Evaluating the Effectiveness of Code2Vec for Bug Prediction When Considering That Not All Bugs Are the Same

by

Kilby Baron

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Bug prediction is an area of research focused on predicting where in a software project future bugs will occur. The purpose of bug prediction models is to help companies spend their quality assurance resources more efficiently by prioritizing the testing of the most defect prone entities. Most bug prediction models are only concerned with predicting whether an entity has a bug, or how many bugs an entity will have, which implies that all bugs have the same importance. In reality, bugs can have vastly different origins, impacts, priorities, and costs; therefore, bug prediction models could potentially be improved if they were able to give an indication of which bugs to prioritize based on an organization's needs. This paper evaluates a possible method for predicting bug attributes related to cost by analyzing over 33,000 bugs from 11 different projects. If bug attributes related to cost can be predicted, then bug prediction models can use the approach to improve the granularity of their results. The cost metrics in this study are bug priority, the experience of the developer who fixed the bug, and the size of the bug fix. First, it is shown that bugs differ along each cost metric, and prioritizing buggy entities along each of these metrics will produce very different results. We then evaluate two methods of predicting cost metrics: traditional deep learning models, and semantic learning models. The results of the analysis found evidence that traditional independent variables show potential as predictors of cost metrics. The semantic learning model was not as successful, but may show more effectiveness in future iterations.

Acknowledgements

I would like to thank my supervisors, Mei Naggappan and Mike Godfrey, and my research partner Gema Rodriguez-Perez, for their guidance.

I would like to thank Harold Valdivia-Garcia for his initial work in this research area.

And I would like to thank my family (including Jane and Shaun Baron, Coreena Rego, Gus, and Beau) for their love and support.

Dedication

This is dedicated to Joey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems have become an ubiquitous part of everyday life. Software systems are essential to a range of tasks and activities: from entertainment (news, games), to work (invoicing programs, asset management), and to the internet of things. Human beings have built their lives around a constant reliance on technology. Behind the scenes, the software companies that people rely on work relentlessly to make sure their products work smoothly and as expected. Despite the rigorous efforts of software developers, bugs in software inevitably occur.

In 2016, the software testing company Tricentis estimated that over the course of a year software failures affected 4.4 billion people and impacted $1.1 trillion in assets [10]. This implies that reducing the rate and impact of bugs by just 1% could save companies billions of dollars. The current method for ensuring software quality is called software quality assurance (SQA). SQA is a collection of standards, such as ISO 9000, and procedures, such as technical reviews and process monitoring, that are administered across every phase of the software development life cycle [7]. In a perfect world, every organization would have a full-fledged quality department. However, most organizations are opposed to the increase in overhead costs and instead rely on project team members to conduct inspections and testing [7]. In other words, many organizations are interested in improving their quality assurance practices only if it does not significantly increase their costs.

Bug prediction is an area of research that explores improving the effectiveness of SQA resources at a low-cost. The goal of bug prediction is to identify which entities in a project are most likely to have bugs. This allows quality assurance teams to catch more bugs by allocating more testing resources to the most bug prone entities [14]. This is particularly useful for organizations that have limited testing resources. More than 200

papers about bug prediction have been published in the past decade [14]. Some papers explore different classifiers — such as Random Forest, Naive Bayes, and Neural Networks — and compare their effectiveness at predicting bugs [5, 11, 16, 20, 26]. Other papers investigate independent variables to find metrics that are highly correlated with bug density [36, 41, 63]. Over the years, thousands of combinations of classifiers and predictors have been tested; however, the dependent variable always falls into one of four categories:

1. Whether an entity has a bug
2. The likelihood that an entity has a bug
3. The number of bugs in an entity
4. The bug density of an entity

Bug prediction models are interested only in whether or not a bug exists in an entity. The output of a typical bug prediction model makes no distinctions between the bugs it predicts, therefore there is no way for organizations to determine which defect-prone entity to prioritize. For example, consider two files, `File_A` and `File_B`. `File_A` is a large file with five bugs; they do not significantly impact the project, and can be fixed by an intern relatively easily. `File_B` is a smaller file and only has a single bug; the bug has resulted in a critical error, and will require an experienced engineer to fix. If a traditional bug prediction model were to predict the occurrence of these bugs, it would suggest that administrators prioritize `File_A` because it has more bugs. Even an effort-aware model would likely prioritize `File_A`, because of the greater rate of bugs per line. However, if a QA team is trying to reduce bug fixing costs, it would want to prioritize `File_B` because the bug in `File_B` is a more serious threat.

In this project we will attempt to add an additional layer of granularity to the output of bug prediction models. Recent studies have shown that bugs can have vastly different origins and impacts [47, 48, 56, 58, 59, 60, 62]. The total cost of a bug is most likely a function of a multitude of factors. For example, there is a positive correlation between experience and wages [9]; therefore, under the same amount of time, a bug fixed by an experienced developer will cost more than a bug fixed by a novice. Similarly, the longer it takes to fix a bug, the more expensive it will be. Facebook uses lines of code committed by a developer as a measure of how much work a developer can do in a unit of time [49]. Therefore, the lines of code required to fix a commit can be used as a rough estimate for how long it will take to fix.

To begin the conversation, this project will try to identify some preliminary methods of predicting bug metrics related to cost. Since no bug prediction models have done this before, this project will evaluate predictors that have been proven effective for predicting the number of bugs in a file. Specifically, the independent variables in this project will be two code metrics (lines of code (LOC) and cyclomatic complexity (CC)), and one process metric (code churn). We will also evaluate one experimental prediction method, code semantics, and compare the results with the traditional models.

These independent variables will be evaluated based on their effectiveness for predicting three bug metrics related to cost: developer experience, bug-fix size, and bug priority. To perform this analysis, data was collected from the issue trackers and code repositories of 11 open source Apache projects written in Java. These resources were used to build a complete bug history for each project, with rich metadata for a total of 33,472 bugs. This data was initially used to find empirical evidence to support the claim that not all bugs are the same. Next, the impact of prioritizing files based on number of bugs instead of cost was evaluated. Then, we measured the correlation between traditional predictors and cost metrics, and built deep learning models to predict our new dependent variables. Finally, the traditional models were compared to another type of deep learning model, an emerging technique called semantic learning, which learns to predict bug metrics based on the source code itself.

1.1 Research Questions

RQ1 Are bugs meaningfully distinguishable from one another in terms of their priority, size, and the experience required to fix them?

RQ2 If bug prediction models could prioritize files based on other bug attributes, would they suggest that developers spend more SQA resources on different entities?

RQ3 Are traditional predictors of bugs correlated with unconventional bug metrics that are related to cost?

RQ4 Can traditional predictors of bugs be used to predict unconventional bug metrics related to cost?

RQ5 Can a semantic learning model predict bug metrics related to cost as effectively as a traditional model?

Chapter 2

Related Work

More than 200 studies have been published in the past decade about bug prediction [14]; it is an expansive research area, with many different techniques, predictors, and target metrics to explore. Some studies aim to evaluate the usefulness of different classifiers such as statistical, machine learning and deep learning models [5, 11, 16, 20, 26], while others compare the predictive power of different independent variables[36, 41, 63]. This section provides a synopsis of the research that has been done so far in the area of bug prediction.

2.1 Classifiers

Many papers are focused on evaluating different classifiers and trying new techniques for bug prediction [5, 11, 16, 20, 26]. For example, in 2018 Bowes et al. compared the performance of Random Forest, Naïve Bayes, RPart and SVM classifiers when predicting defects [5]. Their results showed that different classifiers produce a unique subset of defects, but some classifiers (RPart, Random Forest) are more consistent than others (Naïve Bayes, SVM).

Several studies have suggested that the classification technique used to train defect prediction models has little impact on overall performance[23, 29, 51]. In 2014 Shepperd et al. analyzed 42 high quality studies of defect prediction to determine which factors influence predictive performance[51]. They found that the choice of classifier has little impact on performance, and the most explanatory factor is the research group conducting the study[51].

Since then, some studies have disputed the claim that classifier has no impact. In 2015 Ghotra et al. sought to replicate this study and concluded that some classification techniques such as simple logistic regression tend to outperform more complex classifiers [11]. Ultimately, there is still no consensus which classifier is best for defect prediction, and the research is still ongoing.

This research was taken into account when structuring the approach of RQ4, in which several bug prediction models are built using traditional inputs. Since there is no consensus as to which classifier is the strongest for bug prediction, a neural network was selected for the models in RQ4 so that they would be as similar as possible to the semantic models in the following chapter.

2.2 Independent Variables

The focus of some papers is to investigate the predictive potential of different independent variables in bug prediction models[36, 41, 63]. For the most part, researchers have focused on two types of metrics for defect prediction: product metrics and process metrics. Product metrics are statistical attributes of code files such as lines of code and cyclomatic complexity; their use as independent variables for defect prediction is very common, particularly in early research [6, 15, 24, 28, 34].

These days, product metrics are still used for defect prediction, but many papers prefer to use process metrics[16, 25, 27, 32, 33, 36]. Process metrics are statistics that originate from the software development process such as number of changes or number of developers. In 2013, Rahman et al. analyzed the applicability and efficacy of both code and process metrics from several different perspectives[44]. Their results suggest that product metrics are generally less useful than process metrics for prediction, partly because product metrics tend to change little between releases[44].

Some of the most recent work in the area of defect prediction experiments with a new type of independent variable altogether: code semantics. Rather than using attributes of the source code or development process, some researchers are building models that can detect bugs from the source code itself [42, 57]. For example, in 2020 Piotrowski et al. built a model that uses *code smells* to identify buggy code[42]. Their research found that code smells such as *God Class, God Method, and Message Chains* are particularly correlated with software defects[42].

In research questions 3 and 4 of this paper, we evaluate the strength of traditional independent variables for the task of predicting bug attributes. The independent variables

were selected based on their success in past research, and include lines of code, cyclomatic complexity, and churn.

Meanwhile, other researchers have used code semantics for defect prediction by extracting information from abstract syntax trees (ASTs)[57]. In 2018, Wang et al. used a deep belief network (DBN) to create semantic representations of programs for file level defect prediction[57]. The semantic representations of programs were constructed from tokens in the original files, and outperformed state-of-the-art defect prediction models that use traditional features [57]. In this paper, we will evaluate whether semantic learning can be used to predict additional bug attributes.

2.3 Dependent Variables

Despite the high number of bug prediction studies, when it comes to what these studies are trying to predict *(the dependent variable)*, there are four possible outcomes: (1) whether an entity has a bug or not [17, 21, 22]; (2) the likelihood that an entity has a bug [2, 13, 37]; (3) the number of bugs present in each entity [39, 61, 63]; and (4) the bug density of each entity [22, 35, 54]. Bugs can come in all varieties; they can have different causes, impacts, and fix requirements, but most bug prediction research is not concerned about differences between the bugs that are detected. Considering the complexity that a single bug can have, this indicates that there is a largely unexplored area of bug prediction that remains to be explored.

2.4 Effort-Aware Models

While there are few bug prediction papers that acknowledge the differences between the bugs they detect, some papers still consider how their results can help to improve the efficiency of QA resources [2, 20, 26]. For example, some studies show that it may be more cost effective to allocate QA resources based on inspection effort rather than number of bugs [2, 20, 26]. In 2010, Arisholm et al. built models to predict fault-proneness of Java files, and measured the cost effectiveness of their models by using file size as a proxy for inspection effort [3]. The same technique was used by Kamei et al. to evaluate whether process metrics are still more effective predictors when the prediction models are effort-aware.

Another way of helping users prioritize buggy files is to focus on the prediction of *high-impact* bugs. In 2011, Emad Shihab et al. state that customers are most impacted

by bugs that break functionality, whereas practitioners are most impacted by defects in unusual files (surprise defects) [53]. Therefore, to predict the most important bugs, they train prediction models to detect files containing breakages and surprises. While their prediction models are effective, they note that their models would benefit from being able to predict additional attributes about bugs, such as bug type [53].

This project differs from existing effort-aware defect prediction research in that it aims to multiple factors with which users can prioritize buggy entities. While most effort-aware prediction models use file metrics to estimate inspection effort, this paper ranks buggy entities by attributes of the bugs themselves. This paper is a reproduction and continuation of a dissertation called "Understanding the Impact of Diversity in Software Bugs on Bugs Prediction Models" by Harold Valdivia – Garcia [55]. In his original work, Dr. Valdivia – Garcia argues that current bug prediction models implicitly assume that all bugs are the same [55]. To address this issue he provides an approach to predict blocking bugs early on, and investigates causes of false negative categorizations [55].

This paper explores another method of addressing the argument that bug prediction models implicitly assume that all bugs are the same. The first part of this paper is a reproduction of Dr. Valdivia – Garcia's initial research to validate the claim that not all bugs are the same. All of the data was recollected from the same projects, and the same method was used to compare bugs attributes. Likewise, this paper uses the same methods to analyze the impact of assuming all bugs are the same and measure the correlation between bug atributes and independent variables. After validating the results of the initial study, this paper explores a new avenue to address bug differences in prediction models. First, traditional deep learning models are built and trained to predict additional bug attributes. The results of these models are then compared with similar models that were trained to predict the same attributes using semantic encodings provided by code2vec.

Chapter 3

Data Collection

3.1 Motivation

In this project we propose that bug prediction models could be improved if they were able to provide different attributes about the bugs they detect. Throughout this project we will show how different bugs can be from one another, demonstrate the impact of treating all bugs the same, and explore two ways of performing bug prediction with more granularity.

The bugs in this project are differentiated from one another along three metrics: bug fix size, priority, and the experience of the developer who fixed the bug. Ultimately, our goal is to make bug prediction more useful by creating proof-of-concept models to show that these three metrics can — and should — be predicted. The purpose of this chapter is to outline the data collection process.

3.2 Data Descriptions

This section presents the metrics used as dependent and independent variables. Then, it describes the projects used in the study and the approach that was followed to extract and pre-process the data.

3.2.1 Dependent Variables

Developer experience (Y_{Exp}): Y_{Exp} models the experience of the developer fixing a bug by counting the number of bug-fixing commits contributed by the developer before

the release-date. If a bug has been fixed by two or more developers, the mean value is used. Prior works have used similar metrics to measure experience [60, 52, 31]. Developer experience is used as an independent variable in this study because of its relationship with cost. More experienced developers are capable of resolving more difficult bugs compared to novice developers, however they also cost more to employ. Thus, if a bug requires experience to fix, it will have a higher cost. If bug prediction models can also predict the required experience, then they can be used to expedite bug triage and ultimately cut costs.

Bug-fix size ($Y_{ChgLines}$): $Y_{ChgLines}$ the total number of lines added and removed from an entity to fix a particular bug. In 2016, researchers at Facebook use lines of code committed by a developer as a measure of how much work a developer can do in a unit of time [49]. Following this precedent, this study uses the bug fix size as an approximate function of time and effort. It is understood that this is far from a perfect measure, but it is commonly used in other studies [49]. The more time it takes for a developer to fix a bug, the greater the cost of that bug. Therefore, if bug prediction models can predict the bug fix size, that information can be used to estimate the cost of the bug.

Priority ($Y_{Priority}$): $Y_{Priority}$ is a label that helps QA individuals triage bugs. Each bug is assigned a priority in the project's issue tracking system (e.g., Blocker/Urgent, Critical, Major/Normal, Minor, and Low/Trivial). To calculate the priority for a specific file, we aggregate the priorities of all bugs that affected that file. Priority is initially a categorical variable, but we convert the categories to a numeric representation from one to five; the higher the number, the higher the priority.

Number of Bugs ($Y_{\#\ Bugs}$): $Y_{\#\ Bugs}$ is equal to the total number of bugs in an entity. Since the number of bugs in a file has been used in many past studies [12, 38, 39, 50, 63, 61], it is used as the baseline for this study. In this study, the correlation between the independent variables and bug metrics will be compared to the correlation with the baseline. Later, the baseline will be used to analyze how different dependent variables affect the entities that are prioritized.

Bugginess (Y_{Buggy}): Y_{Buggy} is a binary variable that indicates whether or not an entity contained a bug during a certain period of time. If an entity was modified by at least one bug fixing commit in the given time frame, Y_{Buggy} value is 1. Otherwise it is 0. This variable is used as baseline for the cost-sensitive bug prediction prototypes in Chapters 7 and 8. It is used instead of $Y_{\#\ Bugs}$ because the semantic model requires data at the method level, and the same method cannot have two bugs because fixing the bug will change the method.

Table 3.1: Description and aggregate metrics of the case study projects

Project	Description	Bugs Collected	Commits Collected	Releases	Link %
Accumulo	Distributed Storage System	1618	6691	7	92%
Bookkeeper	Write-ahead logging service	435	2610	10	95%
Camel	Integration Framework	3633	30519	32	90%
Cassandra	Distributed Storage System	5021	13284	23	37%
CXF	Web Service Framework	3802	14140	11	70%
Derby	Relational Database	2650	7472	16	76%
Felix	OSGI Service Platform	2563	12214	433	87%
Hive	SQL-like Engine for Hadoop	7250	13022	24	91%
OpenJPA	Persistence Framework	1057	4746	10	87%
Pig	Query Language for Hadoop	2149	3033	17	74%
Wicket	Web Application Framework	2819	16183	57	80%
Total	–	32997	123914	640	79%

3.2.2 Independent Variables

The purpose of this study is to explore how well certain bug metrics (listed above) can be predicted using traditional bug prediction models. The models in this study are designed as proofs-of-concept, to analyze whether traditional models have potential for predictions of bug metrics. Therefore, while the independent variables in this study are not meant to be the most robust, they are representative of a typical bug prediction model of the past decade.

The two most popular types of independent variables for bug prediction models are code metrics and process metrics [6, 15, 16, 24, 28, 25, 32, 27, 33, 34, 36]. Therefore the independent variables for this study are some of the most popular metrics of each category: Lines of Code (X_{LOC}), cyclomatic complexity (X_{CC}), and churn (X_{Churn}). X_{CC} is a code metric which indicates the complexity of a program, and X_{Churn} is a process metric defined as the total number of lines added and removed from an entity during a given period of time.

3.2.3 Studied Projects

For this study, data was collected from 11 large Apache Software Foundation projects that are commonly used for bug prediction research [8, 45, 46]. The projects are all written in

Java, and they cover a wide range of domains (i.e., SQL and NoSQL databases, Enterprise, Service and Web frameworks, Data Analytics platforms, etc). The projects all share a common issue tracking software, JIRA, which is a central source of data for this project. Table 3.1 summarizes the projects in terms of bug-reports, commits and number of releases.

3.3 Approach

3.3.1 Identifying bug-fixing commits

All of the independent varibles in this study were collected from two sources: JIRA and Git. JIRA is an issue tracking software that contains extensive metadata about each project's bug history. The specific metrics that were collected from JIRA for each project are the list of unique bug IDs, and their associated priorities. This data was only collected for JIRA entries that were marked as "Closed" or "Fixed".

The rest of the data that was collected for this case study comes from Git. Git provides a detailed development history for every project; it was used to collect data for X_{Churn}, $Y_{ChgLines}$, and Y_{Exp}. These metrics were only collected from commits that specifically addressed bugs. We call these *bug-fixing commits*.

Bug-fixing commits were identified using an approach similar to previous studies [45, 46]. When a developer fixes a bug, they will usually note which bug they are fixing in their commit message by referencing the *bug ID*. This bug ID comes from JIRA. Therefore, all commits that have a bug ID from JIRA in their commit message are considered bug-fixing commits. Since, many projects have more than one branch, there was a risk of including duplicate commits in our data set. For example, there were two identical commits fixing bug CAMEL-8091 in release-branches *camel-2.13.x* and *camel-2.14.x*. Commits with identical metadata were removed because duplicates can inflate the metrics and bias the case study.

In total, 33K bugs were extracted, out of which 26K (79%) were successfully linked to at least one commit. Of these 26K bugs (and considering the major releases described in the next section), 5,455 post-release bugs were identified. Similarly, 124K commits were extracted, out of which 31.3K (25.2%) were successfully linked to at least one bug (i.e., bug-fixing commits).

3.3.2 Releases

The data for each project was collected from the three consecutive minor releases with the most bug-fixing commits. The releases and release dates of every project were collected from GitHub. In order to avoid bias, the independent variables had to be collected from a period of time before the dependent variables. This was accomplished using a technique similar to previous studies [35, 37, 40]. Each release was divided into two time periods: pre-release and post-release. Given two consecutive releases A and B, the middle point between them is considered the end of the post-release period for release A and the start of the pre-release period for release B. Independent variables were collected from the pre-release periods and dependent variables were collected from the post-release periods. Table 3.2 contains descriptive statistics for the data sources of each project, divided by release.

3.3.3 Collecting independent variables

Code and process metrics were collected for every target release of every project. The two code metrics, X_{LOC} and X_{CC}, were collected using a software called Understand by SciTools. Given a code project, Understand can output code metrics for every file in the project. This was done for the three target releases of each project.

The process metric, X_{Churn}, was derived for each file using the Git data. First, the bug-fixing commits were filtered to only include commits that were made during pre-release periods. Each commit was then queried using the *git diff –numstat* command, which outputs the number of lines added and deleted for each file modified by that commit. The final of each file is equal to the total number of lines added and deleted by bug-fixing commits during the pre-release period.

3.3.4 Collecting dependent variables

The dependent variables — $Y_{\#\ Bugs}$, $Y_{ChgLines}$, $Y_{Priority}$, and Y_{Exp} — were collected from the post-release period. The specific processes that were used to collect each metric are detailed below.

$Y_{\#\ Bugs}$was calculated by counting the number of unique bug-IDs from the bug-fixing commits that affected that file.

Each bug was assigned a value for Y_{Exp} by counting the number of previous bug-fixing commits from the author that fixed the bug, starting from their first contribution until their

Table 3.2: Information on the releases studied in each of the projects

Project	**Releases**	**# Pre-release Commits**	**# Post-release Commits**	**# Files**	**# Buggy Files**
Accumulo	1.4.0	716	423	801	72
	1.5.0	970	496	871	197
	1.6.0	959	745	1035	197
Bookkeeper	4.1.0	76	82	220	13
	4.2.0	122	123	311	87
	4.3.0	93	57	440	73
Camel	2.10.0	471	807	2570	228
	2.11.0	784	307	2852	137
	2.12.0	412	541	3199	58
Cassandra	1.0.0	421	402	507	109
	1.1.0	411	444	562	143
	1.2.0	561	512	729	101
Cxf	2.5.0	550	392	2799	221
	2.6.0	479	418	2953	138
	2.7.0	386	1349	3021	87
Derby	10.1.0	1	482	1712	169
	10.2.0	945	571	1963	1418
	10.3.0	785	586	2203	439
Felix	1.8.0	1396	3685	56	18
	1.4.0	72	759	40	4
	1.6.0	598	1696	40	24
Hive	2.0.0	613	290	3659	47
	2.1.0	351	832	0	0
	2.2.0	972	0	4038	318
Openjpa	1.0.0	898	187	1002	130
	1.1.0	185	228	1049	29
	1.2.0	271	691	1055	101
Pig	0.10.0	189	144	993	19
	0.11.0	162	124	1030	137
	0.12.0	96	121	1091	62
Wicket	1.3.0	289	620	1304	128
	1.4.0	499	666	1364	174
	1.5.0	1618	538	1555	143

last contribution before the release-date. The Y_{Exp} value for each file is a combination of the experience value of every bug that affected that file in the given release. If the bug was fixed by more than one developer, Y_{Exp} for that bug is equal to the mean experience of each developer.

$Y_{ChgLines}$ was calculated using the same method as churn. The final $Y_{ChgLines}$ of each file is equal to the total number of lines added and deleted by bug-fixing commits during the post-release period.

$Y_{Priority}$ was calculated by converting the issue priority in JIRA to a 5-point scale. Each file's priority is the sum of the priorities of all bugs that affected that file in the given release. If a file has not been touched after the release-date, the post-release metrics are set to zero.

3.3.5 Summary

The remaining chapters of this paper will each address a different research question. Each research question will be answered using the same independent and dependent variables. There are three independent variables: X_{CC}, X_{LOC}, and X_{Churn}. These metrics were selected because they are some of the most common code and process metrics used for bug prediction. The independent variables will be evaluated based on their ability to predict the dependent variables. There are five dependent variables in total; two that will be used as a baseline, Y_{Buggy} and $Y_{\#\ Bugs}$, and three that will be compared to the baseline, $Y_{ChgLines}$, $Y_{Priority}$, and Y_{Exp}.

Chapter 4

Evaluation: Are All Bugs The Same?

4.1 Motivation

The vast majority of research on bug prediction focuses only on the likelihood that an entity will have a bug, and/or how many bugs an entity will have. The problem with this narrow scope is that it ignores the fact that bugs may have different impacts and costs. In this work, we seek to improve the practicality of bug prediction models by increasing the granularity of the results. If bugs can be very different from one another in terms of different attributes related to cost, then administrators of bug prediction models may benefit from being able to predict these attributes. To begin this research, we must first confirm the hypothesis that bugs differ from one another along different cost related attributes. This chapter quantitatively examines the diversity of bugs in terms of Y_{Exp},

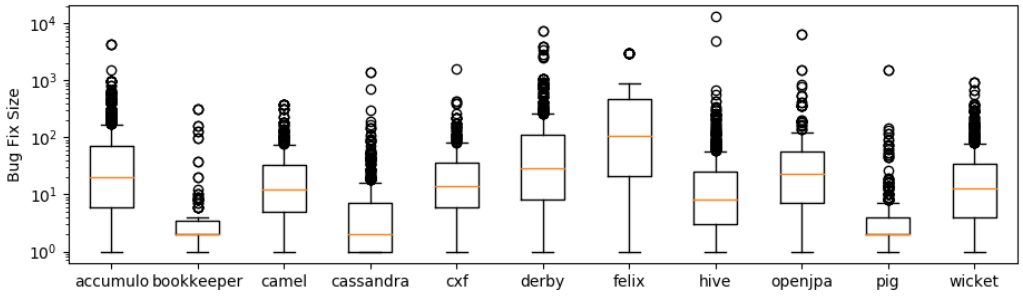

Figure 4.1: Bug – fix size (in log scale) across all projects

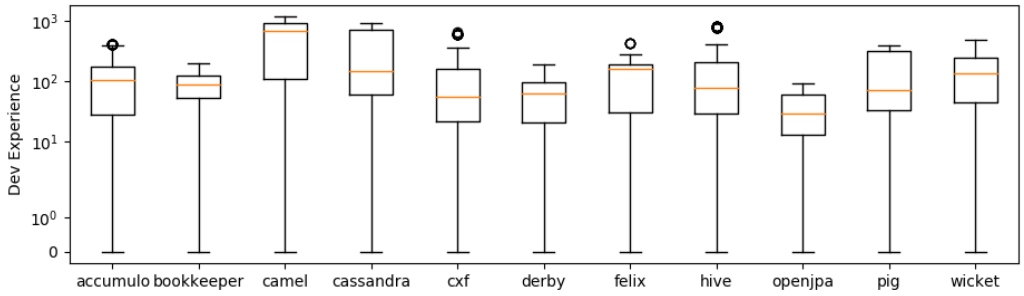

Figure 4.2: Developer Experience (in log scale) across all projects

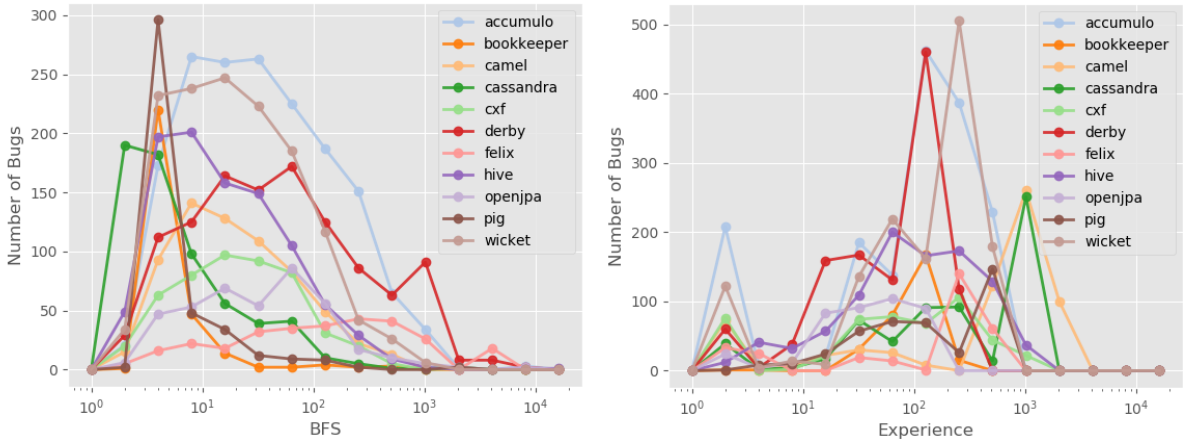

Figure 4.3: Distributions of developer experience and bug-fix size

$Y_{ChgLines}$, and $Y_{Priority}$.

4.2 Approach

This section outlines the approach used to verify the hypothesis that not all bugs are the same. The metrics in this research question were all compared at the bug level, and the differences between bugs were measured in three ways:

1. Examining the distributions of each metric
2. Examining the variation of each metric compared to its mean
3. Measuring the correlations between all pairs of metrics

The distributions of Y_{Exp} and $Y_{ChgLines}$ were compared by examining box-plots and calculating the inter-quartile range (IQR) of each dependent variable. A metric's IQR is the size of the middle 50% of its box-plot. The IQR provides mathematical quantification of each metric's distribution. For example, in the project Camel, Y_{Exp} has an IQR of 833 because the 25th quartile is 111 previous bug-fixing commits and the 75th quartile is 944 previous bug-fixing commits. If a metric is very similar for all bugs of a particular project it will have a narrow distribution. The more the bugs differ from one another, the larger their distribution.

The variability of each metric was measured using relative standard deviations (RSD), also known as the "coefficient of variation". The RSD indicates whether the standard deviations of each distribution are small or large when compared to the mean. For example, consider two projects, A and B, that have average developer experiences of 60 and 200 previous fix commits respectively, and standard deviations of 30 and 40. Although the standard deviation for A is smaller than for B, the coefficient of variation for A (50%) is larger than for B (20%). This indicates that while the developers of B have a greater difference in total number of commits, the developers of A have a greater difference in experience *relative to one another*.

The differences in $Y_{Priority}$ were analyzed slightly differently than the other metrics because $Y_{Priority}$ is a categorical variable. The distribution of $Y_{Priority}$ was assessed by counting the number of bugs that fall into each category, and comparing the tallies for each project.

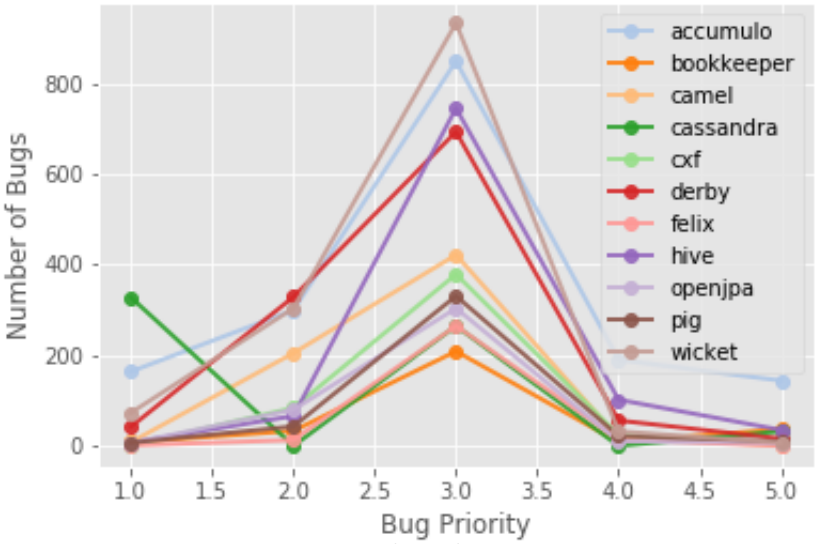

Figure 4.4: Priority distribution across 11 projects

The correlation between metrics was examined by normalizing the metrics of each project and calculating the Pearson correlation coefficient for each pair of metrics. The Pearson correlation coefficient is a value between -1 and 1 that indicates the amount of linear correlation between two variables. The closer the coefficient is to 0, the smaller the correlation between metrics.

4.3 Results

Figure 4.1 displays the distribution of bug fix sizes for each project, and Figure 4.2 displays the distribution of developer experiences for each project. For clarity, Figure 4.3 displays the same distributions using line graphs to provide another perspective. Wider distributions show that not all bugs in each project are the same in terms of $Y_{ChgLines}$ and Y_{Exp}. In fact, the distributions for both metrics are so large, they could only be captured by a log scaled Y axis. The IQR values are presented in Table 4.1.

The box-plots and IQRs for $Y_{ChgLines}$ vary widely between projects. Bookkeeper and Pig have a mean $Y_{ChgLines}$ and IQR of only 2 lines, with only a handful of bug fixes that change over 10 lines of code. Conversely, Derby and Felix both have IQRs of over 100 lines of code, with mean $Y_{ChgLines}$ close to 100 as well. The distributions of Y_{Exp} show a

Table 4.1: Interquartile Ranges for Developer Experience for each of the 11 studied projects

Project	$Y_{ChgLines}$	Y_{Exp}
Accumulo	66	150
Bookkeeper	2	72
Camel	28	833
Cassandra	6	648
CXF	30	138
Derby	102	102
Felix	441	441
Hive	22	22
OpenJPA	50	50
Pig	2	2
Wicket	30	30

similar pattern. The median IQR for Y_{Exp} is 168 previous bug-fixing commits, which is a substantial difference between 25th and 75th percentiles, but the distributions vary widely between projects. The project with the smallest distribution of Y_{Exp} is OpenJPA with an IQR of 48, while Camel has the largest distribution of Y_{Exp} with an IQR of 833 previous bug-fixing commits.

In order to more robustly evaluate the extent to which these metrics differ, we also compare their relative standard deviations (RSD). Table 4.3 shows the RSD calculated for the dependent variables in each of the projects. The variation for each metric is very large compared to their means, with values ranging from 181% to 989% for $Y_{ChgLines}$ and from 52% to 122% for Y_{Exp}.

Since $Y_{Priority}$ is a categorical variable it is analyzed slightly differently. Rather than examining box plots, and measuring the RSD and IQR of the distributions, the number of bugs in each category is tallied for each project. The $Y_{Priority}$ distributions are displayed in Figure 4.4, where each line represents a project, and each point indicates the number of bugs with that priority.

The results of this analysis show that the majority of bugs have a priority level of 3 (i.e., Major/Normal). The average project has almost 500 level 3 bugs, but only 130 level 2 bugs (i.e.,Minor), and less than 50 bugs of every other category. Finally, Table 4.2 shows

Table 4.2: Correlations between dependent variables of each project

Metric A	Metric B	accumulo	bookkeeper	camel	cassandra	cxf	derby	felix	hive	openjpa	pig	wicket
Dev. Exp.	**Bug-fix size**	-0.049	0.131	-0.038	-0.046	-0.003	0.045	-0.038	0.006	0.005	-0.039	0.020
Dev. Exp.	**Priority**	0.045	-0.007	-0.139	-0.021	-0.020	-0.114	-0.131	0.004	0.087	-0.051	-0.010
Priority	**Bug-fix size**	0.062	0.012	0.084	0.021	0.030	0.108	0.011	0.025	-0.011	0.040	0.131

the results for the correlation analysis between dependent variables. The results show that none of the independent variables are strongly correlated with one another. Most of the Pearson coefficients have less than a 0.05 difference from 0.

4.4 Summary

In summary, the results of our analysis show that:

1. All projects have a wide range of bug-fix sizes
2. Most bug fixes are less than 10 lines of code
3. Developer experience is very diverse across all projects
4. The majority of bugs have priority level 3, with level 2 being the next most common
5. $Y_{ChgLines}$, Y_{Exp}, and $Y_{Priority}$ are not strongly correlated with one another

These results show that on any individual metric not all bugs are the same. While $Y_{ChgLines}$ and $Y_{Priority}$ are more tightly clustered than Y_{Exp}, the distributions of every metric show a range of possible values. Moreover, since the metrics in each project are not correlated with one another, using more than one metric to describe a bug will distinguish it from the others even further.

Table 4.3: Relative Standard Deviation of Bug-Fix Size and Developer Experience

Project	**Bug-fix Size**	**Dev. Experience**
Accumulo	296.0%	91.0%
Bookkeeper	410.0%	52.0%
Camel	181.0%	68.0%
Cassandra	544.0%	94.0%
Cxf	251.0%	120.0%
Derby	290.0%	78.0%
Felix	200.0%	70.0%
Hive	989.0%	122.0%
Openjpa	519.0%	78.0%
Pig	749.0%	92.0%
Wicket	210.0%	88.0%

Chapter 5

Impact of Assuming All Bugs Are The Same

5.1 Motivation

In the previous chapter, it was demonstrated that bugs can be very different from one another in terms of their priority as well as the time and experience required to fix them. In RQ2, we hypothesise that to help allocate software QA resources it is important for bug prediction models to consider factors related to cost other than the number of bugs in a file. In this chapter we will determine how much the choice of metric affects the files that are prioritized.

5.2 Approach

To examine how the order of importance changes when prioritizing files along different metrics, the files were sorted in four ways: by the number of bugs per file, by the total bug-fix size per file, by the experience of the developer(s) fixing the bugs in a file as calculated in Section 3.2.1, and by aggregated priority. Each of the sorts results in a different ranking of files: $Rank_{\#\,Bugs}$, $Rank_{ChgLines}$, $Rank_{Exp}$, and $Rank_{Priority}$. For our comparison, we extracted the top 20 files from each ranking, because this represents ≈20% of the median number of buggy files from all of the projects (See Table 3.2).

To illustrate the number of files that we would have missed if we only used $Rank_{\#\,Bugs}$ to prioritize our QA resources, we counted the number of files in $Rank_{Exp}$, $Rank_{ChgLines}$,

Table 5.1: Number of missed files over three releases of each project.

	Number of files missed when using $Rank_{\#\ Bugs}$ instead of								
	$Rank_{ChgLines}$			$Rank_{Exp}$			$Rank_{Priority}$		
Project	R1	R2	R3	R1	R2	R3	R1	R2	R3
Accumulo	8	10	12	6	7	11	4	5	5
Bookkeeper	0	6	13	0	5	5	0	3	5
Camel	14	14	12	9	8	11	9	8	2
Cassandra	11	13	11	7	7	5	4	6	8
Cxf	12	11	10	16	10	5	2	0	2
Derby	12	19	16	9	11	7	4	7	3
Felix	0	1	0	0	3	0	0	3	0
Hive	9	-	19	9	-	8	6	-	5
Openjpa	15	9	10	8	4	7	4	3	3
Pig	2	16	9	2	9	9	2	3	5
Wicket	8	11	8	3	6	10	2	1	3

and $Rank_{Priority}$ that were not present in $Rank_{\#\ Bugs}$. It is important to emphasize that there were no predictions being done here. The file data was simply sorted along four different metrics and the top 20 files were compared.

5.3 Results

Table 5.1 reports the number of files in each ranking that were not in $Rank_{\#\ Bugs}$. The largest difference in ranking is between $Rank_{\#\ Bugs}$ and $Rank_{ChgLines}$. We observe that the difference in the top 20% of files ranges from 0 to 19 files with an overall median of 11. This means that by fixating on the number of bugs in a file, practitioners are putting off many of the bugs that will require the most time to fix.

The difference in ranking between $Rank_{\#\ Bugs}$ and $Rank_{Exp}$ is slightly smaller on average, with a median of 7 different files, a maximum of 16 and a minimum of 0. Finally, the smallest difference in ranking is between $Rank_{\#\ Bugs}$ and $Rank_{Priority}$, which has a median of 3 different files and a maximum of 9. $Rank_{Priority}$ is probably most similar to $Rank_{\#\ Bugs}$

because a file's priority is the sum of the priorities of every bug that affected that file. Since almost all bugs are priority 2 or 3, a file with a second bug is likely to have a higher priority than a file with one bug of the highest priority.

5.4 Discussion

So far, we showed that using number of bugs to prioritize files instead of other dependent variables can cause practitioners to miss most of the top files. However the current analysis might be susceptible to threats from just examining 20 files. It could be that, for example, a ranking of the 40 ($\approx 20\%$) buggiest files might cover the majority of the files in the other ranking as well. Therefore, an additional correlation analysis was performed to validate the findings. For the correlation analysis, the Spearman rank correlation was calculated between number of bugs and every other dependent variable. The results of the correlation analysis are displayed in Table 5.2.

The results of the correlation analysis are in line with the results of the file ranking analysis. We see again that priority is very strongly correlated with number of bugs; therefore, fixating on the number of bugs in a file will not cause developers to miss the bugs with the highest priority. Developer experience has a moderate correlation with number of bugs, with an average Spearman coefficient of 0.63 across all projects and releases. This supports the results of the ranking analysis which suggest that prioritizing only number of bugs will cause practitioners to overlook over 30% of the files that may need more experience to fix. Finally, the smallest correlation is with bug-fix size, which has an average Spearman coefficient of 0.45. This indicates that many of the largest bug fixes may not be prioritized when all bugs are considered the same. Ignoring the bug fix size could have a considerable impact on the efficient use of SQA resources. For example, if a project is working with a time requirement, putting off the largest bugs could mean that they will not be fixed in time.

5.5 Summary

In RQ2, we seek to determine whether being able to predict additional metrics for each bug will have an impact on the files that are prioritized. If the additional variables related to cost are not correlated with number of bugs, then the additional granularity can be used to prioritize bugs based on the developers needs. The analysis found evidence to show that prioritization based on number of bugs in a file can be somewhat different to the

Table 5.2: Spearman correlation coefficient between Bugs and each cost metric

Project	Y bfs			exp			priority		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
accumulo	0.44	0.67	0.48	0.63	0.79	0.58	0.69	0.88	0.83
bookkeeper	0.51	0.61	0.37	0.80	0.81	0.58	0.80	0.91	0.94
camel	0.35	0.28	0.25	0.55	0.40	0.58	0.69	0.56	0.66
cassandra	0.52	0.50	0.48	0.70	0.69	0.62	0.77	0.78	0.66
cxf	0.45	0.39	0.45	0.24	0.27	0.56	0.87	0.81	0.82
derby	0.40	0.66	0.47	0.72	0.77	0.54	0.75	0.81	0.74
felix	0.89	0.42		1.00	0.97		0.94	0.98	
hive	0.54	0.36		0.62	0.50		0.71	0.79	
openjpa	0.15	-0.15	0.50	0.64	0.84	0.67	0.76	0.84	0.91
pig	0.29	0.47	0.50	0.59	0.45	0.47	0.84	0.97	0.92
wicket	0.53	0.57	0.59	0.75	0.64	0.55	0.80	0.84	0.86

prioritization based on other dependent variables. While bug priority is closely correlated with the number of bugs, developer experience is moderately different, and bug fix size is substantially different. Therefore, practitioners may benefit from being able to predict additional bug metrics, because they can choose the dependent variable(s) that matter the most to their organization when prioritizing their software entities.

Chapter 6

Correlation between traditional bug predictors and cost metrics

6.1 Motivation

In RQ1, we provided evidence that not all bugs are the same. In RQ2, we showed that different files would be prioritized if prediction models were able to predict additional bug attributes. In this chapter, we investigate whether traditional independent variables for bug prediction (X_{LOC}, X_{CC}, and X_{Churn}) correlate with bug metrics other than $Y_{\#\ Bugs}$. In so doing, we will investigate whether these independent variables could potentially be used to predict $Y_{Priority}$, $Y_{ChgLines}$, and Y_{Exp}. The prediction of metrics other than bugginess is relatively new to the field of bug prediction. Therefore, to find independent variables the logical place to start is with metrics that have been proven effective for a similar task.

6.2 Approach

The predictive potential of the traditional independent variables was measured using R^2, a technique similar to prior studies[46]. R^2 — also known as the *coefficient of correlation* — indicates how much the variability in one variable can be explained by the variability in another. In other words, R^2 is a measurement of the statistical relationship between two variables.

Before performing the correlation calculation, both the independent and dependent variables were normalized. Next, generalized linear models were build for each dependent

Table 6.1: Average R^2 values in each project when different dependent variables are used

Project	# Bugs	Fix Size	Experience	Priority
Accumulo	18%	13% [0.72X]	9% [0.49X]	15% [0.86X]
Bookkeeper	11%	44% [4.15X]	7% [0.69X]	10% [0.93X]
Camel	7%	3% [0.48X]	6% [0.85X]	6% [0.93X]
Cassandra	39%	21% [0.52X]	27% [0.7X]	33% [0.84X]
Cxf	11%	5% [0.41X]	7% [0.63X]	11% [0.97X]
Derby	9%	1% [0.17X]	4% [0.51X]	12% [1.38X]
Felix	55%	32% [0.59X]	42% [0.77X]	54% [0.99X]
Hive	9%	13% [1.36X]	5% [0.56X]	9% [0.96X]
Openjpa	21%	28% [1.32X]	17% [0.82X]	21% [1.0X]
Pig	10%	5% [0.47X]	6% [0.61X]	10% [1.0X]
Wicket	26%	7% [0.27X]	17% [0.67X]	27% [1.02X]

variable. The regressors for each linear model were X_{LOC}, X_{CC}, and X_{Churn}. The relative contribution of each regressor was measured using the Lindemen, Merenda and Gold (LMG) method. The LMG method works by building the same model for all possible orderings of the independent variables. It then performs an ANOVA analysis to calculate the sequential variance decomposition of each ordering. The final variance contribution of each independent variable is equal to the average contribution from each ordering. In the results section, we will display the coefficients of correlation for each independent variable, as well as the R^2 values when the independent variables are combined.

6.3 Results

Table 6.1 summarizes and compares the average R^2 values in each project. Here, the results for *# Bugs* are used as the baseline to perform the comparison. The variance in every dependent variable, including the baseline, can be explained by variance in the independent variables to different degrees depending on the project. The coefficients of correlation for $Y_{ChgLines}$ are the least consistent; in one project the R^2 value is one fifth the size of the baseline, and in another it is over four times as large. The median R^2 value for $Y_{ChgLines}$ is half that of the baseline, and the average R^2 value is about equal to

the baseline. The results for Y_{Exp} are more consistent, with R^2 values $\approx$ 0.65X less than the baseline on average. Finally, $Y_{Priority}$ is most tightly correlated with the $Y_{\#\ Bugs}$, with nearly identical R^2 values for all projects.

In Table 6.2, we summarize how much each independent variable contributes to the coefficients of correlation in Table 6.1. For example, in the project Accumulo, the coefficient of correlation between the predictors and $Y_{\#\ Bugs}$ is 18%. Table 6.2 shows that 50.2% of this correlation comes from X_{LOC}, 49.4% comes from X_{CC}, and only 0.4% of the correlation comes from X_{Churn}. One interesting takeaway from this analysis is that the process metric X_{Churn} is consistently less correlated with $Y_{\#\ Bugs}$ than the other predictors. This is the opposite of what was expected, since recent works in bug prediction have argued that process metrics are more effective[44]. $Y_{ChgLines}$ was the only metric with which X_{Churn} showed a substantial correlation.

6.4 Discussion

Most research papers on the topic of bug prediction examine either the likelihood that an entity will have a bug or how many bugs an entity might have [2, 13, 17, 22, 21, 37, 35, 39, 54, 61, 63] In RQ1 we demonstrated that this narrow scope ignores the fact that not all bugs are the same, and presented three more metrics along which bugs differ that may affect overall bug cost. In this research question, we analyzed whether the independent variables typically used to predict the number of bugs in an entity might also be effective for predicting these three other metrics. To answer this question, linear regression models were used to measure the correlation between the classic independent variables and new bug related metrics. The intuition is if X_{LOC}, X_{CC}, and X_{Churn} are as strongly correlated with our new metrics as they are with $Y_{\#\ Bugs}$, they will be strong input variables for models that predict new metrics.

The results of the correlation analysis shows that, on average, the independent variables are most strongly correlated with $Y_{\#\ Bugs}$, followed by $Y_{Priority}$, Y_{Exp} and $Y_{ChgLines}$. However, the average correlation with $Y_{\#\ Bugs}$ is only marginally stronger than with the other metrics, and there are several projects in which $Y_{\#\ Bugs}$ does not have the strongest relationship with the predictors. Therefore, classic independent variables could be viable predictors for $Y_{ChgLines}$, Y_{Exp}, and $Y_{Priority}$ depending on the project; however, there is room for new metrics which might better explain the variability in these new bug metrics.

The analysis also examined which independent variables have the strongest relationships with which metrics. While prior works have found that process metrics outperform

product metrics, [36, 32, 12], we found that X_{Churn} consistently has the least relative contribution with all metrics (as shown in Table 6.2). Another interesting finding is that the order of importance among X_{LOC}, X_{CC} and X_{Churn} does not typically change even when the dependent variable changes. This implies that irrespective of the dependent variable chosen, the contribution of the independent variables to the total variability explained is likely to remain the same. Therefore, our current understanding of the relationship between various bug prediction metrics (independent variables) and the $Y_{\#\ Bugs}$ as dependent variable may carry over when predicting new bug metrics.

6.5 Summary

In this chapter, we found that traditional bug predictors show potential for use in prediction models that predict $Y_{Priority}$, $Y_{ChgLines}$, and Y_{Exp}. However, while the correlation with bug $Y_{Priority}$ was especially high, the correlations with the other metrics were not as consistent. Therefore, when using a broader range of dependent variables, researchers might need to investigate new metrics that can explain the variability in their data sets. It was also found that the relative contribution (i.e., order of importance) of independent variables tends to remain the same regardless of the dependent variable being predicted. This suggests that the strongest independent variables that can be used for bug prediction may also be some of the strongest predictors of other bug attributes.

Table 6.2: Average contributions across three releases of each project from each independent variable when different dependent variables are used

		Prediction Models			
Project	Features	Bugs	Fix Size	Experience	Priority
Accumulo	X LOC	50.2%	47.8%	50.3%	50.2%
	X CC	49.4%	45.7%	49.0%	49.4%
	X churn	0.4%	6.6%	0.7%	0.4%
Bookkeeper	X LOC	44.6%	48.7%	45.2%	44.7%
	X CC	41.9%	47.7%	42.1%	41.9%
	X churn	13.4%	3.6%	12.7%	13.4%
Camel	X LOC	47.6%	50.0%	46.7%	47.4%
	X CC	41.2%	38.8%	44.2%	39.9%
	X churn	11.2%	11.3%	9.1%	12.7%
Cassandra	X LOC	40.2%	44.7%	39.6%	40.5%
	X CC	37.6%	42.0%	38.9%	38.3%
	X churn	22.2%	13.3%	21.6%	21.3%
Cxf	X LOC	25.1%	32.8%	17.7%	25.1%
	X CC	31.8%	43.4%	24.1%	31.6%
	X churn	43.1%	23.8%	58.3%	43.3%
Derby	X LOC	42.1%	49.7%	45.4%	43.5%
	X CC	53.5%	47.6%	51.2%	52.4%
	X churn	4.4%	2.7%	3.4%	4.1%
Felix	X LOC	44.9%	50.5%	48.3%	45.3%
	X CC	37.5%	39.2%	40.6%	37.9%
	X churn	17.6%	10.3%	11.1%	16.8%
Hive	X LOC	46.6%	23.1%	47.5%	46.2%
	X CC	46.5%	24.7%	47.4%	46.3%
	X churn	6.8%	52.2%	5.1%	7.5%
Openjpa	X LOC	44.9%	14.6%	49.2%	46.7%
	X CC	38.1%	14.2%	42.6%	40.4%
	X churn	17.0%	71.2%	8.2%	12.9%
Pig	X LOC	38.7%	30.9%	54.8%	39.6%
	X CC	30.1%	29.8%	37.8%	30.7%
	X churn	31.2%	39.3%	7.5%	29.7%
Wicket	X LOC	42.9%	47.1%	42.1%	42.9%
	X CC	40.6%	41.6%	41.2%	40.9%
	X churn	16.5%	11.3%	16.6%	16.2%

Chapter 7

Traditional Prediction Model

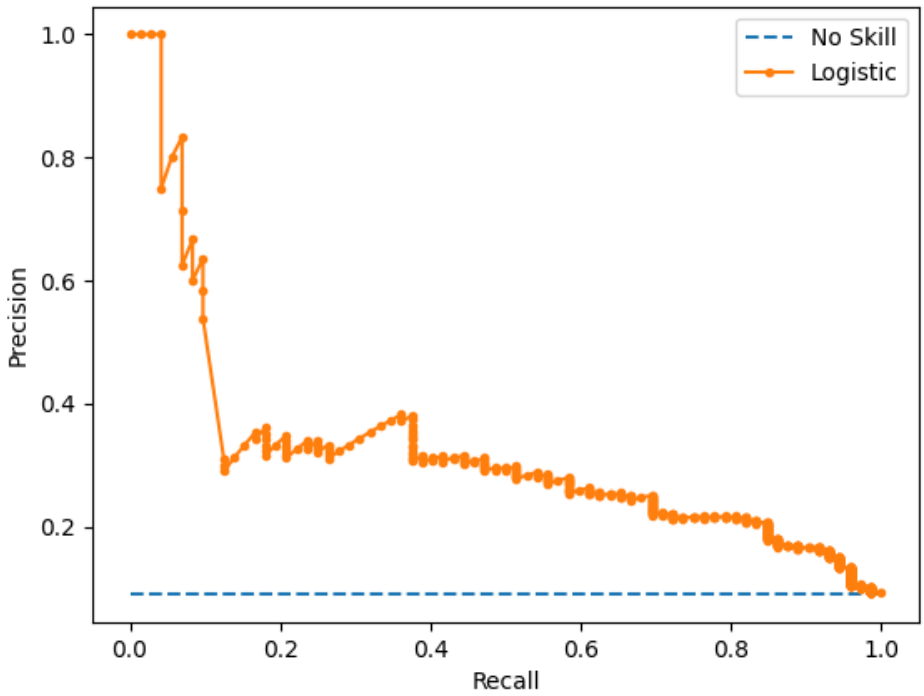

Figure 7.1: Precision-Recall Curve of the Intra-project Model for Accumulo R1

7.1 Motivation

After evaluating the viability of X_{CC}, X_{LOC}, and X_{Churn} as predictors for different bug metrics related to cost, the next step was to build prediction models. Several models were built as a proof of concept to show that $Y_{Priority}$, Y_{Exp}, and $Y_{ChgLines}$ can be predicted at a similar level of accuracy to Y_{Buggy}. Two types of models were built for this experiment, one traditional and one experimental. This chapter will discuss the process of building the traditional models, and evaluate their strength at predicting new metrics. The results of the traditional models will be used as a baseline with which to evaluate the semantic models in the next chapter.

7.2 Data Collection

The traditional models use the same input variables as those used in correlation analysis from Chapter 6: X_{CC}, X_{LOC}, and X_{Churn}. The output variables being predicted are Y_{Buggy}, $Y_{Priority}$, Y_{Exp}, and $Y_{ChgLines}$. These variables were collected for every file of every release of every project. The details of the data collection process and final data set are described in section 3.

In this chapter, traditional independent variables will be used to predict bug attributes. The results of these models will be used as a baseline with which to evaluate another set of models that will be trained to predict the same attributes using a different approach. The models in the next chapter require data to be collected at method level; this leads to some minor differences compared to the file level data used by the traditional models. Therefore, to facilitate the comparison between prediction models, the file level data in this chapter is modified slightly to match the data in the next chapter.

Firstly, only 8 projects are used in the prediction models. This is because three projects — Felix, Pig, and Wicket — did contain enough method level data to perform deep learning. Secondly, the baseline dependent variable has been changed from the number of bugs in an entity ($Y_{\#\ Bugs}$) to whether an entity contains at least one bug (Y_{Buggy}). This is because when a method undergoes a bug fix it is no longer exactly the same method from a semantic perspective; therefore, a method cannot have more than one bug. Since the bugginess metric is changed from continuous to binary, the prediction models are trained to predict the average value for each entity rather than the aggregate.

Table 7.1: Output layer of each prediction model

Output Variable	Variable Type	Function	Nodes
Bugginess	Binary	Sigmoid	1
Priority	Categorical	Softmax	5
Experience	Continuous	Linear	1
Fix Size	Continuous	Linear	1

7.3 Model Design

Separate models were built for each of the target bug metrics. Research has shown that the choice of classifier has a minimal effect on bug prediction models [23]; therefore, the classifier in the traditional models was selected to be as similar as possible to the semantic models that they will be compared to. The semantic models employ a deep learning architecture, therefore the traditional models use a neural network classifier. Every model is a neural network written in python using the Keras library. Every model has 3 input nodes, and three hidden layers using rectified linear unit activation functions with fifty nodes each. Each model also uses the Adam optimization algorithm to account for the sparse and noisy data. The output nodes of each model differ depending on the predicted variable. Details about the output layers of each model are displayed in 7.1.

Two different methods of selecting the training and testing data were tested. First the models were tested using intra-project data; these models were only trained on data from the same project as the test data. The train and test sets were divided by release. The prediction models were only trained on one release at a time and tested on the subsequent release. The models trained on intra-project data are then contrasted with the same models trained on inter-project data. Inter-project data includes data from every project. The test set consists of data from one project, and the training set consists of data from every other project. To ensure that the train and test sets are not derived from too broad a time frame, the models are only tested on one release at a time. The models tested on the first release of one project are trained on the first releases of the other projects, the models tested on the second release are trained only on the second releases of the other projects, and so on.

7.4 Results

The results for the prediction models that use X_{CC}, X_{LOC}, and X_{Churn} to predict bug metrics related to cost show varied levels of effectiveness.

7.4.1 Bugginess

The models that predict Y_{Buggy} are trained to output the probability that a file has a bug with a value between zero and one. The effectiveness of these models is measured using precision-recall curves. Precision the number of true positives divided by all positive predictions, and recall is the ratio of true positives divided by the number of true positives and false negatives. Only a small fraction of the files in each release are buggy, therefore the skill of each model depends its ability to detect these buggy files. This makes precision and recall ideal measures of skill because they are not concerned with true negatives.

To measure the precision and recall of a model, the output of the test set must be converted from probabilities to binary values. This is done using a threshold, where every probability above the threshold is registered as a buggy prediction, and every value below the threshold is considered a clean prediction. The precision and recall of the model will vary widely depending on the threshold; therefore rather than picking one threshold, the precision and recall of each model is calculated using one hundred different thresholds. This is the basis of a precision-recall curve.

A precision-recall curve plots the precision (y-axis) and recall (x-axis) for many thresholds. As an example, Figure 7.1 displays the precision-recall curve of the intra-project model that was tested on Accumulo 1.5.0. If the model has no skill, the precision-recall curve will be a horizontal line whose y-value depends on the ratio of buggy and clean files in the training data. Similarly, the precision-recall curve for a very skilled model will have a precision that is near 1 until the threshold reaches 1. One way of measuring a model's overall skill using the precision-recall curve is by measuring the area under the curve (AUC). The AUC values for the intra-project bugginess models are displayed in Table 7.2 and the AUC values for the inter-project models are displayed in Table 7.3. The median AUC for intra-project models is 0.312 and the median AUC for inter-project models is 0.352.

To interpret the results of the bug prediction models, it helps to have a baseline to compare to. In 2008, Lessmann et al. compared 22 classifiers across 10 data sets from the NASA MDP repository to perform a bias-free comparison [23]. Their results showed that most classifiers achieve a promising result of 0.7 or more[23]. The prototype prediction models in this study are not up to this standard. This is not surprising considering that the

Table 7.2: AUC Scores for Intra-Project Bugginess Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R1	0.328	0.442	0.113	0.518	0.238	0.793	N/A	0.033
R2	0.443	0.423	0.044	0.301	0.158	0.228	N/A	0.324

Table 7.3: AUC Scores for Inter-Project Bugginess Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R0	0.323	0.237	0.256	0.552	0.216	0.369	0.109	0.379
R1	0.428	0.428	0.123	0.496	0.220	0.728		0.184
R2	0.484	0.474	0.079	0.448	0.305	0.352	0.353	0.302

benchmark models use a larger data set and 37 independent variables, while our prototypes use 3[23]. This is because the purpose of this project is not to necessarily beat the state-of-the-art models, but rather to make a case for cost-sensitive prediction models. The difference between the skill of our prototypes and the industry standard is simply an indication of how much stronger the other prediction models will be when using a complete set of attributes.

7.4.2 Priority

The effectiveness of the models that were trained to $Y_{Priority}$ is measured by accuracy. The accuracy of a model is equal to the number of correct predictions divided by the number of false predictions. In total there are 5 categories for $Y_{Priority}$: trivial, minor, major, critical, and blocker. The accuracy of the intra-project model for each project is displayed in Table 7.4, and the accuracy of the inter-project model for each project is in Table 7.5.

The accuracy of the intra-project models varies widely between projects and releases; several models make the correct prediction more than half of the time, while others have accuracies close to zero. $Y_{Priority}$ is a bug metric therefore the priority models were only trained and tested on files that contained bugs. Due to the significantly smaller sample size, it is likely that some of the intra-project models suffer from over-fitting.

The median accuracy for the inter-project models is 0.431. These models were trained on data from all other projects, and therefore benefit from a larger dataset. The reduced risk of overfitting could be one reason that the inter-project models are more accurate. The accuracy of the inter-project models also varies quite a bit between different projects and releases; however, the accuracy rarely falls below one third. This accuracy and consistency

Table 7.4: Accuracy Scores for Intra-Project Priority Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R1	0.168	0.126	0.642	0.266	0.884	0.049	N/A	0.000
R2	0.391	0.877	0.328	0.347	0.701	0.011	N/A	0.089

Table 7.5: Accuracy Scores for Inter-Project Priority Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R0	0.333	0.231	0.548	0.431	0.769	0.527	0.468	0.354
R1	0.178	0.448	0.255	0.413	0.348	0.049		0.207
R2	0.695	0.904	0.672	0.327	0.690	0.387	0.648	0.683

is a promising proof of concept for $Y_{Priority}$ prediction, but there is considerable room for improvement.

7.4.3 Fix Size

The fix size models are trained to predict the number of lines that are likely to be added and deleted as a result from fixing a bug in a particular file ($Y_{ChgLines}$). The skill of these models is measured by median absolute error (MAE). The strength of MAE as a measurement of skill is that it is insensitive to outliers. This is suitable for predictions of $Y_{ChgLines}$ because the majority of bug fixes tend to be quite small, with a handful of very large changes.

The median MAE for intra-project models is 10.4 lines of code, the largest value of error is 35 and the lowest is 6.4. This is once again a promising result for the future of $Y_{ChgLines}$ prediction. The mean $Y_{ChgLines}$ across all projects is 16.4, with an average inter-quartile range of 38 lines. Given the size of a typical bug fixing commit, and the vast degree that they vary, a median error of ten lines is respectable. However, as mentioned above, the MAE measurement smooths out outliers. The median mean squared error (MSE) for the same predictions is 11,748 lines of code. The MSE is drastically larger than the MAE because the models were unable to accurate predict fixes that are several hundred lines

Table 7.6: MAE Scores for Intra-Project Fix Size Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R1	9.679	23.297	7.836	8.321	9.443	6.427	N/A	1016.546
R2	14.660	10.767	9.374	10.204	11.517	35.493	N/A	15.261

Table 7.7: MAE Scores for Inter-Project Fix Size Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R0	240.441	13.049	13.095	15.141	18.837	17.837	22.287	78.673
R1	18.171	19.324	11.938	14.512	13.444	14.174		22.254
R2	17.811	27.750	19.255	12.515	18.639	25.021	18.619	16.139

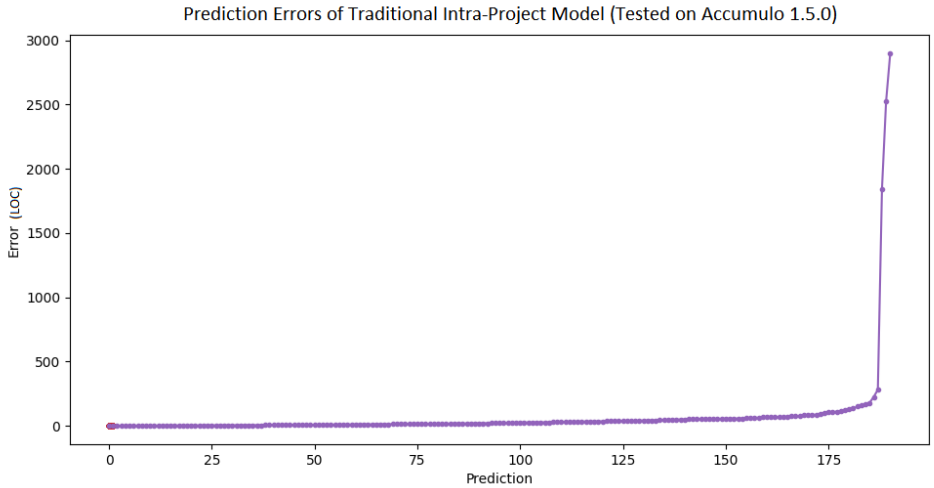

Figure 7.2: This figure illustrates the error of each fix size prediction by the traditional intra-project model tested on Accumulo 1.5.0 data

long. This is illustrated in Figure 7.3, which plots the error of each prediction in ascending order.

The inter-project models for $Y_{ChgLines}$ prediction are much less effective. The median MAE for inter-project models is 18.1 lines of code, the largest median error is 240 lines of code and the smallest error is 11.9 lines of code. It is unsurprising that the inter-project models are less effective given the amount of variance between projects illustrated in the case study in chapter 4. Different projects follow different coding practices, a typical bug fix in one project may be much larger or smaller than a typical bug fix for another. For example, the project Bookkeeper has a mean $Y_{ChgLines}$ of only two lines, whereas the mean $Y_{ChgLines}$ of the Derby project is close to 100. Due to this variance between projects, it is more difficult for a model to predict the size of a bug fix for one project when it was trained on others.

Like the intra-project model, the inter-project model also struggled to accurately predict fix sizes that were several hundred lines long. The median MSE for the intra-project

Table 7.8: MAE Scores for Intra-Project Experience Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R1	124.660	75.367	429.057	289.040	77.311	3.903	N/A	55.232
R2	76.126	64.317	461.283	300.695	143.205	106.759	N/A	20.076

Table 7.9: MAE Scores for Inter-Project Experience Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R0	81.188	186.870	813.737	312.101	232.333	133.937	211.569	89.300
R1	150.780	69.156	1013.879	126.462	176.005	231.834		208.181
R2	100.010	134.091	893.279	178.605	105.259	110.713	130.150	100.985

models is 4521 lines of code. This MSE is quite large, but it is much better than that of the intra-project models. Therefore, while the intra-project models are more effective for predicting the fix size of typical bugs, the inter-project models are able to make more accurate predictions for the largest fix sizes.

7.4.4 Experience

The experience models were trained to predict the number of previous commits that a developer is likely to have made before fixing a bug in a given file. The idea behind this metric is that the number of previous commits could be a good proxy for developer experience, and could be used to determine how much expertise is required to fix a bug. Like the fix-size models, the experience models were scored using median absolute error, because the average number of previous commits can vary widely between files. Results for the intra-project experience models are displayed in Table 7.8, and results for the inter-project experience model are displayed in 7.9.

The results show that the intra-project models are much more accurate than the inter-project models in most cases. The median MAE for intra-project models is 92 commits, and the median MAE for inter-project models is 150 commits. This is unsurprising because the Y_{Exp} values of each project tend to cluster around specific values, where each cluster represents a developer. For example if a junior developer who made 50 commits before a given release fixed 5 bugs, several files would have Y_{Exp} metrics of 50. These reoccurring Y_{Exp} values will be different for different teams of developers, therefore it makes sense that the models trained on data from a single team would be more accurate.

That being said, even though the MAE scores for intra-project models were better, they were still quite poor. The results show that the models have considerable difficulty

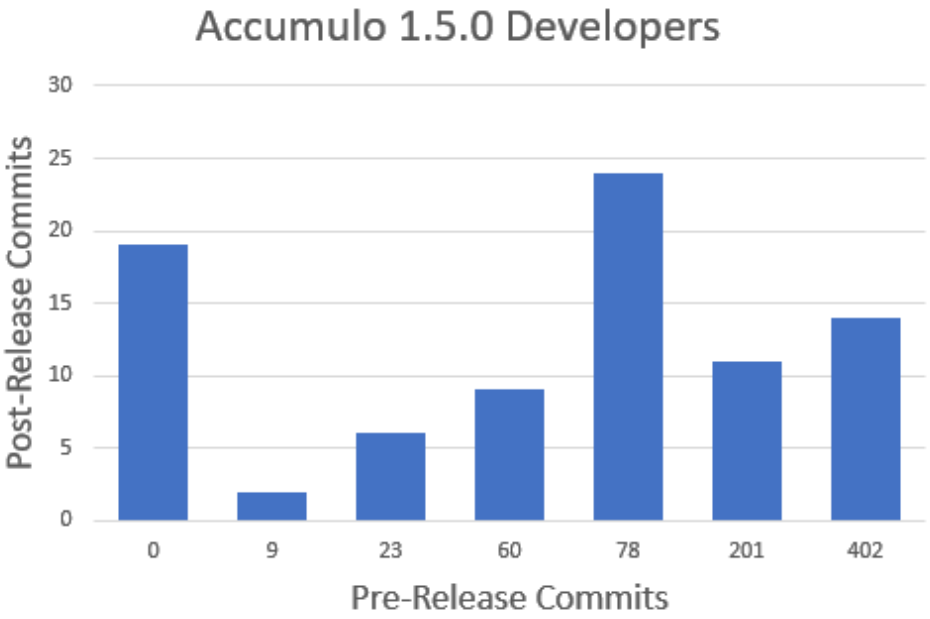

Figure 7.3: This figure illustrates how many commits each developer made in the post-release period of Accumulo 1.5.0, and their various experience levels

predicting which developer would be most likely to fix a given file. For example, there were seven developers who fixed bugs in Accumulo during the post release of 1.5.0. Before the release date each developer had a different number of previous commits: 0, 9, 23, 60, 78, 201, and 402. Figure 7.3 illustrates how many commits each developer made in the post-release period. Most commits in this release were from the developers with less than 80 previous commits; however, the intra-project model that was tested on this data has an MAE of 124. In this project, 124 commits is the difference between a new developer and a veteran, therefore the predictions from this model would have no use in practice.

It is likely that the results for Accumulo 1.5.0 are particularly prone to error because the model had very few buggy files to train from. The model for the subsequent release was trained on three times as much data, and the results are almost twice as accurate. Therefore a major limitation influencing the skill of the experience models is the amount of training data.

7.5 Discussion

This chapter explores whether the independent variables used in traditional bug prediction models can also be used to predict cost metrics. To answer this question, several prediction models were built that use X_{LOC},X_{CC}, and X_{Churn} to predict each cost-related bug metric. While Y_{Exp} proved difficult to predict, the models that predict $Y_{ChgLines}$ and $Y_{Priority}$ showed some promise. While the results were weak to fair, overall it shows lots of room for improvement. It is important to keep in mind that the prediction models in this research question are not intended to beat the state-of-the-art; they were built as a proof of concept to prove the viability of cost-sensitive prediction models.

To gauge the difference in skill between the prototypes in this study and the industry standard, a traditional bug prediction model was built that uses the same three independent variables. The results show that this bug prediction prototype is about half as effective as most models with a full set of features. This provides some insight regarding the potential improvement that can be expected when a full set of independent variables is used. It was shown in RQ3 that the order of importance among independent variables does not typically change even when the dependent variable changes. Therefore, future models that use a complete set of attributes to predict cost metrics may be twice as effective as our prototypes.

This is a promising start to the research of cost-sensitive bug prediction, and provides many promising avenues for future work. The next steps will be to test more classifiers, independent variables, and dependent variables. There are many different classification models that can be tested, such as statistical classifiers, nearest neighbour methods, decision trees, and ensemble methods. Past studies in bug prediction have found that the choice of classifier typically does not have a significant impact [23, 29, 51]. However, this may not be the case for cost metrics. In this study, it is likely that the use of the neural network classifier was detrimental to the overall results. The neural network was selected because the model will be directly compared to the semantic deep learning models in the next chapter. However, due to the limited data set, this likely led to a degree of over fitting that may not have been as acute if another classifier was used.

There are also many independent and dependent variables that can also be evaluated. There are over 50 code metrics that have been used in the past for bug prediction [6, 15, 23, 24, 28, 34], and over 20 process metrics [16, 32, 36]. Ideally, future research would combine many of these metrics to create much stronger cost prediction models. Furthermore, this study only examines three metrics related to cost to as a proof of concept, but the true cost of a bug is likely a function of many more bug attributes.

In summary, the goal of RQ4 was to determine how effective traditional independent variables are for the prediction of bug metrics related to cost. To answer this question, several deep learning models were built that use X_{LOC}, X_{CC}, and X_{Churn} to predict several cost-related metrics including Y_{Exp}, $Y_{ChgLines}$, and $Y_{Priority}$. The results show that $Y_{ChgLines}$ and $Y_{Priority}$ can be predicted with some effectiveness, with lots of room for improvement, but predicting Y_{Exp} is much more difficult. These results will be used as a baseline in the next chapter in order to evaluate how effective semantic learning models are at performing the same task.

Chapter 8

Semantic Learning Model

8.1 Motivation

So far it has been shown that traditional prediction models are capable of predicting bug metrics related to cost with some effectiveness. To evaluate this, file and process metrics were identified that are correlated with the bug metrics being predicted, and this correlation was leveraged to build a prediction model using a neural network. One issue with traditional models is that they assume statistical characteristics can provide everything a model needs to know about a file; however, in reality two files may have exactly the same characteristics but contain completely different code. To address this issue, a second type of model was built that uses the contents of the files themselves to form predictions. Compared to natural language, programs have an explicit syntax which can be represented in an abstract syntax tree (AST) using a relatively small set of tokens. This has been shown to be helpful when it comes to tasks such as code completion and bug detection [1, 57]. In 2018 [57] used a deep belief network (DBN) to create semantic representations of programs for file level defect prediction. The semantic representations of programs in this study were constructed from tokens in the original files, and outperformed state-of-the-art defect prediction models that use traditional features [57]. Semantic representations of software can also be derived from the structure of a program. Code2Vec is a neural model for representing snippets of code as continuous distributed vectors [1]. By decomposing a method into a collection of paths in its AST, Code2Vec is able to distill the semantic meaning of a method into a fixed length vector [1]. The original Code2Vec model was trained to predict a method's name from a semantic representation of its body and is over 75% more accurate than other models that were trained to perform the same task. Code2Vec

is also open-source and encourages the use of its code vectors for other applications. The purpose of this research question is to use semantic representations of code produced using Code2Vec to predict file level bug metrics, and compare the results with the traditional model. The intuition is that the semantic representations will provide insight into each file from a perspective that is unattainable using traditional techniques. Furthermore, by forming predictions for a variety of bug metrics related to cost, the goal is to provide more useful insights than models that assume that all bugs are the same.

8.2 Summary of Code2Vec

Code2Vec is one of the most recent neural models for representing snippets of code as distributed vectors [1]. It was created by researchers from Technion and Facebook in late 2018 and represents the state-of-the-art for research about semantic representations of code. The goal of code2vec is to produce low dimensional vector representations of code snippets called embeddings for use in machine learning pipelines such as code retrieval, classification, and tagging. In other words, given a snippet of source code, code2vec produces a vector that captures the semantic "meaning" of that snippet, with snippets that perform similar functions mapped close to one another. The code2vec path-attention model receives as input a code snippet in some programming language and a parser for that language [1]. A mathematical representation of that code snippet must then be created so that it can be fed into a learning model. This is achieved by representing a snippet of source code as a set of path contexts. Each path context is comprised of three values, $\langle xs, p, xt \rangle$, where xs is a starting node in the AST, xt is a leaf node, and p is the set of nodes that connects them. For example, a possible path context for the statement "x=7;" would be: $\langle x, (NameExprAssignExprIntegerLiteralExpr), 7 \rangle$. Given a code snippet, C, and its AST, first all pairs of terminal nodes are identified. C is then represented as the set of all path contexts that can be derived from each pair of terminal nodes.

During training, the code2vec model learns embeddings for two vocabularies: value_vocab and path_vocab [1]. These are matrices in which every row corresponds to a certain terminal node or path. The values of these matrices are initialized randomly and are learned simultaneously with the network during training. When a bag of path contexts is fed into the network, each component is looked up and mapped to its corresponding embedding. Then three embeddings of each path-context are concatenated to a single context vector.

Next, the context vector is compressed into a combined context vector using a fully connected layer [1]. The fully connected layer converts a context vector of size 3d into a combined context vector of size d by multiplying it with a weighted matrix. This is

done separately for each context vector so that the model can give different attention to every combination of paths and values. This way, the model can give a certain path high attention when observed with certain values, and low attention when the same path is observed with different values.

Finally, the multiple combined context vectors are aggregated into a single vector representation using an attention vector [1]. The attention vector is initialized randomly and learns to apply a scalar weight to each context vector simultaneously with the rest of the network. The aggregated code vector v, which represents the whole code snippet, is a linear combination of the combined context vectors factored by their attention weights. This aggregated code vector is used for the final tag prediction; in the code2vec paper, the tags being predicted are the method names corresponding to each code snippet. The network is trained using a gradient descent algorithm and the standard approach of backpropagating the training error through each of the learned parameters.

The performance of the code2vec model for the task of predicting method names is very exciting. The model was compared to three other recently proposed models that address similar tasks: CNN+Attention by Allamanis et al. [2016], LSTM+Attention by Iyer et al. [2016], and Paths+CRFs by Alon et al. [2018]. In their quantitative analysis, code2vec outperformed every other model, with F1 measures nearly doubling the competition using small datasets and improving the next best model by 20% when trained on a large amount of data. Furthermore, the code2vec model is able to make predictions 100 times faster than the next best model.

The code2vec researchers encourage the community to use their code embeddings for other deep learning tasks such as finding similar programs, code search, and code suggestion. The entire source code for the model is free to download online, and so is a pre-trained version of the model which was optimized for method name prediction. Given the limited scope of this project, we chose to use the pre-trained version of the code2vec model. The intuition behind this decision was that if a vector captures enough semantic information to predict a method name, the same semantic information can be used for preliminary research into bug prediction. A good project for future research in this area would be to modify and retrain the entire code2vec model and compare the results.

8.3 Data Collection

The data for the code2vec model was collected from the same projects and releases as the traditional model so that the results can be directly compared. As described in the previous

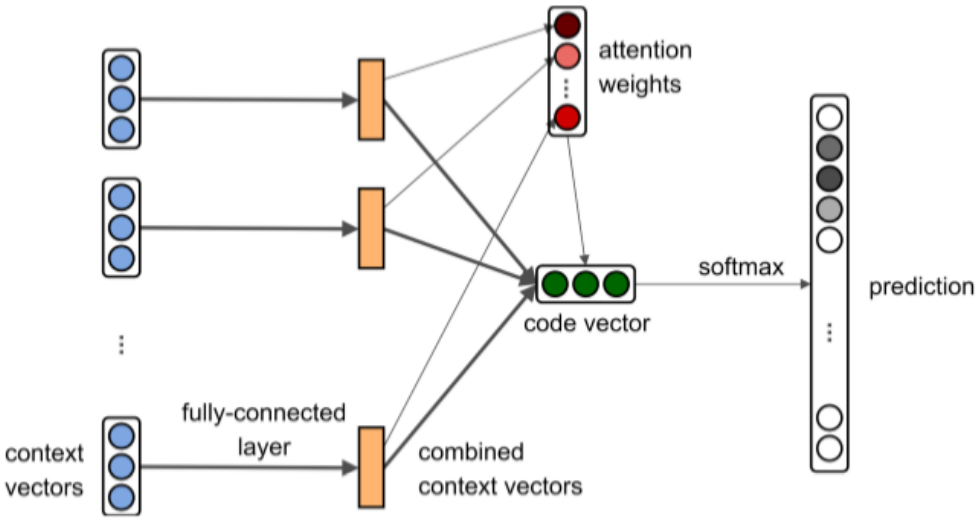

Figure 8.1: Code2Vec Path Attention Network Architecture. Figure reprinted from [1]

section, the metrics for the traditional model were collected at the file level using each project's Git history and issue tracking data. A detailed explanation of the data collection process for this model is described in section 0. The final dataset for the traditional model includes metrics for every file of every target release; however, code2vec is designed to produce vectors of individual methods. Therefore, the dependent variables needed to be recollected at the method level.

The end goal of the data collection process is to build a dataframe that contains a row for every method and the following columns: code2vec vector, Y_{Buggy}, $Y_{ChgLines}$, $Y_{Priority}$, and Y_{Exp}. Two of the variables, $Y_{Priority}$ and Y_{Exp}, are the same at the file and method level. However, the other two variables – Y_{Buggy}, and $Y_{ChgLines}$ – had to be recalculated.

The code2vec source code is designed to take as input an entire Java file, identify each method in the file, and output a vector representation for each method along with its method name. Therefore, an easy way to build a dataset of every method in a project is to pass every file in the project through code2vec and collect the results. This would produce a dataframe with vectors for every method, where each method can be identified by its method name and file name. The issue with this approach is that the method and file names do not create a unique identifier for each method, because a file can have two methods with the same name. Without a unique identifier for each method, the metrics of Y_{Buggy}, and $Y_{ChgLines}$ cannot be accurately assigned to each vector.

To get around this issue, the buggy methods of each file were identified using a Python

script and saved in separate files. The Python script takes as input the git diff between each buggy commit and the commit that preceded it. The diff output contains the source code of every file modified by that commit. Lines that were removed by the commit start with a '-' and lines that were added start with a '+'. The script uses a regular expression to identify every method declaration in the file and counts the number of curly brackets following each method declaration to identify the beginning and end of each method body. If a method contains a line that starts with a '-' or a '+', that line was modified to fix a bug. The 'buggy' version of a method can be reconstructed by removing the lines added by the commit. Likewise, the 'fixed' version of the method can be reconstructed by removing the lines that were removed by the commit. The $Y_{ChgLines}$ for each method is equal to the total number of lines added and removed. The Python script outputs every buggy method in separate files, and writes $Y_{ChgLines}$ and the commit hash directly into each file name.

These files – which contain one buggy method each – are then passed into code2vec. The code2vec source code was modified to output a vector, name, fix size, and hash for each method. The final two metrics, $Y_{Priority}$ and Y_{Exp}, can then be determined for each method using the commit hash and the bug fixing commit data from the previous sections. After collecting the data for every buggy method, only the data for the clean methods remained to be collected.

While the buggy methods were collected at different points throughout each post-release period, the clean methods were all collected from the files as they were on the release date. This was done using a clone of each project at each target release. The files in each clone were modified so that every method that was identified as buggy in the previous step was deleted from the source code. This produced a version of each release that only contained clean methods. Since all of the methods in the modified files are clean, the metrics Y_{Buggy}, $Y_{Priority}$, $Y_{ChgLines}$, and Y_{Exp} are all zero. Therefore, these files were simply passed through code2vec, and the vector encodings were added to the final dataset with every other column set to zero.

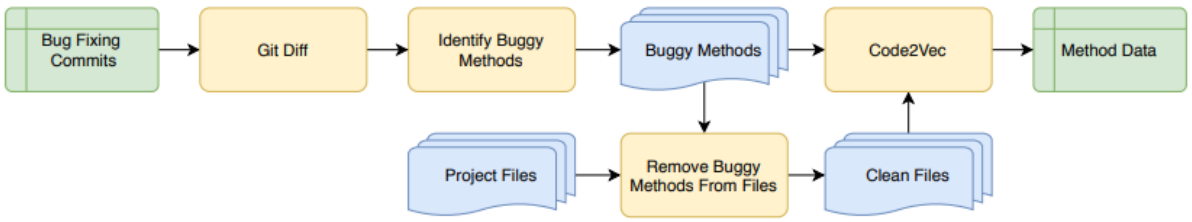

Figure 8.2: Method-Level Data Collection

8.4 Model Design

The purpose of the code2vec model is to make predictions for the same bug metrics as the traditional model using semantic information so that the two models can be compared. Since the code2vec data is collected at the method level, this is done in two steps. First, various models are trained to predict the target bug metrics for each method. A different model is used for each metric. Each model has an input size of 384 and 3 layers of rectified linear units with 100 nodes each. The output layers differ depending on the variable being predicted and are detailed in Table 7.1. As with the traditional models, each model is trained in two ways: inter-project and intra-project. The inter-project models are trained on data from all projects except one, and the intra-project models are trained on data from one release of one project. Each model was trained for 100 epochs. While the bugginess models were trained on all methods, the other models were only trained on buggy methods. This is because the other variables only exist for buggy methods.

Once the method-level models were trained, they were used to produce predictions for every method in the dataset. Intra-project predictions were made by models trained on the previous release of the same project, and inter-project predictions were made by models trained on the same release of every other project. Next, the predictions were sorted into groups based on the file that each method originated from. Each group amounts to a data frame where every row represents a different method in the same file of the same release, and whose columns include inter-project predictions and intra-project predictions of every variable. The predictions in each group were then combined into vectors to use as input for the file-level prediction models. The values in each vector are in descending order based on the predicted Y_{Buggy} for that method. This ordering provides context as to which predictions are most likely to be derived from a buggy method. The models require every input vector to be the same length, therefore the vectors were either trimmed or padded with zeros in order to have a size of 100. The length of 100 was selected so that the majority of files would have all of their methods accounted for in their corresponding vector. The models that make the file level predictions are very similar to those that make the method-level predictions. Each model has an input size of 100 and 3 layers of rectified linear units with 100 nodes each. The output layers detailed in Table 7.1 are the same, and each model is trained separately for inter and intra project data.

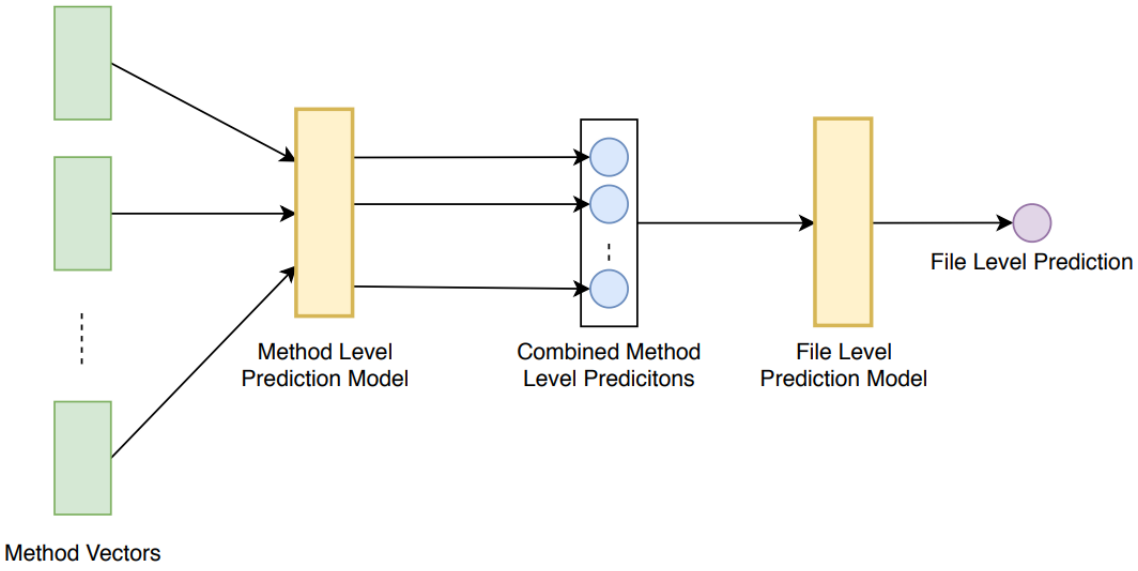

Figure 8.3: File-Level Prediction Model

8.5 Results

The results for the models that were trained to predict bug metrics from a file's semantic structure were less effective overall than the traditional approach.

8.5.1 Bugginess

The bugginess models are trained to output the probability that a file has a bug. The effectiveness of these models are evaluated by measuring the area under the precision-recall curve (AUC) for each release. The AUC scores for the intra-project model are displayed in Table 8.1 and the AUC scores for the inter-project model are displayed in Table 8.2.

The AUC scores show that both the intra-project and inter-project models were unable to produce meaningful predictions using the semantic representations of code2vec. The median AUC for the intra-project models is 0.167 and the median AUC for the inter-project models is 0.140. For context, Figure 8.4 displays the precision-recall curve for Accumulo 1.5.0 which has an AUC of 0.163. The orange line in Figure 8.4 indicates the precision and recall values of the model at each threshold, and the dotted blue line represents the precision-recall curve for a model with no skill.

These results show that the models that were trained to make bug predictions using semantic representations of each file's methods have almost no skill. While this is a negative result, this does not mean that code2vec cannot be used in bug prediction. This prototype could be failing to detect buggy files for any number of reasons, including how code2vec was trained, or how the method data was used to make file-level predictions. The takeaway from this result is that using code2vec for bug prediction will require additional research and experimentation.

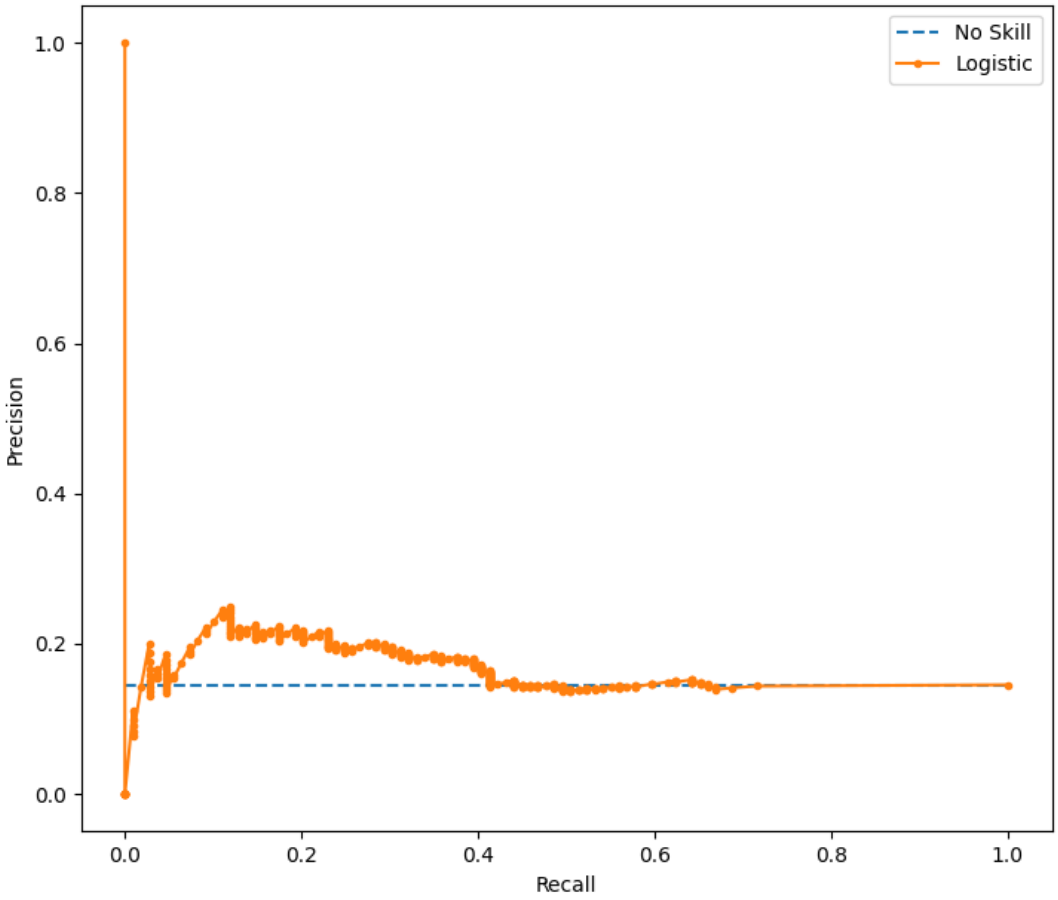

Figure 8.4: Precision-Recall Curve of Intra-Project Bugginess Prediction Model Using code2Vec for Accumulo Release 1.5.0. (AUC score = 0.163)

Table 8.1: AUC Scores for Intra-Project Semantic Bugginess Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R1	0.163	0.290	0.073	0.154	0.064	0.599	N/A	0.102
R2	0.172	0.217	0.026	0.208	0.074	0.226	N/A	0.170

Table 8.2: AUC Scores for Inter-Project Semantic Bugginess Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R0	0.093	0.125	0.140	0.192	0.115	0.153	0.023	0.135
R1	0.211	0.229	0.130	0.252	0.075	0.595		0.057
R2	0.194	0.296	0.036	0.247	0.045	0.205	0.115	0.142

8.5.2 Priority

The priority models are trained to predict the priority of a bug in a given file ($Y_{Priority}$) out of 5 categories. The effectiveness of the priority models is measured by the number of correct predictions divided by the total number of predictions. The accuracy results for the models trained on intra-project data are displayed in Table 8.3 and the accuracy results for the models trained on inter-project data are displayed in Table 8.4.

Like the bug prediction models, the priority models were unable to successfully predict $Y_{Priority}$. The median accuracy for the intra-project models is 0.148 and the median accuracy of the inter-project models is 0.187. For context, a model that predicts a random category every time would have an average accuracy of 0.2. This shows that the models were unable to find a connection between the vector representations of methods and $Y_{Priority}$.

8.5.3 Fix Size

The fix size models were trained to predict the number of lines that would likely be changed to fix a bug in a given file ($Y_{ChgLines}$). The skill of the fix size models is measured using

Table 8.3: Accuracy Scores for Semantic Intra-Project Priority Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R1	0.152	0.500	0.141	0.053	0.149	0.014	N/A	0.393
R2	0.147	0.125	0.298	0.103	0.048	0.523	N/A	0.632

Table 8.4: Accuracy Scores for Semantic Inter-Project Priority Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R0	0.314	0.154	0.187	0.208	0.245	0.132	0.457	0.070
R1	0.230	0.207	0.548	0.107	0.209	0.020		0.321
R2	0.201	0.167	0.175	0.072	0.143	0.061	0.397	0.137

Table 8.5: MAE Scores for Semantic Intra-Project Fix Size Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R1	21.6	10.5	5.9	10.3	8.0	36.2	N/A	22.0
R2	15.7	10.1	11.0	9.0	18.3	31.0	N/A	13.2

median absolute error. Table 8.5 contains the results for the intra-project fix size models and Table 8.6 contains results for the inter-project fix size models.

Compared to the other semantic models, the fix size models show some skill in predicting $Y_{ChgLines}$. The median MAE for intra-project models is 12.1 lines of code and the median MAE for inter-project models is 15.2 lines of code. Recall that the same results for the traditional prediction models were 10.4 and 18.1 respectively. Therefore while the traditional model is slightly more accurate when trained on intra project data, the semantic model is more accurate when trained on inter-project data.

To better understand these results, it helps to consider a typical distribution of $Y_{ChgLines}$. For example, Figure 8.5 is a histogram which shows the distributions of $Y_{ChgLines}$ in Accumulo 1.6.0. The histogram shows that most bug fixes change less than ten lines, but it is very common for files to change up to 100 lines of code. The average $Y_{ChgLines}$ in Accumulo 1.6.0 is 67. With this in mind, the intra-project fix size prediction model with a MAE of 12.1 shows some degree of skill. While its predictions are not precise, the model can broadly determine whether a change will be large or small. However, like the traditional models, both semantic learning models struggled to accurately predict the largest fix sizes. The MSE for the intra-project model is 28860 lines of code, and the MSE for the inter-project model is 165650 lines of code.

8.5.4 Experience

The experience models were trained to predict the number of previous commits that a developer is likely to have made before fixing a bug in a given file (Y_{Exp}). The experience models were scored using median absolute error. Results for the intra-project experience

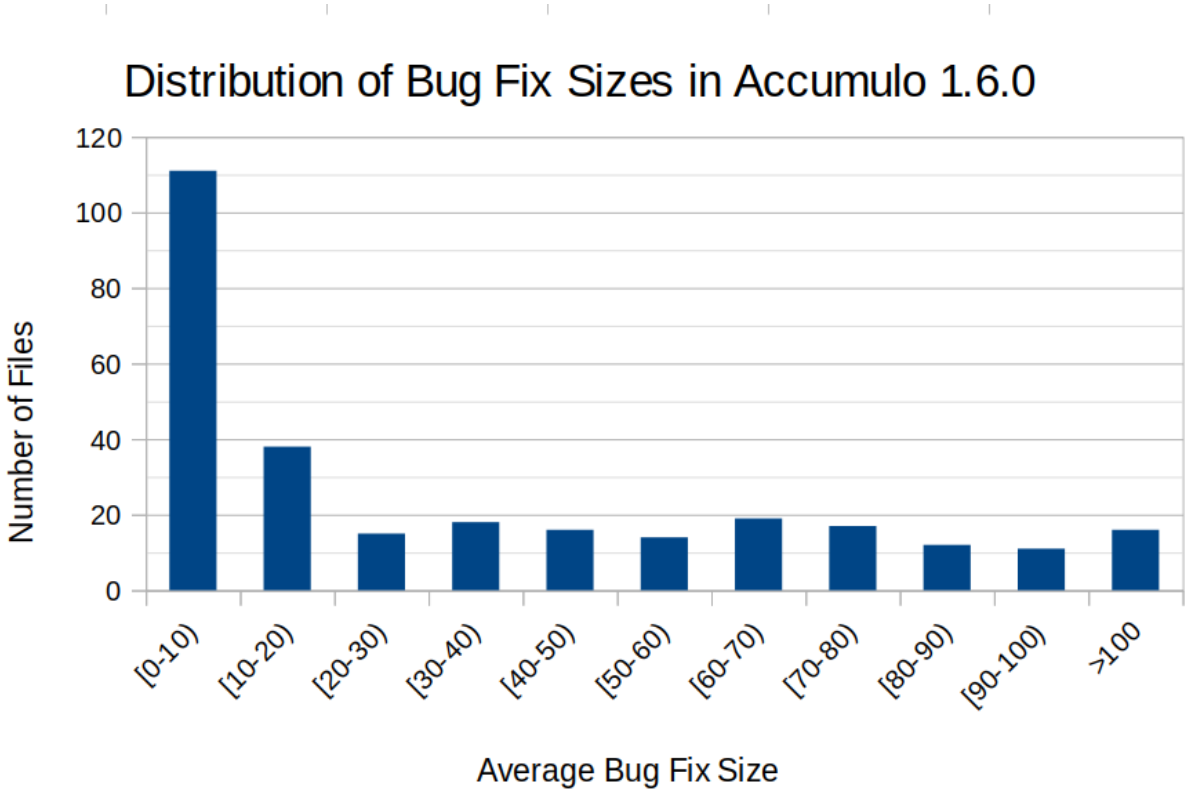

Figure 8.5: Histogram of the Bug Fixes Sizes for Files in Accumulo 1.6.0

models are displayed in Table 8.7, and results for the inter-project experience model are displayed in 8.8.

Once again, the results show that the models have considerable difficulty predicting which developer would be most likely to fix a given file. The median MAE for the intra-project model is 104 previous commits and the median MAE for the inter-project model is 172 previous commits. These results are similar to the traditional experience models in Chapter 7. It makes sense that the intra-project models are more accurate because they were trained and tested on data from the same set of developers. However, the intra-project models have an MAE of over 100 previous commits, which can be the difference between an intermediate developer and a veteran. Therefore, both of the models were unsuccessful at using semantic information to predict Y_{Exp}.

Table 8.6: MAE Scores for Semantic Inter-Project Fix Size Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R0	18.0	11.8	24.8	11.0	13.3	19.1	33.1	30.0
R1	39.1	16.3	15.2	15.6	13.8	14.6		22.0
R2	14.4	13.4	22.1	17.1	12.1	12.8	15.1	12.0

Table 8.7: MAE Scores for Semantic Intra-Project Experience Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R1	278	56	459	233	104	4.7	N/A	63
R2	104	64	431	205	132	101	N/A	31

8.6 Discussion

The semantic learning prototype was ultimately unsuccessful at predicting bug metrics; however, this does not mean that semantic learning cannot be harnessed for cost-sensitive bug prediction. There are many ways that the models in this study could be improved. For example, code2vec could have been leveraged to produce more meaningful encodings for the given task. Code2vec is a public model that was initially trained to predict method names[1]. It works by creating a vector representation of the AST paths of a method, and using that vector to make predictions about the method[1]. The developers of code2vec have encouraged researchers to use their network for tangential tasks. Researchers can either download an uncalibrated version of the code2vec model, or a model that was pre-trained for method name prediction. Due to time and resource limitations, this study uses the pre-trained network. The rationale behind this decision was that the encodings designed to predict method names may contain enough "meaning" to predict bug metrics. The results from this study indicate that this assumption was false.

Code2vec would likely be more successful for bug prediction if the network was re-weighted for every specific metric. The code2vec model has several components that are all designed to be calibrated simultaneously during training[1]. This includes the vectors

Table 8.8: MAE Scores for Semantic Inter-Project Experience Model

	Accumulo	**Bookkeeper**	**Camel**	**Cassandra**	**CXF**	**Derby**	**Hive**	**OpenJPA**
R0	101	182	850	237	172	187	80	222
R1	278	126	1017	147	214	276		54
R2	91	100	948	195	142	84	101	73

that create embeddings for AST paths and tokens, the fully connected layer that produces context vectors, and the attention vector[1]. Retraining code2vec for each metric would optimize every layer of the code2vec network to produce vectors that are specially designed to predict cost metrics. This would be a good area for future research.

Furthermore, code2vec is only one type of semantic learning model; there are many ways that semantics can be utilized for machine learning models. There are already semantic learning models that have successfully been used for defect prediction[57]. One such model creates semantic representations using AST tokens and a deep belief network, and outperforms traditional features in both inter-project and intra-project file level defect prediction[57]. Therefore, even if retraining code2vec for specific metrics is unsuccessful, there are other types of semantic models that can be tested.

In RQ5, we set out to determine whether code2vec can be leveraged to predict bug metrics as effectively as a traditional model. In the end, our prototypes were not as effective as the traditional models; however, there is still potential for cost-sensitive semantic bug prediction models in the future.

Chapter 9

Threats to validity

9.1 Internal Validity

As mentioned in Section 8.2, the code2vec model can be downloaded with random weights, or pre-trained for method name prediction. Due to the scope of this project, the pre-trained models were used. The hope was that the vectors used for method name prediction could be used for the tangential task of bug prediction. The results indicate that this is not the case, however, the effectiveness of code2vec for cost sensitive bug prediction when the whole model is re-weighted remains undetermined.

The results of the bug prediction models only reflect the predictive capabilities of the independent variables that were used as input. This study uses three independent variables that were selected based on their popularity in other studies. There are, however, many other strong independent variables that were not used. Furthermore, the two code metrics that were used, lines of code and cyclomatic complexity, have been shown in other studies to be correlated with one another. It is possible that the models would been more successful if different independent variables were selected.

The effectiveness of the prediction models is also dependent on their architecture and training. The models in this study each have three hidden layers with ReLu activation functions, and were trained for 100 epochs. It is possible that the results could be improved by modifying the model architecture; for example, by increasing the number of hidden layers, trying different activation functions, or training for a greater number of epochs.

9.2 External Validity

The projects in this study were all previously used in bug prediction research, and cover a wide range of domains [8, 45, 46]. That being said, software projects can vary widely in terms of their development processes and the software stack they use. Furthermore, the effectiveness of a bug prediction model is heavily dependent on the amount of data that can be collected from a project. Poor issue tracking or inconsistent practices for ranking bugs may have a considerable impact on the results. The results from this study may not generalize to all projects.

The models that were trained to predict bug attributes rely on data from buggy files. Most of the projects had fewer than 100 buggy files in each release. This likely led to some over fitting, particularly in the intra-project models. The inter-project models were typically trained on about seven times more data than the intra-project models, because they are not limitted to data from one project. Therefore, some differences in skill between intra and inter project models may not generalize.

9.3 Construct Validity

The tools, programming libraries, and research methods used in this study have been used for bug prediction in the past; however, they are not perfect. For example, the files in each release were classified as buggy or clean based on whether or not they were involved in a bug fixing commit. It is possible that not all changes in a commit were made in order to fix a bug. This could lead to some files being miscategorized.

The methods for collecting bug attributes could also be a source of error. The issue tracking information extracted from JIRA was initially created by human developers. The bug classifications and priority estimations are subjective and are not based on any specific rules or requirements. It is therefore possible that the priority classifications are not always accurate. For example, a develop may misclassify a bug's priority in JIRA due to external pressures such as release pressure. Similarly, Herzig et al. [18] estimates that 33% of the issues in JIRA that are classified as bug reports are not bug reports. To combat this, all bug fixing commits that did not contain bug related language in their commit message were omitted.

The experience of a developer is calculated based on the number of previous commits that developer has made in the given project. Prior studies have used similar measures for experience [4, 8, 19, 30, 43]. It is possible that an experienced developer could be given a low

experience score if that developers experience was primarily gained from other projects. Since developer experience is being used as a proxy for cost based on the correlation between experience and wages, this source of error could affect the practical applications of this metric. Moreover, this study assumes that the bugs in the studied projects are always fixed by a developer of appropriate experience. This may not be the case for all bugs, for example, some trivial bugs may have fixed by experienced developers.

We used bug-fix size as an approximation of fixing effort and bug-fixing cost. In the past, lines of code has been used to measure how much work a developer can do in a unit of time [49]. This metric may not always correlate with the effort, or it might not reflect the real fixing-cost. For example, if a developer copies code from another source to fix a bug, their fixing effort will be inflated. Moreover, it is also assumed that every change made to fix a bug is exactly the size that it should be. If a developer fixes a bug with an unnecessary number of lines of code, the fixing effort will be greater than it should be.

Chapter 10

Conclusion

Bug prediction has been a popular area of research for over a decade. This is not surprising, since software companies are currently spending large amounts of money on QA in order to keep the perpetual influx of defects at bay. Most organizations do not have a full SQA department because of the overhead costs, choosing instead to delegate testing responsibilities amongst their developers[7]. Therefore, the prospect of a bug prediction model which can cut a company's QA costs helping them spend their limited testing resources more efficiently is very appealing.

At this point, thousands of different bug prediction prototypes have been tested that use every imaginable combination of classifiers and attributes. The leading models are able to predict defective entities with impressive accuracy, and bug prediction models are beginning to be put to use in industry. However, while traditional bug prediction models can predict bug density with considerable accuracy, they do not differentiate between the bugs that they predict. This leaves an area for improvement which we explored in this paper. If bug prediction model results contained another layer of granularity relating to each bug's attributes, they may be more useful for users in practice.

This paper performs an in-depth analysis of three bug metrics related to cost — priority, fix size, and developer experience — and how they can be used to improve even the strongest bug prediction models. It is shown that bugs can differ greatly along each of these metrics, and they can be used to sort buggy entities in terms of QA cost. This is evidence that if bug prediction models considered these metrics, they could refine their predictions by indicating which bug-prone entities should take priority.

The second half of this study evaluates two methods of predicting bug attributes. First, it was evaluated whether the cost metrics can be predicted by the same models that predict

bugginess. A handful of code and process metrics that are commonly used in bug prediction models were evaluated in their ability to predict the auxiliary cost metrics. It was shown that the correlation between traditional independent variables and cost metrics is similar to the correlation with the number of bugs in an entity. As a proof of concept, several prototype bug prediction models were constructed and trained to predict cost metrics using traditional predictors. Although the prototypes did not perform as well as the strongest bug prediction models, they were still able to predict cost metrics with moderate accuracy. Therefore if the leading traditional models were put to the same task, we consider that there is a good chance that they could predict the cost metrics necessary to vastly improve the applicability of their results.

Finally, the traditional models were used as a baseline with which to evaluate semantic models that were trained to perform the same task. The semantic models use code2vec to produce vector representations of method semantics with which to form predictions. In the end, the semantic models were unsuccessful; however, the investigation into semantic learning revealed some promising new areas of exploration. Overall, there is an abundance of evidence to suggest that cost-sensitive bug prediction models are possible, and many different avenues for future researchers to explore.

10.1 Summary

RQ1 Is there evidence to support the claim that not all bugs are the same?

> Yes. Bugs show considerable differences in terms of their priorities, fix sizes, and experience of the developers who fix them.

RQ2 What is the impact of bug prediction models assuming that all bugs are the same?

> When buggy entities are sorted by different cost metrics, the top entities change considerably. Furthermore, there is little correlation between different cost metrics. Therefore, users may be likely to miss the most important entities depending on their needs.

RQ3 Are traditional predictors of bugs correlated with unconventional bug metrics that are related to cost?

Traditional independent variables used in bug prediction models, including lines of code, cyclomatic complexity, and churn, are correlated with all cost metrics to some degree. However, the correlation is not as strong as with the number of bugs in an entity, and varies depending on the project.

RQ4 How effective are traditional independent variables for predicting unconventional bug metrics related to cost using a deep learning model?

Prototype deep learning models that use traditional independent variables were able to predict cost metrics with some effectiveness. While developer experience was difficult to predict, the code and process metrics showed a moderate ability to predict the priority and size of a bug.

RQ5 How effective is semantic learning for predicting bug metrics related to cost compared to traditional models?

The prototype semantic learning models built using code2vec were unable to predict the target metrics, apart from fix size. However, the study revealed areas of improvement that could be applied to semantic learning models for cost-sensitive bug prediction in the future.

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[2] Erik Arisholm and Lionel C Briand. Predicting fault-prone components in a java legacy system. In *Proc. of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06*, page 8, New York, New York, USA, 9 2006. ACM Press.

[3] Erik Arisholm, Lionel C. Briand, and Eivind 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, 1 2010.

[4] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, page 4, New York, New York, USA, 9 2011. ACM Press.

[5] David Bowes, Tracy Hall, and Jean Petrić. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, 26(2):525–552, 2018.

[6] Mike Chapman and Dan Solomon. The relationship of cyclomatic complexity, essential complexity and error rates. In *Proceedings of the NASA Software Assurance Symposium, Coolfont Resort and Conference Center in Berkley Springs, West Virginia*, 2002.

[7] Murali Chemuturi. *Mastering software quality assurance: best practices, tools and techniques for software developers*. J. Ross Publishing, 2010.

[8] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. An empirical study of dormant bugs. In *Proc. of the 11th Working Conference on Mining*

Software Repositories, volume undefined, pages 82–91, New York, New York, USA, 5 2014. ACM Press.

[9] Christian Dustmann and Costas Meghir. Wages, experience and seniority. *The Review of Economic Studies*, 72(1):77–108, 2005.

[10] Chelsea Frischknecht. Software fail watch: 2016 in review, 2016.

[11] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proc. of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.

[12] Todd L. Graves, Alan F. Karr, James S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 7 2000.

[13] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, page 495, New York, New York, USA, 2010. ACM Press.

[14] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[15] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[16] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proc. - International Conference on Software Engineering*, pages 78–88. IEEE, 5 2009.

[17] Ahmed E. Hassan and Richard C. Holt. The top ten list: dynamic fault prediction. In *21st IEEE International Conference on Software Maintenance*, pages 263–272. IEEE, 2005.

[18] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *2013 35th International Conference on Software Engineering*, pages 392–401. IEEE, 5 2013.

[19] Huzefa Kagdi, Maen Hammad, and Jonathan I. Maletic. Who can help me with this source code change? In *2008 IEEE International Conference on Software Maintenance*, pages 157–166. IEEE, 9 2008.

[20] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 9 2010.

[21] Sunghun Kim, E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 3 2008.

[22] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proc. of the 2006 international workshop on Mining software repositories - MSR '06*, page 119, New York, New York, USA, 5 2006. ACM Press.

[23] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[24] Thomas J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[25] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2018.

[26] Thilo Mende and Rainer Koschke. Effort-Aware Defect Prediction Models. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 107–116. IEEE, 3 2010.

[27] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, page 13, New York, New York, USA, 11 2008. ACM Press.

[28] Tim Menzies, Justin DiStefano, Andres Orrego, and Robert Chapman. Assessing predictors of software defects. In *Proc. Workshop Predictive Software Models*, 2004.

[29] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[30] Audris Mockus and James D. Herbsleb. Expertise browser. In *Proc. of the 24th international conference on Software engineering*, page 503, New York, New York, USA, 5 2002. ACM Press.

[31] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 8 2000.

[32] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. of the 13th international conference on Software engineering*, page 181, New York, New York, USA, 2008. ACM Press.

[33] Sebastian C Müller and Thomas Fritz. Using (bio) metrics to predict code quality online. In *Software Engineering, 2016 IEEE/ACM 38th International Conference on*, pages 452–463. IEEE, 2016.

[34] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 580–586. IEEE, 2005.

[35] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005.*, page 580, New York, New York, USA, 5 2005. ACM Press.

[36] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005.*, pages 284–292. IEEe, 2005.

[37] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proc. of the 28th international conference on Software engineering*, page 452, New York, New York, USA, 5 2006. ACM Press.

[38] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 0 1996.

[39] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[40] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Re-evaluating method-level bug prediction. In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 592–601. IEEE, 2018.

[41] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, page 2, New York, New York, USA, 11 2008. ACM Press.

[42] Paweł Piotrowski and Lech Madeyski. Software defect prediction using bad code smells: A systematic literature review. In *Data-Centric Business and Applications*, pages 77–99. Springer, 2020.

[43] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects. In *Proc. of the 33rd international conference on Software engineering*, page 491, New York, New York, USA, 5 2011. ACM Press.

[44] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 432–441. IEEE, 2013.

[45] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 1, New York, New York, USA, 11 2012. ACM Press.

[46] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. Sample size vs. bias in defect prediction. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, page 147, New York, New York, USA, 8 2013. ACM Press.

[47] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel German, and Jesús M González-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 2019.

[48] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M González-Barahona. What if a bug has a different origin? making sense of bugs without an explicit bug introducing change". In *Proc. of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, United States, 10 2018. Association for Computing Machinery, Inc.

[49] Rossi, Chuck and Shibley, Elisa and Su, Shi and Beck, Kent and Savor, Tony and Stumm, Michael. Continuous deployment of mobile software at facebook (showcase). In *Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 12–23. ACM, 2016.

[50] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proc. of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06*, page 18, New York, New York, USA, 9 2006. ACM Press.

[51] Martin Shepperd, David Bowes, and Tracy Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[52] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto. Predicting Re-opened Bugs: A Case Study on the Eclipse Project. In *2010 17th Working Conference on Reverse Engineering*, pages 249–258. IEEE, 10 2010.

[53] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, page 300, New York, New York, USA, 9 2011. ACM Press.

[54] Piotr Tomaszewski, Jim Håkansson, Håkan Grahn, and Lars Lundberg. Statistical models vs. expert estimation for fault prediction in modified code – an industrial case study. *Journal of Systems and Software*, 80(8):1227–1238, 8 2007.

[55] Harold Valdivia-Garcia and Meiyappan Nagappan. False Negative in Defect Prediction Appendix. *http://people.rit.edu/hv1710/false-negative-dp-appx.pdf*, 2016.

[56] Harold Valdivia-Garcia, Emad Shihab, and Meiyappan Nagappan. Characterizing and predicting blocking bugs in open source projects. *Journal of Systems and Software*, 143:44–58, 2018.

[57] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[58] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Xiaohu Yang. Elblocker: Predicting blocking bugs with ensemble imbalance learning. *Information and Software Technology*, 61:93–106, 1 2015.

[59] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Bo Zhou. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering*, pages 75–109, 3 2015.

[60] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs. In *Proc. of the 8th working conference on Mining software repositories - MSR '11*, page 93, New York, New York, USA, 5 2011. ACM Press.

[61] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. of the 13th international conference on software engineering*, page 531, New York, New York, USA, 2008. ACM Press.

[62] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *2012 34th International Conference on Software Engineering*, pages 1074–1083. IEEE, 6 2012.

[63] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering*, pages 9–9. IEEE, 5 2007.