, there is an intuitive relationship between code and process metrics and effort (e.g., the more complex a piece of code is, the more effort it requires to be addressed).

3.5. Map Metrics and Effort to Files

The effort in the issue tracking system is recorded at the issue level. As mentioned earlier, we map effort to the file level, not the issue level in order to study the relationship between the various code and process metrics and to compare our findings to previous bug prediction work, which are performed at the file level.

Table 4: File Level Effort (measured in hours)

Project	Mean	SD	Min	Max	Skew	Kurtosis
Hibernate	14.07	18.20	0.03	78.00	2.40	5.72
JBoss	26.28	35.70	0.03	280.00	3.93	23.32
Mule	7.23	4.75	0.00	18.00	0.23	-1.24
Spring	15.19	12.31	0.17	44.00	0.64	-0.65

Since issues may touch one or more files, we need to map the issue effort to the file level.

Mapping Effort to Files. We can employ a number of strategies when transforming the effort from the issue to the file level. For example, we can take the average effort of all the issues that touch a file and assign it to the file. However, we decided to assign the effort value of a file to be the maximum effort required to change it in all of the issues that touch the file. There are two main reasons for this choice. First, assigning the maximum effort of all the issues that touch a file is considered to be the worst-case scenario (i.e., the highest possible effort) for that file. Therefore, using the maximum value is seen as a conservative approximation. Second, we experimented with mapping the average, median and minimum effort of all the issues to the file level. However, using maximum effort yielded the best and most meaningful correlations with effort. For example, in Table 5 we show the correlations of the different metrics with *average* effort, whereas in in Table 6 we show the correlations of the different metrics with using *maximum* effort. Clearly, using maximum effort yields higher correlation values.

To illustrate how we perform the mapping, we use the following example. Suppose we have two issues `Issue1` and `Issue2` that took 3,600 and 1,800 seconds, respectively. `Issue1` requires files `fileA` and `fileB` be modified, while `Issue2` requires that `fileA` and `fileC` to be modified. Then we calculate the effort for `fileA` as the maximum of `Issue1` and `Issue2`, which is 3,600 seconds (`Issue1`), the effort for `fileB` is 3,600 seconds (`Issue1`) and the effort for `fileC` is 1,800 seconds (`Issue2`).

Mapping Metrics to Files. Similar to the issues, we also need to map the metrics to the file level, i.e., make each file one data point. Since some files may be touched by multiple issues (and therefore have multiple revisions), we assigned each file its average value of the metrics. For example, if the Cyclomatic Complexity (CC) of a `fileA` is 50 in `Issue1` and 100 in `Issue2`, then we would assign a CC value of ($\frac{50+100}{2} = 75$) to `fileA`.

Table 5: Metric Correlations with Effort (using average effort)

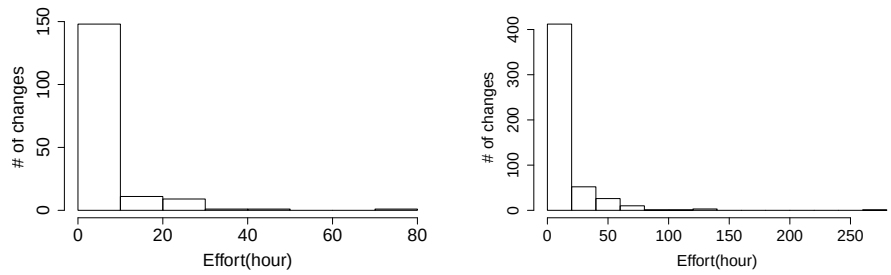
		Complexity			Size			Churn		
	LOC	**Metric**	**Corr.**	**Improv.**	**Metric**	**Corr.**	**Improv.**	**Metric**	**Corr.**	**Improv.**
Hibernate	-0.07 **	CCE_ave	0.05	(+169%)	TNOT	0.00	(+107%)	DEL	-0.07 **	(-4%)
JBoss	-0.06 **	CBO_ave	0.14 ***	(+354%)	TCOLOC	0.04	(+168%)	CHURN	0.14 ***	(+344%)
Mule	-0.08	NOC_max	0.07	(+186%)	NOF_max	0.02	(+124%)	ADD	0.08	(+200%)
Spring	-0.07 ***	DIT_max	0.10 ***	(+237%)	MLOC_ave	0.08 ***	(+209%)	DEL	0.15 ***	(+314%)

(***) $p < 0.01$, (**) $p < 0.05$, (*) $p < 0.1$

Table 6: Metric Correlations with Effort (using maximum effort)

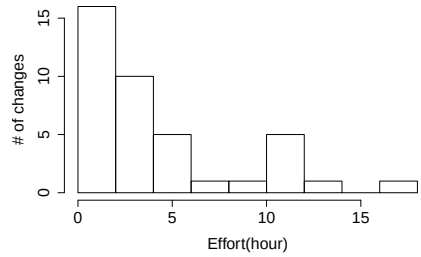
		Complexity			Size			Churn		
	LOC	**Metric**	**Corr.**	**Improv.**	**Metric**	**Corr.**	**Improv.**	**Metric**	**Corr.**	**Improv.**
Hibernate	0.04	CDEN	0.19 ***	(+395%)	MLOC_total	0.10 ***	(+154%)	DEL	-0.04	(-193%)
JBoss	0.02	CBO_ave	0.23 ***	(+1019%)	TCOLOC	0.11 ***	(+415%)	CHURN	0.16 ***	(+688%)
Mule	-0.03	DIT_max	0.16 ***	(+720%)	MLOC_total	0.07	(+356%)	ADD	0.05	(+290%)
Spring	0.06 **	DIT_max	0.33 ***	(+412%)	MLOC_ave	0.30 ***	(+358%)	DEL	0.30 ***	(+362%)

(***) $p < 0.01$, (**) $p < 0.05$, (*) $p < 0.1$

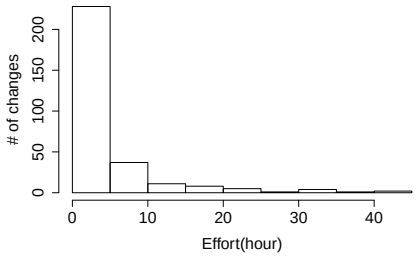


(a) Hibernate

(b) JBoss



(c) Mule



(d) Spring

Figure 2: Distribution of Effort Across Issues

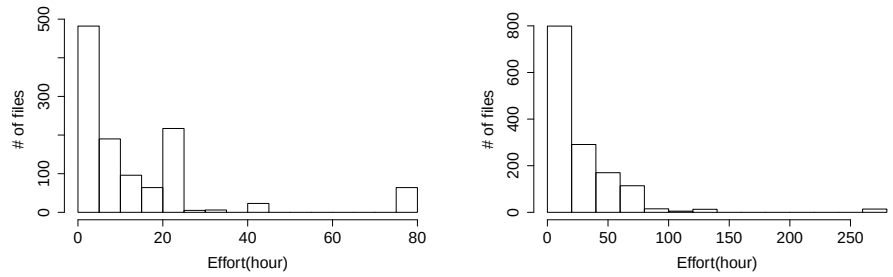
4. Data Exploration

To the best of our knowledge, this is one of the first (in addition to the study by Weiß *et al.* [36]) studies to leverage explicit effort data. Therefore, before delving into our case study results, we first explore our effort data in more detail.

We examine the distribution of the effort data at the issue and file levels. We report a few of the most common descriptive statistics, i.e., mean, min, max, standard deviation (SD) for all projects. To study the skewness of the data, we calculate the skew and kurtosis measures.

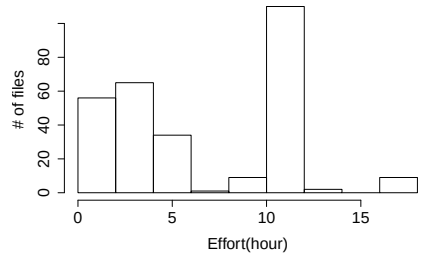
Skew measures the amount of asymmetry in the probability distribution of a variable, in relation to the normal distribution. Positive skew means that the distribution of the metric values are mostly on the low end of the scale, while negative skew indicates that most of the metric values are on the high end of the scale. The normal distribution has a skew value of 0.

Kurtosis on the other hand characterizes the relative peakedness or flatness of a distribution, in relation to the normal distribution. Positive kurtosis indicates

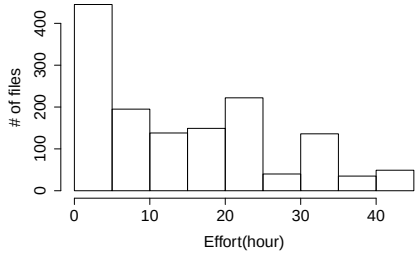


(a) Hibernate

(b) JBoss



(c) Mule



(d) Spring

Figure 3: Distribution of Effort Across Files

that a curve is too tall and negative kurtosis indicates a curve that is too flat. A normal distribution has a kurtosis value of 0.

Issue Level Exploration. Table 3 shows the descriptive statistics for the extracted issues of the Hibernate, JBoss, Mule and Spring projects. From the table, we observe that the average effort required to address an issue is between 4-5 hours for the Hibernate, Mule and Spring projects. The JBoss project, however, has a much higher average effort per issue, approximately 12 hours. One possible explanation for the high mean effort value is the fact that JBoss had the most number of issues and files, therefore, it had more outliers. In fact, the high kurtosis value for JBoss indicates that.

Additionally, the Skew and Kurtosis values for the Hibernate, JBoss and Spring projects are well over 0 (meaning the distribution of the data is far from normally distributed). The effort data for Mule is less skewed.

To get a better idea of the skewness in the distributions, we plot the effort data at the issue level in Figure 2. Clearly, we see that the majority of the issue ef-

fort values are on the lower end of the scale for the Hibernate, JBoss and Spring projects. Mule seems to have a small jump in the number of issues at approximately 12 hours. We examined the issues that had this value of effort and found that these issues had required large changes to many files (more than 10 files were modified to address these issues). Furthermore, we would like to note that since we did not have many issues for the Mule project, its data was very sensitive to anomalies.

File Level Exploration. We also explore the distribution of the data at the file level (Table 4). We see that the average effort per file is higher than that per issue. The main reason for this is the fact that when we map the effort from the issue level to the file level, we assign the file the maximum effort of all the issues that touched it.

The distribution at the file level seems to be skewed for the Hibernate and the JBoss projects as well. However, for Mule and Spring, the skew and kurtosis values are relatively small. This can be confirmed from Figure 3, which plots the distribution of effort at the file level for all four projects. Once again, we see a spike in the number of files that have an effort of approximately 12 hours in the Mule project. After manual inspection of the files with that effort value, we found that once again, these files were the ones changed by the large issues mentioned earlier.

5. Case Study Results

This section discusses our three research questions. First, we examine the correlations between the various metrics and effort. Then, we determine which of LOC, code or process metrics best predicts effort. Finally, we study the implications of our findings on previous effort-aware bug prediction studies.

RQ1. Does LOC correlate better with effort than other code and process metrics?

To examine the relationship between the various metrics and effort, we use correlation analysis. Figures 4, 5, 6 and 7 shows the Spearman correlation of all extracted metrics with effort for the Hibernate, JBoss, Mule and Spring projects, respectively.

In addition, Table 6 presents the correlations values and their statistical significance based on their p-values. The absolute improvement over LOC is shown in brackets, under the Improv. column. Due to the large number of metrics, we only show the correlation between the best performing metrics from each category (i.e., LOC, complexity, size and churn) and effort. For example, for the Hibernate

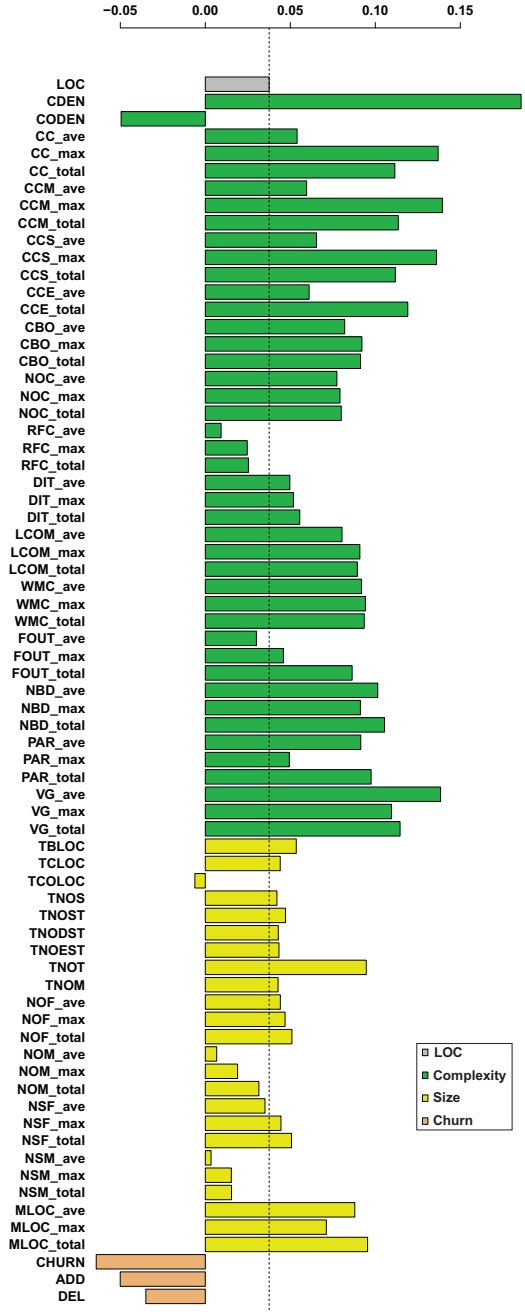


Figure 4: Correlation of Metrics using Maximum Effort in Hibernate

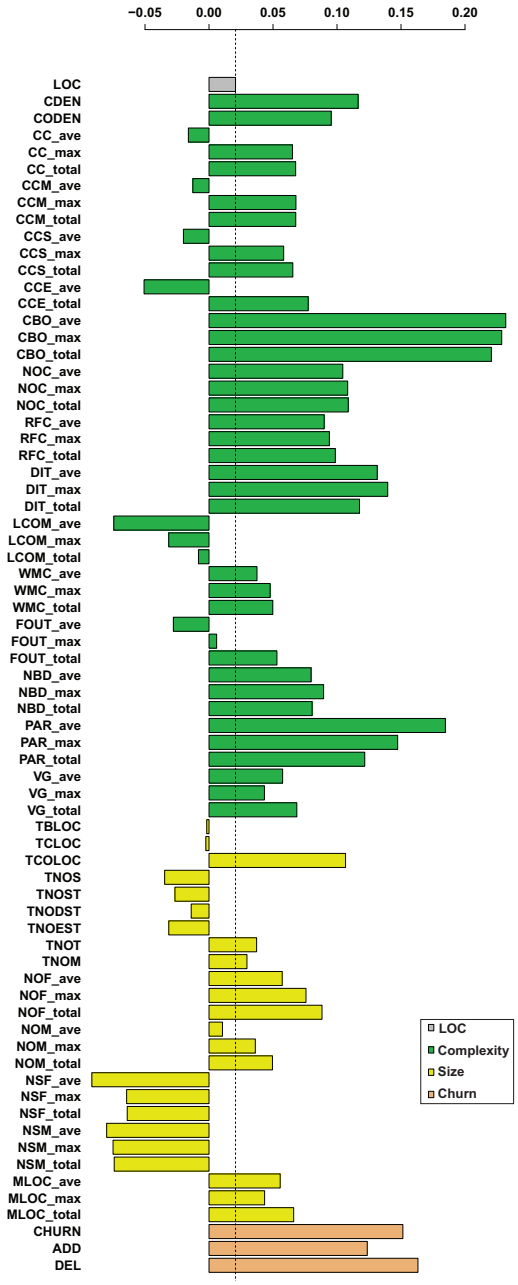


Figure 5: Correlation of Metrics using Maximum Effort in JBoss

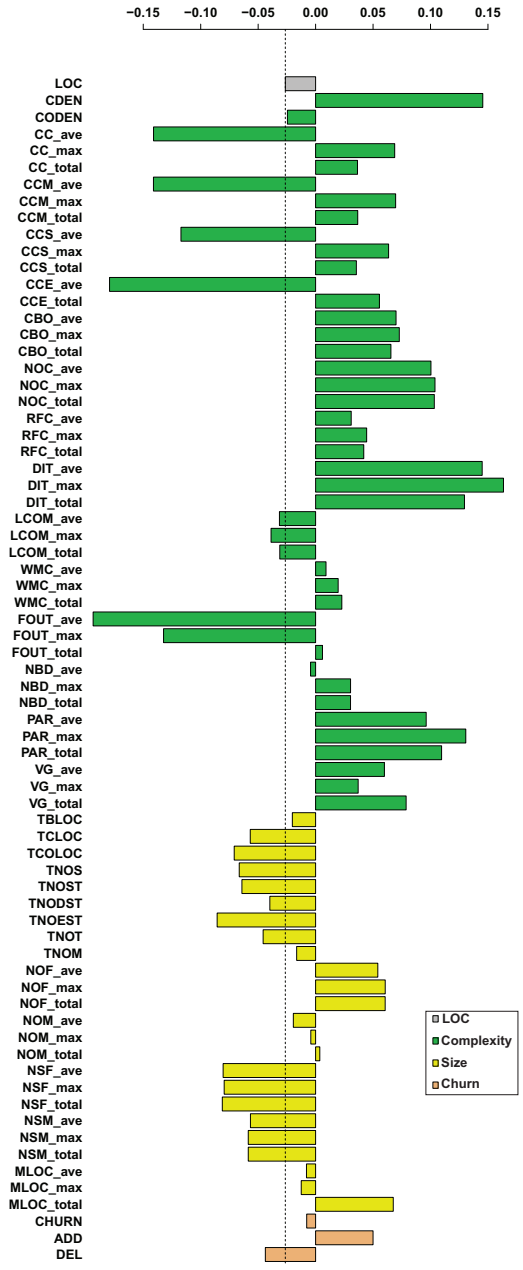


Figure 6: Correlation of Metrics using Maximum Effort in Mule

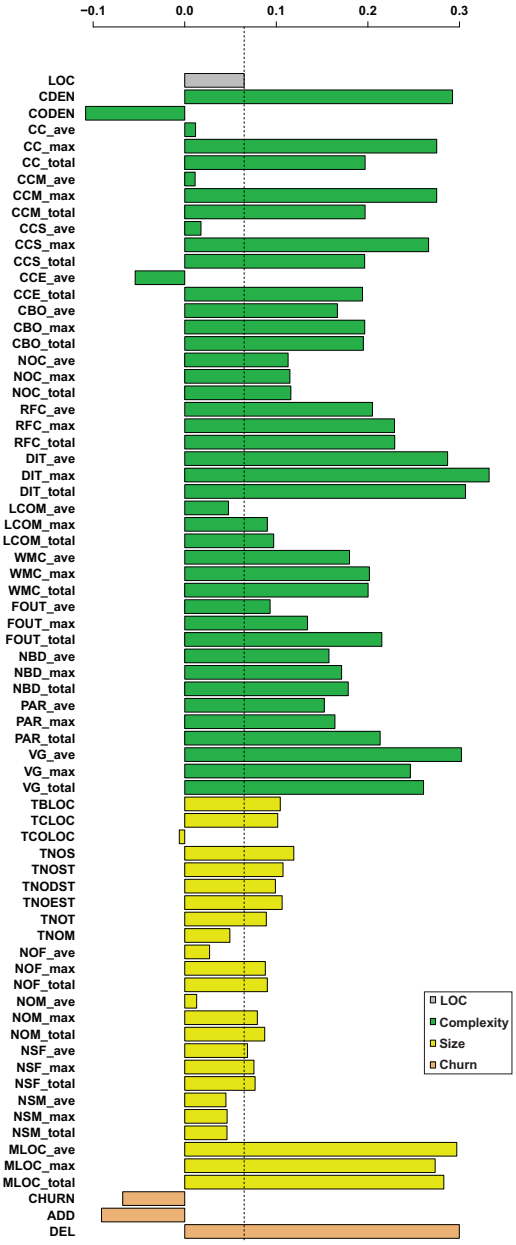


Figure 7: Correlation of Metrics using Maximum Effort in Spring

project, the CBO_max metric is the complexity metric with the highest correlation with effort. The category of metrics that has the highest correlation with effort for each project is shaded in grey. We observe the following:

LOC has weak correlation with effort.

We observe from Table 6 that LOC does not have a high correlation with effort. The highest correlation is 0.13 in the JBoss project. The correlation for the other projects is even lower and for the Mule project this correlation is practically zero.

Complexity metrics have a stronger correlation with effort than LOC.

Table 6 shows that complexity metrics have a stronger correlation with effort than LOC. In fact, complexity metrics have a stronger correlation than size and churn metrics as well. However, we would like to note that the correlation of all of the metrics with effort remains rather low.

Different metrics best correlate with effort for different projects.

Examining the metrics that best correlate with effort in Table 6, we observe that for almost each project, a different metric (in each category) best correlates with effort. Even though we find that complexity metrics correlate best with effort, the specific metric depends on the project.

Although the measured correlations are not that strong, previous work [33, 29] showed that low correlations do not necessarily mean low prediction accuracy. Therefore, we now proceed to build prediction models for effort to investigate whether or not the code and process metrics (e.g., complexity) are a better *predictor* of effort than LOC.

RQ2. Is LOC a better predictors of effort than other code and process metrics?

Motivation. Thus far, we were able to establish that LOC does not have a high correlation with effort and that complexity metrics correlate better with effort. Now, we would like to examine which of LOC or the other code and process metrics best predict effort.

In addition, we would like to know how well code and process metrics predict effort for different projects. For example, we would like to know if there exists one category of metrics that performs well at predicting effort in all projects or whether each project has its own set of metrics that predict effort well. If we can find one metric or category of metrics (e.g., complexity) that outperforms the other categories of metrics in all projects, then we can recommend this category of metrics for future effort-aware studies.

Approach. To determine the best category of metrics to predict effort, we build linear regression models that aim to predict the effort required to address a file. The independent variables of the models are the set of metrics in each category (e.g., churn) and the dependent variable is the effort required to address the file

(measured in seconds). We use a linear regression model (rather than a logistic regression model, for example) since effort is a continuous variable, given in seconds.

To perform our analysis, we divide the data set into: a training set and a test set. The training set contains 90% of the entire data set and is used to train the linear regression model. The test set contains the remaining 10% of the data and is used to test the accuracy of the model. We perform this random splitting 10 times (i.e., 10-fold cross validation) and report the averages of the 10 iterations.

Based on our analysis in Section 4 that showed that the independent variables and the dependent variable have high skew and kurtosis values, we log-transformed the variables to alleviate the effects of the high skew and kurtosis. Log-transformations are commonly employed in prediction work to deal with high skew and kurtosis [37, 33].

We employ a backward elimination stepwise regression technique to select only a relevant, non-redundant set of independent variables [27]. In backward selection stepwise regression, an initial model is built using all independent variables. Then, the deletion of independent variables is tested and independent variables are removed from the model if they do not make a statically significant contribution to the model. This process is repeated until the no further improvements are possible. Finally, the model will contain the best set of independent variables that achieve the highest variance explained.

Furthermore, to avoid problems due to multicollinearity [32], we computed the pairwise correlations of the independent variables and removed any independent variable that had an inter-correlation greater than 0.8. After performing the prediction, we evaluate the performance of the model using two complementary measures. First, we evaluate the *predictive power* of the model by calculating the Spearman correlation between the output of the model with the actual effort for each file. The *explanative power* of the model is evaluated by measuring the R^2 value of the fitted model. These two measures are commonly used in previous work to evaluate the accuracy of prediction models [10].

Results. Table 8 shows the predictive and explanative power of each model. In addition, we report the statistical significance of the correlations for each model.

In all four projects, prediction models built using complexity metrics outperformed the LOC models in terms of predictive and explanative power (shown in grey in the table). We would like to point out here that although the prediction models using complexity metrics were the best performing, their overall predictive and explanative power are still low. We discuss this issue in more detail in Section 6.

Table 7: (Pearson) Effort Prediction Results

Project	Category	Average Predictive Power (Correlations)	Average Explanative Power (R^2)
Hibernate	LOC	0.11 ***	0.00
	Complexity	0.22 ***	0.08
	Size	0.02	0.00
	Churn	-0.11 ***	0.00
	All	**0.26 *** **	**0.10**
JBoss	LOC	0.05 *	0.00
	Complexity	0.30 ***	0.13
	Size	0.13 ***	0.05
	Churn	0.17 ***	0.03
	All	**0.35 *** **	**0.17**
Mule	LOC	-0.20 ***	0.00
	Complexity	0.30 ***	0.11
	Size	0.05	0.01
	Churn	-0.25 ***	0.00
	All	**0.29 *** **	**0.12**
Spring	LOC	0.06 **	0.00
	Complexity	0.28 ***	0.10
	Size	0.24 ***	0.09
	Churn	0.22 ***	0.07
	All	**0.39 *** **	**0.19**

(***) $p < 0.01$, (**) $p < 0.05$, (*) $p < 0.1$

We also built a model that contained metrics from all categories, labeled "All" in Table 8. To assure that only the significant metrics are used, we performed the same stepwise regression technique mentioned above to prune the "All" model as well. The "All" model seems to provide the best performance, providing a significant improvement over the LOC model and a slight improvement over the complexity model in all four projects. This finding shows that although complexity metrics outperform LOC, using all metrics is the best option when predicting effort.

Table 8: (Spearman)Effort Prediction Results

Project	Category	Average Predictive Power (Correlations)	Average Explanative Power (R^2)
Hibernate	LOC	0.10	0.00
	Complexity	0.20 **	0.08
	Size	0.05	0.00
	Churn	-0.05	0.00
	All	**0.27 *** **	**0.10**
JBoss	LOC	0.02	0.00
	Complexity	0.31 ***	0.13
	Size	0.15 *	0.05
	Churn	0.19 **	0.03
	All	**0.36 *** **	**0.17**
Mule	LOC	-0.20	0.00
	Complexity	0.30	0.11
	Size	0.04	0.01
	Churn	-0.20	0.00
	All	**0.28**	**0.12**
Spring	LOC	0.05	0.00
	Complexity	0.29 ***	0.10
	Size	0.26 ***	0.09
	Churn	0.25 ***	0.07
	All	**0.41 *** **	**0.19**

(***) $p < 0.01$, (**) $p < 0.05$, (*) $p < 0.1$

Complexity metrics are better predictors of effort than LOC, churn and size metrics. However, combining LOC, complexity, churn and size metrics provides the best prediction of effort.

Table 9: Correlations of Top 20% Most Risky Files Lists when LOC, Complexity and All Metrics are Used as a Measure for Effort

	JDT			PDE			Platform		
	LOC	**Complexity**	**All**	**LOC**	**Complexity**	**All**	**LOC**	**Complexity**	**All**
LOC	-	-0.33	0.93	-	-0.20	0.94	-	-0.24	0.94
Complexity	-0.33	-	-0.30	-0.20	-	-0.15	-0.24	-	-0.20
All	0.93	-0.30	-	0.94	-0.15	-	0.94	-0.20	-

RQ3. How do our findings impact prior effort-aware bug prediction findings?

Motivation. In the previous research question, we found that using complexity metrics is a better predictor of effort than using LOC alone, however, using all metrics is the best predictor of effort. Now we would like to investigate the impact of the aforementioned findings on the prior work on effort-aware models. Generally speaking, effort-aware models serve two purposes. First, they assist in ranking which files to address first. Second, they provide practitioners with an estimate of the effort (e.g., hours) required to address the selected files.

Therefore, we use the data set from the study done by Kamei *et al.* [19] to see whether using LOC significantly impacts the findings of their study based on the two above mentioned criteria, *file-ranking* and *effort-estimation*.

Approach. In their work, Kamei *et al.* [19] built bug prediction models to predict the number of bugs in a file. Then, the files are prioritized based on their risk values, which is measured as $Risk_{LOC} = \frac{\#Bugs}{LOC}$. The main idea behind this is that files with the most bugs and least effort (measured in LOC) would be prioritized higher.

File-ranking. To examine the impact on the study by Kamei *et al.* [19] in terms of file rankings, we generated three lists of files that are ranked based on their risk. One list used LOC as a measure of effort in the risk equation, another list used complexity metrics and the third list used all of the metrics. Then, we use the Spearman rank correlation to measure the similarity between the three lists.

Effort-estimation. Since we showed that using complexity or all metrics is a better measure of effort, we would like to know if using LOC over- or under estimated the amount of effort, compared to when complexity or all metrics are used. The way we measure this is we generate a list of files using the $Risk_{LOC}$ equation. Then, we measure the amount of effort required to address this list of files when LOC is used, when complexity metrics are used and then all metrics are used as a measure for effort. Finally, we compare the difference in effort for the generated lists of files.

To perform the comparison, we need to first estimate effort (in seconds) from LOC, complexity and all metrics. Because we do not have explicit effort data for Kamei's data set, we estimate effort data using the best performing prediction models from our effort prediction analysis, as shown in Table 8. For LOC, we use the model from the Hibernate project. For complexity, we use the model from the JBoss project and for all metrics we use the model from the Spring project.

Performing the aforementioned steps allows us to compare LOC, complexity and all metrics, in terms of effort, on a level playing field.

Results.
File-ranking. Table 9 shows the correlation coefficients between the ranked lists generated using LOC, complexity and all metrics for the JDT, PDE and Platform projects. Using complexity as a measure of effort significantly affects the results of Kamei's study since the lists generated using complexity have a negative weak correlation with the file list using LOC (which was used as a measure of effort by Kamei *et al.*) for all three projects. On the other hand, the lists generated using LOC have a positive, strong correlation with the file list generated using all metrics, which proved to be the best predictor of effort (as shown in Table 8).

The strong correlation between LOC and all is encouraging since it shows that although LOC was not a good predictor of effort, its use in effort-aware bug prediction models is valid. For the purpose of file ranking, using LOC as a measure of effort performs similar to our best effort predictor, the all metrics model.
Effort-estimation. Figure 8 shows the estimated effort required versus the percentage of bugs. The three lines show the effort estimate when LOC, complexity and all metrics are used as a measure for effort. We observe that LOC slightly under-estimates effort compared to complexity metrics, and both LOC and complexity significantly under-estimate effort compared to all metrics (which was the best predictor of effort). For example, to address 20% of the bugs, using LOC would indicate that approximately 10% of the total effort is needed, complexity indicates that approximately 12% of the total effort is need, whereas all metrics indicates that approximately 35% of the total effort is needed. Therefore, using LOC under-estimates effort by approximately 66% compared to our best predictor of effort.

Due to space limitation, we only show the results for the JDT project, however, a similar trend was observed for the PDE and Platform projects.

Our finding shows that although LOC may not impact the file ranking aspect of effort-aware bug prediction model, it does lead to a different result in terms of effort-estimation. In other words, our findings show that if someone is interested in the effort-aware ranking of files, using effort-aware prediction models, then using LOC is fine. However, if someone is interested in approximating the amount of effort that needs to be spent on these files (e.g., to address these bugs), then LOC is not a good measure since it seems to underestimate this effort.

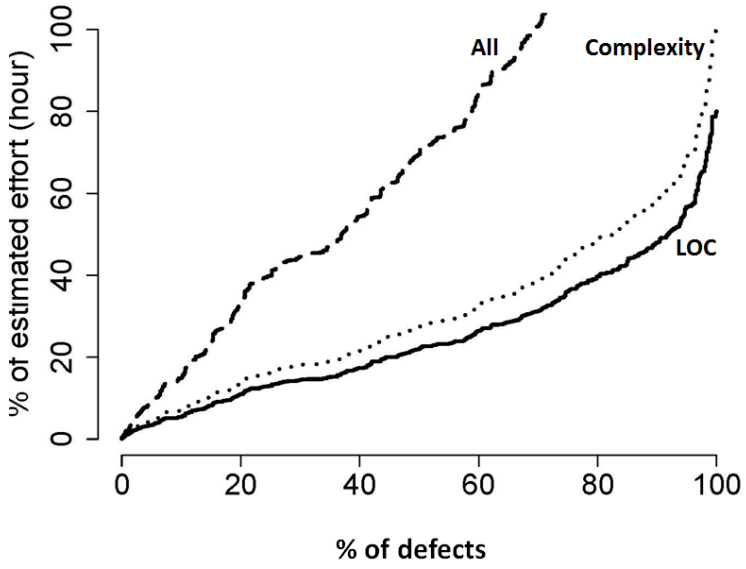


Figure 8: JDT

For the purpose of ranking the most risky files, using LOC as a measure of effort performs similar to our best effort predictor. However, for the purpose of effort-estimation, using LOC under-estimates the amount of effort required compared to our best effort predictor.

6. Discussion

Studying what is a good measure of effort is a very important issue, since recent research shows that effort needs to be taken into account for bug prediction to become more practical [22]. However, there is a lack of empirical evidence of how to best measure effort. This is mainly due to the lack of availability of explicit effort data.

This study is a step in the right direction, however, it is certainly far from perfect. For this reason, we plan to (and encourage other researchers to) expand on this work in the future. To facilitate future studies, we are making our data publicly available to the research community through the PROMISE repository. In particular, we see three directions for immediate improvement:

Collecting Better Quality Effort Data. The four projects used in our study are some of the first to record effort data. However, as we can see from Table 1, most projects have a very small percentage of issues with non-empty effort data. Having such little data is a major obstacle that our community needs to overcome in order to conduct large-scale research. We hope that this study will inspire future developers to carefully input effort data that can be used for future studies.

We plan to revisit our study when more, better quality data is made available. At the same time, we are attempting to obtain such data from commercial projects, who may keep better track of explicit effort data.

Improving Prediction Accuracy. Although our experiments showed that using complexity and all metrics had a higher correlation and could better predict effort than LOC, we believe there is still room for improvement. We believe the prediction accuracy could be improved by obtaining more and better quality data and by exploring more categories of metrics. For example, it may be helpful to include code ownership metrics [6] in addition to the code complexity metrics, since it may take less effort to change complex code by a person who owns the code versus a person who is new to the same piece of complex code.

In addition, we plan to explore the use of different prediction techniques such as regression trees or random forest.

7. Threats to Validity

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual variable.

The collected effort data is input by the developer who made the change. In certain cases, the input data may not reflect the actual effort (e.g., this data maybe misrepresented or inflated due to specific motivations) required to change the file. Also, only about 10% of the reports had an non-empty effort field. This scarcity of the data may impact some of our findings or the quality of our models. That said, as mentioned earlier our data set contains approximately double the amount issues compared to prior state-of-the-art work (i.e., Weiß *et al.* [36]).

Furthermore, we map effort from the issue level to the file level by setting each file's effort as the maximum effort (from all the issues that touched the file) out of the all issues that touch it. In some cases (e.g., when the fix is very easy), using the maximum value might not be representative. However, we chose to be conservative and used the maximum effort value in order to consider the worst case scenario.

When studying the impact of our findings on prior work, we use the coefficients from the best model of the projects we have effort data for (i.e., Hibernate, JBoss, Mule and Spring) to map LOC and complexity to effort measured in seconds for the JDT, PDE and Platform projects. Ideally, we should use the coefficient from the project itself to do this mapping. However, we do not have effort data for the JDT, PDE and Platform projects.

As shown in Table 1, a very small number of issues contain effort data. Having such a small percentage to conduct our studies may affect our findings, however, we would like to note that this type of data is very rarely available. Also, our study uses approximately double the amount of issues compared to the state of the art to using such data. Therefore, we see this study as a starting point and plan to continue to mine such data as it becomes more widely available.

Threats to External Validity considers the generalization of our findings. All of the four projects used are open source projects written in Java, therefore, our results may not generalize to other open source or commercial projects written in other programming languages.

When revisiting the prior effort-aware bug prediction results, we found that using LOC as a measure for effort under-estimates effort, compared to when complexity is used. Although we observed the same phenomena in all three projects, the same trend might not generalize to other project.

When evaluating the relationship of the different metrics with effort, we used Spearman's rank correlation and the R^2 measures. Traditionally, effort estimation work uses measures such as the Mean Magnitude of Relative Error (MMRE) since its primary goal is to measure how accurate an effort estimation is [38]. In our case however, our primary goal was to measure the relationship of the metrics with effort, hence using correlations and the R^2 measures are a better fit for our analysis.

8. Conclusions

Recent work stressed the importance of considering effort in bug prediction models. Most of these studies used LOC as the de facto measure of effort. In this paper, we set out to empirically examine which of LOC, or other code and process metrics are the best measures of effort. Through a study on four open source projects, we found that complexity measures have the highest correlation with effort. Furthermore, we also found that using a combination of LOC, code and complexity metrics provides a better prediction of effort than using LOC alone.

Replication of the work by Kamei *et al.* [19] shows that using LOC in effort-aware bug prediction models provides results that are similar to using all metrics when prioritizing the most risky files. However, for the purpose of effort-estimation, using LOC under-estimates the amount of effort required compared to our best effort predictor.

References

[1] S. N. Ahsan, J. Ferzund, and F. Wotawa. Program file bug fix effort estimation using machine learning methods for oss. In *Proc. Int'l Conf. on Software Engineering & Knowledge Engineering (SEKE'09)*, pages 129–134, 2009.

[2] E. Arisholm and L. C. Briand. Predicting fault-prone components in a java legacy system. In *Proc. Int'l Symposium on Empirical Software Engineering (ISESE'06)*, pages 8–17, 2006.

[3] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[4] M. Auer, A. Trendowicz, B. Graser, E. Haunschmid, and S. Biffl. Optimal project feature weights in analogy-based cost estimation: Improvement and limitations. *IEEE Trans. Softw. Eng.*, 32(2):83–92, 2006.

[5] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[6] C. Bird, N. Nagappan, H. Gall, P. Devanbu, and B. Murphy. An analysis of the effect of code ownership on software quality across windows, eclipse, and firefox. Technical Report MSR-TR-2010-140, Microsoft Research, 2010.

[7] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1981.

[8] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, 1999.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[10] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'10)*, pages 31 –41, 2010.

[11] K. E. Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using object-oriented design metrics. *J. Syst. Softw.*, 56(1):63–75, 2001.

[12] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proc. Int'l Workshop on Recommendation Systems for Software Engineering (RSSE'10)*, pages 52–56, 2010.

[13] M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, and D. Notkin. Future of mining software archives: A roundtable. *IEEE Software*, 26(1):67 –70, 2009.

[14] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[15] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, 2005.

[16] A. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance (FoSM'08)*, pages 48–57, 2008.

[17] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'09)*, pages 78–88, 2009.

[18] R. Hewett and P. Kijsanayothin. On modeling software defect repair time. *Empirical Softw. Engg.*, 14(2):165–186, 2009.

[19] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort aware models. In *Proc. Int'l Conf. on Software Maintenance (ICSM'10)*, pages 1–10, 2010.

[20] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–564, 1999.

[21] T. J. McCabe. A complexity measure. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'76)*, page 407, 1976.

[22] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proc. Int'l Conf. on Predictor Models in Software Engineering (PROMISE'09)*, pages 7:1–7:10, 2009.

[23] T. Menzies, O. Jalali, J. Hihn, D. Baker, and K. Lum. Stable rankings for different effort models. *Automated Software Engg.*, 17(4):409–437, 2010.

[24] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, 2010.

[25] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'08)*, pages 181–190, 2008.

[26] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'05)*, pages 284–292, 2005.

[27] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'05)*, pages 284–292, 2005.

[28] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'06)*, pages 452–461, 2006.

[29] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, 1996.

[30] Scientific Toolworks, Inc. Understand 2.6. `http://www.scitools.com/`.

[31] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Trans. Softw. Eng.*, 23(11):736–743, 1997.

[32] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM'10)*, pages 1–10, 2010.

[33] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker. Does calling structure information improve the accuracy of fault prediction? In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'09)*, pages 61–70, 2009.

[34] Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *IEEE Trans. Softw. Eng.*, 32(2):69–82, 2006.

[35] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[36] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'07)*, pages 1–8, 2007.

[37] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proc. Int'l Workshop on Predictor Models in Software Engineering (PROMISE'07)*, pages 1–7, 2007.

[38] M. Shepperd, M. Cartwright, and G. Kadoda. On Building Prediction Systems for Software Engineers. *Empirical Software Engineering*, 5(3):175-182, 2000.