Scalable Deep Learning for Bug Detection

Rafael-Michael Karampatsis

Doctor of Philosophy
Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh
2020

Dedicated to my
brother Elias
and
to my lovely parents
for encouraging me
to never stop
pursuing my dreams

Declaration

I declare that this thesis has been composed by myself and that this work has not been submitted for any other degree or professional qualification. I confirm that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included. My contribution and those of the other authors to this work have been explicitly indicated at the beginning of each chapter. I confirm that appropriate credit has been given within this thesis where reference has been made to the work of others.

Rafael-Michael Karampatsis
2020

Acknowledgements

I would like to sincerely thank my enthusiastic supervisor Charles Sutton for his kindness, guidance and support throughout the duration of this thesis. I owe him a huge debt of gratitude for teaching me most of what I know now, and giving me the freedom to delve into new subjects in software engineering, natural language processing, machine learning and statistics. Most of all I would like to thank him for the extreme patience and constant encouragement that he showed during supervising me. Many thanks also to my second supervisor Mirella Lapata for her insightful comments.

I would also like to thank Kenneth Heafield, Rico Sennrich, and Ajitha Rajan for their constructive comments and assisting me with my annual reviews. Moreover, I would like to extend my appreciation to my collaborators Hlib Babii, Romain Robbes, and Andrea Janes; with whom I extremely enjoyed working with and it was one of the best collaborations I've experienced. A special thanks goes to the staff and fellow students of the CDT, for creating such a warm atmosphere, it was a pleasure to be part of this community. However, out of all the CDT students I would like to thank Andreas for his everyday support, amazing company, and endless chats over coffee from the very first day to the very last as well as Matthew for his support in difficult situations.

I am grateful to have been able to work with the wonderful group of people that my supervisor has assembled in the CUP group over these last four long years; Akash, Annie, Irene, Kai, Lazar, Maria, Miltos, Ryan, Simao, and Yari. I am indebted to my friends, officemates, and colleagues Alex, Andreas, Andreas, Andrianna, Can, Clara, Christos, Daniel, Dimitra, Dimitris, Foivos, Giorgos, Giulio, Ishaq, Krzystof, Lauren, Lazar, Lea, Lukasz, Markus, Michael, Michalis, Muyang, Nathalie, Nikos, Nelly, Paolo, Pavlos, Philip, Ruochun, Stephanie, Veronica, Wioletta, Ylenia, Yuanhua, and Yun, for making these years so enjoyable. I particularly want to thank Andreas for the fruitful research discussions during which he performed the ultimate form of rubber duck debugging we had together and his invaluable help and support.

Finally, I would like to express my gratitude to my family –my brother and my parents– for their endless support and making me who I am today; I hope this piece will

make them proud as this thesis is dedicated to them.

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh.

Abstract

The application of machine learning (ML) and natural language processing (NLP) methods for creating software engineering (SE) tools is a recent emerging trend. A crucial early decision is how to model software's vocabulary. Unlike in natural language, software developers are free to create any identifiers they like, and can make them arbitrarily complex resulting in an immense out of vocabulary problem. This fundamental fact prohibits training of Neural models on large-scale software corpora.

This thesis aimed on addressing this problem. As an initial step we studied the most common ways for vocabulary reduction previously considered in the software engineering literature and found that they are not enough to obtain a vocabulary of manageable size. Instead this goal was reached by using an adaptation of the Byte-Pair Encoding (BPE) algorithm, which produces an open-vocabulary neural language model (NLM). Experiments on large corpora show that the resulting NLM outperforms other LMs both in perplexity and code completion performance for several programming languages. It continues by showing that the improvement in language modelling transfers to downstream SE tasks by finding that the BPE NLMs are more effective in highlighting buggy code than previous LMs. Driven by this finding and from recent advances in NLP it also investigates the idea of transferring language model representations to program repair systems.

Program repair is an important but difficult software engineering problem. One way to achieve a "sweet spot" of low false positive rates, while maintaining high enough recall to be usable, is to focus on repairing classes of simple bugs, such as bugs with single statement fixes, or that match a small set of bug templates. However, it is very difficult to estimate the recall of repair techniques based on templates or based on repairing simple bugs, as there are no datasets about how often the associated bugs occur in code. To fill this gap, the thesis contributes a large dataset of single statement Java bug-fix changes annotated by whether they match any of a set of 16 bug templates along with a methodology for mining similar datasets. These specific patterns were selected with the criteria that they appear often in open-source Java code and relate to those used by mutation and pattern-based repair tools. They also aim at extracting bugs that compile

both before and after repair as such can be quite tedious to manually spot, yet their fixes are simple. These mined bugs are quite frequent appearing about every 2000 lines of code and that their fixes are very often already present in the code satisfying the popular plastic surgery hypothesis.

Furthermore, it introduces a hypothesis that contextual embeddings offer potential modelling advantages that are specifically suited for modelling source code due to its nature. Contextual embeddings are common in natural language processing but have not been previously applied in software engineering. As such another contribution is the introduction a new set of deep contextualized word representations for computer programs based on the ELMo (embeddings from language models) framework of Peters et al (2018). It is shown that even a low-dimensional embedding trained on a relatively small corpus of programs can improve a state-of-the-art machine learning system for bug detection of single statement fixes. The systems were evaluated on the DeepBugs dataset of synthetic bugs, a new synthetic test dataset, and a small dataset of real JavaScript bugs. Lastly, the final contribution is the first steps at answering whether neural bug-finding is useful in practice by performing an evaluation study over a small set of real bugs.

Lay Summary

The application of machine learning (ML) and natural language processing (NLP) methods for creating software engineering (SE) tools is a recent emerging trend. A crucial decision is how to model software's vocabulary. Software developers are free to create any identifiers they like, and can make them arbitrarily complex resulting in an immense out of vocabulary issue. This prohibits training of neural models on large software corpora.

We studied the most common ways for vocabulary reduction in the SE literature and found that they are not enough to obtain a vocabulary of manageable size. As a solution we use an adaptation of the Byte-Pair Encoding (BPE) algorithm, which produces an open-vocabulary neural language model (NLM). Experiments on large corpora show that our NLM outperforms others for several programming languages. It also finds that the BPE NLMs are more effective in highlighting buggy code than previous models. We then investigate the idea of using language model representations in program repair systems.

Program repair is a difficult SE problem. Ideally we would like really high recall without sacrificing precision. We propose to focus on repairing simple bugs, such as single statement bugs, or that match a small set of templates. However, no datasets exist to estimate the recall of such repair techniques. We contribute a large dataset of single statement Java bug-fix changes annotated by whether they match any of a set of 16 bug templates along with a methodology for mining similar datasets. The patterns were selected so that they compile both before and after repair. They can be tedious to manually spot, yet their fixes are simple. These bugs are quite frequent appearing about every 2000 lines of code.

Furthermore, we hypothesize that embeddings augmented with contextual information offer modelling advantages that are specifically suited for source code. Such embeddings are common in NLP but have not been previously applied in SE. We introduce a new set of deep contextualized word representations for computer programs based on the embeddings from language models framework. We show that even a low-dimensional embedding trained on a small corpus can improve a state-of-the-art ML system for bug detection of single statement fixes. We evaluate the system on synthetic bugs and a small dataset or real bugs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free."
–Steve McConnell [from Code Complete McConnell (2004)]

Software has evolved to be an integral part of our everyday lives. Many of the activities that we perform either for work, transportation, communication, or even pure entertainment would be much harder, less safe and sometimes even impossible to perform without it. For the last decade most of the fastest growing companies are technology focused ones whose activities focus around software (Andreessen, 2011). However, creating high quality software is definitely not an easy task. Its development and maintenance are highly expensive processes that require manual human effort. But this huge involvement of the human factor introduces a great risk. A human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in the source code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so (Binder, 1999). Clear definitions for the terms bug, faults, error, and failure and their differences can be found in Section 2.1 and are also available in the literature (Monperrus, 2018a). Quite often though the terms may be used as synonyms and replace one another.

Naturally this introduces several follow-up questions. Q_1. How severe is potentially the impact of these bugs and how much do they cost? Q_2. How easy are they to locate and fix? Q_3. How early should a bug be fixed during the development circle? Q_4. Are there ways to fix bugs more quickly or with less and even no manual effort? Obviously this is not the first time these questions have been asked and answering them is not the real contribution of this thesis. However, it is important to be aware of them, understand them, and realize that most of them are actually open research problems in order to understand the motivation for this thesis. Next, we will provide some brief but important informations regarding these questions from the existing literature.

Q_1. Bugs result in lost productivity of both the software users and the developers that need to fix them. They also result in a significant economic cost. In 2002 a survey by National Institute of Standards and Technology (NIST) reported that its estimation for the annual cost of bugs in the US was $59.5 billion dollars (Tassey, 2002). A 2008 survey by the FBI regarding over 500 companies reported that security defects on their own cost $289,000 per year (Gordon et al., 2000). Another survey from the same year on 139 North American firms reported that on average they each spent $22 million per year on bug repair. More recently, in 2017 the software testing company Tricentis analyzed 606 software fails from 314 companies aiming to better understand the business and financial impact of software failures. The resulting report indicated that these failures affected 3.6 billion people, and caused $1.7 trillion in financial losses and a cumulative total of 268 years of downtime.[1] It stands out that the economic cost of software failures keeps growing and growing, thus highlighting the importance of finding ways to automate program repair and for developers. Furthermore, there are notorious examples of cases where companies learned the hard way how software defects can cause them significant damages. A few of them are listed below. In 1998 **NASA** lost its $125 million Mars Climate Orbiter because spacecraft engineers failed to convert from English to metric measurements when exchanging vital data before the craft was launched.[2] In 1962 due to a typo in mathematical code a **NASA** launched rocket crashed to the ground five minutes after its launch costing $80 million.[3] In 1990 **AT&T** lost about $60 million and 75 million phones were missed due to a problematic switch reporting to other switches it was linked to that it was working properly. This was caused from just a single faulty line of code.[4] In 2004 one of **EDS Child Support**'s computer systems overpaid 1.9 million people, underpaid 700,000 others, and $7 billion of uncollected child support due to a failure in a newly deployed IT system.[5] In 2012 the **Knight Capital Group** lost $440 million in only 30 minutes by selling it sold all the stocks it accidentally bought that morning because of a simple bug. The company would have bankrupted if a group of investors had not come to its rescue as the loss was four times its profit.[6] Also in 2012 **Apple** during its haste attempt to replace Google Maps with its own application made

[1]https://www.tricentis.com/resources/software-fail-watch-5th-edition/
[2]https://www.latimes.com/archives/la-xpm-1999-oct-01-mn-17288-story.html
[3]https://priceonomics.com/the-typo-that-destroyed-a-space-shuttle/
[4]https://www.olenick.com/blog/articles/infamous-software-bugs-at-t-switches/
[5]https://shorturl.at/fhlr0
[6]https://shorturl.at/mn129

serious lapses in navigation resulting in erased cities, buildings disappearing, flattened landmarks, duplicated islands, warped graphics, and false location data. Not only this resulted in financial loss but also severely damaged its reputation since customers felt that it did prioritize money over their needs.[7] In 2014 **Apple** again by misplacing a single extra repeated line pointing to error code caused all of its devices (iPhones, Macs, Tablets) to lose the ability to perform SSL validation thus causing a very serious security vulnerability and allowing man in the middle attacks.[8] In 2020 a new decentralized cryptocurrency called **YAM** collapsed only two days after its launch due to a single line bug.[9]

Q_2. We will not bother with actually answering this question but it is essential for the reader to understand that program repair is the exact opposite of an easy problem. In order to fix a bug we need to gather enough information and understand what the issue actually is. For this to happen bug reports and human or automatic testing may be required. Very often the bug needs to also be reproduced. Moreover, in order to fix the bug we need to find which lines of the source code do actually cause the bug. This process is known as fault localization. A broad overview of fault localization is available in Sections 2.3 and 2.4. Next a patch that fixes the bug needs to be synthesized. The final step in the process is that we need to make sure the bug has actually been fixed without introducing any new ones. So to summarize the process of bug fixing requires to analyze failed executions, locate the cause of the fault, synthesize a bug fix and validate that the fault has been corrected without introducing new ones (Müllerburg, 1983). As one understands program repair is definitely a complex procedure. In fact many companies offer bug bounties, which are rewards paid out to developers that discover critical flaws in their software.[10,11]

Q_3. Bugs should be spotted and fixed the earliest possible. Preventing software failures is the best possible strategy. According to a 2002 NIST survey (Tassey, 2002) fixing a bug found during the production stage needs about three times more effort than if the same bug was found during the coding stage. Moreover, considering that the software development cycle is a sequential process constituted by the *design*, *implementation*, *testing*, and *maintenance* phases a later study estimated the cost of fixing the bug during each of them. Repairing the bug during the implementation stage costs about 6.5 more times than the design stage. During testing the cost grows even more to about 16 times

[7]https://www.huffingtonpost.co.uk/entry/apple-map-fails-ios-6-maps_n_1901599
[8]https://shorturl.at/ntwV5
[9]https://shorturl.at/uKSV3
[10]https://www.bugcrowd.com/bug-bounty-list/
[11]https://hackerone.com/bug-bounty-programs

and finally during maintenance it sky-rockets to about 100 times (Dawson et al., 2010). It stands out that these findings are of great relevance to test driven development, which relies on software requirements being converted to test cases early in development instead of after it is fully developed. Thus test driven development operates by repeatedly testing the software against all the available test cases during the development.

Q_4. From answering the previous questions it stands out that it would be extremely beneficial if there were automatic ways to locate and fix or at least propose fixes early during the software's development cycle. Inspired from this, the area of automatic program repair (APR) was born. Early attempts targeted the detection and sometimes repair of particular classes of bugs under certain assumptions (Demsky and Rinard, 2003; Perkins et al., 2009). Bug detection (BD) is the first and most essential part of APR as we cannot fix a bug without knowing what the bug is and which lines are causing it. In time more and more research focused on APR and even approaches targeting general repair were presented. Many APR techniques have utilized tests to operate. They assume that a failed test signifies a defect in the software and utilize test coverage to find suspicious lines in the source code. In this thesis we focus on BD but many of the techniques presented extend to APR and thus it might be of interest to the reader to understand the background for both. Chapter 2 provides the required background on BD as well as APR.

Recently there have been amazing advances in various artificial intelligence domains using machine learning, natural language processing (NLP), and deep learning techniques. Inspired from this a new line of research has focused into bringing this techniques into the domain of software engineering and building source code specific models. Such models are a promising approach for solving many problems in software engineering including automatic program repair. This area of research is an intersection of machine learning, natural language processing, software engineering and programming languages research. The literature refers to it as machine learning for software engineering (Amershi et al., 2019) or probabilistic source code models (Allamanis, 2017) or even machine learning for big code (Allamanis et al., 2018a). These models attempt to capture and utilize various source code elements like its highly structured information, coding conventions, and semantics contained in identifier names. Last, a lot of this research has been possible thanks to the broad availability of open source projects in hosting services like GitHub.

The work presented in this thesis initially focuses on a very essential core problem of all these models and any software development tools that utilize NLP methods. That is how to represent software's vocabulary in order to properly model arbitrary complex and out-of-vocabulary identifiers while also ensuring that the resulting model will scale to large software corpora. Chapter 3 introduces a novel and effective solution to this problem. It presents a study of the effects of the vocabulary design choices. The choices studied include how to handle comments, string literals, and white space; whether to filter out infrequent tokens; and whether and how to split compound tokens. It also showcases that the most common ways for vocabulary reduction that were previously considered in the literature do not manage to reduce the vocabulary to a manageable size. Instead the presented approach utilizes a Byte-Pair Encoding (BPE) algorithm (Gage, 1994; Sennrich et al., 2015). Contributing a large-scale open-vocabulary neural language model (NLM) for source code that uses beam search and caching to successfully model and predict out-of-vocabulary tokens while also keeping a small vocabulary size. The NLM is able to scale on large corpora for three programming languages (Java, Python, and C) and is shown to outperform previous state-of-the-art methods in both code completion and measuring the entropy of existing code. It also shows that the improvements in language modelling may transfer to downstream software engineering tasks. This is shown via the NLMs ability to highlight buggy code much more effectively than previous language models. These findings point towards the conclusion that advances in language modelling have the potential to lead to improvements in bug detection and by extension program repair systems or even be used in other machine learning for code systems. There are various possible ways that the language model could potentially be utilized to improve such systems. For example one could use it to rank possible fixes. Alternatively, we could utilize the learned representations as feature input to machine learning systems for bug detection and program repair to perform transfer learning. The aim of transfer learning is to take advantage of strong representation from pre-trained representations on large amount of data to create strong machine learning systems even when there is minimal training data available. Since this is exactly what the NLMs for source code presented in this chapter learn they could make an excellent candidate for machine learning systems based on transfer learning, which have shown incredible results in the domain of NLP (Devlin et al., 2018; Radford et al., 2019) and are only recently starting to appear in the domain of machine learning systems that operate over source code (Feng et al., 2020; Kanade et al., 2020; Karampatsis and Sutton, 2020b).

As discussed previously bug detection automatic program repair are important but also really difficult problems in the domain of software engineering. We also briefly discussed how essential is to find bugs as early as possible during the software development cycle. This objective is super hard if not impossible to achieve with current program repair methods for generic fault fixing. Essentially in an industry setting we would like to at least have a program repair system that achieves a “sweet spot” between low false positive rates and high enough recall. The reason for this is that we need the repair system to almost never report false positives. If the false positive report rate is high this would mean that a lot of developer time will be wasted and as a consequence the developers will stop using the system or will be reluctant to do so. Note that even if the system proposes a fix for the defect, it will still need to be validated by the developers. One possible way of achieving the objective of this “sweet spot” is to maybe focus on repairing classes of simple bugs. These could be bugs with single statement fixes or that match a small set of bug templates. Although fixing this kind of bugs may sound simplistic it really is not since the whole pipeline of bug fixing still needs to happen and the small fix size does not make locating a bug much easier. Moreover, as already discussed even one liner bugs can have serious consequences. However, estimating the recall of repair techniques based on repairing such bugs is actually very difficult. The reason for this is that there are no datasets about how often the associated bugs occur in code. Chapter 4 attempts to fill this gap. It introduces a class of bugs that would compile both before and after repair, which results in them being quite tedious to manually spot. Yet the fixes are really simple that many developers would have strong emotional reactions upon realization and would call them stupid. The reaction to finding one such bug could be compared to stubbing one’s toe. Inspired by this we refer to this class of bugs as "simple stupid bugs" (SStuBs). It also provides a large dataset of single-statement bug-fix changes annotated by whether they match any of a set of 16 SStuB templates. The dataset was mined from open-source GitHub Java projects. The dataset has two variants. A smaller one from 100 popular Maven projects containing 25,539 single-statement bug-fix changes that offers the advantage of being able to build the projects and use the test suite of the projects that have one. While the larger one was mined from 1,000 popular open-source projects. Additionally, it introduces a methodology for automatically mining similar datasets for other programming languages with small adaptations such as deciding the appropriate templates for the language of choice. Finally, it studies the degree in which the fixes are graftable i.e., the necessary ingredients for synthesizing the fix are already present in the code. This is known as the plastic surgery hypothesis (Barr et al., 2014). We find that SStuBs present larger graftability than generic bugs.

The training procedure of language models takes into account previous tokens thus it partially takes into account the context of previously seen tokens. The effectiveness of modelling source code from data especially that of language models in source code models hints that context might be a really important source code aspect (Allamanis et al., 2018a). Many models of source code are based on learned representations called embeddings, which transform words into a continuous vector space (Mikolov et al., 2013b). The current software engineering literature has focused on using static embeddings (Harer et al., 2018; Pradel and Sen, 2018; White et al., 2019), which map a word to the same vector regardless of its context. Recently the field of NLP has found contextual embeddings can improve performance for downstream tasks (Devlin et al., 2018; Liu et al., 2019; Peters et al., 2018; Yang et al., 2019). Chapter 5 discusses potential modelling advantages that contextual embeddings may have that are specifically suited to modelling source code. They may be able to capture important information about an identifier that is contained by surrounding names. They are able to assign a different representation to a variable each time it is used in the program thus allowing to potentially capture a variable's value evolution. Moreover the contextual embeddings could be used in a transfer learning setting as has already been done in NLP to create strong models for tasks where only a small amount of supervised data is available. For these reasons the chapter introduces the first deep contextualized word representations for source code based based on the ELMo framework (Peters et al., 2018). Even low-dimensional embeddings trained on a relatively small corpus of programs are shown to improve over a state-of-the-art machine learning system for detection of single statement bugs. It also makes some first steps at studying whether neural bug-finding is useful in practice, which has not been studied in the existing literature.

1.1 Contributions

The aim of this thesis is to design, build, and use machine learning models that can capture and take advantage of the unique properties of source code without sacrificing scalability. The developed models emphasize on bug detection of a novel class of simple semantic bugs that has not been targeted before in the literature. However, they are not just limited to this problem but can generalize and are able to facilitate in solving many other software engineering problems.

The contributions of the thesis can be summarised as follows:

1. An easily applicable and effective solution for how to model source code's vocabulary that addresses the out-of-vocabulary problem. The contribution can be further divided into three subsequent smaller ones. These are a study of how various modelling choices impact the vocabulary of neural language models on a large-scale corpus of 13,362 projects; the design and implementation of an open vocabulary source code NLM able to scale on large corpora; an extensive evaluation showing that the model outperforms the state of the art on three distinct code corpora (Java, C, Python) in cross-entropy, code completion performance (especially identifiers)

and also produces more effective bug detectors than other language models; a beam search like algorithm for producing full token predictions; and finally an effective approach that quickly adapts the model on new projects to massively improve performance (Karampatsis et al., 2020).

2. It introduces the problem of repairing SStuBs and defines what a SStuB is along with frequent SStuB patterns for Java; it presents a methodology for automatically mining a SStuB dataset and introduces the publicly available ManySStuBs4J dataset; it performs an initial analysis over the dataset answering essential research questions; and finally evaluates the degree to which SStuBs fit the plastic surgery hypothesis (Karampatsis and Sutton, 2020a).

3. It discusses reasons why contextual embeddings would be a good fit for source code; it introduces a new set of deep contextualized word representations for computer programs; it extends a state-of-the-art bug detection system's evaluation procedure and compares the model it, outperforming it along several other baselines; and it introduces the question of whether neural bug-funding is practically useful and offers some first insights to this question by mining a small dataset of real bugs and evaluating the methods on it (Karampatsis and Sutton, 2020b).

1.2 Thesis Layout

This thesis is anticipated to be in the general interest of researchers across a variety of different disciplines. Consequently, in order to ensure its accessibility to the general audiences in software engineering, machine learning, and computer science a considerable amount of effort was devoted to provide the necessary background. The thesis is organised as follows. Chapter 2 provides the necessary background required for the reader to understand the remainder of this thesis. Chapter 3 presents the crucial issue of how to model software's vocabulary to address arbitrary complex and out-of-vocabulary identifiers, a study on the effect of the vocabulary design choices, and a large-scale open-vocabulary neural language model for source code, which successfully addresses the issue achieving state-of-the-art performance. Chapter 4 introduces a novel way of thinking about bug detection and partially automatic program repair that can potentially satisfy the expected "sweet spot" of maintaining high precision with adequate recall by repairing a special family of simple single-statement bugs, a methodology for automatic mining the relative datasets, a large Java dataset of such bugs along with a smaller JavaScript one, and finally an analysis of how often such bugs appear and how often does the fix partially or fully exists in previous versions of the code. Chapter 5 is concerned with the potential of contextual embeddings and their suitability for source code. It also describes one of the first such models for source along with experiments showcasing its improved effectiveness over more traditional types of embeddings. Finally, Chapter 6 concludes the thesis with a summary of the main contributions and discusses possible avenues for future research, such as creating a complete tool that automatically detects SStuBs and proposes fixes for them.

1.3 Declaration of Previous Work

This thesis contains work that has been previously published in conferences that have been co-authored with different people as well as currently unpublished work of which a pre-print version is publicly available in arXiv.org. The author of this thesis has been the first author and main contributor for all of these. Specifically, Chapter 3 contains work published in “Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code” (Karampatsis et al., 2020), which is a consolidation of two unpublished works (Babii et al., 2019; Karampatsis and Sutton, 2019) but also introduced several improvements to the training procedure, investigated additional characteristics of the vocabulary, additional improvements to NLM training, an additional use case for NLMs, and a more thorough empirical evaluation. Chapter 4 contains work published in "How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset" (Karampatsis and Sutton, 2020a) but also a bit of additional work that was not included in the published paper due to space limitations. Finally, Chapter 5 contains currently unpublished work that is available in the pre-print "SCELMo: Source Code Embeddings from Language Models" available in arxiv.org (Karampatsis and Sutton, 2020b).

Chapter 2

Background

"Program testing can be used to show the presence of bugs,
but never to show their absence!"
–Edsger Dijkstra [Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee (Buxton and Randell, 1970)]

This chapter provides essential background knowledge that is necessary for understanding the rest of the thesis. An overview of the general concepts and elements of program repair is provided along with all the necessary background on bug detection. We also review the key approaches employed by past research and review some key work on machine learning for source code.

2.1 Automatic Program Repair

Automatic program repair (APR) also known as automatic bug-fixing concerns the automatic repair of software bugs either without or with minimal intervention of a human programmer (Gazzola et al., 2019; Rinard, 2008). In the literature researchers refer to it under various names such as automatic patch generation, automatic bug repair, automatic software repair, or automatic program repair (Gazzola et al., 2019; Monperrus, 2018a). The main goal is to automatically locate faulty elements of the program e.g., statements and automatically generate correct patches that eliminate the bugs contained in the program without causing software regression (Tan and Roychoudhury, 2015) i.e., introducing new bugs. As discussed by Monperrus (2018a) the literature is full of synonyms for bug like defect, fault, error, failure, mistake and so on. Technically, there is a clearly defined difference between faults, errors, and failures (Avizienis et al., 2004): A failure is an observed unacceptable behaviour, an error is propagating incorrect state prior to the failure (without yet having been noticed), and a fault is the root cause of the error (in particular, incorrect code) (Monperrus, 2018a). However, most of the literature fails to stick with these definitions. In many cases there is no separation between automatic repair of failures, errors, and faults, and bug is used as an umbrella term due to its intuitiveness and wide usage (Monperrus, 2018a). Thus, a bug is a deviation between the expected behaviour of a program execution and what actually happened (Monperrus, 2018a).

The problem of generating a fix for program containing faults relates to the more general problem of automatically generating a program that obeys to a given specification e.g., a set of expected behaviours. That is an instance of automatic programming, which seeks to translate a task specification into a machine executable program for the given task (Biermann, 1985; Parnas, 1985). Automatic programming has been considered since the early ages of computer science and is still an active area of research (Alur et al., 2013; Le and Gulwani, 2014; Polozov and Gulwani, 2015). However, APR has only been considered recently. The first major proliferation of research in APR was caused by the publication of Arcuri (2008); Arcuri and Xin Yao (2008); Arcuri and Yao (2007) and Forrest et al. (2009); Weimer et al. (2010, 2009), who pioneered the successful use of search-based algorithms for fix generation. We will next present the basic main concepts and elements of APR and clarify how it relates/differs to automatic software self-healing.

2.2 Basic Concepts

In the literature there exist two general families of approaches for automatically addressing failures. These are the *software healing* and *program repairing* technologies. Both techniques follow similar principles but have different purposes and utilize different approaches to address the faults.

A *software healing* approach detects software failures in-the-field and addresses them by making adjustments to in order to restore the *normal operation* of a system (Ghosh et al., 2007; Perino, 2013; Psaier and Dustdar, 2011). The failures are not being addressed by modifying the source code but are instead applied at runtime on the actual deployed application in order to prevent or mitigate failures. In case more than one similar failures occur on the same deployed application, then the same healing process may be utilized multiple times. On the other hand, a program repair approach detects a failure, localizes the source code location where a fix could be applied and finally makes an adjustment that attempts to *"fix the fault"*, in order to prevent future failures to be caused by the same fault (Kim et al., 2013; Nguyen et al., 2013b; Weimer et al., 2009). The modification can be done on source code but also on binary code (Monperrus, 2018a). Both approaches may or may not include the intervention of a human in the loop. When a human has no intervention we can refer to them as self-healing and self-repair respectively. Otherwise, we note that the human only supervises the process, e.g., may confirm that a fix is correct or choose the appropriate one from a small list of ranked suggested fixes.

Both types of techniques respond to a failure with the execution of a number of operations but they have different goals. A software healing approach utilizes healing operations, which are applied at runtime to transform a failing execution into a successful one. For example, an application starts and needs to connect to a database but the connection fails. The healing approach could attempt to repeat the connection to the database for a limited amount of times in case the database gets back online soon. Thus, the main objective is to grant software availability despite the occurrence of failures (Ghosh et al., 2007; Perino, 2013; Psaier and Dustdar, 2011) instead of identifying and removing the actual fault. This is achieved by applying healing operations that mask the failures (Carzaniga et al., 2015; Iglesia and Weyns, 2015; Riganelli et al., 2017), or at least minimize their impact (Carzaniga et al., 2013; Ding et al., 2012). While a software repair technique performs repair operations, which are applied on the source code itself in an attempt to remove the fault, which caused the failure. Consequently, their end goal is to fix the faults and not to prevent failures (Kim et al., 2013; Nguyen et al., 2013b; Weimer et al., 2009). Instead encountered failures can optionally be used to extract information that can be utilized by the techniques to improve fault identification and fixing. From the above it stands clear that software healing is suitable for improving the reliability and the resilience of software systems, while APR techniques are suitable during development and software maintenance to assist developers in bug-fixing tasks (Gazzola et al., 2019).

Both style of techniques could produce as output either workarounds or fixes although it is more common for repair techniques to generate fixes. A workaround is defined as a temporary solution to the bug (Carzaniga et al., 2015) that is not designed to be optimal but can be deployed fast, until a developer fixes the failure. It can modify the program's source code but it can also alter other elements such as the system's architecture or a specific execution (Gazzola et al., 2019). Contrastingly, a fix eliminates a bug permanently by modifying the source code via generating a patch. It is also required to be of high quality matching that of a fix created by developers if they had been eliminating the fault.

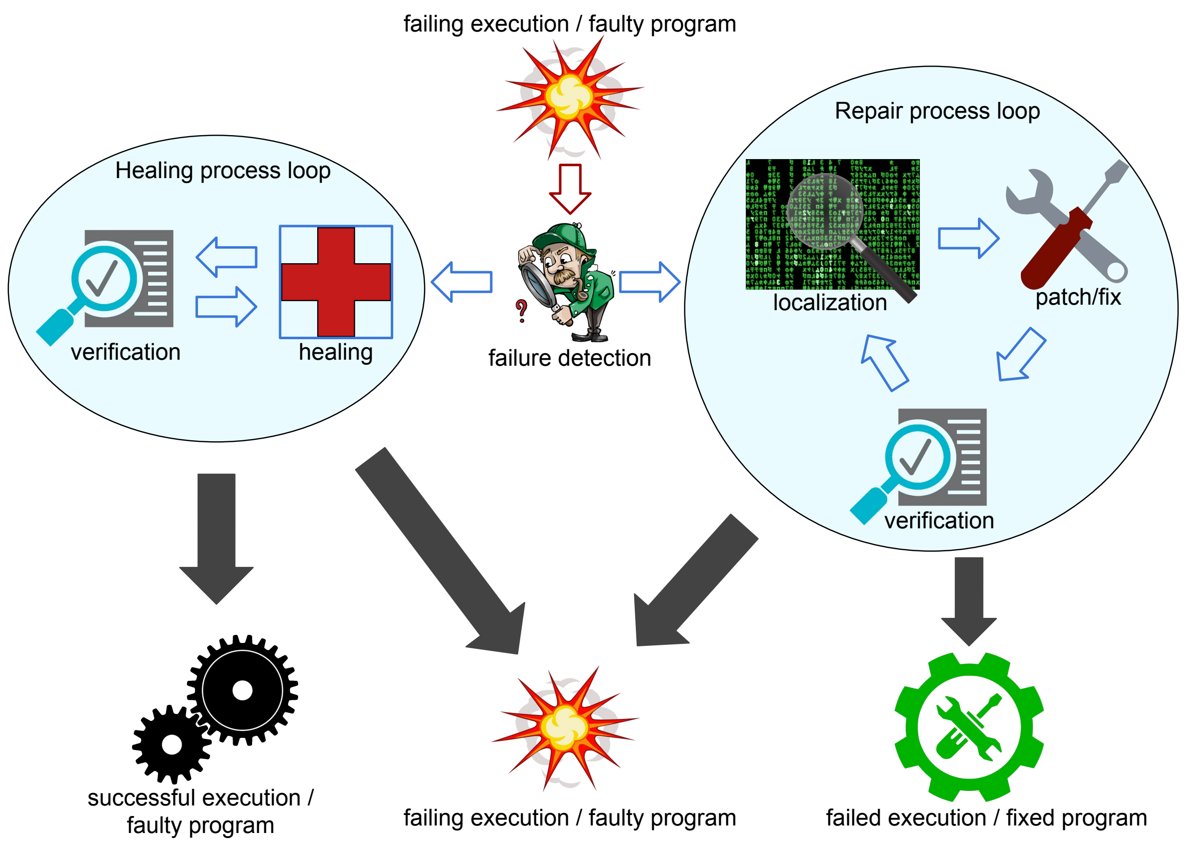

failing execution / faulty program
Healing process loop
verification
healing
failure detection
Repair process loop
localization
patch/fix
verification
successful execution / faulty program
failing execution / faulty program
failed execution / fixed program

Figure 2.1 Overview of the healing and repair processes.

2.2.1 Failure Detection

Both healing and repair approaches are triggered by the existence of a faulty program, which deviates from its intended behaviour for at least one or more executions resulting in a failure. These are spotted by an element of the implemented approach that is responsible for classifying executions as failures or non-failures. An overview of all the elements consisting the process of dealing with program failures is illustrated in Figure 2.1. As shown the failure detection element operates differently depending whether it is serving a healing or repair approach. In the first case, it attempts to spot the symptoms of program failures as early as possible in order to prevent any serious consequences from arising e.g., a system crash or data loss. Consequently, it is implemented as a runtime monitor which checks the application's behaviour (Colin and Mariani, 2004; Jacob et al., 2004). For repair, but not healing, failure detection could simply be based on unit tests. In this case, it extracts the set of failure executions and processes them in order to spot and fix the faults that caused them by exploiting extracted information from the same set of failing executions.

As this thesis focuses on program repair, we refrain from providing the reader with a detailed explanation of how healing approaches operate and the methods contained in this family of approaches. However, we will next provide a simplistic quick overview of their steps shown in Figure 2.1 to let the reader understand how program repair techniques differ from them. The two steps following failure detection in the healing process are healing and verification and can be looped multiple times. The healing step performs a healing operation that prevents or mitigates a detected failure. Next, the verification step follows and checks whether the application is running as expected. However, this step is optional and it is not included in some techniques. In that case, it is simply assumed that any healing operation applied can only have a positive or neutral effect on the system. Otherwise, if the verification step decided that the healing operation was not successful, then the process is repeated unless there no healing operations left available (Chang et al., 2013; Sidiroglou et al., 2009). Lastly, it stands out that the output can be one of two possible outcomes. Either successful "healed" executions where the program still contains faults or failed "unhealed" executions with a still faulty program. In special cases the produced workaround is personally deployed in order to heal future occurrences of the same failure.

On the contrary, as shown in Figure 2.1 a repair approach operates under a different main loop consisting of three main steps, thus it contains an extra step. The main difference is that while healing processes immediately react to a failure by operating on runtime, repair ones first find locations on which fixes can be applied and usually operate on compile time aiming to delivered a fixed program for later executions. We next provide an overview of these three main steps. First, the localization step is tasked with identifying the locations where a fix could be applied to (note that the faulty statements are not always the only good locations for fixes) (Gazzola et al., 2019). Second, the patch/fix step modifies the code in one or more locations identified by the localization step in order to synthesize a patch. Last, the verification step is tasked with deciding whether the synthesized patch was successful in repairing the program. This process can be looped multiple times for different identified locations. Termination of the loop is ensured by stopping when the fault has been fixed, when there are no possible fixes to be generated or when some other termination criterion has been satisfied such as a number of maximum iterations. Consequently, the repair process output can be one of two possible outcomes. Either executions that failed but the identified fault has been fixed and it will not produce further failures, or executions that failed with an unsuccessful repair of the fault. In the following sections we will describe the main strategies that have been utilized in the literature for each of the repair steps as well their evolution.

2.3 Traditional Fault Localization Techniques

This section describes simple intuitive techniques that have been traditionally been used to perform fault localization. Note that the techniques described in this section require a lot of manual effort from developers.

2.3.1 Program Logging

Program logging refers to the process of inserting statements (e.g., print statements) into the code in order to monitor variable values and other program state information (Edwards, 2003). Upon detection of abnormal behaviour during runtime, one or more developers examine the program log in terms of saved log files or printed run-time information to diagnose the underlying cause of failure (Wong et al., 2016).

2.3.2 Assertions

An assertion is an added constrained that must evaluate to true during correct operation of a program. They are specified by developers in the program code using conditional statements, which terminate the execution in case they are evaluated to false. Consequently, they can be used to detect erroneous program behaviour at runtime (this assumes that the defined assertions are correct). A more detailed descriptions of such techniques is provided in (Rosenblum, 1992, 1995).

2.3.3 Breakpoints

A breakpoint allows us to pause the program when its execution has reached a specific predefined point. This allows the developer to examine the current state of the program, modify the value of variables or continue the program's execution in order to monitor a bug's progression. Different types of breakpoints can be configured to trigger such as when the value for an expression changes (e.g., a combination of multiple variables) or upon satisfaction of a user provided predicate. They were used by early studies in the field (Coutant et al., 1988; Hennessy, 1982) to locate bugs by executing the program under a symbolic debugger. But, it is also noteworthy that this is approach is still being used by modern more advanced debugging tools like GNU GDB[1] and Microsoft Visual Studio Debugger.[2] Finally, a major drawback of this approach is that it could potentially take thousands of iterations to run into the problem.[3]

2.3.4 Profiling

Profiling is concerned with the analysis of runtime metrics like execution speed and memory usage. Although such metrics are usually utilized to perform program optimization, they can also be used for debugging purposes. For instance, as discussed in (Wong et al., 2016) it has been used to detect unexpected execution frequencies of different functions (Ball and Larus, 1994), identify memory leaks or poor performing code (Hauswirth and Chilimbi, 2004), and examine the side effects of lazy evaluation (Runciman and Wakeling, 1993). There are several tools that employ profiling for program debugging such as GNU's gprof.[4]

2.4 Advanced Fault Localization Techniques

Modern software systems are characterized by their large size, scale, and complex architecture. Due to this traditional fault localization techniques are ineffective at isolating the root causes of failures (Wong et al., 2016). To address this issue many advanced fault localization techniques have recently emerged based on the idea of causality (Lewis, 1973; Pearl, 2000). It is a related to philosophical theories that focus on the relationship between events and their causes (bugs) and between a phenomenon and its effect (execution failures). Various causality models exist (Pearl, 2000), e.g., counterfactual based, probabilistic or statistical based, and causal calculus models. Probabilistic causality models are the most common utilized in fault localization. Their objective is to identify suspicious code and quantify its responsibility for execution failures.

The techniques described in this section require the existence of an oracle. The oracle can be a formal specification (Wei et al., 2010), an alternative version of the program (Tan and Roychoudhury, 2015), or most frequently a test suite. During the repair process, the test suite is split into two sets. These are passing and failing tests respectively. The

[1]https://www.gnu.org/software/gdb/download/
[2]https://www.microsoft.com/en-gb/download/
[3]http://web.media.mit.edu/~lieber/Lieberary/Softviz/CACM-Debugging/Hairiest.html
[4]https://sourceware.org/binutils/docs/gprof/

repair should not make any of the passing tests fail. A repair is defined as a synthesized patch that consists of one or more code modifications. The repair is assumed to be correct when its application to the code causes it to passes all the tests associated with this bug in a given test suite (Perkins et al., 2009).

The test suite is utilized to identify a small set of suspicious location and optionally as mentioned above to assign each one a responsibility score. These locations are modification candidates for the next stage of the repair. For the majority of methods these locations correspond to program statements. Consequently, the test suite can also be used for verification. For instance, by simply assuming that a bug is fixed when all the tests are successful or when all the relevant ones are.

We will next briefly introduce the various modern families of fault localization techniques.

2.4.1 Slice-Based Fault Localization

Program slicing deletes irrelevant parts of a program abstracting it into a reduced form. The reduced slice should have the same behaviour as the original program under a set of specifications. Program slicing has historically been a very popular fault localization technique with hundreds of papers on this topic. A quick overview on many of these studies can be found in (Binkley and Harman, 2003; Tip, 1994; Xu et al., 2005).

The first family of slicing techniques was proposed back in 1979 by Weiser (1979). This variant is known as static slicing, which aims to reduce the search domain while programmers try to locate bugs in their programs (Weiser, 1981). A static program slice S contains the set of all statements of a program P that could affect the value of a variable v in a statement s for any possible input. The static slice of (s, v) can be computed by first finding all the statements that can have a direct effect on v before statement s is reached. Then, recursively we compute the slices for all variables contained in the extracted statements that affect v. The union of all the extracted slices makes up the final slice. Table 2.1 illustrates the extracted static slice for a small piece of example code.

The main assumption static slicing is based upon is: if the cause of the fail is a statement with a wrong valued variable, then the static slice associated with this specific variable-statement pair should contain the defect. This allows us to narrow the search to the extracted slice instead of the entire program. An extension of this approach constructs a program dice that further reduces the locations that need to be searched. This is achieved by extracting the set of elements that differ between two groups of static slices (Lyle and Weiser, 1987). It is usually applied on the source code itself, but in some cases it can also be applied in binary executables (Kiss et al., 2005) and type checkers (Tip and Dinesh, 2001).

However, static slicing has one major disadvantage. The extracted slice contains all the executable statements that could possibly affect the value of this variable. As a consequence, it can potentially generate a dice containing many statements that should not be included. The reason for this, is that it impossible to predict some runtime values with static analysis.

In an attempt to counter this problem and filter these extra statements from the dice, dynamic slicing was introduced (Agrawal and Horgan, 1990; Korel and Laski, 1988). It allows us to predict runtime values and identify the statements that indeed do affect a particular value at a specific location for a certain execution of a program. For instance, in the case of an if-else block that both the if and else blocks contain statements that affect the slice variable, static slice would include the statements from both blocks. While, for dynamic slicing only the statements of the executed block would be included (e.g., assume that the if block is executed). This is better shown in Table 2.1 that contains the extracted dynamic slice for a small piece of example code. Due to this, dynamic slicing has been a popular approach for fault localization in many studies (Agrawal et al., 1993; Al-Khanjari et al., 2005; Alves et al., 2011; DeMillo et al., 1996; Ju et al., 2014; Kiss et al., 2005; Korel and Laski, 1988; Lian et al., 1997; Liu et al., 2007; Mao et al., 2014; Mohapatra et al., 2004; Pan and Spafford, 1992; Qian and Xu, 2008; Sterling and Olsson, 2005; Wang et al., 2014; Wotawa, 2002, 2010; Zhang et al., 2007a,b, 2005). However, the techniques that utilize dynamic slicing are not limitation free. For instance, they are unable capture execution omission errors. Such errors can result in not executing certain critical statements of a program, resulting in failures (Zhang et al., 2007c). As a solution to this problem, Gyimóthy et al. (1999) propose to locate statements responsible for execution omission errors using relevant slicing. The technique is similar to dynamic slicing and expands upon it. It starts by constructing a dynamic dependence graph but also augments it with potential dependence edges. Then, the relevant slice is computed by taking the transitive closure of the incorrect output on the augmented graph (Wong et al., 2016).

Static and dynamic slicing are not the only families of slicing techniques in the literature. An alternative family of approaches is execution slicing, which uses data flow tests for localizing program bugs (Agrawal et al., 1995). The execution slice of a test case contains the set of statements which were executed by this test. The extracted execution slice for a given test input can be seen in Table 2.1.

Table 2.1 Example showcasing the difference between Static, Dynamic, and Execution slicing for the variable *product*.

	Buggy code piece	Static slice	Dynamic Slice for test input u=2	Execution Slice for test input u=2
l_1	read(v)	read(v)	read(v)	read(v)
l_2	$i = 1;$	$i = 1;$	$i = 1;$	$i = 1;$
l_3	$sum = 0;$	$sum = 0;$	$sum = 0;$	$sum = 0;$
l_4	$product = 1;$	$product = 1;$	$product = 1;$	$product = 1;$
l_5	if($i < v$){	if($i < v$){	if($i < v$){	if($i < v$){
l_6	$sum = sum + 1$	$sum = sum + 1$	$sum = sum + 1$	$sum = sum + 1$
l_7	$product = product * i$	$product = product * i$	$product = product * i$	$product = product * i$
l_8	}else{	}else{	}else{	}else{
l_9	$sum = sum - 1$	$sum = sum - 1$	$sum = sum - 1$	$sum = sum - 1$
l_{10}	$product = product/i$	$product = product/i$	$product = product/i$	$product = product/i$
l_{11}	}	}	}	}
l_{12}	print(sum)	print(sum)	print(sum)	print(sum)
l_{13}	print($product$)	print($product$)	print($product$)	print($product$)

This approach has a significant advantage over static slicing. The former uncovers statements that are executed for a specific input, while the latter uncovers statements that could affect one or more of the variables of interest for any input. A major concept in debugging is that programmers should analyze the program behaviour under the specific test case that fails and not under any generic test case. But static slicing ignores this concept by ignoring the specific input values that reveal a fault. This results in a significant argument for the use of execution over static slicing. Similarly, there is a significant argument over dynamic slicing. The collection of dynamic slices can be a very slow process and might require an excessive amount of disc space. In contrast, it is a much quicker and easier process to construct a specific case's execution slice by collecting code coverage information (i.e., which statements where executed by a test suite) from the test execution. This simplicity led to the development of slice-based debugging tools like χsuds (Agrawal et al., 1998) and eXVantage (Wong and Li, 2006). Moreover, the idea of execution slicing was taken a step further by examining the execution dice for a pair of failed and successful tests to localize faults (Agrawal et al., 1995). Or even further by taking advantage of the following two assumptions (Jones et al., 2002, 2001; Wong et al., 2005). A piece of code is less probable to contain a bug the more tests that execute it. Likewise, for a given bug the more of its tests that execute a piece of code and fail, the greater that probability for the piece to contain this specific bug.

2.4.2 Program Spectrum-Based Fault Localization

Program spectrum-based fault localization (SBFL) is used as the fault localization mechanism by many program repairing techniques (Agarwal and Agrawal, 2014). A program spectrum contains execution information that tracks program behaviour (Harrold

Table 2.2 Example showcasing the difference between Static, Dynamic, and Execution slicing for the variable *product*.

Spectrum Name		Spectrum Description
PIHS	Program Invariants Hit Spectrum	The program properties remaining unchanged during executions
PRCS	Predicate Count Spectrum	The executed predicates
MCSHS	Method Calls Sequence Hit Spectrum	The sequence of executed method calls
TS	Time Spectrum	The execution time of every method or failed executions
BHS	Batch Hit Spectrum	The executed conditional branches
CPS	Complete Path Spectrum	Complete execution path
PHS	Path Hit Spectrum	Intra-procedural, loop free execution path
PCS	Path Count Spectrum	Number of times each intra-procedural, loop-free path is executed
DHS	Data-dependence Hit Spectrum	Definition-use execution pairs
DCS	Data-dependence Count Spectrum	Number of times each definition-use execution pair is executed
OPS	Output Spectrum	The output produced by the execution
ETS	Execution Trace Spectrum	The produced execution trace

et al., 1998; Reps et al., 1997). As discussed in (Wong et al., 2016), this is achieved by extracting program elements such as code coverage, or Executable Statement Hit Spectrum (ESHS) indicating which parts of the program under testing have been covered during execution. Thus, we can limit the search to only the elements involved in a failure. Spectrum based methods are heavily focused on assigning a suspiciousness score to each extracted program element, which represents the probability that it includes a fault. These probabilities are produced by analysing the degree with which the various elements have been covered by passing and failing tests (Eric Wong et al., 2010; Jones and Harrold, 2005; Liblit et al., 2005). The intuition behind the mechanism generating these probabilities is the same one discussed at the end of section 2.4.1. The elements are then checked for fault presentation based on the descending order of suspiciousness until a fault has been located.

The most common elements utilized in the literature are program statements (ESHS). Therefore, for the rest of this section we will focus our discussion on ESHS methods. However, other spectra have been used for fault localization. A short overview of these can be found in Table 2.2.

We next introduce some useful notation for the rest of this section:

P	The program under investigation
N_{CF}	The number of failed test cases, which cover a statement
N_{UF}	The number of failed test cases, which do not cover a statement
N_{CS}	The number of successful test cases, which cover a statement
N_{US}	The number of successful test cases, which do not cover a statement
N_C	The total number of tests, which cover a statement

N_U	The total number of tests, which do not cover a statement
N_S	The total number of succsessful
N_F	The total number of failed tests
t_i	Test number i

Initially, only the failed tests cases were utilized by spectrum-based methods (Agrawal et al., 1991; Korel, 1988; Korel and Laski, 1988; Taha et al., 1989). However, as this strategy was later proved to be ineffective, researched instead focused on the contrast between successful and failed tests. As such it was proposed to use set union and set intersection (Renieres and Reiss, 2003). Set union consists of the source code that is executed by the failed test but by none of the successful ones, while set intersection excludes code executed by all successful tests but not by the failed one. Another ESHS style technique is nearest neighbour (Renieres and Reiss, 2003). It contrasts the failed test with the successful one that is most similar to it based on a given distance metric. The intuition behind the technique it that the closer a statement's execution pattern is to that of failed test cases the more likely it is for the statement to be faulty. Similarly, the farther it is the less suspicious it is. In this sense, a similarity coefficient-based measure can quantify the degree of closeness (i.e., the statement's suspiciousness).

Similarity coefficient-based techniques have had a big impact in fault localization. An early popular such technique is Tarantula (Jones et al., 2001). It uses code coverage and execution results to estimate the suspiciousness S of each statement based on the following formula: $S_{Tarantula}(P) = \frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}}$. Experiments on the Siemens suite (Jones and Harrold, 2005) showed that Tarantula was more effective than other at the time state-of-the-art techniques such as set union, set intersection, and nearest neighbour (Cleve and Zeller, 2005). Table 2.1 illustrates the suspiciousness of each statement for the example program originally introduced in Table 2.1. Interestingly, one can observe the faulty statement of l_7 is ranked first but alongside the non-faulty one of l_6.

However, Tarantula's performance has been matched or even surpassed by other more recent techniques. One such popular technique that is more effective than Tarantula is Ochiai (Abreu et al., 2006). The formula for Ochiai is: $S_{Ochiai}(P) = \frac{N_{CF}}{\sqrt{N_F * (N_{CF} + N_{CS})}}$. From the Tarantula and Ochiai formulas one can easily deduce that for both methods the more failed tests that execute a statement, the more suspicious it is. Similarly, the more successful tests cases that execute it, the less suspicious it is. With 0 representing a statement with minimum scoring suspiciousness and 1 a maximum scoring one.

Table 2.3 Example illustrating the suspiciousness value for each statement using the Tarantula similarity coefficient.

	Code with bug at l_7	$v = 1$	$v = 2$	$v = 5$	N_{CF}	N_{CS}	$S_{Tarantula}$	Rank
l_1	read(v)	●	●	●	2	1	0.5	3
l_2	$i = 1;$	●	●	●	2	1	0.5	3
l_3	$sum = 0;$	●	●	●	2	1	0.5	3
l_4	$product = 1;$	●	●	●	2	1	0.5	3
l_5	if($i < v$){	●	●	●	2	1	0.5	3
l_6	$sum = sum + 1$		●	●	2	0	1	**1**
l_7	$product = product * i$		●	●	2	0	1	**1**
l_8	}else{	●			0	1	0	10
l_9	$sum = sum - 1$	●			0	1	0	10
l_{10}	$product = product/i$	●			0	1	0	10
l_{11}	}	●			0	1	0	10
l_{12}	print(sum)	●	●	●	2	1	0.5	3
l_{13}	print($product$)	●	●	●	2	1	0.5	3
Execution Results		Success	Failure	Failure				

These techniques differ in two main ways with the nearest neighbour technique described above. First, they use multiple failed test cases while nearest neighbour only uses a single one. Last, they utilize every successful test case, while nearest neighbour uses only the most close one to the failed test case.

We will not describe every spectrum based or even every ESHS-based technique nor describe one as panacea that outperforms all others as the existing evidence is that such a technique does not exist (Yoo et al., 2014). However, a plethora of ESHS-based techniques along with their algebraic formulas can be found in Table 2.4.

Nevertheless, ESHS methods do not come without drawbacks. The most major one is that they are imprecise as there is no guarantee that the bug can be fixed in any of the locations they identify. Moreover, although ESHS-based methods are able to assign a suspiciousness score for fault localization and perform verification via the test suite, they require the intervention of either a programmer or some other method to synthesize a patch/fix. Thus they do not provide an end-to-end automatic program repair solution.

2.4.3 Program State-Based Fault Localization

Program state contains important information that can offer useful and rich insights for localizing program bugs. It consists of variables and their values at a particular point of execution (Laplante, 2000). A simple but limited approach is relative debugging (Abramson et al., 1995), which allows locates faults in the program's development version by comparing states to those of a reference version. Another important method is delta debugging (Zeller, 2002; Zeller and Hildebrandt, 2002). It compares the program state of a successful test with that of a failed one via their memory graphs (Zimmermann and

Table 2.4 Alphabetically sorted similarity coefficient techniques and their algebraic forms.

Coefficient	**Algebraic Formula**
Ample	$\lvert\frac{N_{CF}}{N_{CF}+N_{UF}} - \frac{N_{CS}}{N_{CS}+N_{US}}\rvert$
Anderberg	$\frac{N_{CF}}{N_{CF}+2*(N_{UF}+N_{CS})}$
Arithmentic Mean	$\frac{2*(N_{CF}*N_{US}-N_{UF}*N_{CS})}{(N_{CF}+N_{CS})*(N_{US}+N_{UF})+(N_{CF}+N_{UF})*(N_{CS}+N_{US})}$
Baroni-Urbani & Buser	$\frac{\sqrt{(N_{CF}*N_{US})}+N_{CF}}{\sqrt{(N_{CF}*N_{US})}+N_{CF}+N_{CS}+N_{UF}}$
Braun-Banquet	$\frac{N_{CF}}{max(N_{CF}+N_{CS},N_{CF}+N_{UF})}$
Cohen	$\frac{2*(N_{CF}*N_{US}-N_{UF}*N_{CS})}{(N_{CF}+N_{CS})*(N_{US}+N_{CS})+(N_{CF}+N_{UF})*(N_{UF}+N_{US})}$
Dennis	$\frac{(N_{CF}*N_{US})-(N_{CS}*N_{UF})}{\sqrt{n*(N_{CF}+N_{CS})*(N_{CF}+N_{UF})}}$
Dice	$\frac{2N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
Fleiss	$\frac{4*(N_{CF}*N_{US}-N_{UF}*N_{CS})-(N_{UF}-N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})+(2N_{US}+N_{UF}+N_{CS})}$
Fossum	$\frac{n*(N_{CF}-0.5)^2 num}{(N_{CF}+N_{CS})*(N_{CF}+N_{UF})}$
Goodman	$\frac{2N_{CF}-N_{UF}-N_{CS}}{2N_{CF}+N_{UF}+N_{CS}}$
Gower	$\frac{N_C F+N_U S}{\sqrt{N_F*N_C*N_U*N_S}}$
Hamann	$\frac{N_{CF}+N_{US}-N_{UF}-N_{CS}}{N_{CF}+N_{UF}+N_{CS}+N_{US}}$
Hamming	$N_{CF} + N_{US}$
Harmonic Mean	$\frac{(N_{CF}*N_{US}-N_{UF}*N_{CS})((N_{CF}+N_{CS})*(N_{US}+N_{UF})+(N_{CF}+N_{UF})*(N))}{(N_{CF}+N_{CS})*(N_{US}+N_{UF})*(N_{CF}+N_{UF})*(N_{CS}+N_{US})}$
Jaccard	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
Kulcynski	$\frac{N_{CF}}{N_{UF}+N_{US}}$
Michael	$\frac{4*((N_{CF}*N_{US})-(N_{CS}*N_{UF}))}{(N_{CF}+N_{US})^2+(N_{CS}+N_{UF})^2}$
Mountford	$\frac{N_{CF}}{0.5*((N_{CF}*N_{CS})+(N_{CF}*N_{UF}))+(N_{CS}*N_{UF})}$
Ochiai	$\frac{N_{CF}}{\sqrt{N_F*(N_{CF}+N_{CS})}}$
Ochiai2	$\frac{N_{CF}}{\sqrt{(N_{CF}+N_{CS})*(N_{US}+N_{UF})*(N_{CF}+N_{UF})*(N_{CP}+N_{US})}}$
Pearson	$\frac{n*((N_{CF}*N_{US})-(N_{CS}*NUF))^2}{N_C*N_U*NS*NF}$
Phi (Geometric Mean)	$\frac{N_{CF}*N_{US}-N_{UF}*N_{CS}}{\sqrt{(N_{CF}+N_{CS})*(N_{CF}+N_{UF})*(N_{CF}+N_{US})*(N_{UF}+N_{US})}}$
Pierce	$\frac{(N_{CF}*N_{UF})+(N_{UF}*N_{CS})}{(N_{CF}*N_{UF})+(2*(N_{UF}*N_{US}))+(N_{CS}*N_{US})}$
Rogers & Tanimoto	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{CS}+N_{US}+N_{UF}}$
Rogot1	$\frac{1}{2}(\frac{N_{CF}}{2N_{CF}+N_{UF}+N_{CS}} + \frac{N_{US}}{2N_{US}+N_{UF}+N_{CS}})$
Rogot2	$\frac{1}{4}(\frac{N_{CF}}{N_{CF}+N_{CS}} + \frac{N_{CF}}{N_{CF}+N_{UF}} + \frac{N_{US}}{N_{US}+N_{CS}} + \frac{N_{US}}{N_{US}+N_{UF}})$
Scot	$\frac{4(N_{CF}*N_{US}-N_{UF}*N_{CS})-(N_{UF}-N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})*(2N_{US}+N_{UF}+N_{CS})}$
Simple Matching	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{CS}+N_{US}+N_{UF}}$
Sokal	$\frac{2(N_{CF}+N_{US})}{2(N_{CF}+N_{US})+N_{UF}+N_{CS}}$
Sorensen-Dice	$\frac{2N_{CF}}{2N_{CF}+N_{UF}+N_{CS}}$
Tarantula	$\frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F}+\frac{N_{CS}}{N_S}}$
Tarwid	$\frac{(n*N_{CF})-(N_F*N_C)}{(n*N_{CF})+(N_F*N_C)}$
Zoltar	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_C S+\frac{1000*N_{UF}*N_{CS}}{N_{CF}}}$

Zeller, 2001), via modifying the variable values of the successful one into those of the failed one. Thus, it manages to significantly reduce the execution's length.

An extension of delta debugging, the cause transition technique attempts to spot when and where a failure's cause moves from one variable to another one (Cleve and Zeller, 2005). Nevertheless, this technique has the disadvantage of being slow and costly, requiring multiple test executions at each matching point of states of which thousands may exist. Other extensions combine it with other execution alignment techniques to improve its precision, robustness, and efficiency (Sumner and Zhang, 2009, 2010; Xin et al., 2008).

Another important approach is predicate switching (Zhang et al., 2006). It forcefully alters program states to modify the branches executed in failed executions. Any of these predicates that when modified results in a successful program execution is labelled as a critical predicate. A similar approach by Wang and Roychoudhury (2005) studies a failed test's execution path and modifies the branches in it, so that they produce a different outcome resulting in a successful program execution. The branches of which the outcome was modified are then labelled as buggy.

2.4.4 Machine Learning-Based Fault Localization

Machine learning (ML) is the field of study concerned with the question of how to construct programs that automatically improve with experience (Mitchell, 1997). It has not just been successfully employed in multiple fields but instead even shown to have superhuman abilities in numerous fields (e.g., playing go, self driving cars, image classification, etcetera) (Schmidt et al., 2019). The big advantage of ML is that it requires minimal to no human interaction and the resulting models are usually robust and adaptive.

The first wave of ML approaches for APR was concerned with learning to identify the location of a fault based on coverage information of each test as well as their execution results. Wong and Qi (2009) feed the coverage data and the paired execution results to a back-propagation feed forward neural network (Fausett, 1994), which learns to predict the likelihood that a given statement paired with a virtual test case contains a bug. An extension of this approach for object oriented programs was developed by (Ascari et al., 2009). In an attempt to address common issues (e.g., local minima) of feed forward networks, Wong et al. (2012) used a radial basis function (RBF) network. The network is trained similarly and the output represents the fed statement's suspiciousness. Another approach uses execution traces. It generates statement subsequences of length N from the trace data which are referred to as N-grams. Then, it estimates the probability that a specific N-gram exists in a failed execution trace using statistical analysis. Finally, the N-grams are sorted in descending order of the estimated probabilities. This technique is more effective at locating faults than Tarantula (see Section 2.4.2) (Nessa et al., 2008).

2.5 Generate-and-Validate Approaches

As it was illustrated in Figure 2.1 fault localization is followed by two other activities, patch/fix generation and validation. All three consist the repair process loop. As previously discussed the patch/fix generation and validation can be performed by developers. Generate and validate approaches aim to minimize the tax of these two processes on developers. We next provide a quick generic overview of how these two activities operate.

The generate activity aims to modify the original program P and produce a set of new programs as candidate solutions. The set's size is a simple parameter that may have runtime speed effects depending on the algorithm of choice. In order to create candidates these sort of approaches utilize a set of modification operators over the original program P. Thus, applying one of these operators results in a new candidate. Three different types of change operators are common in the literature. *Atomic change operators* modify a single point of P. *Pre-defined template operators* as the name implies modify P based on pre-defined patterns that are given as input to the algorithm. These can range from being rather simplistic to arbitrarily complex. Last, *example-based operators* are quite similar to *Pre-defined template operators*. The main difference is that the templates have to be extracted from data like changes in a versioning system. The extraction process can be either manual or automatic. One could easily note that applying the operators only on the original program P seems a bit limiting. A strong argument for why that is that applying them only on P restricts the candidate programs from accumulating incremental changes. As such it is quite common in the literature to also apply the operators on elements of the set of generated candidate solutions (Ackling et al., 2011; Arcuri, 2008; Assiri and Bieman, 2014; Forrest et al., 2009; Kelk et al., 2013; Kim et al., 2013; Kou et al., 2016; Wilkerson and Tauritz, 2010). Applying the operators on both the original program and the candidate solutions results in a search space for repair S_{rep}. Finally, the operators can be applied randomly, guided by a heuristic or by following a complete brute force strategy that produces every possible candidate based on the utilized fault localization algorithm.

The purpose of the validation activity is to check whether any of the candidate solutions is correct thus fixing the bug. Obviously, the solutions can be manually validated by developers but sometimes this might not be an easy process even for them.

Instead, as previously discussed correctness is automatically validated by running the test suite. A candidate program with no failed tests is considered a possible fix. Following a similar logic candidates that fail many of the tests are discarded and the ones with the most potential are kept to continue the search.

2.5.1 Atomic Change Operators

Atomic change operators are applied on a program's abstract syntax tree AST. When one is applied it modifies the AST in only one spot by inserting a new node, deleting one, or modifying one. The application of the operator must produce a valid AST. Some very simple examples are to modify an expression's operator or to replace a variable in an expression with another one. The advantage of this kind of operators is that they are simple to implement and apply which can make the search fast.

Search Based Techniques

We next provide a brief overview of search based techniques. Most of these utilize three generic groups of atomic change operators. These are deleting an AST statement, copying one from a random position of the AST and inserting it in another one, and modifying a node/element of the AST with a random different one. These generic operators are simple but they have the potential to repair any faults for which the statements needed to create the fix are already contained somewhere else in the program, which is known as the plastic surgery hypothesis (Barr et al., 2014). Consequently, if the statements required for the fix are not already included in the program then these techniques are incapable of repairing it. One instance in this family of techniques that attempts to mitigate this issue is JAFF (Arcuri, 2008, 2011). This is achieved by randomly generating new statements. Although this indeed creates the potential to fix more faults it is extremely improbable to randomly generate the statements needed for a fix.

An extremely popular and impactful technique is GenProg (Forrest et al., 2009; Le Goues et al., 2012; Le Goues et al., 2012a,b; Weimer et al., 2010, 2009), which offers a full test-suite based repair system that utilizes genetic programming to search for patch candidates. Fault localization in GenProg operates in a similar manner to ESHS methods (see Section 2.4.2). Statements executed by only the failing test cases are assigned a maximum suspiciousness score of 1 and those executed by only successful ones are assigned a score of 0. While those that are executed by both failing and non-failing test cases are given a score of 0.1. Alternative scoring schemes were later considered including SBFL ones (Qi et al., 2013). Verification operates similarly to other methods by utilizing the test suite. GenProg's novelty comes from constraining its search for a patch, in a pre-defined space of AST modifications instead of attempting to fix arbitrary faults. However, it can still repair large complex program by generating non-trivial multi-line fixes. The test suite is utilized to decide the correctness of a patch. While the search part is based on a genetic algorithm (Gen and Cheng, 1999). Genetic algorithms are inspired by the concept of natural selection (Darwin, 1859). This kind of algorithm maintains and evolves a population of individual candidates (chromosomes) and consist of three main elements. Mutation and crossover operators, and a fitness function. For each iteration the current populations is scored by the fitness function. This score represents the chromosome's probability to reproduce in order to create chromosomes for the next population. In the case of GenProg the fitness function is its fault localization function described above and candidates with fitness 0 (i.e., do not compile or do not pass any of the test cases) are discarded (Le Goues et al., 2012a). The crossover operator uses two chromosomes. It selects a statement from the first chromosome as a cutoff point and swaps all statements after the cutoff with the remaining statements of the second one. Finally, the mutation operator selects a statement node from the AST of P and performs one out of three different operations with uniform probability. These are deletion of the statement, insertion of another statement after it, or a swap with another statement. In the latter two cases a statement is selected uniformly at random probability from anywhere in the program (Le Goues et al., 2012a).

GenProg also has certain drawbacks. First, there is only a limited number of bugs that it can repair since the correct functionality must already exist in the original program. Second, the fixes might be of low quality since the random modifications can introduce new bugs and/or break the structure of the program. Third, the output patch might not correct the bug but just mask it instead (e.g., by removing certain branches through implicit data-flow) (Tan et al., 2016). Masking the bug is not useful for repair but can offer helpful information for localizing it. Fourth, there is no guarantee that the fitness function will select dissimilar individuals and as a consequence the crossover operator may not produce a diverse enough population. Last, the approach is not well-suited to object-oriented languages like Java. As for instance JGenProg (a GenProg implemenation for Java) was shown to perform poorly (Martinez et al., 2017). However, other approaches such as ARJA have since improved on its performance. Moreover, since the test suite is also human written code it may also contain bugs that could mask a failure, fail for a bug that does not exist, or present a correct fix as incorrect. Furthermore, there has also been some useful research (Martinez et al., 2017; Qi et al., 2015) on criticising this kind of techniques with a main focus on GenProg. As shown most of the outputted patches are actually incorrect. Many patches are just plausible (i.e., they produce correct outputs for all inputs in the test suite) but not correct and the overwhelming majority of the plausible patches are equivalent to a single functionality deletion modification Qi et al. (2015). This also true for Java on the Defects4J (Just et al., 2014a) dataset Martinez et al. (2017).

A technique that works similar to GenProg is Marriagent (Kou et al., 2016). It stands out by using a much different crossover algorithm. Selecting candidates for crossover randomly or those with the best fitness is very probable to produce offspring that are very similar to the existing candidates. To avoid this Marriagent's selection criterion favours diversity. The diversity between two program candidates is estimated based on the common and the total set of changes that have to be applied to each of the programs for it to be generated from the original one. As a consequence, candidate pairs with a larger diversity have also a better selection probability.

An alternative to utilize a heuristic for the search is to use a variant of random search. One such technique is RSRepair (Qi et al., 2014), which was previously known as TrpAutoRepair (Qi et al., 2013). During each iteration it chooses a statement to change based on their suspiciousness and applies a random change operator. RSRepair is much simpler than GenProg as it neither applies the operators on the candidate solutions but only on the original program nor it crossovers candidates. As soon as a candidate is generated it is validated using the test suite. If one or more tests fail the it is discarded and the search continues. Consequently, the resulting search space consists of all the programs that can be created from modifying each of the original program's statements separately. Last an extension of RSRepair, SCRepair (Ji et al., 2016) utilizes a similarity metric to seach for code that is similar enough to the code that needs to be repaired so that the chosen code integrates well with it.

Another well known technique that was also mentioned previously is JAFF (Arcuri, 2008, 2011), which not only uses operators that operate over individual nodes but also over subtrees of the AST. Another novel characteristic is that although it selects statements based on their suspiciousness, it also introduces randomness to the selection by selecting a number of random statements and then chooses the most suspicious one among them.

Three well known techniques that aim on generic fault fixing are pyEDB (Ackling et al., 2011), MUT-APR (Assiri and Bieman, 2018), and CASC (Wilkerson and Tauritz, 2010, 2011; Wilkerson et al., 2012). First, pyEDB is a GenProg like system. It's novelty comes from the way it represents candidates for mutation. They are represented by their delta in relation with the original program. So their basically represented by what changes need to be applied over the original. It is much more efficient representation since the set of changes is much smaller than the modified or original programs. The modifications are represented using two tables, one containing all the possible changes that can be performed on P via relational operators and the other one all the possible valid changes that can be performed over variable names. A 32 bits string is also utilized, which encodes the chosen table, the selected modification, and the AST node that the change is applied upon. Crossover and fault localization operate in the same way to previous methods. Second, MUT-APR is also similar to GenProg but it focuses only on fixing faults that can be repaired by modifying arithmetic, relational, bitwise, and shift operators. Last, CASC also uses genetic programming and modifications based on atomic change operators. However, it does not limit the evolution application only on the original program and the candidate solutions but also on the test cases. So in this case we are not only evolving the candidates that past the most tests but also the tests that discard the most candidates. The problem with this approach is that it is really difficult to not generate incorrect test cases that will induce bias towards the generation of incorrect candidates. However, it could also fix faulty test cases (test code is also written by humans and as such it may contain bugs) but this is really improbable to happen.

Brute-Force Techniques

A brute force technique attempts to search the whole of space all solutions in a systematic and constrained manner. However, there is no unique way for the search to operate nor a fixed set of operators since they aim on fixing different fault types. In general there exist four main modification operators utilized by this kind of techniques. These are replacing operators in the program, inserting or deleting a method call, functionality deletion, and AST modifications.

Debroy and Wong (2010) focus on the set of atomic change operators utilized in mutation testing. Thus it allows replacement of arithmetic, logical, relational, assignment, increment, and decrement operators with another operator of the same kind. So a logical operator can only be replaced with another logical one. Also conditions in *if* and *while* statements can be negated. Search operates by ranking all the statements in the program based on suspiciousness and then modifying them in order. However, only programs with a single applied mutation on the original are allowed. A time limit or number of maximum allowed mutations can be used to end the search before all possible candidates are exhausted. Another popular technique is PACHIKA (Dallmeier et al., 2009), which focuses on object oriented programs. It operates by inferring the preconditions that need to be satisfied for successful execution of each method. Fixes are applied only by removing or adding method calls so that unsatisfied preconditions become satisfied and are limited to methods that take no arguments. Also the removed method calls are not actually being deleted but are wrapped inside an *if* block so that they are not executed when the preconditions do not hold. Finally, another well known technique is Kali that aims on deleting code that is not necessary but may be harmful. It uses SBFL (see Section 2.4.2) and atomic operators focused on removing functionality which are applied only on the top 500 most suspicious statements. As such the proposed fixes by this technique are only plausible and would probably not be considered correct by developers as they may remove functionality.

2.5.2 Pre-Defined Templates

The techniques in this section operate by modifying one or more statements using pre-defined templates. The main advantage of such techniques is that they allow to apply complex changes in multiple locations of the program. Most of the techniques presented here are based on brute-force instead of search-based strategies. This is mainly preferred because applying the complex operators of these techniques would significantly slow down the genetic evolution.

As a consequence we will briefly focus on brute force methods for the remainder of this section. As mentioned before these techniques aim do not focus on a specific class of faults but on fixing any kind of generic ones. AutoFix-E (Wei et al., 2010) is a technique that operates very similar to PACHIKA but is specialized on the programming language Eiffel, which utilizes contracts like function pre-conditions and post-conditions. Essentially contracts provide semantic information about the program that results in more complex fixes than PACHIKA and speeds up the search by allowing the algorithm to distinguish between correct and faulty program states. Another of its characteristics is that it tries to prioritize fixes that cause the smallest possible changes using a special metric that checks the effect of a given fix on the source code and the program's state. A later version of AutoFix, AutoFix-E2 (Pei et al., 2011) improves the application of the templates and also ranks the candidate solutions according to their suspiciousness using SBFL.

A quite well known program repair system is SPR (Long and Rinard, 2015), which uses parameterized templates. It uses a staged process that focuses only on the most promising cases and thus can skip many of the wrong fixes. Because each of the templates is parameterized it does not represent a single program transformation but a whole class of them. The method first applies a set of templates on the program and each on of them is parameterized. Then, it decides if there are any parameter configurations that can make the repair successful. Finally, if a condition is required to produce the necessary values it tries to synthesize and add it in the program. However, the synthesized conditions are only limited to the form: *var op constant*. *var* is any variable name, *op* is only $==$ or $!=$ and *constant* is any constant value. The templates used in SPR resemble atomic change operators but are actually more complex as for instance an atomic operator would only change the operator in a condition while a template can potentially modify all the elements of its elements such variable names and constant values. It can also copy and move around statements to other locations of the source code.

Prophet (Long and Rinard, 2016) improves upon SPR by utilizing a massive database of software revision changes. It's main assumption is that it can learn from previous fixes as similar ones may be re-applicable across a project's lifespan. As these are human fixes they are also more probable to be correct.

2.6 Semantics-Driven Approaches

Semantics-driven approaches operate by creating a formal encoding of the problem. The encoding can be explicit like a formula, which can be solved to get the possible fixes or implicit requiring an analytical procedure, which results to the fix. When a solution is found it does not need to be validated as it is guaranteed to fix the formal encoding of the problem. However, this does not mean that it is the correct fix for the problem and should be accepted by the developers. This is because the discovered solution was found for an approximation of the repair problem and not for the original one. As such although it might provide a fix it can still introduce new issues or not be correct based on other aspects of program that were not included in the approximation.

Usually this kind of techniques involve three processes. The first analyzes the program and extracts semantic information regarding the program's correct and incorrect behaviours from its elements such as the the source code or the available test cases. The second uses the information extracted from the previous one to generate a formal representation, which can be solved to get fixes. As discussed above the extracted representation can be either explicit or implicit. The final process attempts to solve the extracted formal representation. If it finds one or more solutions then it outputs the code changes required for each one unless it finds no solution or runs out of time. In order to generate the code changes (fixes) that need to be applied most of the techniques employ program synthesis, which is the task of automatically finding a program in the underlying programming language that satisfies the user intent expressed in the form of some specification (Gulwani et al., 2017b). As program synthesis is a huge research topic we will not describe it here but instead the reader is pointed to an analytical overview (Gulwani et al., 2017b).

Another important property of semantics-driven techniques is that most of them do not focus on repairing general faults but instead target specific classes as this makes it much easier to extract a formal representation for the repair problem. There exist though some techniques that attempt general fault fixing. Notable instances of such are SemFix (Nguyen et al., 2013b), DirectFix (Mechtaev et al., 2015), Angelix (Mechtaev et al., 2016), and SearchRepair (Ke et al., 2015). We next provide a quick overview of the fault classes that can be repaired and a few noteworthy examples for each one of them. First, we have incorrect or missing conditions in branch (e.g., an *if* statement) or loop statements (e.g., a *while* statement). These techniques usually locate the most suspicious conditions in the program and utilize the available variables that are in scope. Notable examples are NOPOL (DeMarco et al., 2014; Xuan et al., 2017), Dynamoth (Durieux and Monperrus, 2016), Infinitel (Marcote and Monperrus, 2015). Second, we have concurrency faults such as atomicity violations, deadlocks and livelocks. The methods addressing these kinds of faults focus on discovering the areas of the code that are most probably responsible for the faults and then enhance them with synchronization mechanisms that prevent these kind of problems. Some well known techniques are AFix (Jin et al., 2011), CFix (Jin et al., 2012), HFix (Liu et al., 2016), Surendran (Surendran et al., 2014), Axis (Liu and Zhang, 2012), Grail (Liu et al., 2014), and Dfixer (Cai and Cao, 2016). Third, are faults in generating HTML code. Two major methods are PHPQuickFix and PHPRepair (Mohammadi et al., 2018). Fourth, are methods that repair routines which fail at validating input strings in web applications. SemRep (Alkhalaf et al., 2014) is one such method as well as the technique by Yu et al. (2011, 2016). Fifth, are techniques that focus on access control violation faults in PHP web applications and the most well known one is FixMeUp (Son et al., 2013). Last, are methods targeting memory leaks in C programs like LeakFix (Gao et al., 2015).

2.7 Modern Machine Learning for Bug Detection and APR

The techniques discussed in Section 2.4.4 were mainly based on applying off-the-shelf machine learning methods and feeding them with hand-engineered or hand-extracted features. This simplistic approach was not only utilized by early machine learning based fault localization and program repair methods but also by all kind of machine learning for software engineering techniques. In this section we will focus on recent machine learning for APR techniques but an analytical overview of machine learning methods for source code is available in the literature (Allamanis et al., 2018a).

A lot of the work in this area is based on the naturalness hypothesis (Hindle et al., 2012). That is: *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.* Based on this hypothesis many methods assume that large code corpora contain rich patterns that can be exploited to build useful probabilistic machine learning models. These can also be used to infer relationships between the source code itself and other elements like bug reports and comments.

Following the naturalness hypothesis many techniques utilize or are based of generative probabilistic models of code. These describe a stochastic process for generating valid code (i.e., they model how code is written) (Allamanis et al., 2018a). These can either be a probability distribution that can be sampled to generate source code or a transducer that utilizes external context information (e.g., comments) to essentially create a multimodal generative code model. Consequently, these models can be used to assign a non-zero score to a given piece of code that represent its fitness (i.e., how probable is under the given model). Such models have been used in various program repair applications. Bhatia and Singh (2016) utilized them to correct syntax errors in programming assignments, Campbell et al. (2014) to detect syntactic errors, Godefroid et al. (2017); Patra and Pradel (2016) to perform input fuzzing and find software vulnerabilities, Pu et al. (2016) to correct both syntactic and semantic errors in massive open online courses, Wang et al. (2016a) to detect bugs, and finally Ray et al. (2016a) to highlight buggy code and order warnings generated from static bug finders.

Generative models of source code are not the only way to utilize machine learning in this domain. An alternative is to use models that can encode and predict certain properties of the code that are useful to software engineers. In order to achieve this the models learn an intermediate encoding of the source code, e.g., a vector embedding. We note that not all but some generative models are also able to provide such encodings. A very often utilized type of model is to learn a distributed representation (Hinton et al., 1986). This is basically a vector or matrix of real numbers, where instead of each element uniquely representing a property, its meaning is distributed across multiple elements. Using distributed representations and expecting them to be effective representations is not surprising at all as they have extensively been used in the field of natural language processing to encode words (Mikolov et al., 2013a) as well as sentences and documents (Le and Mikolov, 2014). Finally, they have been used by various program repair methods.

Gupta et al. (2018, 2017) use them to fix common typographic errors in C programs without any knowledge of the language's formal syntax, Wang et al. (2016b) to detect defective code regions, Devlin et al. (2017c) to repair misused variables in Python using function embeddings, Azcona et al. (2019); White et al. (2019) to perform automatic program repair in Java using token embeddings, Chen and Monperrus (2018b) to discover the correct ingredient in automatic program repair by computing the distance between pieces of code based on the embeddings cosine similarity, Yin et al. (2018b) to represent source edits and patches, Chen et al. (2019a) to learn to transform buggy source code lines to potentially fixed ones, and finally Pradel and Sen (2018) use them to learn bug detectors for specific bug classes.

2.8 Machine Learning for Source Code

The techniques discussed for the remainder of this thesis may focus on bug detection however nothing limits them from being applied to other software engineering tasks. In fact there is a whole research area that focuses on designing and applying ML and NLP techniques on source code. We would like to first make the reader aware that all the models discussed in the previous section also belong to this more general area of machine learning for source code. As would be expected a lot of the models in this greater category are also based or inspired by the naturalness hypothesis (Hindle et al., 2012). We will next focus our attention to a wide range of applications that have seen benefit via machine learning for source code models and highlight some key papers. However, we note that we will not go through every possible application of such models but only some major essential ones.

Recommender systems (Robillard et al., 2010) aim at making useful suggestions for a software engineering task. Such systems can for example facilitate tasks like code completion suggestions and code review recommendations. Code completion is the most used IDE feature (Amann et al., 2016). Bruch et al. (2009) extract features from code context to suggest completions for method invocations and constructors. Proksch et al. (2015) used Bayesian graphical models Cooper and Herskovits (1992) to improve accuracy upon the same task. Hindle et al. (2012) use a token level n-gram LM, which utilizes n-1 tokens to represent the completion context at each location. Franks et al. (2015); Tu et al. (2014) used a cache n-gram LM to improve performance, taking advantage of the observation that the cache acts as a domain adapted n-gram. Nguyen et al. (2013c) improve accuracy by augmenting the completion context with semantic information. Raychev et al. (2014) augment context with formal properties of the code, which limits incorrect but still probable API call suggestions. Their method was the first to aim towards statistical synthesis of single statements. Bielik et al. (2016); Maddison and Tarlow (2014) create suggestion models based on AST-level LMs. Movshovitz-Attias and Cohen (2013) use topic-like graphical model that captures context information to assist comment completion given a source code snippet. Allamanis et al. (2014, 2015a, 2016) focus on recommender systems that suggest variable, method, and class names based on context by surrounding tokens.

Another useful application is inferring coding conventions. These are syntactic constraints that are not strictly imposed by a programming language's grammar. Thus they affect specific choices like formatting or variable naming e.g., *CamelCase*. Many models in this category focus upon source code's surface structure, e.g., tokens or syntax. Using source code as data for learning they are able to quantify uncertainty over conventions as well as infer them. Allamanis et al. (2014, 2015a, 2016); Bavishi et al. (2018) learn and suggest variable, method,and class naming conventions. Allamanis et al. (2018); Allamanis and Sutton (2014) learn to mine source code idioms, which are conventional syntactic and semantic patterns of code constructs. Parr and Vinju (2016) learn how to format source code by using a set of hand-crafted features derived from the AST and a k-NN classifier.

The next category of applications is concerned with identifying buggy code. As we have already discussed the relevant techniques we point the reader back to Section 2.7.

Another group of applications revolves around code translation, transplantation and identifying code clones. Karaivanov et al. (2014); Nguyen et al. (2015) improve upon usual statistical machine translation models by adding semantic constraints to the translation process in order to reduce the production of invalid code. Code clones (Min and Li Ping, 2019) are similar code snippets in different locations of a code base and it is essential to have the ability identify them. It is an often phenomenon for developers to copy code during development from different parts of the code base. However, this creates the need for fixes such as renaming variables and may result in code clones. Aiming to automate such cleanups Allamanis and Brockschmidt (2017) use structured prediction and distributed representations to adapt a pasted snippet's variables into the target context. Their method relies only on probabilistically representing semantic information and ignores external information like tests.

The applications also extend to translation of code into other types of text such as natural language or pseudocode or even from buggy code to fixed one. Oda et al. (2015) learn to translate Python code to pseudocode written in natural language aiming to produce a more readable version of the code. Iyer et al. (2016) learn to summarize source code as text based on machine translation techniques. Movshovitz-Attias and Cohen (2013) learn to generate code comments based on n-gram and topic models. However, applications are not limited to just one direction but extend to the reverse one. Gulwani et al. (2017b) learn to convert natural language to Excel macros, Gvero and Kuncak (2015) to Java Expressions, Lin (2017); Lin et al. (2018) to shell commands, Quirk et al. (2015) to simple if-then programs, Kushman and Barzilay (2013) to regular expressions, and Zhong et al. (2017) to SQL queries. Yin and Neubig (2017) present a neural architecture for code generation. Finally, (Chen et al., 2019a; Tufano et al., 2018a) use neural machine translation in order to learn to convert buggy statements into fixed ones. A more in depth look at methods for generating natural language from source is also available in the literature (Neubig, 2016).

The last group of applications revolves around program synthesis (Gulwani et al., 2017a), which aims to generate a full or partial program from a given specification. Some program synthesis methods are based on programming by examples and use ML methods to synthesize code. Liang et al. (2010) use graphical models to learn common patterns of programs along similar tasks so that search in program synthesis can better operate. Menon et al. (2013) learn a parameterized probabilistic context free grammar to perform faster synthesis. Singh and Gulwani (2015) learn a classifier using features from the synthesized program that attempts to predict the correct program and utilize it for reranking of the synthesis suggestions. Other researchers have focused on program induction. We point the reader to the appropriate studies that compare such methods (Devlin et al., 2017b; Gaunt et al., 2016).

For a more detailed overview of this area, we point the reader to an excellent survey (Allamanis et al., 2018a) as well as two repositories which aim at archiving all publications regarding machine learning on source code.[5,6]

[5]https://ml4code.github.io/papers.html
[6]https://github.com/ml4code/ml4code.github.io

Chapter 3

Scalable Open-Vocabulary Neural Language Models for Code

"The measure of intelligence is the ability to change."

–Albert Einstein

"I'm still learning."

–Michelangelo

As outlined in Chapters 1 and 2, a new promising direction in software engineering (SE) is to utilize natural language processing and machine learning methods to build source code models. These models have been utilized in a variety of tasks. Some examples are systems to suggest readable names (Allamanis et al., 2015a), summarize source code (Allamanis et al., 2016; Iyer et al., 2016), predict bugs (Pradel and Sen, 2018), detect code clones (White et al., 2016), generate comments (Hu et al., 2018), and perform variable de-obfuscation (Bavishi et al., 2018). The above are only just a few examples of problems for which statistical language modelling techniques have been applied on large source code corpora to create powerful source code models that tackle the problem.

In this chapter[1], we argue that a major issue of these techniques is that code introduces new vocabulary at a far higher rate than natural language, as new identifier names proliferate. Both large vocabularies and out-of-vocabulary issues severely affect Neural Language Models (NLMs) of source code, degrading their performance and rendering

[1]Most of the material in this chapter has appeared before in (Karampatsis et al., 2020) which is a consolidation of two unpublished works (Babii et al., 2019; Karampatsis and Sutton, 2019) but also introduced several improvements. The material in (Karampatsis et al., 2020) has been accepted for publication and will appear in the proceedings of ICSE2020.

them unable to scale. This chapter addresses this issue by: 1) studying how various modelling choices impact the resulting vocabulary on a large-scale corpus of 13,362 projects; 2) presenting an open vocabulary source code NLM that can scale to such a corpus, 100 times larger than in previous work; and 3) showing that such models outperform the state of the art on three distinct code corpora (Java, C, Python). We will next more thoroughly introduce the problem and provide motivation regarding it and the proposed solution.

Many works have taken advantage of the "naturalness" of software (Hindle et al., 2012) to assist software engineering tasks, including code completion (Raychev et al., 2014), improving code readability (Allamanis et al., 2014), program repair (Chen et al., 2018; Santos et al., 2018), identifying buggy code (Ray et al., 2016a) and API migration (Gu et al., 2017), among many others (Allamanis et al., 2018a). These approaches analyze large amounts of source code, ranging from hundreds to thousands of software projects, building machine learning models of source code properties, inspired by techniques from natural language processing (NLP).

When applying any NLP method to create any type of software development tool, a crucial early decision is how to model software's vocabulary. This is all the more important because, unlike in natural language, **software developers are free to create any identifiers they like, and can make them arbitrarily complex**. Furthermore identifiers are often compound words (e.g., `thisIdentifierHas6WordsAnd2Numbers`), causing an explosion of possible identifiers. Because of this fundamental fact, any model that is trained on a large-scale software corpus has to deal with an extremely large and sparse vocabulary (Section 3.1). Rare words can not be modelled effectively. Furthermore, if identifiers were not observed in the training set, many classes of models cannot predict them, which is known as the *out-of-vocabulary (OOV) problem.* Hellendoorn and Devanbu observe this issue for the task of language modelling, showing that a neural language model has difficulties scaling beyond as few as a hundred projects (Hellendoorn and Devanbu, 2017). Given that neural approaches are the state-of-the-art in NLP, finding ways to scale them to a larger software corpus is a very important goal.

The *first contribution* is a thorough study of the effects of the vocabulary design choices that must be made when creating any NLP model of software (Section 3.3). The vocabulary design choices we study include how to handle comments, string literals, and white space; whether to filter out infrequent tokens; and whether and how to split *compound tokens*, such as names that contain camel case and underscores. We examine how these choices affect the vocabulary size, which affects the scalability of models, and how they affect the OOV rate, that is, how often the vocabulary fails to include names that appear in new projects. We find that the choices have a large impact, leading to variations in vocabulary size of up to *three orders of magnitude*. However, we find that the most common ways to reduce vocabulary that were previously considered in the software engineering literature, such as splitting identifiers according to underscores and case, are not enough to obtain a vocabulary of a manageable size; advanced approaches such as adaptations of the Byte-Pair Encoding (BPE) algorithm (Gage, 1994; Sennrich et al., 2015) are needed to reach this goal and deal with the OOV problem.

This empirical study motivates our *second contribution*. Drawing on our results, we develop a large-scale open-vocabulary NLM for source code (Section 3.4). To our knowledge, this is the first BPE NLM for source code reported in the literature. This NLM model leverages BPE, beam search, and caching to both keep vocabulary size low and successfully predict OOV tokens. We show that this NLM is able to scale: we train it on up to 13,362 software projects, yielding the largest NLM trained on source code we are aware of.

Finally, in our *third contribution* we extensively evaluate our NLM (Sections 3.5–3.7). We show that the open-vocabulary NLM outperforms both n-gram LMs and closed vocabulary NLMs for the task of code completion for several languages (Java, C, and Python). To show that the improvement in language modelling transfers to downstream SE tasks, we conduct an experiment similar to Ray et al. (2016a), who showed that language models can be used to highlight buggy code. Indeed, we find that our open-vocabulary NLM is more effective than previous LMs at highlighting buggy code.

More broadly, these contributions may impact future development software tools. First, source code LMs have been used in a diverse variety of tools well beyond the obvious application of autocompletion, ranging from code readability (Allamanis et al., 2014) to program repair (Chen et al., 2018). Our improved NLM could lead to improvements to all of these tools. Second, recent results in NLP (Devlin et al., 2018; Howard and Ruder, 2018; Peters et al., 2018) show that NLMs can be used as upstream tasks in transfer learning, leading to state-of-the-art improvement in downstream tasks: for instance, a model can be pre-trained as an NLM, and later on fine-tuned as a classifier. Improved NLM architectures could lead to improved downstream classifiers, especially if the labelled data is scarce. While transfer learning from language models has been applied in software engineering (Robbes and Janes, 2019), it has not been applied to source code due to the aforementioned vocabulary issues. Finally, the general insights about vocabulary design that we study are not specific to NLMs, but arise whenever we build development tools by applying NLP methods to source code.

We briefly describe the artifacts used in this work and how to obtain them in Section 3.8 and conclude the chapter in Section 3.9.

3.1 Background and Related Work

3.1.1 Language Modeling in NLP

A language model (LM) estimates the probabilities of sequences of words based on a training corpus. In NLP, these models have been applied to tasks such as speech recognition (Creutz et al., 2007) and machine translation (Jean et al., 2015). Early language models were based on n-grams: the probability of a token is computed based on the $n-1$ previous tokens in the sequence. These had success in NLP applications, but have two issues. First, they operate on small amounts of previous context, with n often ranging from 3 to 6 (e.g., $n = 6$ for Java (Hellendoorn and Devanbu, 2017)). Increasing n does not scale well if the vocabulary is large: for a vocabulary of size m, there are m^n possible n-grams. Second, they suffer from data sparsity: not all possible n-grams exist in the corpus. Smoothing (Chen and Goodman, 1999) alleviates—but does not eliminate—the issue.

The current state-of-the-art in NLP is *neural language models (NLM)* (Bengio et al., 2003). NLMs represent words in a continuous vector space, such that words that are semantically similar are close in vector space (Mikolov et al., 2013b), allowing the model to infer relationships between words, even if they do not appear in a specific context during training. This allows these models to better deal with data sparsity, leading to better performance. Given enough training data some arithmetic operations are semantically meaningful (e.g., the closest vector to the sum of `"Germany"` and `"capital"` is the vector corresponding to `"Berlin"` (Mikolov et al., 2013b)). Current NLMs are based on architectures such as recurrent neural networks (RNN) (Mikolov et al., 2010), long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997), or Transformer (Vaswani et al., 2017) that model *long range* dependencies: a study of LSTM NLMs showed that they use context as large as 250 words (Khandelwal et al., 2018), much longer than n-grams.

3.1.2 Difficulties with Large Vocabularies

ML models in general, and NLMs in particular, do not handle large vocabularies well. This is for several reasons:

Scalability. During pre-processing, each word is converted to a numerical representation, first via one-hot-encoding, producing (sparse) vectors of length equal to the vocabulary. NLMs then convert these to word embeddings, dense word vectors of much smaller dimensions (usually in the hundreds), in their first layer. For a vocabulary of size m and embeddings of size n, the embedding layer is a dense matrix of size $m \times n$. A large m (e.g., 100,000 or more) affects the memory required by the model as well as the amount of computation required for training. The output of an NLM is a prediction over the next token, which is a probability distribution over the entire vocabulary. This must be computed once for each token in the training corpus many times during training. This can be prohibitively slow for large vocabularies (Bradbury et al., 2016; Jozefowicz et al., 2016).

Out-of-vocabulary (OOV). In traditional, *closed-vocabulary* models, the vocabulary must be known in advance and will be built based on the training corpus. Any new word encountered at test time, called out-of-vocabulary words, will not be able to be one-hot encoded as the resulting vector would exceed the expected dimensions. A common workaround is to have a specific *unknown* token, and replace any word not previously seen by this token. This loses information, making the NLM unable to predict any new token, which is particularly problematic for source code.

Rare Words. Deriving meaningful embeddings for rare words is difficult since there is very little data to work with. Gong et al. (2018)show that the property that semantically similar words have similar embeddings does not hold for rare words: they hypothesize that since the words are rarely seen, the embeddings are rarely updated and thus stay close to their initialized values. This issue is likely to impact performance: a very large vocabulary has been shown to negatively impact it, particularly with OOV words (Jean et al., 2015).

While the Softmax issue is specific to Language Modeling, *any Neural Model operating on text will be affected by the other issues.* Further, Neural Models using more structural information (e.g., ASTs Alon et al. (2019b) or graphs Allamanis et al. (2018b)) also need to model the textual nature of code.

3.1.3 Handling Large Vocabularies in NLP

An *open vocabulary model* is not restricted to a fixed-sized vocabulary determined at training time. For instance, a character LM predicts each word letter by letter: its vocabulary is the set of characters; the OOV issue vanishes. However, it needs to model longer dependencies than a word NLM, impacting performance. Models using *subword units*, or *subwords*, combine the strengths of character and token LMs. A subword unit is a sequence of characters that occurs as a subsequence of some token in the training set; the model outputs a sequence of subword units instead of a sequence of tokens. Many NLP models have used linguistically-motivated subwords (Bazzi, 2002; Creutz et al., 2007; Luong et al., 2013; Mikolov et al., 2012). Mikolov et al. (2012) found that subword models improved on character models . Sennrich et al. (2015) adapt the Byte-Pair Encoding (BPE) algorithm to decompose words in subwords, improving rare word translation. Bojanowski et al. (2017) represent words as bags of characters n-grams to compute word embeddings even for OOV words. Kim et al. (2016) combine a character CNN with a NLM. Vania and Lopez compare LMs (words, morphs, character n-grams, BPE) on several languages (Vania and Lopez, 2017).

Another approach to the OOV problem are *cache models and copy mechanisms* (Allamanis et al., 2016; Grave et al., 2016; Merity et al., 2016), which allow the model to re-use words that have appeared previously. This helps with the OOV problem, because such models can copy words that are not in their fixed vocabulary, but it does not help the *first* time an OOV word appears.

3.1.4 Language Modeling and Vocabulary in SE

Language Models in Software Engineering (SE). Seminal studies have laid the groundwork for the use of language models on source code: Gabel and Su show that software is very repetitive (Gabel and Su, 2010), which motivates the use of statistical modelling for code. Hindle et al. (2012) compare software to natural language, finding that software is much more repetitive than natural language ; they build language models of source code, finding applications in code completion. Nguyen et al. (2013c) augmented n-gram LMs with semantic information such as the role of a token in the program, e.g., variable, operator, etc. Tu et al. (2014) find that software is even more repetitive taking local context into account. Rahman et al. (2019) find that while some aspects of software are not as repetitive as previously thought (non-syntax elements), others are even more so (API sequences). Casalnuovo et al. (2018) find that even accounting for syntax, code is more repetitive than English. Other models of source code include probabilistic higher order grammars (PHOG) (Bielik et al., 2016), which use ASTs, and several types of RNNs, including LSTMs (Dam et al., 2016; Hellendoorn and Devanbu, 2017; White et al., 2015).

SE Applications of Language Models. Probabilistic code models have enabled many applications in software engineering (see Allamanis (2017) for a survey). One example is recommender systems aiming to aid developers in writing or maintaining code: (Hindle et al., 2012) used a token-level LM for code completion, while later, (Franks et al., 2015) improved on performance with Tu's cache (Tu et al., 2014) and built a code suggestion tool for Eclipse. Another application are recommendation systems for variable, method, and class names (Allamanis et al., 2014, 2015a, 2016) that employ relevant code tokens as the LM context. (Campbell et al., 2014) used n-gram language models to detect syntax error locations in Java code, and later used an NLM for the same purpose (Santos et al., 2018). (Ray et al., 2016a) showed that buggy code has on average lower probability than correct code, and that LMs can spot defects as effectively as popular tools such as FindBugs.

Several approaches use neural machine translation, in which an encoder LM is paired to a decoder LM. Examples include recovering names from minified Javascript code (Bavishi et al., 2018; Vasilescu et al., 2017), or from decompiled C code (Jaffe et al., 2018). Other applications include program repair (Chen and Monperrus, 2018b), learning code changes (Tufano et al., 2019), or generating source code comments (Hu et al., 2018). Gu et al. (2016) generate API usage sequences for a given natural language query. They then learn joint semantic representations of bilingual API call sequences to support API call migration Gu et al. (2017). Yin et al. (2018a) mine pairs of natural language and code from Stack Overflow to support tasks such as code synthesis from natural language.

Large vocabularies in SE. The majority of models of source code used closed vocabulary models. Hellendoorn and Devanbu (2017) rightly notice that NLMs trained on a software corpus would struggle due to vocabulary size, because of the nature of identifiers which are the bulk of source code. To produce an NLM that can be trained in a reasonable amount of time, Hellendoorn and Devanbu impose drastic restrictions which would be expected to reduce predictive accuracy, restricting the training set to 1% of the original corpus Allamanis and Sutton (2013) and the vocabulary to only include words which occur more than 5 times. Even so, the resulting vocabulary size is still exceeds 76,000 words. Moreover, the prediction performance of a NLM is significantly hurt when it has to predict words that are members of its vocabulary. Similarly, Pradel and Sen (2018) had a large vocabulary of 2.4 million unique tokens: they limited it to the 10,000 most common tokens to reduce inaccuracies due to rare words.

To limit this issue, previous work has segmented identifiers via a heuristic called *convention splitting*, which splits identifiers on camel case and underscores (Allamanis et al., 2015a). Even though this segmentation can handle *some* OOV tokens, it is limited to combinations of subtokens appearing in the training set and thus unable to achieve a truly open vocabulary. Additionally, many of these subtokens are still infrequent, which hinders the model's ability to assign high scores to their compositions. For example, despite using convention splitting, the implementation of code2seq from (Alon et al., 2019a) only keeps the 190,000 most common vocabulary words.

Several studies have empirically compared different techniques for automatically splitting identifiers (Enslen et al., 2009; Hill et al., 2014). These works consider the somewhat different problem of splitting identifiers into words in a way that matches human judgment. Subword units may not necessarily produce words that humans recognize, but they can be trivially reassembled into complete tokens before they are shown to a developer. Stemming (Willett, 2006) has also been used to reduce the number of vocabulary words by only keeping their roots; this is however destructive. Malik et al. (2019) combined convention splitting and stemming for type prediction.

Few SE approaches use caches. Tu et al. (2014) and Hellendoorn and Devanbu (2017) use n-gram caches. Li et al. (2018) augment an RNN with a copy mechanism based on pointer networks (Vinyals et al., 2015) to improve OOV code completion. While it can reuse an OOV word after seeing it, it cannot predict the word's first use, learn its representation, or learn its dependencies, unlike our model. Copy mechanisms were also used for program repair (Chen et al., 2018), and method naming (Allamanis et al., 2016).

3.2 Datasets

We use code corpora from three popular programming languages: Java, C, and Python. We choose these languages because they have differences that could affect the performance of LMs. Java has extensively been used in related work (Allamanis and Sutton, 2013; Dam et al., 2016; Hellendoorn and Devanbu, 2017; Hindle et al., 2012; Nguyen et al., 2013c; Tu et al., 2014). Unlike Java, C is not object oriented, procedural, and makes it possible to write very terse code.[2] It's syntax however is very similar to Java. It is also used extensively for low-level system development, a domain in which Java is not used. Python is a multi-paradigm dynamic scripting language with little use of static typing and no strong typing but is also object-oriented. Furthermore, Python has a very subtle difference of using whitespace to delimit code blocks instead of brackets. Last python programmers tend to write very idiomatic code as doing otherwise is strictly discouraged since code written in the "Pythonic" way executes much faster as the language has evolved in order to better support these idioms and a small set of popular libraries is used by a plethora of Python programs. These characteristics of C and Python could be expected to affect LM performance, making them interesting to consider alongside Java. However the reader should be cautioned that it is not our intent to create a comparison between

[2]For examples, see https://www.ioccc.org/.

the languages such as which one is more predictable, which is more terse, etc.S nor should the results presented in this chapter be interpreted in this way. The first reason for this caution is that the training corpora have slightly different sizes across the different languages. Unfortunately, it does not seem possible todefine a fair notion of "same training set size" across programming languages, because tokens in one language might be more informative than others, e.g., Python code has a larger proportion of identifiers. Even if it were possible to do this, different languages have different standard libraries and are typically used to solve problems in different domains. All of these concerns pose serious threats to validity to any attempt to compare programming languages via language modelling, so we do not attempt to draw such conclusions here.

For Java we used the Java Github corpus of (Allamanis and Sutton, 2013), also used in (Hellendoorn and Devanbu, 2017). The C and Python corpora were mined following the procedure described in (Allamanis and Sutton, 2013); the C corpus was mined in (Dudoladov, 2013) and the Python corpus was mined in (Fiott, 2015). For lexical analysis we used the Java lexer implemented in (Hellendoorn and Devanbu, 2017)[3]; for C and Python we used the Pygments[4] library. Descriptive statistics are in Table 3.1.

For Python and C we sampled 1% of the corpus for validation and 1% for testing. Another 10% of the corpus was sampled as a separate data set to learn subword encodings with BPE. The rest of the data was used for training. We also report results on a smaller subset of 2% of our full training set. For Java, we used a slightly different procedure to make our experiment comparable to a previous study (Hellendoorn and Devanbu, 2017). We divide the data into five subsets as in the other two languages. The validation and test sets are the same as in (Hellendoorn and Devanbu, 2017), and our “small train” set is the same as their training set. To obtain the full Java train set, we collect all of the files in the Java Github corpus that do not occur in the validation or test set. Of these, we sampled 1000 random projects for the subword encoding data set, and the remaining projects were used as the full train set.

In the vocabulary study, both training sets and test sets are used. To train LMs, we preprocess the corpora to match (Hellendoorn and Devanbu, 2017), replacing non-ASCII character sequences such as Chinese ideograms inside strings with a special token (`<non-en>`), removing comments, and keeping strings. Note that the lexer in (Hellendoorn and Devanbu, 2017) replaces all strings with length of 15 characters or more with the empty string. In Python, we do not add any special tokens to represent whitespace.

[3]https://github.com/SLP-team/SLP-Core

[4]http://pygments.org/docs/lexers/

Table 3.1 Corpus statistics for each code corpus.

	Java		**C**		**Python**	
	Tokens	Projects	Tokens	Projects	Tokens	Projects
Full	1.44B	13362	1.68B	4601	1.05B	27535
Small	15.74M	107	37.64M	177	20.55M	307
BPE	64.84M	1000	241.38M	741	124.32M	2867
Valid.	3.83M	36	21.97M	141	14.65M	520
Test	5.33M	38	20.88M	73	14.42M	190

3.3 Modeling Vocabulary

In this section we study a series of modelling choices for source code vocabulary. The work in this section was mainly performed by my colleagues (Babii et al., 2019) and I only took part in discussions regarding what vocabulary choices to make and which experiments should be run. The vocabulary choices studied here may be implicitly made by researchers, with or without evaluating alternatives; they may not always be documented in their studies. By making the choices explicit, we can study their impact on the vocabulary. We report results on Java; similar results can be observed for C and Python. Our evaluation criteria are:

Scalability Training of models should scale to thousands of projects. Scalability is influenced by the vocabulary size (number of unique words) and the corpus size (number of tokens). We report both metrics on our full java training set, and compare them to a baseline with percentages. For instance: 11.6M, 100% and 2.43B, 100%.

Information loss Models should be able to represent the original input as much as possible; out-of-vocabulary (OOV) tokens are particularly undesirable. We build a vocabulary on the training set, and compare it with the test set vocabulary; we report the percentage of new vocabulary words seen in the test set. As large vocabularies do not scale, we report OOV for the unfiltered vocabulary, and a smaller vocabulary (the 75,000 most frequent words, as in (Hellendoorn and Devanbu, 2017)). To show trends, we also plot OOV for: full vocabulary, 200K, 100K, 75K, 50K, and 25K. Such as: 42%, 79%, .

Word frequency As rare words have worse representations than frequent ones (Gong et al., 2018), increasing word frequency is desirable. Different modelling choices can increase or decrease the number of rare words. We report the percentage of the vocabulary that has a frequency of 10 or less, and plot a bar chart showing the percentage of vocabulary with frequencies of 1000+, 1000–101, 100–11, 10–2, and 1. For instance: 83%, .

Baseline: 11.6M, 100% • 2.43B, 100% • 42%, 79%, • 83%,
Our baseline is a vocabulary of unsplit tokens, except strings and comments that are split by whitespace (not doing so roughly doubles the vocabulary). This vocabulary is extremely large: more than 11 million unique words on Java-large. The OOV rate on the test set exceeds 40% with the full vocabulary, showing that developers do create many new identifiers. The most common way to shrink vocabulary is to replace infrequent tokens with `<unk>`. Doing so further worsens OOV issues: after reducing the vocabulary to a more manageable 75K, close to 80% of the test vocabulary is unseen in the training set. Many words are infrequent: 83% of vocabulary words have a frequency of 10 or less, with 25% occurring only once.

3.3.1 Filtering the vocabulary

The most common aused in order to reduce the vocabulary is to filter out uncommon tokens (less frequent than a threshold k) and replace them with an `<unk>` token. This is extremely simple to implement in practice, but loses extensive amounts of information and is ineffective: Keeping only the 100K most frequent words (in line with Hellendoorn and Devanbu's 76K Hellendoorn and Devanbu (2017)) means that 4% of the corpus is replaced by `<unk>`, which would be the 6th most frequent token and, if applied indiscriminately, it is rhard to gauge its impact. As such iltering is a destructive modelling choice loses extensive amounts of information: it thus needs to be thoroughly justified. The simplest would be to filter vocabulary items that are deemed less important.

English. 11.4M, 98% • 2.43B, 100% • 35%, 76%, • 83%,
Source code can contain many non-English words in identifiers, strings, and comments, either because developers use other languages, or for testing or internationalization purposes. Handling multilingual corpora is an NLP research topic in itself; we evaluate the simplifying assumption to limit a corpus to English. This is not trivial: dictionary-based heuristics have too many false positives (e.g., acronyms). We use a simple heuristic: a word is non-English if it contains non-ASCII characters. This is imperfect; "café", "naïve", or "Heuristiken" are misclassified. Non-English words are replaced with a `<non-en>` placeholder. Even then, the vocabulary shrinks by only 2%, while OOV drops by only 3% at 75K.

Whitespace. 11.4M, 98% • 1.89B, 78% • 35%, 76%, • 83%,
Some applications (e.g., pretty-printers [3]) may care about the layout of source code. Others may not, giving importance only to syntactic or semantic aspects (unless code layout is syntactically important, such as in Python). Filtering out whitespace reduces the vocabulary only by a handful of tokens, but reduces corpus size by 22% (1.89B tokens).

Comments 10.8M, 93% • 1.26B, 52% • 38%, 78%, • 83%,
Comments often contain natural language, which is much less repetitive than code. While tasks such as detecting self-admitted technical debt (da Silva Maldonado et al., 2017) rely on comments, others do not. Replacing comments by placeholder tokens (e.g., `<comment>`) significantly reduces corpus size (a further 26%), but its effect on vocabulary is limited (6%, given that comments are already split on whitespace).

Strings. 9.5M, 82% • 1.15B, 47% • 39%, 78%, • 83%,
Similarly, string literals can be filtered, replacing them by a placeholder token like `<string>`. This does not reduce corpus size as much (a further 5%), but shrinks vocabulary a further 11%, close to 9.5 million words. This is still extremely large. We also evaluate the configuration used in Hellendoorn and Devanbu (2017): strings are kept, unsplit, but strings longer than 15 characters are replaced by the empty string. For consistency with previous work, we use it as **new baseline**. It increases vocabulary, OOV and infrequent tokens rate: 10.9M, 94% • 1.15B, 47% • 39%, 80%, • 84%,

Full token vocabularies range in the millions, and hence do not scale. OOV and frequency issues are extremely important.

3.3.2 Word Splitting

Identifiers are the bulk of source code and its vocabulary. While new identifiers can be created at will, developers tend to follow *conventions*. When an identifier is made of several words, in most conventions, the words are visually separated to ease reading, either in `camelCase` or in `snake_case` (Binkley et al., 2009). Thus, an effective way to reduce vocabulary is to *split* compound words according to these word delimiters, as was done by (Allamanis et al., 2015a).

The decision whether to split compound words or not has important ramifications. First, it introduces additional complexity: the LM can no longer rely on the assumption that source code is a sequence of tokens. Instead, compound words are predicted as a sequence of subtokens, albeit in a smaller vocabulary. Second, subtokens increase the length of the sequences, making it harder to relate the current subtokens to the past context, as it increases in size. This makes the approach unviable for n-grams as n would need to increase significantly to compensate.

Splitting tokens has advantages: most obviously, the vocabulary can be much smaller. Consequently, the OOV rate is reduced. Third, a model may infer relationships between subtokens, even if the composed word is rare, as the subtokens are more common than the composed word. Finally, using subtokens allows a model to suggest *neologisms*, tokens unseen in the training data (Allamanis et al., 2015a).

Splitting. 1.27M, 12% • 1.81B, 157% • 8%, 20%, • 81%,
Word splitting via conventions drastically reduces the vocabulary, by a close to an order of magnitude (slightly more than a million words), at the cost of increasing corpus size by 57%. The impact on the OOV rate is also very large, as it decreases by a factor of 5 (in the unfiltered case; for a vocabulary of 75K it is a factor of 4). However, the effect on word frequency is limited, with only 3% more words that are more frequent than 10 occurrences.

Case. 1.09M, 10% • 2.16B, 187% • 9%, 21%, • 83%,
Most commonly, words in different case (e.g., `value`, `Value`, `VALUE`) will be distinct words for the LM. This could increase the vocabulary, but removing case loses information. A possible solution is to encode case information in separator tokens (e.g., `Value` becomes `<Upper> value`; `VALUE` becomes `<UPPER> value`). at the cost of increasing sequence length. Case-insensitivity does decrease the vocabulary, but not by much (a further 2%), while corpus size increases significantly (a further 30%). Thus, our following configurations do not adopt it: our **new baseline** keeps case.

Word splitting is effective, but the vocabulary is still large (a million words). OOV and frequency issues are still important.

3.3.3 Subword splitting

As splitting on conventions is not enough, we explore further.

Numbers. 795K, 63% • 1.85B, 102% • 6%, 18%, • 72%,
Numeric literals are responsible for a large proportion of the vocabulary, yet *their* vocabulary is very limited. Thus, an alternative to filtering them out is to model them as a sequence of digits and characters. This yields a considerable decrease in vocabulary with our previous baseline (37%), for only 2% increase in corpus size. For OOV, there is

a slight improvement for a 75K vocabulary (2%), as well as for frequency (28% of words occur 10 times or more).

Spiral. 476K, 37% • 1.89B, 104% • 3%, 9%, • 70%,
Several approaches exist that split a token into subtokens, but go beyond conventions by using Mining Software Repositories techniques, such as Samurai (Enslen et al., 2009), LINSEN (Corazza et al., 2012), Spiral (Hucka, 2018), or even neural approaches (Markovtsev et al., 2018). We applied the Spiral token splitter, which is the state of the art. We observed a further 26% reduction of the vocabulary, for a 2% increase in corpus size compared to number splitting. Spiral was also very effective in terms of OOV, with 9% of unseen word when the vocabulary is limited to 75K, and 3% when unfiltered (476K words). The impact on frequency was limited. Even if this is encouraging, the OOV rate is still high.

Other approaches. Stemming (Willett, 2006) can reduce vocabulary size, but loses information: it is not always obvious how to recover the original word from its stem. We found that applying stemming can further reduce vocabulary by 5%, which does not appear to be a worthwhile tradeoff given the loss of information. Another option is *character models* that achieve an open vocabulary by predicting the source file one character a time. OOV issues vanish, but unfortunately, this drastically inflates sequence lengths, so a character model is not desirable.

While these strategies are effective, they do not go far enough; vocabulary stays in the hundreds of thousands range. There are still OOV issues for unseen data; most words are uncommon.

3.3.4 Subword splitting with BPE

The final alternative we evaluate is subword segmentation via Byte-Pair Encoding (BPE). BPE is an algorithm originally designed for data compression, in which bytes that are not used in the data replace the most frequently occurring byte pairs or sequences (Gage, 1994). In subword segmentation, this corpus is represented as a sequence of subwords starting with characters (the smallest possible subwords). Special end-of-token `</t>` symbols are added to allow us to convert from a sequence of subword units back into a sequence of tokens with ease. The most frequent symbols in the vocabulary are merged to form a new one until a maximum number of merge iterations has been reached. So, instead of bytes and byte sequences we have character and character sequences. The approach was first adapted to build NMT vocabularies (Sennrich et al., 2015): the most frequently occurring sequences of characters are merged to form new vocabulary words. As such the algorithm's objective is to learn the least entropic segmentation into subwords. After the maximum number of merger iterations has been reached and the final vocabulary has been extracted, we utilize it to segment any encountered words (tokens in the case of source code) into subwords based on their frequency (favouring the most frequent one). This is achieved by first adding the `</t>` and splitting the word into characters. Then, we apply the most frequent merge operation in the vocabulary iteratively until we cannot apply any more merges. For example the identifier `DefaultMutableTreeNode` is segmented into the subwords `Default • Mutable • TreeNode</t>` which all three are entries in the final vocabulary. To help the reader with better understanding what BPE's objective is and how actual segmented code looks Figure 3.1 illustrates a small Java function and its segmentation to BPE subwords. More details on how the algorithm operates are provided later in this section.

BPE builds up the vocabulary of subwords iteratively, at each iteration a training corpus is segmented according to the current vocabulary. The initial vocabulary contains all characters in the data set and `</t>`, and the corpus is split into characters and `</t>`. Then, all symbol pairs appearing in the vocabulary are counted. All the appearances of the most frequent pair (S_1, S_2) are replaced with a unique new single symbol S_1S_2, which is added to the vocabulary, without removing S_1 or S_2 (which may still appear alone). This procedure is called a merge operation. The algorithm stops after a given maximum number n of merge operations; this is the only parameter. The final output of the algorithm is (1) the new vocabulary, which contains all the initial characters plus the symbols created from the merge operations, and (2) the ordered list of merge operations performed in each iteration. New data is segmented by splitting it into characters and merging in the same order.

The vocabulary of subword units is learned before training the NLM by segmenting a corpus of code. This is done in such a way that more frequent character n-grams are more likely to be included in the vocabulary of subwords units. This strategy results in a core vocabulary of subword units that occurs frequently across different projects and captures statistical patterns of characters within identifiers.

In order to learn the segmentation we use a modification of *byte pair encoding (BPE)* Gage (1994). BPE is a data compression algorithm that iteratively finds the most frequent pair of bytes in the vocabulary appearing in a given sequence, and then replaces it with a new unused entry. Sennrich, Haddow, and Birch Sennrich et al. (2015) first adapted the algorithm for word segmentation so that instead of merging pairs of bytes, it merges pairs of characters or character sequences. The learned segmentation was used in their neural translation system and resulted in improved translation of rare words.

Java Code:

```java
public AttributeContext(Method setter, Object value) {
    this.value = value;
    this.setter = setter;
}
```

Subword Units:

```
public</t> Attribute Con text</t> (</t> Method</t> set ter</t> ,</t> Object</t> value</t> )</t> {</t>
this</t> .</t> value</t> =</t> value</t> ;</t> this</t> .</t> set ter</t> =</t> set ter</t> ;</t> }</t>
```

Figure 3.1 Example of Java code as a list of subword units.

The only parameter BPE needs is the number of merges (n) to do. Then, it finds the most common pair of successive items in the corpus (initially characters, then subwords). This pair is merged in a new token which is added to the vocabulary; all occurrences of the pair are replaced with the new token. The process is repeated n times. As in Sennrich et al. (2015) we do not consider merging pairs that cross token boundaries. That is, where the merged token would contain `</t>` internally, so that every subword unit is a character subsequence of a token in the data. Common tokens, such as `public` in Figure 3.1 are assigned a full subword unit, whereas less common tokens, such are `setter`, are divided into smaller units, more common units (such as roots, prefixes, and suffixes). This helps with data sparsity issues.

BPE has several advantages. First, no word is OOV; the vocabulary always contains all single characters, so unknown words at test time can be represented using those subwords, if no longer subwords apply. Second, the vocabulary dynamically adapts to the frequency of the sequences: common sequences will be represented by a single word (eg, `exception`), while rare ones will be segmented into more common subword units (such as roots, prefixes and suffixes); this helps with sparsity issues. Finally, BPE allows for fine-grained control of vocabulary size, by tuning the number of merge operations. A larger vocabulary will have more complete words and less sequences, smaller ones will have longer sequences. An example of a Java code snippet segmented into subwords is shown in Figure 3.1. We computed BPE for 1K, 2K, 5K, 10K and 20K merges, on a held-out set of 1K project.

BPE Subwords. 10K, 1% • 1.57B, 137% • 0%, <u>0%</u>, • 1%,
We apply BPE (10K merges) to our Java corpus with preprocessed as in (Hellendoorn and Devanbu, 2017), which we use as a baseline for comparison. As expected, the OOV issues vanish, even for an *extremely small vocabulary.* The corpus size grows, but not more than previous choices we explored. Since BPE merges based on frequency, the

resulting subtokens, no matter their size, are frequent: more than 97% of the remaining words occur more than 1,000 times in the corpus, with very few words that are in the hundreds, and 1% less than ten. Lower amounts of merges result in a smaller vocabulary, at the cost of a larger corpus size. Our largest BPE vocabulary, 20K, is 575 times smaller than our initial baseline; our smallest is 11,500 times smaller.[5]

Qualitative examination. While the goal of BPE is not to produce human-readable tokens, we examine how closely the splits BPE produces match human ones. We inspected 110 random identifiers longer than 25 characters long, and provide anecdotal evidence of the types of splits produced by BPE. Our goal is not to provide strong evidence, but rather to give a sense to the reader of what BPE splits look like in practice. Some of these examples are shown in Table 3.2.

While some subwords are readable at BPE 1K (`File • Output • Service</t>`) or (`layout • inflater • service`), some subwords are unintuitive (`Default • M • ut • able • Tre • e • Node</t>`) or (`m • al • for • me • d • url • exception`), but look good at 5K (`Default • Mutable • TreeNode</t>`) or (`malformed • url • exception`). BPE handles rare words gracefully, producing longer sequences of shorter units as expected. Some examples include rare words due to typos (`in • cul • ded • template</t>`) or foreign words (`v • orm • er • k • medi • en • au • f • lis • ter</t>`). Some rare words are split in root and suffix (`Grid • ify</t>`), but some acronyms may be unexpectedly split (`IB • AN</t>`). Further, BPE can split words correctly without case information (`http • client • lib</t>`, at 5K), and improves with more merges (`httpclient • android • lib</t>`, at 10K). Some words have satisfying splits at low BPEs, yet improve as BPE increases (example e). Finally, the model degrades gracefully for non-English words: those are not out-of-vocabulary, just long sequences (example f).

We classified each of the 110 splits in 3 categories: *good* (reproduces the expected case split), *acceptable* (one word was split in root and prefix or suffix, such as Grid • ify, or an acronym was not well reconstructed, such as I • BAN), and *degraded* (one or more words split incorrectly, or in more than 2 parts). We found 7 degraded splits: 2 foreign words, 2 words with typos (`Fragement, INCULDED`), 1 with rare words (`TheImprisonedGourmet`), an all-lowercase sequence of 8 words, and a word were the split was unclear (`appirate`). Of the *good* splits, 11 were found at BPE 1K (including common words such as `exception`, `configuration`, or `attribute`), 51 at BPE 5K, 28 at BPE 10K, and 8 at BPE 20K. While BPE 1K is too small, 5K is competive, 10K is optimal, and 20K offers disminishing returns.

[5]Note that including non-ASCII characters grows the vocabulary by $\approx$ 5,000 words in each case; a solution is to apply BPE at the *byte* level, as done in (Radford et al., 2018)

Table 3.2 Examples of Byte-Pair Encoding Splits

Configuration	Token / BPE Split
a) Optimal at BPE 1K	
Original	LAYOUT_INFLATER_SERVICE
BPE 1K	layout • inflater • service
b) Optimal at 5K	
Original	MalformedURLException
BPE 1K	m • al • for • me • d • url • exception
BPE 5, 10, 20K	malformed • url • exception
c) Effect of typos	
Original	INCULDED_TEMPLATE
BPE 1, 5, 10, 20K	inc • ul • ded • template
d) Splitting without case	
Original	cmd_reloadquestconfig
BPE 1K	c • m • d • re • load • quest • config
BPE 5,10,20K	cmd • reload • quest • config
e) Continuous improvement	
Original	httpclientandroidlib
BPE 1K	http • client • android • li • b
BPE 5K	http • client • android • lib
BPE 10K	httpclient • android • lib
BPE 20K	httpclientandroidlib
f) Handling non-English words	
Original	vormerkmedienauflister
BPE 5K	vor • mer • k • medi • en • au • f • list • er
BPE 20K	vor • mer • k • medi • en • auf • lister
g) Impact of preserving case	
Original	alternativeEndpointsAndQueries
BPE 5k	alternative • end • points • and • queries
BPE 5k (case)	al • tern • ative • End • points • And • Qu • eries
BPE 10k (case)	alternative • End • points • And • Queries

Adding back strings and comments. Encouraged by these results, we increased the base vocabulary to include all strings and comments (split with spaces). We found that producing human-readable splits requires raising the amount of BPE merges to 10K or 20K. We note that our BPE also includes all numbers and literals: some sequences that were merged were common numbers. For some applications, it may be acceptable to filter uncommon literals with a low OOV threshold (e.g., 3). This may improve vocabulary further.

We find that adding words found in strings and comments appears to have little impact on BPE 5K and 10K, both of which slightly increase the size of the corpus by 1–2%. A vocabulary of 10K words is more than 1,000 times smaller than the initial configuration (11,357,210), at the cost of increasing the number of tokens in the corpus by a factor of 1.7.

However, adding back case has a larger impact, as a relatively large number of words have at least two versions (example g). A second manual inspection of the same splits revealed that more words were decomposed in subwords (e.g., `adjusted` becomes `Adjust•` ed, or `implicitly` becomes `Implicit• ly`). Raising the amount of merges to 20,000 is necessary, but it increases the corpus by 2% only, for a corpus with strings and comments.

We conclude that as a rule of thumb, a BPE with a 1–2% token increase performs very well.

BPE shrinks source code vocabulary *very* effectively. Moreover, most of the vocabulary is frequent, improving embeddings. Finally, many of the subwords are meaningful to humans and correspond to English words or sensible concepts with small exceptions like typos if an appropriate size encoding is used.

3.4 Neural Language Model for Code

We present our NLM for code based on subword units, which is based on a Recurrent Neural Network (RNN). RNN LMs scan an input sequence forward one token at a time, predicting a distribution over each token given all of the previous ones. RNNs with gated units can learn when to forget information from the hidden state and take newer, more important information into account (Hochreiter and Schmidhuber, 1997). Among various gated units, GRUs (Cho et al., 2014) have been shown to perform comparably to LSTMs (Hochreiter and Schmidhuber, 1997) in different applications (Chung et al., 2014).

We intentionally selected a small model as our base model: a single layer GRU NLM built upon subword units learned from BPE (Section 3.3.4). For each vocabulary entry we learn a continuous representation of 512 features, while the GRU state is of the same size. The vocabulary entries (BPE subwords) are represented by assigning a unique id to each one of them based on their frequency. These ids are used to query the embedding matrix for the continuous representation of one or more vocabulary entries that we are interested by using Tensorflow's *embedding_lookup* function. This is equivalent to using one-hot vectors but in this case the *embedding_lookup* function handles this for us in an efficient way and extracts the representations that we queried for. In all our experiments we used a learning rate of 0.1, dropout of 0.5 (Srivastava et al., 2014) and a maximum of 50 epochs of stochastic gradient descent with a minibatch size of 32 (for the small training sets) or 64 (for the full training sets). These hyper-parameters were tuned on the small train and validation sets. After each iteration we measure cross entropy on a validation set (Section 3.5). If the cross entropy is larger than the previous epoch then we halve the learning rate and this can happen for a maximum of 4 times, otherwise training stops. During training of the global model we unroll the GRU for 200 timesteps, following (Khandelwal et al., 2018). Our implementation is open source written in Tensorflow (Abadi et al., 2015).[6] We also experiment with larger capacity models (2048 hidden features and GRU state).

3.4.1 Selecting Subword Units with BPE

In our code LM, we address vocabulary issues by having the model predict *subwords* rather than full tokens at each time step. Subwords are inferred by BPE (Section 3.3.4) on a held out dataset of projects that are separate from the training, validation, and test sets. We experimented with three encoding sizes, i.e., the maximum number of merge operations: 2000, 5000, and 10000. To train the LM, we first segment the train, validation, and test sets using the learned encoding. We transform each token into a character sequence, adding `</t>` after every token. Then we apply in order the merge operations from BPE to merge the characters into subword units in the vocabulary.[7] As in (Sennrich et al., 2015) we do not merge pairs that cross token boundaries that is, where the merged token would contain `</t>` internally, so that every subword unit is a character subsequence of a token in the data. Finally, we train and test the NLM as usual on the data segmented in subword units.

3.4.2 Predicting Tokens from Subword Units

Autocompletion algorithms present a ranked list of k predicted tokens rather than a single best prediction. With a model based on subword units, it is not obvious how to generate the top k predictions, because a single token could be made from many subword units. We approximate these using a custom variation of the beam search algorithm. If the beam is large enough the algorithm can give a good approximation of the top-k complete tokens.

[6] https://github.com/mast-group/OpenVocabCodeNLM
[7] We use the BPE implementation from https://github.com/rsennrich/subword-nmt

The NLM defines a probability $p(s_1 \dots s_N)$ for any subword unit sequence. The goal of the beam search is: given a history $s_1 \dots s_N$ of subword units that already appear in a source file, predict the next most likely *complete token*. A *complete token* is a sequence of subword units $w_1 \dots w_M$ that comprise exactly one token: that is, w_M ends with `</t>` and none of the earlier subword units do. Beam search finds the k highest probability complete tokens, where we denote a single token as the sequence of units $w_1 \dots w_M$, that maximize the model's probability $p(w_1 \dots w_M | s_1 \dots s_N)$. Importantly, the length M of the new complete token is *not* fixed in advance, but the goal is to search over complete tokens of different length.

The algorithm is illustrated in Algorithm 1. Given a value of k and a beam size b, the algorithm starts by querying the model to obtain its predictions of possible subword units, ranked by their probability. In our pseudocode, we assume that the model's predict function returns a ranked list of a given size, and that V is the total size of the vocabulary. The algorithm uses two priority queues: one called candidates which ranks the sequences of subword units that still need to be explored during the search, and one called bestTokens which contains the k highest probability complete tokens that have been expanded so far. Each candidate is a structure with two fields, text which is the concatenation of all the subword units in the candidate, and prob which is the product of the probabilities of each subword unit in the candidate. Both of the priority queues are sorted by the probability of the candidate. The candidate class has an extend method which updates both of these fields in order to add one additional subword unit to the end of the candidate.

Algorithm 1 Predicts top k most likely tokens according to the model to follow the history of subword units $s_1 \ldots s_N$.

```
1: procedure PredictTopK(model, $s_1 \ldots s_N$, $k$, $b$, $V$)
2:   subwords, probs ← model.predict($V, s_1 \ldots s_N$)
3:   # Initialize priority queues of completed tokens
4:   bestTokens ← $k$ highest probability tokens from subwords
5:   candidates ← $b$ highest probability non-tokens from subwords
6:   total ← sum($c$.prob for $c$ in bestTokens)
7:   # Main search loop. Expand $b$ best incomplete tokens
8:   lowest ← min(t.prob for $t$ in bestTokens)
9:   while termination criterion not met do
10:    toExpand ← candidates.popNBest($b$)
11:    for all candidate ∈ toExpand do
12:      subwords, probs ← model.predict($b$, candidate.text)
13:      for all $w$ ∈ subwords do
14:        newCandidate ← candidate.extend($w$, probs[$w$])
15:        if isToken(newCandidate) then
16:          bestTokens.push(newCandidate)
17:          bestTokens.pop()              # Retain top $k$
18:          lowest ← min(t.prob for $t$ in bestTokens)
19:          total ← total + newCandidate.prob
20:          tokensDone ← tokensDone + 1
21:        else
22:          candidates.add(newCandidate)
23:    iters ← iters + 1
24:  return bestTokens
```

The main loop of the search is in lines 9-23. In each iteration, the algorithm pops the b best candidates from the candidates queue, expands them with one additional subword unit, and scores their expansions. If an expansion creates a token (the new subword unit ends with `</t>`) then it is pushed onto the token queue and the worst token is popped. This maintains the invariant that bestTokens has size k. If the new expansion is not a complete token, then it is pushed onto the candidates queue, where it can potentially be expanded in the next iteration. This search procedure is repeated until any of the following termination criteria has been satisfied at line 9:

(a) The number of complete tokens that have been explored during the search exceeds a threshold (in our implementation, we use $\mathsf{tokensDone} > 5000$).

(b) The cumulative probability of all the tokens that have been explored exceeds the threshold, i.e., $\mathsf{total} > 0.8$

(c) A sufficient number of search iterations have been completed, i.e., $\mathsf{iters} > 7$.

(d) The probability of the best candidate is less than the worst current complete top-k tokens, that is,

$$\min\{c.\mathsf{prob} \mid c \in \mathsf{bestTokens}\} \geq \max\{c.\mathsf{prob} \mid c \in \mathsf{candidates})\}.$$

Expanding a candidate cannot increase its probability, so at this point we are guaranteed that no better complete tokens will be found in the remainder of the search.

These criteria ensure that the **beam search always terminates**.

3.4.3 Caching

We also implement a simple caching mechanism for our NLM to exploit the locality of source code, particularly previously defined identifiers. At test time, each time an identifier is encountered, the 5-token history that preceded it is added to a cache alongside it. Differently to n-grams, we do not store probabilities, as the NLM will compute them. If the current 5-token history exists in the cache, the identifiers that followed it are retrieved (this is in practice very small, usually 1 or 2 identifiers). These identifiers are then scored by the NLM, and their probabilities are normalized to 1. The beam search described earlier is then run, and the two probability distributions are merged, according to a cache weight parameter: $cache\_pred \times cache\_weight + beam\_pred \times (1 - cache\_weight)$. The top 10 of the merged predictions are then returned.

We set the cache weight to 0.3. Note that, like beam search, this is a test-time only addition that does not affect training.

3.4.4 Dynamic adaptation to new projects

A *global LM*, trained in a cross-project setting, will perform better if it is adapted to a new project (Hindle et al., 2012; Tu et al., 2014). LMs with n-grams also employ caches for this. Simply training an NLM from scratch on a new project will not have enough data to be effective, while training a new model on both the original training set and the new project would be impractical and computationally expensive.

Instead, we use a simple method of dynamically adapting our global NLMs to a new project. Given a new project, we start with the global NLM and update the model parameters by taking a single gradient step on each encountered sequence in the project after testing on it. This series of updates is equivalent to a single training epoch on the new project. (In our evaluations in Section 3.5, we will split up the project files in such a way that we are never training on our test set.) We unroll the GRU for 20 time steps instead of 200 as in our global models, in order to update the parameters more frequently. We apply only one update for two reasons. First, it is faster, allowing the model to quickly adapt to new identifiers in the project. Second, taking too many gradient steps over the new project could cause the NLM to give too much weight to the new project, losing information about the large training set.

3.5 Evaluation

Intrinsic Evaluation: Language Modeling. A good language model assigns high probabilities to real sentences and low probabilities to wrong ones. For code, fragments that are more likely to occur in human-written code should be assigned higher probability. Precise scoring of code fragments is essential for tasks such as translating a program from one programming language to another (Karaivanov et al., 2014; Nguyen et al., 2013a), code completion (Franks et al., 2015; Raychev et al., 2014), and code synthesis from natural language and vice versa (Allamanis et al., 2015b; Bunel et al., 2018; Desai et al., 2016; Devlin et al., 2017a; Nguyen et al., 2016; Raghothaman et al., 2016)

As in previous work, our intrinsic metric is the standard cross entropy. Cross entropy defines a score over a sequence of tokens t_1, t_2, ..., $t_{|C|}$. For each token t_i, the probability $p(t_i|t_1, ..., t_{i-1})$ of each token is estimated using the model under evaluation. Then the average per token entropy is $H_p(C) = -\frac{1}{|C|}\sum_{i=1}^{|C|} \log p(t_i|t_1, ..., t_{i-1})$. Cross entropy is the average number of bits required in every prediction; lower values are better. It not only takes into account the correctness of the predictions, but also rewards high confidence.

Our NLMs define a distribution over subwords, not tokens. To compute cross entropy for subword NLMs, we segment each token t_i into subwords $t_i = w_{i1} \dots w_{iM}$. Then we compute the product $p(t_i|t_1, ..., t_{i-1}) = \prod_{m=1}^{M} p(w_{im}|t_1, ..., t_{i-1}, w_{i1} \dots w_{i,m-1})$, where the right hand side can be computed by the subword NLM. This probability allows us to compute the cross entropy $H_p(C)$.

Extrinsic evaluation: Code Completion. We report the performance of our LMs on code completion, which is the task of predicting each token in a test corpus given all of the previous tokens in the file. We measure performance with mean reciprocal rank (MRR), as is common in code completion evaluation (Bruch et al., 2009; Hellendoorn and Devanbu, 2017; Raychev et al., 2014; Tu et al., 2014). The evaluation operates over the token's generated by the tokenizer used in (Hellendoorn and Devanbu, 2017) and not over BPE subtokens. This means that when we perform the evaluation it operates over entire identifiers and **not** their subtokens. As such we do not give credit to the models for partially correct completions (which could happen in the BPE models) but these might still be useful in some rare cases to human developers. Also note that the tokenizer used may not always generate a single token for Strings but instead multiple ones. In that case we treat each of them as a separate tokens identically to (Hellendoorn and Devanbu, 2017). This however would perplex the evaluation if that were to be performed over Strings. As users in code completion applications are more interested in identifier performance or numeric literals as well as for consistency we did not treat this as an issue and retained the original evaluation. Each time the LM makes a prediction, we get a ranked list of $k = 10$ predictions. For each one, the reciprocal rank is the multiplicative inverse of the rank of the first correct answer. MRR is the average of reciprocal ranks for a sample of queries Q:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}. \tag{3.1}$$

A simplified description of MRR is that it averages top-k predictive performance across various k. Note that a correct suggestion at rank 1 yields an MRR of 1; at rank 2, 0.5; at rank 10, 0.1. Thus, a small difference in MRR could indicate a large change in the ranked list, especially for higher MRR values.

Code Completion Scenarios. We use three scenarios from previous work (Hellendoorn and Devanbu, 2017): Each *static*, *dynamic*, and *maintenance* settings simulates a different way of incorporating NLMs in an IDE. The task is always to predict test set tokens, but the training sets differ:

Static tests. The model is trained on a fixed training corpus, and later evaluated on a separate test dataset. This is a cross-project setting: train, validation, and tests sets all contain separate projects. This simulates a single global LM that is trained on a large corpus of projects and then deployed to clients without adaption.

Dynamic tests. In addition to the training set, the model can update its parameters *after* it has made predictions on files in the test set (it never trains on test data). The model is allowed to retrain on sequences from files in the test set only *after* it has been scored on its predictions upon them. Our NLMs are adapted using the procedure described in Section 3.4.4. After each project, we restore the model to the global LM learned from the train set only. This simulates a setting in which some files from the project of interest are available for dynamic adaptation.

Software maintenance tests. This scenario is even closer to real world usage, simulating everyday development where programmers make small changes to existing code. The LMs are tested on one file at a time in the test set. For each test file F, the train set plus all other files in the test project except F is used as training data. As this requires retraining the NLM once per file in the test set, this scenario was previously deemed infeasible for NLMs in (Hellendoorn and Devanbu, 2017).

Identifiers only. Recent work observed that LMs for completion perform worse on identifiers than other tokens (Hellendoorn et al., 2019). Therefore, we also report model performance, i.e., entropy and MRR, on identifier tokens only (excluding primitive types). To clarify differences between methods, we also report *recall at rank 1 (R@1)*, the percentage of all identifier usages which are correctly predicted at rank 1, and similarly recall at rank 10 (R@10), the percentage when the correct identifier appears anywhere in the model's top 10 predictions.

3.6 Research Questions

RQ1. How does the performance of subword unit NLMs compare to state-of-the-art LMs for code? We compare subword unit NLMs to standard n-gram LMs (Hindle et al., 2012), cache LMs (Tu et al., 2014), state-of-the-art n-gram LMs with nested caching Hellendoorn and Devanbu (2017), token-level NLMs (White et al., 2015), and heuristic splitting NLMs (Allamanis et al., 2015a). We do not compare with PHOG (Bielik et al., 2016) and pointer network RNNs (Li et al., 2018): both do not have a full implementation available. We do not evaluate character-level NLMs as they have not shown benefits for NLP.

RQ2. Can subword unit NLMs scale to large code corpora? Does the additional training data improve performance? Training on a larger corpus may improve a model's performance, but adding more data tends to have diminishing returns. After some point, a model's performance saturates and does not continue to improve with more data. This saturation point will be different for different models, so we evaluate if NLMs can make better use of large corpora than n-gram models. Moreover, training on larger data uses introduces scaling issues. The n-gram LMs need to estimate counts for every n-gram in the training corpus, while NLMs need to learn input and output embedding matrices whose dimension scale with vocabulary size. Thus, performance in terms of runtime cost, memory usage, and storage becomes important.

RQ3. How does the performance of subword unit NLMs vary across programming languages? In principle the learning methods for NLMs are language agnostic; however, the majority of studies evaluate only on Java. We check if code LMs are equally effective on other programming languages: C's terseness, or Python's lack of type information could negatively impact an LM's performance.

RQ4. Is the dynamic updating effective to adapt subword unit NLMs to new projects? New projects introduce many new identifiers that do not appear even in a large cross-project corpus. An n-gram LM can exploit the strong locality that characterises code through caching (Hindle et al., 2012; Tu et al., 2014) showing that n-gram models significantly benefit from dynamically adapting to the test corpus, as described in Section 3.5. Thus we ask whether NLMs can also benefit from dynamic adaptation via the procedure presented in Section 3.4.4.[8] is effective at dynamically adapting NLMs to new projects. We compare our dynamic adaption technique against two dynamic n-gram models: cache LMs (Tu et al., 2014) and nested cache LMs (Hellendoorn and Devanbu, 2017).

RQ5. Are NLMs useful beyond code completion? NLMs in NLP have shown to be useful in a variety of tasks, including translation or summarization; they have been recently shown to be state of the art in transfer learning. While testing all of these scenarios vastly exceeds the scope of this chapter, we test whether NLMs improve upon n-gram LMs in the task of detecting buggy code (Ray et al., 2016a).

3.7 Results

Table 3.3 presents the evaluation metrics of all scenarios; we refer to it continuously. We used the n-gram implementation[9] used in (Hellendoorn and Devanbu, 2017) with the

[8]A naive approach to the software maintenance scenario retrains the model from scratch for every test file, which was rightly deemed infeasible for NLMs by (Hellendoorn and Devanbu, 2017)

[9]https://github.com/SLP-team/SLP-Core, version 0.1

same parameters (n = 6); all NLMs are ours. We compute MRR on the first million tokens of the test set, as in (Hellendoorn and Devanbu, 2017).

3.7.1 RQ1. Performance of Models

Because the full data set is so large, we compare the different variants of n-gram models against each other on the small Java training set, and then we compare the best n-gram LM against our BPE NLM on the large Java data set. In Table 3.3, we see that the nested cache model has the best performance of the n-gram models, with a large improvement over the simpler models (for example, improving MRR from 58% to 77% on Java against the basic n-gram model). This is consistent with the results of (Hellendoorn and Devanbu, 2017). However, our BPE NLM outperforms it. (Note that cache models can not be evaluated in the static scenario since the cache would adapt to the test set). Moving to the large data set, we find that the BPE NLM still outperforms the nested cache model, even though the nested cache model was specifically designed for code. While previous work (Hellendoorn et al., 2019) found that closed NLMs underperformed on identifiers, we find that our BPE NLMs do not. In the dynamic scenario, 74% of identifiers are predicted within the top 10 predictions, with up to nearly 56% in first position.

Open vs closed vocabulary. To specifically evaluate the effect of relaxing the closed vocabulary assumption, we compare our open vocabulary NLM to two closed vocabulary NLMs: one that uses full tokens (Closed NLM), and another that splits tokens according to conventions (Heuristic NLM). Those models have otherwise the same architecture as the open vocabulary. In both cases, we find that the open-vocabulary NLM significantly outperforms both closed vocabulary NLMs, and can be trained even in the maintenance setting, unlike the closed versions. Of note, our closed vocabulary NLM performs better than the one in (Hellendoorn et al., 2019), as it utilizes a fully connected hidden layer and dropout. Finally, in Table 3.4 we report the performance of the open vocabulary NLMs with different vocabulary sizes, obtained after 2000, 5000, and 10000 BPE merge operations. We see that performance on the small training set is similar across vocabulary sizes: a large vocabulary is not required for good performance.

Caches, and larger capacity. Both our cache and increasing model capacity (from 512 to 2048 features) are beneficial, particularly for the identifiers. The cache improves MRR by 3 to 4%, with more improvements for low ranks, which is especially important for completion. On the small corpus, the large model improves MRR by nearly 3%, a smaller improvement than adding the cache. Both improvements are complementary, increasing identifier MRR by close to 6%.

Table 3.3 Performance of the various models (**bold: best**, underlined: second best).

MODEL	Java							Java Identifiers			C				Python			
	Static		Dynamic		Maintenance		Bugs	Dynamic			Static		Dynamic		Static		Dynamic	
	Ent	MRR	Ent	MRR	Ent	MRR	% Ent ↓	R@1	R@10	MRR	Ent	MRR	Ent	MRR	Ent	MRR	Ent	MRR
Small Train																		
***n*-gram**	6.25	53.16	5.54	56.21	5.30	58.32	1.81	17.24	34.66	22.26	6.51	55.20	4.14	57.34	5.30	43.63	4.81	47.39
Nested	-	-	3.65	66.66	2.94	71.43	-	37.46	56.85	43.87	-	-	3.61	62.25	-	-	4.05	54.02
Cache	-	-	3.43	69.09	3.32	70.23	-	40.13	59.52	46.57	-	-	2.19	75.09	-	-	3.22	62.27
Nested Cache	-	-	2.57	74.55	2.23	77.04	-	49.93	70.09	56.81	-	-	2.01	76.77	-	-	2.89	65.97
Closed NLM	**4.30**	62.28	3.07	71.01	-	-	1.81	30.96	49.93	37.20	4.51	60.45	3.20	72.66	3.96	**81.73**	3.34	84.02
Heuristic NLM	4.46	53.95	3.34	64.05	-	-	1.04	39.54	58.37	45.28	4.82	52.30	3.67	61.43	4.29	65.42	3.56	71.35
BPE NLM (512)	4.77	63.75	2.54	77.02	**1.60**	**78.69**	3.26	45.49	67.37	52.66	4.32	62.78	1.71	76.92	3.91	81.66	2.72	86.28
BPE NLM (512) + cache	-	-	-	77.42	-	-	-	50.49	68.16	56.30	-	-	-	-	-	-	-	-
BPE NLM (2048)	4.77	**64.27**	**2.08**	77.30	-	-	**3.60**	48.22	69.79	55.37	**4.22**	**64.50**	**1.59**	**78.27**	**3.66**	81.71	**2.69**	**86.67**
BPE NLM (2048) + cache	-	-	-	**78.29**	-	-	-	**52.44**	**70.12**	**58.30**	-	-	-	-	-	-	-	-
Large Train																		
Nested Cache	-	-	2.49	75.02	2.17	77.38	-	52.20	72.37	59.09	-	-	1.67	**84.33**	-	-	**1.45**	71.22
BPE NLM (512)	3.15	70.84	1.72	79.94	**1.04**	**81.16**	4.92	51.41	74.13	59.03	3.11	70.94	1.56	77.59	3.04	84.31	2.14	87.06
BPE NLM (512) + cache	-	-	-	80.29	-	-	-	55.68	**74.30**	61.94	-	-	-	-	-	-	-	-
BPE NLM (2048)	**2.40**	**75.81**	**1.23**	82.41	-	-	**5.98**	57.54	72.18	62.91	**2.38**	**80.17**	**1.36**	83.24	**2.09**	**86.17**	1.90	**87.59**
BPE NLM (2048) + cache	-	-	-	**83.27**	-	-	-	**60.74**	73.76	**65.49**	-	-	-	-	-	-	-	-

Table 3.4 Effect of vocabulary size on Java performance of our open-vocabulary models (Python and C are similar).

Vocab Size	**Static**		**Dynamic**		**Maint.**		**D4J Bugs**
	Ent	**MRR**	**Ent**	**MRR**	**Ent**	**MRR**	**% Ent ↓**
Small Train							
2 000	4.90	62.87	2.33	75.66	1.46	77.48	3.07
5 000	4.78	63.80	2.27	77.14	1.51	78.49	3.38
10 000	4.77	63.75	2.54	77.02	1.60	78.69	3.26
Large Train							
2 000	3.59	68.87	1.84	77.69	1.03	78.85	4.09
5 000	3.35	69.87	1.72	79.18	1.06	80.31	4.71
10 000	3.15	70.84	1.72	79.94	1.04	81.16	4.92

Open vocabulary NLMs are effective models of source code, even on a small corpus, yielding state of the art performance.

3.7.2 RQ2. Large Corpora

We contrast performance between small and large training sets.

Leveraging data. When trained on larger corpora, the performance of n-gram models (including nested cache variants) gets saturated and they are unable to effectively leverage the extra information (Hellendoorn and Devanbu, 2017). In contrast, our model can better leverage the increase in training data when trained on the full corpus. In the static scenario, our NLMs decrease entropy by about 1.5 bits, while MRR increases by about 6%. More data helps our NLMs learn to synthesize identifiers from subwords better and with higher confidence.

The improvements are smaller but still exist when the NLMs use dynamic adaptation: for all encoding sizes the entropy improves by 0.5 bits and MRR by 2 to 3%. In contrast, the nested cache n-gram model entropy improves by less than 0.1 bits and MRR by less than 0.4%. From that we conclude that subword unit NLMs can utilize a large code corpus better than n-gram models. As shown in Table 3.4, larger training corpora tend to favor NLMs with larger vocabularies, particularly in terms of MRR; larger models leverage the additional data even better. For all models, the improvements are more visible for identifiers: the large train alone contributes close to 7% of MRR for identifiers, versus 3% overall for the NLM. Finally, larger NLMs (2048 features) are even better at leveraging the additional training data, due to their increased capacity. Similarly, the cache still improves performance further, even with the large training set; both improvements complement each other.

Resource usage. While the nested cache n-gram model is competitive with Java identifiers, this comes at a significant cost: resource usage. Disk usage for n-gram models range from 150 to 500 Mb in the small training set to **6 to 8.5GB** in the large training set. RAM usage is even more problematic, as it ranges from around 5GB in the small training set, up to **50 to 60GB** in the large training set. *This makes the large n-gram models unusable in practice as they exceed the memory requirements of most machines.*

In contrast, the NLMs do not vary significantly with training set size; their size is fixed. They range from 15MB (BPE 2K) to 45MB (BPE 10K) on disk (up to 240MB for the large capacity models). RAM usage for NLMs vary between 2 to 4GB when training (and can be reduced at the expense of speed by reducing batch size), and is considerably lower at inference time (for actual code completion), ranging from 250 to 400MB. *Thus, if we compare practically applicable models, the small NLM outperforms the small nested cache n-gram model by up to 5.13% in identifier MRR, and up to 5.75% recall at 1; the large NLM does so by 8.68% (MRR), and 10.81% (recall at 1).*

The open vocabulary makes training NLMs on large corpora scalable as vocabulary ceases to grow with corpus size; training time scales linearly with added data. Our largest NLM (BPE 10k, 2048 features), can process around 350 to 550 hundred thousand tokens per minute (roughly 100 to 300 projects per hour depending on project size) on a consumer-grade GPU. This makes our dynamic adaptation procedure, which trains one project for one epoch, clearly feasible. Training the initial model is still a large upfront cost, but it takes from a day (small NLM) up to two weeks (large NLM) on our largest dataset, and needs to be performed once. At inference time, predicting 10 tokens with beam search takes a fraction of a second, fast enough for actual use in an IDE, even without additional optimization. This is not true for the closed models.

Open-vocabulary NLMs can scale; furthermore, they leverage the increased training data effectively. Large n-gram models do *not* scale in terms of resources.

3.7.3 RQ3. Multiple Languages

We contrast Java performance with Python and C. We see interesting differences between Java, Python, and C. First, n-gram models perform considerably worse in Python, while NLMs do very well. We hypothesize that this is due to the smaller size of Python projects in our corpus, which reduces opportunity for caching (the average Python project is 2 to 3 times smaller than the average Java project). C projects, on the other hand, are competitive with Java projects, particularly with caching; they are on average 2 times larger. Interestingly, the nested and nested cache n-gram models perform comparatively worse in C than in Java: C projects tend to have a flatter structure, rendering the nesting assumption less effective in this case. Finally, the (not applicable in practice) large n-gram model outperforms our NLMs for C. We observed anectodal evidence that there is considerable duplication in the C corpus, which may affect this result (Allamanis et al., 2018a). For NLMs, the performance is more even across the board, with overall slightly worse performance for C, and somewhat better performance for Python. We see that performance on C and Python is at least as good as Java, providing evidence that our methodology for training subword unit NLMs is indeed language agnostic. We also observe that in general there are strong trends of how the models perform across the three languages. Such as simple n-gram models being the weakest ones across all languages as one would expect. Dynamic adaptation is always useful and more effective for the BPE models than other NLMs. The static closed NLM performs much worse than the static BPE NLMs in Java and C but performs much better in the case of Python and even outperfoming the small static Python BPE NLMs by a very slight margin. The reason for this is probably that Python projects are smaller utilize a core vocabulary across them. This might slightly hinder learning how to the more rare identifiers from subwords. As such dynamic adaptation is essential to improve the BPE NLMs. The heuristic NLM performs the worst in C. The reason for this is probably that segmenting by camel case is not typical in C and may result to uncommon vocabulary entries thus difficult to estimate parameters as well as strange atypical segmentations.

As discussed previously we caution the reader to *not* interpret these results as a comparison of the programming languages as to which is more predictable, more terse, etc. There are many confounds, including size of training corpora, standard libraries and problem domains across languages. It is also hard to strongly reason whether certain performance differences are completely due to language properties without taking into account other dataset features such as the degree of duplication in each corpus but one may definitely attribute part of it to these properties.

Our NLM performance results hold for Java, C, and Python.

3.7.4 RQ4. Dynamic Adaptation

We evaluate the effectiveness of our proposed method for adaption of NLMs in the dynamic and maintenance scenarios. This is crucial for practical usage of NLMs, because the dynamic and maintenance scenarios simulate the setting where the developer is modifying a large, existing project. Using within-project data provides a large performance boost: Even though within each scenario, our NLMs outperform n-grams, most n-gram models in the dynamic scenario outperform NLMs in the static scenario. The improvement due to dynamic adaptation is greater than the improvement due to an NLM. Of note, the situation in the large training set is different: the static large NLM trained on the large training set *outperforms the cache n-gram LMs in the dynamic scenario, and is competitive with it in the maintenance scenario*, in other words, our large data set is so large that it *almost* makes up for not having within-project data, but within-project information is clearly still crucial.

Once we apply the dynamic adaptation method to the NLMs, the picture changes. With dynamic adaptation, our model achieves better cross-entropy than the current state-of-the-art (Hellendoorn and Devanbu, 2017), making it an effective technique to fine-tune an NLM on a specific project. Using this method, it is even possible to evaluate NLMs on the maintenance scenario, which was previously deemed infeasible by Hellendoorn and Devanbu (2017) since multiple models had to be created, each trained on the entirety of the test set minus one file. This is possible for us because the combination of a small vocabulary size and our finetuning method running for only one epoch make this scenario much faster.

Open vs closed NLMs. Interestingly, the difference in performance between the open and closed vocabulary NLMs is larger in the dynamic setting. We hypothesize that dynamic adaptation helps the open-vocabulary model to learn project-specific patterns about OOV words; this is not possible for a closed vocabulary NLM.

Dynamic adaptation for NLMs yields the state of the art; static NLMs are competitive with some dynamic n-gram models, which bodes well for transfer learning.

3.7.5 RQ5. Bug Detection

Previous work has observed that n-gram language models can detect defects as they are less “natural” than correct code (Ray et al., 2016a). In short, defective lines of code have a higher cross-entropy than their correct counterparts. To assess whether our code NLM is applicable beyond code completion, we compare the ability of different language models to differentiate between the two on the well-known `Defects4j` dataset (Just et al., 2014a). Defects4J contains 357 real-world defects from 5 systems. Both a buggy and a corrected version of the system are provided and the changed lines can be extracted. We compute the difference in entropy between the buggy and the fixed version for each of the diff patches provided. The extracted code snippets usually contains a few unchanged surrounding lines that provide useful context for the LMs. We expect a better LM to have a larger entropy difference between the defective and the corrected version.

We compute these metrics only for LMs in a static setting for three reasons: 1) we simulated the setting in which a bug detector is trained on one set of projects and used on unseen ones, 2) it is not clear how caches would be used in this scenario (should the LM "know" which file a bug is in?), and 3) doing so could involve training two LMs for each defect, which is very expensive.

The results are shown in the Java "bugs" column in Tables 3.3 and 3.4. As we hypothesized, open vocabulary NLMs feature a larger entropy drop for fixed files than n-gram LMs or closed NLMs. The drop in entropy is 70% to 100% for the small training set, depending on vocabulary size and model capacity (larger is better). Furthermore, these models benefit from a large training set, with a larger drop of 127 to 173%. We would expect larger capacity models to improve even further. We hypothesize that beyond data sparsity for identifiers, the NLM's long range dependencies are especially useful in this task.

Finally, we note that the corpus used does not contain specific information on the defect type for each instance. However, we note that such information would prove useful as it would for example allow an analysis over the various defect types. Thus, allowing us to investigate whether the models are equally effective across all types of defects or only for some of them. Although this thesis does not answer this question, we strongly encourage any future research that pursues it.

Open-vocabulary NLM are better bug detectors than n-gram LMs, particularly when trained on large corpora.

3.8 Artifacts

Several artifacts were used to conduct this study: data, source code, and models. To improve replication of this work, the specific version of each artifact used in this study can be referenced via a DOI. Table 3.5 lists the DOI of each artifact. The paper Karampatsis et al. (2020) should be referenced when any of these artifacts is used. Optionally, please also cite this chapter of this thesis.

Datasets. The datasets described in 3.2 were published in previous work: The Java corpus was produced by (Allamanis and Sutton, 2013), and also used in (Hellendoorn and Devanbu, 2017). The C corpus was mined in (Dudoladov, 2013) and the Python corpus was mined in (Fiott, 2015). We use the raw datasets for the vocabulary study,

Table 3.5 DOIs of artifacts used or produced by this work

Artifact	**DOI**
Java corpus	https://doi.org/10.7488/ds/1690
C corpus	https://doi.org/10.5281/zenodo.3628775
Python corpus	https://doi.org/10.5281/zenodo.3628784
Java, pre-processed	https://doi.org/10.5281/zenodo.3628665
C, pre-processed	https://doi.org/10.5281/zenodo.3628638
Python, pre-processed	https://doi.org/10.5281/zenodo.3628636
codeprep	https://doi.org/10.5281/zenodo.3627130
OpenVocabCodeNLM	https://doi.org/10.5281/zenodo.3629271
Trained models	https://doi.org/10.5281/zenodo.3628628

but preprocess them for NLM training. Further, we defined training and test sets for the C and Python corpora, and defined the large training set for the Java corpus.

Source code. We implemented the *codeprep* library that supports a variety of pre-processing options for source code. We used *codeprep* to gather the vocabulary statistics presented in Section 3.3. Researchers that wish to use the library to pre-process source code for their own study can find the library at: https://github.com/giganticode/codeprep.

The open vocabulary language model described in 3.4, as well as the scripts implementing the training procedure and the evaluation scenarios are available in the *OpenVocabCodeNLM* library. Researchers wishing to extend our model can find it on GitHub at: https://github.com/mast-group/OpenVocabCodeNLM.

Models. The models that were trained and evaluated in Section 3.7 are also made available for further use. Each model was trained on GPUs for periods ranging from a few hours, up to two weeks. These models can be used as-is for inference in a code completion scenario. Alternatively, they may be fine-tuned for other tasks, such as classification (Howard and Ruder, 2018; Robbes and Janes, 2019).

3.9 Conclusions

Source code has a critical difference with natural language: developers can arbitrarily create new words, greatly increasing vocabulary. This is a great obstacle for *closed-vocabulary* NLMs, which do not scale to large source code corpora. We first extensively studied vocabulary modelling choices, and showed that the only viable option is an *open-vocabulary* NLM; all other vocabulary choices result in large vocabularies, high OOV rates, and rare words.

We then presented a new open-vocabulary NLM for source code. By defining the model on subword units, which are character subsequences of tokens, the model is able to handle identifiers unseen in training while shrinking vocabulary by *three orders of magnitude*. As a consequence, our NLM can scale to very large corpora: we trained it on data sets over a *hundred times larger* than had been used for previous code NLMs. Our NLM also uses *beam search*, *dynamic adaptation*, and *caching* to efficiently generate tokens and adapt to new projects. Finally, we showed that our NLM outperforms recent state-of-the-art models based on adding nested caches to n-gram language models for code completion and bug detection tasks, in a variety of scenarios, and in three programming languages.

Of course, this study has limitations: While we tried to be exhaustive and evaluated a large number of scenarios, we could not evaluate all the possible combinations (hundreds) due to the resources needed, such as some large models or some large training scenarios. For this reason, we also refrained to evaluate other NLM architectures such as LSTMs (Hochreiter and Schmidhuber, 1997), QRNNs (Bradbury et al., 2016), Transformers (Vaswani et al., 2017), or additional neural cache variants (Merity et al., 2016; Vinyals et al., 2015). For the same reason, as in (Hellendoorn and Devanbu, 2017) we also limited MRR to 1 million tokens, which may cause discrepancies with entropy metrics as they are not evaluated on the same test set. We also limited ourselves to three languages, and did not fully evaluate the impact of code duplication (Allamanis et al., 2018a).

We also hope that the simplicity and scalability will enable large capacity models for code, and the transfer learning opportunities they bring (Devlin et al., 2018; Radford et al., 2019); this has been explored in software engineering, albeit not for source code (Robbes and Janes, 2019). Improved language models for code have the potential to enable new tools for aiding code readability (Allamanis et al., 2014), program repair (Bhatia and Singh, 2016; Campbell et al., 2014; Gupta et al., 2017; Ray et al., 2016a), program synthesis (Gulwani et al., 2017b) and translation between programming languages (Karaivanov et al., 2014; Nguyen et al., 2013a). Finally, the technique of using subword units is not limited to language modelling, but can easily be incorporated into any neural model of code, such as models to suggest readable names (Allamanis et al., 2015a), summarizing source code (Allamanis et al., 2016; Iyer et al., 2016), predicting bugs (Pradel and Sen, 2018), detecting code clones (White et al., 2016), comment generation (Hu et al., 2018), and variable de-obfuscation (Bavishi et al., 2018).

Chapter 4

Mining a New Class of Simple Bugs

"No one in the brief history of computing has ever written
a piece of perfect software. It's unlikely that you'll be the first."
–Andy Hunt [The Pragmatic Programmer: From Journeyman to Master
(Hunt and Thomas, 2000)]

As outlined in Chapter 1, fixing bugs is an important but also really difficult problem in the domain of software engineering (SE) and it is essential to find and fix bugs the earliest possible during the software development cycle. Fixing a bug found during the production stage needs about three times more effort than if it was found during the coding stage (Tassey, 2002). Additionally, a later study which considered that the software development cycle is a sequential process constituted by the design, implementation, testing, and maintenance phases estimated the cost of fixing the bug during each of them. The later a bug is fixed the more its cost multiplies. Fixing it during the implementation stage costs about 6.5 times more than the design stage. The cost grows to about 16 times during the testing phase. Finally during maintenance the cost could sky-rocket up to about 100 times (Dawson et al., 2010).

Fixing bugs in programs, that is, program repair, is one of the core tasks in software maintenance, but requires effort to analyze failed executions, locate the cause of the fault, synthesize a bug fix and validate that the fault has been corrected without introducing new ones (Müllerburg, 1983). One could wonder why a developer team does not shift all of its effort to debugging during development every set amount interval of time as a solution. However, it is impossible to shift all of the developer's effort in bug fixing during early stages of development as that would result in pausing the development completely. Moreover, even if we did that, there is no guarantee to know that we have for sure reached a point where there are no bugs in the current software version. In addition, constantly shifting the focus of developers may have catastrophic results as coding is a difficult activity requiring strong focus. As such debugging is actually happening concurrently to development and even after development has finished during the software maintenance phase.

It stands out that having tools which attempt to alleviate most of the manual effort of locating bugs and repairing faults would be of great value to the industry. Passionate about achieving this goal researchers studied and developed appropriate methods and tools, thus creating the field of automatic program repair (APR) (Le Goues et al., 2012; Long and Rinard, 2016; Monperrus, 2018b; Pradel and Sen, 2018). APR has been a valuable resource but it is yet far from being perfect. Ideally, we would like a perfect APR system that reports defects early in the development cycle even when a line or code statement is first written so that the defect cost is minimized. However, the current advancements in the field of automatic program repair have not reached this stage yet, especially for generic fault fixing. Moreover, most APR techniques are based on test suites but the tests are also human written code and may still contain errors and thus potentially hinder the performance of the repair system, mask buggy code parts as correct, or even report code that functions as intended to be faulty. Furthermore, a major concern in the industry is that linters and program repair methods approaches are required to have high precision without risking achieving high enough recall. As an industrial example Google's Tricorder (Sadowski et al., 2015) enforces a false positive rate $< 10\%$. This naturally introduces the question of whether we could instead first achieve this goal by focusing on repairing types of simple bugs, such as one-line bugs, or bugs that fall into a small set of templates, such as mutation operators (Le Goues et al., 2012) or other types of predefined templates (Long and Rinard, 2015, 2016; Pradel and Sen, 2018). Focusing on these types of bugs might be a promising approach on achieving a "sweet spot" of maintaining high precision with adequate recall that is coveted in the industry. However, these have been evaluated on either a relatively small numbers of projects, e.g., 69 defects in 8 applications or on synthetic data. Because of this lack of data, it has not previously been possible to estimate the *recall* of a set of repair templates, that is, the percentage of real-world bugs that can be repaired by one of the templates.

Every programmer has multiple times run into that bug for which they spent hours or days to find only to realize that it was so simple to fix. Yet it took them so much time to identify. Such bugs usually may cause a strong emotional reaction upon realization resulting in the developer calling them stupid. The painful reaction to such a bug could be compared to stubbing one's toe. Although the fix for the bug is small and simple in relation to how much code needs to be written to synthesize the fix, it neither reflects the real effort needed to locate the bug nor its importance and effects. As discussed in Chapter 1 even one liner bugs can have catastrophic consequences. In most cases a reason why these bugs are usually so difficult to manually or even automatically detect is that they are semantic bugs that compile both before and after repair. Aiming to fill the gaps discussed above we will refer onwards to this class of bugs as “simple stupid bugs" (SStuBs).[1] To this end, this chapter[2] introduces the bug class of SStuBs and provides a dataset containing 25,539 single-statement bug-fix changes mined from 100 popular open-source Java Maven projects as well as a larger one containing 153,652 single-statement bug-fix changes mined from 1,000 popular open-source Java projects, annotated by whether they match any of a set of 16 bug templates, inspired by state-of-the-art program repair techniques. The corresponding dataset is referred to as the ManySStuBs4J dataset. The chosen templates aim at extracting bugs that compile both before and after repair as such can be quite tedious to manually spot, yet their fixes are so simple that many developers would call them “stupid” upon realization. Automatic repair of SStuBs is potentially an intermediate step toward more general program repair tools, while already being useful to developers. Additionally, SStuBs might be a good start for the evaluation of machine learning based fault localization and repair methods. An extra distinctive feature of our dataset is that the smaller version is restricted to projects that can be built automatically using Maven. Those that contain a test suite can be built and used to evaluate test based techniques.

This chapter makes the following contributions: 1) It introduces the problem of repairing SStuBs and defines what a SStuB is along with frequent SStuB patterns for Java; 2) it presents a methodology for automatically mining a SStuB dataset and introduces the publicly available ManySStuBs4J dataset; 3) it performs an initial analysis over the dataset answering essential research questions; 4) and finally evaluates the degree to which SStuBs fit the plastic surgery hypothesis.

Simultaneously to the current work, a larger dataset of one-line bugs has been mined (Chen et al., 2019b), but even this dataset does not attempt to classify bugs into templates. Section 4.1 discusses work in the existing literature that relates to the presented dataset.

Section 4.2 describes step by step the methodology for creating the ManySStuBs4J dataset. The patterns are presented along with examples in Section 4.2.6.

[1]The acronym is intended to reflect the fact that, for the authors at least, finding such a bug can feel much like stubbing one’s toe and no offence to any programmer or their feelings is meant.

[2]Most of the material in this chapter has appeared before in (Karampatsis and Sutton, 2020a), which has been accepted for publication and will appear in the proceedings of MSR 2020 and the manuscript was written by myself with Charles Sutton providing feedback and editing.

Section 4.3 prevents an initial analysis of the ManySStuBs4J dataset. We find that 33.04% in the smaller version dataset and 33.47% in the larger version of all of the single-statement bugs that we mine match at least one of the SStuB templates resulting in 10,231 and 63,923 SStuB instances respectively. This indicates that a remarkable number of singe-statement bugs can be repaired with a relatively small set of templates. In further analysis we also estimated the frequency in lines of code with which these pattern based and general single-statement bugs appear. This estimation is based on the size of the project's latest version and reveals that in the smaller dataset version SStuBs appear with a frequency of about 1 per 1,600 lines of code and 1 per 2,500 lines of code for the large version.

Section 4.4 presents and answers research questions related to the dataset. While Section 4.5 studies the degree in which SStuBs satisfy the plastic surgery hypothesis (Barr et al., 2014). Section 4.6 discusses limitations to our study and threats to validity. Finally, Section 4.7 concludes the chapter.

4.1 Related Work

Several previous data sets of real-world bugs have been curated. Defects4J (Just et al., 2014b) is a popular dataset consisting 395 Java bugs. Each bug is fixed in a single commit but the fix may modify multiple source code lines. The ManyBugs dataset (Goues et al., 2015) contains 185 C bugs, a subset of which were used by the GenProg (Le Goues et al., 2012), Prophet (Long and Rinard, 2016) and SPR (Long and Rinard, 2015) papers. Bugs.jar (Saha et al., 2018) is comprised of 1,158 Java bugs and their patches. These datasets have the disadvantage of being relatively small. More recently, a few larger-scale data sets of small bugs have been created. The combined datasets are the CodRep dataset (Chen and Monperrus, 2018a) and the Bugs2Fix dataset (Tufano et al., 2018b) resulting in 40,289 one-line bugs. These datasets are combined into a single dataset of one line bugs in (Chen et al., 2019b). Our datasets are of similar size consisting of 25,539 and 153,652 single-statement bugs. In contrast, our dataset focus on estimating the frequency of SStuB templates, motivated by recent program repair tools and also operates on the statement level, which prevents falsely excluding instances due to formatting or stylistic reasons. Also, the projects from which the small version of our dataset was generated can easily be built using Maven and we provide a list of projects containing tests and which tests fail for each instance (in GitHub repo). Thus test based methods can be evaluated upon them. However, unlike Defects4J that aims in comparing test-based patch generation approaches, it aims in techniques that can accurately highlight SStuBs early in development allowing immediate patching since in many cases the fix might be trivial. Lastly, unlike previous datasets, we take additional steps to filter out refactorings, although we acknowledge that such instances might be rare. In our case however, we were able to filter out almost 5,000 and 35,000 refactored statements for the two dataset versions.

4.2 Methodology

We next describe the methodology we employed to build the dataset. Our data generation tools along with documentation and detailed instructions for how to use them are available in a public GitHub repository[3] and the dataset is publicly available in Zenodo.[4]

4.2.1 Selecting Appropriate Java Projects

In order to mine a high quality dataset we opted to selecting high popularity projects. For the small version of the dataset we selected the 100 most popular open source Java Maven (Miller et al., 2010) projects from GitHub up to 1/4/2017. To allow evaluation of repair tools that might require building the projects, we selected only Maven ones because it is easy to automatically download the required dependencies for every project and build it. In contrast, manual downloading of dependencies would require an immense amount of human effort. To create a ranking for the projects we downloaded the MySQL dump of GHTorrent (Gousios, 2013) up to 1/4/2017. A project's popularity is determined by computing the sum of z-scores of its forks and stars (Allamanis et al., 2015a, 2016). Lastly, we pulled the projects' head commit by 28/1/2019 and considered commits until that date. The same approach was used to rank projects for the larger version. However, the ranking was calculated using a later dump of GHTorrent from 1/1/2019. A download script along with the list of projects for both variants of the dataset are also available to ensure replicability.

4.2.2 Classifying Commits as Bug-Fixing or not

For every project our tool searches historically through all of its commits to locate bug-fixing ones. To decide if a commit fixes a bug, we checked if its commit message contains at least one of the keywords: 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect', 'flaw', and 'type'. This heuristic was previously used by Ray et al. (Ray

[3]https://github.com/mast-group/mineSStuBs
[4]DOI: https://doi.org/10.5281/zenodo.3653444

et al., 2016b) and was shown to achieve 96% accuracy on a set of 300 manually verified commits and 97.6% on a set of 384 manually verified commits (Tufano et al., 2018a). We sampled 100 random commits containing SStuBs from the small version of the dataset and found it to achieve 94% accuracy. The above process produced a total of 115,929 and 883,982 bug-fixing commits for the small and large dataset variants.

4.2.3 Selecting Single Statement Changes

We have opted to restrict the dataset to small bug fixes that do not require much code modification to fix. Additionally, we are interested in bugs that are not just syntactic errors but cases where the code compiles both before and after the bug was located and repaired. As we are interested in simple bugs that involve only a single statement, we filter out any commits that either add or delete a Java file. We also filter out commits which make a multiple-statement change at any single position in the Java file. We do *not* filter out commits that make single-line modifications at more than one position in the same file. Similarly to the diff algorithm, we consider a modification as deleting the old lines/statements and then adding the new ones. To estimate whether a modification spans across multiple statements we calculate the diff for each modified Java file, and for each modified chunk, we count how many statements were modified. In the case of blocks each statement in the block's body is counted as a different statement. For `if` and `while` statements, we count the condition as a separate statement for this purpose. This method allows to us include fixes to single simple statements that span across multiple lines (e.g., due to stylistic reasons) as a simple fix, unlike a line-based approach. Any commits that modify multiple statements in any single position returned by the diff are dropped while we still maintain commits for which a file's diff contains multiple positions with single statement modifications. In the first case it is not trivial to align the deleted and added statements while it is in the latter. For example, one or more of the deleted statements may have been replaced by multiple of the added ones while simultaneously one or more of the deleted statements may have simply been deleted. We note that our tool ignores any changes to comments, blank lines as well as any formatting changes. As one understands the employed methodology is based on the assumption that the extracted statements either belong to individual single-statement edits/bug fixes or could be considered as such under certain constraints. Although this might not always be the case it does allow us to easier extract, analyze, and build bug detection and repair tools tailored for this kind of bugs. Additionally, it is still of value if we can detect each or any of the single statement bugs separately and fix them as this can save a lot of time and money to developers. Moreover, it might be a lot easier to fix each of the separate single statement bugs than concurrently fixing all of them together. It must also be noted that our methodology also allows cases where the same expression containing a bug appeared multiple times in the file. We present anecdotal evidence that this is often the case by manually inspecting the SStuB dataset entries. This filtering produces almost 13,000 and 86,769 commits for the two dataset versions. Lastly, the employed methodology works in a similar way to the popular SZZ algorithm (Sliwerski et al., 2005) and its extensions (Kim et al., 2006; Williams and Spacco, 2008) that have extensively been used to spot fix inducing changes.

4.2.4 Creating Abstract Syntax Trees

Each file in the commit that contains one or more bugs is parsed, yielding an abstract syntax tree (AST) of the file before the repair. Then, for each repaired line in the file we extract the AST after applying the repair only on that line and leaving the rest of the lines as is. Each extracted pair of ASTs (original and single fix) only differ on the node(s) for the modified line. By performing a simultaneous depth-first traversal on the two ASTs we locate the first node on which the two ASTs differ.

4.2.5 Filtering out Clear Refactorings

Although we filter for bug-fixing changes in Step B, there might still exist changes in the data that do not fix a bug or that do not even produce any behavioural changes. This could happen because the commit-message filter had a false positive, or because the change is tangled (Herzig and Zeller, 2013), and contains a bug-fixing modification along with unrelated ones to other files. To reduce the number of non-fixing changes in the dataset, we observe that there is a class of refactorings that can produce small changes, namely renamings. These are extracted via the diffs of the modified files. Our method spots variable, field names, function, or class renaming as well as any uses of them across other modified files in the commit and excludes them. This achieved by utilizing a two step process. In the first step we scan for all the declarations in the original and fixed versions of the project to spot any of them that have been refactored. Those are then saved in data structures (a different one for each type of refactoring) that can be queried to check whether an element has been refactored or not. Last, in the second step we go through all the single statement changes and query the appropriate data structure to check if any element has been refactored.

4.2.6 SStuB Patterns

We next describe the 16 SStuB patterns. We opted to choose patterns that appear often. Many of these have been used in pattern-based repair and mutation tools (Le Goues et al., 2012; Long and Rinard, 2015, 2016; Pradel and Sen, 2018). Here we provide a brief description of each pattern as well an example instance of it taken from dataset.

- *Change Identifier Used* Checks whether an identifier appearing in some expression in the statement was replaced with another one. It is easy for developers to by accident utilize a different identifier than the intended one that has the same type. Copy pasting code is a potential source of such errors. Similarly named identifiers may further contribute to the occurrence of such errors.

- *Change Numeric Literal* Checks whether a numeric literal was replaced with another one. It is easy for developers to mix two numeric values in their program.

- *Change Boolean Literal* Checks whether a Boolean literal was replaced. True is replaced with False and vice-versa. In many cases developers use the opposite Boolean value than the intended one.

- *Change Modifier* Checks whether a variable, function, or class was declared with the wrong modifiers. For example a developer can forget to declare one of the modifiers.

- *Wrong Function Name* Checks if a function with the same parameter list but the wrong name was called. This is a usual pitfall.

- *Same Function More Args* Checks whether an overloaded version of the function with more arguments was called. Functions with multiple overload can often confuse developers.

- *Same Function Less Args* Checks whether an overloaded version of the function with less arguments was called. For instance, a developer can forget to specify one of the arguments and not realize it if the code still compiles due to function overloading.

- *Same Function Change Caller* Checks whether in a function call expression the caller object for it was replaced with another one. When there are multiple variables with the same type a developer can accidentally perform an operation. Copy pasting code or mixing similar variables are common cases of such errors.

- *Same Function Swap Args* Checks whether a function was called with two of its arguments swapped. When multiple function arguments are of the same type, developers can easily swap two of them without realizing. This pattern was also used in DeepBugs (Pradel and Sen, 2018).
- *Change Binary Operator* Checks whether a binary operand was accidentally replaced with another one of the same type. For example, developers very often mix comparison operators in expressions. A similar pattern was also used in DeepBugs (Pradel and Sen, 2018).
- *Change Unary Operator* Checks whether a unary operand was accidentally replaced with another one of the same type (e.g., developers often forget the ! operator in a boolean expression).
- *Change Operand* Checks whether one of the operands in a binary operation was wrong. This pattern was also used in DeepBugs (Pradel and Sen, 2018).
- *More Specific If* Checks whether an extra condition (&& operand) was added in an `if` statement's condition.
- *Less Specific If* Checks whether an extra condition which either this or the original one needs to hold (|| operand) was added in an `if` statement's condition.
- *Missing Throws Exception* Checks whether the fix added a `throws` clause in a function declaration.
- *Delete Throws Exception* Checks whether the fix deleted a `throws` clause in a function declaration.

```
diff –git a/common/src/main/java/com/google/auto/common/MoreTypes.java
    b/common/src/main/java/com/google/auto/common/MoreTypes.java
index d0f40a9..1319092 100644
– a/common/src/main/java/com/google/auto/common/MoreTypes.java
+ b/common/src/main/java/com/google/auto/common/MoreTypes.java
@@ -738,7 +738,7 @@
     * Returns a {@link WildcardType} if the {@link TypeMirror} represents
        a wildcard type or throws
     * an {@link IllegalArgumentException}.
     */
-   public static WildcardType asWildcard(WildcardType maybeWildcardType) {
+   public static WildcardType asWildcard(TypeMirror maybeWildcardType) {
       return maybeWildcardType.accept(WildcardTypeVisitor.INSTANCE, null);
     }
```

Figure 4.1 Example of a *Change Identifier* instance.

```
diff –git a/components/camel-
    pulsar/src/test/java/org/apache/camel/component/pulsar/PulsarConcurrentConsumerInTest.java
    b/components/camel-
    pulsar/src/test/java/org/apache/camel/component/pulsar/PulsarConcurrentConsumerInTest.java
index dcd3a02..1d25f25 100644
– a/components/camel-
    pulsar/src/test/java/org/apache/camel/component/pulsar/PulsarConcurrentConsumerInTest.java
+ b/components/camel-
    pulsar/src/test/java/org/apache/camel/component/pulsar/PulsarConcurrentConsumerInTest.java
@@ -90,12 +90,12 @@
    }

        private PulsarClient concurrentPulsarClient() throws
            PulsarClientException {
-           return new ClientBuilderImpl().serviceUrl(
    getPulsarBrokerUrl()).ioThreads(2).listenerThreads(5).build();
+           return new ClientBuilderImpl().serviceUrl(
    getPulsarBrokerUrl()).ioThreads(5).listenerThreads(5).build();
       return maybeWildcardType.accept(WildcardTypeVisitor.INSTANCE, null);
      }
```

Figure 4.2 Example of a *Change Numeric Literal* instance.

```
diff –git a/components/camel-sjms/src/main/java/org/apache/camel/component/sjms/jms/JmsObjectFactory.java b/components/camel-sjms/src/main/java/org/apache/camel/component/sjms/jms/JmsObjectFactory.java
index 3ed3a24..382ed68 100644
– a/components/camel-sjms/src/main/java/org/apache/camel/component/sjms/jms/JmsObjectFactory.java
+ b/components/camel-sjms/src/main/java/org/apache/camel/component/sjms/jms/JmsObjectFactory.java
@@ -88,7 +88,8 @@
                String messageSelector,
                boolean topic,
                String durableSubscriptionId) throws Exception {
-          return createMessageConsumer(session, destinationName, messageSelector, topic, durableSubscriptionId, true);
+          // noLocal is default false accordingly to JMS spec
+          return createMessageConsumer(session, destinationName, messageSelector, topic, durableSubscriptionId, false);
     }
```

Figure 4.3 Example of a *Boolean Literal* instance.

```
diff –git a/src/test/java/com/puppycrawl/tools/checkstyle/BaseCheckTestSupport.java b/src/test/java/com/puppycrawl/tools/checkstyle/BaseCheckTestSupport.java
index 67f89b8..c3b3ebf 100644
– a/src/test/java/com/puppycrawl/tools/checkstyle/BaseCheckTestSupport.java
+ b/src/test/java/com/puppycrawl/tools/checkstyle/BaseCheckTestSupport.java
@@ -100,7 +100,7 @@
                    + filename).getCanonicalPath();
     }

-    protected void verifyAst(String expectedTextPrintFileName, String actualJavaFileName)
+    protected static void verifyAst(String expectedTextPrintFileName, String actualJavaFileName)
              throws Exception {
          verifyAst(expectedTextPrintFileName, actualJavaFileName, false);
     }
```

Figure 4.4 Example of a *Change Modifier* instance.

```
diff –git a/modules/DesktopDataLaboratory/src/main/java/org/gephi/desk-
    top/datalab/ConfigurationPanel.java
    b/modules/DesktopDataLaboratory/src/main/java/org/gephi/desk-
    top/datalab/ConfigurationPanel.java
index f28c614..f295bd3 1006444
– a/modules/DesktopDataLaboratory/src/main/java/org/gephi/desk-
    top/datalab/ConfigurationPanel.java
+ b/modules/DesktopDataLaboratory/src/main/java/org/gephi/desk-
    top/datalab/ConfigurationPanel.java
@@ -130,7 +130,7 @@
     }

     private boolean canChangeTimeRepresentation(GraphModel graphModel) {
–        if (graphModel.getGraph().getEdgeCount() > 0) {
+        if (graphModel.getGraph().getNodeCount() > 0) {
             return false;//Graph has to be empty
         }
```

Figure 4.5 Example of a *Wrong Function Name* instance.

```
diff –git a/ee/src/main/java/org/jboss/as/ee/component/ComponentDescription.java
    b/ee/src/main/java/org/jboss/as/ee/component/ComponentDescription.java
index f9b99d2..76f83cc 100644
– a/ee/src/main/java/org/jboss/as/ee/component/ComponentDescription.java
+ b/ee/src/main/java/org/jboss/as/ee/component/ComponentDescription.java
@@ -543,7 +543,7 @@
                         configuration.getModuleName(),
                         configuration.getApplicationName()
                 );
–
    injectionConfiguration.getSource().getResourceValue(serviceBuilder,
    context, managedReferenceFactoryValue);
+               injectionConfiguration.getSource().getResourceValue(
    resolutionContext, serviceBuilder, context,
    managedReferenceFactoryValue);
         }
     }
 }
```

Figure 4.6 Example of a *Same Function More Args* instance.

```
diff -git a/src/main/java/com/zaxxer/hikari/pool/HikariPool.java
    b/src/main/java/com/zaxxer/hikari/pool/HikariPool.java
index 19b49e7..3a87b7f 100644
- a/src/main/java/com/zaxxer/hikari/pool/HikariPool.java
+ b/src/main/java/com/zaxxer/hikari/pool/HikariPool.java
@@ 167,7 +167,7 @@
                final long now = clockSource.currentTime();
                if (poolEntry.evict ||
                    (clockSource.elapsedMillis(poolEntry.lastAccessed, now) >
                    ALIVE_BYPASS_WINDOW_MS &&
                    !isConnectionAlive(poolEntry.connection))) {
                    closeConnection(poolEntry, "(connection evicted or
                        dead)"); // Throw away the dead connection and try
                        again
-                   timeout = hardTimeout -
    clockSource.elapsedMillis(startTime, now);
+                   timeout = hardTimeout -
    clockSource.elapsedMillis(startTime);
                }
                else {
                    metricsTracker.recordBorrowStats(poolEntry, startTime);
```

Figure 4.7 Example of a *Same Function Less Args* instance.

```
diff -git a/metrics-servlet/src/test/java/com/yammer/metrics/reporting/tests/AdminServletTest.java
    b/metrics-servlet/src/test/java/com/yammer/metrics/reporting/tests/AdminServletTest.java
index e9d1b4e..4f8601e
- a/metrics-servlet/src/test/java/com/yammer/metrics/reporting/tests/AdminServletTest.java
+ b/metrics-servlet/src/test/java/com/yammer/metrics/reporting/tests/AdminServletTest.java
@@ -42,7 +42,7 @@

      @Before
      public void setUp() throws Exception {
-         when(context.getContextPath()).thenReturn("/context");
+         when(request.getContextPath()).thenReturn("/context");

          when(config.getServletContext()).thenReturn(context);
```

Figure 4.8 Example of a *Same Function Change Caller* instance.

```
diff --git a/servers/src/main/java/tachyon/master/BlockInfo.java
    b/servers/src/main/java/tachyon/master/BlockInfo.java
index 10f3b21..ec659db 100644
--- a/servers/src/main/java/tachyon/master/BlockInfo.java
+++ b/servers/src/main/java/tachyon/master/BlockInfo.java
@@ -187,7 +187,8 @@
           } catch (NumberFormatException nfe) {
             continue;
           }
-          ret.add(new NetAddress(resolvedHost, resolvedPort, -1));
+         // The resolved port is the data transfer port not the rpc port
+          ret.add(new NetAddress(resolvedHost, -1, resolvedPort));
         }
       }
     }
```

Figure 4.9 Example of a *Same Function Swap Args* instance.

```
diff --git a/core/server/worker/src/main/java/alluxio/worker/netty/DataServerReadHandler.java
    b/core/server/worker/src/main/java/alluxio/worker/netty/DataServerReadHandler.java
index 97a07fa..195d89a 100644
--- a/core/server/worker/src/main/java/alluxio/worker/netty/DataServerReadHandler.java
+++ b/core/server/worker/src/main/java/alluxio/worker/netty/DataServerReadHandler.java
@@ -393,7 +393,7 @@
     @GuardedBy("mLock")
     private boolean shouldRestartPacketReader() {
       return !mPacketReaderActive && !tooManyPendingPackets() &&
          mPosToQueue < mRequest.mEnd
-           && mError != null && !mCancel && !mEof;
+           && mError == null && !mCancel && !mEof;
     }
   }
```

Figure 4.10 Example of a *Change Binary Operator* instance.

```
diff –git a/core/client/src/main/java/alluxio/client/file/FileInStream.java
    b/core/client/src/main/java/alluxio/client/file/FileInStream.java
index b263009..5592db2 100644
– a/core/client/src/main/java/alluxio/client/file/FileInStream.java
+ b/core/client/src/main/java/alluxio/client/file/FileInStream.java
@@ -454,7 +454,7 @@

     // If this block is read from a remote worker but we don't have a
        local worker, don't cache
     if (mCurrentBlockInStream instanceof RemoteBlockInStream
–        && BlockStoreContext.INSTANCE.hasLocalWorker()) {
+        && !BlockStoreContext.INSTANCE.hasLocalWorker()) {
       return;
     }
```

Figure 4.11 Example of a *Change Unary Operator* instance.

```
diff –git a/modules/VisualizationImpl/src/main/java/org/gephi/visual-
    ization/swing/StandardGraphIO.java
    b/modules/VisualizationImpl/src/main/java/org/gephi/visual-
    ization/swing/StandardGraphIO.java
index fd49c8a..67b74a9 100644
– a/modules/VisualizationImpl/src/main/java/org/gephi/visualization/swing/StandardGraphIO.java
+ b/modules/VisualizationImpl/src/main/java/org/gephi/visualization/swing/StandardGraphIO.java
@@ -470,7 +470,7 @@
           float newCameraLocation = Math.max(newCameraLocationX,
              newCameraLocationY);

           graphDrawable.cameraLocation[0] = limits.getMinXoctree() +
              graphWidth / 2;
–          graphDrawable.cameraLocation[1] = limits.getMinYoctree() +
    graphWidth / 2;
+          graphDrawable.cameraLocation[1] = limits.getMinYoctree() +
    graphHeight / 2;
           graphDrawable.cameraLocation[2] = newCameraLocation;

           graphDrawable.cameraTarget[0] = graphDrawable.cameraLocation[0];
```

Figure 4.12 Example of a *Change Operand* instance.

```
diff –git a/hazelcast/src/main/java/com/hazelcast/impl/ConcurrentMapManager.java
    b/hazelcast/src/main/java/com/hazelcast/impl/ConcurrentMapManager.java
index b01c711..85eb787 100644
– a/hazelcast/src/main/java/com/hazelcast/impl/ConcurrentMapManager.java
+ b/hazelcast/src/main/java/com/hazelcast/impl/ConcurrentMapManager.java
@@ -546,7 +546,7 @@
            }
            for (Future\u003cPairs\u003e future : lsFutures) {
                Pairs pairs = future.get();
-               if (pairs != null) {
+               if (pairs != null && pairs.getKeyValues()!=null) {
                    for (KeyValue keyValue : pairs.getKeyValues()) {
                        results.addKeyValue(keyValue);
                    }
```

Figure 4.13 Example of a *More Specific If* instance.

```
diff –git a/modules/swagger-core/src/main/java/io/swagger/v3/core/jackson/ModelResolver.java
    b/modules/swagger-core/src/main/java/io/swagger/v3/core/jackson/ModelResolver.java
index baea6e8..aeca799 100644
– a/modules/swagger-core/src/main/java/io/swagger/v3/core/jackson/ModelResolver.java
+ b/modules/swagger-core/src/main/java/io/swagger/v3/core/jackson/ModelResolver.java
@@ -999,7 +999,7 @@
                    }
                }
            }
-           if (subtypeProps.isEmpty()) {
+           if (subtypeProps == null || subtypeProps.isEmpty()) {
                child.setProperties(null);
            }
        }
```

Figure 4.14 Example of a *Less Specific If* instance.

```
diff –git a/example/src/main/java/io/netty/example/securechat/SecureChatServer.java
    b/example/src/main/java/io/netty/example/securechat/SecureChatServer.java
index 6dad108..19a9dac 100644
– a/example/src/main/java/io/netty/example/securechat/SecureChatServer.java
+ b/example/src/main/java/io/netty/example/securechat/SecureChatServer.java
@@ -31,7 +31,7 @@
            this.port \u003d port;
        }

-       public void run() {
+       public void run() throws InterruptedException {
            ServerBootstrap b \u003d new ServerBootstrap();
            try {
                b.eventLoop(new NioEventLoop(), new NioEventLoop())
```

Figure 4.15 Example of a *Missing Throws Exception* instance.

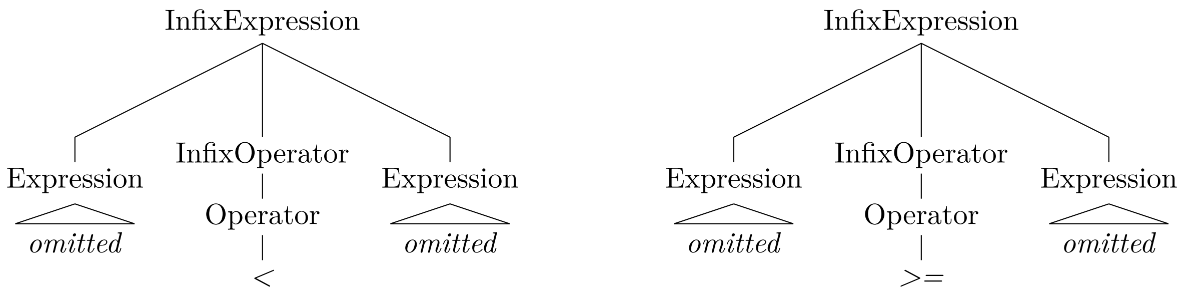
```
diff –git a/core/server/src/main/java/tachyon/web/WebInterfaceAbstractMetricsServlet.java
    b/core/server/src/main/java/tachyon/web/WebInterfaceAbstractMetricsServlet.java
index 23ae5cd..2773882 100644
– a/core/server/src/main/java/tachyon/web/WebInterfaceAbstractMetricsServlet.java
+ b/core/server/src/main/java/tachyon/web/WebInterfaceAbstractMetricsServlet.java
@@ -43,13 +43,12 @@
   }

   /**
–   * Populates key, value pairs for UI display.
+   * Populates operation metrics for displaying in the UI
    *
    * @param request The {@link HttpServletRequest} object
–   * @throws IOException if an I/O error occurs
    */
   protected void populateCountersValues(Map<String, Metric> operations,
–      Map<String, Counter> rpcInvocations, HttpServletRequest request)
   throws IOException {
+      Map<String, Counter> rpcInvocations, HttpServletRequest request){

     for (Map.Entry<String, Metric> entry : operations.entrySet()) {
       if (entry.getValue() instanceof Gauge) {
```

Figure 4.16 Example of a *Delete Throws Exception* instance.

4.2.7 SStuB Pattern Matching

Finally, each pair of ASTs is automatically checked for fitting any of the SStuB patterns. To achieve this we utilize the fact that we discussed previously in Section 4.2.4. That is if we utilize the AST of the file before applying any repair changes and then apply only a single statement change, then this pair of ASTs differs only by a very small subtree of the AST. As discussed we can start on the roots of the two trees and perform a simultaneous DFS to find the first subtree that the two ASTs differ upon. Then, each pattern is expressed as satisfying whether a pattern specific mutation operation on the original AST can produce the fixed one. We note though that to satisfy the pattern we also make sure that no other illegal changes exist that would dissatisfy the pattern. The subtrees are representing by their root AST node which contains references to its children nodes and so on, as such we only need the root nodes. These roots are always subclasses of eclipse's ASTNode class.[5] If any of the AST subtree pair roots is not of the correct type then the pattern is immediately violated. We then move into comparing the pairs of child nodes. For instance in order to check whether the wrong operator pattern is satisfied. We know

[5]https://help.eclipse.org/2020-09/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTNode.html



(a) Original statement subtree (b) Fixed statement subtree

Figure 4.17 Example input pair of subtrees for the *Change Binary Operator* pattern.

that the roots should be *InfixExpressions* and their children should be an *Expression* (left child), an *InfixOperator*, and another *Expression* (right child). First, we need to make sure that the left child *Expression* pairs and the right *Expression* pairs are equal. These represent the left operands and right operands of the *InfixExpression* respectively. If any of these pairs are not equal then the pattern is not satisfied. Last, we need to check that the InfixOperator pair elements are not equal. We will not describe the exacts checks that each pattern applies as they are trivial to understand by knowing the bug type and types of the root's children nodes. An example of two subtrees that would be an input pair are illustrated in Figure 4.17. All instances are added to the single-statement dataset, while only those that match SStuB patterns are saved in the SStuBs one.

4.3 ManySStuBs4J Dataset Statistics

The ManySStuBs4J dataset consists of 10,231 and 63,923 instances of single statement bugs mined from 12,598 and 86,771 bug-fix commits with only single-statement changes respectively for each version. Consequently, on average almost 2 single statement bugs and 0.75 SStuBs were mined per valid commit. The data is saved in JSON files and detailed information is available in the GitHub repository. Each SStuB instance is also annotated with the SStuB pattern satisfied, the project's name, the Java file's name, the hashes of the fix inducing commit and its parent, the line at which the bug starts, and the AST subtree's location. In some cases a statement might fit more than one patterns. In those cases it is counted as separate instances. However, in most cases the patterns are distinct. The statistics for each of the 16 SStuB patterns of the ManySStuBs4J dataset are shown in Table 4.1. Patterns that are similar are grouped together (e.g., patterns that concern functions) and sorted in descending frequency order. The three

Table 4.1 Statistics for each SStuB pattern The first two columns correspond to the smaller version of the dataset while the last two to the large version.

Pattern Name	***SStuBs***	***Ratio***	***SStuBs L***	***Ratio L***
Change Identifier Used	3265	12.78%	22668	14.75%
Change Numeric Literal	1137	4.45%	5447	3.55%
Change Modifier	1852	7.25%	5011	3.26%
Change Boolean Literal	169	0.66%	1842	1.20%
Wrong Function Name	1486	5.82%	10179	6.62%
Same Function More Args	758	2.97%	5100	3.32%
Same Function Less Args	179	0.70%	1588	1.03%
Same Function Wrong Caller	187	0.73%	1504	0.98%
Same Function Swap Args	127	0.50%	612	0.39%
Change Binary Operator	275	1.08%	2241	1.46%
Change Unary Operator	170	0.67%	1016	0.66%
Change Operand	120	0.47%	807	0.53%
Less Specific If	215	0.84%	2813	1.83%
More Specific If	175	0.69%	2381	1.55%
Missing Throws Exception	68	0.27%	206	0.13%
Delete Throws Exception	48	0.19%	508	0.33%
TOTAL NO DOUBLE COUNTS	8438	33.04%	51433	33.47%
TOTAL	10231	40.06%	63923	41.59%

most common SStuB patterns are *Change Identifier Used*, *Wrong Function Name*, and *Change Numeric Literal*.

We note that the mined bugs have not been annotated by severity and we expect that to vary. Some of the bugs appear in test code. Although bugs in test code will not reach a final product, they can have significant effect on it as they can potentially mask important bugs in it. Test oracle errors can bring confusion that slows down the debugging process while fixing them improvGB es the performance of fault localization algorithms (Guo et al., 2015). Such bugs might also be quite tedious to locate as it is very rare to test a test suite and even if we follow that logic we would have to endlessly create tests for the tests. By design we do not attempt to restrict the bugs to those that have a failing test case. The goal is to reproduce the situtations that the bugs happen in the wild. Lastly, as it was recently shown, unit tested code does not appear to be associated with fewer failures while increased coverage is associated with more failures (Chioteli et al., 2019).

It should also be noted that we have not performed a detailed analysis over how interesting or serious the SStuBs are. This is left as an open research question. Currently we can only provide anecdotal evidence by manually inspecting a small set of them suggesting that many but not all the instances in the dataset are interesting as there are definitely exceptions. In many cases but not always these changes would have an effect over the program's output. Last, the degree in which these changes could be masked has also not been thoroughly investigated but we note than in certain instances masking would be impossible.

4.4 Research Questions

Although the paper focuses on the dataset, we run a simple analysis to support our design decision to focus our new dataset on SStuBs. In order to explore whether the SStuB patterns are useful targets for program repair techniques, we asked two research questions.

RQ1. Are SStuBs common in open-source code?

We measured for each SStuB type the percentage of single statement modifications that are not clear refactorings and fit the pattern. These are visualized in Table 4.1. For each project P we also estimated the following two densities for the mined SStuBs: (a) the number of SStuBs in project P / total lines in P at the final snapshot and (b) the number of SStuBs in project P / total lines added and deleted in P by the final snapshot. Thus, estimating the frequency per line of code modifications in the project's history. That is counting any line that was added or deleted to the project from the start to its latest version. A line modification is counted twice (once as a deletion and once as an addition). Once for deleting the old and once for adding the new line. Comments and empty lines were excluded from these estimations. We found that in the smaller version of the dataset SStuBs appear with densities of about 2,400 and 30,000 lines of code (LOC) respectively.

We also estimated the same densities for the larger dataset variant. We found that such bugs appear with a frequency of about 1,600 and 20,000 LOC respectively. As a threat to validity, we acknowledge that the number of LOC in the final snapshot may not be the most informative denominator for a measure of bug density, but developing better ones is a thorny issue left for future work.

RQ2. Can SStuBs be spotted by existing tools such as static analyzers?

We measure the proportion of bugs in our dataset that can be identified by the popular static analysis tool SpotBugs.[6] If SpotBugs reports any bug for the line containing the SStuB then we consider that SpotBugs successfully detected it. We find that SpotBugs could only locate about 12% of SStuBs while also reporting more than 200 million possible bugs when configured to report all warnings, even those with low confidence. In fact, as explained the actual recall is even lower. This is confirmed by a recent study where three static bug detectors including SpotBugs located only 4.5% of bugs (Habib and Pradel, 2018). This means that a developer would have to look through hundreds of thousands of warnings produced by SpotBugs to locate a single SStuB. This highlights the necessity for tools that are specifically built to detect SStuBs. The scripts used to run and evaluate SpotBugs are also available in our repository.

4.5 Evaluating the Plastic Surgery Hypothesis

The SStuB patterns presented in this chapter relate to those used by mutation and pattern-based repair tools. It follows that they could easily be applied as operators in genetic programming based approaches. These approaches rely on the insight that the content of new code can often be assembled out of fragments of code that already exist in the code base. This insight has been dubbed the plastic surgery hypothesis (Barr et al., 2014). Taking advantage of this insight is what allows genetic programming based approaches to limit their vast search space (Arcuri and Xin Yao, 2008; Weimer, 2006). One reason why this insight holds is that the same bug appears in multiple locations, but, when, fixed, is not likely to be fixed everywhere (Barr et al., 2014). However this is probably not the only reason why it holds. For example, many bugs are introduced via copy pasting code and making small changes. Forgetting to make one such change or applying the wrong one introduces a bug. In this case the fragments of code needed to synthesize the fix are highly likely to already exist in the source code. Thus, many source code change that take place during development can often be constructed from

[6]https://spotbugs.github.io/

snippets of code already located in the same program. These snippets are called grafts (Weimer et al., 2009).

Since genetic programming approaches are based on the graftability (i.e., to what degree the fixes are grafts) of existing code it is reasonable to question in what degree are commits graftable in general. Barr et al. (2014) studied a set of 15,723 commits and found that on average 11% of these are 100% graftable from their parent commit. While 43% are partially graftable from their parent commit falling into the interval [0.5..1). What is even more interesting is that the same study found no strong correlation between commit size and graftability. Inspired by these results we investigated to what extend SStuB fixes contained in the dataset are graftable. The motivation behind this experiment is to investigate whether SStuBs do present a particularly high degree of graftability. Developers would benefit greatly if they could receive fix recommendations that are correct with extremely high probability especially for those instances that are not trivial to fix. Additionally if SStuBs do employ high graftability that would indicate that specific APR methods that take full account of this could emerge. Assuming an effective SStuB specific fault localization system we could potentially build an APR system that does finally achieve the coveted in the industry "sweet spot" between high precision and adequate recall. For instance a repair method could quickly find the highly similar statements in the program and use them to synthesize the repair. In that case genetic programming approaches would also greatly benefit as the search space would be extremely limited and they would be able to fix but only in cases that a correct test suite is available.

In order to compute graftability for each instance in the dataset we retrieved the associated fixed version of the SStuB instance provided by the dataset and tokenized it using the JavaParser library (van Bruggen et al., 2020). We then retrieve the buggy version of the commit (parent commit of the fixed version) and tokenize the source code files of that version using again JavaParser. Finally, we check whether the sequence of tokens for the fix exists in the tokens of the buggy version. Note that we excluded any *Change Modifier* instances from the experiment as these statements are mainly definitions. Even if we excluded the defined identifier most possible modifiers would be expected to appear somewhere in the program inflating the results. Not surprisingly we found that SStuBs are indeed characterized by high graftability. Specifically, that almost 59% of SStuBs have grafts in the buggy version of the code. Figure 4.18 illustrates an actual example for the dataset where the correct function call appears just a couple of lines later.

```
          assertEquals(1, taskService.createTaskQuery()
                  .or()
                  .taskInvolvedUser("involvedUser")
-                 .taskInvolvedGroups(groups)
-                 .taskInvolvedGroupsIn(groups)
                  .endOr()
                  .count());


          assertEquals(1,historyService.createHistoricTaskInstanceQuery()
                  .or()
                  .taskInvolvedUser("involvedUser")
                  .taskInvolvedGroupsIn(groups)
                  .endOr()
                  .count());
```

Figure 4.18 Example of a graft from the SStuB dataset.

4.6 Limitations - Threats to Validity

Although unlikely, it is possible for our SZZ like methodology to extract a pair of aligned statements that are unrelated (i.e., one line was deleted and one was added). We do spot refactorings but there is no guarantee that we have detected 100% of them. The heuristic used to spot bug fixing commits could introduce false positives, but this is mitigated by the fact that we focus on single line commits and as already discussed the false positive rate is low. Our dataset will not be useful for evaluating whether repair systems are good at fixing larger bugs. Our dataset is restricted to Java but could be replicated for other languages by using a parser and creating a module that checks if an AST pair fits any of the SStuB patterns. The precise set of patterns might vary across languages and determining these might be an interesting direction for future work.

4.7 Conclusions

We introduced a new, large-scale dataset of real-world SStuBs, simple one-statement bugs, in Java for the evaluation of program repair techniques. The distinguishing feature of our dataset is that where possible, the SStuBs are categorized into one of 16 bug templates, which are inspired by those considered in state-of-the-art program repair methods. These types of bugs often result in code that compiles, which means that they are particularly interesting for automated repair. We find that SStuBs occur relatively often — one per 1,600 LOC in the projects we study — making them potentially a promising evaluation dataset for repair techniques that could be used to estimate their actual recall. We also find that most of the SStuB instances are graftable from existing code, which could lead into an interesting direction of future work. The data could also be used to answer other research questions, such as empirical questions about how and when simple bugs are introduced, or about evaluating program repair techniques for small bugs. Also, it can aid in evaluating machine learning systems that learn to localize simple bugs via examples (Pradel and Sen, 2018) or a language model's entropy (Karampatsis et al., 2020). Moreover, SStuBs might be a good fit for errors to automatically fix and provide feedback about in massive open online courses. Moreover, coverage information for the maven projects with tests suits in the dataset could be used to estimate how often do tests cover SStuBs. Last, we note that although it would be reasonable to use the dataset extracted in this chapter for later parts of this thesis, this is not the case. A major reason for this are timing constraints like submitting the PhD thesis on time while also working from home with limited resources due to the current pandemic situation. However, there is ongoing work using this dataset and a challenge proposal centring around it was submitted for the mining challenge track of the mining software repositories (MSR) 2021 conference. The proposal was accepted and the dataset is the subject of the mining challenge track of MSR 2021 (Karampatsis et al., 2021).

Chapter 5

Low Resource Contextual Embeddings for Source Code

"A man is known by the company he keeps."

–Aesop

Learning rich representations for source code is an open problem that has the potential to enable software engineering and development tools. Some work on machine learning for source code has used hand engineered features (Long and Rinard, 2016), but designing and implementing such features can be tedious and error-prone. Moreover, these would be hard to reuse between different systems, especially for multiple goals and domains. For this reason, other work considers the task of learning a representation of source code from data (Allamanis et al., 2018a). Many models of source code are based on learned representations called embeddings, which transform words into a continuous vector space (Mikolov et al., 2013b). Currently in software engineering (SE) researchers have used static embeddings (Harer et al., 2018; Pradel and Sen, 2018; White et al., 2019), which map a word to the same vector regardless of its context. However, recent work in natural language processing (NLP) has found that contextual embeddings can lead to better performance (Devlin et al., 2018; Liu et al., 2019; Peters et al., 2018; Yang et al., 2019). Contextualized embeddings assign a different vector to a word based on the context it is used. For NLP this has the advantage that it can model phenomena like polysemy. A natural question to ask is if these methods would also be beneficial for learning better SE representations. The context that we utilize in this work is based on previous and following tokens. However, in the future other kinds of contextual information could be utilized and prove very useful. Some examples are data or control dependencies or utilizing class hierarchy information. Such elements may contain very essential information for source code.

In this chapter[1], we introduce a new set of contextual embeddings for source code. Contextual embeddings have several potential modelling advantages that are specifically suited to modelling source code:

- Surrounding names contain important information about an identifier. For example, for a variable name, surrounding tokens might include functions that take that variable as an argument or assignments to the variable. These tokens provide indirect information about possible values the variable could take, and so should affect its representation. Even keywords can have very different meanings based on their context. For instance, a private function is not the same as a private variable or a private class (in the case of Java / C++).

- Contextual embeddings assign a different representation to a variable each time it is used in the program. By doing this, they can potentially capture how a variable's value evolves through the program execution.

- Contextual embeddings enable the use of transfer learning. Pre-training a large neural language model and querying it for contextualized representations while simultaneously fine-tuning for the specific task is a very effective technique for supervised tasks for which there is a small amount of supervised data available. As a result only a small model needs to be fine-tuned atop the pre-trained model, without the need for task-specific architectures nor the need of training a large model for each task separately.

This chapter highlights the potential of contextual code embeddings for program repair. Automatically finding bugs in code is an important open problem in SE. Even simple bugs can be hard to spot and repair. A promising approach to this end is name-based bug detection, introduced by DeepBugs (Pradel and Sen, 2018). The current state-of-the-art in name-based bug detection relies on static representations from Word2Vec (Mikolov et al., 2013b) to learn a classifier that distinguishes correct from incorrect code for a specific bug pattern. We introduce a new set of contextualized embeddings for code and explore its usefulness on the task of name-based bug detection. Our method significantly outperforms DeepBugs as well as other static representations methods on both the DeepBugs dataset as well as a new previously unused test set of JavaScript projects. We

[1]Most of the material in this chapter has appeared before as an unpublished pre-print in (Karampatsis and Sutton, 2020b) and the manuscript was written by myself with Charles Sutton providing feedback and editing.

could have also evaluated the method on the SStuBs dataset. However, due to time constrains as well as issues with the ongoing pandemic we chose to evaluate with the DeepBugs dataset as it was simpler and faster to do since otherwise we would have to reimplement DeepBugs for Java and add a mechanism that generates synthetic bugs for every SStuB pattern in the dataset. Doing so also allows comparison with previous work which is essential. We note though that there is ongoing work on this as well as exploring other problems with the SStuBs dataset of Chapter 4 not included in the scope of this thesis (Karampatsis et al., 2021). The implementation is available in a public GitHub repository.[2]

We next provide a brief overview of the rest of the chapter. Section 5.1 discusses related work. Section 5.2 offers a detailed explanation of the architecture of ELMo as well as the training procedure, and the adaptation procedure tha learns weight for new tasks. Section 5.3 describes the parameters choices for training SCELMo as well as the training, validation, and test data. Section 5.4 offers a detailed description of how the learned contextual embeddings can be incorporated within a recent machine learning-based bug detection system to improve it. It also presents the baselines used during the evaluation, the name extraction heuristic used along with the expansions we performed over it and presents two variants of the model. The normal SCELMo model and a baseline, which instead of the normal code sequence creates queries with heuristic approach that also uses the name extraction heuristic. This baselines is called No-Context ELMO. We note the models are not evaluated against a BERT model (Devlin et al., 2018) as during the time this work was performed no such model trained on source code was available and training from scratch a BERT model in academia is currently infeasible. Section 5.5 describes the experiments we performed and interprets the results for the various evaluation scenarios. Section 5.6 introduces the question of whether neural bug-finding is practically useful and offers some first insights to this question by mining a small dataset of real bugs using the methodology described in Chapter 4 and using it to evaluate the model. Finally, Section 5.7 concludes the chapter.

5.1 Related Work

Unsupervised static word embeddings have been extensively used to improve the accuracy of supervised tasks in NLP (Turian et al., 2010). Notable examples of such methods are Word2Vec (Mikolov et al., 2013b) and GloVe (Pennington et al., 2014). However, the above models learn only a single context-independent word representation. To overcome

[2]https://github.com/mast-group/DeepSStuBs

this problem some models (Bojanowski et al., 2017; Wieting et al., 2016) enhance the representations with subword information, which can also somewhat deal with out-of-vocabulary words. Another approach is to learn a different representation for every word sense (Neelakantan et al., 2014) but this requires knowing the set of word senses in advance. More recent methods overcome the above issues by learning contextualized embeddings. Melamud et al. (2016) encode the context surrounding a pivot word using a bidirectional LSTM. Peters et al. (2018) use a deep bidirectional LSTM, learning word embeddings as functions of its internal states, calling the method Embeddings using Language Models (ELMo). We discuss ELMo in detail in Section 5.2. Devlin et al. (2018) introduced bidirectional encoder representations from transformers (BERT). This method learns pre-trained contextual embeddings by jointly conditioning on left and right context via an attention mechanism.

Program repair is an important task in software engineering and programming languages. For a detailed review see Gazzola et al. (2019); Monperrus (2018a). Many recent program repair methods are based on machine learning. Yin et al. (2018b) learn to represent code edits using a gated graph neural network (GGNN) (Li et al., 2016). Allamanis et al. (2018c) learn to identify a particular class of bugs called variable misuse bugs, using a GGNN. Chen et al. (2019a) introduce SequenceR which learns to transform buggy lines into fixed ones via machine translation. Our work is orthogonal to these approaches and can be used as input in other models.

Our work is also related to code representation methods many of which have also been used in program repair. Harer et al. (2018) learn Word2Vec embeddings for C/C++ tokens to predict software vulnerabilities. White et al. (2019) learn Word2Vec embeddings for Java tokens and utilize them in program repair. Alon et al. (2019b) learn code embeddings using abstract syntax tree paths. A more detailed overview can be found in (Allamanis et al., 2018a; Chen and Monperrus, 2019).

Simultaneously to this work other researchers focused on pre-training contextual embeddings for code and evaluated on a similar bug detection problem on python code (Kanade et al., 2020) and on making a bimodal model programming and natural languages (Feng et al., 2020). Last, we clarify that our approach differs from theirs for the following reasons: 1) It is based on bidirectional LSTMs while theirs uses transformers; 2) and ours is a low-resource model that requires maximum 2 or 3 GPUs to be trained while theirs requires TPUs and thus ours is a much better fit for academia.

5.2 Embeddings from Language Models (ELMo)

ELMo (Peters et al., 2018) computes word embeddings from the hidden states of a language model. Consequently, the embeddings of each token depend on its context of the input sequence, even out-of-vocabulary (OOV) tokens have effective input representations. In this section, we briefly describe the ELMo embeddings.

The first step is that a neural language model is trained to maximize the likelihood of a training corpus. The architecture used by ELMo a bidirectional LSTM with L layers and character convolutions in the input layer. Let the input be a sequence of tokens $(t_1, ...t_N)$. For each token t_k, denote by $\boldsymbol{x}_k^{LM}$ the input representation from the character convolution. Consequently, this representation passes through L layers of forward and backward LSTMs. Then each layer $j \in \{1, ..., L\}$ of the forward LSTM computes a hidden state $\overrightarrow{\boldsymbol{h}_{k,j}^{LM}}$, and likewise the hidden states of the backward LSTM are denoted by $\overleftarrow{\boldsymbol{h}_{k,j}^{LM}}$. The parameters for the token representation and for the output softmax layer are tied for both directions, while different parameters are learned for each direction of the LSTMs.

During training the model maximizes the log likelihood of both the forward and backwards LSTM jointly as shown in Equation 5.1:

$$\sum_{k=1}^{N}(logp(t_k|t1, ..., t_{k-1}; \Theta_x, \overrightarrow{\Theta}, \Theta_s)) + (logp(t_k|tk + 1, ..., t_N; \Theta_x, \overleftarrow{\Theta}, \Theta_s)) \quad (5.1)$$

The parameters Θ_x for the token representation and Θ_s for the Softmax layer are tied for both directions, while different parameters are learned for each direction of the LSTMs.

After the language model has been trained, we can use it within another downstream task by combining the hidden states of the language model from each LSTM layer. This process is called ELMo. For each token t_k of a sentence in the test set, the language model computes $2L + 1$ hidden states, one in each direction for each layer, and then the input layer. To make the following more compact, we can write these as $h_{k,0}^{LM} = x_k^{LM}$ for the input layer, and then $h_{k,j}^{LM} = [\overrightarrow{h_{k,j}^{LM}}, \overleftarrow{h_{k,j}^{LM}}]$ for all of the other layers. The set of these vectors is

$$R_k = \{h_{k,j}^{LM}|j = 0, ..., L\}. \quad (5.2)$$

To create the final representation that is fed to downstream tasks, ELMo collapses the set of representations into a single vector E_k for token t_k. A simplistic approach is to only select the top layer, so that $E_k = h_{k,L}^{LM}$. A more general one, which use in this work, is to combine the layers via fine-tuned task specific weights $\mathbf{s} = (s_1 \dots s_L)$ for every layer.

Then we can compute the embedding for token k as

$$E_k = \gamma \sum_{j=0}^{L} s_j h_{k,j}^{LM}, \tag{5.3}$$

where γ is an additional scalar parameter that scales the entire vector. In our experiments we did not performed fine-tuning and thus used equal weights $s_j = 1/(L+1)$ for each layer and $\gamma = 1$. However, our implementation also supports all the aforementioned ways of collapsing the set of representations.

A potential drawback of the method is that it still utilizes a softmax output layer with a fixed vocabulary that does not scale effectively and it still predicts UNK for OOV tokens which may have a negative effect on the representations.

5.3 Training Source Code ELMo

We describe Source Code ELMo (SCELMo), which trains ELMo on corpora of source code. However, we note that normally ELMo models in other domains are able to effectively utilize much larger representations. The code was tokenized using the esprima JavaScript tokenizer[3]. For training the ELMo model we used a corpus of 150,000 JavaScript Files (Raychev et al. 2016) consisting of various open-source projects. This corpus has previously been used on several tasks (Bavishi et al., 2018; Pradel and Sen, 2018; Raychev et al., 2016). We applied the patch released by Allamanis et al. (2018a) to filter out code duplication as this phenomenon was shown on this and other corpora to result in inflation of performance metrics. This resulted in 64750 training files and 33229 validation files. Since the validation set contains files from the same projects as the train the contained instances might be too similar and unrealistic overestimating. To address this we also created a test set of 500 random JavaScript projects sampled from the top 20,000 open-source JavaScript projects as of May 2019. The test corpus has not been previously utilized in previous work and is a better reflection of the performance of the learned bug detectors. Lastly, it is important to know what the performance of the method will be if we do not have access to training data from the projects on which we would like to find bugs. This is common in practice for many real case scenarios. For training the ELMo model, we use an embedding size of 100 features for each of the forward and backward LSTMs so that each layer sums up to 200 features.

[3]https://esprima.org/

5.4 Contextual Embeddings for Program Repair

In this section, we describe how contextual embeddings can be incorporated within a recent machine learning-based bug detection system, the DeepBugs system of Pradel and Sen (2018). In the first part of this section, we give background about the DeepBugs system, and then we describe how we incorporate SCELMo within DeepBugs. DeepBugs treats the problem of finding a bug as a classification problem. The system considers a set of specific bug types, which are small mistakes that might be made in a program, such as swapping two arguments. For each bug type, DeepBugs trains a binary classifier that takes a program statement as input and predicts whether the statement contains that type of bug. At test time, this classifier can be run for every statement in the program to attempt to detect bugs.

In order to train the model both examples of correct and incorrect (buggy) code are necessary. DeepBugs treats the existing code as correct and randomly mutates it to obtain buggy code. To obtain training examples, we extract all expressions from the source code which are either the function calls with exactly two arguments and all binary expressions. To create instances of buggy code we mutate each of the correct instances. As such, arguments in function calls are swapped, the binary operator in binary expressions is replaced with another random one, and finally randomly either the left or the right operand is replaced by another random binary operand that appears in the same file. Then the classification task is a binary task to predict whether the instance is correct, i.e., it comes from the original code, or whether it is buggy, i.e., it was one of the randomly mutated examples. The validation and test sets are mutated in the same way as the training set. The split between correct and buggy instances has 50/50 class distribution as for each original code instance exactly one mutated buggy counterpart is created.

The architecture for the classifier is a feedforward network with a single hidden layer of 200 dimensions with Relu activations and a sigmoid output layer. For both the input and hidden layers a dropout of 0.2. The network was trained in all experiments for 10 epochs with a batch size of 50 and the RMSProp optimizer. We note that for maintaining a consistent comparison with DeepBugs we kept all the above parameters as well as the optimizer's parameters fixed to the values reported in Pradel and Sen (2018). Tuning these parameters would probably result in at least a small performance increase for our method.

In our experiments, we consider three bug types that address a set of common programming mistakes: swapped arguments of function calls, using the wrong binary operator and using an incorrect binary operand in a binary expression. We focused on

```
// Argument order is inversed.
var delay = 1000;
setTimeout(delay, function() { // Function should be first.
    logMessage(msgValue);
});
```

Listing 5.1 Swapped Arguments Bug

```
// Less instead of greater was used.
if ( x < maximum ) {
  do_something();
  maximum = x;
}
```

Listing 5.2 Incorrect Binary Operator

```
// Call to .length is missing.
if ( index < matrix ) {
  do_something();
}
```

Listing 5.3 Incorrect Binary Opernad

Figure 5.1 Bug type examples.

these bug types as they are the ones that have already appeared in previous work and thus our model should definitely be evaluated on them. The methodology can easily be applied to other bug types. However, as mentioned earlier we did not do so due to certain constraints but it is definitely an essential and reasonable direction for future work. Figure 5.1 illustrates an example of each of the three bug types used in this work.

5.4.1 Input to the Classifier

A key question is how a statement from the source code is converted into a feature vector that can be used within the classifier. DeepBugs uses a set of heuristics that, given a statement and a bug type, return a sequence of identifiers from the statement that are most likely to be relevant. For instance, for the call to setTimeout in Listing 1 the following sequence of identifiers would be extracted: *[setTimeout, delay, function]*. A detailed description of the heuristics is available in Appendix 5.4.2.

These heuristics result in a sequence of program identifiers. These are converted to continuous vectors using word embeddings, concatenated, and this is the input to the classifier. DeepBugs uses Word2Vec embeddings trained on a corpus of code. In our experiments, we train classifiers using three different types of word embeddings. First, we kept the 10,000 most frequent identifiers/literals and assigned to each of them a *random embedding* of 200 features. Second, to reproduce the results of Pradel and Sen (2018), we use the CBOW variant of *Word2Vec* to learn representations consisting of 200 features for the 10,000 most frequent identifiers/literals. Finally, we train a *FastText* embeddings (Bojanowski et al., 2017) on the training set to learn identifier embeddings that contain subword information. The subwords used by FastText are all the character trigrams that appear in the training corpus. Identifiers are therefore composed of multiple subwords. To represent an identifier, we sum the embeddings of each of its subwords and summing them up. This allows the identifier embeddings to contain information about the structure and morphology of identifiers. This also allows the FastText embeddings, unlike the Word2Vec ones, to represent OOV words as a combination of character trigrams.

Note that DeepBugs can detect bugs only in statements that do not contain OOV (out-of-vocabulary) identifiers, because its Word2Vec embeddings cannot extract features for OOV names. Instead our implementation does not skip such instances. Since the original work discarded any instances that contain OOV identifiers we neither know how the method performs on such instances nor how often those appear in the utilized dataset of DeepBugs. Moreover, DeepBugs supported only a specific subset of AST nodes and skipped the rest. For example if a call's argument is a complex expression consisting of other expressions then the call would be skipped. However, we expanded the implementation to support all kinds of AST nodes and to not skip instances with nested expressions as discussed in Section 5.4.2. We note that we still skip an instance if one of its main parts (e.g., a function call's argument) is a complex expression longer than 1,000 characters as such expressions might be overly long to reason about.

5.4.2 Name Extraction Heuristic

In order for DeepBugs to operate it is necessary to extract identifiers or literals for each expression part of the statement. The bug detector for swapped arguments utilizes the following elements of the function call:

Base Object: The expression on which the function is called.

Callee: The called function.

Argument 1: The expression consisting the first argument of the called function.

Argument 2: The expression consisting the first argument of the called function.

Similarly the bug detectors for incorrect binary operators and operands utilize the following elements of the binary expression:

Binary Operator: The binary operator utilized in the expression.

Left Operand: The left operand of the binary expression.

Right Operand: The right operand of the binary expression.

We next describe the extraction heuristic, which is shared by all the bug detectors. The heuristic takes as input a node n representing an expression and returns $name(n)$ based on the following rules:

- Identifier: return its name.
- Literal: return its value.
- this expression: return *this*.
- Update expression with argument x: return $name(x)$.
- Member expression accessing a property p: return $name(p)$.
- Member expression accessing a property $base[p]$: return $name(base)$.
- Call expression $base.callee(...)$: return $name(callee)$.
- Property node n: If $n.key$ does not exist return $name(n.value)$. If $name(n.key)$ does not exist return $name(n.value)$. Otherwise randomly return either $name(n.value)$ or $name()n.key)$.
- Binary expression with left operand l and right operand r: Run the heuristic on both l and r to retrieve $name(l)$ and $name(r)$. If $name(l)$ does not exist return $name(r)$. If $name(r)$ does not exist return $name(l)$. Otherwise randomly return either $name(l)$ ir $name(r)$.
- Logical expression with left operand l and right operand r: Run the heuristic on both l and r to retrieve $name(l)$ and $name(r)$. If $name(l)$ does not exist return $name(r)$. If $name(r)$ does not exist return $name(l)$. Otherwise randomly return either $name(l)$ ir $name(r)$.

- Assignment expression with left operand l and right operand r: Run the heuristic on both l and r to retrieve $name(l)$ and $name(r)$. If $name(l)$ does not exist return $name(r)$. If $name(r)$ does not exist return $name(l)$. Otherwise, randomly return either $name(l)$ ir $name(r)$.
- Unary expression with argument u : Return $name(u)$.
- Array expression with elements l_i : For all l_i that $name(l_i)$ exists randomly choose one of them and return $name(l_i)$.
- Conditional expression with operands c, l, and r: Randomly choose one out of c, l, r for which a name exists and return its name.
- Function expression: return $function$.
- Object expression: return {.
- New expression with a constructor function call c: return $name(c)$.

All random decisions follow a uniform distribution.

5.4.3 Connecting SCELMo to the Bug Detector

We investigated two variants of the bug detection model, which query SCELMo in different ways to get features for the classifier. The first utilizes the heuristic of Section 5.4.2 to extract a small set of identifiers or literals that represent the code piece. For example, for an incorrect binary operand instance we extract one identifier or literal for the left and right operands respectively, and we also extract its binary operator. Then, those are concatenated to form a query to the network. In the case of function calls we extract the identifier corresponding to the name of the called function, one identifier or literal for the first and second argument respectively and an identifiers for the expression on which the function is called. We also add the appropriate syntax tokens (a '.' if necessary, ',' between the two arguments, and left and right parentheses) to create a query that resembles a function call. This baseline approach creates simplistic fixed size queries for the network but does not utilize its full potential since the queries do not necessarily resemble actual code, nor correct code similar to the sequences in the training set for the embeddings. We will refer to this baseline as No-Context ELMo.

Our proposed method, we compute SCELMo embeddings to the language model all the tokens of the instances for which we need representations. Valid instances are functions calls that contain exactly two arguments and binary expressions. To create a fixed-size representation we extract only the features corresponding a fixed set of tokens. Specifically, for functions calls we use the representations corresponding to the first token of the expression on which the function is called, the function name, the first token of the first argument and the first token of the second argument. While, for binary expressions we use those of the first token of the left operand, the binary operator, and the first token of the right operand. Since the representations contain contextual information, the returned vectors can capture information about the rest of the tokens in the code sequence.

5.5 Evaluation

We next discuss the experiments we performed and their corresponding results. We measured the performance of the three baselines as well as those of non-contextual ELMO and SCELMO. Measuring the performance of non-contextual ELMO allows us to evaluate how much improvement is due to specifics of the language model architecture, such as the character convolutional layer which can handle OOVs, and how much is due to the contextual information itself.

5.5.1 Performance on Validation Set

In our first experiment we evaluate the performance of the methods in tasks where training data from the same projects are available. The evaluation performed in this experiment gives a good estimation of how our method performs compared to the previous state-of-the-art technique of DeepBugs. One main difference however is that the evaluation now also includes instances which contain OOV. As a consequence the bug detections tasks are harder than those presented by Pradel and Sen (2018) as their evaluation does not include in both the training and validation set any instance for which an extracted identifier is OOV. Table 5.1 illustrates the performance of the baselines and our models. As one would expect the FastText baseline improves over Word2Vec for all bug types due to the subword information. Moreover, our model SCELMo massively outperforms all other methods. In addition, even no-context ELMo the heuristic version of SCELMo that does not utilize contextual information at test time outperforms the baseline methods showcasing how powerful the pretrained representations are. Finally, although SCELMo achieves 100% accuracy on the Wrong Binary Operator task we caution the reader to not interpret the result as SCELMo having learned a perfect bug detection system. In fact such high accuracy might be a sign of the system overfitting. The main reason behind

Table 5.1 Comparison of ELMo versus non-contextual embeddings for bug detection on a validation set of projects. Data is restricted to expressions that contain only single names.

	Random	Word2Vec	FastText	No-Context ELMo	SCELMo
Swapped Arguments	86.18%	87.38%	89.55%	90.02%	92.11%
Wrong Binary Operator	90.47%	91.05%	91.11%	92.47%	100.00%
Wrong Binary Operand	75.56%	77.06%	79.74%	81.71%	84.23%

this probably lies to the mechanism that generates synthetic bugs for the pattern. The buggy instances generated may contain any possible operator. As such many instances will correspond to unnatural code (Hindle et al., 2012) that we would probably not run into in practice, thus will not usually correspond to actual JavaScript. A possible direction for mitigating this would be to only generate instances with an operator of the same. However, we have not performed any such experiments and thus cannot provide any hard evidence on this but just a reasonable hypothesis.

5.5.2 Including Complex Expressions

In our next experiment we also included instances that contain elements that are complex or nested expressions. For instance, in the original work if one the arguments of a function call or one of the operands of a binary expression is an expression consisting of other expressions then the instance would not be included in the dataset. Several AST node types such as a `NewExpression` node or an `ObjectExpression` were not supported. Figure 5.2 a few examples of instances that would be previously skipped [4]. Such instances were skipped by Pradel and Sen (2018) and not included in their results. We do note though that we still skip very long expressions that contain more than 1000 tokens.

```
// First argument is binary expression
doComputation(x + find_min(components), callback);
```

```
// Second argument is an unsupported node
factory.test(simulator, new Car('Eagle', 'Talon TSi',
   1993));
```

Figure 5.2 Examples of instances that would be skipped by DeepBugs.

[4]The AST is extracted using the acorn parser https://github.com/acornjs/acorn

Table 5.2 Comparison of SCELMo versus static embeddings on bug detection on a validation set of projects. Complex expressions are included in this validation set.

	Random	Word2Vec	FastText	No-Context ELMo	SCELMo
Swapped Arguments	86.37%	87.68%	90.37%	90.83%	92.27%
Wrong Binary Operator	91.12%	91.68%	91.92%	92.75%	100.00%
Wrong Binary Operand	72.73%	74.31%	77.41%	79.65%	87.10%

Table 5.3 Comparison of SCELMo versus static embeddings on bug detection on an external test set of 500 JavaScript projects.

	Random	Word2Vec	FastText	No-Context ELMo	SCELMo
Swapped Arguments	75.79%	78.22%	79.40%	81.37%	84.25%
Wrong Binary Operator	82.95%	85.54%	83.15%	86.54%	99.99%
Wrong Binary Operand	67.46%	69.50%	72.55%	75.74%	83.59%

Similarly to the previous experiment SCELMo significantly outperforms all other models. This is evident in Table 5.2. As in the previous section we caution any reader that although SCELMo achieves 100% accuracy on the Wrong Binary Operator task they should not interpret the result as SCELMo having learned a perfect bug detection system. Lastly, we clarify that the results of this section should not be directly compared to those of the previous one as for this experiment the training set is also larger.

5.5.3 External Test Evaluation

The last experiment's objective is to showcase how the various models would perform on unseen projects as this better illustrates the generalizability of the techniques. The configuration utilized is identical to that of the previous section. By looking at Table 5.3 one can notice that the baselines have a major drop in performance. This is a common finding in machine learning models of code, namely, that applying a trained model to a new software project is much more difficult than to a new file in the same project. In contrast, SCELMo offers up to 15% improvement in accuracy compared to Word2Vec baseline. In fact, impressively enough SCELMo on the external test set is better than the evaluation set one of the baselines.

5.5.4 OOV Statistics

In order to better understand the above results we measured the OOV rate of the basic elements of the code instances appearing in the dataset. Here the OOV rate is calculated based on the vocabulary of 10000 entries utilized by the Word2Vec and random baseline models. These are illustrated in Tables 5.4 and 5.5. We measured the OOV rates for both the version of the dataset used in Section 5.5.4, which we call Train and Validation, and that used in Section 5.5.2, which we call Extended Train and Extended Validation.

Tables 5.4 and 5.5 describe the OOV rates for different parts of the expression types that are considered by the DeepBugs bug detector. A detailed description of the identifiers extraction heuristic can be found in Appendix 5.4.2. We first focus on the swapped arguments bug pattern and consider all of the method call that have exactly two arguments. Each method call contains the function name, a name of the first argument, a name of the second argument, and a base object. The base object is the identifier that would be extracted from the expression (if such an expression exists) on which the function is called. For instance, from the following expression: *window.navigator.userAgent.indexOf("Chrome")*, *userAgent* would be extracted as the base object. Table 5.4 shows for each of the components how often they are OOV. In the expanded version of the dataset if one of the arguments is a complex expression then it is converted into a name based on the heuristic described in Section 5.4.2. The resulting statistics contain valuable information as for instance, it is almost impossible for the Word2Vec baseline to reason about a swap arguments bug if the identifiers extracted for both arguments are OOV.

In a similar manner for the incorrect operand and operator bug patterns we consider all the binary operations. Each binary expression consists of a left and right operand and a name is extracted for each of them. For each operand we also measured the frequency with which the operand corresponds to certain common types such as identifier, literal or a *ThisExpression*.

Table 5.4 OOV statistics for calls with exactly two arguments (Swapped arguments instances). The statistics are calculated on variants of the DeepBugs dataset.

	Train	Expanded Train	Validation	Expanded Validation
Two Arguments Calls	**574656**	**888526**	**289061**	**453486**
Calls Missing Base Object	25.07%	28.63%	25.63%	28.80%
Base Object Missing or OOV	34.56%	37.38%	35.57%	38.07%
Function Name OOV	20.69%	17.07%	20.33%	16.94%
First Argument OOV	31.01%	36.99%	31.64%	37.15%
Second Argument OOV	27.25%	22.86%	27.94%	23.49%
Both Arguments OOV	11.33%	9.57%	11.96%	10.16%
Base and Function Name OOV	10.20%	8.32%	10.39%	8.61%
Base and Arguments OOV	4.21%	3.31%	4.88%	3.77%
Function Name and Arguments OOV	2.86%	2.26%	2.85%	2.28%
All Elements OOV	1.53%	1.18%	1.61%	1.27%

Table 5.5 OOV statistics for binary operations.

	Train	Expanded Train	Validation	Expanded Validation
Binary Operations	**1075175**	**1578776**	**540823**	**797108**
Left Operand OOV	25.40%	28.84%	26.04%	29.55%
Right Operand OOV	20.37%	23.98%	20.74%	24.55%
Both Operands OOV	7.82%	11.29%	8.24%	11.88%
Unknown Left Operand Type	83.36%	87.80%	83.14%	87.74%
Unknown Right Operand Type	48.48%	47.23%	48.47%	47.05%
Both Operand Types Unknown	33.34%	36.06%	33.20%	35.87%
All OOV or Unknown	3.59%	4.03%	3.81%	4.3%

5.6 Is Neural Bug-Finding Useful in Practice?

Although related work (Allamanis et al., 2018c; Pradel and Sen, 2018; Vasic et al., 2019) has shown that there is great potential for embedding based neural bug finders, the evaluation has mostly focused on synthetic bugs introduced by mutating the original code. However, there is no strong indication that the synthetic bugs correlate to real ones, apart from a small study of the top 50 warnings for each bug type produced by DeepBugs. A good example is the mutation operation utilized for the incorrect binary operator bug. There is no guarantee that the generated instances will correspond to natural code (Hindle et al., 2012) as any operator can be replaced with any other operator. Similar issues might appear for the patterns but in the incorrect binary operator case its much more obvious how unnatural code may arise. This can potentially create a classifier with a high bias towards correlating buggy code to unnatural or worst case syntactically incorrect code, thus hindering the model's ability to generalize on real bugs. Ideally, in an industrial environment we would like the resulting models to achieve a false positive rate of less than 10 % (Sadowski et al., 2015). Sadly, high true positive rates are not to be expected as well since static bug detectors were shown to be able to detect less than 5% of bugs (Habib and Pradel, 2018) contained in the Defects4J corpus (Just et al., 2014b) and less than 12% in a single-statement bugs corpus (Karampatsis and Sutton, 2020a). We note that in the second case the static analysis tool is given credit by reported any warning for the buggy line, so the actual percentage might be lower than the reported one.

We next make a first step on investigating the practical usefulness of our methods by applying the classifiers of the previous section on a small corpus of real JavaScript bugs. We think that this is a very hard yet interesting problem that should be carefully examined in future work. Our current investigation is limited and only spans across one language and three bug types. However, the method is easy to apply on other languages

Table 5.6 Real bug mined instances.

	Swapped Arguments	Wrong Binary Operator	Wrong Binary Operand
Mined Instances	303	80	1007

Table 5.7 Real bug identification task recall and false positive rate (FPR).

	Word2Vec-Recall	Word2Vec-FPR	SCELMo-Recall	SCELMo-FPR
Swapped Arguments	3.34%	0.33%	49.67%	33.78%
Wrong Binary Operator	8.95%	7.70%	0.00%	0.00%
Wrong Binary Operand	11.99%	12.11%	15.81%	14.34%

as well as different bug types. What we need to do is to first extract a dataset of real bugs for the language of interest. This can be easily be achieved by for example utilizing the SStuBs mining tool from Chapter 4. We would need a parser for that language as well as to define the SStuB patterns to be used for extraction. Last we would need to implement the methodology of DeepBugs for synthetic bug instance generation so that we can create training examples.

As discussed above, in order to mine a corpus of real bug changes we used the methodology described in Chapter 4 (Karampatsis and Sutton, 2020a). We note that we adapted the implementation to utilize the Rhino JavaScript parser[5]. The methodology extracts bug fixing commits and filters them to only keep those that contain small single-statement changes. Finally, it classifies each pair of modified statements by whether they fit a set of mutation patterns. The resulting dataset is shown in Table 5.6.

Finally, we queried the DeepBugs and SCELMo with each buggy instance as well as its fixed variant and measured the percentage of correctly classified instances for each of the two categories. We also ignored any instances for which the JavaScript parser utilized for both failed to extract an AST. We classified as bugs any instances that were assigned a probability to be a bug $> 75\%$. In an actual system this threshold should ideally be tuned on a validation set.

Table 5.7 suggests that there might indeed be some potential for future practical applications of neural bug finding techniques. Both are able to uncover some of the bugs. However, the results also suggest that careful tuning of the predictions threshold might be necessary, especially if we take into account the industrial need to comply with a low false positive rate (FPR). For instance, raising SCELMo's prediction threshold to 80% for the swap arguments bug results in finding only 3.34% of the bugs but correctly classifying 100% of the repaired function calls, thus achieving 0.0% false positive rate.

[5]https://github.com/mozilla/rhino

Moreover, since SCELMo could not uncover any of the real binary operator bugs, future work could investigate the effect of utilizing different mutation strategies for the purpose of artificial bug-induction. Future work could also investigate if fine-tuning on small set of real bugs could result in more robust classifiers.

5.7 Conclusion

We have presented SCELMo, which is to our knowledge the first language-model based embeddings for source code. Contextual embeddings have many potential advantages for source code, because surrounding tokens can indirectly provide information about tokens, e.g., about likely values of variables. We highlight the utility of SCELMo embeddings by using them within a recent state-of-the-art machine learning based bug detector. The SCELMo embeddings yield a dramatic improvement in bug detection performance, especially on lines of code that contain out-of-vocabulary tokens and complex expressions that can cause difficulty for the method. Our method focuses on previous and following tokens to extract contextual information. However, there a lot of other kinds of contextual information present in code that we could utilize. Some examples include but are not limited to using data or control dependencies or utilizing class hierarchy information. Graph or transformer models would be a great starting point for the implementations of such models. We think that this would be an excellent direction for future work that holds an extreme amount of potential.

Chapter 6

Summary

"Obstacles don't have to stop you. If you run into a wall, don't turn around and give up. Figure out how to climb it, go through it, or work around it."
–Michael Jordan

Automatic bug detection (ABD) and by extension automatic program repair (APR) have enabled us to automate part of the process of discovering and fixing bugs. Such tools can locate and/or repair software bugs either without or with minimal intervention of a human programmer (Gazzola et al., 2019; Rinard, 2008). As bugs cost the global economy more and more every year with the cost having reached trillions of US dollars, ABD and APR are a promising approach for significantly reducing this cost by detecting errors early in the software's development cycle and shipping programs with less errors. To do so, these methods utilize an oracle (e.g., a test suite) that performs fault localization. Most fault localization techniques are based on the idea of causality (Lewis, 1973; Pearl, 2000) and order the suspected statements based on their possible responsibility for causing the fault. Coupled with fault localization is patch generation and validation. Generate-and-validate techniques aim to minimize the tax of these two processes on developers. These techniques generate fix candidates by applying either pre-defined or learned operators on the original code and optionally other candidates. The candidates are usually ranked and the search may generate one or more plausible patches that may or may not fix the defect. Consequently, these patches still need to be validated by developers. Moreover, some techniques focus on generic fault detection and fixing while others tackle only specific classes of bugs. Finally, some approaches attempt to extract a formal representation that encodes the original problem e.g., with a formula. Such methods attempt to solve the extracted formal representation. For any solutions found they generate the code changes required using program synthesis (Gulwani et al., 2017b).

Recently, there have been incredible advances in the fields of machine learning and natural language processing mainly based on deep learning models like neural language models. As researches attempted to introduce, apply, and modify these techniques the area of machine learning for software engineering was born. This area of research is an intersection of machine learning, natural language processing, software engineering and programming languages research (Allamanis et al., 2018a; Amershi et al., 2019). Researchers have produced many models that have shown promising results and great future potential. By entering the software engineering domain these methods also entered the fields of BD and by extension APR.

However, applying machine learning techniques on source code corpora introduces new problems that must be overcome. A major issue is that code introduces new vocabulary at a far higher rate than natural language, as new identifier names proliferate. Both large vocabularies and out-of-vocabulary issues severely affect Neural Language Models (NLMs) of source code, degrading their performance and rendering them unable to scale. Chapter 3 concerned with this exact problem and proposed an effective solution for the problem based on learning the least entropic segmentation into subwords using the byte pair encoding algorithm. It first provided empirical evidence of why current options for modelling the vocabulary are inadequate and that our approach is a much better fit for the problem. Based on this, it described and showcased the first open-vocabulary neural language model for source code. Our approach was able to scale on source-code corpora at least 100 times larger than previous work. Extensive evaluation on three code corpora (Java, C, Python) showed that our open-vocabulary neural language model outperforms the state-of-the-art language models for source code, while also requiring significantly less resources. The evaluation was based on measuring the entropy of correct code on a test set, code completion performance on unseen tokens as well as unseen identifiers, and on highlighting buggy code. Moreover, it introduces a beam search like decoder that generates token predictions and a procedure for quick dynamic adaptation to new projects that allow effective real time predictions of tokens. It also introduces a very simple cache mechanism for identifiers that greatly enhances the model's performance on them. Additionally, the raw and preprocessed datasets utilized as well as the learned models were released to the community. Finally, the released models could be of great value to the software engineering community as they can also be used as upstream tasks in transfer learning, possibly leading to state-of-the-art improvement in downstream tasks.

Fixing bugs early in the development cycle is very essential, especially in an industrial setting. Large projects have possibly hundreds of bugs that need to be fixed every day and hundred others that remain undetected. To address this problem, ABD can be utilized to propose suspicious code to developers. To get the most of these tools the reports should be bugs with extremely high probabilty and should be bugs that are hard to manually spot. Chapter 4 is concerned with this exact matter. It introduces a new class of bugs that are mainly semantic errors for which the code compiles both before and after the fix was applied. These fixes for these bugs are small (i.e., only a single statements needs to be modified) yet they might be quite tedious to manually spot. Moreover, the unique characteristic is that they are annotated by whether they match any of a set of 16 bug templates, inspired by state-of-the-art program repair techniques. Due to how developers feel upon discovering such a bug we refer to these bugs as simple stupid bugs (SStuBs) The chapter continues by introducing the SStuB 16 patterns along with a methodology for mining them. Using this methodology a Java version of the dataset was mined, viz., the ManySStuBs4J dataset. In addition, it posed and answered important research questions. The first was whether SStuBs actually appear often in open-source code. This was measured by defining two densities measures and indeed SStuBs were shown to appear with high frequency. The next question was whether they could simply be spotted by existing tools such as static analyzers. However, it was shown that a popular static analysis tool (SpotBugs) could only locate about 12% of SStuBs while also reporting more than 200 million possible bugs and giving credit to the tool for any warning referring to the line(s) containing the fault instead of reporting the actual exact fault. If we consider that no developer would look through thousands of warnings to spot just one SStuB, it is highlighted how important it is to build SStuB specific localization tools. Another question posed by the chapter was whether these bugs satisfy the plastic surgery hypothesis, thus the fixes already exist in the buggy version of the code. Indeed almost 59% of them already exist, suggesting that this is an aspect that definitely should not be ignored by future research.

Both this thesis and the current literature have utilized continuous embeddings of tokens in computer programs, which have been used to support a variety of software development tools. But until now researchers have only utilized static embeddings to build such tools. However, during the last two years the field of natural language processing has seen significant advances in state-of-the-art performance by utilizing pre-trained contextual embeddings. This should not come as a complete surprise as the dynamic adaptation procedure in Chapter 3 may perform small modifications to the embeddings of its vocabulary elements. As such with a grain of salt it could be considered as the first usage of non-static embeddings in this domain. However, it did not capture contextual information. Chapter 5 discussed reasons why contextual embeddings would be a good fit for source code. It introduces a new set of deep contextualized word representations for computer programs. Our model was named Source Code Embeddings from Language Models (SCELMo) and is built upon the ELMo framework (Peters et al., 2018). We evaluated the learned embeddings on a JavaScript bug detection task based on the DeepBugs (Pradel and Sen, 2018) framework but also introduced an external test set, improvements to the evaluation procedure and expanded the name extraction heuristic. Even with low-dimensional embeddings trained on a relatively small corpus of programs, the presented methodology outperformed both static Word2Vec (Mikolov et al., 2013b) embeddings and FastText embeddings that contain subword information. As building entirely new models for every new problem would be a hassle, the chapter proposed that we could instead utilize transfer learning where we transfer the pre-trained embeddings to the problem at hand. This would allow to overcome the lack of supervised data for problems where only a few training instances are available. Another concern was that although there has been prior work in named based bug detection, the evaluation utilized only synthetic data. For this reason, we mined a small dataset based on the methodology of Chapter 4 and made the first steps for answering whether named based bug detection is useful in practice. Moreover, we note that simultaneously to this work other researchers focused on applying them on a similar problem (Kanade et al., 2020) and on making a bimodal model programming and natural languages (Feng et al., 2020). The effectiveness of these models offers extra confirmation about the validity of our results and that contextual embeddings were indeed a good fit for source code. Last, we clarify that our approach differed from theirs for the following reasons: 1) It is based on bidirectional LSTMs while theirs uses transformers; 2) and ours is a low-resource model that requires maximum 2 or 3 GPUs to be trained while theirs requires TPUs and thus ours is a much better fit for academia.

6.1 Applicability of the Techniques in an Industrial Setting

The techniques proposed in this thesis are not limited to an academic setting and may easily adapted for industrial usage. We next discuss how this could be achieved and what resources are required for each of them in order to achieve this objective.

The open-vocabulary neural language models presented in Chapter 3 are all publicly available for all three languages along with both raw and pre-processed versions of the corpora that generated them. These models are small in size and only an appropriate middleware that incorporates them either locally or in a server needs to be developed.

This would generate queries for predictions or retrieve representations from the model. However, a reasonable question is what needs to be done in order to learn an equivalent model for a new programming language. To achieve this we would first need enough source code. This an easy problem to solve as we can utilize the exact same methodology presented in this thesis. That is to mine open-source projects from GitHub. Alternative a big company might have access to its large codebase which could either be used on its own or paired with the open-source projects. A tokenizer for the language is also essential to tokenize the data and create the training files but such a tool is trivial to find for any programming language. The training scripts are all publicly available and similar software would be easy to develop as all the code used is documented and open-source. The GPU resources required are minimal and the training process duration depends on the training corpus size. We note that for very large corpora, training could last at least a couple of weeks unless the model is parallelized. Last, to focus on specific token type such as identifiers or numeric literals a parser for the programming language of interest is required but such a tool is trivial to find and use in the industry.

The tool for mining SStuBs showcased in Chapter 4 is open-source and publicly available. It uses maven so it is straightforward to build and run for Java code. The only requirement for it to operate is to have a corpus of GitHub Java projects downloaded inside a folder. The tool needs to be pointed to that folder as well as some other folder that will store the dataset. As such it is trivial to generate a new SStuB dataset on a different corpus of Java projects. Adding a new SStuB pattern is not very hard either. The most essential component would be to define a function that takes as inputs two appropriate AST nodes and checks whether the pattern is satisfied or not. However, we also need to explain what needs to be done in order to mine SStuBs for a new language. Obviously, we would need a corpus of projects for that language and most large companies will probably have their own code base, and even if that is not the case for a company then that is easy to address by mining open-source projects. The core methodology for selecting bug commits remains the same with minimal changes. We would also need to define what the SStuB patterns would be used for that language as not all of those used for Java might be suitable and essential ones could be missing. This might require some effort and careful thinking. In order to implement the patterns a parser is necessary so that can get essential information for AST node types etcetera. The dataset provided or any new mined datasets could be used to facilitate software engineering tasks. A trivial use case would be to evaluate the effectiveness of bug detection and program repair tools for simple bugs. However, in an industrial setting we could use it for other purposes. For instance, we could utilize the Change Identifier instances as examples to improve code completion.

Finally, although SCELMo is a low resource model tailored for easy training in an academic setting, nothing really prohibits it from being used in the industry. To train a new model we only need a corpus for that specific language and a parser to create training data, which as discussed before are trivial to obtain. As the resources required are only a few GPUs it is very easy to satisfy them. In order to use the model to get representations for other models we would need to develop an appropriate but simple middleware that incorporates it either locally or in a server. This will generate and queries and retrieve representations from the model. In order to train a bug detector for a new language we would need a parser and to implement the mutation operators for each bug pattern. Then, we can easily just query SCELMo for contextual representations as nothing would need change for the core training loop and methodology. Last, we note though that the SCELMo model could be utilized to provide contextual representations for any supervised model operating over source code and not just for bug detectors.

6.2 Future Work

The questions posed by this thesis along with the experimental results have shed some new light into effectively applying deep learning models in the software engineering domain in a scalable and efficient manner and many of their contributions focused on BD. However, they were not able to answer every possible question in this domain and have also revealed some new exciting questions and directions for future work.

In Chapter 3 we introduced the first open-vocabulary neural language model for source code. While multiple segmentations are possible even with the same vocabulary, BPE splits words into unique sequences. Consequently, this may hinder the model from learning the compositionality of tokens and more importantly it results to a model that is not robust to segmentation errors. This problem has also appeared in the domain of natural language processing Kudo (2018). BPE-dropout is a promising good fit solution (Provilkov et al., 2019). Extending the training procedure to use BPE-dropout could result in possible improvements and a more robust model. Another exciting idea is to give more focus on other transfer learning techniques. The effectiveness of the dynamic adaptation algorithm indicates that there is great potential for adapting neural language models of code to new problems with only a few instances. A popular such technique for transfer learning technique is MAML (Finn et al., 2017). In fact, just a few days before completing the authorship of this thesis, the first work on applying MAML on source code has appeared (Shrivastava et al., 2020). Another possible direction would be to explore methods for incorporating pre-training information from external domains as proposed only a handful of days ago (Xu et al., 2020).

Chapter 4 introduced SStuBs along with the ManySStuBs4j dataset but also opened new questions. ManySStuBs4j could be used to answer other research questions, such as empirical questions about how and when simple bugs are introduced. It could also be used to evaluate bug detection and program repair techniques for small bugs. Another question regarding both Chapters 3 and 4 is since the open-vocabulary neural language model is quite effective at highlighting buggy code, could we take advantage of this to build a fault localization system for small bugs or even to score possible fixes for such bugs? Since most SStuBs seem to be graftable from existing code, could we maybe utilize this to our advantage to build effective SStuB specific BD and APR systems?

The contextual embeddings and SCELMo model presented in Chapter 5 also open many avenues for future work. Combining SCELMo with BPE processing is a straightforward direction to explore, which potentially could improve the model and speed up training. Moreover, the ManySStuBs4j dataset could be used to perform a more thorough evaluation of a Java SCELMo model, but also possibly as a benchmark between different methods. The dataset will be used as the task of the mining challenge track of the mining software repositories (MSR) 2021 conference (Karampatsis et al., 2021). Another alternative would be to study the effectiveness of improvements suggested in the literature (Liu et al., 2019) over the BERT models in this new domain or explore methods that aim on lowering the resources required by BERT such as ALBERT (Lan et al., 2019). Moreover, future work could explain the effectiveness of contextual embeddings in other software engineering problems. An example problem that might be a great fit is type inference for dynamic languages (Pandi et al., 2020). Another direction would be to study whether contextual embeddings do indeed capture source code specific phenomena like the evolution of a variable's value. Although we introduced the question of whether neural bug finding is practically useful we only did some first baby steps on answering it. Thus future work could focus on providing a satisfying answer to this question.

An exciting decade of research lies ahead! We'll see you on the other side.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. (page 58)

Abramson, D., Foster, I., Michalakes, J., and Sosic, R. (1995). Relative debugging and its application to the development of large numerical models. In *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pages 51–51. (page 23)

Abreu, R., Zoeteweij, P., and Gemund, A. J. C. v. (2006). An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, page 39–46, USA. IEEE Computer Society. (page 22)

Ackling, T., Alexander, B., and Grunert, I. (2011). Evolving patches for software repair. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, page 1427–1434, New York, NY, USA. Association for Computing Machinery. (pages 26 and 29)

Agarwal, P. and Agrawal, A. P. (2014). Fault-localization techniques for software systems: A literature review. *SIGSOFT Softw. Eng. Notes*, 39(5):1–8. (page 20)

Agrawal, H., Alberi, J. L., Horgan, J. R., Li, J. J., London, S., Wong, W. E., Ghosh, S., and Wilde, N. (1998). Mining system tests to aid software maintenance. *Computer*, 31(7):64–73. (page 20)

Agrawal, H., DeMillo, R. A., and Spafford, E. H. (1991). An execution-backtracking approach to debugging. *IEEE Softw.*, 8(3):21–26. (page 22)

Agrawal, H., Demillo, R. A., and Spafford, E. H. (1993). Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616. (page 19)

Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256. (page 19)

Agrawal, H., Horgan, J. R., London, S., and Wong, W. E. (1995). Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151. (pages 19 and 20)

Al-Khanjari, Z. A., Woodward, M. R., Ramadhan, H. A., and Kutti, N. S. (2005). The efficiency of critical slicing in fault localization. *Software Quality Journal*, 13(2):129–153. (page 19)

Alkhalaf, M., Aydin, A., and Bultan, T. (2014). Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 225–236, New York, NY, USA. Association for Computing Machinery. (page 33)

Allamanis, M. (2017). *Learning natural coding conventions*. PhD thesis. (pages 4 and 44)

Allamanis, M., Barr, E. T., Bird, C., Devanbu, P., Marron, M., and Sutton, C. (2018). Mining semantic loop idioms. *IEEE Transactions on Software Engineering*, 44(7):651–668. (page 36)

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2014). Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, New York, NY, USA. ACM. (pages 36, 40, 41, 45, and 74)

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2015a). Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA. ACM. (pages 36, 39, 45, 50, 51, 64, 74, and 79)

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018a). A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37. (pages 4, 6, 34, 37, 40, 70, 74, 99, 102, 104, and 118)

Allamanis, M. and Brockschmidt, M. (2017). Smartpaste: Learning to adapt source code. (page 36)

Allamanis, M., Brockschmidt, M., and Khademi, M. (2018b). Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations.* (page 43)

Allamanis, M., Brockschmidt, M., and Khademi, M. (2018c). Learning to represent programs with graphs. In *International Conference on Learning Representations.* (pages 102 and 114)

Allamanis, M., Peng, H., and Sutton, C. A. (2016). A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001. (pages 36, 39, 44, 45, 46, 74, and 79)

Allamanis, M. and Sutton, C. (2013). Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA. IEEE Press. (pages 45, 46, 47, and 72)

Allamanis, M. and Sutton, C. (2014). Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA. ACM. (page 36)

Allamanis, M., Tarlow, D., Gordon, A., and Wei, Y. (2015b). Bimodal modelling of source code and natural language. In Blei, D. and Bach, F., editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2123–2132. JMLR Workshop and Conference Proceedings. (page 62)

Alon, U., Brody, S., Levy, O., and Yahav, E. (2019a). code2seq: Generating sequences from structured representations of code. In *Proceedings of ICLR 2019*. (page 46)

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019b). Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29. (pages 43 and 102)

Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. (page 12)

Alves, E., Gligoric, M., Jagannath, V., and d'Amorim, M. (2011). Fault-localization using dynamic slicing and change impact analysis. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 520–523, Washington, DC, USA. IEEE Computer Society. (page 19)

Amann, S., Proksch, S., Nadi, S., and Mezini, M. (2016). A study of visual studio usage in practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134. (page 35)

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., and Zimmermann, T. (2019). Software engineering for machine learning: A case study. In *International Conference on Software Engineering (ICSE 2019) - Software Engineering in Practice track*. IEEE Computer Society. (pages 4 and 118)

Andreessen, M. (2011). Why software is eating the world. *The Wall Street Journal*. (page 1)

Arcuri, A. (2008). On the automation of fixing software bugs. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 1003–1006, New York, NY, USA. ACM. (pages 12, 26, 27, and 29)

Arcuri, A. (2011). Evolutionary repair of faulty software. *Appl. Soft Comput.*, 11(4):3494–3514. (pages 27 and 29)

Arcuri, A. and Xin Yao (2008). A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 162–168. (pages 12 and 95)

Arcuri, A. and Yao, X. (2007). Coevolving programs and unit tests from their specification. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 397–400, New York, NY, USA. ACM. (page 12)

Ascari, L. C., Araki, L. Y., Pozo, A. R. T., and Vergilio, S. R. (2009). Exploring machine learning techniques for fault localization. In *2009 10th Latin American Test Workshop*, pages 1–6. (page 25)

Assiri, F. Y. and Bieman, J. M. (2014). An assessment of the quality of automated program operator repair. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, page 273–282, USA. IEEE Computer Society. (page 26)

Assiri, F. Y. and Bieman, J. M. (2018). Mut-apr: Mutation-based automated program repair research tool. In Arai, K., Kapoor, S., and Bhatia, R., editors, *Advances in Information and Communication Networks*, pages 256–270, Cham. Springer International Publishing. (page 29)

Avizienis, A., Laprie, J. ., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33. (page 11)

Azcona, D., Arora, P., Hsiao, I.-H., and Smeaton, A. (2019). User2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, LAK19, page 86–95, New York, NY, USA. Association for Computing Machinery. (page 35)

Babii, H., Janes, A., and Robbes, R. (2019). Modeling vocabulary for big code machine learning. (pages 9, 39, and 48)

Ball, T. and Larus, J. R. (1994). Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360. (page 17)

Barr, E. T., Brun, Y., Devanbu, P., Harman, M., and Sarro, F. (2014). The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 306–317, New York, NY, USA. Association for Computing Machinery. (pages 6, 27, 78, 95, and 96)

Bavishi, R., Pradel, M., and Sen, K. (2018). Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *CoRR*, abs/1809.05193. (pages 36, 39, 45, 74, and 104)

Bazzi, I. (2002). *Modelling Out-of-vocabulary Words for Robust Speech Recognition*. PhD thesis, Cambridge, MA, USA. AAI0804528. (page 44)

Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155. (page 42)

Bhatia, S. and Singh, R. (2016). Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129. (pages 34 and 74)

Bielik, P., ETHZ, I., Raychev, V., and Vechev, M. (2016). PHOG: Probabilistic model for code. *Proceedings of the 33rd International Conference on Machine Learning (ICML-16)*. (pages 36, 44, and 64)

Biermann, A. W. (1985). Automatic programming : A tutorial on formal methodologies. *Journal of Symbolic Computation*, 1(2):119 – 142. (page 12)

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA. (page 1)

Binkley, D., Davis, M., Lawrie, D., and Morrell, C. (2009). To camelcase or under_score. In *2009 IEEE 17th International Conference on Program Comprehension (ICPC 2009)*, pages 158–167. IEEE. (page 50)

Binkley, D. and Harman, M. (2003). A survey of empirical results on program slicing. In *ADVANCES IN COMPUTERS, 62:105–178*, pages 105–178. (page 18)

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146. (pages 44, 102, and 107)

Bradbury, J., Merity, S., Xiong, C., and Socher, R. (2016). Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*. (pages 43 and 74)

Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA. ACM. (pages 35 and 63)

Bunel, R., Hausknecht, M. J., Devlin, J., Singh, R., and Kohli, P. (2018). Leveraging grammar and reinforcement learning for neural program synthesis. *CoRR*, abs/1805.04276. (page 62)

Buxton, J. N. and Randell, B. (1970). *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO.* (page 11)

Cai, Y. and Cao, L. (2016). Fixing deadlocks via lock pre-acquisitions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1109–1120, New York, NY, USA. Association for Computing Machinery. (page 33)

Campbell, J. C., Hindle, A., and Amaral, J. N. (2014). Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 252–261, New York, NY, USA. ACM. (pages 34, 45, and 74)

Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., and Pezzè, M. (2013). Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 782–791, Piscataway, NJ, USA. IEEE Press. (page 13)

Carzaniga, A., Gorla, A., Perino, N., and Pezzè, M. (2015). Automatic workarounds: Exploiting the intrinsic redundancy of web applications. *ACM Trans. Softw. Eng. Methodol.*, 24(3):16:1–16:42. (page 13)

Casalnuovo, C., Sagae, K., and Devanbu, P. (2018). Studying the difference between natural and programming language corpora. *Empirical Software Engineering*, pages 1–46. (page 44)

Chang, H., Mariani, L., and Pezzè, M. (2013). Exception handlers for healing component-based systems. *ACM Trans. Softw. Eng. Methodol.*, 22:30:1–30:40. (page 15)

Chen, S. F. and Goodman, J. (1999). An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394. (page 42)

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L., Poshyvanyk, D., and Monperrus, M. (2019a). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *CoRR*, abs/1901.01808. (pages 35, 37, and 102)

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L., Poshyvanyk, D., and Monperrus, M. (2019b). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *CoRR*, abs/1901.01808. (pages 77 and 78)

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. (2018). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *arXiv preprint arXiv:1901.01808.* (pages 40, 41, and 46)

Chen, Z. and Monperrus, M. (2018a). The codrep machine learning on source code competition. *CoRR*, abs/1807.03200. (page 78)

Chen, Z. and Monperrus, M. (2018b). The remarkable role of similarity in redundancy-based program repair. *CoRR*, abs/1811.05703. (pages 35 and 45)

Chen, Z. and Monperrus, M. (2019). A literature study of embeddings on source code. *CoRR*, abs/1904.03061. (page 102)

Chioteli, E., Batas, I., and Spinellis, D. (2019). Does unit-tested code crash? a case study of eclipse. *arXiv preprint arXiv:1903.04055.* (page 94)

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078. (page 57)

Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555. (page 57)

Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 342–351, New York, NY, USA. Association for Computing Machinery. (pages 22 and 25)

Colin, S. and Mariani, L. (2004). 18 run-time verification. pages 525–555. (page 14)

Cooper, G. F. and Herskovits, E. (1992). A bayesian method for the induction of probabilistic networks from data. *Mach. Learn.*, 9(4):309–347. (page 35)

Corazza, A., Di Martino, S., and Maggio, V. (2012). Linsen: An efficient approach to split identifiers and expand abbreviations. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 233–242. IEEE. (page 52)

Coutant, D. S., Meloy, S., and Ruscetta, M. (1988). Doc: A practical approach to source-level debugging of globally optimized code. *SIGPLAN Not.*, 23(7):125–134. (page 16)

Creutz, M., Hirsimäki, T., Kurimo, M., Puurula, A., Pylkkönen, J., Siivola, V., Varjokallio, M., Arisoy, E., Saraçlar, M., and Stolcke, A. (2007). Morph-based speech recognition and modeling of out-of-vocabulary words across languages. *ACM Transactions on Speech and Language Processing (TSLP)*, 5(1):3. (pages 42 and 44)

da Silva Maldonado, E., Shihab, E., and Tsantalis, N. (2017). Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062. (page 50)

Dallmeier, V., Zeller, A., and Meyer, B. (2009). Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, page 550–554, USA. IEEE Computer Society. (page 30)

Dam, H. K., Tran, T., and Pham, T. (2016). A deep language model for software code. *arXiv preprint arXiv:1608.02715.* (pages 44 and 46)

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection.* Murray, London. or the Preservation of Favored Races in the Struggle for Life. (page 28)

Dawson, M., Burrell, D., Rahim, E., and Brewster, S. (2010). Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3:49–53. (pages 4 and 75)

Debroy, V. and Wong, W. E. (2010). Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, page 65–74, USA. IEEE Computer Society. (page 30)

DeMarco, F., Xuan, J., Le Berre, D., and Monperrus, M. (2014). Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, page 30–39, New York, NY, USA. Association for Computing Machinery. (page 33)

DeMillo, R. A., Pan, H., and Spafford, E. H. (1996). Critical slicing for software fault localization. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 121–134, New York, NY, USA. ACM. (page 19)

Demsky, B. and Rinard, M. (2003). Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, page 78–95, New York, NY, USA. Association for Computing Machinery. (page 4)

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., and Roy, S. (2016). Program synthesis using natural language. *CoRR*, abs/1509.00413. (page 62)

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805. (pages 5, 7, 41, 74, 99, 101, and 102)

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., and Kohli, P. (2017a). Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469. (page 62)

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. (2017b). Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 990–998. JMLR.org. (page 37)

Devlin, J., Uesato, J., Singh, R., and Kohli, P. (2017c). Semantic code repair using neuro-symbolic transformation networks. *CoRR*, abs/1710.11054. (page 35)

Ding, R., Fu, Q., Lou, J.-G., Lin, Q., Zhang, D., Shen, J., and Xie, T. (2012). Healing online service systems via mining historical issue repositories. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 318–321, New York, NY, USA. ACM. (page 13)

Dudoladov, S. (2013). Statistical NLP for computer program source code: An information theoretic perspective on programming lanuguage verbosity. Master's thesis, School of Informatics, University of Edinburgh, United Kingdom. (pages 47 and 72)

Durieux, T. and Monperrus, M. (2016). Dynamoth: Dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, page 85–91, New York, NY, USA. Association for Computing Machinery. (page 33)

Edwards, Jermaine Charles (Cedar Park, T. (2003). Method, system, and program for logging statements to monitor execution of a program. (page 16)

Enslen, E., Hill, E., Pollock, L., and Vijay-Shanker, K. (2009). Mining source code to automatically split identifiers for software analysis. (pages 46 and 52)

Eric Wong, W., Debroy, V., and Choi, B. (2010). A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.*, 83(2):188–208. (page 21)

Fausett, L. (1994). *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice-Hall, Inc., USA. (page 25)

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. (pages 5, 102, and 120)

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1126–1135. JMLR.org. (page 122)

Fiott, S. (2015). An Investigation of Statistical Language Modelling of Different Programming Language Types Using Large Corpora. Master's thesis, School of Informatics, University of Edinburgh, United Kingdom. (pages 47 and 72)

Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. (2009). A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954, New York, NY, USA. ACM. (pages 12, 26, and 27)

Franks, C., Tu, Z., Devanbu, P. T., and Hellendoorn, V. (2015). CACHECA: A cache language model based code suggestion tool. In Bertolino, A., Canfora, G., and Elbaum, S. G., editors, *ICSE (2)*, pages 705–708. IEEE Computer Society. ISBN 978-1-4799-1934-5 (Vol. I + II ???). (pages 35, 44, and 62)

Gabel, M. and Su, Z. (2010). A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA. ACM. (page 44)

Gage, P. (1994). A new algorithm for data compression. *C Users J.*, 12(2):23–38. (pages 5, 41, 52, and 53)

Gao, Q., Xiong, Y., Mi, Y., Zhang, L., Yang, W., Zhou, Z., Xie, B., and Mei, H. (2015). Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 459–470. IEEE Press. (page 33)

Gaunt, A. L., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., and Tarlow, D. (2016). Terpret: A probabilistic programming language for program induction. (page 37)

Gazzola, L., Micucci, D., and Mariani, L. (2019). Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(01):34–67. (pages 11, 13, 15, 102, and 117)

Gen, M. and Cheng, R. (1999). *Genetic Algorithms*. John Wiley & Sons, Inc., USA, 1st edition. (page 28)

Ghosh, D., Sharman, R., Rao, H. R., and Upadhyaya, S. (2007). Self-healing systems — survey and synthesis. *Decision Support Systems*, 42(4):2164 – 2185. Decision Support Systems in Emerging Economies. (pages 12 and 13)

Godefroid, P., Peleg, H., and Singh, R. (2017). Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 50–59. IEEE Press. (page 34)

Gong, C., He, D., Tan, X., Qin, T., Wang, L., and Liu, T. (2018). FRAGE: frequency-agnostic word representation. In *Proceedings of NeurIPS 2018*, pages 1341–1352. (pages 43 and 48)

Gordon, L., Loeb, M., Lucyshyn, W., and Richardson, R. (2000). Csi/fbi computer crime and security survey. *Computer Security Institute*, 22. (page 2)

Goues, C. L., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., and Weimer, W. (2015). The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256. (page 78)

Gousios, G. (2013). The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA. IEEE Press. (page 79)

Grave, E., Joulin, A., Cissé, M., Grangier, D., and Jégou, H. (2016). Efficient softmax approximation for gpus. *CoRR*, abs/1609.04309. (page 44)

Gu, X., Zhang, H., Zhang, D., and Kim, S. (2016). Deep API learning. *CoRR*, abs/1605.08535. (page 45)

Gu, X., Zhang, H., Zhang, D., and Kim, S. (2017). Deepam: Migrate apis with multi-modal sequence to sequence learning. *arXiv preprint arXiv:1704.07734*. (pages 40 and 45)

Gulwani, S., Polozov, O., and Singh, R. (2017a). Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119. (page 37)

Gulwani, S., Polozov, O., Singh, R., et al. (2017b). Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119. (pages 32, 33, 37, 74, and 117)

Guo, X., Zhou, M., Song, X., Gu, M., and Sun, J. (2015). First, debug the test oracle. *IEEE Transactions on Software Engineering*, 41(10):986–1000. (page 93)

Gupta, R., Kanade, A., and Shevade, S. (2018). Deep reinforcement learning for programming language correction. (page 35)

Gupta, R., Pal, S., Kanade, A., and Shevade, S. (2017). DeepFix: Fixing common c language errors by deep learning. In *National Conference on Artificial Intelligence (AAAI)*. (pages 35 and 74)

Gvero, T. and Kuncak, V. (2015). Synthesizing java expressions from free-form queries. *SIGPLAN Not.*, 50(10):416–432. (page 37)

Gyimóthy, T., Beszédes, A., and Forgács, I. (1999). An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 303–321, Berlin, Heidelberg. Springer-Verlag. (page 19)

Habib, A. and Pradel, M. (2018). How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 317–328, New York, NY, USA. ACM. (pages 95 and 114)

Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A., Hamilton, L. H., Centeno, G. I., Key, J. R., Ellingwood, P. M., McConley, M. W., Opper, J. M., Chin, S. P., and Lazovich, T. (2018). Automated software vulnerability detection with machine learning. *CoRR*, abs/1803.04497. (pages 7, 99, and 102)

Harrold, M. J., Rothermel, G., Wu, R., and Yi, L. (1998). An empirical investigation of program spectra. *SIGPLAN Not.*, 33(7):83–90. (page 20)

Hauswirth, M. and Chilimbi, T. M. (2004). Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 156–164, New York, NY, USA. ACM. (page 17)

Hellendoorn, V. J. and Devanbu, P. (2017). Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 763–773, New York, NY, USA. ACM. (pages 40, 42, 44, 45, 46, 47, 48, 49, 50, 54, 63, 64, 65, 66, 68, 71, 72, and 74)

Hellendoorn, V. J., Proksch, S., Gall, H. C., and Bacchelli, A. (2019). When code completion fails: a case study on real-world completions. In *Proceedings of the 41st International Conference on Software Engineering*, pages 960–970. IEEE Press. (pages 64 and 66)

Hennessy, J. (1982). Symbolic debugging of optimized code. *ACM Trans. Program. Lang. Syst.*, 4(3):323–344. (page 16)

Herzig, K. and Zeller, A. (2013). The impact of tangled code changes. In *Working Conference on Mining Software Repositories*, pages 121–130. IEEE Press. (page 81)

Hill, E., Binkley, D., Lawrie, D., Pollock, L., and Vijay-Shanker, K. (2014). An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780. (page 46)

Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. (2012). On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA. IEEE Press. (pages 34, 35, 40, 44, 46, 61, 64, 65, 111, and 114)

Hinton, G. E., McClelland, J. L., and Rumelhart, D. E. (1986). *Distributed Representations*, page 77–109. MIT Press, Cambridge, MA, USA. (page 34)

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780. (pages 42, 57, and 74)

Howard, J. and Ruder, S. (2018). Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 328–339. (pages 41 and 73)

Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. (2018). Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 200–210, New York, NY, USA. ACM. (pages 39, 45, and 74)

Hucka, M. (2018). Spiral: splitters for identifiers in source code files. *J. Open Source Software*, 3(24):653. (page 52)

Hunt, A. and Thomas, D. (2000). *The Pragmatic programmer : from journeyman to master.* Addison-Wesley, Boston [etc.]. (page 75)

Iglesia, D. G. D. L. and Weyns, D. (2015). MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.*, 10(3):15:1–15:31. (page 13)

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics. (pages 36, 39, and 74)

Jacob, B., Lanyon, R., Devaprasad, H., Nadgir, K., and Yassin, A. (2004). *A Practical Guide to the IBM Autonomic Computing Toolkit*. IBM redbooks. IBM, International Support Organization. (page 14)

Jaffe, A., Lacomis, J., Schwartz, E. J., Le Goues, C., and Vasilescu, B. (2018). Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of ICPC 2018*, pages 20–30. (page 45)

Jean, S., Cho, K., Memisevic, R., and Bengio, Y. (2015). On using very large target vocabulary for neural machine translation. In *Proceedings of ACL 2015*, pages 1–10. (pages 42 and 43)

Ji, T., Chen, L., Mao, X., and Yi, X. (2016). Automated program repair by using similar code containing fix ingredients. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 197–202. (page 29)

Jin, G., Song, L., Zhang, W., Lu, S., and Liblit, B. (2011). Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 389–400, New York, NY, USA. Association for Computing Machinery. (page 33)

Jin, G., Zhang, W., and Deng, D. (2012). Automated concurrency-bug fixing. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 221–236, Hollywood, CA. USENIX. (page 33)

Jones, J. A. and Harrold, M. J. (2005). Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA. ACM. (pages 21 and 22)

Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA. ACM. (page 20)

Jones, J. A., Harrold, M. J., and Stasko, J. T. (2001). Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*, pages 71–75. (pages 20 and 22)

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*. (page 43)

Ju, X., Jiang, S., Chen, X., Wang, X., Zhang, Y., and Cao, H. (2014). Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software*, 90:3 – 17. (page 19)

Just, R., Jalali, D., and Ernst, M. D. (2014a). Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM. (pages 29 and 71)

Just, R., Jalali, D., and Ernst, M. D. (2014b). Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA. ACM. (pages 78 and 114)

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. (2020). Pre-trained contextual embedding of source code. (pages 5, 102, and 120)

Karaivanov, S., Raychev, V., and Vechev, M. (2014). Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 173–184, New York, NY, USA. ACM. (pages 36, 62, and 74)

Karampatsis, R.-M., Allamanis, M., and Sutton, C. (2021). Msr mining challenge: The life of simple, stupid bugs (sstubs). In *Proceedings of the International Conference on Mining Software Repositories (MSR 2021).* (pages 98, 101, and 123)

Karampatsis, R. M., Babii, H., Robbes, R., Sutton, C., and Janes, A. (2020). Big code != big vocabulary: Open-vocabulary models for source code. In *to appear in Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20. ACM. (pages 8, 9, 39, 72, and 98)

Karampatsis, R.-M. and Sutton, C. (2019). Maybe deep neural networks are the best choice for modeling source code. (pages 9 and 39)

Karampatsis, R. M. and Sutton, C. (2020a). How often do single-statement bugs occur? the manysstubs4j dataset. In *to appear in Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20. ACM. (pages 8, 9, 77, 114, and 115)

Karampatsis, R. M. and Sutton, C. (2020b). Scelmo: Source code embeddings from language models. (pages 5, 8, 9, and 100)

Ke, Y., Stolee, K. T., Goues, C. L., and Brun, Y. (2015). Repairing programs with semantic code search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 295–306. IEEE Press. (page 33)

Kelk, D., Jalbert, K., and Bradbury, J. S. (2013). Automatically repairing concurrency bugs with arc. In Lourenço, J. M. and Farchi, E., editors, *Multicore Software Engineering, Performance, and Tools*, pages 73–84, Berlin, Heidelberg. Springer Berlin Heidelberg. (page 26)

Khandelwal, U., He, H., Qi, P., and Jurafsky, D. (2018). Sharp nearby, fuzzy far away: How neural language models use context. In *Proceedings of ACL 2018*, pages 284–294. (pages 42 and 58)

Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA. IEEE Press. (pages 12, 13, and 26)

Kim, S., Zimmermann, T., Pan, K., and Whitehead, E. J. J. (2006). Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90. (page 81)

Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2016). Character-aware neural language models. In *Proceedings of AAAI 2016*, pages 2741–2749. (page 44)

Kiss, A., Jász, J., and Gyimóthy, T. (2005). Using dynamic information in the interprocedural static slicing of binary executables. *Software Quality Journal*, 13(3):227–245. (page 19)

Korel, B. (1988). Pelas-program error-locating assistant system. *IEEE Trans. Softw. Eng.*, 14(9):1253–1260. (page 22)

Korel, B. and Laski, J. (1988). Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163. (page 19)

Korel, B. and Laski, J. (1988). Stad-a system for testing and debugging: user perspective. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pages 13–20. (page 22)

Kou, R., Higo, Y., and Kusumoto, S. (2016). A capable crossover technique on automatic program repair. In Ryotaro Kou, Yoshiki Higo, S. K., editor, *Proc. of 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 045–050. (pages 26 and 29)

Kudo, T. (2018). Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia. Association for Computational Linguistics. (page 122)

Kushman, N. and Barzilay, R. (2013). Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 826–836, Atlanta, Georgia. Association for Computational Linguistics. (page 37)

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. (page 123)

Laplante, P. A. (2000). *Dictionary of Computer Science Engineering and Technology*. CRC Press, Inc., USA. (page 23)

Le, Q. and Mikolov, T. (2014). Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–1188–II–1196. JMLR.org. (page 34)

Le, V. and Gulwani, S. (2014). Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 542–553, New York, NY, USA. ACM. (page 12)

Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA. IEEE Press. (pages 27, 76, 78, and 82)

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012a). GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72. (pages 27 and 28)

Le Goues, C., Wemer, W., and Forrest, S. (2012b). Representations and operators for improving evolutionary software repair. In *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*, pages 959–966. (page 27)

Lewis, D. K. (1973). Causation. *Journal of Philosophy*, 70(17):556–567. (pages 17 and 117)

Li, J., Wang, Y., Lyu, M. R., and King, I. (2018). Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4159–4165. International Joint Conferences on Artificial Intelligence Organization. (pages 46 and 64)

Li, Y., Zemel, R., Brockschmidt, M., and Tarlow, D. (2016). Gated graph sequence neural networks. In *Proceedings of ICLR'16*. (page 102)

Lian, L., Kusumoto, S., Kikuno, T., ichi Matsumoto, K., and Torii, K. (1997). A new fault localizing method for the program debugging process. *Information and Software Technology*, 39(4):271 – 284. (page 19)

Liang, P., Jordan, M. I., and Klein, D. (2010). Learning programs: A hierarchical bayesian approach. In Fürnkranz, J. and Joachims, T., editors, *ICML*, pages 639–646. Omnipress. (page 37)

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA. ACM. (page 21)

Lin, X. V. (2017). Program synthesis from natural language using recurrent neural networks. (page 37)

Lin, X. V., Wang, C., Zettlemoyer, L., and Ernst, M. D. (2018). NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA). (page 37)

Liu, C., Zhang, X., Han, J., Zhang, Y., and Bhargava, B. (2007). Indexing noncrashing failures: A dynamic program slicing-based approach. pages 455 – 464. (page 19)

Liu, H., Chen, Y., and Lu, S. (2016). Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 715–726, New York, NY, USA. Association for Computing Machinery. (page 33)

Liu, P., Tripp, O., and Zhang, C. (2014). Grail: Context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 318–329, New York, NY, USA. Association for Computing Machinery. (page 33)

Liu, P. and Zhang, C. (2012). Axis: Automatically fixing atomicity violations through solving control constraints. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 299–309. IEEE Press. (page 33)

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692. (pages 7, 99, and 123)

Long, F. and Rinard, M. (2015). Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA. ACM. (pages 31, 76, 78, and 82)

Long, F. and Rinard, M. (2016). Automatic patch generation by learning correct code. *SIGPLAN Not.*, 51(1):298–312. (pages 32, 76, 78, 82, and 99)

Luong, T., Socher, R., and Manning, C. (2013). Better word representations with recursive neural networks for morphology. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 104–113. Association for Computational Linguistics. (page 44)

Lyle, J. R. and Weiser, M. (1987). Automatic program bug location by program slicing. (page 18)

Maddison, C. J. and Tarlow, D. (2014). Structured generative models of natural source code. *CoRR*, abs/1401.0514. (page 36)

Malik, R. S., Patra, J., and Pradel, M. (2019). Nl2type: inferring javascript function types from natural language information. In *Proceedings of ICSE 2019*, pages 304–315. (page 46)

Mao, X., Lei, Y., Dai, Z., Qi, Y., and Wang, C. (2014). Slice-based statistical fault localization. *J. Syst. Softw.*, 89(C):51–62. (page 19)

Marcote, S. R. L. and Monperrus, M. (2015). Automatic repair of infinite loops. *CoRR*, abs/1504.05078. (page 33)

Markovtsev, V., Long, W., Bulychev, E., Keramitas, R., Slavnov, K., and Markowski, G. (2018). Splitting source code identifiers using bidirectional lstm recurrent neural network. *arXiv preprint arXiv:1805.11651.* (page 52)

Martinez, M., Durieux, T., Sommerard, R., Xuan, J., and Monperrus, M. (2017). Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Softw. Engg.*, 22(4):1936–1964. (pages 28 and 29)

McConnell, S. (2004). *Code Complete, Second Edition.* Microsoft Press, USA. (page 1)

Mechtaev, S., Yi, J., and Roychoudhury, A. (2015). Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 448–458. IEEE Press. (page 33)

Mechtaev, S., Yi, J., and Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 691–701, New York, NY, USA. Association for Computing Machinery. (page 33)

Melamud, O., Goldberger, J., and Dagan, I. (2016). context2vec: Learning generic context embedding with bidirectional LSTM. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pages 51–61, Berlin, Germany. Association for Computational Linguistics. (page 102)

Menon, A. K., Tamuz, O., Gulwani, S., Lampson, B., and Kalai, A. T. (2013). A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page I–187–I–195. JMLR.org. (page 37)

Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2016). Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843.* (pages 44 and 74)

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. (page 34)

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In Kobayashi, T., Hirose, K., and Nakamura, S., editors, *INTERSPEECH*, pages 1045–1048. ISCA. (page 42)

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA. Curran Associates Inc. (pages 6, 42, 99, 100, 101, and 120)

Mikolov, T., Sutskever, I., Deoras, A., Hai Son, L., Kombrink, S., and Cernock, J. (2012). Subword language modeling with neural networks. (page 44)

Miller, F. P., Vandome, A. F., and McBrewster, J. (2010). *Apache Maven*. Alpha Press. (page 79)

Min, H. and Li Ping, Z. (2019). Survey on software clone detection research. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*, ICMSS 2019, page 9–16, New York, NY, USA. Association for Computing Machinery. (page 36)

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition. (page 25)

Mohammadi, M., Chu, B., and Lipford, H. R. (2018). Automated detecting and repair of cross-site scripting vulnerabilities. *CoRR*, abs/1804.01862. (page 33)

Mohapatra, D. P., Mall, R., and Kumar, R. (2004). An edge marking technique for dynamic slicing of object-oriented programs. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 60–65 vol.1. (page 19)

Monperrus, M. (2018a). Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24. (pages 1, 11, 12, 13, and 102)

Monperrus, M. (2018b). Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24. (page 76)

Movshovitz-Attias, D. and Cohen, W. W. (2013). Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 35–40, Sofia, Bulgaria. Association for Computational Linguistics. (pages 36 and 37)

Müllerburg, M. A. F. (1983). The role of debugging within software engineering environments. *SIGPLAN Not.*, 18(8):81–90. (pages 3 and 75)

Neelakantan, A., Shankar, J., Passos, A., and McCallum, A. (2014). Efficient non-parametric estimation of multiple embeddings per word in vector space. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1059–1069, Doha, Qatar. Association for Computational Linguistics. (page 102)

Nessa, S., Abedin, M., Wong, W. E., Khan, L., and Qi, Y. (2008). Software fault localization using n-gram analysis. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 548–559. Springer. (page 26)

Neubig, G. (2016). Survey of methods to generate natural language from source code. (page 37)

Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. (2013a). Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 651–654, New York, NY, USA. ACM. (pages 62 and 74)

Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. (2015). Divide-and-conquer approach for multi-phase statistical migration for source code. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 585–596. IEEE Press. (page 36)

Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. (2013b). Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA. IEEE Press. (pages 12, 13, and 33)

Nguyen, T., Rigby, P. C., Nguyen, A. T., Karanfil, M., and Nguyen, T. N. (2016). T2api: Synthesizing api code usage templates from english texts with statistical translation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1013–1017, New York, NY, USA. ACM. (page 62)

Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. (2013c). A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA. ACM. (pages 35, 44, and 46)

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., and Nakamura, S. (2015). Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. (page 36)

Pan, H. and Spafford, E. H. (1992). Heuristics for automatic localization of software faults. Technical report. (page 19)

Pandi, I. V., Barr, E. T., Gordon, A. D., and Sutton, C. (2020). Opttyper: Probabilistic type inference by optimising logical and natural constraints. (page 123)

Parnas, D. L. (1985). Software aspects of strategic defense systems. *Commun. ACM*, 28(12):1326–1335. (page 12)

Parr, T. and Vinju, J. (2016). Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, page 137–151, New York, NY, USA. Association for Computing Machinery. (page 36)

Patra, J. and Pradel, M. (2016). Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. (page 34)

Pearl, J. (2000). *Causality: Models, Reasoning, and Inference.* Cambridge University Press, New York, NY, USA. (pages 17 and 117)

Pei, Y., Wei, Y., Furia, C. A., Nordio, M., and Meyer, B. (2011). Code-based automated program fixing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, page 392–395, USA. IEEE Computer Society. (page 31)

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics. (page 101)

Perino, N. (2013). A framework for self-healing software systems. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1397–1400, Piscataway, NJ, USA. IEEE Press. (pages 12 and 13)

Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., and et al. (2009). Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 87–102, New York, NY, USA. Association for Computing Machinery. (pages 4 and 18)

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237. Association for Computational Linguistics. (pages 7, 41, 99, 102, 103, and 120)

Polozov, O. and Gulwani, S. (2015). Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 107–126, New York, NY, USA. ACM. (page 12)

Pradel, M. and Sen, K. (2018). Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25. (pages 7, 35, 39, 45, 74, 76, 82, 83, 98, 99, 100, 104, 105, 107, 110, 111, 114, and 120)

Proksch, S., Lerch, J., and Mezini, M. (2015). Intelligent code completion with bayesian networks. *ACM Trans. Softw. Eng. Methodol.*, 25(1). (page 35)

Provilkov, I., Emelianenko, D., and Voita, E. (2019). Bpe-dropout: Simple and effective subword regularization. (page 122)

Psaier, H. and Dustdar, S. (2011). A survey on self-healing systems: Approaches and systems. *Computing*, 91(1):43–73. (pages 12 and 13)

Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. (2016). Sk_p: A neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2016, page 39–40, New York, NY, USA. Association for Computing Machinery. (page 34)

Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. (2014). The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 254–265, New York, NY, USA. Association for Computing Machinery. (page 29)

Qi, Y., Mao, X., Lei, Y., and Wang, C. (2013). Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 191–201, New York, NY, USA. Association for Computing Machinery. (pages 27 and 29)

Qi, Z., Long, F., Achour, S., and Rinard, M. (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 24–36, New York, NY, USA. Association for Computing Machinery. (pages 28 and 29)

Qian, J. and Xu, B. (2008). Scenario oriented program slicing. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 748–752, New York, NY, USA. ACM. (page 19)

Quirk, C., Mooney, R., and Galley, M. (2015). Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, Beijing, China. Association for Computational Linguistics. (page 37)

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training. *Available: https://blog.openai.com/language-unsupervised/.* (page 55)

Radford, A., Wu, J., Child, R., Luan, D., Amodei, L., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *Available: https://blog.openai.com/better-language-models/.* (pages 5 and 74)

Raghothaman, M., Wei, Y., and Hamadi, Y. (2016). Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 357–367, New York, NY, USA. ACM. (page 62)

Rahman, M., Palani, D., and Rigby, P. C. (2019). Natural software revisited. In *Proceedings of ICSE 2019*, pages 37–48. (page 44)

Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., and Devanbu, P. (2016a). On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA. ACM. (pages 34, 40, 41, 45, 65, 71, and 74)

Ray, B., Hellendoorn, V., Tu, Z., Nguyen, C., Godhane, S., Bacchelli, A., and Devanbu, P. (2016b). On the" naturalness" of buggy code. ICSE '16. ACM. (page 79)

Raychev, V., Bielik, P., Vechev, M., and Krause, A. (2016). Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 761–774, New York, NY, USA. ACM. (page 104)

Raychev, V., Vechev, M., and Yahav, E. (2014). Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA. ACM. (pages 35, 40, 62, and 63)

Renieres, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. (page 22)

Reps, T. W., Ball, T., Das, M., and Larus, J. R. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC '97/FSE-5*. (page 21)

Riganelli, O., Micucci, D., and Mariani, L. (2017). Policy enforcement with proactive libraries. *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).* (page 13)

Rinard, M. C. (2008). Technical perspective: Patching program errors. *Commun. ACM*, 51(12):86–86. (pages 11 and 117)

Robbes, R. and Janes, A. (2019). Leveraging small software engineering data sets with pre-trained neural networks. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*, page in press. (pages 41, 73, and 74)

Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Softw.*, 27(4):80–86. (page 35)

Rosenblum, D. S. (1992). Towards a method of programming with assertions. In *International Conference on Software Engineering*, pages 92–104. (page 16)

Rosenblum, D. S. (1995). A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31. (page 16)

Runciman, C. and Wakeling, D. (1993). Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245. (page 17)

Sadowski, C., van Gogh, J., Jaspan, C., Soederberg, E., and Winter, C. (2015). Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE).* (pages 76 and 114)

Saha, R. K., Lyu, Y., Lam, W., Yoshida, H., and Prasad, M. R. (2018). Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 10–13, New York, NY, USA. ACM. (page 78)

Santos, E. A., Campbell, J. C., Patel, D., Hindle, A., and Amaral, J. N. (2018). Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE. (pages 40 and 45)

Schmidt, J., Marques, M. R., Botti, S., and Marques, M. A. (2019). Recent advances and applications of machine learning in solid-state materials science. *npj Computational Materials*, 5(1):1–36. (page 25)

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909. (pages 5, 41, 44, 52, 53, 54, and 58)

Shrivastava, D., Larochelle, H., and Tarlow, D. (2020). On-the-fly adaptation of source code models using meta-learning. (page 123)

Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., and Keromytis, A. D. (2009). Assure: Automatic software self-healing using rescue points. *SIGARCH Comput. Archit. News*, 37(1):37–48. (page 15)

Singh, R. and Gulwani, S. (2015). Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, pages 398–414. Springer. (page 37)

Sliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *International Workshop on Mining Software Repositories.* ACM. (page 81)

Son, S., Mckinley, K. S., and Shmatikov, V. (2013). Fix me up: Repairing access-control bugs in web applications. In *In Network and Distributed System Security Symposium.* (page 33)

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958. (page 58)

Sterling, C. D. and Olsson, R. A. (2005). Automated bug isolation via program chipping. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 23–32, New York, NY, USA. ACM. (page 19)

Sumner, W. N. and Zhang, X. (2009). Algorithms for automatically computing the causal paths of failures. In Chechik, M. and Wirsing, M., editors, *Fundamental Approaches to Software Engineering*, pages 355–369, Berlin, Heidelberg. Springer Berlin Heidelberg. (page 25)

Sumner, W. N. and Zhang, X. (2010). Memory indexing: Canonicalizing addresses across executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 217–226, New York, NY, USA. Association for Computing Machinery. (page 25)

Surendran, R., Raman, R., Chaudhuri, S., Mellor-Crummey, J., and Sarkar, V. (2014). Test-driven repair of data races in structured parallel programs. *SIGPLAN Not.*, 49(6):15–25. (page 33)

Taha, A. ., Thebaut, S. M., and Liu, S. . (1989). An approach to software fault localization and revalidation based on incremental data flow analysis. In *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*, pages 527–534. (page 22)

Tan, S. H. and Roychoudhury, A. (2015). relifix: Automated repair of software regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 471–482. (pages 11 and 17)

Tan, S. H., Yoshida, H., Prasad, M., and Roychoudhury, A. (2016). Anti-patterns in search-based program repair. (page 28)

Tassey, G. (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing: Final Report*. Diane Publishing Company. (pages 2, 3, and 75)

Tip, F. (1994). A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands. (page 18)

Tip, F. and Dinesh, T. B. (2001). A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10(1):5–55. (page 19)

Tu, Z., Su, Z., and Devanbu, P. T. (2014). On the localness of software. In *Proceedings of the 22nd Symposium on the Foundations of Software Engineering*, pages 269–280. ACM. (pages 35, 44, 45, 46, 61, 63, 64, and 65)

Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., and Poshyvanyk, D. (2019). On learning meaningful code changes via neural machine translation. In *Proceedings of ICSE 2019*, pages 25–36. (page 45)

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., and Poshyvanyk, D. (2018a). An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 832–837, New York, NY, USA. ACM. (pages 37 and 80)

Tufano, M., Watson, C., Bavota, G., Penta, M. D., White, M., and Poshyvanyk, D. (2018b). An empirical study on learning bug-fixing patches in the wild via neural machine translation. *CoRR*, abs/1812.08693. (page 78)

Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 384–394, Stroudsburg, PA, USA. Association for Computational Linguistics. (page 101)

van Bruggen, D., Tomassetti, F., Howell, R., Langkabel, M., Smith, N., Bosch, A., Skoruppa, M., Maximilien, C., ThLeu, Panayiotis, (@skirsch79), S. K., Simon, Beleites, J., Tibackx, W., Rouél, A., jean pierre L, Schipper, D., edefazio, Mathiponds, you want to know, W., Beckett, R., ptitjes, kotari4u, Wyrich, M., Morais, R., bresai, Ty, Lebouc, R., Implex1v, and Haumacher, B. (2020). javaparser/javaparser: Release javaparser- parent-3.15.21. (page 96)

Vania, C. and Lopez, A. (2017). From characters to words to in between: Do we capture morphology? In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2016–2027. Association for Computational Linguistics. (page 44)

Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and singh, R. (2019). Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*. (page 114)

Vasilescu, B., Casalnuovo, C., and Devanbu, P. (2017). Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of ESEC/FSE 2017*, pages 683–693. (page 45)

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008. (pages 42 and 74)

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc. (pages 46 and 74)

Wang, S., Chollak, D., Movshovitz-Attias, D., and Tan, L. (2016a). Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 708–719, New York, NY, USA. Association for Computing Machinery. (page 34)

Wang, S., Liu, T., and Tan, L. (2016b). Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 297–308, New York, NY, USA. Association for Computing Machinery. (page 35)

Wang, T. and Roychoudhury, A. (2005). Automated path generation for software fault localization. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 347–351. ACM. (page 25)

Wang, Y., Patil, H., Pereira, C., Lueck, G., Gupta, R., and Neamtiu, I. (2014). Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108, New York, NY, USA. ACM. (page 19)

Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., and Zeller, A. (2010). Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 61–72, New York, NY, USA. ACM. (pages 17 and 31)

Weimer, W. (2006). Patches as better bug reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, page 181–190, New York, NY, USA. Association for Computing Machinery. (page 95)

Weimer, W., Forrest, S., Le Goues, C., and Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116. (pages 12 and 27)

Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA. IEEE Computer Society. (pages 12, 13, 27, and 96)

Weiser, M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA. IEEE Press. (page 18)

Weiser, M. D. (1979). *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, Ann Arbor, MI, USA. AAI8007856. (page 18)

White, M., Tufano, M., Martinez, M., Monperrus, M., and Poshyvanyk, D. (2019). Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490. (pages 7, 35, 99, and 102)

White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA. ACM. (pages 39 and 74)

White, M., Vendome, C., Linares-Vásquez, M., and Poshyvanyk, D. (2015). Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA. IEEE Press. (pages 44 and 64)

Wieting, J., Bansal, M., Gimpel, K., and Livescu, K. (2016). Charagram: Embedding words and sentences via character n-grams. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1504–1515, Austin, Texas. Association for Computational Linguistics. (page 102)

Wilkerson, J. L. and Tauritz, D. (2010). Coevolutionary automated software correction. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, page 1391–1392, New York, NY, USA. Association for Computing Machinery. (pages 26 and 29)

Wilkerson, J. L. and Tauritz, D. R. (2011). Scalability of the coevolutionary automated software correction system. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, page 243–244, New York, NY, USA. Association for Computing Machinery. (page 29)

Wilkerson, J. L., Tauritz, D. R., and Bridges, J. M. (2012). Multi-objective coevolutionary automated software correction. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, page 1229–1236, New York, NY, USA. Association for Computing Machinery. (page 29)

Willett, P. (2006). The porter stemming algorithm: then and now. *Program*, 40(3):219–223. (pages 46 and 52)

Williams, C. and Spacco, J. (2008). SZZ revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 32–36, New York, NY, USA. ACM. (page 81)

Wong, W. and Li, J. (2006). An integrated solution for testing and analyzing java applications in an industrial setting. volume 2005, pages 8 pp.–. (page 20)

Wong, W. E., Debroy, V., Golden, R., Xu, X., and Thuraisingham, B. (2012). Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169. (page 25)

Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740. (pages 16, 17, 19, and 21)

Wong, W. E. and Qi, Y. (2009). Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597. (page 25)

Wong, W. E., Sugeta, T., Qi, Y., and Maldonado, J. C. (2005). Smart debugging software architectural design in sdl. *J. Syst. Softw.*, 76(1):15–28. (page 20)

Wotawa, F. (2002). On the relationship between model-based debugging and program slicing. *Artif. Intell.*, 135(1-2):125–143. (page 19)

Wotawa, F. (2010). Fault localization based on dynamic slicing and hitting-set computation. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 161–170, Washington, DC, USA. IEEE Computer Society. (page 19)

Xin, B., Sumner, W. N., and Zhang, X. (2008). Efficient program execution indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 238–248, New York, NY, USA. Association for Computing Machinery. (page 25)

Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L. (2005). A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36. (page 18)

Xu, F. F., Jiang, Z., Yin, P., Vasilescu, B., and Neubig, G. (2020). Incorporating external knowledge through pre-training for natural language to code generation. (page 123)

Xuan, J., Martinez, M., DeMarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D., and Monperrus, M. (2017). Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Softw. Eng.*, 43(1):34–55. (page 33)

Yang, Z., Dai, Z., Yang, Y., Carbonell, J. G., Salakhutdinov, R., and Le, Q. V. (2019). XLNet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237. (pages 7 and 99)

Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. (2018a). Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of MSR 2018*, pages 476–486. (page 45)

Yin, P. and Neubig, G. (2017). A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics. (page 37)

Yin, P., Neubig, G., Allamanis, M., Brockschmidt, M., and Gaunt, A. L. (2018b). Learning to represent edits. *CoRR*, abs/1810.13337. (pages 35 and 102)

Yoo, S., Xie, X., Kuo, F.-C., Chen, T. Y., and Harman, M. (2014). No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. (page 23)

Yu, F., Alkhalaf, M., and Bultan, T. (2011). Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 251–260, New York, NY, USA. Association for Computing Machinery. (page 33)

Yu, F., Shueh, C.-Y., Lin, C.-H., Chen, Y.-F., Wang, B.-Y., and Bultan, T. (2016). Optimal sanitization synthesis for web application vulnerability repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 189–200, New York, NY, USA. Association for Computing Machinery. (page 33)

Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, page 1–10, New York, NY, USA. Association for Computing Machinery. (page 23)

Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200. (page 23)

Zhang, X., Gupta, N., and Gupta, R. (2006). Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 272–281, New York, NY, USA. Association for Computing Machinery. (page 25)

Zhang, X., Gupta, N., and Gupta, R. (2007a). Locating faulty code by multiple points slicing. *Softw. Pract. Exper.*, 37(9):935–961. (page 19)

Zhang, X., Gupta, N., and Gupta, R. (2007b). A study of effectiveness of dynamic slicing in locating real faults. *Empirical Softw. Engg.*, 12(2):143–160. (page 19)

Zhang, X., He, H., Gupta, N., and Gupta, R. (2005). Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 33–42, New York, NY, USA. ACM. (page 19)

Zhang, X., Tallam, S., Gupta, N., and Gupta, R. (2007c). Towards locating execution omission errors. *SIGPLAN Not.*, 42(6):415–424. (page 19)

Zhong, V., Xiong, C., and Socher, R. (2017). Seq2sql: Generating structured queries from natural language using reinforcement learning. (page 37)

Zimmermann, T. and Zeller, A. (2001). Visualizing memory graphs. In *Revised Lectures on Software Visualization, International Seminar*, page 191–204, Berlin, Heidelberg. Springer-Verlag. (page 23)