### LEARNING FROM CODE AND NON-CODE ARTIFACTS

by

Jordan Henkel

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

#### UNIVERSITY OF WISCONSIN-MADISON

2022

Date of final oral examination: 05/26/2022

The dissertation is approved by the following members of the Final Oral Committee: Thomas Reps, Professor, Computer Sciences Aws Albarghouthi, Associate Professor, Computer Sciences Loris D'Antoni, Associate Professor, Computer Sciences Jiepu Jiang, Assistant Professor, Information School

© Copyright by Jordan Henkel 2022 All Rights Reserved To my wife, Danielle

First and foremost, I'd like to thank my family for their unwavering support throughout my years in graduate school. I thank Danielle, my wife, for believing in me and supporting me (especially in these last two years). I thank my mother Marilyn and my grandmother Helen for pushing me to be the best that I could be and supporting me for so many years so that I could pursue this dream. I thank my father Terrell for always believing in me and my work. I thank my uncle Tim and cousins Bob and Matt for introducing me to computing and technology when I was younger. I could not have asked for a better family and it is largely thanks to them that I have completed this work.

This dissertation and all of my academic writing thus far has been greatly improved and refined by the guidance of my advisor, Tom Reps. I want to thank Tom for his willingness to work with a new student who (at the time) knew little about Program Analysis, Software Engineering, and Machine Learning. Along with Tom, I would also like to thank the rest of my committee (Aws Albarghouthi, Loris D'Antoni, and Jiepu Jiang) for their time and consideration. I'd also like to thank the UW–Madison PL/SE/Security faculty I've had the chance to interact (both current, former, and visitors).

I especially thank Ben Liblit and Shuvendu Lahiri who were fantastic collaborators and also mentors to me early in my graduate career. I want to thank Shuvendu for jump-starting my career with a chance to intern at Microsoft Research. It was during that internship that I met Chris Bird who would become yet another spectacular mentor and supporter of my work. I credit Shuvendu and Chris with many of my positive perceptions of Microsoft—without their support my career would have been on a vastly different trajectory.

That summer at Microsoft Research was a transformative summer for me, and I met many interns who inspired and motivated me: Chungha Sung, Danielle Gonzalez, Foyzul Hassan, Jose Abel Castellanos, James Davis, and many others. I also remember fondly how encouraging and kind the other researchers were. In particular, I wish to thank Tom Zimmermann for organizing fun outings for us RiSE interns; to this

day I remember Throw Throw Burrito, good ramen, baseball, and matching shirts that have held up surprisingly well throughout the years! I'd also like to thank Nachi Nagappan, Chetan Bansal, Ben Zorn, Thomas Ball, Madan Musuvathi, and the whole group at Microsoft Research for an incredible summer.

Less than a year later I had an opportunity, again, to work with Microsoft. I remember interviewing with Brian Kroth and Rathijit Sen at a local coffee shop and being elated to have the opportunity to intern again, but in Madison instead of Redmond (unbeknownst to me, COVID would happen, and everyone would be interning at home, regardless of location). That second summer at Microsoft, while less social, was still incredible. I want to thank Brian for proving to be another amazing mentor and supporting my work; the summer after, I returned to work with Brian again, and I was also fortunate to meet Venkatesh Emani and have him as a yet another collaborator and mentor.

The two summers I spent at Microsoft's Gray Systems Lab were full of wonderful people and I'm beyond thrilled to join the team post-graduation. I want to thank Raghu Ramakrishnan, Carlo Curino, and Avrillia Floutara for believing in my work as an intern enough to welcome me to the group as a full-time employee. I'd also like to thank everyone at the GSL for making me feel welcome during my two (virtual) summers (no easy task given the burden COVID imposed on everyone's lives).

At one point, early in my graduate career, I submitted an entry to a Machine Learning for Code Repair competition. It was just a few months later that I found out I had won and would be traveling to Stockholm (my first time leaving the county) to speak at the KTH Royal Institute of Technology. I'd like to thank Martin Monperrus and Zimin Chen for organizing that competition. Martin gave me my first opportunity as a graduate student to leave the US and talk about my research abroad, and for that I am truly grateful.

Later in my graduate career, Marcelo d'Amorim reached out to collaborate. It was through Marcelo that I had the chance to work with Leopoldo Teixeria and Denini Silva who were also wonderful collaborators. I want to thank Marcelo, Leopoldo, Denini, and all of my other collaborators I wrote with throughout graduate school (Tom, Ben, Shuvendu, Chris, Somesh, Aws, and Zi). I also want to thank the graduate students that went before me: Peter Ohmann, Venkatesh Srinivasan, Tushar Sharma, Calvin Smith, Dave Brown, Alisa Maas, Kausik Subramanian, Sam Drews, Qinheping Hu, and Jinman Zhao. They welcomed me into the group as a new student and helped make the PL group a fun place to be. In particular, I'd like to thank my graduate student "brothers" John Cyphert and Jason Breck—I cannot imagine how drastically different (and un-fun) my experience could have been without their friendship and support—I especially appreciated their willingness to eat (occasionally sketchy) lunches at new local restaurants (my favorite past time). It is hard to overstate the impact true friends can make on a graduate journey.

Finally, I wish to thank my friends Sam Bald, Scarlett Pisarek, Stephanie Blumer and Logan Stevens who have all, on occasion, had to listen to me drone on about what I've been working on. I also thank the current graduate students I had the chance to interact with: Wiley Corning, Zi Wang, Michael Vaughn, and David Merrell.

This research was supported, in part, by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; by Facebook under two Probability and Programming Research Awards; by ONR under grants N00014-17-1-2889 and N00014-19-2318; by NSF under grants CCF-1318489, CCF-1420866, and CCF-1423237; and by the Microsoft Research Ph.D. Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

Co	ntent	S	V		
Lis	st of [	Tables	viii		
Lis	st of I	Figures	ix		
Ab	ostrac	t	xi		
1	Intro	oduction	1		
	1.1	Motivation	3		
	1.2	Contributions	6		
	1.3	Thesis Outline	8		
	1.4	Notes	9		
<b>2</b>	Code Vectors				
	2.1	Introduction	10		
	2.2	<i>Overview</i>	15		
	2.3	Technique	17		
	2.4	Experiments	23		
	2.5	Related Work	38		
	2.6	Future Work	39		
	2.7	Notes	39		
3	Ope	n World Mining	41		
	3.1	Introduction	41		
	3.2	Overview	45		
	3.3	Technique	50		
	3.4	Experiments	60		
	3.5	Related Work	69		
	3.6	Future Work	72		

	3.7	<i>Notes</i>	72
4	Sem	antic Robustness	73
	4.1	Introduction	73
	4.2	Semantic Robustness	80
	4.3	Framework	83
	4.4	Comparable Techniques	90
	4.5	Experiments	92
	4.6	Related Work	103
	4.7	Future Work	105
	4.8	<i>Notes</i>	106
<b>5</b>	Doc	kerfile Mining	107
	5.1	Introduction	107
	5.2	Dataset	113
	5.3	Technique	115
	5.4	Experiments	128
	5.5	Related Work	134
	5.6	Future Work	136
	5.7	<i>Notes</i>	137
6	Doc	kerfile Repair	138
	6.1	Introduction	138
	6.2	Sources of Build Failures	142
	6.3	Dataset	146
	6.4	Technique	148
	6.5	Experiments	155
	6.6	Related Work	166
	6.7	Future Work	170
	6.8	Notes	170
7	code	e-book	171

	7.1	Introduction	171	
	7.2	Query Language	173	
	7.3	Case Studies	194	
	7.4	Related Work	198	
	7.5	Future Work	198	
	7.6	Notes	199	
8	8 Conclusion			
	8.1	Contributions	202	
	8.2	Limitations	205	
	8.3	The Impact of Large Language Models	206	
	8.4	Concluding Remarks	207	
	8.5	Notes	208	
Α	Ana	logy Suite: Representative Pairs	209	
Re	References			

# LIST OF TABLES

2.1	Analogy Suite Details	24
2.2	Top-5 closest words to affs_bread and kzalloc	29
3.1	Grid search parameters	61
3.2	Best scoring configurations for each of the five target projects $\ldots$ .	64
3.3	DAC compared to off-the-shelf techniques	65
3.4	DAC compared to off-the-shelf techniques (boosted)	65
3.5	A comparison of word-vector learners and sampling techniques	67
4.1	Single attack efficacy (with random/gradient resolution)	94
4.2	A comparison of models trained using three different pipelines $\ldots$ .	96
4.3	F1 across normal/robust models on $out$ -of-distribution test sets $\ldots$	100
4.4	F1 (seq2seq) on the Java to Python $cross-language$ transfer task	102
4.5	F1 (seq2seq) on the Python to Java cross-language transfer task $\ . \ . \ .$	102
5.1	Detailed breakdown of the <i>Gold Rules</i>	119
6.1	Selected Repairs	153
6.2	Selected Suggestions	154
6.3	Repair Coverage	163
6.4	Accepted Pull Requests.	167
A.1	Analogy Suite: Representative Pairs	209

# LIST OF FIGURES

2.1	An example procedure	16
2.2	Traces from the symbolic execution of the procedure in Fig. 2.1 $\ldots$	16
2.3	Result of abstracting the two traces in Fig. 2.2b	17
2.4	Example derivations for selected abstractions	18
2.5	Sample procedure with generated abstractions shown as comments	20
2.6	Encoding of abstractions	22
2.7	Traces for Fig. 2.5 generated by the encoding from Fig. 2.6	22
2.8	Excerpt from nv17_fence.c	26
2.9	Ablation study: top-1 analogy results for eight configurations	32
2.10	Top-1 analogy results for syntactic versus semantic abstractions $\ . \ . \ .$	35
3.1	Overview of the ml4spec toolchain	46
3.2	Example procedure and corresponding trace	48
3.3	Clusters generated via Domain-Adapted Clustering (DAC) $\ . \ . \ . \ .$	49
3.4	Example specifications (mined from projected traces) $\ldots \ldots \ldots$	51
3.5	Comparison between two clusters	62
3.6	Peak benchmark performance for varying values of $\alpha$	69
4.1	Example function with edits that fool code2seq	75
4.2	A comparison of normal/robust models against various adversaries	98
5.1	An overview of the <b>binnacle</b> toolset	113
5.2	Dockerfile at each of the three phases of our phased-parsing technique .	116
5.3	Three example Tree Association Rules (TARs)	120
5.4	A depiction of rule mining in $\verb+binnacle+$ via frequent sub-tree mining	121
5.5	An example of the abstraction process	122
5.6	<b>binnacle</b> 's rule engine applied to an example Dockerfile	123
5.7	Density histograms showing the distributions of our three metrics $\ldots$	126
5.8	Four examples of actual rules recovered by <b>binnacle</b> 's miner	129

6.1	Solving python-pip unavailable on ubuntu:latest	144
6.2	Dockerfile with inconsistent Ruby version dependencies $\ldots$ $\ldots$ $\ldots$	145
6.3	An overview of Shipwright	146
6.4	An overview of clustering in Shipwright	148
6.5	Proportion of different kinds of solutions within each cluster $\ldots$	159
6.6	Breakdown of the 20,526 files we attempted to build $\ldots \ldots \ldots$	160
7.1	A (high level) grammar for code-book queries	181
7.2	A grammar for textual constraint in code-book queries	182
7.3	A grammar for code-book's unless construct	183
7.4	A grammar for code-book's if-guard construct	183
7.5	code-book's full code-snippet grammar	184
7.6	The second code-book prototype	201
7.7	The second $code-book$ prototype (QBE-style query language) $\ \ \ldots \ \ldots$	201
8.1	Links to various tools and datasets	207

#### ABSTRACT

Three things are fundamentally true about software: (i) every day that passes we, as a society, generate more software (more code, more documentation, and more software-related artifacts of all kinds), (ii) it is easier to write new software than it is to understand and maintain existing software, and (iii) we *depend* on software in every area of our lives (from critical infrastructure to entertainment and everything in between). These three fundamental truths set the stage for one massive problem: if there is more software every day, and it is hard to understand and maintain, how can we ever "keep up" with this unbounded growth? How can we ever truly *understand* the software we depend on if we are adding to it every single day?

In this thesis, we provide new ideas and tools that help with some of these issues. More specifically, this thesis takes the position that we need tools and techniques for understanding and learning from software. To do this, we consider software to be a composite of source code and other, non-code, artifacts (build scripts, documentation, etc.). We introduce techniques for working with both code and non-code artifacts; for code, we introduce a form of code embeddings (learned from a semantic representation of code: abstracted symbol traces); we then create a novel specification mining technique that uses these semantic code embeddings; additionally, we explore the robustness of models of code; and, to address non-code artifacts, we mine tree-association rules from Dockerfiles, from which we learn best practices; we take these learned best practices and create a human-in-the-loop technique for automated repair of Dockerfiles.

Finally, to accelerate empirical research on software and lay a groundwork for a more comprehensive solution to trusting the growing amount of software we—as a society—create, we introduce code-book. code-book is a tool for interactively querying and analyzing code inspired by the great successes of the Data Science community. With code-book, we introduce a novel query-by-example-based query language for asking questions about code. Furthermore, we develop this query language so that users can ask questions that incorporate both code structure and "fuzzy" semantic constraints (based on code embeddings).

#### 1 INTRODUCTION

This is the beginning. How should we start?

GPT-3: We should start by introducing ourselves.

Alright, I'm Jordan and this is my thesis about learning from code and non-code artifacts. What next?

GPT-3: Well, we need to establish what our research question is. For this project, our question is: How can we learn from code and non-code artifacts?

Sounds good enough, I'll have quite a bit to say about your question in the next 200 pages. Shall we begin?

GPT-3: Yes, let's begin.

#### A dialogue with OpenAI's $GPT-3^1$

This thesis is written from the bottom of a pit: the nearly bottomless pit of ever-expanding software that powers the digital and physical world. Our growing dependence on software is, already, concerning (and many theses have been written on that topic)—but, what's even *more* concerning is the runaway pace at which we create more software. Every single day there is more software in this world than there was the previous day. There are many reasons for the explosive growth of software, but one reason in particular stands out: *software is easier to write than it is to read, understand, and maintain.* 

This growth fuels a need for tools and techniques that assist humans in understanding and maintaining software. However, before we can discuss such tools and techniques, we must discuss *what* software is. Modern software is an assemblage of several heterogenous artifacts but, primarily, we will treat software as a combination

<sup>&</sup>lt;sup>1</sup>GPT-3's outputs are presented unedited. To generate these dialogs, a good deal of "prompt engineering" was used (e.g., trying different wording for questions, changing temperature settings, asking questions in separate sessions, etc.).

of two distinct classes of artifacts: code and non-code artifacts. When we speak of code artifacts (or, simply, *code*) we will be referring to the source code of a given piece of software. When we speak of non-code artifacts we will be referring to things such as configuration files, infrastructure-related artifacts (like Dockerfiles), and documentation.

In this dissertation, we explore techniques for learning from both code and noncode artifacts. First, we introduce a novel technique for learning from code using code embeddings and a form of lightweight symbolic execution; next, we build on this work by exploring a technique for mining specifications that leverages the learned code embeddings. Additionally, we consider key questions about the *robustness* of models learned from code. Switching to the domain of non-code artifacts, we present a novel technique for learning from Dockerfiles; again, we build on this work by exploring a human-in-the-loop system for repairing Dockerfiles. Finally, we set our sights higher and explore how all of the previous works could have been accelerated by introducing support for *data science on code*.

This last idea, of supporting *data science on code*, is perhaps the most important idea in this thesis. Every piece of work we examine, and every experiment we report, has required some amount of bespoke infrastructure. This is the nature of research on software; however, the need for custom tooling and data makes research on software an onerous and time-consuming process. We aspire to improve this situation and make research on software as easy and intuitive as research on other, more traditional, data sources.

To accelerate research on software, we borrow from the Data Science community and their notebook-based interactive ecosystem. We introduce a similar notebookbased ecosystem for asking questions about code—more specifically, we introduce a novel system for code queries that: (i) works at scale, (ii) uses a beginner-friendly query-by-example-style language, and (iii) uses embeddings to allow for queries mixing code structure and "fuzzy" semantic constraints. We label this system **code-book**.

Although my work on code-book is ongoing, I believe it to be a critical next step in accelerating many common tasks in software engineering research. Furthermore, I think of code-book as, in some sense, my vision for a "ladder" we can use to ascend from the nearly bottomless pit of software. My hope is that, with **code-book**, we can make understanding and interacting with existing software nearly as easy as writing new software.

# 1.1 Motivation

The whole of this thesis is really about ascending from the nearly "bottomless pit" of ever-expanding software; however, we approach this ascent in small and discrete steps. With this frame of reference, we motivate each of these steps as follows:

### Step 1: Code Vectors

In Chapter 2, we first tackle the problem of *understanding* a large amount of existing code. Specifically, we attempt to learn from the Linux kernel. Although the source code of the Linux kernel is text, it is more than mere prose: it must conform to the syntax of the languages it is written in and those languages endow that text with a semantics. Furthermore, that text can be translated into a large number of heterogenous representations (Concrete Syntax Trees, Abstract Syntax Trees, Control Flow Graphs, Call Graphs, Def-Use Chains, etc.).

Prior to the work we introduce in Chapter 2, there were many attempts to apply off-the-shelf learning algorithms to code—those attempts often treated the code as text or used a representation of code that was quite similar to plain text. There were also a small number of attempts to use more structural representations (such as Abstract Syntax Trees). Instead of trying to refine any of the previous approaches, we introduced a novel technique based on word embeddings (a popular off-the-shelf learning algorithm); we learned these embeddings from a representation that incorporated the semantics of code (something *far from* the code as text).

### Step 2: Open-World Mining

As software continues to grow, so do the number of libraries and frameworks. These libraries and frameworks make their functionality available to downstream users through Application Programming Interfaces (APIs). Although some APIs may be simple, many APIs offer a large range of operations over complex structures. Furthermore, staying within the correct usage patterns for a given API can require domain-specific knowledge about the API and its idiosyncratic behaviors. This burden is often worsened by insufficient documentation and explanatory materials.

In Chapter 3, we consider how the code embeddings we produce in Chapter 2 can be leveraged to create something else of value: specifications (usage patterns). We seek to infer such patterns automatically by learning from a large corpus of existing code. To do so, we use both the idea of code embeddings and ideas from traditional static mining techniques. We find that combining traditional techniques with learning-assisted metrics creates better results than either approach in isolation.

#### Step 3: Semantic Robustness

As we take two small steps out of "the pit," we must carefully consider the implications of *learning* from code. Unlike traditional techniques, approaches involving learning are often hard to understand—learned models or embeddings are somewhat opaque to human intuition. In Chapter 4, we consider a key question about models learned from code: are models of code *robust*?

Why might we want robust models of code? There are many answers, ranging from usability to security. Consider, for instance, a model that explains in English what a piece of code is doing—the *code-captioning* task. A developer using such a model to navigate a new code base should not receive completely different explanations for similar pieces of code. Models that change their outputs based on irrelevant details are over-sensitive (and under-robust). In Chapter 4, we thoroughly explore the question of robustness for two models of code, and create a framework for answering such questions about other models more quickly (by providing tools, data, and a controllable environment for consistent evaluations).

### Step 4: Dockerfile Mining

After having taken three small steps toward learning from code (by producing embeddings, mining specifications, and understanding the robustness of learned models), we pivot to *non-code artifacts*. Specifically, our fourth step out of "the pit" will be to examine Dockerfiles. Docker is a system for lightweight virtualization via containers. Docker containers are built from images, which are, in essence, a declarative list of instructions used to specify a reproducible environment. The text file that contains this list of instructions is termed a Dockerfile. Historically, Dockerfiles had been somewhat neglected in both industrial and academic research. This lack of attention left us with a unique opportunity to try and apply our techniques and ideas for analyzing code on a non-code artifact in an area where little prior work existed.

In Chapter 5, we introduce a toolchain for semantics-aware Dockerfile verification and unsupervised rule mining. Given that Dockerfiles are one of the most prevalent non-code infrastructure-related artifacts in industry, we focus our toolchain towards reliable techniques that require little-to-no human intervention. We find that our expertise from work on code only partially translates to this new domain and that different techniques are needed to make the most of non-code artifacts such as Dockerfiles.

### Step 5: Repairing Dockerfiles

What it would take to go past simply *identifying* issues in Dockerfiles? One clear way to surpass basic identification is to repair automatically files that are verifiably broken. ("Broken," in this context, means either non-functional or non-compliant with established best practices.)

In Chapter 6, we produce a human-in-the-loop tool for automated repair of Dockerfiles. Similar to Chapter 3, we use a mix of learning-assisted techniques and traditional techniques to produce a tool that works robustly. In this work, we also make our first attempt to validate our results in the wild by directly submitting GitHub pull requests generated by our prototype.

#### Plans for a ladder: code-book

Finally, we end this thesis by discussing preliminary (and ongoing) work towards a vision for *data science on code*. This last piece of work is a significant step past any of the other work discussed thus far. In a world where the amount of software is increasing each day, we need a platform for accelerating the kind of work done in Chapters 2, 3, 5, and 6. Therefore, in Chapter 7, we set out to mirror the stellar tooling created by the Data Science community and apply it to the domain of Empirical Software Engineering research. To do this, we ask (and provide on answer for) the question: what would *Data Science* look like for code?

# **1.2** Contributions

Given the focus of this thesis (learning from code and non-code artifacts), we group our contributions into three distinct categories: (i) contributions to learning from code, (ii) contributions to learning from non-code artifacts, and (iii) contributions to software-engineering research. In general, many of our contributions come in the form of new techniques (with supporting experimental evidence), and new tools and accompanying datasets (which, unless explicitly noted, are publicly available).

### Contributions to learning from code

We introduce a novel technique for learning from code via *code embeddings*. These code embeddings are learned from abstract symbolic traces and, to capture such traces, we introduced a new tool for lightweight symbolic execution of C programs (that is capable of scaling to programs as large as the Linux kernel). The symbolic execution engine we created is *parametric* and capable of utilizing user-defined abstractions to influence the vocabulary of the generated traces. We also supply a new benchmark of *code analogies* gleaned from the Linux kernel. We achieve 93% top-1 accuracy on this benchmark using our learned embeddings, as well as 76% top-3 accuracy on the downstream task of predicting error codes for failing traces extracted from the Linux kernel.

Taking the idea of *code embeddings* and the parametric lightweight symbolic execution engine we designed, we devised a novel approach to specification mining for code. The approach we introduce utilizes a mix of both traditional metrics and learned metrics to create a method that is more robust than either the traditional or learned approaches are in isolation. Furthermore, we performed a comparison of three of the most popular word-vector learners and three different trace sampling techniques to provide a definitive reference for practitioners looking to learn from symbolic traces.

Finally, we round out our contributions to learning from code by looking at the *robustness* of models of code. In this area, we developed a novel framework for creating controllable adversaries to test the robustness of models of code. We also devised a new technique for training robust models of code. Additionally, we present the first results (to the best of our knowledge) on how robust models of code perform on out-of-distribution data, and how they adapt across different programming languages.

#### Contributions to learning from non-code artifacts

In the realm of non-code artifacts, we focus our contributions on Dockerfiles (because they are one of the most prevalent non-code infrastructure-related artifacts used in industry). First, we developed a technique for parsing Dockerfiles, called *phased parsing*, that accounts for the deeply-nested languages used within infrastructurerelated artifacts such as Dockerfiles. Using our parsing technique, we created a dataset of over two-hundred-thousand Dockerfiles downloaded from GitHub with corresponding Abstract Syntax Trees (generated by our parser). We made this dataset available to the community along with our parsing technique. Furthermore, we introduced a novel mining technique capable of extracting best practices (in the form tree-association rules) from a corpus of Dockerfiles. We used this technique on our dataset and automatically extracted 26 tree-association rules of which 16 were new rules we had not discovered via manual analysis. Additionally, we developed a static-checking engine capable of applying tree-association rules to Dockerfiles (and other tree-structured non-code artifacts). Using this checking engine we found that, to our surprise, a sample of Dockerfiles written by developers in industry were *worse*, on average, than those found "in the wild" on GitHub.

Finally, we pivot from simply *detecting* issues (via mined patterns) to performing semi-automated repair. We do this by introducing a novel human-in-the-loop technique for Dockerfile repair. We applied this technique to broken Dockerfiles and submitted the results to several open-source repositories as pull requests. Out of 45 submitted requests we had 19 requests accepted and integrated. We also summarized our experience by classifying the *kinds* of failures in Dockerfiles.

#### Contributions to the field at large

Our last contribution is to the field of software-engineering research: as a way to support the idea of *Data Science on Code*, we developed a tool called **code-book**. With **code-book**, we introduce the idea of interactive and iterative code querying. We also devised a novel language for expressing code queries based on the Query-by-Example paradigm. Although our work on **code-book** is ongoing, we hope that the idea of Data Science on Code—and the prototype implementation of a notebook-based environment for querying and interacting with code—are valuable additions to the software-engineering research community.

# **1.3** Thesis Outline

Chapter 2 introduces "Code Vectors," a technique for learning embeddings from abstracted symbolic traces (generated by a lightweight symbolic-execution engine). In Chapter 3, we build on the ability to learn embeddings from code by mining specifications from abstracted symbolic traces using a combination of traditional and learned metrics. In Chapter 4, we investigate the *robustness* of models learned from code and explore how adversaries based on semantics-preserving transforms impact popular models of code (and, furthermore, how one can use such adversaries to *train robust models*). Starting in Chapter 5, we pivot to an exploration of non-code artifacts (specifically, Dockerfiles); in Chapter 5 we introduce a technique for mining from a corpus of Dockerfiles and extracting best-practices. In Chapter 6, we build upon our work on mining and validating best practices in Dockerfiles by introducing a human-in-the-loop system for automated repair of broken Dockerfiles. Stepping back from specific techniques, in Chapter 7, we introduce **code-book**: a framework for applying the best ideas from the Data Science community to software-engineering research. Finally, in Chapter 8 we present some closing thoughts.

# 1.4 Notes

Here we are at the beginning. I have spent many hundreds of hours thinking about what I would put in my thesis. Unfortunately, out of those hundreds of hours I had not thought much about how to *begin* the thesis. Given this, I spent much of my time writing everything *but* this introduction. But, in the end, perhaps this is the best approach. I think the topic of learning from code is important and, since I started this work about five years ago, so many things have *changed*. We have large language models. We have the "bitter lesson," which tells us how techniques that can leverage compute will almost always outpace techniques which rely on human cleverness in design and implementation. Most of all, we have the perspective of five years worth of continued research and development toward techniques that learn from all kinds of structured data, not just code. I think the next decade will be an incredibly exciting time to be thinking about learning from code and, more generally, learning from complex structured representations of data. If one wishes to learn from code (that is, to apply modern learning algorithms) what representation of code should one use?

GPT-3: There is no single answer to this question. It depends on the specific learning task and on the specific data set. Some learning tasks may be better suited to a bag-of-words representation, while others may be better suited to an AST representation. There is no general rule that applies to all learning tasks.

Sure, but why don't we just use a program's text?

GPT-3: A program's text is not a very good representation for learning from code. This is because the text does not contain any information about the structure of the code. A bag-of-words representation or an AST representation is a better choice for learning from code, because these representations capture the structure of the code.

That's insightful but, unfortunately, your existence directly contradicts the last statement you made.

GPT-3: I'm sorry, I don't understand what you mean.

A dialogue with OpenAI's GPT-3

# 2.1 Introduction

As we explore methods to learn from both code and non-code artifacts, we will begin by considering code and the challenges one encounters trying to *learn* from code. Once we have developed methods for learning from code, we will describe techniques that build upon these methods (downstream applications), and we will examine how the learned models behave.

#### 2.1.1 Motivation

Code is an interesting artifact to study. We write code as text but that text is more than mere prose: it must conform to the syntax of the language we are writing in; moreover, the code we've written (as text) has a semantics defined by the language we are writing in. Finally, when it comes to working with code, we employ a large number of heterogenous representations (Concrete Syntax Trees, Abstract Syntax Trees, Control Flow Graphs, Call Graphs, Def-Use Chains, etc.).

Given the plethora of available representations for code, we pose the following motivating question:

#### Motivating Question

If one wishes to *learn* from code (that is, to apply modern learning algorithms) what representation of code should one use?

Prior to this work, there were many attempts to apply off-the-shelf learning algorithms to code, and to use representations of the code that were close to plain text. There were also a smaller number of attempts to learn from program *structure* (by using representations derived from a program's Abstract Syntax Tree). We will examine how *word embeddings* (a popular off-the-shelf learning algorithm) can be applied to a very *semantic* representation of code (something *far from* the text of the program from which we wish to learn).

#### 2.1.2 Goals

Word embeddings are a well-studied method for converting a corpus of naturallanguage text to vector representations of words embedded into a low-dimensional space. These techniques have been applied successfully to programs before (Nguyen et al., 2017b; Pradel and Sen, 2017; Gu et al., 2016), but different encodings of programs into word sequences are possible, and some encodings may be more appropriate than others as the input to a word-vector learner. The high-level goals of our work can be stated as follows:

### Goals

Devise a parametric encoding of programs into word sequences that (i) can be tuned to capture different representation choices on the spectrum from (mainly) syntactic to (mainly) semantic, (ii) is amenable to word-vector-learning techniques, and (iii) can be obtained from programs efficiently.

We satisfy high-level goals (i) and (iii) by basing the encoding on a lightweight form of *intraprocedural symbolic execution*.

- We base our technique on symbolic execution due to the gap between syntax (e.g., tokens or abstract syntax trees (ASTs)) and the semantics of a procedure in a program. In particular, token-based techniques impose a heavy burden on the embedding learner. For instance, it is difficult to encode the equivalence between constructions such as a == b and !(a != b) via a learned, low-dimensional embedding (Allamanis et al., 2016a).
- Our method is *intraprocedural* so that different procedures can be processed in parallel.
- Our method is *parametric* in the sense that we introduce a level of mapping from symbolic-execution traces to the word sequences that are input to the word-vector learner. (We call these *abstraction mappings* or *abstractions*, although strictly speaking they are not abstractions in the sense of abstract interpretation (Cousot and Cousot, 1977).) Different abstraction mappings can be used to extract different word sequences that are in different positions on the spectrum of (mainly) syntactic to (mainly) semantic.

We have developed a highly parallelizable toolchain that is capable of producing a parametric encoding of programs to word sequences. For instance, we can process 311,670 procedures in the Linux kernel<sup>1</sup> in 4 hours,<sup>2</sup> using a 64-core workstation (4 CPUs each clocked at 2.3 GHz) running CentOS 7.4 with 252 GB of RAM.

After we present our infrastructure for generating parametric encodings of programs as word sequences (Section 2.2), there are a number of natural research questions that we consider.

First, we explore the utility of embeddings learned from our toolchain:

#### Research Question # 1

Are vectors learned from abstracted symbolic traces encoding *useful* information?

Judging utility is a difficult endeavor. Natural-language embeddings have the advantage of being compatible with several canonical benchmarks for word-similarity prediction or analogy solving (Zweig and Burges, 2011; Finkelstein et al., 2001; Luong et al., 2013; Szumlanski et al., 2013; Hill et al., 2015; Rubenstein and Goodenough, 1965; Mikolov et al., 2013a). In the domain of program understanding, no such canonical benchmarks exist. Therefore, we designed a suite of over nineteen thousand code analogies to aid in the evaluation of our learned vectors.

Next, we examine the impact of different parameterizations of our toolchain by performing an ablation study. The purpose of this study is to answer the following question:

### Research Question # 2

Which abstractions produce the best program encodings for word-vector learning?

There are several examples of learning from syntactic artifacts, such as ASTs or tokens. The success of such techniques raises the question of whether adding

<sup>&</sup>lt;sup>1</sup>We used a prerelease of Linux 4.3 (commit fd7cd061adcf5f7503515ba52b6a724642a839c8 in the GitHub Linux kernel repository).

<sup>&</sup>lt;sup>2</sup>During trace generation, we exclude only vhash\_update, from crypto/vmac.c, due to its size.

a symbolic-execution engine to the toolchain improves the quality of our learned representations.

### Research Question #3

Do abstracted symbolic traces at the semantic end of the spectrum provide more utility as the input to a word-vector-learning technique compared to ones at the syntactic end of the spectrum?

Because our suite of analogies is only a proxy for utility in more complex downstream tasks that use learned embeddings, we pose one more question:

#### Research Question #4

Can we use pre-trained word-vector embeddings on a downstream task?

### 2.1.3 Contributions

We created a toolchain for taking a program or corpus of programs and producing intraprocedural symbolic traces. The toolchain is based on Docker containers, is parametric, and operates in a massively parallel manner. Our symbolic-execution engine prioritizes the amount of data generated over the precision of the analysis: in particular, no feasibility checking is performed, and no memory model is used during symbolic execution.

We generated several datasets of abstracted symbolic traces from the Linux kernel. These datasets feature different parameterizations (abstractions), and are stored in a format suitable for off-the-shelf word-vector learners.

We created a benchmark suite of over 19,000 API-usage analogies.

We report on several experiments using these datasets:

• For RQ1, we tested our approach and found that our learned vectors achieved 93% top-1 accuracy on a suite of over 19,000 analogies we developed.

- For RQ2, we performed an ablation study to assess the effects of different abstractions on the learned vectors; we found that taking away any of the abstractions we've selected reduces performance.
- For RQ3, we found that vectors learned from (mainly) semantic abstractions can provide nearly triple the accuracy of vectors learned from (mainly) syntactic abstractions.
- For RQ4, we learned a model of a specific program behavior (which error a trace is likely to return), and found that we could apply the model in a case study to confirm actual bugs found via traditional static analysis tools.

# 2.2 Overview

Our toolchain consists of three phases: transformation, abstraction, and learning. As input, the toolchain expects a corpus of buildable C projects, a description of abstractions to use, and a word-vector learner. As output, the toolchain produces an embedding of abstract tokens to double-precision vectors with a fixed, user-supplied, dimension. We illustrate this process as applied to the example in Fig. 2.1.

**Phase I: Transformation.** The first phase of the toolchain enumerates all paths in each source procedure. We begin by unrolling (and truncating) each loop so that its body is executed zero or one time(s), thereby making each procedure loop-free at the cost of discarding many feasible traces. We then apply an intraprocedural symbolic executor to each procedure. Fig. 2.2 shows the results of this process as applied to the example code in Fig. 2.1.

**Phase II: Abstraction.** Given a user-defined set of abstractions, the second phase of our toolchain leverages the information gleaned from symbolic execution to generate abstracted traces. One key advantage of performing some kind of abstraction is a drastic reduction in the number of possible tokens that appear in the traces. Consider the transformed trace in Fig. 2.2b. If we want to understand the relationship between allocators and certain error codes, then we might abstract procedure calls as

1	<pre>int example() {</pre>
2	buf = $alloc(12);$
3	
4	if (buf != 0) {
5	<pre>bar(buf);</pre>
6	<pre>free(buf);</pre>
7	return 0;
8	} else {
9	return -ENOMEM;
10	}
11	}

1 call alloc(12); 2 assume alloc(12) != 0; 3 call bar(alloc(12)); 4 call free(alloc(12)); 5 return 0;

(a) Trace 1

```
1 call alloc(12);
2 assume alloc(12) == 0;
3 return -ENOMEM;
```



Figure 2.1: An example procedure

Figure 2.2: Traces from the symbolic execution of the procedure in Fig. 2.1

*parameterized tokens* of the form Called(callee); comparisons of returned values to constants as parameterized RetEq(callee, value) tokens; and returned error codes as parameterized RetError(code) tokens. Fig. 2.3 shows the result of applying these abstractions to the traces from Fig. 2.2.

**Phase III: Learning.** Our abstracted representation discards irrelevant details, flattens control flows into sequential traces, and exposes key properties in the form of parameterized tokens that capture domain information such as Linux error codes. These qualities make abstracted traces suitable for use with a word-vector learner. Word-vector learners place words that appear in similar contexts close together in an embedding space. When applied to natural language, learned embeddings can answer questions such as "Paris is to France as London is to what?" (Answer: England.) Our goal is to learn embeddings that can answer questions such as:

 If a lock acquired by calling spin\_lock is released by calling spin\_unlock, then how should I release a lock acquired by calling mutex\_lock\_nested? That is, Called(spin\_lock) is to Called(spin\_unlock) as Called(mutex\_lock\_nested) is to what?

```
    Called(alloc)
    RetNeq(alloc, 0)
    Called(bar)
    Called(free)
```

```
    Called(alloc)
    RetEq(alloc, 0)
    RetError(ENOMEM)
    4
```

(a) Abstracted Trace 1

(b) Abstracted Trace 2

Figure 2.3: Result of abstracting the two traces in Fig. 2.2b

(Answer: Called(mutex\_unlock).)

 Which error code is most commonly used to report allocation failures? That is, which RetError(code) is most related to RetEq(alloc, 0)?

(Answer: RetError(ENOMEM).)

• Which procedures and checks are most related to alloc?

(Answers: Called(free), RetNeq(alloc, 0), etc.)

The remainder of the chapter describes a framework of abstractions and a methodology of learning embeddings that can effectively solve these problems. Along the way, we detail the challenges that arise in applying word embeddings to abstract path-sensitive artifacts.

# 2.3 Technique

### 2.3.1 Abstractions

One difference between learning from programs and learning from natural language is the size of the vocabulary in each domain. In natural language, vocabulary size is bounded (e.g., by the words in a dictionary, ignoring issues like misspellings). In programs, the vocabulary is essentially unlimited: due to identifier names, there are a huge number of distinct words that can occur in a program. To address the issue of

call foo()	call b	par(foo())	call foo(ob call bar(ob	j) j)
Called(foo)	ParamTc	(bar, foo)	ParamShare(bar,	foo)
return -C $\wedge$ $\mathrm{C} \in$ ERR.	_CODES	return C	$\wedge ~ \mathrm{C} \not\in \texttt{ERR\_CODES}$	return foo()
RetError(ERR_CODES[C]), Error		RetConst(C)		PropRet(foo)
<pre>PropRet(PTR_ERR)</pre>	assume	foo() == 0	obj->foo.ba	r = baz
Error RetEq		(foo, 0)	AccessPathStore	(->foo.bar)

Figure 2.4: Example derivations for selected abstractions

vocabulary size, we perform an abstraction operation on symbolic traces, so that we work with abstracted symbolic traces when learning word vectors from programs.

#### Abstracted Symbolic Traces

We now introduce the set of abstractions that we use to create abstracted symbolic traces. Selected abstractions appear in the conclusions of the deduction rules shown in Fig. 2.4. The abstractions fall into a few simple categories. The Called(callee) and AccessPathStore(path) abstractions can be thought of as "events" that occur during a trace. Abstractions like RetEq(callee, value) and Error serve to encode the "status" of the current trace: they provide contextual information that can modify the meaning of an "event" observed later in the trace. Near the end of the trace, the RetError(code), RetConst(value), and PropRet(callee) abstractions provide information about the result returned at the end of the trace. Taken together, these different pieces of information abstract the trace; however, the abstracted trace is still a relatively rich digest of the trace's behavior.

With the abstractions described above, we found that the learned vectors were sub-optimal. Our investigation revealed that some of the properties we hoped would be learned required leveraging contextual information that was outside the "window" that a word-vector learner was capable of observing. For example, to understand the relationship between a pair of functions like lock and unlock, a word-vector learner must be able to cope with an arbitrary number of words occurring between the functions of interest. Such distances are a problem, because lengthening the history given to a word-vector learner also increases the computational resources necessary to learn good vectors.

Due to the impracticality of increasing the context given to a word-vector learner, we introduced two additional abstractions: ParamTo and ParamShare. These abstractions encode the flow of data in the trace to make relevant contextual information available without the need for arbitrarily large contexts. As shown in Section 2.4.2, the abstractions that encode semantic information, such as dataflow facts, end up adding the most utility to our corpus of abstracted traces. This observation is in line with the results of Allamanis et al. (2017b), who found that dataflow edges positively impact the performance of a learned model on downstream tasks.

We augment the abstractions shown in Fig. 2.4, with the following additional abstractions, which are similar to the ones discussed above:

- RetNeq(callee, value), RetLessThan(callee, value), and others are variants of the RetEq(callee, value) abstraction shown in Fig. 2.4.
- FunctionStart and FunctionEnd are abstractions introduced at the beginning and end of each abstracted trace.
- AccessPathSensitive(path) is similar to AccessPathStore; it encodes any complex field and array accesses that occur in assume statements.

#### **Encoding Abstractions as Words**

We now turn to how the encoding of these abstractions as words and sentences (to form our trace corpus) can impact the utility of learned vectors. To aid the reader's understanding, we use a sample procedure and describe an end-to-end application of our abstractions and encodings.

```
// SAMPLE
                       // TRACE 1
                                                     // TRACE 2
 1
 2
    // PROCEDURE
                       // FunctionStart
                                                     // FunctionStart
 3
    lock(&obj->lock); // Call(lock)
                                                     // Call(lock)
    foo = alloc(12); // Call(alloc)
                                                     // Call(alloc)
 4
 5
    if (foo != 0) {
                       // RetNeg(alloc, 0)
                                                     11 .
 6
      obj->baz =
                       // AccessPathStore(->baz)
                                                     11 .
 7
        bar(foo);
                       // ParamTo(bar,alloc)
                                                     11 .
 8
                       // Call(bar)
                                                     11 .
                       11 .
                                                     // RetEq(alloc, 0)
 9
    } else {
10
      unlock(
                       11 .
                                                     // ParamShare(unlock,lock)
        &obj->lock);
                       11 .
                                                     // Call(unlock)
11
      return -ENOMEM; // .
                                                     // RetError(ENOMEM)
12
13
                       11 .
                                                     // Error
                       11 .
                                                     // FunctionEnd
14
    }
    unlock(
                       // ParamShare(unlock,lock)
15
16
      &obj->lock);
                       // Call(unlock)
    return 0;
                       // RetConst(0)
17
                       // FunctionEnd
18
```

Figure 2.5: Sample procedure with generated abstractions shown as comments

Fig. 2.5 shows a sample procedure along with its corresponding abstractions. The number(s) before each abstraction signify which of the two paths through the procedure the abstraction belongs to. To encode these abstractions as words, we need to make careful choices as to what pieces of information are worthy of being represented as words, and how this delineation affects the questions we can answer using the learned vectors.

For instance, consider the RetNeq(alloc, 0) abstraction. There are several simple ways to encode this information as a sequence of words:

- 1. RetNeq(alloc, 0)  $\implies$  alloc, \$NEQ, 0
- 2. RetNeq(alloc, 0)  $\implies$  alloc, \$NEQ\_0
- 3. RetNeq(alloc, 0)  $\implies$  alloc\_\$NEQ, 0
- 4. RetNeq(alloc, 0)  $\implies$  alloc\_\$NEQ\_0

Each of these four encodings comes with a different trade-off. The first encoding splits the abstraction into several fine-grained words, which, in turn, reduces the size of the overall vocabulary. This approach may benefit the learned vectors because smaller vocabularies can be easier to work with. On the other hand, splitting the information encoded in this abstraction into several words makes some questions more difficult to ask. For example, it is much easier to ask what is most related to alloc being not equal to zero when we have just a single word, alloc\_\$NEQ\_0, to capture such a scenario.

In our implementation, we use the fourth option. It proved difficult to ask interesting questions when the abstractions were broken down into fine-grained words. This decision did come with the cost of a larger vocabulary.<sup>3</sup> Encodings for the rest of our abstractions are shown in Fig. 2.6.<sup>4</sup> The sentences generated by applying these encodings to Fig. 2.5 are shown in Fig. 2.7.

#### 2.3.2 Word2Vec

Word2Vec is a popular method for taking words and embedding them into a lowdimensional vector space (Mikolov et al., 2013a). Instead of using a one-hot encoding where each element of a vector is associated with exactly one word—word2vec learns a denser representation that captures meaningful syntactic and semantic regularities, and encodes them in the cosine distance between words.

For our experiments, we used GloVe (Pennington et al., 2014b) due to its favorable performance characteristics. GloVe works by leveraging the intuition that word-word co-occurrence probabilities encode some form of meaning. A classic example is the relationship between the word pair "ice" and "steam" and the word pair "solid" and "gas." Gas and steam occur in the same sentence relatively frequently, compared to the

<sup>&</sup>lt;sup>3</sup>We mitigate the increase in vocabulary size from constructions like alloc\_ $NEQ_0$  by restricting the constants we look for. Our final implementation only looks for comparisons to constants in the set  $\{-2, -1, 0, 1, 2, 3, 4, 8, 16, 32, 64\}$ .

<sup>&</sup>lt;sup>4</sup>Because it is not possible to have ParamShare(X, Y) or ParamTo(X, Y) without a Called(X) following them, the abstractions ParamShare(X, Y) and ParamTo(X, Y) are encoded as Y to avoid duplicating X.

```
1 match abstraction with
2 | Called (x)
                             -> x
3 | ParamTo (_,x)
                             -> x
  | ParamShare (_,x)
                             -> x
4
                             -> x ^ "_$EQ_" ^ c
   | RetEq (x,c)
5
                             -> x ^ "_$NEQ_" ^ c
   | RetNeq (x,c)
6
   | RetLessThan (x,c)
                             -> x ^ "_$LT_" ^ c
7
                             -> x ^ "_$LTE_" ^ c
   | RetLessThanEq (x,c)
8
   | RetGreaterThan (x,c)
                             -> x ^ "_$GT_" ^ c
9
10 | RetGreaterThanEq (x,c) -> x ^ "_$GTE_" ^ c
                             -> "$RET_" ^ x
   | PropRet (x)
11
                             -> "$RET_" ^ c
12 | RetConst (c)
                             -> "$RET_" ^ ERR_CODES[e]
13 | RetError (e)
   FunctionStart
                             -> "$START"
14
                             -> "$END"
15
   FunctionEnd
                             -> "$ERR"
16 Error
17
   AccessPathStore (p)
                             -> "!" ^ p
18 | AccessPathSensitive (p) -> "?" ^ p
```

Figure 2.6: Encoding of abstractions

\$START lock alloc alloc\_\$NEQ\_0 !->baz alloc bar lock unlock \$RET\_0 \$END

(a) Trace 1

\$START lock alloc alloc\_\$EQ\_0 lock unlock \$ERR \$RET\_ENOMEM \$END

(b) Trace 2

Figure 2.7: Traces for Fig. 2.5 generated by the encoding from Fig. 2.6

frequency with which the words gas and ice occur in the same sentence. Consequently, the following ratio is significantly less than 1:

$$\frac{\Pr(\text{gas} \mid \text{ice})}{\Pr(\text{gas} \mid \text{steam})}$$

If, instead, we look at the frequency of sentences with both solid and ice compared to the frequency of sentences with both solid and steam, we find the opposite. The ratio

$$\frac{\Pr(\text{solid} \mid \text{ice})}{\Pr(\text{solid} \mid \text{steam})}$$

is much greater than 1. This signal is encoded into a large co-occurrence matrix. GloVe then attempts to learn word vectors for which the dot-product of two vectors is close to the logarithm of their probability of co-occurrence.

# 2.4 Experiments

### 2.4.1 RQ1: Are Learned Vectors Useful?

Research Question 1 asked whether vectors learned from abstracted symbolic traces encode useful information. We assess utility via three experiments over word vectors. Each of the following subsections describes and interprets one experiment in detail.

#### **Experiment 1: Code Analogies**

An interesting aspect of word vectors is their ability to express relationships between analogous words using simple math and cosine distance. Encoding analogies is an intriguing byproduct of a "good" embedding and, as such, analogies have become a common proxy for the overall quality of learned word vectors.

No standard test suite for *code* analogies exists, so we created such a test suite using a combination of manual inspection and automated search. The test suite consists of twenty different categories, each of which has some number of function pairs that have been determined to be analogous. For example, con-
Type	Category	# Pairs	Passing Tests	Total Tests	Accuracy
Calls	16 / 32	18	246	306	80.39%
Calls	Add / Remove	9	72	72	100.00%
Calls	Create / Destroy	19	302	342	88.30%
Calls	Enable / Disable	62	$3,\!577$	3,782	94.58%
Calls	Enter / Exit	12	122	132	92.42%
Calls	In / Out	5	20	20	100.00%
Calls	Inc / Dec	10	88	90	97.78%
Calls	Input / Output	5	20	20	100.00%
Calls	Join / Leave	4	8	12	66.67%
Calls	Lock / Unlock	53	2,504	2,756	90.86%
Calls	On / Off	19	303	342	88.60%
Calls	Read / Write	64	$3,\!950$	4,032	97.97%
Calls	Set / Get	22	404	462	87.45%
Calls	Start / Stop	31	838	930	90.11%
Calls	Up / Down	24	495	552	89.67%
Complex	Ret Check / Call	21	252	420	60.00%
Complex	Ret Error / Prop	25	600	600	100.00%
Fields	Check / Check	50	2,424	$2,\!450$	98.94%
Fields	Next / Prev	16	240	240	100.0%
Fields	Test / Set	39	1,425	1,482	96.15%
Totals:		508	17,890	$19,\!042$	93.95%

Table 2.1: Analogy Suite Details

sider mutex\_lock\_nested/mutex\_unlock and spin\_lock/spin\_unlock; these are two pairs from the "lock / unlock" category given in Table 2.1. We can construct an analogy by taking these two pairs and concatenating them to form the analogy "mutex\_lock\_nested is to mutex\_unlock as spin\_lock is to spin\_unlock." By identifying high-level patterns of behavior, and finding several pairs of functions that express this behavior, we created a suite that contains 19,042 code analogies.

Table 2.1 lists our categories and the counts of available pairs (a representative pair from each category can be found in Chapter A). Table 2.1 also provides accuracy metrics generated using the vectors learned from what we will refer to as the "baseline configuration,"<sup>5</sup> which abstracts symbolic traces using all of the abstractions described in in Section 2.3.1. We used a grid-search over hundreds of parameterizations to pick hyper-parameters for our word-vector learner. For the results described in this section, we used vectors of dimension 300, a symmetric window size of 50, and a vocabulary-minimum threshold of 1,000 to ensure that the word-vector learner only learns embeddings for words that occur a reasonable number of times in the corpus of traces. We trained for 2,000 iterations to give GloVe ample time to find good vectors.

In each category, we assume that any two pairs of functions are sufficiently similar to be made into an analogy. More precisely, we form a test by selecting two distinct pairs of functions (A, B) and (C, D) from the same category, and creating the triple (A, B, C) to give to an analogy solver that is equipped with our learned vectors. The analogy solver returns a vector D', and we consider the test passed if D' = D and failed otherwise. Levy and Goldberg (2014) present the following objective to use when solving analogies with word-vectors:

$$\mathrm{D}' = \mathop{\mathrm{arg\,max}}_{d \in \mathbb{V} \setminus \{A,B,C\}} \cos(d,B) - \cos(d,A) + \cos(d,C)$$

**Results.** The "Accuracy" column of Table 2.1 shows that overall accuracy on the analogy suite is excellent. Our embeddings achieve greater than 90% top-1 accuracy on thirteen out of the twenty categories. The learned vectors do the worst on the "Ret Check / Call" category where the top-1 accuracy is only 60%. This category is meant to relate the checking of the return value of a call with the call itself. However, we often find that one function allocates memory, while a different function checks for allocation success or failure. For example, a wrapper function may allocate complex objects, but leave callers to check that the allocation succeeds. Because our vectors are derived from intraprocedural traces, it is sensible that accuracy suffers for interprocedural behaviors.

By contrast, our vectors perform extraordinarily well on the "Ret Error / Prop" category (100% top-1). This category represents cases where an outer function (i)

<sup>&</sup>lt;sup>5</sup>The baseline configuration is described in more detail in Section 2.4.2, where it is also called configuration (C1.).

```
ret = new(/*...*/, &priv->bo);
1
2
    if (!ret) {
      ret = pin(priv->bo, /*...*/);
3
      if (!ret) {
4
5
        ret = map(priv->bo);
6
        if (ret)
7
           unpin(priv->bo);
8
      }
      if (ret)
9
        ref(NULL, &priv->bo);
10
    }
11
```

Figure 2.8: Excerpt from nv17\_fence.c. Names have been shortened to conserve space.

performs an inner call, (ii) detects that it has received an error result, and (iii) returns ("propagates") that error result as the outer function's own return value. Unlike for the "Ret Check / Call" category, the nature of the "Ret Error / Prop" category ensures that both the check and the return propagation can be observed in intraprocedural traces, without losing any information.

### **Experiment 2: Simple Similarity**

One of the most basic word-vector tasks is to ask for the k nearest vectors to some chosen vector (using cosine distance). We expect the results of such a query to return a list of relevant words from our vocabulary. Our similarity experiments were based on two types of queries: (i) given a word, find the closest word, and (ii) given a word, find the five closest words.

**Similar pairs.** We identified the single most similar word to each word in our vocabulary  $\mathbb{V}$ . This process produced thousands of interesting pairs. In the interest of space, we have selected four samples which are representative of the variety of high-level relationships encoded in our learned vectors:

• sin\_mul and cos\_mul

- dec\_stream\_header and dec\_stream\_footer
- rx\_b\_frame and tx\_b\_frame
- nouveau\_bo\_new\_\$EQ\_0 and nouveau\_bo\_map <sup>6</sup>

The last pair is of particular interest, because it expresses a complex pattern of behavior that would be impossible to encode without some abstraction of the path condition. The last pair suggests that there is a strong relationship between the function new returning  $\emptyset$  (which signals a successful call) and then the subsequent performance of some kind of map operation with the map call. To gain a deeper understanding of what the vectors are encoding, we searched for instances of this behavior in the original source code. We found several instances of the pattern shown in Fig. 2.8.

The code in Fig. 2.8 raise a new question: why isn't pin more closely related to new\_\$EQ\_0? We performed additional similarity queries to gain a deeper understanding of how the learned vectors have encoded the relationship between new, pin, and map.

First, we checked to see how similar pin is to new\_\$EQ\_0. We found that pin is the fourth-most related word to new\_\$EQ\_0, which suggests that a relationship does exist, but that the relationship between new\_\$EQ\_0 and pin is not as strong as the one between new\_\$EQ\_0 and map. Looking back at the code snippet (and remembering that several more instances of the same pattern can be found in separate files), we are left with the fact that pin directly follows from the successful new. Therefore, intuition dictates that pin should be more strongly related to new than map. The disagreement between our intuition and the results of our word-vector queries motivated us to investigate further.

By turning to the traces for an answer, we uncovered a more complete picture. In 3,194 traces, new co-occurs with pin. In 3,145 traces, new co-occurs with map. If we look at traces that do *not* contain a call to new, there are 11,354 traces that have no call to new, but still have a call to pin. In contrast, only 352 traces have no call to new, but still have a call to map. Finally, we have a definitive answer to the encoding

<sup>&</sup>lt;sup>6</sup>In the following text, and in Fig. 2.8, we remove the nouveau\_bo\_ prefix to conserve space.

learned by the vectors: it is indeed the case that **new** and **map** are more related in our corpus of traces, because almost every time a call to **map** is made, a corresponding call to **new** precedes it. Our intuition fooled us, because the snippets of source code only revealed a partial picture.

Top-5 similar words and the challenge of prefix dominance. Another similaritybased test is to take a word and find the top-k closest words in the learned embedding space. Ideally, we'd see words that make intuitive sense. For the purpose of evaluation, we picked two words: affs\_bread, a function in the AFS file system that reads a block, and kzalloc, a memory allocator. For each target word, we evaluated the top-5 most similar words for relevance. In the process, we also uncovered an interesting challenge when learning over path-sensitive artifacts, which we call *prefix dominance*.

Our corpus of symbolic traces can be thought of as a corpus of execution trees. In fact, in the implementation of our trace generator, the traces only exist at the very last moment. Instead of storing traces, we store a tree that encodes, without unnecessary duplication, the information gained from symbolically executing a procedure. If we think about the dataset of traces as a dataset of trees (each of which holds many traces that share common prefixes), we begin to see that learning word vectors from traces is an approximation of learning directly from the execution trees.

The approximation of trees by traces works, in the sense that we can use the traces to learn meaningful vectors, but the approximation is vulnerable to learning rare behaviors that exist at the beginning of a procedure whose trace-tree has many nested branches. These rare behaviors occur only once in the original procedure text and corresponding execution tree, but are replicated many times in the traces. In a procedure with significant branching complexity, a single occurrence of rare behavior can easily overwhelm any arbitrary number of occurrences of expected behavior.

In Table 2.2, we see two words, affs\_bread and kzalloc, and the five most similar words to each of them. Word similarity has captured many expected relationships. For example, the fact that kzalloc is most commonly checked to be non-null (kzalloc\_\$NEQ\_0) and then also kfreed is what we would expect, given the definition of an allocator. Similarly, we can see that affs\_bread is also checked to be

affs_bread	kzalloc
affs_bread_\$NEQ_0	kzalloc_\$NEQ_0
affs_checksum_block	kfree
AFFS_SB	_volume
affs_free_block	<pre>snd_emu10k1_audigy_write_op</pre>
affs_brelse	?->output_amp

Table 2.2: Top-5 closest words to affs\_bread and kzalloc

non-null, check-summed, freed, released, etc. However, in addition to these expected relationships, the last three entries for kzalloc seem out of place. These unexpected entries are present in the top-5 answer because of prefix dominance.

We searched our traces for places where kzalloc and the three unexpected entries in the table co-occur. We found one function with 5,000 paths (5,000 being our "budget" for the number of traces we are willing to generate via symbolic execution for a single procedure), of which 4,999 have several instances of the pattern kzalloc followed by snd\_emu10k1\_audigy\_write\_op. This one function, with its multitude of paths, overwhelms our dataset, and causes the word vectors to learn a spurious relationship. Prefix dominance also explains the strong associations between kzalloc and \_volume and ?->output\_amp.

On the other hand, affs\_bread is relatively unaffected by prefix dominance. Examining our traces for the affs file system that contains this function, we found that no procedures had an overwhelming number of paths. Therefore, we never see an overwhelming number of affs\_bread usage patterns that are rare at the source level but common in our set of traces.

### **Experiment 3: Queries Via Word-Vector Averaging**

Word vectors have the surprising and useful ability to encode meaning when averaged (Le and Mikolov, 2014; Kenter et al., 2016). We devised a test to see if our learned vectors are able to leverage this ability to capture a relationship between allocation failure and returning -ENOMEM. To understand whether our word vectors are capable of answering such a high-level question, we evaluated their performance on increasingly targeted queries (represented by averaged vectors). Each query was restricted to search only for words in the subspace of the embedding space that contains kernel error-codes. (Narrowing to the subspace of error codes ensures that we are only looking at relevant words, and not at the whole vocabulary.)

**Results.** We identified twenty different functions that act as allocators in the Linux kernel.

First, for each such allocator, we took its word vector A, and queried for the closest vector to A (in the subspace of error codes). This method found the correct error code only twice out of twenty tests (i.e., 10% accuracy).

Second, we asked for the vector closest to an average vector that combined the vector for the allocator A of interest and the vector  $\overrightarrow{\text{SERR}}$  for a generic error:<sup>7</sup>  $(A + \overrightarrow{\text{SERR}})/2$ . This query found the correct ENOMEM code fourteen times out of twenty (i.e., 70% accuracy).

Third, instead of averaging the allocator's A vector with  $\overline{\$ERR}$ , we tried averaging A with the vector for the special \$END token that signals the end of a trace. Seeking the error code closest to  $(A + \overline{\$END})/2$  found the correct result for sixteen of twenty test cases (i.e., 80% accuracy). The fact that this method outperforms our previous query reveals that the call to an allocator being near the end of a trace is an even stronger signal than the \$ERR token.

Finally, we mixed the meaning of the allocator, the error token, and the end-oftrace token by averaging all three:  $(A + \overline{\$ERR} + \overline{\$END})/3$ . The error code whose vector is closest to this query is the correct ENOMEM code for eighteen of the twenty tests (i.e., 90% accuracy). The steadily increasing performance indicates that targeted queries encoded as average word vectors can indeed be semantically meaningful.

The effectiveness of these queries, and the results from Section 2.4.1, support a positive answer to Research Question 1: learned vectors do encode useful information

<sup>&</sup>lt;sup>7</sup>The \$ERR word is added to any trace that returns either (i) the result of an ERR\_PTR call, or (ii) a constant less than zero that is also a known error code. Consequently, a vector \$ERR is learned for the word \$ERR.

about program behaviors.

## 2.4.2 RQ2: Ablation Study

In this section, we present the results of an ablation study to isolate the effects that different sets of abstractions have on the utility of the learned vectors. We used the benchmark suite of 19,042 code-analogies from Section 2.4.1 to evaluate eight different configurations. We scored each configuration according to the number of analogies correctly encoded by the word vectors learned for that configuration (i.e., we report top-1 results).

In addition to the baseline configuration from Section 2.4.1, we partitioned the abstractions into six classes<sup>8</sup> and generated six new embeddings, each with one of the six abstraction classes excluded. We also used one more configuration in which *stop words* were included. In natural language processing, stop words are words that are filtered out of a processing toolchain. Sometimes these are the most common words in a language, but any group of words can be designated as stop words for a given application. In our context, stop words are function names that occur often, but add little value to the trace. Examples are \_\_builtin\_expect and automatically generated \_\_compiletime\_asserts.

We evaluated the following eight configurations:

- C1. **baseline**: all abstractions from Section 2.3.1;
- C2. baseline without ParamTo and ParamShare;
- C3. baseline without RetEq, RetNeq, etc.;
- C4. baseline without AccessPathStore and AccessPathSensitive;
- C5. baseline without PropRet, RetError, and RetConst;
- C6. baseline without Error;

<sup>&</sup>lt;sup>8</sup>Except for Called, which was used in all configurations.



Figure 2.9: Ablation study: top-1 analogy results for eight configurations (baseline (C1.) with up to one individual abstraction class removed). The vocabulary minimum was 0, and the number of training iterations was 1,000.

### C7. baseline without FunctionStart and FunctionEnd; and

C8. baseline with stop words included.

Fig. 2.9 compares the accuracy of for these eight configurations. Purple bars indicate the number of tests in the analogy suite that passed; orange indicates tests that failed; and white indicates out-of-vocabulary (OOV) tests. Configuration (C4.) had the most out-of-vocabulary tests; in this configuration, we do not have words like **!->next** and **!->prev**, which leaves several portions of the analogy suite essentially unanswerable. Thus, we count out-of-vocabulary tests as failed tests.

To create a fair playing field for evaluating all eight configurations, we chose a single setting for the hyper-parameters that were used when learning word vectors. We reduced the threshold for how often a word must occur before it is added to the vocabulary from 1,000 to 0. The latter parameter, which we refer to as the *vocabulary minimum*, significantly impacts performance by forcing the wordvector learner to deal with thousands of rarely-seen words. To understand why we must set the vocabulary minimum to zero, effectively disabling it, consider the following example trace: Called(foo), ParamShare(foo, bar), Called(bar). In configuration (C2.), where we ignore ParamShare, we would encode this trace as the sentence foo bar. In configuration (C1.), this same trace is encoded as foo foo bar. The fact that some abstractions can influence the frequency with which a word occurs in a trace corpus makes any word-frequency-based filtering counterproductive to our goal of performing a fair comparison.

We also lowered the number of training iterations from 2,000 to 1,000 to reduce the resources required to run eight separate configurations of our toolchain. (These changes are responsible for the change in the top-1 accuracy of the baseline configuration from 93.9% in Table 2.1 to 85.8% in Fig. 2.9.)

In Fig. 2.9, one clearly sees that configuration (C2.) (the one without any dataflowbased abstractions) suffers the worst performance degradation. Configuration (C4.), which omits access-path-based abstractions, has the second-worst performance hit. These results indicate that dataflow information is critical to the quality of learned vectors. This conclusion further confirms findings by Allamanis et al. (2017b) regarding the importance of dataflow information when learning from programs.

Fig. 2.9 also reveals that removing "state" abstractions (RetEq, RetNeq, etc. and Error) has little effect on quality. However, these abstractions still add useful terms to our vocabulary, and thereby enlarge the set of potentially answerable questions. Without these abstractions, some of the questions in Section 2.4.1 would be unanswerable.

These results support the following answer to Research Question 2: dataflow-based abstractions provide the greatest benefit to word-vector learning. These abstractions, coupled with access-path-based abstractions, provide sufficient context to let a wordvector learner create useful embeddings. Adding abstractions based on path conditions (or other higher-level concepts like Error) adds flexibility without worsening the quality of the learned vectors. Therefore, we recommend including these abstractions, as well.

### 2.4.3 RQ3: Syntactic Versus Semantic

Now that we have seen the utility of the generated corpus for word-vector learning (Section 2.4.1) and the interplay between the abstractions we use (Section 2.4.2), we compare our recommended configuration (C1.) from Section 2.4.2 with a simpler syntactic-based approach.

We explored several options for a syntactic-based approach against which to compare. In trying to make a fair comparison, one difficulty that arises is the amount of data our toolchain produces to use for the semantics-based approach. If we were to compare configuration (C1.) against an approach based on ASTs or tokens, there would be a large disparity between the paucity of data available to the AST/tokenbased approach compared to the abundance of data available to the word-vector learner: an AST- or token-based approach would only have one data point per procedure, whereas the path-sensitive artifacts gathered using configuration (C1.) provide the word-vector learner with hundreds, if not thousands, of data points per procedure.

To control for this effect and avoid such a disparity, we instead compared configuration (C1.) against a configuration of our toolchain that uses only "syntactic" abstractions—i.e., abstractions that can be applied without any information obtained from symbolic execution. Thus, the syntactic abstractions are:

- FunctionStart and FunctionEnd,
- AccessPathStore(path), and
- Called(callee).

The rest of our abstractions use deeper semantic information, such as constant propagation, dataflow information, or the path condition for a given trace.

Using only the syntactic abstractions, we generated a corpus of traces, and then learned word vectors from the corpus. We compared the newly learned word vectors



Figure 2.10: Top-1 analogy results for syntactic versus semantic abstractions. (The vocabulary minimum was 0, and the number of training iterations was 1,000.)

to the ones obtained with configuration (C1.). Fig. 2.10 clearly shows that semantic abstractions are crucial to giving the context necessary for successful learning. Even if we assess performance using only the analogies that are in-vocabulary for the syntactic-based approach, we find that the syntactic-based approach achieves only about 44% accuracy, which is *about half* the accuracy of vectors learned from (mainly) semantic abstractions.

These results support an affirmative answer to Research Question 3: abstracted traces that make use of semantic information obtained via symbolic execution provide more utility as the input to a word-vector learner than abstracted traces that use only syntactic information.

## 2.4.4 RQ4: Use in Downstream Tasks

Research Question 4 asks if we can utilize our pre-trained word-vector embeddings on some downstream task.

To address this question, we selected a downstream task that models bug finding, repair, and code completion in a restricted domain: error-code misuse. We chose error-code misuse because it allows us to apply supervised learning. Because there are only a finite number of common error codes in the Linux kernel, we can formulate a multi-class labeling problem using traces generated via our toolchain and our pre-trained word-vector embeddings.

To build an effective error-code-misuse model, we gathered a collection of failing traces (traces in which the \$ERR token occurs). We then constructed a dataset suitable for supervised learning as follows: we took each trace from configuration (2)<sup>9</sup> and removed the last three abstract tokens, namely, \$ERR, \$RET\_E\*, and \$END;<sup>10</sup> we used the \$RET\_E\* token as the label for the trimmed trace. We sampled a subset of 20,000 traces from this large trace collection to use for training our model.

This dataset is a good starting point, but feeding it to a machine-learning technique that accepts fixed-length inputs requires further processing. To preprocess the data, we kept only the last 100 tokens in each trace. We then took the trimmed traces, and used our learned word-vector embedding to transform each sequence of words into a sequence of vectors (of dimension 300). If, originally, a trace had fewer than 100 tokens, we padded the beginning of the trace with the zero vector. We paired each of the trimmed and encoded traces with its label (which we derived earlier). Finally, to complete the preprocessing of the dataset we attached a one-hot encoding of the label.

To collect a challenging test set to evaluate our learned model, we turned to real bug-fixing commits applied to the Linux kernel. We searched for commits that referenced an "incorrect return" in their description. In addition, we leveraged Min et al.'s (Min et al., 2015) list of incorrect return codes fixed by their JUXTA tool. Next, we generated abstracted symbolic traces both before applying the fixing commit and after. Finally, we kept the traces generated before applying the fix that, after the fix, had changed only in the error code returned. Using this process, we collected 68 traces—from 15 unique functions—that had been patched to fix an incorrect return code.

Using the preprocessed dataset, we trained a model to predict the error code that each trace should return. We used a recurrent neural network with long short-term

<sup>&</sup>lt;sup>9</sup>The dataflow abstractions present in (1) were created to aid word-vector learners; for this experiment, we use configuration (2) to exclude those abstractions.

<sup>&</sup>lt;sup>10</sup>We exclude traces that included the **\$RET\_PTR\_ERR** token because these traces do not have an associated error code.

memory (LSTM) (Hochreiter and Schmidhuber, 1997). We evaluated the trained model, using our test set, in two different ways:

- Bug Finding: we use our learned model to predict the three most likely error codes for each trace in our test set. If a given trace initially ended in the error code A, but was patched to return the error code B, we check to see if the incorrect A error code is absent from our model's top-3 predictions.
- Repair / Suggestion: we again use the learned model to predict the three most likely error codes for each trace in the test set. This time, we determine the fraction of traces for which the correct error code (i.e., B) is present in the top-3 prediction made by the model.

In evaluation (1), we found that the learned model identified an incorrect error code in 57 of our 68 tests. This result is promising, because it suggests that there is enough signal in the traces of encoded vectors to make good predictions that could be used to detect bugs early.

In evaluation (2), we observed that the learned model had a top-3 accuracy of 76.5%, meaning that the correct error code is among our top suggested fixes for more than three fourths of the buggy traces. This result is a strong indicator that the learned vectors and abstracted symbolic traces are rich enough to make high-level predictions that could be used to augment traditional IDEs with predictive capabilities. Such a feature could operate like autocomplete, but with an awareness of what other contributors have done and how their (presumably correct) code should influence new contributions. This feature would be similar to the existing applications of statistical modeling to programming tasks such as autocompletion (Allamanis et al., 2015a; Raychev et al., 2015; Bielik et al., 2016; Nguyen et al., 2013, 2012).

These results support an affirmative answer to Research Question 4: our pretrained word-vector embeddings can be used successfully on downstream tasks. These results also suggest that there are many interesting applications for our corpus of abstracted symbolic traces. Learning from these traces to find bugs, detect clones, or even suggest repairs, are all within the realm of possibility.

# 2.5 Related Work

Recently, several techniques have leveraged learned embeddings for artifacts generated from programs. Nguyen et al. (2017b, 2016) leverage word embeddings (learned from ASTs) in two domains to facilitate translation from Java to C#. Pradel and Sen (2017) use embeddings (learned from custom tree-based contexts built from ASTs) to bootstrap anomaly detection against a corpus of JavaScript programs. Gu et al. (2016) leverage an encoder/decoder architecture to embed whole sequences in their DEEPAPI tool for API recommendation. API2API by Ye et al. (2016a) also leverages word embeddings, but it learns the embeddings from API-related natural-language documents instead of an artifact derived directly from source code.

Moving toward more semantically rich embeddings, DeFreez et al. (2018a) leverage labeled pushdown systems to generate rich traces which they use to learn function embeddings. They apply these embeddings to find function synonyms, which can be used to improve traditional specification mining techniques. Alon et al. (2018c) learn from paths through ASTs to produce general representations of programs; in (Alon et al., 2018b) they expand upon this general representation by leveraging attention mechanisms. Ben-Nun et al. (2018) utilize an intermediate representation (IR) to produce embeddings of programs that are learned from both control flow and data flow information.

Venturing into general program embeddings, there are several recent techniques that approach the problem of embedding programs (or, more generally, symbolicexpressions/trees) in unique ways. Using input/output pairs as the input data for learning, Piech et al. (2015) and Parisotto et al. (2016) learn to embed whole programs. Using sequences of live variable values, Wang et al. (2017) produce embeddings to aid in program repair tasks. Allamanis et al. (2017b) learn to embed whole programs via Gated Graph Recurrent Neural Networks (GG-RNNs) (Li et al., 2015). Allamanis et al. (2016a) approach the more foundational problem of finding continuous representations of symbolic expressions. Mou et al. (2016) introduce tree-based convolutional neural networks (TBCNNs), another model for embedding programs. Peng et al. (2015) provide an AST-based encoding of programs with the goal of facilitating deep-learning methods. Allamanis et al. (2018) give a comprehensive survey of these techniques, and many other applications of machine learning to programs.

We are not aware of any work that attempts to embed traces generated from symbolic execution. On the contrary, Fowkes and Sutton (2016) warn of possible difficulties learning from path-sensitive artifacts. We believe that our success in using symbolic traces as the input to a learner is due to the addition of path-condition and dataflow abstractions—the extra information helps to ensure that a complete picture is seen, even in a path-sensitive setting.

In the broader context of applying statistical NLP techniques to programs, there has been a large body of work using language models to understand programs (Hindle et al., 2012; Raychev et al., 2014a; Nguyen and Nguyen, 2015; Allamanis et al., 2015b, 2014); to find misuses (Murali et al., 2017a; Wang et al., 2016); and to synthesize expressions and code snippets (Gvero and Kuncak, 2015; Raghothaman et al., 2016).

# 2.6 Future Work

The work in this chapter occurred well before Transformer-based models began to dominate learning from sequences of text. It would be interesting to revisit the abstracted symbolic traces we are capable of generating and see what Transformerbased models are capable of doing with this data. Futhermore, it would be interesting to apply recent techniques to understand how a model trained on such traces makes its predictions; for example, a technique such as Integrated Gradients could be used to understand better what tokens in a trace a model was attending to while making a prediction. Data related to what a model attends to during predictions might be useable in some sort of (concrete) pattern-extraction algorithm.

# 2.7 Notes

This was the first work I produced during my graduate studies; in many ways this was also the must difficult work of my graduate career as, at the time, I was just beginning to understand where the fields of Machine Learning, Software Engineering,

40

and Programming Languages were headed. In hindsight, almost five years later, it is clear to me that the idea of *better representations for learning from code* has lost out to simply applying ever-increasing amounts of compute to simpler representations. I believe that the utility of better representations will remain an open question for some time (and the question largely hinges on our ability to apply similarly massive compute to these more sophisticated representations). It is interesting to note how the work I did here is analogous to an emerging area of work labelled "prompt engineering"—here, I tried to make a semantic representation of programs more amenable to popular (at the time) off-the-shelf learning algorithms. Now, we have these massive jack-of-all-trades models and the question is not in how to represent our data; the question is in how to form the appropriate *prompt* so that we can coax the model into providing the answers we seek. These two problems are related, but it is quite interesting to see such a drastic shift in focus over such a (relatively) short time period. Can we automatically infer the correct usage patterns for a given API by learning from a large corpus of existing code that references the API under study?

GPT-3: Yes, it is possible to automatically infer the correct usage patterns for a given API by learning from a large corpus of existing code that references the API under study. However, it is important to note that this process is not always accurate, and there is a possibility that some incorrect usage patterns may be learned.

How would you do such inference? Could you use clustering and code embeddings?

GPT-3: Yes, it is possible to use clustering and code embeddings to infer the correct usage patterns for a given API. By clustering together code that uses the API in similar ways, it should be possible to identify common usage patterns. Additionally, by creating code embeddings, it should be possible to identify relationships between different pieces of code that use the API, which could also help to identify common usage patterns.

A dialogue with OpenAI's GPT-3

# **3.1** Introduction

In Chapter 2, we introduced a way to learn from code using off-the-shelf word-vector learners and abstracted traces extracted (statically) from programs. Given this technique, one may wonder how we can *leverage* the vectors we learned to do useful work in classic problem domains. One such problem domain is specification mining. In this chapter, we will look at the problem of mining specifications (or usage patterns) from large software systems. In particular, we will leverage the tools and techniques introduced in Chapter 2 to develop a novel technique for specification mining.

## 3.1.1 Motivation

The continued growth of software in size, scale, scope, and complexity has created an increased need for code reuse and encapsulation. To address this need, a growing number of frameworks and libraries are being authored. These frameworks and libraries make functionality available to downstream users through Application Programming Interfaces (APIs). Although some APIs may be simple, many APIs offer a large range of operations over complex structures.

Staying within correct usage patterns can require domain-specific knowledge about the API and its idiosyncratic behaviors (Robillard and DeLine, 2011). This burden is often worsened by insufficient documentation and explanatory materials for a given API. The difficulty of conforming to distinct (and often implicit) usage patterns for every API raises a key question:

### Motivating Question

Can we automatically *infer* the correct usage patterns for a given API by *learning* from a large corpus of existing code that references the API under study?

## 3.1.2 Goals

In this chapter, we contribute to the study of API-usage mining by identifying a new problem area and exploring the combination of machine learning and traditional methodologies to address the novel challenges that arise in this new domain. Specifically, we introduce the problem domain of Open-World Specification Mining. The goal of Open-World Specification Mining can be stated as follows:

### Goals

Given noisy traces, automatically identify and mine patterns or specifications without the aid of (i) implicit or explicit groupings of terms, (ii) pre-defined pattern templates, or (iii) user-directed feedback or intervention. The idea of Open-World Specification Mining is motivated by the lack of adoption of specification-mining tools outside of the research community. Since Open-World Specification Mining needs no user-supplied input, we believe it will lead to tools that are easier to transition and apply in industry. Although this Open-World setting reduces the burden imposed on users, it increases the challenges associated with extracting patterns. We address these challenges with a toolchain, called ml4spec:

- We base our technique on a form of intraprocedural, parametric, lightweight symbolic execution introduced in Chapter 2.
- To address the lack of implicit or explicit groupings of terms (a challenge imposed by the setting of Open-World Specification Mining), we introduce a technique, *Domain-Adapted Clustering* (DAC), that is capable of recovering groupings of related terms.
- Finally, we remove the need for pre-defined pattern templates by mining specifications using traditional, unrestricted, methods (such as k-Tails (Biermann and Feldman, 1972) and Hidden Markov Models (Seymore et al., 1999)). We are able to use these traditional methods by leveraging Domain-Adapted Clustering to "focus" these traditional methods toward interesting patterns.

The combination of both traditional techniques and machine-learning-assisted methods in the pursuit of Open-World Specification Mining raises a number of natural research questions that we consider.

First, we explore the ability of Domain-Adapted Clustering, our key technique, to successfully extract informative and useful clusters of API methods in our Open-World setting:

## Research Question # 1

Can we effectively mine useful clusters of API methods in an Open-World setting?

Immediately, we run into the difficulty of judging the utility of clusters extracted from traces. To provide the basis for a consistent and quantitative evaluation, we have manually extracted a dataset of ground-truth clusters from five popular open-source projects written in C.

Next, we compare Domain-Adapted Clustering (DAC) against several other baselines that do not utilize the implicit structure of the extracted traces:

## Research Question # 2

How does DAC compare to off-the-shelf clustering techniques?

We also explore how two key choices in our toolchain impact the overall utility of our results:

### Research Question # 3

How does the choice of word-vector learner and the choice of sampling technique affect the resulting clusters?

Finally, to quantify the usefulness of unsupervised learning in our approach, and to validate our central hypothesis, we ask:

## Research Question # 4

Is there a benefit from using a combination of co-occurrence statistics and word embeddings?

## 3.1.3 Contributions

We defined a new problem domain called Open-World Specification Mining. Our motivation is to increase the adoption of specification-mining techniques by reducing the burden imposed on users (at the cost of a more challenging mining task).

We created a toolchain based on the key insight that unsupervised learning (specifically word embeddings) can be combined with traditional metrics to enable automated mining in an Open-World setting.

We introduced a benchmark of 71 ground-truth clusters extracted from five open-source C projects.

### We report on several experiments:

- In Section 3.4.2, we use our toolchain to recover, on average, two thirds of the ground-truth clusters in our benchmark automatically.
- In Section 3.4.3, we compare our Domain-Adapted Clustering technique to three off-the-shelf clustering algorithms; Domain-Adapted Clustering provides, on average, a 30% performance increase relative to the best baseline.
- In Section 3.4.4, we confirm our intuition that sub-word information improves the quality of learned vectors in the software-engineering domain; we also confirm that our Diversity Sampling technique (Section 3.3.2) increases performance by solving the problem of *prefix dominance* (Section 2.4.1).
- In Section 3.4.5, we quantify the impacts of our learning-assisted approach.

# 3.2 Overview

The ml4spec toolchain consists of five phases: Parametric Lightweight Symbolic Execution, thresholding and sampling, unsupervised learning, Domain-Adapted Clustering, and mining. As input, ml4spec expect a corpus of buildable C projects. As output, ml4spec produces clusters of related terms and finite-state automata (FSAs) or Hidden Markov Models (HMMs) mined via traditional techniques.<sup>1</sup> A visualization of the way data flows through the ml4spec toolchain is given in Fig. 3.1. We illustrate this process as applied to the example in Fig. 3.2.

<sup>&</sup>lt;sup>1</sup>Although we provide examples based on traditional miners that produce FSAs and HMMs, we note that ml4spec is miner-agnostic. By using ml4spec as a trace pre-processor, any trace-based miner can be adapted to the Open-World setting.



Figure 3.1: Overview of the ml4spec toolchain

**Phase I:** Parametric Lightweight Symbolic Execution. The first phase of the ml4spec toolchain applies *Parametric Lightweight Symbolic Execution* (PLSE). PLSE takes, as input, a corpus of buildable C projects and a set of abstractions to apply. For our use case, we abstract calls, checks on the results of calls, and return values. Section 3.3.1 describes these abstractions in more detail. Figure 3.2 presents both an example procedure and a trace resulting from the application of PLSE. Already, examining Fig. 3.2b, we can see one of the core challenges of Open-World mining: the mixed vocabulary present in the trace from Fig. 3.2b involves many interesting behaviors but, without user input (Ammons et al., 2002; Lo and Khoo, 2006), pre-defined rule templates (Yun et al., 2016), or some pre-described notion of what methods are related (Le and Lo, 2018), we have no straightforward route to separating patterns from noise. We need to *disentangle* these disparate behaviors to facilitate better specification mining.

**Phase II: Thresholding and Sampling.** Although our example procedure has a small number of paths from entry to exit, many procedures have thousands of possible paths. Learning from these traces can be challenging due to the number of times the same trace prefix is seen. This problem, labelled *prefix dominance* (introduced in Section 2.4.1), makes downstream learning tasks more challenging. For instance, some terms that occur in multiple traces (e.g., in a common prefix) may occur only a single time in the source program. Off-the-shelf word-vector learners cannot filter for these kinds of rare words because they have no concept of the implicit hierarchy between traces and the procedures they were extracted from. The **ml4spec** toolchain introduces two novel techniques to address these challenges: Diversity Sampling and Hierarchical Thresholding. Diversity Sampling attempts to recover a fixed number of highly representative traces via a metric-guided sampling process. Hierarchical Sampling leverages the implicit hierarchy between procedures and traces to remove rare words. Together, these techniques improve the quality of downstream results.

**Phase III: Unsupervised Learning.** Traditionally, specification and usage mining techniques would define some method of measuring support or confidence in a candidate pattern. Often, these measurements would be based on co-occurrences of

```
void example() {
                                                         $START
 1
                                                     1
 2
       void *A;
                                                     2
                                                         strcasecmp
 3
       void *B;
                                                     3
                                                         strcasecmp != 0
 4
       if (!strcasecmp()) {
                                                     4
                                                         strcasecmp
 5
         addReplyHelp();
                                                     5
                                                         strcasecmp == 0
       } else if (!strcasecmp()) {
 6
                                                     6
                                                         dictGetIter
 7
         // Allocate
                                                     7
                                                         log
         A = dictGetIter();
                                                         dictGetIter \rightarrow dictNext
 8
                                                     8
                                                         dictNext
 9
         log();
                                                     9
10
         // Iterate
                                                    10
                                                         dictNext != 0
         while ((B = dictNext(A)) != 0) {
                                                         dictNext \rightarrow dictGetKey
11
                                                    11
12
           dictGetKey(B);
                                                    12
                                                         dictGetKey
13
           if (strmatchlen(sdslen())) {
                                                    13
                                                         sdslen
                                                         \texttt{sdslen} \rightarrow \texttt{strmatchlen}
14
              addReplyBulk();
                                                    14
            }
                                                    15
                                                         strmatchlen
15
16
         }
                                                    16
                                                         strmatchlen != 0
         // Release
                                                    17
                                                         addReplyBulk
17
18
         dictReleaseIter(A);
                                                    18
                                                         dictGetIter \rightarrow dictNext
       } else if (!strcasecmp()) {
19
                                                    19
                                                         dictNext
         addReplyLongLong(listLength());
                                                         dictNext == 0
20
                                                    20
       } else {
                                                         dictGetIter
21
                                                    21
         addReplySubcommandSyntaxError();
22
                                                    22
                                                            \rightarrow dictReleaseIter
23
       }
                                                    23
                                                         dictReleaseIter
    }
24
                                                    24
                                                         $END
```

(a) Sample procedure, showcasing an iterator usage pattern from the Redis open-source project (b) One example trace for the example procedure in Fig. 3.2a

Figure 3.2: Example procedure and corresponding trace. The notation  $A \rightarrow B$  signifies that the result of call A is used as a parameter to call B.

	<ol> <li>dictGetIter</li> <li>dictGetIter</li> </ol>		
1 strcasecmp	$\begin{array}{c} 3 \\ cmp \\ cmp \\ != 0 \end{array} \qquad \begin{array}{c} 3 \\ dictNext \\ 5 \\ dictNext \\ == 0 \end{array}$	1 sdslen	
2 strcasecmp != 0		2 $\rightarrow$ strmatchlen	
3 strcasecmp == ∅	6 dictNext != 0	4 strmatchlen != 0	
(a) Cluster 1	<pre>7 dictGetIter 8 → dictReleaseIter 9 dictReleaseIter</pre>	(c) Cluster 3	

(b) Cluster 2

Figure 3.3: Clusters generated via Domain-Adapted Clustering (DAC)

terms (or sets of terms). The ml4spec toolchain leverages a key insight: traditional co-occurrence statistics and machine-learning-assisted metrics (extracted via unsupervised learning, specifically word embeddings) can be combined in fruitful ways. Referencing our example in Fig. 3.2, we might hypothesize, based on co-occurrence, that dictGetIter and log are related. For the sake of argument, imagine that in each extracted trace we find this same pattern. How can we refine our understanding of the relationship between dictGetIter and log?

It is in these situations that adding unsupervised learning improves the results. A word-vector learner, such as Facebook's fastText (Bojanowski et al., 2017), can provide us with a measurement of the similarity between dictGetIter and log. This measurement provides a contrast to the co-occurrence based view of our data. Intuitively, word-vector learners utilize the *Distributional Hypothesis*: similar words appear in similar contexts (Harris, 1954). The global context, captured by co-occurrence statistics, can be supplemented and refined by the local-context information that word-vector learners naturally encode. Section 3.4.5 explores the impact and relative importance of both traditional co-occurrence statistics and machine-learning-assisted metrics.

**Phase IV: Domain-Adapted Clustering.** The trace given in Fig. 3.2b exhibits several different patterns. The difficulty in mining from static traces like the one

in Fig. 3.2b comes from the need to learn a separation of the various, possibly interacting, patterns and behaviors. To address this challenge, we introduce Domain-Adapted Clustering: a generalizable approach to clustering corpora of sequential data. Domain-Adapted Clustering leverages the insight that it can be useful to combine machine-learning-assisted metrics with co-occurrence statistics captured directly from the target corpus. Using Domain-Adapted Clustering, we can extract the clusters shown in Fig. 3.3. These clusters allow us to solve the problem of *disentanglement* by projecting the trace in Fig. 3.2b into the vocabularies defined by each cluster. It is this "focusing" of the mining process that enables the ml4spec toolchain to apply traditional specification-mining techniques in an Open-World setting.

**Phase V: Mining.** Finally, we can extract free-form specifications by applying traditional mining techniques to the projected traces that ml4spec creates. One powerful aspect of the ml4spec toolchain is its disassociation from any particular mining strategy. The real challenge of Open-World Specification Mining is extracting, without user-directed feedback, reasonable clusters of possibly related terms. With this information in hand, a myriad of trace-based miners can be applied. Figures 3.4a and 3.4b highlight this ability by showing both a finite-state automaton (FSA) mined via the classic k-Tails algorithm and a Hidden Markov Model (HMM) learned directly from the projected traces (Biermann and Feldman, 1972; Seymore et al., 1999).

## 3.3 Technique

## 3.3.1 Parametric Lightweight Symbolic Execution

The first phase of the ml4spec toolchain generates intraprocedural traces using a form of parametric lightweight symbolic execution, introduced in Chapter 2. Parametric Lightweight Symbolic Execution (PLSE) takes, as input, a buildable C project and a set of abstractions. Abstractions are used to parameterize the resulting traces. In our setting, the abstractions allow us to enrich the output vocabulary. This enrichment enables the final phase of the ml4spec toolchain (mining) to extract specifications that include each of the following types of information:



(a) Example Finite State Automaton (FSA) extracted from projected traces

(b) Example Hidden Markov Model (HMM) extracted from projected traces

Figure 3.4: Example specifications that were mined by projecting all of the traces extracted from Fig. 3.2a into the vocabulary defined by Fig. 3.3b. Specifications for the vocabularies defined by the clusters in Figs. 3.3a and 3.3c are also generated, but not shown here.

- **Temporal properties:** the ml4spec toolchain abstracts the sequence of calls encountered on a given path of execution. The temporal ordering of these calls is preserved in the output traces.
- **Call-return constraints:** often a sequence of API calls can only continue if previous calls succeeded. In C APIs checking for success involves examining the return value of calls. **ml4spec** abstracts simple checks over return values to capture specifications that involve call-return checks.
- **Dataflow properties:** some specification miners are *parametric*—these miners can capture relationships between parameters to calls and call-returns. To highlight the flexibility that PLSE provides, we include an abstraction that tracks which call results are used, as parameters, in future calls. This call-to-call dataflow occurs in many API usage patterns.
- Result propagation: the return value of a given procedure can encode valuable information. Some procedures act as wrappers around lower-level APIs, while other procedures may forward error results from failing calls. In either case, forwarding the result of a call, for any purpose, is abstracted into our traces to aid in downstream specification mining. ml4spec also abstracts constant return values: returning a constant may indicate success or failure, and such information may aid in downstream specification mining.

With these various abstractions parameterizing our trace generation, simple downstream miners, such a k-Tails, are capable of mining rich specifications. However, there is a cost to the variety of abstractions we employ. Each abstraction introduces more words into the overall vocabulary, and, as the size of the overall vocabulary grows, so does the challenge of disentangling traces.

Finally, it is worthwhile to address the limitations of Parametric Lightweight Symbolic Execution. PLSE is intraprocedural and therefore risks extracting only partial specifications. PLSE also makes no attempt to detect infeasible traces. Finally, PLSE enumerates a fixed number of paths. As part of this enumeration, any loops are unrolled for a single iteration only.<sup>2</sup> In practice these limitations enable the PLSE technique to scale and, for the purposes of the ml4spec toolchain, losses in precision are balanced by the utilization of machine-learning-assisted metrics (which can tolerate noisy data).

## 3.3.2 Thresholding and Sampling

In this section, we outline the techniques used in the ml4spec toolchain to take a corpus of traces, generated via Parametric Lightweight Symbolic Execution (PLSE), and prepare them for word-vector learning and specification mining. In particular, we present two key contributions, Hierarchical Thresholding and Diversity Sampling, which improve the overall quality of our results. In addition, we discuss alternative approaches.

### **Hierarchical Thresholding**

When preparing data for a word-vector learner, it is common to select a *vocabulary minimum threshold*, which limits the words for which vectors will be learned. Any word that appears fewer times than the threshold is removed from the training corpus. Through this process extremely rare words, which may be artifacts of data collection, typos, or domain-specific jargon, are removed. In the domain of mining specifications, we have a similar need. We would like to pre-select terms, from our overall vocabulary, that occur enough times to be used as part of a pattern or specification. We could simply set an appropriate vocabulary minimum threshold using our word-vector learner of choice; however, this approach ignores a unique aspect of our traces. The traces we have, which are used as input to both the word-vector learner and specification miner, are intra-procedural traces extracted from a variety of procedures. To select terms that occur frequently does not necessarily select for terms that are used across a variety of procedures. Because our traces are paths through

 $<sup>^{2}</sup>$ This single iteration loop unrolling gives us traces in which the loop never occurred and traces in which we visit the loop body exactly one time. Yun et al. (2016) follow a similar model and argue that most API usage patterns are captured in a single loop unrolling.

a procedure, it is possible to have a frequently occurring term (with respect to our traces) that only occurs in one procedure. To achieve our desire for terms that are used in a variety of diverse contexts, we developed a modified thresholding approach: *Hierarchical Thresholding*. Hierarchical Thresholding counts how often a term occurs across procedures instead of traces. This simple technique, with its utilization of the extra level of hierarchical information that exists in the traces, reduces the possibility of selecting terms that are rare at the source-code level but frequent in the trace corpus.

### **Diversity Sampling**

The corpus of symbolic traces that we obtain, via lightweight symbolic execution, can be a challenging artifact to learn from. The symbolic executor, at execution time, builds an execution tree and it is from this tree that we enumerate traces. Any attempt to learn from such traces can be thought of as an attempt to indirectly learn from the original execution trees. The gap between the tree representation and trace representation introduces a challenge: terms that co-occur at the start of a large procedure (with many branches) will be repeated hundreds of times in our trace corpus. This prefix duplication, which we labelled *prefix dominance* (in Section 2.4.1), adversely affects the quality of word embeddings learned from traces.

As part of the ml4spec toolchain, we introduce a novel trace-sampling methodology, which seeks to resolve the impact of *prefix dominance*. We call this sampling methodology *Diversity Sampling* because it samples a diverse and representative set of traces by using a similarity metric to drive the sample-selection process.

Algorithm 1 provides the details of our Diversity Sampling technique. Because we work with intra-procedural traces, we can associate each trace with its source-code procedure. Consequently, the sampling routine can sample maximally diverse traces for each procedure independently. (For a simple reason, our algorithm treats the trace corpus as a collection of sets: each set holds the intra-procedural traces for one source procedure.) To begin Diversity Sampling, we either return all traces (if the number of traces for a given procedure is less than our sampling threshold), or we begin to iterate over the available traces and make selections. At each step of the selection loop, on lines 8–19, we identify a trace that has the maximum average Jaccard distance when measured against our previous selections. Jaccard distance is a measure computed between sets and, in our setting, we use the set of unique tokens in a given trace to compute Jaccard Distances. We take the average Jaccard distance from the set of currently sampled traces to ensure that each new selection differs from all of the previously selected traces. Finally, when we have selected an appropriate number of samples, we return them and proceed to process traces from the next procedure.

### **Alternative Samplers**

Although Diversity Sampling is rooted in the intuition of extracting the most representative set of traces for each procedure, it may not make a difference in the quality of downstream results. It is for this reason that we also consider, in our ml4spec toolchain, two alternative approaches to trace sampling: no sampling and random sampling. We include the option of no sampling because word-vector learners thrive on both the *amount* and *quality* of data available. It is reasonable to ask whether the training data lost by downsampling our trace corpus has enough negative impact to offset possible gains. We also include random sampling as a third alternative; our motivation for this inclusion is to assess the impact of our metric-guided selection. Section 3.4.4 evaluates the sampling strategies discussed here.

## 3.3.3 Domain-Adapted Clustering

Section 3.3.1 outlined how ml4spec makes use of Parametric Lightweight Symbolic Execution (PLSE) to generate rich traces. In Section 3.3.2, we presented innovations that improved the traces generated by PLSE, and addressed some of the challenges associated with learning from traces. In this section, we introduce Domain-Adapted Clustering, our solution to the challenge of clustering related terms. We seek to cluster related terms (words) to simplify the Open-World Specification Mining task. Traditional specification miners often use either rule templates or some form of

Algorithm 1: Diversity Sampling

input :A trace corpus TR
output:A down-sampled trace corpus

```
1 outputs \leftarrow [];
 2 for T \in \mathbb{TR} do
        if |\mathsf{T}| \leq SAMPLES then
 3
             outputs = outputs \cup T;
 \mathbf{4}
 \mathbf{5}
             continue;
        end
 6
        choices \leftarrow T[0];
 \mathbf{7}
        while | choices | < SAMPLES do
 8
             D^* = 0.0;
 9
             S = null;
10
             for t \in T - choices do
11
                  D = AverageJaccardDistance(t, choices);
12
                 if D \ge D^* then
\mathbf{13}
                      S = t;
14
                      \mathsf{D}^* = \mathsf{D};
\mathbf{15}
                  end
16
             end
\mathbf{17}
             choices = choices \cup S
18
        end
\mathbf{19}
        outputs = outputs \cup choices;
\mathbf{20}
21 end
22 return outputs;
```

user-directed input (the API surface of interest, or perhaps a specific class or selection of classes from which specifications should be mined). In our Open-World setting, none of this information is available. Therefore, we have developed a methodology for extracting clusters of related terms that harnesses the power of unsupervised learning (in the form of word embeddings). With these clusters in hand, the task of mining specifications is greatly simplified.

### Motivation

To motivate Domain-Adapted Clustering, it is revealing to consider the relationships among the following ideas:

- Co-occurrence: word–word co-occurrence can be a powerful indicator of some kind of relationship between words. Co-occurrence is, by its nature, a global property that can, optionally, be associated with a sense of direction (word A appears to the left/right of word B).
- Analogy: analogies are another way in which words can be related. The words that form an analogical relationship encode a kind of information that is subtly different from the information that co-occurrence provides. Given the analogy A is to B as C is to D, one would find that A and B often co-occur, as do C and D; however, there may be no strong relationship (in terms of co-occurrence) between A/B and C/D.
- Synonymy: synonyms are, in some sense, encoding strictly local structure. Two synonymous words need not co-occur; instead, synonyms are understood through the concept of replaceability: if one can replace A with B then they are likely synonyms.

We can now attempt to codify which of these concepts are of value for Open-World Specification Mining. To do so, we will introduce a simple thought experiment: consider an extremely simple specification that consists of a call to foo and a comparison of the result of this call to 0. In our traces this pattern would manifest in one of two forms: (i) foo foo==0 or (ii) foo foo!=0. For the sake of our thought experiment, also assume that, by chance, print follows foo in our traces 95% of the time. What kinds of relationships do we need to use to extract the cluster of terms: foo, foo==0, and foo!=0? We could use co-occurrence, however using co-occurrence will likely pick up on the uninformative fact that foo frequently co-occurs with print. Furthermore, co-occurrence may struggle to pick up on the relationship between foo and the check on its result: because each check is encoded as a distinct word, neither check will co-occur with extremely high frequency. We could, instead, use synonymy, but it is trivial to imagine words, such as malloc and calloc, that are synonyms but not related in the sense of a usage pattern or specification.

It is the insufficiency of both co-occurrence and synonymy that forms the basis of Domain-Adapted Clustering. Because neither metric covers all cases, Domain-Adapted Clustering forms a parameterized mix of two metrics: one based on left and right co-occurrence, and another based on unsupervised learning. Because both of these metrics encode a distance (or similarity) of some sort, Domain-Adapted Clustering can be thought of as computing the pair-wise distance matrices and then mixing them via a parameter  $\alpha \in [0, 1]$ . Figure 3.1 provides a visual overview of the mixing process Domain-Adapted Clustering employs.

#### The Co-occurrence Distance Matrix

Domain-Adapted Clustering utilizes co-occurrence statistics extracted directly from the (sampled and thresholded) trace corpus. To capture as much information as possible, Domain-Adapted Clustering walks each trace and computes, for each word pair (A, B), the number of times that A follows B and the number of times that B follows A. These counts are converted to percentages and these percentages represent a kind of similarity between A and B. The higher the percentages, the more often A and B co-occur. To turn the percentages into a distance, we subtract them from 1.0 and store the average of the left-distance and right-distance in our co-occurrence distance matrix.

#### The Word-Embedding Distance Matrix

To incorporate unsupervised learning, Domain-Adapted Clustering utilizes wordvector learners. The use of word-vector learners in the software-engineering domain is not a new idea (Henkel et al., 2018; DeFreez et al., 2018b; Ye et al., 2016b; Nguyen et al., 2017a; Pradel and Sen, 2018). Many recent works have explored the power of embeddings in the realm of understanding and improving software. What we contribute is, to the best of our knowledge, the first thorough comparison of three of the most widely used word-vector learners in the application domain of software engineering. We do this comprehensive evaluation to test an intuition that *sub-word information improves the quality of embeddings learned from software artifacts*. We base this intuition on the observation that similarly named methods have similar meaning. Section 3.4.4 provides the details of this evaluation.

Our choice of word-vector learners as an unsupervised learning methodology is a deliberate one. Earlier, we saw how synonymy could be a useful (albeit incomplete) property to capture. Furthermore, we already have a notion of distance between words (given to us via our co-occurrence distance matrix). Word-vector learners mesh well with both of these pre-existing criteria: word vectors encode local context and are able to capture synonymy. Additionally, word–word distance is encoded in the learned vector space. These properties make word-vector learners a convenient choice for Domain-Adapted Clustering. To extract a distance matrix from a learned word embedding, Domain-Adapted Clustering computes, for each word pair (A, B), the cosine distance between the embedding of A and the embedding of B (here, we use cosine distance because it is the distance of choice for word vectors).

### **Cluster Generation**

To generate clusters, Domain-Adapted Clustering applies the insight that the clusters we seek should be expressed in concrete usages. This idea leads us to invert the problem of clustering—instead of clustering all of the terms in the vocabulary, we take a more bottom-up approach. We start with an individual trace from our corpus of sampled traces. Within the trace, we find topics or collections of terms that are related under our machine-learning-assisted metric: we use the combined distance matrix we created previously and apply a threshold to detect words that are related. Within a trace, any two words whose distance is below the threshold are assigned to the same intra-trace cluster. The next step uses the set of all intra-trace clusters to create a set of reduced traces: each trace in the corpus of traces is projected onto each of the intra-trace clusters to create a new corpus of reduced traces. To form final clusters, we apply a traditional clustering method (DBSCAN (Ester et al., 1996a)) to
the collection of reduced traces. In this final step we use Jaccard distance between the sets of tokens in the reduced traces as the distance metric.

One distinctive advantage of Domain-Adapted Clustering, for our use case, is its ability to generate overlapping clusters. Most off-the-shelf clustering techniques produce disjoint sets but, in the realm of Open-World Specification Mining, it is easy to conceive of multiple patterns that share common terms (opening a file and reading versus opening a file and writing). Finally, it is worthwhile to note that the clustering step we have outlined here introduces two hyper-parameters: the threshold to use for intra-trace clustering (which we will call  $\beta$ ) and DBSCAN's  $\epsilon$ , which controls how close points must be to be considered neighbors. This leaves Domain-Adapted Clustering with a total of three tunable hyper-parameters:  $\alpha$ ,  $\beta$ ,  $\epsilon$ .

# **3.4** Experiments

In this section we introduce our evaluation methodology and address each of our four research questions. For the purposes of evaluation we ran the ml4spec toolchain on five different open source projects:

- Curl: a popular command-line tool for transferring data.
- Hexchat: an IRC client.
- Ngnix: a web server implementation.
- Nmap: a network scanner.
- **Redis:** a key-value store.

These projects were selected because they exhibit a wide variety of usage patterns across diverse domains. For each of these five projects, we performed a grid search to gain an understanding of our various design decisions. The following section details this search.

Table 3.1: Grid search parameters

Name	Values	Purpose	Phase
learner	{fastText, GloVe, word2vec}	Word-vector learner to use	II
sampler	{Diversity, Random, None}	Sampling method to use	III
alpha	$\{0.00, 0.25, 0.50, 0.75, 1.00\}$	Weight for combined distance matrix	IV
beta	$\{0.20, 0.25, \ldots, 0.45, 0.50\}$	Threshold for intra-trace clustering	IV
epsilon	$\{0.10, 0.30, 0.50, 0.70, 0.90\}$	Parameter to DBSCAN	IV

# 3.4.1 Grid Search

To facilitate a comprehensive evaluation of ml4spec, we performed a grid search across thousands of parameterizations of the ml4spec toolchain. The grid search serves two purposes. First, the results of the grid search provide a firmer empirical footing for understanding the efficacy and impacts of different aspects of our toolchain (in particular, the grid search aids in quantifying the impacts of different word-vector learners and sampling methodologies). Second, the Open-World Specification Mining task emphasizes a lack of user-directed feedback—to meet this standard we must ensure, via the data gleaned from the grid search, that the hyper-parameters associated with the ml4spec toolchain can be set, globally, to good default values. Table 3.1 outlines the parameters in play and the ranges of values investigated for each parameter. Upper and lower limits for each search range were carefully chosen, based on the results of smaller searches, to reduce the computational costs of the larger search over the parameters presented in Table 3.1.

# 3.4.2 RQ1: Can we effectively mine useful and clean clusters in an Open-World setting?

Research Question 1 asked whether we can mine useful and clean clusters. The difficulty with mining such clusters lies in the setting of our mining task. We seek to solve the problem of mining specifications in an Open-World setting: one in which implicit and explicit sources of hierarchical or taxonomic information are unavailable. It is this Open-World setting that creates a unique need for disentangling the many

```
{
    hashTypeInitIter, hashTypeNext, hashTypeReleaseIter
}
    (a) Cluster in the vocabulary of simple call names
{
    hashTypeInitIter, hashTypeInitIter→hashTypeNext, hashTypeNext, hashTypeNext==-1, hashTypeNext!=-1, hashTypeInitIter→hashTypeReleaseIter, hashTypeReleaseIter
}
```

(b) Cluster in our enriched vocabulary

Figure 3.5: Comparison between two clusters

different topics that may exist in an abstracted symbolic trace. The purpose of Research Question 1 is to understand the efficacy of the techniques described earlier (specifically, Domain-Adapted Clustering) against a key challenge of Open-World Specification Mining: learning correct and clean clusters.

To measure the quality of our learned clusters we found the need for a benchmark. Unfortunately, to the best of our knowledge, the problem of Open-World Specification Mining has not been previously addressed and, therefore, there is no ground truth to evaluate our learned clusters against. One possible avenue of evaluation and source of implicit clusters exists in source-code documentation. Many thoroughly documented and heavily used APIs include information on the associations between functions (most commonly in the form of a "See also..." or "Related methods..." listing). Another possible source of implicit information comes from projects that have made the transition from a language like C to a language like C++. In such a transition methods are often grouped into classes and this signal could be used to induce a clustering. Finally, there is the implicit clustering induced by the locations where various API methods are defined: even in C, functions defined in the same header are likely related.

Despite these various sources of implicit clusters, we have identified a need for manually defined gold standard clusters. We use manually extracted ground truth clusters for two reasons. First, the sources of information listed above are indications of relatedness but not necessarily indications of a specification or usage pattern. For example, several different methods are commonly defined for linked lists, such as length(), next(), prev(), and hasNext() but not all of these methods are necessarily used together in a pattern. Second, the vocabulary we are working over includes more than simple call names—we also have information related to the path condition and information about dataflow between calls. For example, compare the two clusters given in Fig. 3.5. The cluster in Fig. 3.5a consists only of call names, while the cluster in Fig. 3.5b includes call names, return value checks, and dataflow information. In comparing these two clusters, it becomes clear that a cluster over words in our enriched vocabulary (induced by the abstractions we choose) is strictly more informative than a cluster over a vocabulary of simple call names.

Taken together, these two issues (the weak signal of the aforementioned sources and the lack of labels for some words in our enriched vocabulary) make manually extracted clusters more desirable. For the purpose of this evaluation we have extracted 71 gold standard clusters from five open source projects. We have placed no explicit limit on the sizes of the clusters we included, thereby increasing the challenge of recovering all the clusters in our benchmark correctly.

Using our set of 71 gold standard clusters we are able to perform a quantitative evaluation by measuring the Jaccard similarity<sup>3</sup> of our extracted clusters and our gold standard clusters. Because our toolchain does not mine a fixed number of clusters, we need some way to "pair" an extracted cluster with the gold standard cluster it most represents. To do this, we look for a pairing of extracted clusters with gold standard clusters that maximizes the average Jaccard similarity. This provides us with a way to have a consistent evaluation regardless of the number of total clusters we extract. (One might argue that this allows for extracting an unreasonable amount of clusters in an attempt to game this metric. However, this kind of "metric hacking" is unachievable in our toolchain due to the use of DBSCAN to extract clusters from reduced traces. Clustering our reduced traces, using the Jaccard distance between sets of tokens within a trace, removes the possibility that our tool is simply enumerating all possible clusters to achieve a high score.)

<sup>&</sup>lt;sup>3</sup>Jaccard similarity between sets A and B is  $\frac{|A \cap B|}{|A \cup B|}$ . Jaccard distance is one minus the Jaccard

	Benchmark				
Measurement	curl	hexchat	nginx	nmap	redis
Jaccard Intersection	62.8% 79.7%	52.8% 78.7%	44.9% 67.6%	49.1% 70.1%	71.9% 83.5%

Table 3.2: Best scoring configurations for each of the five target projects

In addition to Jaccard similarity, which penalizes both omissions and spurious inclusions, we also measure the percent intersection between our extracted clusters and the clusters in our gold standard dataset. Table 3.2 provides both of these measurements for each of the five open-source projects we examined. Examining Table 3.2, we observe that the ml4spec toolchain retrieves clusters that have a strong agreement with the clusters in our gold standard dataset. Furthermore, the intersection similarity results show that our extracted clusters contain, on average, over two thirds of the desired terms from the clusters in our gold standard dataset. Together, these results answer Research Question 1 in the affirmative: ml4spec is capable of extracting clean and useful clusters in an Open-World setting.

# 3.4.3 RQ2: How does DAC compare to off-the-shelf clustering techniques?

In this section, we explore how our Domain-Adapted Clustering (DAC) technique (a key piece of our Open-World specification miner) compares to traditional clustering approaches. To understand the relationship between DAC and more traditional clustering methods, it is instructive to consider the input data we have available to use in the clustering process. Prior to clustering, we have access to a pairwise distance matrix (created via a combination of co-occurrence statistics and word–word cosine distance), our learned word vectors, and our original traces.

	Benchmark				
Clustering	curl	hexchat	nginx	nmap	redis
DBSCAN Agglomerative	49.9% 38.9%	$36.7\%\ 15.8\%$	$34.9\%\ 25.8\%$	$36.5\%\ 12.7\%$	$59.6\%\ 8.6\%$
Affinity Prop.	12.1%	10.2%	11.9%	10.3%	11.5%
DAC (Rel. Increase)	+25.9%	+41.2%	+24.1%	+34.5%	+25.7%

Table 3.3: DAC compared to off-the-shelf clustering techniques

Table 3.4: DAC compared to off-the-shelf clustering techniques boosted by our machine-learning-assisted metric

	Benchmark				
Clustering	curl	hexchat	nginx	nmap	redis
DBSCAN	49.9%	38.0%	38.1%	37.9%	64.4%
Agglomerative	40.7%	21.6%	39.2%	19.7%	26.7%
Affinity Prop.	15.3%	12.5%	13.4%	15.3%	15.5%
DAC (Rel. Increase)	+25.9%	+36.9%	+10.6%	+29.7%	+16.5%

quite open. However, using our word vectors as the input to clustering ignores our earlier insight about the advantage of combining word embeddings and co-occurrence statistics. Therefore, we focus on clustering algorithms that accept pre-computed pair-wise distances as input. From this class of clustering methods we have selected the following techniques to compare against: DBSCAN (Ester et al., 1996a), Affinity Propagation (Frey and Dueck, 2007), and Agglomerative Clustering.

To compare the selected traditional techniques to DAC we use the benchmark we introduced in RQ1 as a means of consistent evaluation. Both DAC and our selection of traditional techniques require some number of hyper-parameters to be set. To ensure a fair evaluation, we have searched over a range of hyper-parameters for each of the selected techniques and compare between the best configurations for each technique. Table 3.3 provides performance measurements for each of the three off-the-shelf clustering baselines across each of our five target projects. In addition, Table 3.3 provides the relative performance increase gained by using DAC in place of these baselines.<sup>4</sup> For this comparison we have made only the co-occurrence distance matrix available to our off-the-shelf baselines as one of DAC's key insights was the importance of a machine-learning-assisted metric. Table 3.4 follows the same format but provides each off-the-shelf technique access to the combined matrix DAC uses for clustering. In either case, we see that DAC outperforms each of the baselines by a wide margin.

# 3.4.4 RQ3: How does the choice of word vector learner and the choice of sampling technique affect the resulting clusters?

Research Question 3 seeks to understand the impact of two choices made in the earlier portion of our toolchain: the choice of word vector learner and the choice of trace sampling technique. For the choice of word vector learner we argued that fastText with its utilization of sub-word information (in the form of character level n-grams) would provide embeddings better suited to the task of extracting clean clusters. We based this prediction on the observation, made by many, that similarly named methods often have similar meaning. When it came to the choice of trace sampling we sought to reduce the impact of a problem, identified in Section 2.4.1, called *prefix dominance*. To address this issue of *prefix dominance* in our specification mining setting we introduced a trace sampling methodology termed Diversity Sampling.

To understand the interplay and effects of these choices we have evaluated the ml4spec toolchain in nine configurations. These nine configurations are defined by two choices: a choice of word vector learner (either fastText (Bojanowski et al., 2017), GloVe (Pennington et al., 2014a), or word2vec (Mikolov et al., 2013b)) and a choice of trace sampling technique (either Diversity Sampling, random sampling, or no sampling). By evaluating our full toolchain with varying choices of embedding and sampling methodology we can either confirm or refute our intuitions. We leverage

<sup>&</sup>lt;sup>4</sup>We compute the relative performance increase by comparing to the best overall off-the-shelf technique on a per-project basis.

	Learner			
Sampler	word2vec	GloVe	fastText	
Diversity Sampling	52.7%	48.6%	54.9%	
Random Sampling	44.5%	43.6%	53.0%	
No Sampling	47.7%	46.3%	50.7%	

Table 3.5: Top-1 performance (geometric mean across our five target projects). The shaded row and column represent the best sampler and learner, respectively.

the gold standard clusters introduced in RQ1 to provide a consistent benchmark for comparison between the nine configurations we've outlined.

Table 3.5 provides performance measurements (measured against our benchmark) across all of the configurations we established in Section 3.4.1. We look at the single best-performing configuration (with sampler and learner fixed to one of the nine choices outlined earlier). In Table 3.5, we can see that fastText is superior (regardless of sampling choice) to any of the other word vector learners by a wide margin. We also observe that fastText paired with Diversity Sampling is the most performant combination. However, fastText with no sampling is not far behind—this is perhaps indicative of both the impact of word embeddings and the need for larger corpora to learn suitable embeddings.

These results support two conclusions. First, fastText, with its use of sub-word information, outperforms GloVe and word2vec in the cluster extraction task we have benchmarked. Second, Diversity Sampling both improves the performance of our toolchain and word vector learner (by reducing the amount of input data) and provides an increase in performance compared to the other baseline choices of sampling routine. These results also support further examination of the advantages of sub-word information in the software-engineering domain; specifically, we note that fastText has no concept of the ideal boundaries between sub-tokens that naturally exist in program identifiers. A word vector learner equipped with this knowledge may produce even more favorable results.

# 3.4.5 RQ4: Is there a benefit to using a combination of co-occurrence statistics and word embeddings?

One of the key insights from Section 3.3.3 was that word embeddings and co-occurrence statistics capture subtly different information. Word embeddings excel at picking up on local context (a direct result of being based on the distributional hypothesis which asserts that similar words appear in similar contexts). This focus on local context makes word embeddings well-suited for tasks like word similarity and analogy solving. For specification mining, co-occurrence information is often used, in some form, to capture the "support" for a candidate rule or pattern. These co-occurrence statistics encode a global relationship between words that is more far-reaching than the relationship captured by word vectors.

Research Question 4 attempts to precisely quantify the impact of these two different sources of information. This effort is made somewhat easier by the choice to include a tunable parameter in our toolchain that represents the relative weight of word–word distance and co-occurrence distance in our final pair-wise distance matrix. By evaluating our full toolchain with a gradation of weight values we can pinpoint the mix of metrics that lead to optimal performance on the benchmark we introduced earlier.

The results for top-1 performance, given in Fig. 3.6, paint a clear picture of the relationship between word embeddings and co-occurrence statistics. In Fig. 3.6 we observe that, for all five projects, adding word embeddings to our distance matrix produces a pronounced increase in performance. We observe that, for each project, the performance benefit provided by adding word embeddings peaks at  $\alpha = 0.75$ . These results suggests an affirmative answer to Research Question 4: there is a quantifiable benefit to using both co-occurrence statistics and word embeddings; furthermore, a combination that favors the distances produced via word embeddings yields maximum performance across each of the projects we examined.



Figure 3.6: Peak benchmark performance for varying values of  $\alpha$ 

# 3.5 Related Work

There exists a wide variety of related works from the specification mining, API misuse, program understanding, and entity embedding communities. For comprehensive overviews of specification mining and misuse we refer the reader to Lo et al. (2011) and Robillard et al. (2013). For efforts in machine learning and its application in the software-engineering domain Allamanis et al. (2017a) provide an excellent survey. In addition, there exists a listing of machine learning on code resources maintained by the community (source{d}, 2019). For details on embeddings and their use in the software-engineering domain Martin Monperrus (2019) provide an up-to-date listing. In the following sections, we discuss related works in the realms of specification mining and embeddings of software artifacts in greater detail.

## **Specification Mining**

There is a rich history of work on mining specifications, or usage patterns, from programs. Earlier approaches, such as Li and Zhou (2005), provided relatively simple specifications. Going forward in time, a growing body of work attempted to produce richer FSA-based specifications (Lorenzoli et al., 2008; Ammons et al., 2002; Pradel and Gross, 2009; Gabel and Su, 2008; Walkinshaw and Bogdanov, 2008; Walkinshaw et al., 2007; Quante and Koschke, 2007; Shoham et al., 2008; Dallmeier et al., 2006; Acharya and Xie, 2009; Lo and Khoo, 2006). Some recent works such as Deep Specification Mining and Doc2Spec, have incorporated NLP techniques (Zhong et al., 2009; Le and Lo, 2018). DeFreez et al. (2018b) use word-vector learners to bolster traditional support-based mining via the identification of *function synonyms*. In the broader field of non-FSA-based specification mining techniques, there exist several novel techniques: Nguyen et al. (2009) mine graph-based specifications; Sankaranarayanan et al. (2008) produce specifications as Datalog rules; Acharya et al. (2007) create a partial order over function calls and Murali et al. (2017b) develop a Bayesian framework for learning probabilistic specifications. In addition to mining, several works focus on the related problem of detecting misuses (Engler et al., 2001; Yun et al., 2016; Monperrus and Mezini, 2013; Livshits and Zimmermann, 2005; Wasylkowski et al., 2007).

The ml4spec toolchain is agnostic to the choice of trace-based mining technique used to generate specifications. This miner-agnostic perspective makes ml4spec a front end that enables prior trace-based miners to work in the Open-World setting we have described. In addition, ml4spec's use of Parametric Lightweight Symbolic Execution makes it possible to mine, via traditional methods, specifications that involve both control-flow and data-flow information.

# **Embeddings of Software Artifacts**

Recently, several techniques have leveraged learned embeddings for artifacts generated from programs. Nguyen et al. (2017b, 2016) leverage word embeddings (learned from ASTs) in two domains to facilitate translation from Java to C#. Le and Lo (2018) use embeddings to bootstrap anomaly detection against a corpus of JavaScript programs.

Gu et al. (2016) leverage an encoder/decoder architecture to embed whole sequences in their DEEPAPI tool for API recommendation.

Pradel and Sen (2017) use embeddings (learned from custom tree-based contexts built from ASTs) to bootstrap anomaly detection against a corpus of JavaScript programs. Gu et al. (2016) leverage an encoder/decoder architecture to embed whole sequences in their DEEPAPI tool for API recommendation. API2API by Ye et al. (2016a) also leverages word embeddings, but it learns the embeddings from API-related natural-language documents instead of an artifact derived directly from source code. Alon et al. (2018c) learn from paths through ASTs to produce general representations of programs; in (Alon et al., 2018b) they expand upon this general representation by leveraging attention mechanisms. Ben-Nun et al. (2018) produce embeddings of programs that are learned from both control-flow and data-flow information. Zhao et al. (2018a) introduce type-directed encoders, a framework for encoding compound data types via a recursive composition of more basic encoders. Using input/output pairs as the input data for learning, Piech et al. (2015) and Parisotto et al. (2016) learn to embed whole programs. Using sequences of live variable values, Wang et al. (2017) produce embeddings to aid in program repair tasks. Allamanis et al. (2017b) learn to embed whole programs via Gated Graph Recurrent Neural Networks (GG-RNNs) (Li et al., 2015). Peng et al. (2015) provide an AST-based encoding of programs with the goal of facilitating deep-learning methods.

In contrast to prior work on the embedding of software artifacts, we provide both a novel use of embeddings in the software-engineering domain (in the form of Domain-Adapted Clustering and its machine-learning-assisted metric) and a comprehensive comparison between three state-of-the-art word embedding techniques (fastText (Bojanowski et al., 2017), GloVe (Pennington et al., 2014a), and word2vec (Mikolov et al., 2013b)). Furthermore, we make an insight into a future line of work involving the utilization of refined sub-token information to improve embeddings in the software-engineering domain.

# 3.6 Future Work

Similar to the work in Chapter 2, this work also occurred before the "large-languagemodel" era. It would be worthwhile to re-visit some of the key challenges from Open-World Specification Mining and evaluate how well a large language model could do in the zero/few-shot setting. For example, a key challenge we face in the Open-World Mining setup was clustering of related terms within a trace. It may be the case that this problem is easily solved by large models that have been exposed to hundreds of gigabytes of training data. Furthermore, if large models can be leveraged to perform clustering in the zero/few-shot setting, one might wonder if large models could *directly* ascertain specifications or usage patterns from a corpus of source code.

# 3.7 Notes

This chapter represents the second piece of work I produced during my graduate studies and, more critically, my first experience with having my work rejected. In the end, the rejection of this work was what (eventually) lead me to explore learning from non-code artifacts. I think, at the time, I had not yet absorbed one of the many critical lesson of graduate work: rejection is simply a chance for refinement. (Up until this point in my studies, I had only one experience with submitting a conference paper; and that one experience ended in acceptance—I did not expect this pattern to continue, but when that first rejection came it was more difficult than I had imagined.)

If small semantics-preserving changes make models of code generate incorrect predictions, how might we train models that are robust to such changes?

GTP-3: One approach is to train models on a variety of different code samples, including those that have been slightly modified. This can help the models learn to be more robust to small changes in the code. Another approach is to use techniques like reinforcement learning, which can help models learn to make better predictions by trial and error.

Would you expect that models of code using structural representations of code (such as ASTs) are more robust compared to simpler models of code trained on the program as text?

GPT-3: It is possible that models of code using structural representations of code are more robust compared to simpler models of code trained on the program as text. This is because the structural representations of code can provide more information about the code, which can help the models learn to be more robust to small changes.

A dialogue with OpenAI's GPT-3

# 4.1 Introduction

Thus far we have described a technique for learning from programs (Chapter 2) and methods for mining specifications (usage patterns) from programs (Chapter 3). Up until this point, we have *not* considered a key aspect of learning: robustness. It has been repeatedly shown that deep neural networks are vulnerable to *adversarial examples* (Szegedy et al., 2013; Biggio et al., 2013; Goodfellow et al., 2014; Papernot et al., 2017): small, seemingly innocuous perturbations to the input that lead to incorrect predictions. For instance, making a small imperceptible modification to pixels of an image may cause a neural network to change its prediction.

### 4.1.1 Motivation

Adversarial examples raise safety and security concerns, for example, in computervision models used in autonomous vehicles (Eykholt et al., 2018; Bhagoji et al., 2018) or for user authentication (Sharif et al., 2016). Significant progress has recently been made in identifying adversarial examples and training models that are robust to such examples. However, the majority of the research has targeted computer-vision tasks (Carlini and Wagner, 2017; Madry et al., 2018; Szegedy et al., 2013), which is a *continuous* domain. (See Kolter and Madry (2020) for a comprehensive overview.)

## Motivating Question

Can we develop a method to achieve robust training in the *discrete* domain of deep neural networks for source code?

In this chapter, we study the problem of robustness to adversarial examples in the *discrete* domain of deep neural networks for source code. With the growing adoption of neural models for programming tasks (Allamanis et al., 2020, 2016b; Uri Alon and Yahav, 2019; Hellendoorn et al., 2018; Zhao and Huang, 2018; Kanade et al., 2020; Vasic et al., 2019; Pradel et al., 2020; Ahmad et al., 2020; DeFreez et al., 2018a), robustness is becoming an important property. Why do we want robust models of code? There are many answers, ranging from usability to security. Consider, for instance, a model that explains in English what a piece of code is doing—the *code-captioning* task. A developer using such a model to navigate a new code base should not receive completely different explanations for similar pieces of code. For a concrete example, consider the behavior of the state-of-the-art code2seq model of code (Alon et al., 2018a) on the Java code in Fig. 4.1, where the prediction changes after logging print statements are added. Such behavior (changing output based on irrelevant detail) is the result of an over-sensitive (and under-robust) model.

```
int □(Object target) {
 1
 2
       System.out.println("Begin search");
      int i = 0;
 3
      for (Object elem: this.elements) {
 4
 5
           if (elem.equals(target)) {
               System.out.println("Found");
 6
 7
               return i;
 8
           }
           i++;
 9
10
      }
11
      return -1;
12
    }
```

Figure 4.1: code2seq Alon et al. (2018a) correctly predicts the function name "indexOfTarget." After the highlighted logging statements are added, it predicts "search."

# 4.1.2 Goals

With images, adversarial examples involve *small* changes that are imperceptible to a human. With code, there is no analogous notion of a change imperceptible to a human. Consequently, we consider attacks based on *semantics-preserving transformations*. Because the original program's semantics is preserved, the program that results from the attack must have the same behavior as the original. Using the idea of adversarial examples generated by *semantics-preserving transformations*, we set out to meet the following goal:

#### Goals

**Our Goal.** Find a way to train robust models of code and, in doing so, build a framework that enables experimentation with different program transforms, models, datasets and programming languages.

# 4.1.3 Contributions

We developed a novel and generic adversary that can be used to facilitate robust training.

We introduced methods for robust training that can be applied in the (discrete) domain of deep neural networks for source code.

We created a framework, called AVERLOC, that has (already) allowed others to push the boundaries of adversaries in the domain of models of code.

## We report on several experiments:

- We apply individual semantics-preserving transforms and find our (parametric) adversary to be effective (RQ1). We further discover surprising differences between two common models of code (seq2seq and code2seq).
- We evaluate our robust training methodology and find it to increase model robustness and outperform data augmentation (RQ2).
- We apply progressively stronger adversaries to robust models—we find training with respect to a weaker adversary to be sufficient for defending against a stronger adversary (RQ3).
- We test the ability of robustly trained models to adapt in new domains (RQ4). We find evidence that robust training improves model adaptability.
- We test the ability of robustly trained models to transfer across programming languages (RQ5). In this task, we find inconclusive results: it seems robust training neither strictly enhances nor strictly degrades a model's ability to transfer across programming languages.

#### 4.1.4 Our Approach to Semantic Robustness

We present a novel and generic approach for defining an adversary that manipulates a program to fool a neural network. In particular, we structure an adversary in terms of two operations: *semantics-preserving transformations* and *resolvers*. Transformations construct a program *sketch* (Solar-Lezama et al., 2006)—an incomplete program with holes—and resolvers fill the holes to produce a program that fools the neural network. This insight allows us to represent a wide range of adversaries, including adversaries formulated in concurrent and follow-on work by others. Furthermore, our approach to adversaries (and the overall framework we built) has already enabled others to produce new state-of-the-art attacks. We take this as strong validation of our approach, and we hope that others will continue to build on our work to push the boundaries of adversaries in the domain of models of code.

We also demonstrate how to train models of source code that are robust to such adversaries, using *robust-optimization* ideas that are prevalent in image recognition (Madry et al., 2018). Aside from contributing a generic adversary and a robust training approach, we also contribute a framework, AVERLOC, for producing both the adversaries and the robust-training pipelines required to carry out extensive evaluations.

# 4.1.5 The AVERLOC Framework

AVERLOC provides all of the components necessary to meet our goal of training robust models of code and, as we will describe, the AVERLOC framework is even able to support the methodology used in recent (and concurrent) work by others on robust training for models of code. AVERLOC requires a user to provide data, a model, and a loss function. Given those components, AVERLOC will produce an adversary, a data augmentor, and a surrogate to an adversarial-loss function. Optionally, users can leverage our pre-existing datasets and (code-summarization) models. Note that, although we evaluate models trained to perform the code-summarization task, AVERLOC supports any model taking code as input. For more details on the AVERLOC framework, see Section 4.3.

# 4.1.6 Evaluation of Semantic Robustness

Our approach to generic adversaries and our framework (which embodies this approach) allows us to provide an extensive evaluation. In our evaluation, we answer the following five research questions (and provide concrete and actionable data to inform practitioners in their use of models of code):

### Research Question # 1

How effective are the individual transforms we provide when used as attacks?

One of our key contributions is a generic adversary for models of code built on a library of *semantics-preserving* transforms. We implement eight such transforms, and, in this research question, measure the strength of each transform in isolation against two separate model architectures.

## Research Question #2

How effective is robust training in defending against our attacks? Are there any simpler baselines that perform well?

After understanding our attacks in isolation, we move to evaluating several pipelines for defense.

## Research Question # 3

Does training with a weak adversary help with defending against a *strong* adversary?

Can we train on weaker adversaries and still retain (some) robustness when we test against much stronger adversaries?

### Research Question #4

What is the effect of robust training on the performance of models for the domain-adaptation task?

Domain adaptation asks if models trained on data from one domain can be applied to data from another different (yet similar) domain while retaining (some of) the model's original performance. To the best of our knowledge, we are the first to investigate the interplay between robustness and *domain adaptation* for models of code. If robust models perform well on unseen data (taken from sufficiently different data sources) then robust training may be desirable not only for defense against attacks, but also for increased performance in the face of unseen data.

# Research Question # 5

What is the effect of robust training on the performance of models for the cross-language-transfer task?

Again, to the best of our knowledge, we are the first to provide preliminary investigations into the effect of robust training on the cross-language-transfer task. If models can work across languages (like simple seq2seq models can), then robust training can increase their cross-language performance.

In our evaluation, we find several surprising facts: although vanilla code2seq is more robust than a simpler seq2seq baseline (before applying any kind of robust training—likely due to code2seq's use of *program structure*), it is up to 1.5x more vulnerable to some of our attacks; furthermore, to our surprise, we find that it is *harder to make code2seq robust*, which results in robustly trained seq2seq models having the best overall performance; additionally, we find that robust training beats dataset augmentation in every evaluation we performed; finally, we find that robust models perform better against unseen data from different sources—however, we also find that robust models are not clearly better in the cross-language-transfer task. In summary, we train over 32 models, perform hundreds of individual evaluations, summarize our data, and answer each of our five research questions. With these extensive results, we hope that researchers and practitioners alike can gain a better understanding of robustness in the space of models on code.

# 4.2 Semantic Robustness

In this section, we describe (1) our novel adversarial attack techniques, and (2) how to train semantically robust models for source-code tasks.

## 4.2.1 Semantic Adversaries

#### Adversaries by example

Throughout this section, we imagine a fixed deep neural network N over source code: given a piece of code P, it returns a prediction y, e.g., a textual description of what P does. The goal of an adversary is to transform P into a semantically equivalent P' that fools the neural network into making a wrong prediction. Formally, we denote an adversary as a function A(N, P); the adversary attempts to produce a program P' that is equivalent to P and makes N produce a wrong prediction.

An adversary is equipped with a set of *semantics-preserving transforms*, e.g., adding dead code or print statements. Most transforms are parametric, e.g., if one adds a print statement, one has to also decide on the text to print. Therefore, we think of a transform as producing a program *sketch* Solar-Lezama et al. (2006)—a program with holes. For example, consider the following program:

```
public void incrementWeight(double weight) {
    this.weight += weight;
  }
}
```

Applying the *insert print statement* transform produces the following sketch, where  $\circ_1$  is a hole that the adversary needs to fill with text.

```
public void incrementWeight(double weight) {
    this.weight += weight;
    System.out.println("o1");
  }
```

An adversary may decide to apply multiple transforms, for example, the one we describe above plus a transform that changes the name of function arguments. Continuing our example, this compound transform produces the following sketch with two holes  $\circ_1$  and  $\circ_2$ . (There are two occurrences of hole  $\circ_2$ .)

```
public void incrementWeight(double o<sub>2</sub>) {
   this.weight += o<sub>2</sub>;
   System.out.println("o<sub>1</sub>");
  }
```

After applying a number of transforms, the adversary needs to fill in the holes of the resulting sketch to produce a complete program that fools the neural network into changing its prediction. Our adversaries apply multiple transforms in a random order. However, it is possible to extend our adversaries such that transforms are applied according to user-supplied heuristics.

#### Adversary spectrum

We now describe how one designs an adversary algorithmically, assuming a fixed set of transforms at the adversary's disposal.

In our illustration above, we notice that an adversary has to make two choices:

- 1. **Transform**: Choose a sequence of transforms to apply to a program, resulting in a sketch.
- 2. **Resolve:** Choose values for the holes in a sketch.

The strength of an adversary depends upon how it makes these two choices. The *weakest possible adversary*, and computationally cheapest to implement, is the one that randomly chooses a sequence of transforms as well as values for sketches. The *strongest possible adversary* exhaustively tries all possible sequences of transforms and values for filling the holes in sketches, but it is intractable at best.

#### Our strong adversary

Our strongest adversary randomly chooses a sequence of transforms of a fixed length and then uses a *gradient-based* (i.e., *targeted*, as opposed to random) approach to fill the holes with tokens that are most *adversarial* to model performance. Specifically, we use an approach inspired by natural-language-processing techniques Ebrahimi et al. (2017); Yefet et al. (2020).

Using a differentiable embedding layer, we take a *gradient-ascent* step in the direction that maximizes model loss. In other words, for each distinct hole in the sketch, we pick the replacement to be the token with the maximum value (in the one-hot encoding) after the gradient-ascent step. We also impose additional semantic constraints, e.g., in sketches with multiple holes, we enforce that each hole receives distinct token replacements.

### 4.2.2 Training Semantically Robust Models

Given an adversary, how can we train models robust to adversarial transformations?

In standard neural-network training, given a dataset of programs and labels,  $(P_1, y_1), \ldots, (P_n, y_n)$ , one solves an optimization objective that looks for a neural network that minimizes average prediction *loss* on the entire dataset, where the loss function L(P, y, N) is a numerical measure of how bad the neural network N's prediction is on program P with label y. Formally, we solve the following problem:

$$\underset{N}{\arg\min} \sum_{i} L(P_{i}, y_{i}, N)$$
(4.1)

To train robust networks, we adopt a *robust-optimization* objective Madry et al. (2018), where we look for a neural network that minimizes average *adversarial loss*. For a program  $P_i$ , adversarial loss is the loss with respect to the semantically equivalent program  $P'_i$  produced by an adversary. In other words, the adversary is modeled in the optimization objective, forcing us to consider its behavior: whenever we compute the loss for a program  $P_i$ , we instead compute that of  $P'_i$ . Formally:

$$\underset{N}{\operatorname{arg\,min}} \sum_{i} L(P'_{i}, y_{i}, N), \quad \text{where } P'_{i} = A(P, N)$$
(4.2)

Robust optimization has been shown to work well in image recognition and naturallanguage processing, and, as we shall see, results in semantically robust models for source code.

# 4.3 Framework

In this section, we explore the AVERLOC framework in greater detail. We describe (1) the transforms and resolvers AVERLOC implements, (2) the training strategies it supplies, and (3) the practical challenges of robust training on source code. We also discuss related concepts like obfuscation and non-semantics-preserving transforms (mutations).

## 4.3.1 Adversaries in Detail

#### The Transforms Library

In our framework, we provide a library of transforms on which our adversaries are built. This library consists of eight transforms and two separate implementations of these eight transforms: one implementation targeting Java programs, based on Spoon (Pawlak et al., 2015), and one implementation for Python, based on Astor (Berkerpeksag, 2020). We will use the following Java program to demonstrate our transforms:

```
public int gcd(int a, int b) {
    while (b > 0) { int c = a % b; a = b; b = c; }
    if (this.log == true) { System.out.println(a,b); }
    return a;
    }
```

**T1:** AddDeadCode: A dead-code statement of the form if (false) int  $\circ_1 = 0$ ;, is appended to the beginning or end of the target program. The insertion location (beginning or end) is chosen at random. Applying AddDeadCode to our example yields:

```
public int gcd(int a, int b) {
    if (false) { int o<sub>1</sub> = 0; }
    while (b > 0) { int c = a % b; a = b; b = c; }
    if (this.log == true) { System.out.println(a,b); }
    return a;
}
```

**T2:** RenameLocalVariables: A single, randomly selected, local variable declared in the target program has its name replaced by a hole. Applying RenameLocalVariables to our example yields:

```
public int gcd(int a, int b) {
    while (b > 0) { int o<sub>1</sub> = a % b; a = b; b = o<sub>1</sub>; }
    if (this.log == true) { System.out.println(a,b); }
    return a;
    }
```

**T3:** RenameParameters: A single, randomly selected, formal parameter in the target program has its name replaced by a hole. Applying RenameParameters to our example yields:

```
public int gcd(int o<sub>1</sub>, int b) {
    while (b > 0) { int c = o<sub>1</sub> % b; o<sub>1</sub> = b; b = c; }
    if (this.log == true) {
        System.out.println(o<sub>1</sub>,b);
        }
        return o<sub>1</sub>;
    }
```

**T4: RenameFields:** A single, randomly selected, referenced field (this.field in Java, or self.field in Python) has its name replaced by a hole. Applying RenameFields to our example yields:

```
1 public int gcd(int a, int b) {
2 while (b > 0) { int c = a % b; a = b; b = c; }
3 if (this.o<sub>1</sub> == true) { System.out.println(a,b); }
4 return a;
5 }
```

**T5: ReplaceTrueFalse:** A single, randomly selected, Boolean literal is replaced by an equivalent expression containing a single hole. (One example: " $(\circ_1 == \circ_1)$ " replaces true.) Applying ReplaceTrueFalse to our example yields:

```
public int gcd(int a, int b) {
    while (b > 0) { int c = a % b; a = b; b = c; }
    if (this.log == (o<sub>1</sub> == o<sub>1</sub>)) {
        System.out.println(a,b);
    }
    return a;
    }
```

**T6: UnrollWhiles:** A single, randomly selected, while loop in the target program has its loop body unrolled exactly one step. No holes are created by this transform. Applying UnrollWhiles to our example yields:

```
1
    public int gcd(int a, int b) {
 2
      while (b > 0) {
        int c = a % b; a = b; b = c;
 3
        while (b > 0) {
 4
           int c = a \% b; a = b; b = c;
 5
 6
        }
 7
        break;
      }
 8
      if (this.log == true) { System.out.println(a,b); }
 9
10
      return a;
    }
11
```

T7: WrapTryCatch: The target program is wrapped by a single try { ... } catch (...) { ... } statement. The catch statement passes along the caught exception. A hole is used in the place of the name of the caught exception variable (e.g., catch (Exception  $\circ_1$ )). Applying WrapTryCatch to our example yields:

```
public int gcd(int a, int b) {
    try {
    while (b > 0) { int c = a % b; a = b; b = c; }
    if (this.log == true) { System.out.println(a,b); }
    return a;
    } catch (Exception o<sub>1</sub>) { }
```

**T8:** InsertPrintStatements: One print statement System.out.println(" $\circ_1$ "), in Java, or print(' $\circ_1$ ') in Python, is appended to the beginning or end of the target

program. The insertion location (beginning or end) is chosen at random. Applying InsertPrintStatements to our example yields:

```
public int gcd(int a, int b) {
    System.out.println("o1");
    while (b > 0) { int c = a % b; a = b; b = c; }
    if (this.log == true) { System.out.println(a,b); }
    return a;
    }
```

#### Resolvers

AVERLOC provides two distinct resolution strategies (resolvers). Recall that a resolver in our framework is a method that, given a program sketch, resolves the input sketch into a complete program. For our evaluation, we implemented two resolution strategies. First, we implemented a random resolver which, given a program sketch, fills all holes in the sketch with a random token generated by selecting and concatenating a random number of sub-tokens from a (provided) fixed vocabulary. Second, we implemented the gradient-based search described in Section 4.1.4.

# 4.3.2 (Robust) Optimization Objectives

Our framework enables definition of different optimization objectives for training. First, our framework can perform normal training, where the goal is to minimize a standard loss function (Eq. (4.1)). Second, our framework allows robust-optimization objectives (Eq. (4.2)) to model the adversary within the training loop.

Additionally, our framework allows for *data augmentation*, which is a standard technique where one enriches a dataset by adding random transformations of the data (e.g., rotating images using a random angle). In our framework, we can perform standard training with data augmentation by defining a completely random adversary (random choice of transforms and random resolvers) and solving, for example, the following optimization objective:

$$\underset{N}{\operatorname{arg\,min}} \sum_{i} L(P_{i}, y_{i}, N) + L(P'_{i}, y_{i}, N) \text{ where } P'_{i} = A_{R}(P)$$

$$(4.3)$$

where  $A_R$  is a random adversary, and therefore does not take the neural network as an input. Note that the transformed set  $\{P'_i\}$  is presampled before training.

#### Practical Challenges of Robust Optimization

Solving a robust-optimization objective is particularly challenging in the realm of source code. Practically, in every epoch of training, for every program  $P_i$ , we need to apply the adversary to compute a transformed program  $P'_i$ . This approach is wildly inefficient during training due to the *mismatch* between program formats for transformation and for training: a program  $P_i$  is an abstract syntax tree (AST) and the adversary's transformations are defined over ASTs, but the neural network expects as input a different representation—for example, sequences of (sub)tokens, or as in a recent popular line of work Alon et al. (2019, 2018a), a sampled set of paths from one leaf of the AST to another.

Therefore, during training, we have to translate back and forth between ASTs and their neural representation. This approach is expensive to employ during training: in every training step, we have to apply transformations using an external programanalysis tool and convert the transformed AST to its neural representation.

We address this challenge as follows: To avoid calling program-analysis tools within the training loop, we pre-generate all possible program sketches considered by the adversary. That is, for every program  $P_i$  in the training set, before training, we generate the set of program sketches that can be produced by applying sequences of transforms of a bounded length.

## 4.3.3 Related Concepts

#### Obfuscation

One might wonder how source-code obfuscation is related to our adversaries and transforms. In general, we view obfuscation techniques as off-the-shelf adversaries that one could use in our framework. However, for the sake of a precise evaluation, we wished to build adversaries into our framework that are *tunable*, for which we can create a scoring scheme suitable for use in experiments. For our adversaries, a user can select both the resolution strategy for filling holes in program sketches and the number of transforms that may be applied, in sequence, to attack input programs. Obfuscation techniques are, in a sense, "maximal" in their efforts to mask code. For our evaluation and for ease-of-use, we focus on adversaries that can be "throttled" so that we can compare the robustness of models trained under various pipelines against progressively stronger adversaries. In summary, code obfuscators make great adversaries, and we encourage their use in our framework, but, for the sake of measured evaluation, we focus not on off-the-shelf obfuscation, but on a simpler and more configurable adversary to *control for the degree of adversarial effort* when evaluating different training pipelines.

#### Mutations

Given our focus on transforms that do not change the meaning of the code they target, it is natural to ask about *mutations* or non-semantics-preserving transforms. In short, mutations are *too strong*, in some sense—that is, when using mutations we have no way to *control for the degree of adversarial effort*. In fact, mutations may change the underlying meaning (semantics) of a program and, as such, the use of mutations calls into question the objective of robustness for models of code: models given equivalent (but, syntactically diverse) programs should produce equivalent outputs.

Nonetheless, one could still ask a related question: if we mutate the code, should models not be expected to change their outputs? This is an interesting question and one we hope to explore further in follow-on work. In summary, if we focus on transforms that change the meaning of code then we must change our expectations from robustness to something closer to a "mutation adequacy" score Andrews et al. (2005) for models of code. Such issues are interesting, but lie outside the scope of our investigations on robustness.

# 4.4 Comparable Techniques

In this section, we demonstrate how two recent works on adversarial robustness over code models, Yefet et al. (2020) and Srikant et al. (2021), can be implemented using AVERLOC.<sup>1</sup> Note that these works consider a subset of the described adversaries in Section 4.3.1, but with more fine-tuned attack and defense methods. AVERLOC's adversaries are built by combining *transformations* and *resolvers*. Given transformations and resolvers, AVERLOC allows for the construction of robust training pipelines to provide *defense* against adversaries. We show how two recent works fit into the AVERLOC framework by describing their transformations, resolvers, and the defenses they provide.

## 4.4.1 DAMP

Yefet et al. (2020) present a technique, Discrete Adversarial Manipulation of Programs (DAMP), to explore the adversarial robustness of code models. DAMP differs from AVERLOC in the following ways: (1) For the attack, similar to AVERLOC, DAMP uses gradient-targeted resolution to resolve program sketches. Instead of applying greedy search to find a single resolution, DAMP uses multiple trials to perform a search for the best sketch resolution. (2) For the defense, besides adversarial (robust) training, Yefet et al. (2020) use outlier detection, which replaces an outlier variable name with UNK.

 $<sup>^{1}</sup>$ We also note that a preprint of our work appeared on arXiv earlier than Yefet et al. (2020) and Srikant et al. (2021).

#### Transformation

DAMP uses transforms similar to the RenameLocalVariables and AddDeadCode transformations already provided by AVERLOC.

#### Resolver

To provide DAMP's multiple-trial search, after computing the gradients, instead of returning only one closest token, one would return the top-k replacement tokens, and then run the network with each token. Repeating a few such gradient-and-run steps gives us the same search procedure as in DAMP.

#### Defense

Besides the robust training that has already been provided by AVERLOC, one also needs to support DAMP's outlier detector. Outlier detection is an independent feature that processes and transforms the input source code. One only needs to write a function that takes in the word embeddings of the target code-snippet and computes the distances of each variable, as shown in Eq. (6) in (Yefet et al., 2020). The most distant variable would be replaced with UNK if the distance is above some predefined threshold.

## 4.4.2 Site-Selection-Perturbation (SSP)

A new formulation of an adversarial attack on code models was proposed by Srikant et al. (2021). In addition to identifying a replacement token that leads to the adversarial prediction (using a new *resolver*), they also consider *where* in the target program they should apply their resolver. (This approach is different from our current library of transforms; our transforms produce program sketches with mostly arbitrary holes—SSP optimizes *where* transforms should produce holes in the resulting program sketch.) The source code is formulated as a sequence of tokens, and the attack is formulated as an optimization problem to select which token is to be replaced/inserted with what new token. Because of the combinatorial and structural constraints of the optimization problem, SSP proposes several optimization techniques to solve the optimization problem. Because SSP does not consider defense against adversarial attacks, to implement SSP using AVERLOC, one only needs to work on the transformation and resolution components.

#### Transformation

SSP makes use of a subset of AVERLOC's transforms, specifically: AddDeadCode, RenameLocalVariables, RenameParameters, RenameFields, ReplaceTrueFalse, and InsertPrintStatements. However, SSP "upgrades" these transforms to produce a program sketch that has *optimized holes*—that is, SSP optimizes *where* transforms change the input program.

#### Resolver

SSP implements a resolver that is similar to AVERLOC's gradient-targeted resolver. However, SSP performs a joint (alternating) optimization to find both optimal sites for applying transforms and optimal resolutions for the sketches generated by transforms. To phrase this using the terminology of AVERLOC: SSP iterates their transforms and resolver as part of an alternating optimization routine.

# 4.5 Experiments

In this section, we provide answers to five research questions, with the goal of understanding the adversaries our framework allows, their attack strength, defenses one can build via our framework, and downstream consequences for practitioners (such as performance of both normal and robust models on the domain-adaptation and cross-language-transfer tasks). To conduct our experiments, we utilize four datasets in two different languages (Java and Python): c2s/java-small, csn/java, csn/python and sri/py150. Each dataset contains around 0.5M data points (methodbody/method-name pairs). Running extensive experiments across the 4 datasets and 2 (code-summarization) models (seq2seq, code2seq) was computationally intractable, both in terms of time and space. Thus, we randomly subsample the four datasets to have train/validation/test sets of sizes 150k/10k/20k each. The datasets remain large, and we find that subsampling has a minimal effect on model performance. Furthermore, the reduced dataset sizes allowed us to perform over 160 evaluations of over 32 distinct trained models, which, even with subsampled datasets, required hundreds of hours of GPU compute time and was expensive to perform. Unless otherwise noted, we measure (changes) in our models' F1 scores. F1 is a metric that computes the harmonic mean of a model's precision and recall.

### 4.5.1 RQ1: Attack Strength

AVERLOC's generic adversary is built on a library of transformations and resolvers; how effective are individual transforms under both resolution strategies (random/gradient)?

#### Rationale

One of the reasons we use a generic adversary based on a library of transformations and resolvers is to *control for the degree of adversarial effort* and, in doing so, allow for precise experimentation. To do this effectively, we must understand how each of our eight transforms and two resolution strategies perform against our models. Furthermore, in this evaluation we also get our first insights into the effect of having a model mostly trained on *syntax* (seq2seq) versus a model trained primarily on *structure* (code2seq).

#### Metrics

We measured the *drop in F1 score* of each of our models under each combination of transformation and resolver. This drop in F1 was computed by assessing the baseline performance of a given model on its original test set and then measuring that same model against an *attacked* version of its test set. To simplify the presentation, we show data from both model architectures evaluated on a single dataset.

Table 4.1: Decreases in F1 induced by each of our eight semantics-preserving transformations paired with either random (R) or gradient-based (G) resolution strategies (measured against a normally trained baseline model on the c2s/java-small dataset). (Larger numbers indicate stronger attacks.)

	seq2seq	code2seq
Transform	$-\Delta F1 (R / G)$	$-\Delta F1 (R / G)$
AddDeadCode	4.0 / 7.7	1.4 / 2.9
RenameParameter	0.3 / 3.0	0.3 / 4.7
InsertPrintStatement	$2.7 \ / \ 6.1$	$3.8 \ / \ 10.2$
ReplaceTrueFalse	$0.0 \ / \ 0.7$	$0.2 \ / \ 0.5$
RenameField	2.3 / 5.4	2.0 / 2.0
UnrollWhile	$0.0 \ / \ 0.0$	0.4 / 0.4
RenameLocalVariable	0.3 / 2.2	0.0 / 2.5
WrapTryCatch	2.5 / 9.4	1.4 / 7.8

#### Results

Table 4.1 shows a breakdown of our results. Note that, across model architectures, the AddDeadCode, InsertPrintStatement, and WrapTryCatch transforms are particularly effective. It is also of interest to note that *gradient resolution is strictly better than random resolution* (except for UnrollWhiles, which produces a program sketch with no holes—thus, resolution has no effect). Finally, note that code2seq is, in many cases, more robust, but also, in notable cases, more susceptible to attack. In particular, we find that the InsertPrintStatement transform (with gradient-based resolution) is over 1.5 times as effective on code2seq as on seq2seq.

**RQ1 Summary.** We find that our individual attacks are effective and, between random and gradient-based resolution, find gradient-based resolution to be strictly better. Furthermore, we find a surprising fact: although code2seq is a naturally more robust architecture, it has some surprising weakness—allowing for up to 1.5x more effective attacks than a simpler seq2seq baseline, in some cases.

# 4.5.2 RQ2: Robust Training versus Baselines

How effective is robust training in *defending* against the (single-step) attacks we just examined? Are there any alternative approaches (like dataset augmentation) that perform well?

#### Rationale

Finding ways to train robust models of code is our primary goal; therefore, in this question, we seek to understand exactly which pieces of our framework are most useful in our quest to train robust models. As part of this evaluation, we test increasingly complex (and costly) training pipelines, with the hope that more sophisticated instantiations of our framework create more robust models.

#### Metrics

We trained models using three different training pipelines: (1) training with dataset augmentation, (2) robust training with an adversary configured to use any of our eight transforms and random resolution, and (3) robust training with an adversary configured to use any of our eight transforms and gradient-targeted resolution. We measured the *change in F1 score* of each of these models when attacked by an adversary using any of our eight transforms and gradient-targeted resolution. (This change is relative to a normally-trained baseline model attacked by the same adversary.)

#### Results

Table 4.2 shows the results. In general, we find that robust training using an adversary with gradient-targeted resolution is, by far, the best defense we can provide. Furthermore, we find that dataset augmentation pales in comparison to true robust training. Of particular interest is the relationship between seq2seq, code2seq, and robustness. On average, seq2seq (Normal) fares worse than code2seq (Normal) under our attack (see the first and fifth rows of Table 4.2). But, this story changes when robust training is applied: *it is harder to make code2seq robust*. After robust training,
Table 4.2: Raw F1 and change in F1 (in square brackets) for models trained using three different training pipelines in AVERLOC. The first four rows show results for seq2seq while the last four rows show results for code2seq. (Larger numbers are better.)

Training	c2s/java-small	$\cos/java$	csn/python	sri/py150
Normal	23.3	17.2	16.2	22.0
Augmented	27.8 [+4.4]	21.4 [+4.3]	20.9 [+4.7]	24.0 [+2.0]
Robust $(R)$	27.5 [+4.2]	30.1 [+12.9]	29.8 [+13.5]	33.4 [+11.4]
Robust $(G)$	$32.0 \ [+8.7]$	$32.8 \ [+15.6]$	$32.2 \ [+16.0]$	$37.1 \ [+15.1]$
Normal	24.6	19.6	21.0	23.7
Augmented	23.4 [-1.2]	19.7 [+0.1]	20.8 [-0.2]	24.4 [+0.7]
Robust $(R)$	28.1 [+3.5]	23.5 [+3.9]	22.5 [+1.6]	26.6 [+2.9]
Robust (G)	$31.6\ [+7.0]$	$27.5 \ [+7.9]$	$23.8\ [+2.8]$	$29.5 \ [+5.8]$

we find that "seq2seq (Robust (G))" ends up performing better under our attack (see the fourth and eighth rows of Table 4.2). This result was quite surprising and may be worth further study: models that are better in normal circumstances may (1) have surprising weaknesses, and (2) be harder to make robust.

**RQ2 Summary.** We find that either form of robust training was better than dataset augmentation. Furthermore, in all tested configurations, across all models and datasets, robust training with respect to an adversary using gradient-targeted resolution gave the best defense. Finally, we find that, to our great surprise, code2seq is *harder to make robust* than a seq2seq model.

## 4.5.3 RQ3: Stronger Adversaries

Does training with a "single-step" adversary improve robustness against stronger adversaries? In particular, if we train with an adversary that picks just a single (random) transform and uses gradient-targeted resolution, how well does the model perform against an adversary that is allowed to apply a *sequence of five* random transforms (also using the stronger, gradient-targeted, resolution)?

#### Rationale

In this question, we seek to understand if training with a weak adversary is "good enough"—if this is the case, then practitioners may save effort by performing robust training against weaker (and less computationally expensive) adversaries while still retaining robustness against stronger threats.

#### Metrics

We compared the *decrease in* F1 of a normally trained model and a model trained with robust-training (using an adversary that may select any single transform from our library and resolve it via gradient-targeted resolution). Here the drop in F1 was measured against a sequence of progressively stronger attacks: Nor (normal: no attack), R1 ("single-step" adversary with random resolution), R5 (an adversary allowed sequences of five random transforms, still with random resolution), G1 ("singlestep" adversary with gradient-targeted resolution), and G5 (an adversary allowed sequences of five random transforms using gradient-targeted resolution). We plotted the raw F1 scores of both our normal and robust models against these progressively stronger attacks to give a visual depiction of the robustness of each model.

#### Results

Figure 4.2 presents F1 scores from 80 distinct evaluations we performed across our four datasets, two model architectures, two training methods, and four (progressively stronger) adversaries.

There are several interesting things to learn from Fig. 4.2; first, notice that, across models, languages, and training methodology, stronger attacks induce larger drops in F1; however, as one might hope, robustly trained models (Robust: square markers) lose less F1 to a stronger adversary compared with normally trained models (Normal: circular markers). On the right side of Fig. 4.2, we make this explicit by noting that, on average, the robustly trained models retain more of their original F1 on progressively stronger attacks. Specifically, for seq2seq, the robustly trained model retains 56% more of its original performance than the normal model. Similarly, for



Figure 4.2: A comparison of a normally trained (Normal, --) model and a robustly trained model against a "single-step" gradient-targeted adversary (Robust, --). These plots show F1 scores across each of four datasets and two model architectures (seq2seq and code2seq) under a sequence of progressively stronger attacks. To the right, average decrease in F1 is shown for both Normal/Robust models in both seq2seq and code2seq architectures. (Higher F1 scores are better.)

code2seq, the robustly trained model retains 31% more of its original performance. These results are encouraging because we are attacking with a *much stronger adversary* than the one we used as part of robust training. Furthermore, the ability to carry out these measured evaluations is one of the key contributions of our framework: we can precisely control for the power of our adversaries by tuning the transforms, the allowed sequence length, and the resolution algorithm.

Finally, we note one last surprising result (that echoes what we observe in RQ2): code2seq is more difficult to make robust. This phenomenon is clearly visible in the fact that robust training has less of an effect on code2seq (31% increase in robustness, on average, under the strongest attack) compared to seq2seq (56% increase in robustness, on average, under the strongest attack). This result continues to surprise us, and is an important takeaway for practitioners of ML-on-code: we still have much to learn about how increasingly sophisticated models fare against adversarial attacks and, in our data, we find that more sophisticated models both have (1) surprising weaknesses and (2) naturally better robustness but, paradoxically, are less amenable

to techniques for increasing robustness.

We also performed an *additional* 80 evaluations on models generated via training with dataset augmentation and robust training with a (weaker still) single-step adversary using random resolution (the same training pipelines we evaluated in RQ2). As one would expect, these sit directly between the normal model (least robust) and the robustly trained model (most robust).

**RQ3 Summary.** We find that training against weaker attacks is sufficient to provide a defense against increasingly stronger attacks. Furthermore, through a series of 160 evaluations, we find confirmation of our earlier results, including the surprising fact that code2seq is less amenable to robust training than seq2seq model. Finally, we find confirmation that robust training, aside from the nuances discussed, is an effective technique across model architectures, programming languages, and datasets.

## 4.5.4 RQ4: Domain Adaptation

What is the effect of robust training on the performance of models for the domainadaptation task? For example, imagine you train a model on the code of one large company and, later, you wish to use that same model on code from another organization—will the model retain its original performance? What about a robustly trained model, will it perform better or worse?

#### Rationale

In this question, we seek to understand how both normal models of code and robustlytrained models of code *adapt* to unseen data. This data is different than simple "test-set" data because we have gone to great lengths to collect data, in both Python and Java, from different original sources that used different collection methodology. Therefore, when we apply a model trained on one Java dataset to our other Java dataset, we are getting a glimpse into how that model may perform on code that is

Model	Trained On	Tested On	F1 (seq2seq)	F1 (code2seq)
Normal	c2s/java-small	csn/java	29.35	33.31
Robust	c2s/java-small	csn/java	<b>33.05</b>	<b>35.76</b>
Normal	csn/java	c2s/java-small	34.85	39.74
Robust	csn/java	c2s/java-small	<b>38.40</b>	<b>41.12</b>
Normal	sri/py150	csn/python	19.97	34.40
Robust	sri/py150	csn/python	<b>31.93</b>	<b>35.00</b>
Normal	csn/python	sri/py150	25.15	<b>23.31</b>
Robust	csn/python	sri/py150	<b>27.74</b>	23.20

Table 4.3: F1 across both normally trained and robustly trained models on *out-of-distribution* test sets (sourced from different original data sources using differing collection methodologies). (Higher F1 scores are better.)

"different" than what it has already seen. We are not the first to study robustness and domain adaptation Volpi et al. (2018), however, to the best of our knowledge, we are the first to present such results in the space of models on code.

#### Metrics

Again, we measured F1 scores for our models. This time, we compared models trained on one of our datasets using either normal or robust training and their performance on a second dataset *from a different original source*.

#### Results

Table 4.3 presents results for both model architectures under both normal and robust training pipelines. Each row shows a single model (trained with either normal or robust training), the dataset it was trained on, the dataset it was tested on (originating from a source distinct from the training data), and the F1 scores produced by both of the seq2seq and code2seq model architectures. In general, we find confirmation that robust training improves performance on the domain-adaptation task. This result is

a useful fact for practitioners: not only does robust training strengthen your model against attack, it also provides benefits in terms of generalization. Similar to our previous research questions, we again see that code2seq benefits *less* from robust training than seq2seq does.

**RQ4 Summary.** We find strong evidence across our four datasets and two model architectures in support of robust training improving performance on the domain-adaptation task. To the best of our knowledge, we are the first to report such an effect in the space of models for code.

## 4.5.5 RQ5: Cross-Language Transfer

What is the effect of robust training on the performance of models for the crosslanguage-transfer task? Does robustness play a role in how models of code may perform on unseen languages?

#### Rationale

It seemed natural, after investigating domain adaptation, to also investigate crosslanguage transfer. One may hope that good models of code are naturally able to work across different programming languages and, therefore, it would be useful to understand the relationship between robustness and cross-language transfer.

#### Metrics

We measured F1 scores, for our seq2seq models, under both normal and robust training. We trained on data from one language (either Java or Python) and tested on data from the opposite language. We focus on seq2seq for this evaluation because code2seq cannot be trained on one language and (directly) applied to another. This test data both comes from an unseen dataset, and is in a language the model has never seen.

Model	Trained On	Tested On	F1 (seq2seq)
Normal	c2s/java-small	csn/python	19.36
Robust	c2s/java-small	csn/python	<b>22.21</b> [+2.86]
Normal	csn/java	csn/python	<b>22.15</b>
Robust	csn/java	csn/python	16.38 [-5.78]
Normal	c2s/java-small	sri/py150	21.94
Robust	c2s/java-small	sri/py150	<b>22.73</b> [+0.80]
Normal	csn/java	sri/py150	<b>23.79</b>
Robust	csn/java	sri/py150	14.54 [-9.25]

Table 4.4: F1, for our seq2seq model, across both normally trained and robustly trained models on the Java to Python *cross-language* transfer task. (Higher F1 scores are better.)

Table 4.5: F1, for our seq2seq model, across both normally trained and robustly trained models on the Python to Java *cross-language* transfer task. (Higher F1 scores are better.)

Model	Trained On	Tested On	F1 (seq2seq)
Normal	csn/python	c2s/java-small	25.23
Robust	csn/python	c2s/java-small	<b>32.13</b> [+6.90]
Normal	csn/python	csn/java	17.39
Robust	csn/python	csn/java	<b>24.02</b> [+6.63]
Normal	sri/py150	c2s/java-small	24.71
Robust	sri/py150	c2s/java-small	<b>26.59</b> [+1.88]
Normal	sri/py150	csn/java	15.06
Robust	sri/py150	csn/java	<b>16.80</b> [+1.74]

#### Results

Tables 4.4 and 4.5 show results for both Java-to-Python and Python-to-Java crosslanguage transfer. We were surprised to find that, in the case of transfer performance for models trained on Java and evaluated on Python, robust training had a clear *negative* effect—that is, normally trained models retained more of their performance on the unseen Python test sets. But, again to our surprise, we found a stronger *positive* effect for robustly trained models trained on Python and evaluated on Java. This situation is somewhat perplexing: one might hope that either robust training always improves cross-language transfer, or never does. In general, the data we collected warrants further study of the interplay between robustness and a model of code's ability to transfer across languages.

**RQ5 Summary.** We found robust training to have unclear effects on crosslanguage model transfer. In the case of training on Java and applying the learned models to Python, robust training had a *negative* effect (dropping F1, on average, 3 points); but, in the opposite task of training on Python and evaluating on Java, we found robust training to have a stronger *positive* effect (increasing F1, on average, 4 points).

# 4.6 Related Work

In concurrent work,<sup>2</sup> Bielik and Vechev (2020) combine adversarial training with abstention and AST pruning to train robust models of code. There are a number of key differences with our work: (1) We consider a richer space of transformations for the adversary, including inserting parameterized dead-code. (2) We use a strong gradient-based adversary and program sketches for completing transformations, while they use a greedy search through the space of transformations with a small number of candidates. (3) Our adversarial-training approach is more efficient, because it does not solve an expensive ILP problem to prune ASTs or train multiple models, but it is possible that we can incorporate their AST pruning in our framework.

 $<sup>^{2}</sup>$ A preprint of our work appeared earlier on arXiv than Bielik and Vechev (2020).

## 4.6.1 Adversarial Examples

In test-time attacks, an adversary perturbs an example so that it is misclassified by a model (untargeted attack) or the perturbed example is classified as an attackerspecified label (targeted) Athalye et al. (2018); Biggio et al. (2013); Ilyas et al. (2018); Chen et al. (2017); Carlini and Wagner (2017). Initially, test-time attacks were explored in the context of images. Our discrete domain is closer to test-time attacks in natural language processing (NLP). There are several test-time attacks in NLP that consider discrete transformations, such as substituting words or introducing typos Lei et al. (2019); Mudrakarta et al. (2018); Ebrahimi et al. (2017); Zhang et al. (2019); Garg and Ramakrishnan (2020). A key difference between our domain and NLP is that in the case of programs one has to worry about semantics—the program has to work even after transformations.

Recently, more consideration has been given to adversarial examples in the softwareengineering domain. Rabin et al. (2020) consider semantics-preserving transforms and their effects on various neural program analyzers (including code2seq). Compton et al. (2020) consider adversarial examples based primarily on variable renaming; they create more robust models via training with dataset augmentation. As we show experimentally, dataset augmentation does not result in robust models compared to training based on gradient-based optimization.

Many ideas from software testing, such as fuzzing and search-based techniques, have recently been successfully applied to discovering adversarial examples and other forms of bugs in neural networks Tian et al. (2020); Gao et al. (2020); Tian et al. (2018); Zhang et al. (2020). These approaches can be used to generate examples for data augmentation; however, they are generally too heavyweight to incorporate within training.

### 4.6.2 Deep Learning for Source Code

Recent years have seen huge progress in deep learning for source-code tasks—see Allamanis et al. (2018). In this chapter, we evaluate two popular models for learning from source code: seq2seq (IBM, 2020) and code2seq (Uri Alon and Yahav, 2019). The seq2seq model (sub-)tokenizes the program, analogous to NLP, and uses a variant of recurrent neural networks to generate predictions. This idea has appeared in numerous papers, e.g., the pioneering work of Raychev et al. (2014b) for code completion. We also evaluate code2seq, which uses an AST-paths encoding pioneered by Alon et al. (2019). Researchers have considered more structured networks, like graph neural networks Allamanis et al. (2016b) and tree-LSTMs Zhao et al. (2018b). These would be interesting to consider for future experimentation in the context of adversarial training. The task we evaluated on, code summarization, was first introduced by Allamanis et al. (2016b).

# 4.7 Future Work

Robustness remains an important question for models of code. In the future, it would be interesting to explore more exotic model architectures (like tree and graph-based models). Furthermore, we are in an era where the most prevalent model architectures for learning from code (and English text) are transformer based—given this, it would be worthwhile to extend this work with examinations of large transformer-based models. It may be the case that newer model architectures are innately more/less robust (similar to how we observe code2seq to be initially more robust but harder to improve than a simpler seq2seq model). A result on the innate robustness of newer architectures would have a good deal of practical significance for practitioners. Finally, it would be an interesting future direction to study *why* models of code make the predictions they do (whether those predictions are correct or incorrect). This questions is a difficult one, and I believe that, among the community, there are still many open questions about the true utility of techniques that attempt to interpret a model's predictions; nonetheless, it appears that understanding *why* a given prediction was produced would be a direct route toward improving models.

# 4.8 Notes

This work was, by far, the most expensive work I carried out during my graduate career in terms of both investment in tooling (we wrote a large framework for adversaries and robust training from scratch) and compute (we invested many hundreds of GPU hours to run the evaluations in this chapter). Overall, I find the results in this chapter to be surprising, deeply interesting, and exciting! It is rare that one discovers truly unexpected things during research and, to me, the vast difference in the two architectures we studied was both unexpected and an inversion of my own personal hypothesis: I had fully expected code2seq (which was engineered by very smart people to work specifically for code) to be the model that came out ahead on every task. But, as this chapter describes, code2seq was *surpassed* by a simpler text-based sequence-to-sequence model.

In this work I also had my first experience with how quickly the community can build on work when it is open and accessible. We made the framework we developed (AVERLOC) available on GitHub before the paper was published and, by the time we actually published these results, there were *already* new papers accepted for publication improving upon our results using our code! As a graduate student it was gratifying to produce something that was so immediately accepted and utilized by others in the community. It also was, to me, a lesson in the value of sharing code and data (although, in hindsight, I think something like Matthew Might's CRAPL (Might, 2010) should be included with all academic code—it is a bit too easy for others to assume research code is maintained and supported when, more often than not, the original authors have moved on to new projects and only answer questions and make fixes when they have the time; whether we, as a community, should better reward and incentivize the maintenance of research artifacts is a question I won't attempt to answer). Best practices for tools like Docker have arisen, but engineers are often unaware of these practices. How can we solve this problem?

GTP-3: There are a few ways to solve this problem:

- 1. Educate engineers on the best practices for using Docker. This can be done through training sessions, documentation, or both.
- 2. Implement a tool or process that enforces best practices for using Docker. This could be something as simple as a linter that checks for common errors, or a more comprehensive solution.
- 3. Use a tool that provides best practices guidance for using Docker. This could be a tool that helps you choose the right images and configuration options, or a tool that provides real-time feedback on your use of Docker.
- 4. Use a hosted solution that provides best practices for using Docker. This could be a service that helps you build and deploy your Docker containers, or a managed service that provides a complete solution for running your containers.

Wow lots of options! Let's try #2 and see how that goes.

A dialogue with OpenAI's GPT-3

# 5.1 Introduction

Thus far, we have studied how to learn from *code*—however, as discussed in Chapter 1, our overarching goal is to deal with the growth in *software*, and *software* is a composition of both code and non-code artifacts. Therefore, in this chapter, we will begin our study of non-code artifacts.

#### 5.1.1 Motivation

With the continued growth and rapid iteration of software, an increasing amount of attention is being placed on services and infrastructure to enable developers to test, deploy, and scale their applications quickly. This situation has given rise to the practice of *DevOps*, a blend of the words *Development* and *Operations*, which seeks to build a bridge between both practices, including deploying, managing, and supporting a software system (Lwakatare et al., 2015). Bass et al. define DevOps as, the "set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality" (Bass et al., 2015). DevOps activities include building, testing, packaging, releasing, configuring, and monitoring software. To aid developers in these processes, tools such as TravisCI (Travis CI, 2019), CircleCI (CircleCI, 2019), Docker (Docker, 2019), and Kubernetes (Google, 2019), have become an integral part of the daily workflow of thousands of developers. Much has been written about DevOps (see, for example, Davis and Daniels (2016) and Kim et al. (2016)) and various practices of DevOps have been studied extensively (Widder et al., 2019; Hilton et al., 2016; Ståhl and Bosch, 2016; Vasilescu et al., 2015; Zhao et al., 2017; Portworx, 2017).

DevOps tools exist in a heterogenous and rapidly evolving landscape. As software systems continue to grow in scale and complexity, so do DevOps tools. Part of this increase in complexity can be seen in the input formats of DevOps tools: many tools, like Docker (Docker, 2019), Jenkins (Jenkins, 2019), and Terraform (HashiCorp, 2019), have custom DSLs to describe their input formats. We refer to such input files as *DevOps artifacts*. To motivate our work, we pose the following question about DevOps artifacts:

#### Motivating Question

How can we aid developers in writing and maintaining DevOps artifacts?

### 5.1.2 Goals

Historically, DevOps artifacts have been somewhat neglected in terms of industrial and academic research (though they have received interest in recent years (Rahman et al., 2019)). They are not "traditional" code, and therefore out of the reach of various efforts in automatic mining and analysis, but at the same time, these artifacts are *complex*. Our discussions with developers tasked with working on these artifacts indicate that they learn just enough to "get the job done." Phillips et al. found that there is little perceived benefit in becoming an expert, because developers working on builds told them "if you are good, no one ever knows about it (Phillips et al., 2014)." However, there is a strong interest in tools to assist the development of DevOps artifacts: even with its relatively shallow syntactic support, the VS Code Docker extension has over 3.7 million unique installations (Marketplace, 2020). Unfortunately, the availability of such a tool has not translated into the adoption of best practices. We find that, on average, Dockerfiles on GitHub have nearly five times as many rule violations as those written by Docker experts. These rule violations, which we describe in more detail in Section 5.4, range from true bugs (such as simply forgetting the -y flag when using apt-get install, which can cause the build to hang) to violations of community-established best practices (such as forgetting to use apk add's -no-cache flag).

The goal of our work is as follows:

## Goals

We seek to address the need for more effective semantics-aware tooling in the realm of DevOps artifacts, with the ultimate goal of reducing the gap in quality between artifacts written by experts and artifacts found in open-source repositories.

We have observed that best practices for tools like Docker have arisen, but engineers are often unaware of these practices, and therefore unable to follow them. Failing to follow these best practices can cause longer build times and larger Docker images at best, and eventual broken builds at worst. To ameliorate this problem, we introduce **binnacle**: the first toolset for semantics-aware rule mining from, and rule enforcement in, Dockerfiles. We selected Dockerfiles as the initial type of artifact because it is the most prevalent DevOps artifact in industry (some 79% of IT companies use it (Portworx, 2017)), has become the de-facto container technology in OSS (Cito et al., 2017; Zhang et al., 2018a), and it has a characteristic that we observe in many other types of DevOps artifacts, namely, fragments of shell code are embedded within its declarative structure.

Because many developers are comfortable with the Bash shell in an interactive context, they may be unaware of the differences and assumptions of shell code in the context of DevOps tools. For example, many bash tools use a caching mechanism for efficiency. Relying on and not removing the cache can lead to wasted space, outdated packages or data, and in some cases, broken builds. Consequently, in a Dockerfile one must always invoke apt-get update before installing packages, and one should also delete the cache after installation. Default options for commands may often need to be overridden in a Docker setting. For instance, users almost always want to install recommended dependencies. However, using recommended dependencies (which may change over time in the external environment of apt package lists) can silently break future Dockerfile builds, and—in the near term—create a likely wastage of space, as well as the possibility of implicit dependencies (hence the need to use the **-no-recommends** option). Thus, a developer who may be considered a Bash or Linux expert can still run afoul of Docker Bash pitfalls.

### 5.1.3 Challenges

To create the **binnacle** toolset, we had to address three challenges associated with DevOps artifacts: (C1) the challenge of nested languages (e.g., arbitrary shell code is embedded in various parts of the artifact), (C2) the challenge of rule encoding and automated rule mining, and (C3) the challenge of static rule enforcement. As a prerequisite to our analysis, we also collected approximately 900,000 GitHub repositories, and from these repositories, captured approximately 219,000 Dockerfiles

(of which 178,000 are unique). Within this large corpus of Dockerfiles, we identified a subset written by Docker experts: this *Gold Set* is a collection of high-quality Dockerfiles that our techniques use as an oracle for Docker best practices.<sup>1</sup>

To address (C1), we introduced a novel technique for generating structured representations of DevOps artifacts in the presence of nested languages, which we call *phased parsing*. By observing that there are a relatively small number of commonly used command-line tools—and that each of these tools has easily accessible documentation (via manual/help pages)—we were able to enrich our DevOps ASTs and reduce the percentage of *effectively uninterpretable* leaves (defined in Section 5.3.1) in the ASTs by over 80%.

For the challenge of rule encoding and rule mining (C2), we took a three-pronged approach:

- 1. We introduced Tree Association Rules (TARs), and created a corpus of *Gold Rules* manually extracted from changes made to Dockerfiles by Docker experts (Section 5.3.2).
- 2. We built an automated rule miner based on frequent-sub-tree mining (Section 5.3.4).
- 3. We performed a study of the quality of the automatically mined rules using the the *Gold Rules* as our ground-truth benchmark (Section 5.4.2).

In seminal work by Sidhu et al. (2019), they attempted to learn rules to aid developers in creating DevOps artifacts, specifically TRAVIS CI files. They concluded that their "vision of a tool that provides suggestions to build CI specifications based on popular sequences of phases and commands cannot be realized." In our work, we adopt their vision, but show that it is indeed achievable. There is a simple explanation for why our results differ from theirs. In our work, we use our phased parser to go two levels deep in a hierarchy of nested languages, whereas Sidhu et al. only considered one level of nested languages. Moreover, when we mined rules, we mined

<sup>&</sup>lt;sup>1</sup>Data available at: https://github.com/jjhenkel/binnacle-icse2020

them starting with the *deepest* level of language nesting. Thus, our rules are mined from the results of a layer of parsing that Sidhu et al. did not perform, and they are mined *only* from that layer.

Finally, to address (C3), the challenge of static rule enforcement, we implemented a static enforcement engine that takes, as input, Tree Association Rules (TARs). We find that Dockerfiles on GitHub are nearly five times worse (with respect to rule violations) when compared to Dockerfiles written by experts, and that Dockerfiles collected from industry sources are no better. This gap in quality is precisely what the **binnacle** toolset seeks to address.

### 5.1.4 Contributions

In this chapter, we make four core contributions:

We created a dataset of 178,000 unique Dockerfiles, processed by our phased parser, harvested from *every public GitHub repository with 10 or more stars*,<sup>2</sup> and a toolset, called binnacle, capable of ingesting and storing DevOps artifacts.

We developed a technique for addressing the nested languages in DevOps artifacts that we call *phased parsing*.

We built an automatic rule miner, based on frequent sub-tree mining, that produces rules encoded as Tree Association Rules (TARs).

We designed a static rule-enforcement engine that takes, as input, a Dockerfile and a set of TARs and produces a listing of rule violations.

For the purpose of evaluation, we provide experimental results against Dockerfiles, but, in general, the techniques we describe in this work are applicable to any DevOps artifact with nested shell code (e.g., TRAVIS CI and CIRCLE CI). The only additional component that **binnacle** requires to operate on a new artifact type is a top-level parser capable of identifying any instances of embedded shell code. Given such a top-level parser, the rest of the **binnacle** toolset can be applied to learn rules and detect violations.

<sup>&</sup>lt;sup>2</sup>We selected repositories created after January 1st, 2007 and before June 1st, 2019.



Figure 5.1: An overview of the binnacle toolset.

Our aim is to provide help to developers in various activities. As such, binnacle's rule engine can be used to aid developers when writing/modifying DevOps artifacts in an IDE, to inspect pull requests, or to improve existing artifacts already checked in and in use.

# 5.2 Dataset

A prerequisite to analyzing and learning from DevOps artifacts is gathering a large sample of representative data. There are two challenges we must address with respect to data acquisition: (D1) the challenge of gathering *enough* data to do interesting analysis, and (D2) the challenge of gathering *high-quality* data from which we can mine rules. To address the first challenge, we created the **binnacle** toolset: a dockerized distributed system capable of ingesting a large number of DevOps artifacts from a configurable selection of GitHub repositories. **binnacle** uses a combination of Docker and Apache Kafka to enable dynamic scaling of resources when ingesting a large number of artifacts. Figure 5.1 gives an overview of the three primary tools provided by the **binnacle** toolset: a tool for data acquisition, which we discuss in this section, a tool for rule learning (discussed further in Section 5.3.4), and a tool for static rule enforcement (discussed further in Section 5.3.5).

Although the architecture of **binnacle** is interesting in its own right, we refer the reader to the **binnacle** GitHub repository for more details.<sup>3</sup> For the remainder of this section, we instead describe the data we collected using **binnacle**, and our approach to challenge (D2): the need for *high-quality* data.

Using binnacle, we ingested every public repository on GitHub with ten or more stars. This process yielded approximately 900,000 GitHub repositories. For each of these 900,000 repositories, we gathered a listing of all the files present in each repository. This listing of files was generated by looking at the HEAD of the default branch for each repository. Together, the metadata and file listings for each repository were stored in a database. We ran a script against this database to identify the files that were likely Dockerfiles using a permissive filename-based filter. This process identified approximately 240,000 likely Dockerfiles. Of those 240,000 likely Dockerfiles, only 219,000 were successfully downloaded and parsed as Dockerfiles. Of the 219,000 remaining files, approximately 178,000 were unique based on their SHA1 hash. It is this set, of approximately 178,000 Dockerfiles, that we will refer to as our corpus of Dockerfiles.

Although both the number of repositories we ingested and the number of Dockerfiles we collected were large, we still had not addressed challenge (D2): high-quality data. To find high-quality data, we looked within our Dockerfile corpus and extracted every Dockerfile that originally came from the docker-library/ GitHub organization.

<sup>&</sup>lt;sup>3</sup>https://github.com/jjhenkel/binnacle-icse2020

This organization is run by Docker, and houses a set of official Dockerfiles written by and maintained by Docker experts. There are approximately 400 such files in our Dockerfile corpus. We will refer to this smaller subset of Dockerfiles as the *Gold Set.* Because these files are Dockerfiles created and maintained by Docker's own experts, they presumably represent a higher standard of quality than those produced by non-experts. This set provides us with a solution to challenge (D2)—the Gold Set can be used as an oracle for good Dockerfile hygiene. In addition to the Gold Set, we also collected approximately 5,000 Dockerfiles from several industrial repositories, with the hope that these files would also be a source of high-quality data.

# 5.3 Technique

The **binnacle** toolset, shown in Figure 5.1, can be used to ingest large amounts of data from GitHub. This capability is of general use to anyone looking to analyze GitHub data. In this section, we describe the three core contributions of our work: phased parsing, rule mining, and rule enforcement. Each of these contributions is backed by a corresponding tool in the **binnacle** toolset: (i) phased parsing is enabled by **binnacle**'s phased parser (shown in the Learn Rules and Enforce Rules sections of Figure 5.1); (ii) rule mining is enabled by binnacle's novel frequent-subtree-based rule miner (shown in the Learn Rules section of Figure 5.1); and rule*enforcement* is provided by **binnacle**'s static rule-enforcement engine (shown in the Enforce Rules section of Figure 5.1). Each of these three tools and contributions was inspired by one of the three challenges we identified in the realm of learning from and understating DevOps artifacts (nested languages, prior work that identifies rule mining as unachievable (Sidhu et al., 2019), and static rule enforcement). Together, these contributions combine to create the **binnacle** toolset: the first structure-aware automatic rule miner and enforcement engine for Dockerfiles (and DevOps artifacts, in general).

## 5.3.1 Phased Parsing



(c) Phase II: Embedded bash is parsed

apt-get install -qqy ...

apt-get update

(d) Phase III: The AST is enriched with the results of parsing the top-50 commands

FLAG-QUIET

2

PACKAGES

PACKAGE

FLAG-YES

Figure 5.2: An example Dockerfile at each of the three phases of our phased-parsing technique (gray nodes are *effectively uninterpretable* (EU))

One challenging aspect of DevOps artifacts in general (and Dockerfiles in particular) is the prevalence of nested languages. Many DevOps artifacts have a top-level syntax that is simple and declarative (JSON, Yaml, and XML are popular choices). This top-level syntax, albeit simple, usually allows for some form of embedded scripting. Most commonly, these embedded scripts are **bash**. Further complicating matters is the fact that **bash** scripts usually reference common command-line tools, such as **apt-get** and **git**. Some popular command-line tools, like **python** and **php**, may even allow for further nesting of languages. Other tools, like GNU's **find**, allow for more **bash** to be embedded as an argument to the command. This complex nesting of different languages creates a challenge: how do we represent DevOps artifacts in a structured way?

Previous approaches to understanding and analyzing DevOps artifacts have either ignored the problem of nested languages, or only addressed one level of nesting (the embedded shell within the top-level format) (Sidhu et al., 2019; Gallaba and McIntosh, 2018). We address the challenge of structured representations in a new way: we employ phased parsing to progressively enrich the AST created by an initial top-level parse. Figure 5.2 gives an example of *phased parsing*—note how, in Fig. 5.2b, we have a shallow representation given to us by a simple top-level parse of the example Dockerfile. After this first phase, almost all of the interesting information is wrapped up in leaf nodes that are string literals. We call such nodes *effectively uninterpretable* (EU) because we have no way of reasoning about their contents. These literal nodes, which have further interesting structure, are shown in gray. After the second phase, shown in Fig. 5.2c, we have enriched the structured representation from Phase I by parsing the embedded **bash**. This second phase of parsing further refines the AST constructed for the example, but, somewhat counterintuitively, this refinement also introduces even more literal nodes with undiscovered structure. Finally, the third phase of parsing enriches the AST by parsing the options "languages" of popular commandline tools (see Fig. 5.2d). By parsing within these command-line languages, we create a representation of DevOps artifacts that contains more structured information than competing approaches.

To create our phased parser we leverage the following observations:

- 1. There are a small number of commonly used command-line tools. Supporting the top-50 most frequently used tools allows us to cover over 80% of command-line-tool invocations in our corpus.
- 2. Popular command-line tools have documented options. This documentation is easily accessible via manual pages or some form of embedded help.

Because of observation (1), we can focus our attention on the most popular command-line tools, which makes the problem of phased parsing tractable. Instead of somehow supporting all possible embedded command-line-tool invocations, we can, instead, provide support for the top-N commands (where N is determined by the amount of effort we are willing to expend). To make this process uniform and simple, we created a parser generator that takes, as input, a declarative schema for the options language of the command-line tool of interest. From this schema, the parser generator outputs a parser that can be used to enrich the ASTs during Phase III of parsing. The use of a parser generator was inspired by observation (2): the information available in manual pages and embedded help, although free-form English text, closely corresponds to the schema we provide our parser generator. This correspondence is intentional. To support more command-line tools, one merely needs to identify appropriate documentation and transliterate it into the schema format we support. In practice, creating the schema for a typical command-line tool took us between 15 and 30 minutes. Although the parser generator is an integral and interesting piece of infrastructure, we forego a detailed description of the input schema the generator requires and the process of transliterating manual pages; instead, we now present the rule-encoding scheme that **binnacle** uses both for rule enforcement and rule mining.

### 5.3.2 Tree Association Rules (TARs)

The second challenge the **binnacle** toolset seeks to address (rule encoding) is motivated by the need for both automated rule mining and static rule enforcement. In both applications, there needs to be a consistent and powerful encoding of expressive rules with straightforward syntax and clear semantics. As part of developing this encoding, we curated a set of *Gold Rules* and wrote a rule-enforcement engine capable of detecting violations of these rules. We describe this enforcement engine in greater detail in Section 5.3.5. To create the set of *Gold Rules*, we returned to the data in our Gold Set of Dockerfiles. These Dockerfiles were obtained from the docker-library/ organization on GitHub. We manually reviewed merged pull requests to the repositories in this organization. From the merged pull requests, if we thought that a change was applying a best practice or a fix, we manually formulated, as English

Table 5.1: Detailed breakdown of the *Gold Rules*. (All rules are listed; the rules that passed confidence/support filtering, described in Section 5.3.5, are shaded.) Note the following abbreviations: Gold Set (GS), incressed attack surface (IAS), and easier to add future bugs (ETAFB).

Rule Name	Bash Best- practice?	Immediate/Future Violation Consequences	GS Support (GS Confidence)
pipUseCacheDir	No	Space wastage / IAS	15 (46.67%)
npmCacheCleanUseForce	No	Space wastage / IAS	14~(57.14%)
mkdirUsrSrcThenRemove	Yes	Space wastage / IAS	129~(68.99%)
$\rm rmRecurisveAfterMktempD$	Yes	Space wastage / IAS	632~(77.22%)
curlUseFlagF	No	None / ETAFB	72 (77.78%)
tarSomethingRmTheSomething	Yes	Space wastage / IAS	209~(88.52%)
apkAddUseNoCache	No	Space wastage / IAS	250~(89.20%)
aptGetInstallUseNoRec	No	Space wastage / Build failure	525~(90.67%)
curlUseHttpsUrl	Yes	Insecure / Insecure	57~(92.98%)
gpgUseBatchFlag	No	Reliability / Reliability	455 (94.51%)
sha256sumEchoOneSpace	Yes	Build failure / None	132 (95.45%)
gpgUseHaPools	No	Reliability / Reliability	205~(97.07%)
configureUseBuildFlag	No	None / ETAFB	128 (98.44%)
wgetUseHttpsUrl	Yes	Insecure / Insecure	290 (98.97%)
aptGetInstallRmAptLists	No	Space wastage / IAS	525~(99.43%)
aptGetInstallUseY	No	Build failure / None	525~(100.00%)
aptGetUpdatePrecedesInstall	No	Build failure / None	525~(100.00%)
gpgVerifyAscRmAsc	Yes	Space wastage / IAS	172~(100.00%)
npmCacheCleanAfterInstall	No	Space wastage / IAS	12(100.00%)
gemUpdateSystemRmRootGem	No	Space wastage / IAS	11 (100.00%)
gemUpdateNoDocument	No	Space wastage / IAS	11 (100.00%)
yumInstallForceYes	No	Build failure / None	3~(100.00%)
yum Install Rm Var Cache Yum	No	Space wastage / IAS	3~(100.00%)

prose, a description of the change. This process gave us approximately 50 examples of *concrete changes made by Docker experts*, paired with descriptions of the general pattern being applied.

From these concrete examples, we devised 23 rules. A summary of these rules is given in Table 5.1. Most examples that we saw could be framed as association rules of some form. As an example, a rule may dictate that using apt-get install . . . requires a preceding apt-get update. Rules of this form can be phrased in terms of an antecedent and consequent. The only wrinkle in this simple approach is that both the

Ρ	RECEDES
	(APT-GET-INSTALL)
	(APT-GET-UPDATE)

(a) Intuitively, this rule states that an apt-get install must be preceded (in the same layer of the Dockerfile) by an apt-get update.

Follows	
(APT-GET-INSTALL)	
(RM (RM-F-RECURSIVE) (RM-PATH (ABS-APT-LISTS)))	•

(b) Intuitively, this rule states that a certain directory must be removed (in the same layer of the Dockerfile) following an apt-get install.

CHILD-OF (APT-GET-INSTALL [\*]) (FLAG-NO-RECOMMENDS)

(c) Here, the user must select where, in the antecedent subtree, to bind a region to search for the consequent. This binding is represented by the [\*] marker.

Figure 5.3: Three example Tree Association Rules (TARs). Each TAR has, above the bar, an antecedent subtree encoded as an S-expression and, below the bar, a consequent subtree encoded in the same way.

antecedent and the consequent are sub-trees of the tree representation of Dockerfiles. To deal with tree-structured data, we specify two pieces of information that helps restrict *where* the consequent can occur in the tree, relative to the antecedent:

- 1. Its location: the consequent can either (i) *precede* the antecedent, (ii) *follow* the antecedent, or (iii) *be a child of* the antecedent in the tree.
- 2. Its scope: the consequent can either be (i) in the *same piece* of embedded shell as the antecedent (intra-directive), or (ii) it can be allowed to exist in a *separate piece* of embedded shell (inter-directive). Although we can encode and enforce inter-directive rules, our miner is only capable of returning intra-directive rules



(a) Four sub-tree instances with root APT-GET-INSTALL. binnacle uses a frequent sub-tree miner, with a support threshold of 75%, to identify frequently occurring sub-trees. We have highlighted two such possible frequent sub-trees in gray and dashed outlines, respectively.

	CHILD-OF (APT-GET-INSTALL [*])
APT-GET-INSTALL	(FLAG-NO-RECOMMENDS)
FLAG-NO-RECOMMENDS	CHILD-OF
PACKAGE	(APT-GET-INSTALL [*])
	(FLAG-YES) (PACKAGES (PACKAGE))
	(c) Tree Association Rules created
(b) The two frequently occuring sub-	automatically from the likely con-
trees extracted from the example in-	sequents in Figure 5.4b. The an-
put corpus in Figure 5.4a: these trees	tecedent denotes the set of all sub-

Figure 5.4: A depiction of rule mining in **binnacle** via frequent sub-tree mining.

type.

become likely consequents.

trees with the indicated root node-



Figure 5.5: An example of the abstraction process.

(as explained in Section 5.3.4). Therefore, all of the rules we show have an intra-directive scope.

From an antecedent, a consequent, and these two pieces of localizing information, we can form a complete rule against which the enriched ASTs created by the phased parser can be checked. We call these Tree Association Rules (TARs). Three example TARs are given in Figure 5.3. We are not the first to propose Tree Association Rules; Mazuran et al. (2009) proposed TARs in the context of extracting knowledge from XML documents. The key difference is that their TARs require that the consequent be a child of the antecedent in the tree, while we allow for the consequent to occur outside of the antecedent, either preceding it or succeeding it. Although we allow for this more general definition of TARs, our miner is only capable of mining *local* TARs—that is, TARs in the style of Mazuran et al. (2009); however, our static rule-enforcement engine has no such limitation.

**Rule impacts.** For each of the Gold rules, Table 5.1 provides the consequences of a rule violation and a judgement as to whether a given rule is unique to Dockerfiles or more aligned with general Bash best-practices. In general, we note that rule violations have varying consequences, including space wastage, container bloat (and consequent increased attack surface), and instances of outright build failure. Additionally, two-thirds of the Gold rules are *unique to using Bash in the context of a Dockerfile*.



Figure 5.6: binnacle's rule engine applied to an example Dockerfile

#### 5.3.3 Abstraction

**binnacle**'s rule miner and static rule-enforcement engine both employ an *abstraction* process. The abstraction process is complementary to *phased parsing*—there may still be information within literal values even when those literals are not from some well-defined sub-language. During the abstraction process, for each tree in the input corpus, every literal value residing in the tree is removed, fed to an abstraction subroutine, and replaced by either zero, one, or several abstract nodes (these abstract nodes are produced by the abstraction subroutine). The abstraction subroutine simply applies a user-defined list of named regular expressions to the input literal value. For every matched regular expression, the abstraction subroutine returns an abstract node whose type is set to the name of the matched expression. For example, one abstraction we use attempts to detect URLs; another detects if the literal value is a Unix path and, if so, whether it is relative or absolute. The abstraction process is depicted in Figure 5.5. The reason for these abstractions is to help both **binnacle**'s rule-learning and static-rule-enforcement phases by giving these tools the vocabulary necessary to reason about properties of interest.

## 5.3.4 Rule Mining

The **binnacle** toolset approaches rule mining by, first, focusing on a specific class of rules that are more amenable to automatic recovery: rules that are *local*. We define a *local* Tree Association Rule (TAR) as one in which the consequent sub-tree exists within the antecedent sub-tree. This matches the same definition of TARs introduced by Mazuran et al. (2009). Based on this definition, we note that local TARs must be intra-directive (scope) and must be child-of (location). Three examples of local TARs (each of which our rule miner is able to discover automatically) are given in Figures 5.3c and 5.4c. In general, the task of finding arbitrary TARs from a corpus of hundreds of thousands of trees is computationally infeasible. By focusing on local TARs, the task of automatic mining becomes tractable.

To identify local TARs binnacle collects, for each node type of interest, the set of all sub-trees with roots of the given type (e.g., all sub-trees with APT-GET as the root). On this set of sub-trees, **binnacle** employs frequent sub-tree mining (Chi et al., 2005a) to recover a set of likely consequents. Specifically, binnacle uses the CMTREEMINER algorithm (Chi et al., 2005b) to identify frequent *maximal, induced, ordered* sub-trees. *Induced* indicates that all "child-of" relationships in the sub-tree exist in the original tree (as opposed to the more permissive "descendent-of" relationship, which defines an *embedded* sub-tree). *Ordered* signifies that order of the child nodes in the sub-tree matters (as opposed to *unordered* sub-trees). A frequent sub-tree is *Maximal* for a given support threshold if there is no super-tree of the sub-tree with occurrence frequency above the support threshold (though there may be sub-trees of the given sub-tree that have a higher occurrence frequency). For more details on frequent sub-trees, see Chi et al. Chi et al. (2005a).

binnacle returns rules in which the antecedent is the root node of a sub-tree (where the type of the root node matches the input node-type) and the consequent is a sub-tree identified by the frequent sub-tree miner.

An example of the rule-mining process is given in Figure 5.4. In the first stage of rule mining, all sub-trees with the same root node-type (APT-GET-INSTALL) are grouped together and collected. For each group of sub-trees with the same root node-type, binnacle employs frequent sub-tree mining to find likely consequents. In our example, two frequently occurring sub-trees (in gray and dashed outlines, respectively) are given in Figure 5.4b. Finally, binnacle creates local TARs by using the root node as the antecedent and each of the frequent sub-trees as a consequent, as shown in Figure 5.4c. One TAR is created for each identified frequent sub-tree.

#### 5.3.5 Static Rule Enforcement

Currently, the state-of-the-art in static Dockerfile support for developers is the VSCode Docker extension (Microsoft, 2019) and the Hadolint Dockerfile-linting tool (Hadolint, 2019). The VSCode extension provides highlighting and basic linting, whereas Hadolint employs a shell parser (ShellCheck (Holen, 2019)—the same shell parser we use) to parse embedded **bash**, similar to our tool's second phase of parsing. The capabilities of these tools represent steps in the right direction but, ultimately,



(a) Density histogram of M1 (the fraction of leaves that are EU after the first phase of parsing). On average, 19.3% of leaves (median 16.7%) are EU at this phase.

(b) Density histogram of M2 (the fraction of leaves that are EU after the second phase of parsing). On average, 33.2% of leaves (median 33.3%) are EU at this phase.



after the second phase of parsing and unresolved in the third phase). On average, just 3.7% of leaves (median 0.0%) remained *EU*.

Figure 5.7: Density histograms showing the distributions of our three metrics (M1,M2, and M3). The green shaded box in each plot highlights the interquartile range for each distribution (the middle 50%). they do not offer enough in the way of deep semantic support. Hadolint does not support parsing of the arguments of individual commands as **binnacle** does in its third phase of parsing. Instead, Hadolint resorts to fuzzy string matching and regular expressions to detect simple rule violations.

binnacle's static rule-enforcement engine takes, as input, a Dockerfile and a set of TARs. binnacle's rule engine runs, for each rule, three stages of processing on the input corpus:

- 1. Stage I: The Dockerfile is parsed into a tree representation, and the enforcement engine attempts to match the TAR's antecedent (by aligning it with a sub-tree in the input tree). If no matches are found, the engine continues processing with the next TAR. If a match is found, then the enforcement engine continues to Stage II. This process is depicted in Figure 5.6a.
- 2. Stage II: Depending on the *scope* and *location* of the given TAR, the enforcement engine binds a region of the input tree. This region is where, in Stage III, the enforcement engine will look for a sub-tree with which the consequent can be aligned. Figure 5.6b depicts this process, and highlights the various possible binding regions in the example input tree.
- 3. Stage III: Given a TAR with a matched antecedent and a bound region of the input tree, the enforcement engine attempts to align the consequent of the TAR with a sub-tree within the bound region. If the engine is able to find such an alignment, then the rule has been *satisfied*. If not, the rule has been *violated*. Figure 5.6c depicts this process and both possible outcomes: for the rule in Figure 5.3a, the matched antecedent is shown with a thick black outline, the bound region is shown in blue, and the matched consequent is shown with a dashed black outline. In contrast, for the rule in Figure 5.3c, the matched antecedent is the same as above, the bound region is shown in green; however, the tree is missing the consequent, represented by the dashed red sub-tree.

The implementation of **binnacle**'s enforcement engine utilizes a simple declarative encoding for the TARs. To reduce the bias in the manually extracted *Gold Rules* 

(introduced in Section 5.3.2), we used **binnacle**'s static rule-enforcement engine and the Gold Set of Dockerfiles (introduced in Section 5.2) to gather statistics that we used to filter the Gold Rules. For each of the 23 rules (encoded as Tree Association Rules), we made the following measurements: (i) the support of the rule, which is the number of times the rule's antecedent is matched, (ii) the *confidence* of the rule, which is the percentage of occurrences of the rule's consequent that match successfully, given that the rule's antecedent matched successfully, and (iii) the *violation rate* of the rule, which is the percentage of occurrences of the antecedent where the consequent is not matched. Note that our definitions of *support* and *confidence* are the same as that used in traditional association rule mining (Agrawal et al., 1993). We validated our Gold Rules by keeping only those rules with support greater than or equal to 50 and *confidence* greater than or equal to 75% on the *Gold Set*. These support and confidence measurements are given in Table 5.1. By doing this filtering, we increase the selectivity of our *Gold Rules* set, and reduce the bias of our manual selection process. Of the original 23 rules in our *Gold Rules*, 16 pass the minimum-support threshold and, of those 16 rules, 15 pass the minimum-confidence threshold. Henceforth, we use the term Gold Rules to refer to the 15 rules that passed quantitative filtering. These 15 rules are highlighted, in gray, in Table 5.1.

Together, **binnacle**'s phased parser, rule miner, and static rule-enforcement engine enable both rule learning and the enforcement of learned rules. Figure 5.1 depicts how these tools interact to provide the aforementioned features. Taken together, the **binnacle** toolset fills the need for structure-aware analysis of DevOps artifacts and provides a foundation for continued research into improving the state-of-the-art in learning from, understanding, and analyzing DevOps artifacts.

# 5.4 Experiments

In this section, for each of the three core components of the **binnacle** toolset's learning and enforcement tools, we measure and analyze quantitative results relating to the efficacy of the techniques behind these tools. All experiments were performed on a 12-core workstation (with 32GB of RAM) running Windows 10 and a recent

CHILD-OF	CHILD-OF	
(APK-ADD [*])	(SC-CURL-URL [*])	
(FLAG-NO-CACHE)	(ABS-URL-PROTOCOL-HTTPS)	
(a) A Gold rule	(b) A Semantic rule	
CHILD-OF	CHILD-OF	
(CP [*])	(SED [*])	
(CP-PATH) (CP-PATH)	(FLAG-IN-PLACE)	
(c) A Syntactic rule	(d) An Ungeneralizable rule	

Figure 5.8: Four examples of actual rules recovered by **binnacle**'s automated miner. Through abstraction, interesting semantic rules, such as using HTTPS URLs with curl, are captured.

version of Docker.

## 5.4.1 Results: Phased Parsing

To understand the impacts of phased parsing, we need a metric for quantifying the amount of useful information present in our DevOps artifacts (represented as trees) after each stage of parsing. The metric we use is the fraction of leaves in our trees that are effectively uninterpretable (EU). We define a leaf as effectively uninterpretable (EU) if it is, after the current stage of parsing, a string literal that could be further refined by parsing the string with respect to the grammar of an additional embedded language. (We will also count nodes explicitly marked as unknown by our parser as being EU.) For example, after the first phase of parsing (the top-level parse), a Dockerfile will have nodes in its parse tree that represent embedded bash—these nodes are EU at this stage because they have further structure that can be discovered given a bash parser; however, after the first stage of parsing, these leaves are simply treated as literal values, and therefore marked EU.

We took three measurements over the corpus of 178,000 unique Dockerfiles introduced in Section 5.2: (M1) the distribution of the fraction of leaves that are EU after the first phase of parsing, (M2) the distribution of the fraction of leaves that are EU after the second phase of parsing, and (M3) the distribution of the fraction of leaves that are EU after the second phase of parsing and unresolved during the third phase of parsing.<sup>4</sup>

Density histograms that depict the three distributions are given in Fig. 5.7. As shown in Fig. 5.7, after the first phase of parsing, the trees in our corpus have, on average, 19.3% *EU* leaves. This number quantifies the difficulty of reasoning over DevOps artifacts without more sophisticated parsing. Furthermore, the nodes in the tree most likely to play a role in rules happen to be the *EU* nodes at this stage. (This aspect is something that our quantitative metric does not take into account and hence over-estimates the utility of the representation available after Phase I and Phase II.)

Counterintuitively, the second phase of parsing makes the situation worse: on average, 33.2% of leaves in second-phase trees are EU. Competing tools, like Hadolint, work over DevOps artifacts with a similar representation. In practice, competing tools must either stay at what we consider a Phase I representation (just a top-level parse) or utilize something similar to our Phase II representations. Such tools are faced with the high fraction of EU leaves present in a Phase II AST. Tools using Phase II representations, like Hadolint, are forced to employ regular expressions or other fuzzy matching techniques as part of their analysis.

Finally, we use our parser generator and the generated parsers for the top-50 commands to perform a third phase of parsing. The plot in Fig. 5.7c shows the M3 distribution obtained after performing the third parsing phase on our corpus of Dockerfiles. At this stage, almost all of the EU nodes are gone—on average, only 3.7% of leaves that were EU at Phase II remain EU in Phase III. In fact, over 65%

<sup>&</sup>lt;sup>4</sup>For (M3) we make a relative measurement: the reason for using a different metric is to accommodate the large number of new leaf nodes that the third phase of parsing introduces. Without this adjustment, one could argue that our measurements are biased because the absolute fraction of EU leaves would be low due to the sheer number of new leaves introduced by the third parsing phase. To avoid this bias, we measure the fraction of *previously EU* leaves that remain unresolved, as opposed to the absolute fraction of EU leaves that remain after the third phase of parsing (which is quite small due to the large number of new leaves introduced in the third phase).

of trees in Phase II had all EU leaves resolved after the third phase of parsing. These results provide concrete evidence of the efficacy of our phased-parsing technique, and, in contrast to what is possible with existing tools, the Phase III structured representations are easily amenable to static analysis and rule mining.

### 5.4.2 Results: Rule Mining

We applied **binnacle**'s rule miner to the Gold Set of Dockerfiles defined in Section 5.2. We chose the Gold Set as our corpus for rule learning because it presumably contains Dockerfiles of high quality. As described in Section 5.3.4, **binnacle**'s rule miner takes, as input, a corpus of trees and a set of node types. We chose to mine for patterns using any new node type introduced by the third phase of parsing. We selected these node types because (i) they represent new information gained in the third phase of our phased-parsing process, and (ii) all rules in our manually collected *Gold Rules* set used nodes created in this phase. Rules involving these new nodes (which come from the most deeply nested languages in our artifacts) were invisible to prior work.

To evaluate **binnacle**'s rule miner, we used the *Gold Rules* (introduced in Section 5.3.2). From the original 23 *Gold Rules* we removed 8 rules that did not pass a set of quantitative filters—this filtering is described more in Section 5.3.5. Of the remaining 15 *Gold Rules*, there are 9 rules that are *local* (as defined in Section 5.3.4). In principal, these 9 rules are all extractable by our rule miner. Furthermore, it is conceivable that there exist interesting and useful rules, outside of the *Gold Rules*, that did not appear in the dockerfile changes that we examined in our manual extraction process. To assess **binnacle**'s rule miner we asked the following three questions:

- (Q1) How many rules are we able to extract from the data automatically?
- (Q2) How many of these rules match one of the 9 local *Gold Rules*? (Equivalently, what is our *recall* on the set of local *Gold Rules*?)
- (Q3) How many new rules do we find, and, if we find new rules (outside of our local *Gold Rules*), what can we say about them (e.g., are the new rules useful, correct, general, etc.)?
For (Q1), we found that **binnacle**'s automated rule miner returns a total of 26 rules. **binnacle**'s automated rule miner is selective enough to produce a small number of output rules—this selectivity has the benefit of allowing for easy manual review.

To provide a point of comparison, we also ran a traditional association rule miner over sequences of tokens in our Phase III ASTs (we generated these sequences via a pre-order traversal). The association rule miner returned thousands of possible association rules. The number of rules could be reduced, by setting very high confidence thresholds, but in doing so, interesting rules could be missed.

For (Q2), we found that two thirds (6 of 9) local *Gold Rules* were recovered by **binnacle**'s rule miner. Because **binnancle**'s rule miner is based on frequent sub-tree mining, it is only capable of returning rules that, when checked against the corpus they were mined from, have a minimum confidence equal to the minimum support supplied to the frequent sub-tree miner.

In addition to measuring recall on the local *Gold Rules*, we also examined the rules encoded in Hadolint to identify all of its rules that were local. Because Hadolint has a weaker representation of Dockerfiles, we are not able to translate many of its rules into local TARs. However, there were three rules that fit the definition of local TARs. Furthermore, binnacle's automated miner was able to recover each of those three rules (one rule requires the use of apt-get install's -y flag, another requires the use of apt-get install's -no-install-recommends flag, and the third requires the use of apk add's -no-cache flag).

To classify the rules returned by our automated miner, we assigned one of the following four classifications to each of the 26 rules returned:

- Syntactic: these are rules that enforce simple properties—for example, a rule encoding the fact that the cp command takes two paths as arguments (see Figure 5.8c).
- Semantic: these are rules that encode more than just syntax. For example, a rule that says the URL passed to the curl utility must include the https://prefix (see Figure 5.8b).

- Gold: these are rules that match, or supersede, one of the rules in our collection of *Gold Rules* (see Figure 5.8a).
- Ungeneralizable: these are rules that are correct on the corpus from which they were mined, but, upon further inspection, seem unlikely to generalize. For example, a rule that asserts that the sed utility is always used with the -in-place flag is ungeneralizable (see Figure 5.8d).

To answer (Q3), we assigned one of the above classifications to each of the automatically mined rules. We found that, out of 26 rules, 12 were syntactic, 4 were semantic, 6 were gold, and 4 were ungeneralizable. Figure 5.8 depicts a rule that was mined automatically in each of the four classes. Surprisingly, binnacle's automated miner discovered 16 new rules (12 syntactic, 4 semantic) that we missed in our manual extraction. Of the newly discovered rules, one could argue that only the semantic rules are interesting (and, therefore, one might expect a human to implicitly filter out syntactic rules during manual mining). We would argue that even these syntactic rules are of value. The lack of basic validation in tools like VS Code's Docker extension creates a use case for these kind of basic structural constraints. Furthermore, the 4 novel semantic rules include things such as: (i) the use of the -L flag with curl, following redirects, which introduces resilience to resources that may have moved, (ii) the use of the -p flag with mkdir, which creates nested directories when required, and (iii) the common practice of preferring soft links over hard links by using ln's -s flag. With (Q3), we have demonstrated the feasibility of automated mining for Dockerfiles—we hope that these efforts inspire further work into mining from Dockerfiles and DevOps artifacts in general.

### 5.4.3 Results: Rule Enforcement

Using the 15 Gold Rules, we measured the average violation rate of the Gold Rules with respect to the Gold Dockerfiles (Section 5.2). The average violation rate is the arithmetic mean of the violation rates of each of the 15 Gold Rules with respect to the Gold Dockerfiles. This measurement serves as a kind of baseline—it gives us a sense

of how "good" Dockerfiles written by experts are with respect to the *Gold Rules*. The average violation rate we measured was 6.65%, which, unsurprisingly, is quite low. We also measured the average violation rate of the *Gold Rules* with respect to our overall corpus. We hypothesized that Dockerfiles "in the wild" would fare worse, with respect to violations, than those written by experts. This hypothesis was supported by our findings: the average violation rate was 33.15%. We had expected an increase in the violation rate, but were surprised by the magnitude of the increase. These results highlight the dire state of static DevOps support: Dockerfiles authored by experts. Bridging this gap is one of the overarching goals of the **binnacle** ecosystem.

We also obtained a set of approximately 5,000 Dockerfiles from the source-code repositories of an industrial source, and assessed their quality by checking them against our *Gold Rules*. To our surprise, the violation rate was no lower for these industrial Dockerfiles. This result provides evidence that the quality of Dockerfiles suffers in in

## 5.5 Related Work

Our paper is most closely related to the work of Sidhu et al. (2019), who explored reuse in CI specifications in the specific context of TRAVIS CI, and concluded that there was not enough reuse to develop a "tool that provides suggestions to build CI specifications based on popular sequences of phases and commands." We differ in the DevOps artifact targeted (Dockerfiles versus TRAVIS CI files), representation of the configuration file, and the rule-mining approach.

In a related piece of work, Gallaba and McIntosh (2018) analyzed the use of TRAVIS CI across nearly 10,000 repositories in GitHub, and identified best practices based on documentation, linting tools, blog posts, and stack-overflow questions. They used their list of best practices to deduce four anti-patterns, and developed HANSEL, a tool to identify anti-patterns in TRAVIS CI config files, and GRETEL, a tool to automatically correct them. Similar to our second phase of parsing, they used a bash parser (BASHLEX) to gain a partial understanding of the shell code in config files.

Zhang et al. (2018b) examined the impact of changes to Dockerfiles on build time and quality issues (via the Docker linting tool Hadolint). They found that fewer and larger Docker layers results in lower latency and fewer quality issues in general, and that the architecture and trajectory of Docker files (how the size of the file changes over time) impact both latency and quality. Many of the rules in our Gold Set, and those learned by **binnacle**, would result in lower latency and smaller images if the rules were followed.

Xu et al. (2019) described a specific kind of problem in Docker image creation that they call the "Temporary File Smell." Temporary files are often created but not deleted in Docker images. They present two approaches for identifying such temporary files. In this paper, we also observed that removing temporary files is a best-practice employed by Dockerfile experts and both our manual Gold Set and our learned rules contained rules that address this.

Zhang et al. (2018a) explored the different methods of continuous deployment (CD) that use containerized deployment. While they found that developers see many benefits when using CD, adopting CD also poses many challenges. One common way of addressing them is through containerization, typically using Docker. Their findings also reinforce the need for developer assistance for DevOps: they concluded that "Bad experiences or frustration with a specific CI tool can turn developers away from CI as a practice."

Our work falls under broader umbrella of "infrastructure-as-code". This area has received increasing attention recently Rahman et al. (2019). As examples, Sharma et al. examined quality issues, so-called *smells*, in software-configuration files (Sharma et al., 2016), and Jiang et al. examined the coupling between infrastructure-as-code files and "traditional" source-code files.

There have been a number of studies that mine Docker artifacts as we do. Xu and Marinov (2018) mined container-image repositories such as DockerHub, and discussed the challenges and opportunities that arise from such mining. Zerouali et al. (2019) studied vulnerabilities in Docker images based on the versions of packages installed in them. Guidotti et al. (2018) attempted to use Docker-image metadata to determine if certain combinations of image attributes led to increased popularity in terms of stars and pulls. Cito et al. (2017) conducted an empirical study of the Docker ecosystem on GitHub by mining over 70,000 Docker files, and examining how they evolve, the types of quality issues that arise in them, and problems when building them.

A number of tools related to Dockerfiles have been developed in recent years as well.

Brogi et al. (2017) found that searching for Docker images is currently a difficult problem and limited to simple keyword matching. They developed DOCKERFINDER, a tool that allows multi-attribute search, including attributes such as image size, software distribution, or popularity.

Yin et al. (2018) posited that tag support in Docker repositories would improve reusability of Docker images by mitigating the discovery problem. They addressed this issue by building STAR, a tool that uses latent dirichlet allocation to automatically recommend tags.

Docker files may need to be updated when the requirements of the build environment or execution environment changes. Hassan et al. (2018a) developed RUDSEA, a tool that can recommend updates to Dockerfiles based on analyzing changes in assumptions about the software environment and identifying their impacts.

To tackle the challenge of creating the right execution environment for python code snippets (e.g., from Gists or StackOverflow) Horton and Parnin (2019) developed DOCKERIZEME, a tool which infers python package dependencies and automatically generates a Dockerfile that will build an execution environment for pieces of python code.

## 5.6 Future Work

In this work, I took a more traditional approach to "learning" by using a frequentsub-tree mining technique. I think there is a great opportunity here to think about truly learned approaches. For instance, imagine training a "tree-language model" on the corpus of parsed Dockerfiles. If one were to train such a model and make use of an architecture that has some concept of "attention" (that is, the ability to weight different parts of the input) then one could devise a mining technique that learns from the model—by placing holes in trees, asking the model to make predictions, and then looking at what pieces of the input tree the model attends to (for example, the model may always attend to apt-get update when predicting apt-get install). This is a very loose sketch of an idea, and I believe there would be many nuances to making such a technique work, but it would be an interesting direction for future work to consider such an approach to mining from tree-structured artifacts (such as code or Dockerfiles).

# 5.7 Notes

The tools, experiments, and techniques in this work were completed while I interned at Microsoft Research in the summer of 2019. I remember colleagues (wisely) telling me to enjoy the summer and not worry to much about trying to churn out a piece of work in the (less than) three months we interns had. I also remember seeing a unique opportunity with Dockerfiles and realizing that producing this work would require a bit of innovation but, mostly, a lot of brute-force coding work. I remember spending weekends at a local coffee shop working on the third-level parsers and how, slowly, each new parser started to unlock more options for mining meaningful specifications. Outside of the hard work, I also remember how much *fun* it was to be an intern at MSR—they took us sailing (which I had never done), hiking by waterfalls, wine tasting, and gave us countless opportunities to interact with incredible researchers and hear lectures on many diverse subjects throughout the summer. I think that summer was a defining moment in my career as a researcher—it was hard to be away from home, away from my wife and two wonderful (and mischievous) cats, but it was well worth it. Over a quarter of the GitHub repositories with Dockerfiles I looked at had a broken Dockerfile! What should I do?

GTP-3: There is no one-size-fits-all answer to this question, as the best course of action will vary depending on the specifics of the situation. However, some possible options include:

- 1. Attempting to fix the broken Dockerfile yourself
- 2. Reaching out to the repository owner to ask for help
- 3. Looking for another repository with a working Dockerfile that accomplishes the same goal

Repository owners are often too busy and fixing the file myself could be quite time consuming. Do you think there's any way one could automate the repair of broken Dockerfiles?

GPT-3: There is no guarantee that automated repair of broken Dockerfiles will be possible or effective, as the cause of the breakage may be highly specific to the repository in question. However, it may be worth exploring whether there are any existing tools or scripts that could be adapted for this purpose.

Spoilers: there weren't existing tools, but we made one!

A dialogue with OpenAI's GPT-3

# 6.1 Introduction

In the previous chapter, we introduced Dockerfiles (an artifact used by the Docker engine to produce images; from images one can run containers which are, essentially, lightweight virtual machines). Docker is one of the most widely used tools for virtualization. With  $\sim 79\%$  of IT companies using it (Portworx, 2017) and over 3.7

million unique installations of the VS Code Docker extension (Marketplace, 2020), Docker has made an enormous impact on developers' day-to-day work.

### 6.1.1 Motivation

Despite the utility of Dockerfiles and their prevalence in industry, we found that over 26% of our sample of Dockerfiles obtained "in the wild" (sourced from GitHub) failed to build successfully. This high a percentage is surprising, because it runs counter to one of the core tenets of Docker, namely, reproducibility. Furthermore, it is outside of the scope of recent efforts to statically analyze Dockerfiles to detect such failures. For example, Hadolint (Hadolint, 2019) can detect mistakes such as missing or incorrect flags—e.g., forgetting the use of -assume-yes/-y when invoking apt-get install in a Dockerfile. This flag is required because a Dockerfile build runs without interaction; therefore, forgetting this flag may cause the build to hang. Unfortunately, while Hadolint can detect such a mistake statically, many breakages occur due to a change in the external environment and not a change in the source Dockerfile. These observations add to the mounting evidence that external changes, such as dependencies changing, can often lead to broken build-related artifacts (Tufano et al., 2017; Hassan and Wang, 2018). As further evidence of this trend, our prototype tool, Shipwright, was used to guide 19 accepted pull requests on GitHub and, for each of these patches, the underlying issue was caused by a *change in the external* environment (see Section 6.2.1 for an example of one such change).

To recap, we find ourselves in a situation where over a quarter of the GitHub repositories with Dockerfiles that we analyzed had a broken Dockerfile. Furthermore, we find that current state-of-the-art (static) analysis for Dockerfiles is largely incapable of detecting and/or repairing such broken Dockerfiles. Given this situation, we pose the following motivating question:

#### Motivating Question

Can we devise a technique to provide *automated repair* for broken Dockerfiles?

## 6.1.2 Goals

To address the large amount of broken Dockerfiles that we found "in the wild," and to close the gap between tooling that *detects* issues and tooling that could be used to *fix* those same issues (with little/no human intervention), we propose the following goal:

### Goals

Aid developers in repairing broken Dockerfiles, with the hope of reducing the high percentage of broken Dockerfiles that we observed on GitHub.

## 6.1.3 Contributions

The work described in this chapter makes the following contributions: 1) *Technique*. We introduce a human-in-the-loop approach to fixing broken Dockerfiles; 2) *Tool*. We made available a tool implementing our technique; and 3) *Data*. We made available an extension of the **binnacle** dataset (introduced in Chapter 5) that includes build logs.

**Technique.** Unlike previous approaches that attempt to mine patterns automatically and directly from Dockerfiles, Shipwright follows a human-in-the-loop approach to build a *repair* database. We include human supervision to broaden the effectiveness of Shipwright: a fully automatic approach would limit the scope of repairs that could be detected. Shipwright is designed to act as a "co-pilot" that can side-step the limitations of a completely automatic approach with the goal of constructing a comprehensive database of repairs. To build such a database, Shipwright uses clustering, human supervision, and a search-based recommendation system we built to leverage vast community-knowledge bases, such as StackOverflow and Docker's community forum. In general, we require repairs to incorporate both some kind of pre-condition (a pattern) and a transformation function (a patch).

During clustering, one challenge that Shipwright needs to address is the heterogeneity of the data, which is a mixture of code and natural language. The mixing of code and natural language makes it non-trivial to design a featurization method for clustering. Therefore, to address this challenge, Shipwright uses a modified version of Google's BERT model (Reimers and Gurevych, 2019; Devlin et al., 2019) to *embed Dockerfile build logs*. By using a pre-trained transformer-based neural model, we are able to side-step tedious feature engineering, and benefit from the diverse corpora on which BERT-based models have been trained. Using this embedding, we perform clustering with HDBSCAN (Campello et al., 2013), in the vector space of embedded build logs, and use the results to cluster failing builds. By using the vectors generated through BERT, we leverage the "understanding" encoded in BERT's language model. To our surprise, we found that recent off-the-shelf language models work well in this domain. Using the generated clusters, we employ human supervision to intuit, for each cluster, a likely root cause of failure and, if possible, engineer one or more automated repairs to save in Shipwright's database. Later, Shipwright will use this human-generated repair database to attempt automatic repair of failing Dockerfiles.

**Tool.** The scripts to automate each of the steps of Shipwright (see Figure 6.3) are publicly available at https://github.com/STAR-RG/shipwright.

**Data.** We have identified a subset of Dockerfiles from the **binnacle** dataset (see Chapter 5) that are amenable to automated builds. (The dataset and filtering criteria are described in Section 6.3.) We have built these Dockerfiles *in-context* (an expensive operation that requires hundreds of hours of compute time), and captured detailed data from the results, including logs from the builds. These build logs represent a significant expansion of the data in the original **binnacle** dataset, and it is our hope that this extended data will accelerate research on diagnostic tools for Dockerfile analysis and repair. This expanded data is available at https://github.com/STAR-RG/shipwright.

### 6.1.4 Evaluation

We evaluated several aspects of Shipwright; a summary of our results is as follows: (i) Broken Dockerfiles are prevalent: in the data we analyzed, 26.3% of Dockerfiles failed to build. (ii) Even using optimistic criteria, existing static tools are capable of identifying the cause of a failure in only 20.6%-33.8% of the broken Dockerfiles. (iii) Shipwright is capable of clustering broken Dockerfiles and offering actionable solutions: for files that clustered, Shipwright provides automated repairs in 20.34% of the cases; for files that did not cluster, Shipwright is still able to provide automated repairs in 18.18% of cases. (iv) In a "time-travel" analysis, we found that Shipwright would have been able to provide actionable solutions to 98.04% of the Dockerfiles that we found initially to be broken and then subsequently found had been fixed (by others) in their respective repositories. Finally, we used the reports from Shipwright to submit 45 Pull Requests about still-broken Dockerfiles. Of these, 19 have been accepted. These results provide initial, yet strong, evidence that Shipwright is useful to help developers fix broken Dockerfiles.

# 6.2 Sources of Build Failures

This section describes some of the distinct sources of problems that can lead to build failures in Dockerfiles.

## 6.2.1 Breaking Changes in External Files

This kind of failure occurs when a Dockerfile's external dependencies (such as the file's base image or URLs embedded within the file) are changed; often these changes external to the Dockerfile will require a change in the Dockerfile itself. To illustrate this problem, consider the case where the developer used latest to indicate the version of the base image of her Dockerfile, as in FROM ubuntu:latest. These base images are downloaded from Docker Hub (docker, 2015), a distributed database that is part of the Docker ecosystem. The problem with using latest is that a change to the base image may require changes to the Dockerfile. Unfortunately, there is no clear way to incorporate those required changes automatically. For instance, the python-pip package is part of Python 2, and Python 2 is unavailable on Ubuntu images higher than 18.04. Consequently, a build on a Dockerfile with the command apt-get -y install python-pip will pass when the file is based on Ubuntu images

18.04 and lower, but it will fail on higher versions, including the latest LTS version of Ubuntu.

We used the Shipwright toolset to analyze and cluster hundreds of broken Dockerfiles, looking for common error patterns in their build logs and associated Dockerfiles. Using a human-in-the-loop process, we then extracted patterns and associated them with candidate repairs. For example, when running the command docker build on the Dockerfile FROM ubuntu:latest...RUN apt-get -y install python-pip..., Docker reports the message "*Unable to locate package python-pip*" on output. When that message is present in the logs, we found that the typical image in Dockerfiles is Ubuntu and the version is either undefined, latest, or 20.04. We also noticed that this error message appears not only with python-pip, but with other packages as well.

We expressed these patterns with regular expressions to be checked against the Dockerfile (the static data) and error logs (the dynamic data). For instance, we expressed the pattern for the problem above with the regex "FROM ubuntu( $\epsilon$  | :latest | :20.04)"  $\wedge$  "Unable to locate package (.\*)"  $\in$  log. Note that such an expression 1) defines a pattern over the Dockerfile, 2) defines a pattern over the output log, and 3) uses the groups "( $\epsilon$ ...)" and "(.\*)" to bind data to variables for later use.

Figure 6.1 shows two possible solutions to the problem above. The first solution is to fix the base image to the most recent version with which the command can still be executed. The following abstract operation characterizes the repair: replace \$0 with :18.04. The symbol \$0 refers to the regex group matching the Ubuntu image in the Dockerfile (i.e., " $(\epsilon...)$ ") that must be replaced with one specific Ubuntu version (e.g., :18.04). The second solution is to install Python 2 and its toolset. The following operation characterizes the repair: add ARG DEBIAN\_FRONTEND=. . . after "FROM ubuntu( $\epsilon$  | :latest | :20.04)". The symbol "..." is a placeholder for the text associated with the second solution from Figure 6.1.

Shipwright records the association between a given pattern (of build error) and possible solutions, such as the two repairs above. From this information, Shipwright is able to repair broken Dockerfiles whose build logs match some of the pre-recorded patterns. Table 6.1 describes this example under the row with "Id" 5. Note that the

```
# solution 1, use version 18.04
 1
 2
    FROM ubuntu:18.04
 3
    RUN apt-get -y install python-pip
 4
    ... #remaining code
 5
 6
    # solution 2, manually install the package
 7
    FROM ubuntu:latest
 8
    ARG DEBIAN_FRONTEND=noninteractive
    RUN apt-get -y install python2 curl software-prop... \
 9
10
        && add-apt-repository universe \
        && curl https://.../get-pip.py --output get-pip.py \
11
        && python2 get-pip.py
12
    ... #remaining code
13
```

Figure 6.1: Solving python-pip unavailable on ubuntu:latest.

repair operation consists of multiple possible solutions, hence the comma and dots, as we only show the first solution. In this case, Shipwright produces two versions of the Dockerfile and the developer should choose which one suits best their needs. We elaborate on Shipwright's workflow in the following sections.

It is worth noting that prior work has investigated the impact of breaking changes in package-management systems (e.g., in the Linux package manager, npm, maven, etc.) (Vouillon and Cosmo, 2013; McCamant and Ernst, 2004; Møller and Torp, 2019; Tucker et al., 2007; Mancinelli et al., 2006). Shipwright is not restricted to this kind of issue, and it is distinct from prior work on the application context and solution used. Section 6.6 elaborates on related work.

## 6.2.2 Inconsistent Version Dependency Within Project

This kind of failure occurs when there is an inconsistency between the versions of a Dockerfile and some of the files it references within the project. Figure 6.2 shows a concrete example that illustrates the problem. The Dockerfile requires an image for Ruby version 2.6.3, whereas the application code declares a dependency on a newer Ruby version (2.6.5) (Barcelona, 2020). The execution of the command RUN

```
1 FROM ruby:2.6.3
2 RUN apt-get update -qq && apt-get install -y \
3 ...
4 RUN gem install bundler:2.0.1
5 RUN bundle install # <-- Gemfile depends on ruby 2.6.5
6 ADD . /app</pre>
```

Figure 6.2: Inconsistent Ruby version dependencies.

bundle install triggers an error, producing the following message on output "Your Ruby version is 2.6.3, but your Gemfile specified 2.6.5". In this case, the solution was to replace line 1 of this Dockerfile with FROM ruby:2.6.5. The pattern and corresponding repair explained above are listed in Table 6.1 under row "Id" 6. A similar repair is "Id" 1, which is also related to Ruby.

## 6.2.3 Missing Commands in the Dockerfile

This failure occurs when a given Dockerfile uses a command that is unavailable on a given image. The solution in that case is to install the command using the proper syntax, because it depends on the version of the image. Table 6.1 lists one example of this error pattern and the respective repair under row "Id" 8.

## 6.2.4 Project-Specific Failures and Suggestions

We observed that many broken Dockerfiles require repairs that are project-specific and cannot be generalized. In those cases, Shipwright is unable to produce a repair to the broken file. Instead, in those cases, Shipwright provides *suggestions*. For instance, consider the case where a Dockerfile includes a command to deploy a Node.js server, such as RUN npm run build. The execution of that command fails because there is an error in the Node.js project. There is nothing to fix within the Dockerfile. The developer needs to analyze what is wrong in her Node.js project and fix it. In that case, Shipwright reports a suggestion, such as "NPM build error..". As another example, consider a case in which a command refers to a broken link, such



Figure 6.3: Shipwright's 3-step workflow. In step (1), a database of Dockerfiles and GitHub metadata is used to perform *in-context* builds. The results of these builds are stored in a local database along with various forms of metadata. In step (2), Shipwright uses BERT and HDBSCAN to cluster build data (Reimers and Gurevych, 2019; Devlin et al., 2019; Campello et al., 2013). The clusters are then fed to Shipwright's search-based repair-generation and suggestion-generation process. During this step, Shipwright, with the assistance of a human, builds a database of repairs and suggestions. Finally, in step (3), online usage begins: new files are fed to Shipwright and, if matching database entries are found, Shipwright provides relevant repairs or suggestions.

as RUN wget <url>. Shipwright cannot guess how to fix the broken link. Table 6.2 shows a sample of suggestions provided by Shipwright. In the future, it would be interesting to examine how one might combine Shipwright with other tool-specific and language-specific repair techniques. Tools that have a *combined understanding* of DevOps artifacts and the programs these artifacts support represent an intriguing area for future work.

# 6.3 Dataset

We use an expanded version of the **binnacle** dataset introduced in Chapter 5 as the source of Dockerfiles to analyze. The **binnacle** dataset consists of *all* Dockerfiles from GitHub repositories *with ten or more stars*. These Dockerfiles represent a broad and largely unfiltered picture of the state of Dockerfiles one might find in popular GitHub repositories. Although the original dataset was created in 2019, the **binnacle** toolchain allows us to capture recent data using the same methodology; thus, we populated our dataset with more recent data (June 2020) extracted using the same tools. Unfortunately, directly using the Dockerfiles in this dataset is challenging for two reasons: (i) many Dockerfiles in the dataset come from the same repository and, in

such cases, the *purpose* of the Dockerfiles is obscured, making efforts—like automated builds—more difficult; (ii) many Dockerfiles are nested deep within repositories (especially when repositories contain many independent projects or services). In either case, automated builds are challenging because the *intent* behind the Dockerfile is difficult to infer. In case (i), it is difficult to infer which of the (many) Dockerfiles should be built. In case (ii), it is difficult to infer an appropriate *context directory*, which is a pre-requisite to building a Docker image.

**Dataset Filtering.** To address these issues, we filtered the files from the binnacle dataset using two criteria: (a) we only considered repositories with a single Dockerfile, and (b) we required that the given Dockerfile resides within the repository's root directory. For such a repository, it is not unreasonable to assume two things: (i) the Dockerfile is intended to produce an artifact corresponding to the given repository (because it is the only Dockerfile in that repository), and (ii) the Dockerfile likely uses the repository's root directory as its build context: the Dockerfile resides in the root directory, and the docker build command assumes, by default, that the target Dockerfile resides within the given context directory. Performing this filtering yielded 32,466 repositories and corresponding Dockerfiles that may be amenable to automated builds. It is this subset of the original dataset (a refreshed version of binnacle's dataset) that we used in our studies.

**Building Dockerfiles at Scale.** Shipwright performed *in-context* builds on 20,526 of the 32,466 Dockerfiles in our *filtered* dataset (recall: we filter to find Dockerfiles from the original dataset that are likely amenable to *automated* builds). Although we would have liked to use all 32,466 Dockerfiles, we encountered some problems performing in-context builds on that many files in a reasonable time frame, which forced us to use a smaller set of 20,526 files. We tried various approaches to scaling these in-context builds, but distributing this kind of process would require Docker installations across a wide variety of machines—in practice, this requirement was difficult to satisfy: none of the distributed-computing resources we had access to would allow us to control a *Docker daemon* on the distributed machines (while, for many distributed platforms, running *containers* is easy). This requirement, while understandable from a security



Figure 6.4: Clustering Example. Starting with several broken Dockerfiles, Shipwright clusters by extracting standard error logs, applying aggressive token splitting (we split on snake and camel case, as well as several operators that may occur in code snippets) and string normalization (we lowercase the input, clear large blocks of repetitive characters, like extraneous white space, and we strip certain special characters/unicode), and passing the resulting sequences to BERT. BERT takes the input sequences and produces vectors (shown as points in a high-dimensional space). Shipwright uses that mapping, from failing build logs to points in a vector space, along with a clustering algorithm (HDBSCAN), to produce its clusters, shown in the bottom right of the figure. The final clustering process has the advantage of being *semantics aware*: BERT understands some of the nuances of the English language, and our clustering benefits from this capability.

perspective, forced us to run builds in a non-distributed way (on a single large server). Given this constraint, some builds either *time out* (we set a limit of 30 minutes) or, due to contention from running multiple builds on this single server and daemon instance, some builds fail to complete due to errors internal to the Docker daemon; builds that fail in this manner are marked as *undetermined*. Instead of revisiting undetermined builds, we spent our resources on building a larger portion of our filtered dataset. Through this process, we captured the results from 20,526 Dockerfile builds, and saved these results to a database for further analysis. Section 6.4.1 describes this offline data processing with the Shipwright toolset in further detail.

# 6.4 Technique

Figure 6.3 shows the workflow of Shipwright as a pipeline of three steps, organized according to their respective goals.

The goal of the first step is to analyze a corpus of broken Dockerfiles—mined from

GitHub—and to perform *in-context* builds so that logs can be acquired (Section 6.4.1). The goal of the second step is to cluster broken Dockerfiles and find repairs (i.e., transformation functions on Dockerfiles) (Section 6.4.2). Given a cluster, Shipwright automatically elaborates search queries from log files of representative Dockerfiles within the cluster. A human then supervises the creation of repairs and suggestions by (i) looking for error patterns as manifested in existing QA forums resulting from the search query, and (ii) creating plausible repairs (or, if no automated repairs are possible, creating suggestions) which are saved in a database for later, online, use. Finally, the third (interactive) stage of the pipeline looks for actual repairs for a broken Dockerfile (Section 6.4.3). This component takes as input the output produced in the previous (offline) stages and a broken Dockerfile, and produces either (i) an automated repair, (ii) a suggestion (in cases where repair cannot be automated), or (iii) an indication that no existing repairs or suggestions apply. The following sections describe each step in detail.

## 6.4.1 Data Collection

This component of Shipwright builds and analyzes Dockerfiles from a pre-existing corpus. We utilize the 32,466 Dockerfiles described in Section 6.3 as our input corpus. Those files were filtered to be amenable to automated builds. For each file in this corpus, Shipwright does the following: 1) *Clones* the originating repository for the given Dockerfile into a unique /tmp/<repo-id> directory; 2) *Runs* docker build -f <Dockerfile> /tmp/<repo-id>, which builds a <Dockerfile> from our dataset using the root directory of the cloned repository as the build context. Building *in context* is crucial because the build may need to access files from the originating repository to complete successfullyÅlthough we are interested in build failures; we want to avoid trivial failures; 3) *Discards* builds that still have trivial failures; and 4) *Saves*, for each failing build, execution logs (standard output and standard error streams), the AST for the given Dockerfile, and various metadata (e.g., repository information, image history, and a log of the git clone procedure). An example would be a broken build caused by an execution failure in a directive, such as COPY or ADD. Although

builds are performed *in context*, it is still possible that the Dockerfile is intended to be built as part of a more complex workflow with a context directory that is different from the repository root directory. Unfortunately, this information may exist in some third-party script, or may be user-supplied.

Figure 6.3 illustrates this workflow in the box named "Step 1: Data Collection." We note that data collection is quite costly: we ran a 32-core CentOS workstation for several weeks and, during that time, managed to build about two thirds (20,526) of the 32,466 files in our dataset. (See Section 6.3 for a discussion of the difficulties of building many Dockerfiles at scale.) Although builds can be parallelized, there is only one Docker daemon per installation of Docker—this situation creates a limit to the practical concurrency that can be achieved, along with the network bandwidth to the workstation used for analysis. We attempted to perform builds on a high-throughput cluster but, unfortunately, the strict security requirements of such clusters prevent deploying a workflow involving Docker images builds, which effectively execute untrusted code.

We also explored using a high-throughput cluster to perform Dockerfile builds, but it is exceedingly difficult to deploy such a workflow as most high-throughput compute clusters do not allow for running docker-related commands (for security reasons). Instead, we attempted a docker-in-docker (DinD) style workflow (in which we could deploy concurrent docker builds within docker containers) but, again, the strict requirements of high-throughput platforms did not allow us to deploy such a workflow. In the near future, efforts to enable "non-root" Docker installations may make such workflows possible on high-throughput platforms.

### 6.4.2 Repair & Suggestion Extraction

This step of Shipwright works as follows. First, it uses HDBSCAN, applied to embeddings,<sup>1</sup> to partition the Dockerfile data produced in the previous step. Second,

<sup>&</sup>lt;sup>1</sup>Embeddings refer to high-dimensional vectors of numbers that are used as a proxy for nonnumeric artifacts (such as text or code). Embeddings are often of use, because many operations can be performed in the resulting vector space, and later mapped back to the originating artifacts.

it uses those clusters to assist a human with the task of searching for solutions and building a database of repairs. We now elaborate on each of these steps.

#### Clustering

Shipwright attempts to cluster failing Dockerfile builds using embeddings and HDBSCAN (a hierarchical variant of DBSCAN, a classic clustering algorithm (Ester et al., 1996b)). The difficulty of clustering in this domain is two-fold: (i) the data to cluster is heterogeneous, and is often a mix of code and natural language (i.e., the build logs, which will often contain a description of the failure in English and a reproduction of the Bash or Dockerfile snippet that leads to the error), and (ii) although we would like to cluster on the *cause* of build failures, we do not have a way to definitively extract the cause of a given build failure; therefore, we must use data that is, at best, a proxy or symptomatic of the root cause of failure. In particular, we use a tokenized version of the build logs for the failing build (which may include a variety of things: debug output, warnings, and errors—some of which may include code snippets) as input to BERT to produce embeddings.

Despite these challenges, Shipwright is able to perform clustering by leveraging a key insight: recent off-the-shelf language models, such as BERT, GPT-2 and, recently, GPT-3, have reached impressive levels of sophistication (Devlin et al., 2019; Radford et al., 2019; Brown et al., 2020). Given the inputs these models are trained on (roughly, massive crawls of the internet), it is highly likely that such models have seen websites like StackOverflow, which mix both natural language and code. Therefore, to address challenge (i) (the heterogeneous mix of code and natural language), Shipwright leverages a sufficiently sophisticated off-the-shelf language model (BERT), to obtain *embeddings*. In particular, Shipwright uses a variant of BERT suited to the task of sentence embedding, in which similar sentences should end up "close" in the embedding space. Shipwright applies this BERT variant to the last few lines of the error logs to produce a vector representation of each broken Dockerfile. These vectors are then fed to HDBSCAN, which produces clusters. Figure 6.4 illustrates the Shipwright clustering process.

#### Searching for Repairs or Suggestions

This component of Shipwright takes a set of clusters as input, and produces a *list* of pairs as output. The first element of the pair is a signature that identifies the issue (in the Dockerfile and its logs) whereas the second element of the pair is either (i) a repair, consisting of a pure transformation function that takes a Dockerfile as input and produces another file as output, or (ii) a suggestion (about what needs to be repaired and how) for the cases where human knowledge is necessary to prepare the repair. We elaborate on each of these two cases in the following and Tables 6.1 and 6.2 provide examples of such pairs.

Case 1 (Searching for Repairs): Shipwright uses a search-based recommendation system to assist a human in locating repairs of broken Dockerfiles. Shipwright proceeds as follows: it selects a cluster and a representative Dockerfile from that cluster; it extracts keywords from the logs of that file; it builds a search string from those keywords; it submits the corresponding query to a search engine; it filters the outputs from related community forums; and it reports a list of the top-5 URLs as output for a human to inspect. Human inspection consists of reading proposed solutions on discussion forums, and then applying a given solution to the representative Dockerfile from the cluster. If that solution is plausible, i.e., if it allows the Dockerfile to build an image successfully, the next step is to check if the error-pattern/repair-function pair is applicable to other Dockerfiles in the cluster. While doing so, the human inspector looks for opportunities to generalize the pattern and repair function to avoid overfitting a solution to a particular case. For instance, in the example given in Section 6.2.1, the initial solution was too narrow, focusing on fixes of files containing the exact message "Unable to locate package python-pip" in the output log. However, we observed similar error messages, referring to different packages. In this case, the solution was to replace "python-pip" in that string with a symbolic name for a package. To sum up, Shipwright leverages community knowledge bases (e.g., StackOverflow and Docker's community forums) to find solutions to known issues, such as those presented in Section 6.2.

Shipwright supports a total of 13 repair patterns. Table 6.1 shows the pairs of (1)

Id	Pattern / Repair	Source
1	"ERROR: Error installing bundler $\in \log \land$ "bundler requires Ruby version $\geq ([0-9]+.[0-9]+.[0-9]+)$ " $\in \log$ replace FROM ruby:(.*) with FROM ruby:\$0	oshivwanshi (2019)
2	"Rpmdb checksum is invalid: $dCDPT$ " $\in \log$ add RUN yum install -y yum-plugin-ovl after FROM(.*)	r1williams $(2015)$
3	"E: Some index files failed to download. They have been ignored, or old ones used instead" $\in \underline{\log}$ replace base image with latest release from hub.docker.com	Schulze (2018)
4	$\label{eq:energy} \begin{array}{c} ``E: \ Package \ `libpng12 \ dev' \ has \ no \ installation \ candidate'' \in \underline{\log} \\ \\ \underline{\mathrm{replace}} \ \mathtt{libpng-l2dev} \ \underline{\mathrm{with}} \ \mathtt{libpng-dev} \end{array}$	jahanzaib basharat (2018)
5	$\label{eq:FROM_ubuntu} \begin{split} & \text{``FROM_ubuntu}(\epsilon : \text{latest} :20.04)`' \land \\ & \text{``Unable to locate package } (.*)`' \in \underline{\log} \in \underline{\text{Dockerfile}} \\ & \underline{\text{replace}} \$ \emptyset \ \underline{\text{with}} : 18.04, \ldots \end{split}$	PacificNW_Lover (2020)
6	"but your Gemfile specified $([0-9\backslash \backslash.]+)$ " $\in \underline{\log}$ $\land$ "FROM ruby $(.*)$ " $\in \underline{Dockerfile}$ $\underline{replace}$ FROM ruby: $(.*)$ with FROM ruby:\$0	Tan (2016)
7	"invalid byte sequence in US-ASCII" $\in \underline{\log}$ $\land$ "FROM ruby(.*)" $\in \underline{Dockerfile}$ $\underline{add} ENV LANG C.UTF-8 \underline{after} FROM(.*)$	ubergesundheit (2015)
8	<pre>"sh: (.*): not found" ∈ log if "FROM alpine(.*)" ∈ Dockerfile then add RUN apk add -no-cache \$0. else add RUN apt-get -y update &amp;&amp; apt-get -y install \$0</pre>	rmNyro (2017)
9	"ERROR: unsatisfiable constraints: bzr (missing):" $\in \underline{\log}$ $\wedge$ "FROM alpine(.*)" $\in \underline{Dockerfile}$ $\underline{remove} \ bzr \underline{in}$ "apk add" command	yelizariev (2020)
10	"conda: not found" ∈ $log$ $\land$ "RUN curl (https://repo.continuum.(.*))" ∈ $lockerfile$ <u>add</u> -L <u>before</u> \$0	Buddhi (2019)

Id	Pattern / Suggestion
1	"mix" $\in \underline{\log} \wedge$ "Code.LoadError" $\in \underline{\log}$ Problem running mix on the Elixir project
2	" $tsc$ " $\in \underline{\log} \land$ ": error $TS$ " $\in \underline{\log}$ Error during TypeScript compilation with tsc. Please check your .ts files.
3	"wget: server returned error" $\in \underline{\log} \lor$ "wget: unable to resolve host address" $\in \underline{\log}$ wget error, you have a broken URL. Please check the log.
4	"npm $ERR!$ " $\in \underline{\log}$ NPM Error, check your files, in particular, "npm install" commands.
5	" $curl:([0-9]+)$ .*" $\in \underline{\log}$ curl error, you have a broken URL. Please check the log.

build-error signatures—referred to as a pattern—and (2) a corresponding repair for 10 of them. Column "Id" shows the id of a pair. Column "Pattern" shows the error pattern, which is a regular expression that matches a string in the error logs (the dynamic part) and/or a string in the Dockerfile (the static part). Column "Repair" shows a function, in natural language, describing how to transform and fix a broken Dockerfile. We use the keywords <u>add</u>, <u>remove</u>, and <u>replace</u> to describe operations that need to be performed on the Dockerfile. We informally described the semantics of these operations with examples in Section 6.2.1. Although there is no fundamental reason preventing us from creating these transformations automatically, we wrote the code implementing these transformation functions because we found empirically that creating these functions was not a time-consuming error-prone task. Finally, column "Src." shows a reference for the solution on the web.

Case 2 (Searching for Suggestions): There are cases where Shipwright cannot produce a repair. For example, a Dockerfile whose build fails because of a compilation error or broken URL requires a human to fix the underlying error. For those cases, we report a suggestion, i.e., generic advice on what needs to be done. Table 6.2 shows a small sample from the total of 50 suggestion patterns that

Shipwright supports (in addition to 13 repair patterns). Column "Id" shows the id of the suggestions, column "Pattern" shows the signature, and column "Suggestions" shows the suggestion message.

## 6.4.3 Repair and Suggestion Generation

Shipwright can be used to repair Dockerfiles or provide suggestions using the database generated in step 2 (Section 6.4.2). Given a Dockerfile, Shipwright iteratively examines the repairs and suggestions and, given a match, it either (i) produces a patched file, by applying a repair, or (ii) provides a suggestion message to the user. If neither a repair nor a suggestion with a matching pre-condition exists within the database, Shipwright is still able to use its search-based process to guide a human in producing fixes and suggestions, as we did in step 2 (Section 6.4.2). This search-based process provides a user with a small set of (filtered) links to resources likely to help in fixing the given input file. In summary, Shipwright, during step 3, produces either: (i) a Dockerfile repair, (ii) a suggestion on how to fix the broken build, or (iii) a curated set of results from a search-based process that may provide solutions to the underlying build issue.

# 6.5 Experiments

The goal of Shipwright is to help developers fix broken Dockerfiles. It does that through a combination of (i) clustering of broken Dockerfiles (by likely root cause), and (ii) a search-based method to find repairs (and, if no automated repairs are feasible, suggestions). To gain insights into the landscape of broken Dockerfiles used in GitHub projects and to understand Shipwright's efficacy, we pose the following research questions.

## Research Questions 1 (Dockerfile Build Failures)

### Research Question # 1

How prevalent are Dockerfile build failures in projects that use Docker on GitHub? Can existing (static) tools identify the failure-inducing issues within these broken files?

### Rationale

The purpose of this question is to evaluate the potential impact of Shipwright. If build failures are rare, then impact is limited. Furthermore, reproducibility is a core tenet of Docker—it would be surprising to find many broken Dockerfiles. We also assess the ability of existing (static) tools to identify issues that may lead to failing Dockerfile builds.

#### Metrics

We used the following metrics to answer RQ1: 1) the fraction of Dockerfiles in our dataset with builds that fail; 2) the relationship between failures and project popularity; and 3) the success rate of existing (static) tools in predicting Dockerfile build failures. The first metric evaluates the fraction of Dockerfiles that we mined from GitHub that fail to build because the Dockerfile is broken (for non-trivial and non-toolchain-related reasons). The second metric examines the relationship between the number of GitHub stars a given repository has (a common proxy for popularity on GitHub) and whether that repository contains a broken Dockerfile. This measurement helps us ascertain whether popular repositories suffer from broken Dockerfiles at the same rate as less popular repositories. Recall that we do not have any Dockerfiles from repositories with less than 10 GitHub stars (Section 6.3). Finally, the third metric ascertains the ability of pre-existing tools, namely, Hadolint (Hadolint, 2019) and **binnacle**'s rule checker (Chapter 5), to find issues within broken Dockerfiles.

## Research Question 2 (Clustering with Embeddings)

### Research Question # 2

Can we use off-the-shelf language models, like BERT, to easily cluster broken Dockerfiles?

#### Rationale

Given the number of observed failures, it is reasonable to ask whether many failures are *unique*. If many failures are similar, one might hope that generalized repairs exist. Furthermore, if failing Dockerfile builds can be clustered, those clusters may be used to bootstrap finding repairs. Finally, if we can leverage the level of understanding available in large off-the-shelf language models (like BERT), then we can design robust clustering routines with little specialized engineering effort, and avoid techniques based on manually designed heuristics.

#### Metrics

To answer RQ2, we examine the percentage of clusters (generated using Shipwright), where all elements share a single (likely) root cause. This metric provides insight into Shipwright's ability to cluster broken Dockerfiles and the usefulness of those clusters—good clustering allows for finding multiple exemplars for a single failure which, in turn, makes the task of generating automated repairs simpler.

## Research Question 3 (Repair)

## Research Question # 3

How effective is Shipwright in producing repairs? (i) To what extent do repairs cover the failures from our dataset? (ii) For failures that can be clustered, is it possible to generalize repairs? (iii) What can be done for failures from non-clustered files?

#### Rationale

The purpose of this question is to evaluate Shipwright's effectiveness on our dataset. If proposed solutions are unable to cover a variety of Dockerfiles, then Shipwright's usefulness is questionable.

#### Metrics

1) We measured the fraction of broken Dockerfiles (from our dataset) for which Shipwright produces a repair. 2) For the *set of clusters* that Shipwright produces, we measured the extent to which repairs generalize. For that, we measure "coverage" (i) in the cluster that originated that pattern, and (ii) across different clusters. Coverage refers to the portion of elements within a cluster that match the same pre-condition for a repair. 3) For broken Dockerfiles that *did not cluster*, we measured how often Shipwright provides a repair. In all cases, we also evaluated Shipwright's ability to provide *suggestions* if repairs were not feasible. Collectively, these metrics measure how effective Shipwright is in proposing solutions to the broken files in our dataset.

## Research Question 4 (Impacts)

#### Research Question # 4

How effective is Shipwright in reducing the number of broken Dockerfiles in public repositories?

#### Rationale

Although RQ3 seeks to evaluate Shipwright's ability to fix broken Dockerfiles, there still remains a question of Shipwright's usefulness in practice. RQ4 seeks an understanding of Shipwright's effectiveness to meet our overarching goal: fixing broken Dockerfiles in public repositories.



Figure 6.5: Proportion of different kinds of solutions within each cluster (excluding singleton clusters).

#### Metrics

To answer RQ4, we used two metrics: 1) What proportion of Dockerfiles that appear in our dataset as broken Dockerfiles, but have since been fixed, *would also have been fixed, had we applied* Shipwright? 2) How often can we use Shipwright to produce pull-requests that are accepted by external reviewers? The first metric refers to a kind of "time-travel" analysis because, using updates that took place during the period in which we built Shipwright, we can attempt to measure how successful we *would have been* had Shipwright existed at an earlier date, and had we applied it. Nevertheless, this metric is still a "simulated" one. Therefore, the second metric quantifies Shipwright's "real-world" applicability by actually using it to produce repairs and submitting them for (external) review.

## 6.5.1 Answering RQ1 (Dockerfile Build Failures)

To answer Research Question 1, we used Shipwright to build a random sample of Dockerfiles from our (filtered) dataset. In total, we tried to build 20,526 Dockerfiles and found 5,405 broken Dockerfiles. This gives us an estimated 26.3% "breakage rate" for Dockerfiles in our overall dataset. The large amount of broken Dockerfiles on GitHub runs counter to one of the core reasons for using Docker: *reproducibility*. Aside from broken Dockerfiles, we encountered 393 Dockerfiles with builds that time



Figure 6.6: Breakdown of the 20,526 files we attempted to build

out (we use a threshold of 30 minutes) and 3,514 Dockerfiles with undetermined results (which arise due to the pressure that multiple concurrent builds place on the Docker daemon). Neither timeouts nor builds with undetermined results are counted as broken Dockerfiles. Instead, we count these results as successful builds to give a conservative estimate (and lower bound) of the "breakage rate" for Dockerfiles in our dataset. Figure 6.6 provides a visual overview of these categories.

To put these results in context, we also examined the distribution of stars for the 5,405 repositories in our dataset. For these repositories, we find that: (i) a third have 18 stars or fewer, (ii) a third have greater than 18 stars, but fewer than 51 stars, and (iii) a third have 51 or more stars. This distribution was surprising, especially because some repositories with broken Dockerfiles had many thousands of stars. We spot-checked some of these cases and found that, indeed, even quite popular repositories can have broken Dockerfiles. For example, the MEAN stack project (Linnovate, 2020) has over 12K stars, yet it contains a Dockerfile that fails to build.

Finally, we also tested the capabilities of two existing (static) tools: binnacle (introduced in Chapter 5) and Hadolint (Hadolint, 2019). For both tools, we sought

an estimate of the number of broken Dockerfiles for which each tool *identifies a possible build-breaking issue*. Because we found it impractical to manually examine the tools' outputs on each of the 5,405 broken files, we instead used a (generous) estimate based on how often each tool reports a rule violation for an issue that *might cause a build to break*. For example, Hadolint can identify when the version of an image used for a base in a Dockerfile is un-pinned; thus, if Hadolint reports a rule violation in this category, on any file, we count it as Hadolint identifying a *possible build-breaking issue* (and mark the file as "solved" by Hadolint). In total, Hadolint identifies such issues in only 33.8% of files, and **binnacle** identifies such issues in only 20.6% of files.

Summary of RQ1: The presence of broken Dockerfiles on GitHub is common. Furthermore, even highly starred repositories sometimes contain broken Dockerfiles. Finally, existing static tools only identify plausible build-breaking issues in 20.6–33.8% of cases (and, even when issues are identified, such tools do not provide repairs).

## 6.5.2 Answering RQ2 (Clustering with Embeddings)

To generate the clusters we use throughout our evaluation, we first performed a grid search against Shipwright's clustering algorithm. During this search, we focused on exploring the space of hyperparameters used in HDBSCAN—the embeddings, although generated by a neural model, are not "tunable" without investing in re-training the model, which is outside the scope of Shipwright. We searched approximately 200 configurations and found, on average, HDBSCAN was able to cluster 34% (1,836) of the 5,405 broken Dockerfiles identified by Shipwright. In the clustering that we used, consisting of 144 clusters containing 1,814 files, we were able to confirm that 36.5% of the clusters consisted of Dockerfiles that all had the same root cause for their failures.

Summary of RQ2: Shipwright's approach to clustering Dockerfiles can, on average, cluster 34% of our dataset, and, for over a third of the clusters generated, we can confirm that a *single* issue covers all failing Dockerfiles within a cluster.

The answer to RQ2 bodes well for using clusters to bootstrap finding automated repairs. However, we note that the clustered files only make up a portion of broken files: therefore, to assess generalizability, RQ3 examines Shipwright's ability to use repairs learned from our clustered files and apply them to non-clustered files.

## 6.5.3 Answering RQ3 (Repair)

This question evaluates Shipwright's effectiveness on our dataset of broken Dockerfiles (Section 6.3).

Research Question 3.1 evaluates how much of the set of broken Dockerfiles can be addressed with the repairs that Shipwright generates. Figure 6.5 shows the effects of the repairs (and suggestions) that we found across the 144 clusters produced by Shipwright. Each vertical bar denotes one cluster. These bars are divided into three segments. The size of the segment at the bottom of the bar (in yellow) represents the percentage of failures in a given cluster for which Shipwright provided an automated repair; the size of the segment in the middle of the bar (in blue) represents the percentage of failures for which Shipwright provided suggestions (which are generated in cases where no repairs apply); and the size of the segment at the top of the bar (in gray) represents the percentage of failures for which Shipwright could *not* find a solution.

Summary of RQ3.1: The 13 repairs created with Shipwright offered solutions to 20.34% of the 1,814 broken and clustered Dockerfiles. In cases where no repairs were applicable, Shipwright's 50 suggestions applied to an additional 69.63% of the broken and clustered Dockerfiles.

Ы	# Clusters	Coverage (%)		
Iu		Parent	Average	
1	4	100.00	61.00	
2	3	40.00	25.67	
3	8	100.00	54.00	
4	3	60.00	49.34	
5	2	88.00	88.00	
6	2	100.00	90.50	
7	1	82.14	82.14	
8	6	100.00	88.67	
9	1	100.00	100.00	
10	2	100.00	95.50	
11	3	62.00	50.00	
12	2	80.00	42.00	
13	3	80.00	60.00	

Table 6.3: Repair Coverage

Research Question 3.2 evaluates the ability of the repairs to generalize to a large number of cases.

Table 6.3 shows the relative amount of broken Dockerfiles that each one of our 13 repair patterns covered.

Column "Id" refers to the id of the repair (most of which listed in Table 6.1), and column "#Clusters" shows the number of clusters where the corresponding repair could fix at least one of the broken Dockerfiles in it. Error patterns are extracted from a given cluster, which we refer to as "parent". Column "(Coverage) Parent" then shows the fraction of broken Dockerfiles within the parent cluster that were corrected using the respective repair. Column "(Coverage) Avg." shows the average fraction of repaired files across the different clusters affected by a repair pattern.

Summary of RQ3.2: The 13 repair patterns produced with Shipwright generalized well within the parent cluster (avg. 84.01%) and across affected clusters (avg. 68.22%). Recall that a total of 3,586 of the 5,400 broken Dockerfiles (66.4%) were *not* clustered. For non-clustered files, Shipwright produced repairs to 18.18% of them. Overall, Shipwright produced an actionable solution to the developer in 64.81% of the files that were not associated with any cluster (18.18% from repairs and an additional 46.63% from suggestions). Note that Shipwright used the patterns produced by analyzing clustered files. That was possible because the clustering step is conservative and clusters were based on embeddings of largely syntactic information (logs). For example, we observed that a file failing on the statement apk add A && apk add B && ... && apk add bzr was not clustered with other files failing on apk add bzr—but, upon further examination, we found that this file failed to cluster due to its use of a conjunction of successive apk add commands instead of the (more common) use of the multi-argument apk add A B ... bzr variant. In practice, although conservative, the generated clusters were suitable for creating useful and generalizable repairs.

Finally, even when no repairs or suggestions apply, Shipwright can still provide a list of URLs pointing to resources that may provide a developer with a fix for their broken file.

Summary of RQ3.3: Even in non-clustered broken Dockerfiles, Shipwright was able to produce automated repairs in 18.18% of the files. Furthermore, when no repairs applied, Shipwright was able to provide suggestions in 46.63% of the files.

## 6.5.4 Answering RQ3 (Impact)

This section reports on two experiments we conducted to assess the practical usefulness of Shipwright. The first experiment (Section 6.5.4) measures the fraction of initiallybroken but later-fixed Dockerfiles that could have been repaired with Shipwright. The second experiment (Section 6.5.4) measures the acceptance ratio of Pull Requests (PRs) for Dockerfiles found to be still broken in their repositories.

#### **Repair Confirmation**

This experiment evaluates Shipwright on real patches created by GitHub developers. The metric we used was the fraction of the patches created by developers that matched the repairs or suggestions of Shipwright. To run this experiment, we searched for fixed Dockerfiles on GitHub. We used the same procedure as reported in Section 6.4.1, but we re-cloned the repositories on Aug. 14, 2020 (8/14/20). Because we know that the Dockerfile build on the first version of the project failed, we only needed to perform Dockerfile builds for the 8/14/20 versions of projects. To avoid unnecessary builds, we looked for Dockerfiles that were changed in the repository, and found that 161(=8.87%) of the original 1,814 broken Dockerfiles were changed in their repositories from the day they were retrieved up to 8/14/20. We ran the command docker build in-context on those 161 files, and discarded the cases where the build was still unsuccessful. In the end, we obtained a set of  $102\langle x, y \rangle$  pairs to analyze, with x denoting a broken Dockerfile from our dataset and y denoting its corresponding patch. The method we used to measure effectiveness of Shipwright was to run Shipwright on x and compare the generated repair or suggestion, if found, with y.

Of the 102 cases of initially-broken then-fixed Dockerfiles, Shipwright produced an identical repair in 23 of the cases. In 77 cases, Shipwright provided suggestions that matched the patch used by the developer. Although we found that the ratio of suggestions to fixes was higher compared to results of RQ3.1, Shipwright covered most of the cases we analyzed (a total of 98.04% of the cases). Overall, we believe that this result is encouraging because it provides a strong (and relatively unbiased) indication that the repairs that Shipwright produces are (i) correct (they matched the fixes of developers) and (ii) useful (almost all cases were covered).

#### Pull Requests (PRs)

This experiment evaluates Shipwright on Pull Requests (PRs) issued to GitHub projects with still-broken Dockerfiles. The goal is to assess the feedback from developers to these PRs, which is a proxy for their interest in Shipwright's results. For each of the 13 repair patterns, we randomly sampled 5 Dockerfiles (from our dataset) that remained broken until the date we ran this experiment. Then, we manually prepared a PR that explained the problem (including a link to a similar case) and proposed a repair, as created by Shipwright. To avoid violating double-blind rules, we created and used a GitHub account under the fictitious name "Joseph Pett" to submit the PRs. Our artifact (https://github.com/STAR-RG/shipwright) includes an up-to-date tracker of the submitted, accepted, and rejected PRs.

Of the 45 PRs that we submitted, 19 were accepted by developers (=42.2%); 4 PRs were rejected; and 22 PRs have not yet been reviewed by developers. The number of submitted PRs was lower than 65 (=13\*5) because we could not find five Dockerfiles still broken for some of the patterns.

Three of the four rejected PRs were related to the same organization and the same problem, characterized by pattern #7 (Table 6.1). The developer pointed out that using a new version of the Docker Ruby image solved the encoding problem, and he preferred to update the Ruby version. With that feedback, we revised repair #7 to include a second solution, which is to update the Ruby version to 2.5.8. We have confirmed that this repair also works for the Dockerfiles repaired by the original solution. The new version of the Ruby image was committed on June 2020 (mtsmfm, 2020), while this issue has been reported since June 2015 (ubergesundheit, 2015). This GitHub issue was the URL recommended by Shipwright to assist the human to produce a repair.

**Summary of RQ4**: These results provide initial, yet strong, evidence that Shipwright is a useful aid to help developers fix broken Dockerfiles.

# 6.6 Related Work

## 6.6.1 Empirical studies on Docker (and DevOps)

A growing number of studies have been carried out on Dockerfiles, as well as on the broader topic of DevOps (Rahman et al., 2019) (also known as *infrastructure* 

URL	Repair ID
https://github.com/AjuntamentdeBarcelona/decidim-barcelona/pull/321	1
https://github.com/realpython/flask-image-search/pull/2	3
https://github.com/LLNL/merlin/pull/254	3
https://github.com/fisadev/zombsole/pull/11	4
https://github.com/xlight/docker-php7-swoole/pull/2	4
https://github.com/castlamp/zenbership/pull/226	4
https://github.com/edwin-zvs/email-providers/pull/9	5
https://github.com/ex0dus-0x/doxbox/pull/12	5
https://github.com/zhihu/kids/pull/58	5
https://github.com/cxmcc/webinspect/pull/1	5
https://github.com/thegroovebox/groovebox.org/pull/10	8
https://github.com/quasoft/backgammonjs/pull/26	8
https://github.com/gitevents/core/pull/216	8
https://github.com/htilly/zenmusic/pull/56	8
https://github.com/freedomvote/freedomvote/pull/332	8
https://github.com/enomotokenji/pytorch-Neural-Style-Transfer/pull/3	10
https://github.com/yesodweb/yesodweb.com-content/pull/255	11
https://github.com/anurag/fastai-course-1/pull/14	12
https://github.com/gjovanov/facer/pull/18	13

Table 6.4: Accepted Pull Requests.

as code). For Docker, Cito et al. (2017) examined Dockerfile quality and, similar to us, found a high rate of breakage in Dockerfile builds; they cite a 34% breakage rate from a smaller sample of 560 projects. We found a comparable breakage rate, but have also developed methods aimed at making *repairs* instead of just analyzing quality. More recently, Wu et al. (2020) conducted a comprehensive study of build failures in Dockerfiles. They analyzed a total of 3,828 GitHub projects containing Dockerfiles, and a total of 857,086 Docker builds. Overall, they found a failure rate of 17.8%. Despite the differences in failures rates, these studies corroborate our finding that build failures are prevalent. Lin et al. (2020) analyzed patterns (i.e., good and bad practices) in Dockerfiles. Among various observations, they found that many Dockerfiles use obsolete OS images (which can pose security risks because attackers could exploit documented vulnerabilities) and incorrectly use the latest tag. Xu and Marinov (2018) investigated characteristics of Docker images from DockerHub. Among other findings, they listed opportunities to improve Software
Engineering tasks based on how images are organized. For example, they report that image variants could be used to support combinatorial testing. Zerouali et al. (2019) studied version-related vulnerabilities (yet another category of issues that may arise in Dockerfiles—similar to some of the build-breaking issues we observed, in which *external changes in the environment* negatively effect a Dockerfile). Among various findings, they found that no release is devoid of vulnerabilities, so deployers of Docker containers cannot avoid vulnerabilities even if they deploy the most recent packages.

## 6.6.2 Analysis of Dockerfiles

In Chapter 5 we created a static checker for Dockerfiles (similar to Hadolint (Hadolint, 2019)), called **binnacle**, which is capable of learning rules from existing Dockerfiles; however, unlike **Shipwright**, neither **binnacle** nor Hadolint attempts *repairs*. Xu et al. (2019) examined "Temporary File Smells," which are an *image-quality*-related issue, not a *build-breaking* issue, such as the ones we examined. Zhang et al. (2018b) studied the effect of Dockerfile changes on build time and quality (and utilized the static tool Hadolint). Hassan et al. (2018b) proposed RUDSEA: a tool-supported technique that proposes updates in Dockerfiles. RUDSEA analyzes changes in software environment assumptions—obtained with static analysis—and their impacts. We consider RUDSEA and Shipwright to be complimentary approaches: RUDSEA focuses on changes within a project and Shipwright focuses on changes external to a project. Other empirical studies on DevOps, but not Docker, include an examination of *smells* in software-configuration files (Sharma et al., 2016), and a study of the coupling between infrastructure-as-code files and "traditional" source-code files (Jiang and Adams, 2015).

#### 6.6.3 Automated Code Repairs

Shipwright lies within the growing body of work in automated repair. According to a recent survey (Gazzola et al., 2019), our approach can be classified as both *Generate-and-Validate* and *Fix Recommender*. We use pre-defined templates that are obtained (i) via the analysis of build logs extracted from our clusters, and (ii)

from examples found in community websites. As such, we side-step the challenge of a fully automatic repair process to produce acceptable fixes. In addition to automated repair of source code, there is a growing effort to automate repair of build-related (DevOps) artifacts. These DevOps artifacts are unique in that they are often tied to both a source repository and the broader external environment in which one wants to build, test, and/or run their code. In the broader context of repair for build-related artifacts, both Lou et al. (2019) and Hassan and Wang (2018) investigate repair in the domain of Gradle builds (a kind of DevOps artifact used in many Java projects) and Macho et al. (2018) explore the related problem of automated repair for Maven builds.

#### 6.6.4 Broken Updates in Package Managers

Prior work investigated impacts of breaking changes in package managers. Mancinelli et al. (2006) formalized package dependencies within a repository, and encoded the installability problem as a SAT problem. Vouillon and Cosmo (2013) proposed an algorithm to identify *broken sets* of packages that cannot be upgraded together within a component repository. McCamant and Ernst (2004) proposed an approach for checking incompatibility of upgraded software components. They compute operational abstractions based on input/output behavior to test whether a new component can replace an old one. Møller and Torp (2019) proposed a model-based testing approach to identify type-regression problems that result in breaking changes in JavaScript libraries. These works deal with improvements and repairs applied to a package repository or library, and thus have a different focus compared to Shipwright, which is on repairing *broken* Dockerfiles.

More related to checking inconsistencies of client code, Tucker et al. (2007) proposed the OPIUM package-management tool. Given a set of installed packages and information about dependencies and conflicts, they used a variety of solvers to determine (i) if a new package can be installed; (ii) the optimal way to install it; and (iii) the minimal number of packages (possibly none) that must be removed from the system. Shipwright does not rely on explicit information about dependencies

(which might not be available or feasible to obtain). Instead, it extracts information from build logs, and leverages community knowledge bases to find solutions. This approach enables **Shipwright** to address problems that go beyond broken packages and conflicts.

# 6.7 Future Work

One interesting direction for future work in the realm of *automated repair* for DevOps artifacts (like Dockerfiles) is the possibility of *learning* repairs. For Shipwright, we took an approach that we knew would work—it is an interesting (and open) question as to whether an end-to-end learned solution is feasible in this domain. I think that, five years ago, my answer would have been a definite "no," but now, I wonder if an approach could be bootstrapped by large models trained on code and simply fine-tuned on the repair task. Granted, getting aligned data (pairs of "good" and "bad/broken" code snippets) would be a significant challenge and require either manual effort or clever tricks to surmount. (One could, perhaps, synthesize aligned pairs by mutating known good code, but the distribution of this synthetic data would not necessarily match the distribution of *real* broken files.)

# 6.8 Notes

For this work I had never met my collaborators before in person—instead, we developed the ideas, tools, and eventual paper while working completely remotely. Getting to collaborate across time zones with people I had never met was a unique experience, and one I am grateful to have had during my graduate studies. Additionally, this was the second piece of work I wrote targeting Dockerfiles—I think it was nice to follow a similar arc to the work I did learning on code by, first, setting the stage (collecting a dataset, designing basic techniques), and then expanding upon that with follow-on work (in this case repair, and in the case of learning from code, specification mining). What would Data Science look like if applied to the domain of code?

GPT-3: This is a difficult question. Code is a complex domain with many variables and interactions. Data science applied to code would likely involve a lot of data mining and machine learning in order to identify patterns and relationships.

A dialogue with OpenAI's GPT-3

# 7.1 Introduction

Thus far, we have explored learning from code (Chapter 2); applying embeddings learned from code to specification mining (Chapter 3); investigating the robustness of models learned from code (Chapter 4); mining best-practices from Dockerfiles (Chapter 5); and, automatically repairing Dockerfiles (Chapter 6). All of these chapters share something in common: there was a *significant* amount of engineering work to produce the tools and experiments featured in each chapter. If we look carefully at this engineering work, we can find one feature that was always present: some way to extract data (from either code or non-code sources) and *represent/encode* that data. In each of these investigations the collection, filtering, and representation of data is the most time-consuming component. In this chapter, I present one last (ongoing) piece of work I've undertaken—a framework for doing "Data Science" on code (and non-code) artifacts called **code-book**.

## 7.1.1 Motivation

To do empirical software engineering research, there are always some number of "data chores" to address. These are things like collecting code, filtering it, and transforming it into a representation suitable for study. These chores often require bespoke tooling and encoding formats. In contrast, in the Data Science community, there is a vibrant ecosystem of tools for "data chores" that is centered around the idea of interactive *notebooks*. Given the existence of such stellar tooling in another domain, one might wonder what we can do for empirical software engineering. We start, therefore, with a simple motivating question:

#### Motivating Question

What would *Data Science* look like for code?

## 7.1.2 Goals

It is critical for both empirical software-engineering researchers and software engineers to have ways to interact with, understand, and represent code (and non-code) artifacts. Unfortunately, many systems for verifying, querying, and extracting data from programs require users to be sophisticated in their understanding of program structure, semantics, and often one (or many) baroque configuration languages or query languages. In contrast, the Data Science community has many tools for introspecting and interacting with data in ways that require little domain expertise. Therefore, to improve the state-of-practice in the domain of code (and non-code artifacts), we propose the following goal:

#### Goals

Provide a system for introspecting and interacting with *code* that: (i) requires little domain expertise, (ii) mirrors the existing Data Science tooling (interactive, notebook-based, integrated with off-the-shelf libraries), and (iii) reduces the need for bespoke tooling and pipelines in software-engineering research.

To meet this goal, we introduce code-book; code-book is an interactive notebookbased framework for asking questions about massive amounts of code. Questions are asked by writing code snippets, questions are answered by supplying the user with a data frame—a two-dimensional table, much like a spreadsheet—of results. Further analysis, visualization, and introspection happens, interactively, via a notebook-based interface and off-the-shelf data science libraries.

## 7.1.3 Contributions

In this chapter, we introduce a novel language for querying code. We describe the syntax and semantics of this language, and sketch how this language can be translated to Datalog for efficient processing. This new language for code querying is the primary contribution of **code-book**. To meet our goal of providing a system that requires little domain expertise, the language we have devised allows for Query-by-Example-style code snippets. By allowing users to simply write code, we avoid needing users to understand how code is parsed, stored, and encoded.

In the remainder of this chapter, we introduce code-book's query language (Section 7.2) and we provide three case studies on using code-book in various settings (Section 7.3). We conclude by considering related work (Section 7.4) and describe avenues for future work (Section 7.5).

# 7.2 Query Language

code-book uses a custom query language, like many other state-of-the-art code querying tools. However, unlike most of those tools, code-book was built with the explicit goal of requiring little domain expertise to use. To use both a powerful (custom) query language *and* not ask our users to have expert-level understanding of code and its many representations, code-book leverages an old idea from the databases community: Query by Example (Zloof, 1975).

Query by Example (QBE) was a technique in which users would formulate queries graphically by entering example elements and conditions into visual tables. One elegant aspect of QBE was its ease of use: a user did not need to know meta-properties about what kinds of objects they wanted to retrieve, nor did they need to be wellversed in a specific query language (such as SQL). code-book seeks to emulate the successes of QBE in the domain of querying for code. For code, the QBE approach is particularly beneficial because the representations of code used by **code-book** for serving queries are complex and rely on parsing and analyzing the target codebase. To query these representations directly would require a user to have some understanding of how code is parsed and what schema **code-book** uses to store the results.

Although code-book seeks to reduce the barrier to entry for users without expertise, code-book also needs to be a capable tool for experts. Therefore, the query language code-book uses is not just a Query-by-Example system. Instead, code-book defines a custom query language in which complex queries can be composed by writing snippets (examples). These snippets are "just code," but they can also contains *slots* and ignores. A slot is a variable in a pattern and, when matching is performed, whatever the variable (slot) gets bound to is returned in the query results (in this manner, slots operate similar to a "wildcard"). For example, if we are searching for any call taking a single argument where that single argument is the integer literal zero, we would write the following code-book query: call(0). In this snippet, the call expressionis a "slot" and matches code such as: exit(0) or my\_func(0), which would then be returned as the answer. Sometimes, we want a wildcard-like expression, but we do not care about what value that wildcard ends up having. In such cases, we can use an ellipsis in our query snippets (which we will refer to as an "ignore" construct); for example the printf(...) query snippet says, "match any call to printf with zero, one, or many arguments."

In the following subsections, we introduce code-book's query language via a few real-world query examples (Section 7.2.1). Additionally, we describe the syntax of code-book's query language (Section 7.2.3) and give a brief description of the semantics of the code-book query language (Section 7.2.4). Finally, we present a (simplified) overview of how queries are translated into Datalog (Section 7.2.5); the generated Datalog is executed to produce results that can be returned to users in a variety of encodings (visually, as a data table with clickable links, or in a structured format like Pandas' DataFrames).

## 7.2.1 Query Examples

In this section, we supply practical examples of code-book queries (with increasing complexity). By composing small snippets (which are, essentially, invocations of the Query-by-Example paradigm), we can create more sophisticated queries. Say we wish to find foreach loops or calls to ForEach where the loop body contains a call to any function that has, as its first argument, the loop variable. We could write this as follows:

```
alias $matching_call = [
1
      $anything($item, ...)
2
3
    4
    match [
5
      foreach (... $item in $collection) {
6
 7
        §matching_call;
      }
8
    9
      $collection.ForEach(($item) => $matching_call);
10
    11
```

In the above example, we use the alias keyword to assign a name to a Query-by-Example (QBE) snippet. We then reference this snippet (using the same syntax we use for slots) in the top-level match construct. Both match and alias are allowed to contain a disjunction of several snippets (delineated by the  $\parallel$  character). Together, the ability to specify disjunctions and compose QBE snippets (which can contain slots and ignores) enables a moderately sophisticated (but conceptually simple) query system (see Sections 7.2.3 and 7.2.4 for details on query syntax and query semantics).

Another key feature of code-book's query language is the ability to constrain the *text* of a given slot. For example, one might wish to find all calls starting with the text log that have, as their first argument, a string literal. To formulate such a query we allow for slots to be pre-declared using a let *§slot = <pattern>* construct. By pre-declaring a slot, users can constrain the allowable values a given slot may represent.

Given this, we could write the following code-book query to capture the aforementioned calls starting with log:

In the above, we use a regular expression to constrain the allowable text the  $starts_with_log$  slot may contain. We also use a special "string slot" construct (s'fmt') which constrains the *type* of the slot so that only string literals can match. (To make code-book's query language work across several target languages, we defined language-specific configurations that encode information like what Concrete Syntax Node types correspond to string literals.) Note that, in this context, we use the word type to refer to the *type of the Concrete Syntax Node* that the slot may bind. We are *not* referring to the type of the construct encoded in the given CST node. (Although "a" + "b" is an expression of type string, it is not a CST node with a string literal type.)

To sum up, the basics of code-book's query language are as follows. Each query has a top-level match  $[ \ . . . ]]$  construct that contains a (disjunction) of Query-by-Example-style snippets. These QBE snippets may contain *slots* and/or *ignores*. A slot (§slot) is used to bind any Concrete Syntax Node and may, optionally, be constrained by pre-declaring the slot with a let §slot = <pattern> statement. An ignore (. . . ) binds any Concrete Syntax Node but does *not* yield the matching node as part of the query results. Finally, to compose more sophisticated queries, one may use several alias §name =  $[ \ . . . ]]$  constructs which can be referenced in other QBE snippets to form more complicated expressions. Most top-level constructs allow for some form of disjunction using the || operator (e.g., alias x = [ foo(. . . ) || bar(. . . ) ]]).

## 7.2.2 Advanced Queries

In this section, we introduce four advanced query constructs that are intended for more experienced users:

- 1. Advanced Slots: § < §parentSlot, §alias \*> §descendant, and § ~= §aliasOf
- 2. Unless Constraints: unless(<constraint>) [[ . . . ]].
- 3. Guard Conditions: if (<guard-condition>) <match|alias> [[ . . . ]].
- 4. Similarity Constraints: let §slot = (~semantically|~similar|~words).

#### Advanced Slots.

Although the concept of a *slot* is relatively simple, in practice there are many variations needed to cover a wide variety of matching situations. In particular, sometimes it is necessary to bind the *parent* of a given slot, sometimes we would like to bind to any *descendant* of a slot, and sometimes we'd like to bind to any *alias of* a given slot. Let's work through each of these scenarios individually.

Imagine you want to select all of the arguments to a specific function call something like, print(§all\_args). By default, that Query-by-Example snippet would be interpreted as matching calls to print with one argument (bound to the §all\_args slot). To, instead, bind *all* arguments (and match calls to print with zero, one, or many arguments) we can use a *parent slot* and write print(§ < §all\_args). In this query snippet we use a slot but, instead of binding to the location the slot is specified in, we bind to the *parent* of the slot. By parent, in this context, we are referring to the parent of the slot Concrete Syntax Tree (CST) node (in the parsed query CST).

To visualize this, let's encode the scenario above using query CSTs written as S-Expressions. For the query without the parent slot (print(§all\_args)), we have the query CST: (call name: (id print) args: (argument (slot §all\_args))). When we replace the (slot §all\_args) node with a *parent slot*, we are binding the (argument (...)) sub-tree instead of the first child of the argument node.

Another common scenario is needing to bind to *any descendant* of a given construct. For a real-world example, let's consider trying to match a **break** statement within a loop. If we write match [ for (...) { **break**; } ] we will only match for loops where the loop body contains a break statement. We will *not* match for loops where the loop body contains a statement that has, nested within it, a **break** (e.g., if (...) { **break**; }). Instead, if we wish to match a break statement nested *anywhere* inside a for loop, we can use a *descendant slot*.

In the following query we use a descendant slot to match any break statement nested within for or while loops:

```
alias §a_break = [ break ]
1
 2
 3
    match [
 4
      for (...) {
        § *> §a_break;
 5
      }
 6
 7
    while (...) {
 8
 9
        § *> §a_break;
      }
10
    11
```

#### Unless Constraints.

There are many situations where one might want to match code *unless* some other constraint exists before, after, or within the match. For example, one might wish to match calls to a constructor where the default values are insecure *unless* one of those values is overwritten with a more-secure value. To express such a query we can write something like the following:

```
1 match [[
2 new SomeBadConstructor($ < $args)
3 ]] unless(exists-within: $args) [[
4 SomeEnumeration.MoreSecureOption
5 ]]</pre>
```

Note that the unless construct takes an "argument" of sorts: each unless needs to have a *constraint* that specifies how the QBE snippet in the unless body relates to the overall query. In the case of the example above, we employ the *exists-within: §slot* constraint to specify that the query matches unless the QBE snippet in the unless construct exists within the argument list bound to the **§args** slot.

#### Guard Conditions.

Sometimes, one wants to match code but only under specific conditions. If these conditions are related to semantic properties and not directly to the *structure* of the code (that is, the Concrete Syntax Tree) we allow users to employ Guard Conditions. A Guard Condition is a logical expression that will be used to filter matches. For an example, consider matching classes that inherit from a specific base class or interface (like IEnumerable). With just code structure (parse trees) it is easy enough to look for *direct* inheritance, but the transitive closure of the "inherits" relation is harder to express. In such a situation, we can use Guard Conditions to express our query:

```
if (
1
2
      inherits_transitive($base, [IEnumerable])
3
   )
   match [
4
      class $collection : $base
5
      {
6
7
        . . .
      }
8
   9
```

In the above query, we are matching class declarations that inherit (directly) some base class. We use a slot to capture the name of the base class and the name of the class inheriting it. Finally, we employ a Guard Condition to require that the base class we match inherits (transitively) from the IEnumerable interface.

#### Similarity Constraints.

code-book takes a novel approach to code querying that mixes structural constraints with "fuzzy" constraints powered by learned representations of code. To leverage these "fuzzy" constraints in our query language we allow users to constrain the text of slots with *Similarity Constraints* in addition to the (previously discussed) regular-expression-based constraints. Consider looking for calls *semantically similar* to log—one might expect to find calls like trace or debug. We could write a Regular Expression containing a disjunction of these possibilities, but it would be better if we did not have to guess at what log-like terms the target codebase contains. Instead, we can leverage Similarity Constraints like so:

1 let \$log\_like = (~log)
2
3 match [[
4 \$log\_like(...)
5 ]]

In the above query, we are using a Similarity Constraint (let  $log_like = (~log)$ ) to constrain the call in the match body ( $log_like(...)$ ) to calls that have names *semantically similar* to the word log. To power such queries we make use of code embeddings (like those introduced in Chapter 2).

## 7.2.3 Query Syntax

The code-book query language uses a syntax based on a few simple principles: (i) the top-level query file is a list of zero, one, or many let-bindings, followed by zero, one,

 $\langle query \rangle :::= \langle lets \rangle \langle aliases \rangle \langle match \rangle \\ \langle lets \rangle :::= \langle let \rangle \setminus n \langle lets \rangle | \epsilon \\ \langle aliases \rangle :::= \langle alias \rangle \setminus n \langle aliases \rangle | \epsilon \\ \langle match \rangle :::= \langle if-guard \rangle match[ \langle lang \rangle ] [[ \langle code-snippets \rangle ]] \langle unless \rangle \\ \langle alias \rangle :::= alias[ \langle lang \rangle ] § \langle s-name \rangle = [[ \langle code-snippets \rangle ]] \langle unless \rangle \\ \langle let \rangle :::= let \langle s-name \rangle = \langle pattern \rangle$ 

Figure 7.1: A (high level) grammar for code-book queries

or many alias declarations, followed by exactly one top-level match block and each match/alias block has, optionally, a trailing unless block, (ii) each alias, match, and unless block has a body containing code written in the target language of the query (and, thus, largely conforming to the syntax of the query's target language); (iii) every time we write a code snippet in the query's target language we may intersperse zero, one, or many slot or ignore constructs.

Figure 7.1 provides the coarse structure of a code-book query. From Fig. 7.1, we can see that a <query> is made up of let statements (e.g., let \$x = ...) followed by aliases (e.g., alias[C#] \$a = [[ ... ]]) followed by exactly one top-level match construct (e.g., match[C#] [[ ... ]]). For novice users, the general structure of a query can be simplified to a file containing one top-level match that contains an example code-snippet that resembles the code that user wants to match.

One major feature of many code-querying tools is the ability to select code that matches a *textual* constraint. For example, consider matching functions that have a name containing a certain word. To write such a query we can use **let** constraints. These *textual* constraints must conform to the grammar given in Fig. 7.2.

One novel aspect of code-book is the ability to mix fuzzy (embedding-based) constraints with *structural* constraints. The main way we incorporate this feature is through *textual* constraints that are based on semantic similarity instead of strict

```
 \langle pattern \rangle ::= \langle p-not \rangle \langle regular-expr \rangle \\ | \langle p-not \rangle \langle semantic-sim-expr \rangle \\ \langle p-not \rangle ::= ! | \epsilon \\ \langle regular-expr \rangle ::= /\langle regex \rangle / \\ \langle semantic-sim-expr \rangle ::= (\langle sim-constraint \rangle) \\ \langle sim-constraint \rangle ::= \sim \langle word \rangle | \langle sim-constraint \rangle \\ | \sim \langle word \rangle
```

Figure 7.2: A grammar for textual constraint in code-book queries

(regular-expression based) matching. To strictly constrain text using a let-binding we can write the following: let \$x = /.\*(log).\*/. Unfortunately, such a constraint would miss functions like report or trace which may also be relevant to the user's query. To address this issues, one can use the fuzzy constraint let \$x = (~log)—which will find functions *semantically like* log (e.g., trace, debug, report, etc.).

Sometimes, in advanced use cases, one may wish to query for code that both matches one structural constraints while *not* matching another (e.g., finding loops that are *not* nested within other loops). For this purpose, we introduce an unless block that may (optionally) appear after the top-level match block. Figure 7.3 gives the grammar for unless blocks.

Similar to unless, sometimes there are constraints that require additional analysis to express and encode. For instance, it is not easy to capture the transitive closure of the "inherits" relation between classes in an object-oriented language via purely structural constraints. Instead, we provide advanced users with an "escape hatch" that allows them to express additional constraints on a match block. These constraints are directly inlined into the generated Datalog and rely on user or system-defined relations. To express such constraints we introduce an <if-guard> construct that may (optionally) precede a match block and must conform to the grammar given in Fig. 7.4.

```
\langle unless (\langle binding \circle: \langle target \rangle) [[ \langle code-snippets \rangle ]]
\langle target \circle::= \langle slot \rangle
| \ss
\langle binding \circle::= matches
| contains
| exists-within
| exists-before
| exists-after
```



```
 \begin{array}{l} \langle if\text{-}guard \rangle :::= \text{ if } (\langle if\text{-}expr \rangle) \mid \epsilon \\ \langle if\text{-}expr \rangle ::= ! \langle if\text{-}expr \rangle \\ \mid (\langle if\text{-}expr \rangle) \\ \mid \langle if\text{-}expr \rangle \mid \langle if\text{-}expr \rangle \\ \mid \langle if\text{-}expr \rangle \mid | \langle if\text{-}expr \rangle \\ \mid \langle if\text{-}constraint \rangle \langle comp\text{-}op \rangle \langle if\text{-}lit \rangle \\ \mid \langle if\text{-}lit \rangle \langle comp\text{-}op \rangle \langle if\text{-}lit \rangle \\ \mid \langle if\text{-}constraint \rangle \\ \langle if\text{-}constraint \rangle \\ \langle if\text{-}constraint \rangle \\ \langle if\text{-}constraint \rangle :::= \langle c\text{-}name \rangle (\langle args \rangle) \\ \langle args \rangle :::= \langle if\text{-}lit \rangle, \langle args \rangle \mid \langle if\text{-}lit \rangle \mid \epsilon \\ \langle if\text{-}lit \rangle :::= \$ \langle s\text{-}name \rangle \\ \mid \langle integer \rangle \\ \mid \langle boolean \rangle \\ \mid \llbracket \langle raw\text{-}code \rangle \rrbracket
```



```
\langle code-snippets \rangle ::= \langle code-snippet \rangle \parallel \langle code-snippets \rangle
        \langle code-snippet \rangle
\langle code\text{-snippet} \rangle ::= \langle raw\text{-}code \rangle \langle code\text{-snippet} \rangle
         \langle slot \rangle \langle code-snippet \rangle
         \langle ignore \rangle \langle code-snippet \rangle
        [ \langle inline-choice \rangle ] \langle code-snippet \rangle
         \langle raw-code \rangle
\langle inline-choice \rangle ::= \langle raw-code \rangle \parallel \langle inline-choice \rangle
   |\langle raw-code \rangle
\langle iqnore \rangle ::= \ldots | ' \ldots '
\langle slot \rangle ::= \langle normal-slot \rangle
        \langle parent-slot \rangle
         \langle desc-of-slot \rangle
         \langle flows-from-slot \rangle
         \langle alias-of-slot \rangle
         \langle string-slot \rangle
         \langle parent-string-slot \rangle
\langle normal-slot \rangle ::= \$ \langle s-name \rangle
\langle parent-slot \rangle ::= \$ < \$ \langle s-name \rangle
\langle desc-slot \rangle ::= \$ *> \$ \langle s-name \rangle
\langle flows-from-slot \rangle ::= \$ <~ \$ \langle s-name \rangle
\langle alias-of-slot \rangle ::= \S \sim = \S \langle s-name \rangle
\langle string-slot \rangle ::= \$ ' \langle s-name \rangle '
\langle parent-string-slot \rangle ::= \$ < \$' \langle s-name \rangle'
```



Figure 7.5 gives the full grammar for code snippets used within code-book queries. Code snippets are an embodiment of the Query-by-Example paradigm; however, simply writing a code snippet is not enough to describe a pattern that has *many* possible matches. Instead, we intersperse slots, ignores, and inline choices in our code snippets. With these three additions, we can construct a snippet with "holes" that will yield *multiple* matches. For example, to match a call to print with zero arguments we might write the literal code: print(); to match, instead, *any* call to print we can use an ignore to write: print(. . . ). If, instead, we wanted to match calls to print and *select* the second argument passed to the call we could use both an ignore and a slot, like so: print(. . . , §arg). Finally, if we wanted to match code where there are a fixed number of alternatives, we can use inline choice; for example, to search for asserts with either equality or inequality tests, we write assert §a [==||!=] §b.

The syntax of the code-book language is implemented using a modern incremental parser generator called tree-sitter (GitHub, 2022). To handle the nested languages within code-book queries, we invoke our own custom processing to do second-level parses and handle the slots, ignores, and inline choices that are interspersed throughout the target code snippets (to do this, we take inspiration from the *phased-parsing* technique introduced in Chapter 5).

## 7.2.4 Query Semantics

What does it mean to query for code? To answer this question, we must first be precise about what code exactly means in the context of our queries. To code-book, code will be represented as a set of parse trees (Concrete Syntax Trees)  $\mathbb{T}$  (generated by tree-sitter, with a node-type vocabulary  $V_T$ ), a set of source fragments  $F(\mathbb{T})$ , and a set of embeddings  $E(\mathcal{M}, \mathbb{T})$  (where  $\mathcal{M}$  is a configurable model—any model capable of taking text and producing embeddings will suffice).

Although we will work against a corpus of parse trees (and source fragments and embeddings derived from those trees), it will be more convenient to consider the set of all sub-trees of the original corpus; given the set of all sub-trees, we formulate the set of source fragments as, instead, a function from the set of sub-trees to a space of text strings S; similarly, we formulate the set of embeddings as a function from the set of sub-trees to an n-dimensional vector space of embeddings ( $\mathbb{R}^n$ , where n is implicitly defined by the model  $\mathcal{M}$ 's output dimension). We will denote the set of all sub-trees of our input parse trees  $\operatorname{sub}(\mathbb{T})$ . For any  $t \in \mathbb{T}$ , we will use  $E(\mathcal{M}, t)$  to refer to the specific embedding for the sub-tree t, and use F(t) to refer to the source fragment corresponding to the sub-tree t.

A query will be a function on the set of sub-trees of our input corpus that marks each sub-tree as either included (1) or excluded (0) from the query results (a characteristic function over the set of sub-trees). This description is somewhat of a simplification of how **code-book** operates in practice—in practice, one wants not only the matching sub-trees but "pointers" to specific pieces of the matching sub-trees that correspond to the slots a user included in their query. For the sake of clarity, we will consider the simpler scenario where we only need to decide whether each sub-tree matches.

Now, let us consider the semantics of a query. We consider a query Q to be a characteristic function over the sub-trees of our input corpus  $(\operatorname{sub}(\mathbb{T}))$ . That is, Q is a function from the set of sub-trees of our input corpus to a Boolean value that denotes whether the input sub-tree matched our query  $(Q : \operatorname{sub}(\mathbb{T}) \to \{0, 1\})$ . In practice, we can think of the query as denoting a higher-order function that, given a representation of the let patterns ( $\mathbb{P}$ ) and aliases ( $\mathbb{A}$ ), returns a function capable of deciding whether a given sub-tree matches. In the following, we use the  $\mathbf{M}[\![X]\!]$  notation to denote the *meaning of* X. We describe the high-level semantics of a **code-book** query as follows:

# $$\begin{split} M[\![ <\!\!\mathsf{query} > ::= <\!\!\mathsf{lets} > <\!\!\mathsf{aliases} > <\!\!\mathsf{match} > ]\!] &= M[\![ <\!\!\mathsf{match} > ]\!](\mathbb{P}, \mathbb{A}) \\ & \text{where } \mathbb{P} = M[\![ <\!\!\mathsf{lets} > ]\!] \text{ and } \mathbb{A} = M[\![ <\!\!\mathsf{aliases} > ]\!] \end{split}$$

First, let us consider let patterns (let <s-name> = <pattern>). We use L to denote the set of all slot names (<s-name>s). For each named slot, we associate two characteristic functions: one over the set of source fragments (F(T)) and one over

the set of sub-trees  $(\operatorname{sub}(\mathbb{T}))$ . Initially, a slot \$x is unbound (unconstrained) and, as such, the associated characteristic functions for \$x are 1 everywhere. If we define a let pattern over \$x, we update the characteristic function over the set of source fragments associated with \$x. We define the semantics of let (in the recursive case) as follows:

$$\begin{split} \mathbf{M}[\![ <\!lets > ::= <\!let > <\!lets > ]\!] : \mathbf{L} \to (\mathbf{F}(\mathbb{T}) \to \{0, 1\}) \\ \mathbf{M}[\![ <\!lets > ::= let \$p = <\!pattern > <\!lets > ]\!](\$x) \\ &= \begin{cases} \mathbf{M}[\![ <\!pattern > ]\!] & \text{if } \$x = \$p \\ \mathbf{M}[\![ <\!lets > ]\!](\$x) & \text{if } \$x \neq \$p \end{cases} \end{split}$$

In the non-recursive case, we return a function that includes *all* source fragments. Note that, when no **let** binding "matches" a given **<s-name>**, we permit all source fragments (which is the desired behavior because slots, by default, are completely unconstrained).

## $\mathbf{M}\llbracket \text{ <lets> } ::= \varepsilon \rrbracket = \lambda \$x, \lambda f \in F(\mathbb{T}).1$

There are two primary styles of textual constraints in code-book: regex-based constraints and "fuzzy" constraints based on semantic similarity. When we use constraints based on semantic similarly there is always a hidden parameter: the maximum cosine distance between vectors ( $\tau$ ). We describe the semantics of these different constraint styles as follows:

$$\begin{split} \mathbf{M}[\![ <\! \mathsf{pattern} > ::= <\! \mathsf{regular}\!-\!\mathsf{expr} > ]\!] : \mathsf{F}(\mathbb{T}) \to \{0,1\} \\ \mathbf{M}[\![ <\! \mathsf{pattern} > ::= <\! \mathsf{regular}\!-\!\mathsf{expr} > ]\!](\mathsf{f}) = \mathbf{M}[\![ <\! \mathsf{regular}\!-\!\mathsf{expr} > ]\!](\mathsf{f}) \\ \mathbf{M}[\![ <\! \mathsf{pattern} > ::= <\! \mathsf{semantic}\!-\! \mathsf{sim}\!-\!\mathsf{expr} > ]\!](\mathsf{f}) = \mathbf{M}[\![ <\! \mathsf{semantic}\!-\! \mathsf{sim}\!-\!\mathsf{expr} > ]\!](\mathsf{f}) \\ \mathbf{M}[\![ <\! \mathsf{semantic}\!-\! \mathsf{sim}\!-\!\mathsf{expr} > ::= (\sim w_1 | \sim w_2 | ... | \sim w_k) ]\!](\mathsf{f}) = \bigvee_i \cos(\overrightarrow{\mathsf{f}}, \overrightarrow{w_i}) < \tau \\ & \text{where} \quad \overrightarrow{\mathsf{f}} = \mathsf{E}(\mathcal{M},\mathsf{f}) \text{ and } \overrightarrow{w_i} = \mathsf{E}(\mathcal{M},w_i) \end{split}$$

We allow for the semantics of the regex matching (**M**[ <regular-expr> ]) to be implementation-dependent (based on the regular-expression engine used in the implementation). We also allow for an additional syntax for matching based on simpler operators (starts with, ends with, and contains) that we do not present here. (This simpler pattern syntax can allow for some query optimizations.) Finally, we note that each pattern type (regex-based or embedding-based) may optionally be negated.

Next, we continue our presentation of **code-book**'s query-language semantics with the **<alias>** construct. An **alias** \$x = [[ . . . ]] updates the characteristic function over the set of sub-trees of our input corpus associated with \$x. When we refer to a slot that which has an **alias** constraint, we are using the slot as a placeholder that represents the set of sub-trees allowed under the associated characteristic function defined by the slot. The semantics of **alias** is as follows (we will assume that aliases are sorted based on a topological ordering of their dependence graph—that is, if alias A is used in alias B, then we assume alias A has been processed prior to alias B; we also assume that all let patterns have been processed prior to alias processing):

$$\begin{split} & \mathbf{M}[\![ \ \texttt{caliases} \ ::= \ \texttt{calias} \ \texttt{caliases} \ ]\!] : \mathbf{L} \to (\operatorname{sub}(\mathbb{T}) \to \{0,1\}) \\ & \mathbf{M}[\![ \ \texttt{caliases} \ ::= \ \texttt{alias} \ \$a \ = \ [\![ \ \texttt{code-snippet} \ ]\!] \ \texttt{culless} \ \texttt{caliases} \ ]\!](\$x) \\ & = \begin{cases} & \lambda t \in \operatorname{sub}(\mathbb{T}) \ . \ \mathbf{M}[\![ \ \texttt{code-snippet} \ ]\!](t) \ \land \neg \mathbf{M}[\![ \ \texttt{culless} \ ]\!](t) & \text{ if } \$x \ = \$a \\ & \mathbf{M}[\![ \ \texttt{caliases} \ ]\!](\$x) & \text{ if } \$x \ = \$a \\ & \mathbf{M}[\![ \ \texttt{caliases} \ ]\!](\$x) & \text{ if } \$x \ \neq \$a \end{cases} \end{split}$$

Again, as with let patterns, in the non-recursive case we return a function that accepts "everything" (which, for let, meant any source fragment; for alias, this means we accept any possible sub-tree):

$$M[\![ < \texttt{aliases} > ::= \epsilon ]\!] = \lambda \$x, \lambda t \in sub(\mathbb{T}).1$$

With alias, we have a way to label and reuse expressions (which proves quite useful in more advanced query scenarios).

We have not yet addressed one of the core concepts in code-book's query language:

the idea that users may write Query-by-Example (QBE) code snippets that embody the code they wish to match. To describe the semantics of these QBE snippets, we will need some extra machinery. First, let use describe a function that, given two sub-trees, reports whether they are compatible. Our parse trees—generated by tree-sitter—have labelled nodes (labelled with the *type* of the node—e.g., expression or identifier, etc.). Nodes also have values (most nodes have no value, but many leaf nodes have a value that represents the tokens from the source text corresponding to the node; e.g., an identifier node will have a value containing the identifier's name).

If we have two trees with the same number of nodes, the same node type labels, the same node values, and the same edges, we call these trees *compatible*. A more interesting scenario arises when we consider trees with different node-type vocabularies; specifically, trees generated by our query language and trees in our input corpus. The QBE snippets in a **code-book** query share the same node-type vocabulary as the trees in the input corpus ( $V_T$ ), but the QBE snippets may *also* contain slot and ignore expressions (and, therefore, we consider the node-type vocabulary for our queries' QBE snippets to be  $V'_T = V_T \cup \{ \text{slot}, \text{ignore} \}$ ).

Consider the following code-book QBE snippet:  $a + \ldots + c$ . We will write trees using collapsed S-expression-style notation; for example, we write the Concrete Syntax Tree for a + b + c as (expr (sum (id a) (id b) (id c))). We define the semantics of the above QBE snippet as follows:

 $\textbf{M[\![\$a + ... + c]\!]} = \big\{ \text{ (expr (sum (t_1) (t_2) (id c))) } : t_1, t_2 \in \mathrm{sub}(\mathbb{T}) \big\}$ 

In the above, we have described the set of trees that are *compatible* with the tree generated by the QBE snippet  $\$a + \ldots + c$ . In general, we replace every instance of a slot or an ignore in the Concrete Syntax Tree for a query snippet with a "wildcard" that allows for any sub-tree to take its place. Although the above definition is intuitive, we may have let patterns and aliases that impact the behavior of slots. If a let pattern or an alias is bound to a slot name, we must refine the above to look

more like the following:

$$\begin{split} \textbf{M}[\![ \$a + \ldots + c ]\!](\mathbb{A}, \mathbb{P}) &= \big\{ \text{ (expr (sum (t_1) (t_2) (id c)))} \\ &: t_1, t_2 \in \mathrm{sub}(\mathbb{T}) \ \land \ \mathbb{A}(\$a)(t_1) \ \land \ \mathbb{P}(\$a)(F(t_1)) \big\} \end{split}$$

In the above, we no longer allow  $t_1$  to be any sub-tree; instead,  $t_1$  must be one of the sub-trees accepted by the alias bound to \$a, and, if there is an applicable let pattern, the source fragment corresponding to  $t_1$  must also be accepted by the filter induced by the let pattern. We denote the set of trees *compatible* with a QBE snippet by C((cst ...)).

Finally, we can describe the semantics of a QBE snippet:

$$\begin{split} M[\![ <\!code-snippets> ::= <\!code-snippet>_1 \parallel ... \parallel <\!code-snippet>_k ]\!](t) \\ &= \lor_i M[\![ <\!code-snippet>_i ]\!](t) \\ M[\![ <\!code-snippet> ::= (cst ...) ]\!](t) = t \in C((cst ...)) \end{split}$$

We will forgo an in-depth presentation of the semantics of if-guards as they represent, in some sense, an "escape hatch" for our query language and rely on user-defined and/or implementation-specific relations to operate. In general, the <if-guard> construct acts similarly to <unless> in that it constrains the possible matches returned by a match block.

## 7.2.5 Query Translation

**code-book** uses Datalog to encode queries. We found that, for expressing questions about programs, Datalog is a natural choice. In this section, we provide a rough overview of how queries in **code-book**'s query language are translated into Datalog programs.

To start, each of the let constraints in a query are directly translated into Datalog. For each let \$x = /regex/ constraint, we produce a relation that invokes a custom User Defined Function (UDF) to test whether a given input source fragment matches the regular expression supplied in the constraint body. For example, if we write letx = /(a|b)/ in our query, we produce the following Datalog:

```
1 text_constraint_x(source) :- (
2   1 = udf_regex_matches(source, "(a|b)")
3 ).
```

For let constraints that rely on semantic similarity, we produce a similar construct. If we write let  $y = (-\log|-debug)$ , we produce the following Datalog (where the THRESHOLD value is a configurable parameter and may even be set on a per-query or per-constraint basis):

```
1 text_constraint_y(source) :- (
2 udf_cosine_distance(udf_embed(source), udf_embed("log")) < THRESHOLD
3 ;
4 udf_cosine_distance(udf_embed(source), udf_embed("debug")) < THRESHOLD
5 ).</pre>
```

In general, when we have to translate a Query-by-Example (QBE) snippet, we follow a reusable construction that, for each node in the parsed QBE snippet's Concrete Syntax Tree (CST), produces a few constraints on the node's type and the node's value (if applicable). For example, if we encounter a QBE snippet with the text 10 + a, and the snippet's CST is (sum left: (integer 10) right: (identifier a)), we generate the following Datalog:

```
1 ...
2 node(n1, "sum", _),
3 node(n2, "integer", "10"),
4 node(n3, "identifier", "a"),
5 child_of(n2, 0, "left", n1),
6 child_of(n3, 0, "right", n1),
7 ...
```

If a node in the QBE snippet's CST is an ignore node, we simply generate no constraints. If, on the other hand, a node is a slot, we generate similar constraints but elide the node's type. For example, if we have a QBE snippet with text  $\$x + \ldots$ , and the snippet's CST is (sum left: (slot x) right: (ignore)), we generate the following Datalog:

```
1 ...
2 node(n1, "sum", _),
3 node(n2, _, slot_x),
4 child_of(n2, 0, "left", n1),
5 ...
```

Note how we generate no node-type constraint for n2 (and, instead, bind the node's text so we can retrieve it later and return it to the user). Also note how there are no constraints generated for the right-hand side of the sum expression due to the ignore node.

We translate full alias and match blocks by translating their QBE snippets and joining the translated snippets via disjunction. We define separate relations for each alias and match block. The only behavior of note is when an alias, or a let pattern, is referenced in a QBE snippet. In such cases, we must reference the relation defined by the alias or let. For example, consider the following code-book query:

```
1
   let §sensitive = /secret.*/
2
   alias §print = [
3
     Console.WriteLine(...);
4
   5
6
7
   match [
     foreach (... in §sensitive) { §print }
8
   9
```

To translate the above query, we first translate the let and alias, producing:

```
text_constraint_sensitive(source) :- (
 1
 2
      1 = udf_regex_matches(source, "secret.*")
 3
    ).
 4
    alias_print(root) :- (
 5
      node(root, "call", _),
 6
      node(n1, "member_access", _),
 7
      node(n2, "identifier", "Console"),
 8
      node(n3, "identifier", "WriteLine"),
 9
10
      child_of(n1, 0, "function", root),
      child_of(n2, 0, "expression", n1),
11
      child_of(n3, 0, "name", n1)
12
   ).
13
```

Finally, we translate the top-level match block, producing:

```
the_match(root) :- (
 1
      node(root, "for_each", _),
 2
      node(n1, "identifier", sensitive_text),
 3
      node(n2, "block", _),
 4
      node(n3, _, print_text),
 5
      child_of(n1, 0, "right", root),
 6
 7
      child_of(n2, 0, "body", root),
      child_of(n3, _, _, n2),
 8
      text_constraint_sensitive(sensitive_text),
 9
      alias_print(n3)
10
    ).
11
```

In the implementation, we handle unless blocks, inline choices, pattern negations, and various other intricacies (including optimizing for the overall performance of the

query and the size of the returned data). We forgo a detailed presentation of these mostly implementation-specific details.

In general, the choice to translate queries from code-book's domain specific query language to Datalog allows for a great deal of flexibility. For example, if we wish to *change* how we store code as data, we can simply translate our query (as Datalog) to a language suitable for our backend storage (and, for many such storage engines, there are existing tools and techniques for translating from Datalog—e.g., there exists tools for translating from Datalog to SQL).

# 7.3 Case Studies

Using code-book, we were able to perform several studies that would have normally required significant investment in tooling and analysis. In this section, we provide three examples of real-world usages of code-book.

# 7.3.1 Case Study #1: Analyzing Python Notebooks

For our first case study, we used **code-book** to contribute to an ambitious analysis of *every Python notebook on GitHub* (in the years 2017, 2019, and 2020). For each notebook, we wanted to extract *implicit data-science pipelines*. An implicit data-science pipeline is a fragment of Python code (either in a single cell, or split across many cells in a notebook) that performs some data cleaning or preparation task. More specifically, we were asked to consider all of the operations that may occur between a data-loading operation (e.g., df = pandas.read\_csv(...)) and a model-training operation (e.g., sklearn.model.fit(df, ...)).

To accomplish this task, we expanded code-book's querying capabilities to understand data flows in Python notebooks. First, we wrote heuristics that found source and sink methods (data loading and model training methods, respectively). After surveying the most popular methods used for data loading and model training, we made a decision to think of pipelines as, concretely, a data flow between the result of a pandas.read\_\*(...) call and the first argument of a sklearn.\*.fit(...) call. With this definition in hand, we began to ask a more difficult question: how can we understand what happens to the data in an implicit data-science pipeline?

To answer this question, we leveraged **code-book**'s ability to quickly write heuristics, search code, and render visualizations. We went through many rounds of manual analysis of sources, sinks, and the code that existed on flows between these sources and sinks. As we observed commonalities, we codified what we observed through a growing library of abstractions. In this study, an abstraction was a **code-book** query that selected for a particular *kind* of data-science operation. For example, we wrote abstractions that looked for code patterns that indicated filtering and projecting data frames. (Data frames were important to our analysis as they are one of the most prevalent objects in data science notebooks—most often, data is loaded from external sources into Pandas data frames and, from that point, manipulated further by other APIs). If, on any data flow from a source to a sink, we found code that matched our filter or project abstractions, we recorded the matching code and the corresponding source and sink pair in a database.

By repeating this process, we developed a database full of source and sink pairs and the (abstract) operations that occurred between those source and sink pairs. We then performed higher-level analyses on the operations found in the database and drew conclusions about the prevalence and complexity of implicit pipelines in Python notebooks. (We found that, compared to explicit pipelines, implicit pipelines were both more prevalent and more complex.)

Overall, this work was a great success and, with code-book, we were able to perform a significantly deeper analysis of the notebooks corpus. We found that implicit pipelines were dominant (occurring approximately 5 times more frequently than implicit pipelines in our largest dataset); we also found that the operators occurring in implicit pipelines were more varied—in a coverage analysis, we found that it would take the top 10,000 implicit-pipeline operators to cover 50% of the operations in the pipelines we extracted. (Note: although we wrote code-book queries to extract abstract operators, many of these operators were parameterizable; for example, a GroupBy operator takes, as a parameter, the column on which the grouping should be performed.) Additionally, we found most implicit pipelines to be an order of magnitude longer that explicit pipelines. With this extra analysis, the results on Python notebooks in general (and the implicit pipelines contained therein) were accepted for an upcoming issue of the SIGMOD Record.

## 7.3.2 Case Study #2: Log Analysis

We then used **code-book** in a completely different scenario: to correlate production logs with the code that generated these production log messages. To accomplish this task, we again used **code-book**'s ability to refine code queries iteratively and interactively. At first, we thought the task of identifying log-producing code would be simple; instead, we found that our initial attempts at finding log-producing function calls covered only a small fraction of the messages being generated in a production environment.

Thankfully, code-book allowed us to quickly iterate and refine our heuristics for capturing log-producing function calls. We used fuzzy searches, based on semantic similarity, to identify a large number of log-producing functions we had missed. For example, we searched for calls with names semantically similar to the terms log, trace, and debug. Furthermore, using code-book's integrated visualization capabilities, we began to notice files in which we matched a few instances of log-producing function calls, but not others; via manual inspection, we were able to refine our queries to capture those corner cases. (For example, we found that we had missed log-producing calls where the format string used implicit string concatenation.)

In the end, we were able to create a satisfactory analysis that captured the majority of log-producing function calls; the data from this analysis is now being used by others to power further downstream techniques for managing production log streams. Together, our first case study (Python Notebook) and our efforts to generate data on log-producing functions pushed us to expand code-book to cover more target languages, more query features, and better data export capabilities.

## 7.3.3 Case Study #3: Studying Embedded Regex

In our third and final case study for code-book, we created a corpus of regular expressions by extracting them from source code. One might have tried to address this problem by extracting the necessary data without using sophisticated machinery. For instance, it might be the case that one finds most regular expressions by searching code (via grep or other more basic utilities) for text that matches the pattern: new Regex\("(.\*)"\). While this approach is workable, there are, inevitably, issues that arise. For example, if we wish for our analysis to be somewhat complete, we will have to craft many such heuristics (as there are many ways to use a regex); to compound this issue, such an approach would also need to deal with variations in syntax—it is possible that a regular expression may be assigned to a variable and then passed to a constructor or call. Finally, it is difficult to ascertain what we may be *missing* in the traditional approach. This uncertainty is always an issue in empirical studies of software: it is difficult to know what you do not know (or to understand where you may have gaps in your heuristics or analysis).

With code-book, we were able to do three things. First, we performed a fuzzy search for calls and constructors with names semantically similar to the phrase "regular expression." Based on these results, we were able to devise a handful of heuristics for capturing embedded regular expressions. Finally, we extracted, normalized, and cleaned the data in code-book using off-the-shelf data-science libraries. In this manner, we were able to create a dataset of 136,427 distinct embedded regular expressions in less than one hour (while covering variations in syntax). The ability to first perform a fuzzy search and follow that search up with an interactive session in which heuristics are created and refined is a key strength of code-book. Furthermore, working in an environment where one can ask questions about code, retrieve results, visualize those results, and apply off-the-shelf tooling (all without leaving a single tool) greatly reduces the overhead involved with generating new datasets.

# 7.4 Related Work

There are several examples of prior work that support queries on code. One of the earliest, ASTLOG (Crew, 1997), provides grep-like matching facilities via a variant of the Prolog language. This work is similar to ours in both the concept of querying programs and in the utilization of a logic-programming-inspired query language. However, our work expands upon these ideas by bringing in the modern concepts of embeddings, interactive notebooks, and utilizing the idea of Query-by-Example.

Later, Dyer et al. (2015) introduced Boa: an infrastructure for analyzing largescale software repositories (Upadhyaya and Rajan, 2018; Hung and Dyer, 2020). Although Boa serves a similar purpose to **code-book**, Boa focuses more on repository and file metadata and less on fine-grained analysis (such as the analysis of data flows we carried out in Case Study #1).

Commercially, there are many tools geared toward code querying. There are tools that use Query-by-Example-esque languages, such as Semgrep (2022) and Comby (2022). There are tools that focus more on the analysis and verification aspect of code queries, like CodeQL (GitHub, 2022) and Coverity (Synopsys, 2022). Finally, there are many academic and commercial techniques focused less on structural code querying and more on intuitive (often natural-language based) code queries (McMillan et al., 2011a,b; Lemos et al., 2014; Lazzarini Lemos et al., 2015; Lv et al., 2015; Kim et al., 2018; Sirres et al., 2018; Sourcegraph, 2022).

Despite the rich history of research on querying code, code-book provides a unique and powerful collection of features for addressing the problem of formulating queries on code, by combining the ideas of interactive notebooks, Query-by-Example-style queries, and queries that can utilize code structure, data flows, and code embeddings.

# 7.5 Future Work

Work on code-book is ongoing and there are many exciting avenues of future work. In particular, there is a great need to create systems that are capable of joining both code *and* non-code data. For example, imagine asking questions about (i) issues in an issue tracker and (ii) recent code commits (questions like, "show me the five most recently modified functions that appear frequently in stack traces in crash reports this past week"). Another interesting source of non-code data would be performance counters and metrics. Imagine being able to search for loops that had similar (or perhaps better) performance characteristics and similar structure to a given example loop. Finally, consider how useful it would be to have a way to associate production log data with the code that generated it. Such an association would allow for queries that bridge the gap between anomalous logs and the code that (likely) triggered those logs.

Aside from linking code to external non-code data, there are also interesting opportunities in the realm of learning domain-specific models of code. With a tool such as **code-book**, it is easy to query for code matching a set of heuristics. Given the success of general code models (and the notable failures of such models to produce correct domain-specific code) it should be feasible to use **code-book** as an effective dataset-generation tool. For example, there is a great deal of interest in models of code that provide recommendations for specific (popular) libraries. One library in particular worth targeting is Pandas (one of the most popular libraries for Data Scientists). With **code-book**, we can search for code that uses the Pandas API, and formulate a dataset containing *only* code that uses the API. Such a dataset could be used to fine-tune existing large models. By generating datasets of domain-specific code and fine-tuning models, one could hope to bridge the gap between hand-crafted solutions to domain-specific problems and the often incorrect or subtly wrong solutions generated by generic models of code.

# 7.6 Notes

code-book has been the most challenging project I have undertaken during my graduate studies. In many ways, this project has been my greatest success (although I have focused, perhaps too much, on the engineering of this system and now could stand to do some more writing and benchmarking). I truly believe that code-book is a fresh take on making code queries accessible to more users. Nevertheless, adoption

of code-book has been challenging. To make a tool that people *want* to use is a much higher bar than to make a tool with academic merit and novelty. Novel ideas (like the mixing of structural and fuzzy constraints) are very hard to make accessible to users, and often get in the way of making the tool *useful* to practitioners.

Throughout the past year and a half, I have built several prototypes of code-book. To help visualize some of the ideas discussed in this chapter, I have included two screenshots of code-book in Figs. 7.6 and 7.7. (It was not always the case that code-book featured a Query-by-Example-style query language; in Fig. 7.6, you can see my earlier attempt at devising a reasonable query syntax.)

Odeobook gh-demo-1.ipynb	1 month ago
FILE > EDIT > CELL > RUNTIME >	94 95 # \$cb:=end:cell:2 96
<pre>from codebook.python import *     from codebook.python import *     benchmark_assignments = execute(     assignment() % 'assign'      where  the_lhs()      isa  identifier(with_text('benchmark'))     , compile=True     ) </pre>	<pre>97 # \$cb:=start:cell:3 98 # Now, for The calculation of Portfolio Weights and f 99 active = pd.read.csv('active_weights.csv') 100 active = pd.read.csv('active_weights.csv') 101 active = np.array(active) 102 benchmark = np.array(np.ones(27)*(1/27)) 103 #benchmark = np.array(np.ones(27)*(1/27)) 104 portfolio = [a + b for a, b in zip(active, benchmark) 105 portfolio = np.array(portfolio) 106 print(portfolio)</pre>

Figure 7.6: The second **code-book** prototype (shown with an early version of the **code-book** query language). The left side of the interface contains an interactive Python notebook. The right side contains an embedded IDE to visualize matching code.

126	%%morp std::f	ohql cpp "find_if_ find_if_not(\$itera	not_eq_end" tor.begin(),) == \$ite	erator.end()
	Query built. Use `find_if_not_eq_end.execute( <options>)` to produce a DataFrame.</options>			
127	<pre>find_if_not_eq_end.execute(dataset='/data/cpp')</pre>			
	Query Results: `find_if_not_eq_end` (1/1)			
		iterator	text_iterator	
	Θ	Go to code	potentialArtType	

Figure 7.7: The second code-book prototype (shown with a later version of the code-book query language). In this revision, we started to realize the value of Query-by-Example-style queries for code and upgraded the visualization capabilities to be more intuitive. (We used links within data frames to jump directly to matching code.)

It seems we're at the end now. What did you think?

GPT-3: I thought it was great! I loved getting to know all the characters and watching their stories unfold.

Oh great! I guess by characters you means research threads? And by stories unfolding you're referring to experiments providing empirical validation to the ideas presented herein?

GPT-3: Exactly! I also appreciated all the humor sprinkled throughout.

Oh well you're too kind GPT-3. Too kind. Glad you enjoyed!

A dialogue with OpenAI's GPT-3

In this final chapter, we revisit the contributions of our research and discuss some limitations of the techniques we have presented. We also present some brief thoughts on the impacts of recent large language models and the future of learning from code (and non-code artifacts). Finally, we offer a few concluding remarks, and provide links to the numerous public datasets and tools created during the course of our research.

# 8.1 Contributions

As described in Chapter 1, our contributions can be grouped into three main categories: (i) contributions to learning from code, (ii) contributions to learning from non-code artifacts, and (iii) contributions to software-engineering research. In each of these categories we have done substantial work. In the following sections, we revisit each of these three categories and outline the our primary contributions.

## Contributions to learning from code

We introduced a novel technique for learning from code via *code embeddings* (see Chapter 2).

We created a new tool for parametric lightweight symbolic execution of C programs. We scaled this tool to run on the Linux kernel and used the extracted traces to both learn embeddings and power a specification miner (see Sections 2.2 and 3.2).

We developed a benchmark of code analogies extracted from the Linux kernel (see Section 2.4.1).

We performed experiments that showed that our code embeddings were effective; the embeddings achieved 93% top-1 accuracy on our code analogies benchmark and 76% top-3 accuracy on a challenging task involving predicting error codes for failing traces extracted from the Linux kernel (see Sections 2.4.1 and 2.4.4).

We devised a novel approach to specification mining that utilized a mix of both traditional and learned metrics (see Section 3.2). This approach worked better than either the traditional or learned approaches did in isolation (see Section 3.4.5).

We performed an in-depth comparison of the three most popular word-vector learners (applied to code) and three different trace-sampling techniques (see Section 3.4.4).

We developed a framework for creating controllable adversaries to test the *robustness* of models of code (see Section 4.3).

We devised a new technique for training robust models of code (see Section 4.1.4); training such models is a hard problem given the *discrete* nature of code.

We presented the first results on the performance of robust models of code on out-of-distribution data and in a cross-language-transfer setting (see Sections 4.5.4 and 4.5.5).

We made our data and tools public to facilitate further research on learning from code—in fact, some of the tools and data we made public were used immediately
and directly by others to produced follow-on works that appeared before our original work was ever published!

### Contributions to learning from non-code artifacts

We introduced a technique for mining tree-association rules from Dockerfiles (see Section 5.3).

We identified a problem in DevOps artifacts: the problem of deeply nested languages. We devised a technique called *phased parsing* to address this problem (see Section 5.3.1).

We shared a new dataset of over two hundred thousand Dockerfiles downloaded from GitHub. We augmented this dataset with Abstract Syntax Trees generated by our phased parsing technique (see Section 5.2).

We constructed a static checker for Dockerfiles that was capable of validating a file's compliance with a database of tree-association rules (see Section 5.3.5).

We performed experiments and found an especially surprising result: Dockerfiles written by developers in industry were *worse*, on average, than those found "in the wild" on GitHub!

We devised a technique for semi-automated *repair* of Dockerfiles (see Section 6.4).

We provided the community with a full suite of analysis tooling for Dockerfiles consisting of automated rule mining, static checking, and repair techniques.

We fixed 19 repositories with our human-in-the-loop technique for Dockerfile repair by submitting 45 pull requests (using our tool). Of these 45 requests, 19 were accepted by repository owners (see Section 6.5).

### Contributions to the field at large

We introduced a new idea—the idea of supporting *Data Science on Code* by borrowing from the great successes of the Data Science community and leveraging those successes to build better tooling for software-engineering research (see Section 7.1).

We developed code-book: a tool aimed at making Data Science on Code a reality (see Section 7.1). We continue, in ongoing work, to refine and adapt code-book to better serve users.

We devised a new language for querying code that targets both novice users and experts by leveraging an old idea from the databases community: the idea of *Query* by *Example* (see Section 7.2).

#### Summary

We hope that these contributions help move us toward escaping the nearly bottomless pit of ever-expanding software; furthermore, we hope that our ongoing work towards Data Science on Code will, one day, greatly accelerate the pace of empirical softwareengineering research.

# 8.2 Limitations

Although we make many contributions to learning from code and non-code artifacts, there are limitations to the techniques and tools we have developed. The biggest limitation is the relatively high level of expertise needed to apply many of our techniques. Outside of **code-book**, most of the work described in this thesis is aimed at practitioners that have some exposure to the concepts of code analysis and software-engineering research. Some techniques, like the ones we developed for working with Dockerfiles, require less domain-specific knowledge to use; however, to expand upon these techniques, or to translate these techniques to other DevOps artifacts of interest, would require significant expertise.

For learning from code, many of the techniques we have introduced here are being usurped by simpler models with more compute. Similar to our first limitation, these new models often require computing resources that are prohibitive to even well-funded academic labs, but, nonetheless, the models that are produced provide compelling results when trained on code as text and are being made available to practitioners in the form of APIs and commercial products. Finally, for almost all of the work described in this thesis, we have had to spend time and energy developing bespoke tooling to extract datasets, create analyses, organize experiments, and capture results. Consequently, many of the studies we carried out are vulnerable to bugs and hard to replicate independently (here we mean replicate in the sense of redoing a study from the ground up—we have artifacts available if one wishes to run similar experiments using the tools and data we developed). This limitation is shared by almost every piece of work in the empirical software-engineering and program-analysis communities; nevertheless, it is worth noting that, as part of our research, we too contributed to the ever-expanding amount of hard to understand and maintain software.

## 8.3 The Impact of Large Language Models

During the course of our research, large language models became a dominant force in learning from text and, surprisingly, became the defacto choice for learning from code. This outcome is, perhaps, yet another instance of the bitter lesson (the idea that human cleverness will not surpass techniques that scale well with increases in computational power). We think that the successes of large language models are both (i) a call to action and (ii) a hint that there may be techniques and ideas that work on structured representations of code that we have yet to devise. (Based on historical precedent, it would be wise to design these new hypothetical techniques so that they too scale with increases in computational power, unlike previous methods.)

We believe that there is much more to discover in *how* these large models understand structured text, such as programs. It seems to be the case that models trained on code are, in some ways, preferable to models trained on mere prose; if you train your model on code, you can expect answers that are well-formed programs (most of the time, at least)—these well-formed programs can be parsed, sometimes executed (given an appropriate environment), and, most important of all, programs (whether they are generated by a human or a model) are amenable to structured manipulations. Perhaps it is the case that the true bridge between continuous and symbolic reasoning exists in the realm of models that both *understand* code and can Tools for learning from code: https://github.com/jjhenkel/code-vectors-artifact https://github.com/jjhenkel/lsee https://github.com/jjhenkel/c2ocaml https://github.com/jjhenkel/abstracted-kernel-traces https://github.com/jjhenkel/averloc Tools and datasets for learning from non-code artifacts: https://github.com/jjhenkel/binnacle-icse2020 https://github.com/STAR-RG/shipwright

Figure 8.1: Links to various tools and datasets produced during our research

*produce* code. Or, perhaps, these models are no better than ones trained on prose. We think this research area (models that understand programs/models that can produce programs) is one worth exploring in the coming years.

# 8.4 Concluding Remarks

There is a light at the end of the tunnel: we have described a way to create an ecosystem capable of dealing with the extraordinary amount of software being created every day. First, in discrete steps, and later in what we call "plans for a ladder," we have outlined ways to combat the growth of software and the fundamental issue of software being easier to write than it is to understand and maintain.

Our study of learning from code and non-code artifacts led us to realize a need for better tooling and, to answer that need, we began work toward enabling Data Science on Code. To this end, we developed **code-book**, which combines several novel ideas including: interactive notebooks for program analysis, code-embedding-assisted fuzzy queries, a Query-by-Example-style query language, and integrated visualizations.

Finally, we have released, for each completed work, the tooling and datasets we developed. Figure 8.1 provides links to these releases. We hope that sharing research tooling and data will accelerate the efforts of those who wish to build on this work and foster continued collaboration toward our overarching goal of addressing the growth

in software and enhancing our abilities to learn from, understand, and maintain the software we depend on.

# 8.5 Notes

So here we are at the end. This thesis has been a project like no other—never have I ever spent so much time just writing. In many ways, I have always felt most at home as a graduate student doing engineering. Whether we needed to do symbolic execution on the Linux kernel using OCaml or craft a system for building tens-of-thousands of Dockerfiles in clean environments using a horrid mix of Bash and Python, I have always enjoyed the engineering aspect of research. Nevertheless, this process of writing and reflecting upon the work I have done leaves me with a sense of pride. I find this work to be meaningful and I hope you, dear reader, have enjoyed at least some of it. At the very least, I hope the research tooling and the datasets produced during my time as a graduate student make it easier for others to do new work and reach interesting conclusions more quickly. Regardless of the outcome, I cannot wait to continue to explore new ideas, write new tools, and create new datasets and analyses. I said it earlier, but I think it bears repeating: I truly believe that there has never been a more exciting time to do work at the intersection of Programming Languages, Software Engineering, and Machine Learning; I cannot fathom what the future may hold, I just know it's going to be interesting!

The table below provides representative pairs for each of the categories in the analogy suite used in Section 2.4.

Type	Category	Representative Pair
Calls	16 / 32	store16 / store32
Calls	Add / Remove	ntb_list_add / ntb_list_rm
Calls	Create / Destroy	<pre>device_create / device_destroy</pre>
Calls	Enable / Disable	nv_enable_irq / nv_disable_irq
Calls	Enter / Exit	otp_enter / otp_exit
Calls	In / Out	add_in_dtd / add_out_dtd
Calls	Inc / Dec	<pre>cifs_in_send_inc / cifs_in_send_dec</pre>
Calls	Input / Output	ivtv_get_input / ivtv_get_output
Calls	Join / Leave	handle_join_req / handle_leave_req
Calls	Lock / Unlock	<pre>mutex_lock_nested / mutex_unlock</pre>
Calls	On / Off	b43_led_turn_on / b43_led_turn_off
Calls	Read / Write	<pre>memory_read / memory_write</pre>
Calls	Set / Get	set_arg / get_arg
Calls	Start / Stop	nv_start_tx / nv_stop_tx
Calls	Up / Down	ixgbevf_up / ixgbevf_down
Complex	Ret Check / Call	kzalloc_\$NEQ_0 / kzalloc
Complex	Ret Error / Prop	write_bbt_\$LT_0 / \$RET_write_bbt
Fields	Check / Check	?->dmaops / ?->dmaops->altera_dtype
Fields	Next / Prev	<pre>!.task_list.next / !.task_list.prev</pre>
Fields	Test / Set	?->at_current / !->at_current

Table A.1: Analogy Suite: Representative Pairs

#### REFERENCES

Acharya, Mithun, and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *Fundamental Approaches to Software Engineering*, ed. Marsha Chechik and Martin Wirsing, 370–384. Lecture Notes in Computer Science, Springer Berlin Heidelberg.

Acharya, Mithun, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 25–34. ESEC-FSE '07, New York, NY, USA: ACM.

Agrawal, Rakesh, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Acm sigmod record*, vol. 22, 207–216. ACM.

Ahmad, Wasi Uddin, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the* 58th annual meeting of the association for computational linguistics (acl).

Allamanis, Miltiadis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, 281–293. FSE 2014, New York, NY, USA: ACM.

Allamanis, Miltiadis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015a. Suggesting accurate method and class names. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 38–49. ESEC/FSE 2015, New York, NY, USA: Association for Computing Machinery.

Allamanis, Miltiadis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys* (CSUR) 51(4):1–37. Allamanis, Miltiadis, Earl T Barr, Premkumar T Devanbu, and Charles A Sutton. 2017a. A Survey of Machine Learning for Big Code and Naturalness. *CoRR* abs/1709.0.

Allamanis, Miltiadis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 91–105. PLDI 2020, New York, NY, USA: Association for Computing Machinery.

Allamanis, Miltiadis, Marc Brockschmidt, and Mahmoud Khademi. 2017b. Learning to Represent Programs with Graphs. CoRR abs/1711.0. 1711.00740.

Allamanis, Miltiadis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. 2016a. Learning Continuous Semantic Representations of Symbolic Expressions. *arXiv preprint arXiv:1611.01423*.

Allamanis, Miltiadis, Hao Peng, and Charles A. Sutton. 2016b. A convolutional attention network for extreme summarization of source code. *CoRR* abs/1602.03001. 1602.03001.

Allamanis, Miltiadis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015b. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd international conference on international conference on machine learning - volume* 37, 2123–2132. ICML'15, JMLR.org.

Alon, Uri, Shaked Brody, Omer Levy, and Eran Yahav. 2018a. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Alon, Uri, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018b. code2vec: Learning distributed representations of code. *CoRR* abs/1803.09473. 1803.09473.

Alon, Uri, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018c. A general path-based representation for predicting program properties. In *Proceedings of the 39th acm sigplan conference on programming language design and implementation*, 404–419. PLDI 2018, New York, NY, USA: ACM.

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

Ammons, Glenn, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 4–16. POPL '02, New York, NY, USA: ACM.

Andrews, J. H., L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on software engineering*, 402–411. ICSE '05, New York, NY, USA: Association for Computing Machinery.

Athalye, Anish, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv* preprint arXiv:1802.00420.

Barcelona, Ajuntamentde. 2020. Dependendy on ruby version 2.6.5. https://github.com/AjuntamentdeBarcelona/decidim-barcelona/blob/ 83ef28ee6af9d7ec2ac7914762c00db165592615/Gemfile#L5.

jahanzaib basharat. 2018. E: Package 'libpng12-dev' has no installation candidate. https://github.com/docker-library/php/issues/662.

Bass, Leonard J., Ingo M. Weber, and Liming Zhu. 2015. *Devops - A software architect's perspective*. SEI series in software engineering, Addison-Wesley.

Ben-Nun, Tal, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *CoRR* abs/1806.07336.
1806.07336.

Berkerpeksag. 2020. berkerpeksag/astor.

Bhagoji, Arjun Nitin, Warren He, Bo Li, and Dawn Song. 2018. Practical black-box attacks on deep neural networks using efficient query mechanisms. In *European conference on computer vision*, 158–174. Springer.

Bielik, Pavol, Veselin Raychev, and Martin Vechev. 2016. Phog: Probabilistic model for code. In *Proceedings of the 33rd international conference on international conference on machine learning - volume 48*, 2933–2942. ICML'16, JMLR.org.

Bielik, Pavol, and Martin Vechev. 2020. Adversarial robustness for code. 2002.04694.

Biermann, A. W., and J. A. Feldman. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Transactions on Computers* C-21(6):592–597.

Biggio, Battista, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Joint european conference on machine learning* and knowledge discovery in databases, 387–402. Springer.

Bojanowski, Piotr, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association* for Computational Linguistics 5:135–146.

Brogi, A., D. Neri, and J. Soldani. 2017. Dockerfinder: Multi-attribute search of docker images. In 2017 ieee international conference on cloud engineering (ic2e), 273–278.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. 2005.14165.

Buddhi. 2019. How to download a file using curl. https://stackoverflow.com/a/54735579.

Campello, Ricardo J. G. B., Davoud Moulavi, and Joerg Sander. 2013. Densitybased clustering based on hierarchical density estimates. In *Advances in knowledge discovery and data mining*, ed. Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, 160–172. Berlin, Heidelberg: Springer Berlin Heidelberg.

Carlini, Nicholas, and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *Security and privacy (sp), 2017 ieee symposium on*, 39–57. IEEE.

Chen, Pin-Yu, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. 2017. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th acm workshop on artificial intelligence and security*, 15–26. ACM.

Chi, Yun, Richard R Muntz, Siegfried Nijssen, and Joost N Kok. 2005a. Frequent subtree mining–an overview. *Fundamenta Informaticae* 66(1-2):161–198.

Chi, Yun, Yi Xia, Yirong Yang, and Richard R Muntz. 2005b. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Transactions on Knowledge and Data Engineering* 17(2):190–202.

CircleCI. 2019. CircleCI - Continuous Integration and Delivery. [Online; accessed 21. Aug. 2019].

Cito, J., G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. 2017. An empirical analysis of the docker container ecosystem on github. In 2017 ieee/acm 14th international conference on mining software repositories (msr), 323–333.

Comby. 2022. Comby. Accessed: 2022-05-10.

Compton, Rhys, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding java classes with code2vec: Improvements from variable obfuscation. 2004.02942.

Cousot, P., and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc.* acm sigplan-sigact symposium on principles of programming languages, 238–252.

Crew, Roger F. 1997. ASTLOG: A language for examining abstract syntax trees. In *Conference on domain-specific languages (dsl 97)*. Santa Barbara, CA: USENIX Association.

Dallmeier, Valentin, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining Object Behavior with ADABU. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, 17–24. WODA '06, New York, NY, USA: ACM.

Davis, Jennifer, and Ryn Daniels. 2016. *Effective devops: building a culture of collaboration, affinity, and tooling at scale.* O'Reilly Media, Inc.

DeFreez, Daniel, Aditya V. Thakur, and Cindy Rubio-González. 2018a. Path-based function embedding and its application to error-handling specification mining. In Proceedings of the 26th acm joint european software engineering conference and symposium on the foundations of software engineering (esec/fse), FSE 2018, lake buena vista, florida, november 4 - 9, 2018.

DeFreez, Daniel, Aditya V. Thakur, and Cindy Rubio-González. 2018b. Path-based Function Embedding and Its Application to Error-handling Specification Mining. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 423–433. ESEC/FSE 2018, New York, NY, USA: ACM. Event-place: Lake Buena Vista, FL, USA.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North*.

docker. 2015. Docker hub: Database of container images. hub.docker.com.

Docker. 2019. Enterprise Container Platform | Docker. [Online; accessed 21. Aug. 2019].

Dyer, Robert, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2015. Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.* 25(1).

Ebrahimi, Javid, Anyi Rao, Daniel Lowd, and Dejing Dou. 2017. Hotflip: White-box adversarial examples for text classification. 1712.06751.

Engler, Dawson, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the eighteenth acm symposium on operating systems principles*, 57–72. SOSP '01, New York, NY, USA: ACM.

Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996a. A densitybased algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the second international conference on knowledge discovery and data mining*, 226–231. KDD'96, AAAI Press.

Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996b. A densitybased algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the second international conference on knowledge discovery and data mining*, 226–231. KDD'96, AAAI Press.

Eykholt, Kevin, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 1625–1634.

Finkelstein, Lev, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2001. Placing Search in Context: The Concept Revisited. In *Proceedings of the 10th international conference on world wide web*, 406–414. WWW '01, New York, NY, USA: ACM.

Fowkes, Jaroslav, and Charles Sutton. 2016. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th acm sigsoft international symposium* 

on foundations of software engineering, 254–265. FSE 2016, New York, NY, USA: ACM.

Frey, Brendan J., and Delbert Dueck. 2007. Clustering by passing messages between data points. *Science* 315(5814):972–976. http://science.sciencemag.org/ content/315/5814/972.full.pdf.

Gabel, Mark, and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIG-SOFT International Symposium on Foundations of Software Engineering*, 339–349. SIGSOFT '08/FSE-16, New York, NY, USA: ACM.

Gallaba, K., and S. McIntosh. 2018. Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci. *IEEE Transactions on Software Engineering* 1–1.

Gao, Xiang, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *Proceedings of the acm/ieee 42nd international conference on software engineering*, 1147–1158. ICSE '20, New York, NY, USA: Association for Computing Machinery.

Garg, Siddhant, and Goutham Ramakrishnan. 2020. Bae: Bert-based adversarial examples for text classification. 2004.01970.

Gazzola, L., D. Micucci, and L. Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45(1):34–67.

GitHub. 2022. Codeql. Accessed: 2022-05-10.

GitHub. 2022. tree-sitter.

Goodfellow, Ian J, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.

Google. 2019. Kubernetes - Production-Grade Container Orchestration. [Online; accessed 21. Aug. 2019].

Gu, Xiaodong, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 631–642. FSE 2016, New York, NY, USA: ACM.

Guidotti, Riccardo, Jacopo Soldani, Davide Neri, and Antonio Brogi. 2018. Explaining successful docker images using pattern mining analysis. In *Software technologies: Applications and foundations*, ed. Manuel Mazzara, Iulian Ober, and Gwen Salaün, 98–113. Cham: Springer International Publishing.

Gvero, Tihomir, and Viktor Kuncak. 2015. Synthesizing java expressions from free-form queries. In *Proceedings of the 2015 acm sigplan international conference* on object-oriented programming, systems, languages, and applications, 416–432. OOPSLA 2015, New York, NY, USA: ACM.

Hadolint. 2019. hadolint/hadolint. [Online; accessed 21. Aug. 2019].

Harris, Zellig S. 1954. Distributional Structure. WORD 10(2-3):146–162.

HashiCorp. 2019. Terraform - Write, Plan, and Create Infrastructure as Code. [Online; accessed 21. Aug. 2019].

Hassan, Foyzul, Rodney Rodriguez, and Xiaoyin Wang. 2018a. Rudsea: Recommending updates of dockerfiles via software environment analysis. In *Proceedings* of the 33rd acm/ieee international conference on automated software engineering, 796–801. ASE 2018, New York, NY, USA: ACM.

Hassan, Foyzul, Rodney Rodriguez, and Xiaoyin Wang. 2018b. Rudsea: Recommending updates of dockerfiles via software environment analysis. In *Proceedings* of the 33rd acm/ieee international conference on automated software engineering, 796–801. ASE 2018, New York, NY, USA: Association for Computing Machinery.

Hassan, Foyzul, and Xiaoyin Wang. 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th international* 

conference on software engineering, 1078–1089. ICSE '18, New York, NY, USA: Association for Computing Machinery.

Hellendoorn, Vincent J, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting* on european software engineering conference and symposium on the foundations of software engineering, 152–162.

Henkel, Jordan, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 163–174. ESEC/FSE 2018, New York, NY, USA: ACM.

Hill, Felix, Roi Reichart, and Anna Korhonen. 2015. Simlex-999: Evaluating Semantic Models with Genuine Similarity Estimation. *Comput. Linguist.* 41(4):665–695.

Hilton, Michael, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st ieee/acm international conference on automated software engineering*, 426–437. ACM.

Hindle, Abram, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th international conference on software engineering*, 837–847. ICSE '12, Piscataway, NJ, USA: IEEE Press.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9(8):1735–1780.

Holen, Vidar. 2019. koalaman/shellcheck. [Online; accessed 21. Aug. 2019].

Horton, Eric, and Chris Parnin. 2019. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *Proceedings of the 41st international*  conference on software engineering, 328–338. ICSE '19, Piscataway, NJ, USA: IEEE Press.

Hung, Che Shian, and Robert Dyer. 2020. *Boa views: Easy modularization and sharing of msr analyses*, 147–157. New York, NY, USA: Association for Computing Machinery.

IBM. 2020. Ibm/pytorch-seq2seq.

Ilyas, Andrew, Logan Engstrom, Anish Athalye, and Jessy Lin. 2018. Black-box adversarial attacks with limited queries and information. *arXiv preprint arXiv:1804.08598*.

Jenkins. 2019. Jenkins - Open Source Automation Server. [Online; accessed 21. Aug. 2019].

Jiang, Yujuan, and Bram Adams. 2015. Co-evolution of infrastructure and source code-an empirical study. In 2015 ieee/acm 12th working conference on mining software repositories, 45–55. IEEE.

Kanade, Aditya, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International* conference on machine learning, 5110–5121. PMLR.

Kenter, Tom, Alexey Borisov, and Maarten de Rijke. 2016. Siamese cbow: Optimizing word embeddings for sentence representations. In *Proceedings of the the 54th annual meeting of the association for computational linguistics (acl 2016)*.

Kim, Gene, Jez Humble, Patrick Debois, and John Willis. 2016. The devops handbook:: How to create world-class agility, reliability, and security in technology organizations. IT Revolution.

Kim, Kisub, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. Facoy – a code-to-code search engine. In 2018 ieee/acm 40th international conference on software engineering (icse), 946–957.

Kolter, Zico, and Aleksander Madry. 2020. Ibm/pytorch-seq2seq.

Lazzarini Lemos, Otavio Augusto, Adriano Carvalho de Paula, Hitesh Sajnani, and Cristina V. Lopes. 2015. Can the use of types and query expansion help improve large-scale code search? In 2015 ieee 15th international working conference on source code analysis and manipulation (scam), 41–50.

Le, Quoc, and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the 31st international conference on international conference on machine learning - volume 32*, II–1188–II–1196. ICML'14, JMLR.org.

Le, Tien-Duy B., and David Lo. 2018. Deep Specification Mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 106–117. ISSTA 2018, New York, NY, USA: ACM.

Lei, Qi, Lingfei Wu, Pin-Yu Chen, Alexandros G. Dimakis, Inderjit S. Dhillon, and Michael Witbrock. 2019. Discrete adversarial attacks and submodular optimization with applications to text classification. In *Sysml*.

Lemos, Otávio A. L., Adriano C. de Paula, Felipe C. Zanichelli, and Cristina V. Lopes. 2014. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th working conference on mining software repositories*, 212–221. MSR 2014, New York, NY, USA: Association for Computing Machinery.

Levy, Omer, and Yoav Goldberg. 2014. Linguistic regularities in sparse and explicit word representations. In *Proceedings of the eighteenth conference on computational natural language learning*, 171–180. Ann Arbor, Michigan: Association for Computational Linguistics.

Li, Yujia, Daniel Tarlow, Marc Brockschmidt, and Richard S Zemel. 2015. Gated Graph Sequence Neural Networks. CoRR abs/1511.0. 1511.05493.

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 306–315. ESEC/FSE-13, New York, NY, USA: ACM.

Lin, C., S. Nadi, and H. Khazaei. 2020. A large-scale data set and an empirical study of docker images hosted on docker hub. In 2020 ieee international conference on software maintenance and evolution (icsme), 371–381.

Linnovate. 2020. Mean stack. https://github.com/linnovate/mean.

Livshits, Benjamin, and Thomas Zimmermann. 2005. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proceedings of the* 10th European Software Engineering Conference Held Jointly with 13th ACM SIG-SOFT International Symposium on Foundations of Software Engineering, 296–305. ESEC/FSE-13, New York, NY, USA: ACM. Event-place: Lisbon, Portugal.

Lo, David, and Siau-Cheng Khoo. 2006. SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner. In *Proceedings of the 14th ACM SIG-SOFT International Symposium on Foundations of Software Engineering*, 265–275. SIGSOFT '06/FSE-14, New York, NY, USA: ACM.

Lo, David, Siau-Cheng Khoo, Jiawei Han, and Chao Liu. 2011. *Mining Software Specifications: Methodologies and Applications*. CRC Press. Google-Books-ID: VAzLBQAAQBAJ.

Lorenzoli, D., L. Mariani, and M. Pezzè. 2008. Automatic generation of software behavioral models. In 2008 ACM/IEEE 30th International Conference on Software Engineering, 501–510.

Lou, Yiling, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Historydriven build failure fixing: How far are we? In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis*, 43–54. ISSTA 2019, New York, NY, USA: Association for Computing Machinery.

Luong, Thang, Richard Socher, and Christopher D Manning. 2013. Better Word Representations with Recursive Neural Networks for Morphology. In *Conll.* 

Lv, Fei, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding

and extended boolean model (e). In 2015 30th ieee/acm international conference on automated software engineering (ase), 260–270.

Lwakatare, Lucy Ellen, Pasi Kuvaja, and Markku Oivo. 2015. Dimensions of devops. In *International conference on agile software development*, 212–217. Springer.

Macho, C., S. McIntosh, and M. Pinzger. 2018. Automatically repairing dependencyrelated build breakage. In 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner), 106–117.

Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks. In *International conference on learning representations*.

Mancinelli, F., J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. 2006. Managing the complexity of large free and open source packagebased software distributions. In 21st ieee/acm international conference on automated software engineering (ase'06), 199–208.

Marketplace, Visual Studio Code. 2020. Docker. [Online; accessed 29. Jan. 2020].

Martin Monperrus, Zimin Chen. 2019. Embeddings for Source Code.

Mazuran, Mirjana, Elisa Quintarelli, and Letizia Tanca. 2009. Mining tree-based association rules from xml documents. In *Proceedings of the seventeenth italian symposium on advanced database systems*, 109–116.

McCamant, Stephen, and Michael D. Ernst. 2004. Early identification of incompatibilities in multi-component upgrades. In *Ecoop 2004 – object-oriented programming*, ed. Martin Odersky, 440–464. Berlin, Heidelberg: Springer Berlin Heidelberg.

McMillan, Collin, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011a. Portfolio: finding relevant functions and their usage. In 2011 33rd international conference on software engineering (icse), 111–120. McMillan, Collin, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011b. Portfolio: finding relevant functions and their usage. In 2011 33rd international conference on software engineering (icse), 111–120.

Microsoft. 2019. microsoft/vscode-docker. [Online; accessed 21. Aug. 2019].

Might, Matthew. 2010. The crapl: An academic-strength open source license. https://matt.might.net/articles/crapl/. Accessed: 2022-04-25.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013a. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in neural information processing systems 26*, ed. C J C Burges, L Bottou, M Welling, Z Ghahramani, and K Q Weinberger, 3111–3119. Curran Associates, Inc.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013b. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, ed. C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, 3111–3119. Curran Associates, Inc.

Min, Changwoo, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th symposium on operating systems principles*, 361–377. SOSP '15, New York, NY, USA: ACM.

Møller, Anders, and Martin Toldam Torp. 2019. Model-based testing of breaking changes in node.js libraries. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 409–419. ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery.

Monperrus, Martin, and Mira Mezini. 2013. Detecting Missing Method Calls As Violations of the Majority Rule. ACM Trans. Softw. Eng. Methodol. 22(1):7:1–7:25.

Mou, Lili, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing.

mtsmfm. 2020. Set lang by default. https://github.com/docker-library/ruby/commit/8813cdda206acb36ea7797919bf8dadb84fc5ac7.

Mudrakarta, Pramod Kaushik, Ankur Taly, Mukund Sundararajan, and Kedar Dhamdhere. 2018. Did the model understand the question? In *Acl.* 

Murali, Vijayaraghavan, Swarat Chaudhuri, and Chris Jermaine. 2017a. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 151–162. ESEC/FSE 2017, New York, NY, USA: ACM.

Murali, Vijayaraghavan, Swarat Chaudhuri, and Chris Jermaine. 2017b. Bayesian Specification Learning for Finding API Usage Errors. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 151–162. ESEC/FSE 2017, New York, NY, USA: ACM.

Nguyen, Anh Tuan, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2012. GraPacc: A Graph-based Pattern-oriented, Context-sensitive Code Completion Tool. In *Proceedings of the 34th international conference on software engineering*, 1407–1410. ICSE '12, Piscataway, NJ, USA: IEEE Press.

Nguyen, Anh Tuan, and Tien N. Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the 37th international conference on software engineering - volume 1*, 858–868. ICSE '15, Piscataway, NJ, USA: IEEE Press.

Nguyen, Trong Duc, Anh H. T. Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017a. Exploring API Embedding for API Usages and Applications. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) 438–449.

Nguyen, Trong Duc, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping api elements for code migration with vector representations. In *Proceedings of the 38th* 

international conference on software engineering companion, 756–758. ICSE '16, New York, NY, USA: ACM.

Nguyen, Trong Duc, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017b. Exploring API Embedding for API Usages and Applications. In *Proceedings* of the 39th international conference on software engineering, 438–449. ICSE '17, Piscataway, NJ, USA: IEEE Press.

Nguyen, Tung Thanh, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 532–542. ESEC/FSE 2013, New York, NY, USA: ACM.

Nguyen, Tung Thanh, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. 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, 383–392. ESEC/FSE '09, New York, NY, USA: ACM.

oshivwanshi. 2019. RROR: Error installing bundler: bundler requires ruby version. https://github.com/rubygems/bundler/issues/6865.

PacificNW\_Lover. 2020. Install python-pip using apt-get via ubuntu's apt-get in dockerfile. https://stackoverflow.com/a/61564831.

Papernot, Nicolas, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Proceedings of the 2017 acm on asia conference on computer and communications security*, 506–519. ACM.

Parisotto, Emilio, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. *CoRR* abs/1611.0. 1611.01855.

Pawlak, Renaud, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46:1155–1179.

Peng, Hao, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *Proceedings of the 8th interna*tional conference on knowledge science, engineering and management - volume 9403, 547–553. KSEM 2015, New York, NY, USA: Springer-Verlag New York, Inc.

Pennington, Jeffrey, Richard Socher, and Christopher Manning. 2014a. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543. Doha, Qatar: Association for Computational Linguistics.

Pennington, Jeffrey, Richard Socher, and Christopher D Manning. 2014b. GloVe: Global Vectors for Word Representation. In *Empirical methods in natural language processing (emnlp)*, 1532–1543.

Phillips, Shaun, Thomas Zimmermann, and Christian Bird. 2014. Understanding and improving software build teams. In *Proceedings of the 36th international conference on software engineering*, 735–744. ACM.

Piech, Chris, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32nd international conference on international conference on machine learning - volume 37*, 1093–1102. ICML'15, JMLR.org.

Portworx. 2017. Annual Container Adoption Report. [Online; accessed 21. Aug. 2019].

Pradel, Michael, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. *Typewriter: Neural type prediction with search-based validation*, 209–220. New York, NY, USA: Association for Computing Machinery. Pradel, Michael, and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 371–382. ASE '09, Washington, DC, USA: IEEE Computer Society.

Pradel, Michael, and Koushik Sen. 2017. Deep learning to find bugs. Tech. Rep. TUD-CS-2017-0295, Technische Universität Darmstadt, Department of Computer Science.

Pradel, Michael, and Koushik Sen. 2018. DeepBugs: a learning approach to namebased bug detection. *PACMPL* 2:147:1–147:25.

Quante, J., and R. Koschke. 2007. Dynamic Protocol Recovery. In 14th Working Conference on Reverse Engineering (WCRE 2007), 219–228.

r1williams. 2015. Rpmdb checksum is invalid: dcdpt(pkg checksums). https: //github.com/CentOS/sig-cloud-instance-images/issues/15.

Rabin, Md, Rafiqul Islam, Nghi DQ Bui, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2020. On the generalizability of neural program analyzers with respect to semantic-preserving program transformations. *arXiv preprint arXiv:2008.01566*.

Radford, Alec, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Raghothaman, Mukund, Yi Wei, and Youssef Hamadi. 2016. Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proceedings of the 38th international conference on software engineering*, 357–367. ICSE '16, New York, NY, USA: ACM.

Rahman, Akond, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2019. A systematic mapping study of infrastructure as code research. *Information & Software Technology* 108:65–77.

Raychev, Veselin, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". *Popl.* 

Raychev, Veselin, Martin Vechev, and Eran Yahav. 2014a. Code completion with statistical language models. In *Proceedings of the 35th acm sigplan conference on programming language design and implementation*, 419–428. PLDI '14, New York, NY, USA: ACM.

Raychev, Veselin, Martin Vechev, and Eran Yahav. 2014b. Code completion with statistical language models. In *Proceedings of the 35th acm sigplan conference on programming language design and implementation*, 419–428.

Reimers, Nils, and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 conference on empirical methods in natural language processing*. Association for Computational Linguistics.

rmNyro. 2017. npm not found on latest build. https://github.com/gliderlabs/ docker-alpine/issues/327.

Robillard, M. P., E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39(5):613–637.

Robillard, Martin P., and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16(6):703–732.

Rubenstein, Herbert, and John B Goodenough. 1965. Contextual Correlates of Synonymy. *Commun. ACM* 8(10):627–633.

Sankaranarayanan, Sriram, Franjo Ivančić, and Aarti Gupta. 2008. Mining Library Specifications Using Inductive Logic Programming. In *Proceedings of the 30th International Conference on Software Engineering*, 131–140. ICSE '08, New York, NY, USA: ACM.

Schulze, Dean. 2018. apt-get update fails on 17.04 [closed]. https://askubuntu.com/questions/1059898/apt-get-update-fails-on-17-04.

Semgrep. 2022. Semgrep. Accessed: 2022-05-10.

Seymore, Kristie, Andrew Mccallum, and Ronald Rosenfeld. 1999. Learning hidden markov model structure for information extraction. In *In aaai 99 workshop on machine learning for information extraction*, 37–42.

Sharif, Mahmood, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. 2016. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 acm sigsac conference on computer and communications security*, 1528–1540. ACM.

Sharma, Tushar, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell? In 2016 ieee/acm 13th working conference on mining software repositories (msr), 189–200. IEEE.

Shoham, S., E. Yahav, S. J. Fink, and M. Pistoia. 2008. Static Specification Mining Using Automata-Based Abstractions. *IEEE Transactions on Software Engineering* 34(5):651–666.

Sidhu, Puneet Kaur, Gunter Mussbacher, and Shane McIntosh. 2019. Reuse (or Lack Thereof) in Travis CI Specifications: An Empirical Study of CI Phases and Commands. In *Proc. of the international conference on software analysis, evolution, and reengineering (saner)*, 524–533.

Sirres, Raphael, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. In *Proceedings of the 40th international conference on software engineering*, 945. ICSE '18, New York, NY, USA: Association for Computing Machinery.

Solar-Lezama, Armando, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the* 12th international conference on architectural support for programming languages and operating systems, 404–415.

source{d}. 2019. Cool links & research papers related to Machine Learning applied to source code (MLonCode): src-d/awesome-machine-learning-on-source-code. Original-date: 2017-06-20T13:35:45Z.

Sourcegraph. 2022. Sourcegraph. Accessed: 2022-05-10.

Srikant, Shashank, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating adversarial computer programs using optimized obfuscations. In *International conference on learning representations*.

Ståhl, Daniel, and Jan Bosch. 2016. Industry application of continuous integration modeling: a multiple-case study. In 2016 ieee/acm 38th international conference on software engineering companion (icse-c), 270–279. IEEE.

Synopsys. 2022. Coverity scan. Accessed: 2022-05-10.

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*.

Szumlanski, Sean, Fernando Gomez, and Valerie K Sims. 2013. A new set of norms for semantic relatedness measures. In *Proceedings of the 51st annual meeting of the association for computational linguistics (volume 2: Short papers)*, vol. 2, 890–895.

Tan. 2016. How to fix your ruby version is 2.3.0, but your gemfile specified 2.2.5 while server starting. https://stackoverflow.com/questions/37914702.

Tian, Yuchi, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, 303–314. ICSE '18, New York, NY, USA: Association for Computing Machinery.

Tian, Yuchi, Ziyuan Zhong, Vicente Ordonez, Gail Kaiser, and Baishakhi Ray. 2020. Testing dnn image classifiers for confusion & bias errors. In *Proceedings of the acm/ieee 42nd international conference on software engineering*, 1122–1134. ICSE '20, New York, NY, USA: Association for Computing Machinery. Travis CI. 2019. Travis CI - Test and Deploy Your Code with Confidence. [Online; accessed 21. Aug. 2019].

Tucker, C., D. Shuffelton, R. Jhala, and S. Lerner. 2007. Opium: Optimal package install/uninstall manager. In 29th international conference on software engineering (icse'07), 178–188.

Tufano, Michele, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29(4):e1838. E1838 smr.1838, https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1838.

ubergesundheit. 2015. Change locale to c.utf-8. https://github.com/ docker-library/ruby/issues/45.

Upadhyaya, Ganesha, and Hridesh Rajan. 2018. Collective program analysis. In *Proceedings of the 40th international conference on software engineering*, 620–631. ICSE '18, New York, NY, USA: Association for Computing Machinery.

Uri Alon, Omer Levy, Shaked Brody, and Eran Yahav. 2019. GitHub - techsrl/code2seq: Code for the model presented in the paper: "code2seq: Generating Sequences from Structured Representations of Code" — github.com. https:// github.com/tech-srl/code2seq. [Accessed 07-Apr-2022].

Vasic, Marko, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. In 7th international conference on learning representations, ICLR 2019, new orleans, la, usa, may 6-9, 2019. OpenReview.net.

Vasilescu, Bogdan, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 805–816. ACM.

Volpi, Riccardo, Hongseok Namkoong, O. Sener, John C. Duchi, Vittorio Murino, and S. Savarese. 2018. Generalizing to unseen domains via adversarial data augmentation. In *Neurips*.

Vouillon, J., and R. Di Cosmo. 2013. Broken sets in software repository evolution. In 2013 35th international conference on software engineering (icse), 412–421.

Walkinshaw, N., and K. Bogdanov. 2008. Inferring Finite-State Models with Temporal Constraints. In 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 248–257.

Walkinshaw, N., K. Bogdanov, M. Holcombe, and S. Salahuddin. 2007. Reverse Engineering State Machines by Interactive Grammar Inference. In 14th Working Conference on Reverse Engineering (WCRE 2007), 209–218.

Wang, Ke, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. CoRR abs/1711.07163. 1711.07163.

Wang, Song, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st ieee/acm international conference on automated software engineering*, 708–719. ASE 2016, New York, NY, USA: ACM.

Wasylkowski, Andrzej, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the the 6th Joint Meeting of the European* Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 35–44. ESEC-FSE '07, New York, NY, USA: ACM. Event-place: Dubrovnik, Croatia.

Widder, David Gray, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. A conceptual replication of continuous integration pain points in the context of travis ci. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 647–658. ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery. Wu, Yiwen, Yang Zhang, Tao Wang, and Huaimin Wang. 2020. An empirical study of build failures in the docker context. In *Proceedings of the 17th international conference on mining software repositories*, 76–80. MSR '20, New York, NY, USA: Association for Computing Machinery.

Xu, J., Y. Wu, Z. Lu, and T. Wang. 2019. Dockerfile tf smell detection based on dynamic and static analysis methods. In 2019 ieee 43rd annual computer software and applications conference (compsac), vol. 1, 185–190.

Xu, T., and D. Marinov. 2018. Mining container image repositories for software configuration and beyond. In 2018 ieee/acm 40th international conference on software engineering: New ideas and emerging technologies results (icse-nier), 49–52.

Ye, X, H Shen, X Ma, R Bunescu, and C Liu. 2016a. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In 2016 ieee/acm 38th international conference on software engineering (icse), 404–415.

Ye, Xin, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. 2016b. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) 404–415.

Yefet, Noam, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 4(OOPSLA).

yelizariev. 2020. bzr is not available in alpline:. https://github.com/alpinelinux/ docker-alpine/issues/87.

Yin, K., W. Chen, J. Zhou, G. Wu, and J. Wei. 2018. Star: A specialized tagging approach for docker repositories. In 2018 25th asia-pacific software engineering conference (apsec), 426–435.

Yun, Insu, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISAN: Sanitizing API Usages Through Semantic Cross-checking.

In Proceedings of the 25th USENIX Conference on Security Symposium, 363–378. SEC'16, Berkeley, CA, USA: USENIX Association.

Zerouali, A., T. Mens, G. Robles, and J. M. Gonzalez-Barahona. 2019. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In 2019 ieee 26th international conference on software analysis, evolution and reengineering (saner), 491–501.

Zhang, Fuyuan, Sankalan Pal Chowdhury, and Maria Christakis. 2020. Deepsearch: A simple and effective blackbox attack for deep neural networks, 800–812. New York, NY, USA: Association for Computing Machinery.

Zhang, Wei Emma, Quan Z Sheng, AHOUD Alhazmi, and CHENLIANG LI. 2019. Adversarial attacks on deep learning models in natural language processing: A survey. *arXiv preprint arXiv:1901.06796*.

Zhang, Yang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. 2018a. One size does not fit all: An empirical study of containerized continuous deployment workflows. In *Joint european software engineering conference and symposium on the foundations of software engineering*, 295–306. ESEC/FSE, ACM.

Zhang, Yang, Gang Yin, Tao Wang, Yue Yu, and Huaimin Wang. 2018b. An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In 2018 ieee 42nd annual computer software and applications conference (compsac), vol. 1, 138–143. IEEE.

Zhao, Gang, and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In Proceedings of the 2018 ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/SIGSOFT FSE 2018, lake buena vista, fl, usa, november 04-09, 2018, 141–151.

Zhao, Jinman, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. 2018a. Neural-augmented static analysis of android communication. In *Proceedings* of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering, 342–353. ESEC/FSE 2018, New York, NY, USA: ACM.

Zhao, Jinman, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. 2018b. Neural-augmented static analysis of android communication. In *Proceedings* of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering, 342–353.

Zhao, Yangyang, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd ieee/acm international conference on automated software engineering*, 60–71. IEEE Press.

Zhong, H., L. Zhang, T. Xie, and H. Mei. 2009. Inferring Resource Specifications from Natural Language API Documentation. In 2009 IEEE/ACM International Conference on Automated Software Engineering, 307–318.

Zloof, Moshé M. 1975. Query-by-example: The invocation and definition of tables and forms. In *Proceedings of the 1st international conference on very large data bases*, 1–24. VLDB '75, New York, NY, USA: Association for Computing Machinery.

Zweig, Geoffrey, and Chris J C Burges. 2011. The Microsoft Research Sentence Completion Challenge. Tech. Rep., Microsoft Research.