Understanding Computer Programs: Computational and Cognitive Perspectives

by

Shashank Srikant

B. Tech., National Institute of Technology Kurukshetra (2011)
S.M., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by:	Shashank Srikant Department of Electrical Engineering and Computer Science May 15, 2023
Certified by:	Una-May O'Reilly Principal Research Scientist of the Computer Science and Artificial Intelligence Laboratory Thesis Supervisor
Accepted by:	Leslie Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Understanding Computer Programs:
Computational and Cognitive Perspectives

by

Shashank Srikant

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis, I study the understanding of computer programs (code) from two perspectives: *computational* and *cognitive*. I ask what the human bases of understanding code are, and attempt to determine whether computational models trained on code corpora (also known as code models) share similar bases.

From the computational perspective, I start by proposing a framework to test the robustness of the information learned by code models (chapter 2). This establishes a baseline measure for how well models comprehend code. I then describe techniques for improving the robustness of these models while retaining their accuracy (chapter 3). I then propose a way forward for code models to learn and reason about concurrent programs from their execution traces (chapter 4). In doing so, I also demonstrate the limitations of heuristics developed over the past four decades for detecting data races in concurrent programs, highlighting the need for evaluating these heuristics further.

In the cognitive aspect, I study how our brains comprehend code using fMRI to analyze programmers' brains (chapter 5). I show that our brains encode information about comprehended code similar to how code models encode that information (chapter 6). I show how the framework I develop in chapter 2 can be used to automatically generate stimuli for experiments in psycholinguistics and cognitive neuroscience (chapter 7), which can improve our understanding of how our minds and brains comprehend programs. Finally, I propose a probabilistic framework which models the mechanism of finding *important* parts of a program when comprehending it (chapter 8).

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist of the Computer Science and Artificial Intelligence Laboratory

Acknowledgments

The research presented in this thesis was done under the mentorship of Dr. Una-May O'Reilly. In Una-May, I found an advisor who cared deeply about, and shared a fascination for, programs and understanding them. She showed great confidence in me when I wanted to address the idea of program understanding from diverse perspectives–ML, cognitive neuroscience, and program analysis. Her enthusiasm in understanding and refining with me the questions I investigate in this thesis has been infectious and inspiring. Thank you Una-May for being the inspiring mentor and human being you are. All of the research presented in this thesis was conducted in collaboration with her.

A significant portion of my work in cognitive neuroscience and code model representations that I present in this thesis was done in collaboration with the labs of professors Ev Fedorenko and Sijia Liu. A cold-email to Ev to study the brain bases of code understanding in 2018 and a joint-proposal to explore the robustness of code models with Sijia in 2019 started my collaboration. Their relentless pursuit of their topics of expertise, their availability to engage with me on various (often half-baked) ideas I presented over the years, and their warmth and kindness in readily accepting me as a collaborator is something I am grateful for, and that I hopefully can emulate.

I thank Ev, Sijia, and Prof. Armando Solar-Lezama for the helpful feedback they provided as my thesis readers. Armando's course on program analysis was not only my first course at MIT but also the most influential, as it helped me appreciate the program analysis perspective to some of the solutions presented in this thesis. Thank you Amanda Abrams for patiently working with me on all my thesis-related administrative work, and thanks Nicole Hoffman, Janet Fischer, Alicia Duarte, and Prof. Leslie Kolodziejski, Prof. Berthold Horn for your diligence and care in making the administrative processes a breeze to navigate through.

I had the pleasure of working with, and I am thankful to, the following excellent collaborators who have contributed to the work I present in this thesis. I also mention in brief the *origin stories* for each of these collaborations.

• **Chapter 2** was in collaboration with Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, and Gaoyuan Zhang. Thank you David Cox for the helpful discussions. The chapter, in full, is a reprint of the material as it appears in Srikant et al. [2021], published at ICLR 2021.
The work was a result of a joint-proposal in 2020 between Sijia and ALFA, and which began my collaboration with Sijia's group.

• **Chapter 3** was in collaboration with Jinghan Jia, Sijia Liu, Tamara Mitrovska, and Chuang Gan. Jinghan contributed equally with me as a primary author, who also diligently handled the fairly involved implementation details. The chapter, in full, is a reprint of the material as it appears in Jia et al. [2022], published at SANER 2023.

• **Chapter 4.** was in collaboration with Teodor Rares Begu, Malavika Samak, and Michael Wang. Specifically, Section 4.3 in the chapter refers to a thesis I mentored, authored by Teodor Rares Begu [Rares Begu, 2020]. Section 4.4, in full, is a re-print of the material as it appears in Wang et al. [2023] published at SOAP 2023 at PLDI. I was inspired to study the problem of concurrency bug detection after having unsuccessfully addressed it in my S.M. thesis [Srikant, 2020]. As a side project, I proposed using a toy language to simulate concurrent threads and assess the limitations of various ML models. Teodor led the project, refining and implementing it in 2020. Soon after, my work on min-max optimization informed the theoretical formulation I propose in this chapter. The problem surfaced again when I reached out to Malavika Samak in December 2021, during a talk by Prof. Michael Pradel, to learn more about her work. She too had coincidentally considered using a data-driven approach to reason about concurrent programs. We then scoped the problem, and realized the lack of any labeled datasets. Michael Wang, who joined the ALFA group as a graduate student then, joined this project, took charge, and developed the solution which came to be *RaceInjector*.

• **Chapter 5** was in collaboration with Anna Ivanova, Yotaro Sueoka, Hope Kean, Riva Dhamala, Marina Bers, and Ev Fedorenko. Steve Shannon and Atsushi Takahashi provided valuable support at the Martinos imaging center. This chapter is, in full, a reprint of the arXiv report Srikant et al. [2023a], which in turn is a rewrite of Ivanova

et al. [2020], published at eLife 2020. The chapter informs the results of the eLife work to a computer science audience; the original eLife work was written to primarily inform a cognitive neuroscience audience.

A cold-email to Ev in September 2018 began this collaboration. Anna from Ev lab had also serendipitously begun working on understanding the brain bases of code comprehension in the week I emailed Ev. The neuroimaging perspective of program understanding intrigued me from my days at Aspiring Minds. I even had conversations about fMRI recordings with faculty in India before joining graduate school (see Chapter 1 for an excerpt). Serendipity ensured I went through with my desire to study this problem.

- **Chapter 6** was in collaboration with Benjamin Lipkin, Anna Ivanova and Ev Fedorenko. Ben Lipkin contributed equally with me as a primary author in this work; he played a central role in carefully analyzing all the data, which eventually ensured the success of this project. The chapter, in full, is a reprint of the material as it appears in Srikant et al. [2022], published at NeurIPS 2022.
- **Chapter 7** was done with Greta Tuckute from Ev lab. It is, in full, a reprint of the material as it appears in Srikant et al. [2023b], in submission at the time of writing this thesis.

During a casual dinner conversation in December 2021, Greta introduced me to the problem of stimuli generation, for which she was exploring alternate solutions at the time. I repurposed the solution I present in chapter 2 to address the problem.

- **Chapter 8.** The push to explore behavioral responses to code comprehension came from a growing sense of frustration. Around 2020, I grew jaded by the constant stream of code models applied to software tasks willy-nilly. It seemed then that models were being trained for the sake of it, without taking into account if programmers cared for those tasks, and worse, likely ignoring more fundamental tasks that programmers needed help with, such as reading others' code. I decided to seek out startups and other software product research groups (as opposed to the typical large research groups) in this field which could offer me exposure and access to programmer behavior in the software development process. I cold-emailed Stephen Magill in 2021, co-founder of

Muse Dev Inc (now acquired by Sonatype Inc.), in whose group I spent a summer and tested some ideas on code comprehensibility prediction. I later developed other ideas on this theme which came to be the work I present in this chapter. Ev later introduced me to Prof. Yevgeni Berzak who was working on similar ideas. Thanks to Prof. Daniel Jackson for sharing with me his thoughts on program understandability, and David Darais for informing me of Stephen Magill's group.

- **Unpublished work.** The following works were also related to the theme explored in this thesis. With Tamara Mitrovska, then an undergraduate researcher, I pursued some more challenging directions on probing code models for their understanding. With Erik Hemberg, I proposed a way to use satisfiability modulo theory (SMT) to detect loopholes that aid tax avoidance in contract law. We demonstrated success on a small class of contract laws. This work demonstrates how symbolic approaches can enhance machine comprehension of domain-specific languages, such as contract law, which are generally considered inscrutable. With Stephen Magill at Sonatype Inc., I analyzed $\sim$ 100K Java programs in production to test what makes a snippet of code harder to understand. I had access to before-after snapshots of code reviews and fixes for these programs, from which I learned patterns.

All of the research presented in this thesis was funded by a grant from the MIT-IBM Watson AI lab. For this, I am very grateful to David Cox, Aude Oliva, and the lab. Research I did during the summers of 2020 and 2021 was supported by the MIT-IBM Watson AI lab and Stephen Magill's team at Sonatype Inc. (previously Muse Dev Inc.) respectively. The first year of my PhD was partially funded by the FinTech initiative at CSAIL. MIT's generous undergraduate research programs (Quest for Intelligence and the MIT SuperUROP program) supported multiple superb undergraduate researchers I worked with, including my collaborators Tamara (chapters 2, 3) and Teodor (chapter 4). My gratitude to all these sources for funding me and my collaborators.

I thank the following for their outstanding support with the computational resources I needed for my experiments: John Cohn, Jessie Rosenberg, Christopher Laibinis, and Luke Inglis from MIT-IBM AI lab and IBM Research; the *ninjas* at TIG, CSAIL - Jonathan Proulx, Alex Closs, Garrett Wollman, and Shaohao Chen from BCS.

A significant portion of the work in this thesis is built on the effort of several researchers—those who prepared and publicly released datasets, codebases and ML models, those who put out helpful blogs and videos explaining their work, those who thanklessly answered queries on public forums, and those who have diligently worked on several important open source efforts such as Pytorch and Hugging Face. I thank them all; I stand on the shoulders of such giants.

Kim Martineau, Rachel Gordon, Steven Nadis, Matt Busekroos, Jake Lambert, Phil Arsenault, and Erin Underwood from CSAIL, MIT News and EmTech MIT helped in communicating much of the work presented in this thesis to the public via portals like MIT News, CSAIL spotlights, talks at EmTech, and posts on social media.

Several courses I took in computer science and cognitive science have shaped my views on the topics I address in this thesis. Thanks especially to professors Armando Solar-Lezama, Regina Barzilay, Nickolai Zeldovich, Frans Kaashoek, Nancy Kanwisher, Pawan Sinha, Josh Tenenbaum, Ted Gibson, and Athulya Aravind for their inspiring courses. They posed big-picture questions and constantly encouraged us to think beyond.

It takes a village to raise a child. Some instilled a sense of rigor and curiosity while others mentored me while I was finding my way into academia. Thanks to the many inspiring teachers from my school; Prof. Jitender Chhabra – my undergraduate advisor; Varun Aggarwal – my manager at Aspiring Minds; Sumit Gulwani, professors Rupesh Nasre, Jitender Chhabra and Lav Varshney – my letter writers to graduate school; Sumit for recommending a visit to the Microsoft Research India lab in 2017, and to Bill Thies, Sriram Rajamani and Swami Manohar for being my hosts there and talking to me about life after graduate school; professors Bogdan Vasilescu and Jonathan Aldrich for being welcoming hosts at CMU in 2017. A special note of thanks to Varun who patiently taught me to do rigorous research and eventually introduced me to Una-May, and for setting up Aspiring Minds where I met excellent colleagues to do fun research with.

Lastly, thanks to the many new friends I made during my stint here, and older ones who kept in touch, many through long phone calls, all of whom ensured I remained in

high spirits. Cybersecurity, privacy, and GPT-4 can operate with reduced internet data, so I will avoid mentioning them all. Michael Collins, previous flatmates, and friends at Chateau ensured I had a comfortable stay. Multiple instructors ensured I stayed physically healthy. Members of ALFA shared my enthusiasm for the outdoors, finding good food spots, and having fun at work. Thank you all for the warm memories. I am grateful to Una-May and Blake for their benevolent gesture of offering their summer home to a few of us labmates when COVID was at its worst in the USA; they set a very high bar for care and empathy. Thanks to my aunts and their families, who made me a part of theirs during my stay here. Thanks to my parents for making me who I am, and all my family for their support through the years and for visiting me here. Thanks to my partner for always making me smile. Thanks also to the many canine friends who kept me company through my stint here; they are too important to remain anonymous: Kencha, Julie, Todd, Maisy, Talula, Zuko, Bucky, Luna, and Alpha. Olivia, Lily, and Cascade make the list despite their feline forms.

This perhaps will be the most read page of my thesis. If I've missed mentioning you, know that I'll always appreciate all that you do.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The central theme of this thesis is how we humans understand computer programs, and how we can teach machines to understand programs the way we do. It stems from two puzzling observations I made before starting graduate school, which were the following.

1.1 Puzzle 1 - Human intelligence tasks

Before graduate school, I worked for a research group where we assessed and quantified skills which signal employability in a labor market. Skill assessments had until then largely been confined to objective tests (multiple-choice questions) because subjective assessments (free-form responses) were harder to assess. Our group was one of the first to view the problem of subjective assessments of skills as problems in computer science. We demonstrated how many free-form response assessments can be cast as problems in machine learning (ML) [Srikant et al., 2019]. Test-takers' responses were treated as data-points in a high dimensional space, from which we predicted their true, latent, underlying score. We developed novel assessments for skills like spoken English [Shashidhar et al., 2015a], written English [Shashidhar et al., 2015b, Unnam et al., 2019], fine motor skills [Singh and Aggarwal, 2016] and situational judgement [Stemler et al., 2016], in addition to domain-general skills like logic and quantitative reasoning [Aggarwal et al., 2016].

One area that I was closely involved in was the assessment of computer programming skills [Srikant and Aggarwal, 2014a, Singh et al., 2016, Takhar and Aggarwal, 2019]. We trained predictive models to look beyond the functional correctness of programs, and assessed their partial correctness based on their semantic content. This helped a common issue arising in program assessments: zero credit for a failed test-suite despite having written a program that was *almost* what was expected. We successfully demonstrated how to utilize corpus-level statistics to assess programs that match those written by expert programmers.

As background, the notion of *code models*—language models or statistical models trained on code corpora, was beginning to be put to test around the same time we were developing our predictive models for assessing programs (*circa* 2013). Works by Allamanis and Sutton [2013] and Raychev et al. [2015] demonstrated practical applications like code summarization and variable renaming in Java and Javascript respectively. The idea of using a corpus of programs to train language models and other statistical models to ease developer workload was becoming mainstream. Allamanis et al. [2018b] surveys subsequent works in this space.

To train our machine learning (ML) models on code, we were routinely annotating *ground truth labels* for the programs in our corpus. This required experienced programmers evaluating a subset of programs in our corpus with a carefully constructed rubric. In watching expert programmers perform this task, I made a puzzling observation: experts were needed to annotate even the simplest of programming tasks, and irrespective of their expertise, they found annotation challenging and time consuming. This was in contrast to domains like speech, images, and text understanding, in which most tasks which are hard to describe using algorithms can be quite easily done by non-experts, essentially invoking their innate *human intelligence*. Examples of such tasks include identifying objects in an image or identifying inarticulate speech or text samples. Amazon's Mechanical Turk [Paolacci et al., 2010], a popular crowdsourcing platform, uses the term human intelligence tasks (HIT) to describe such tasks. Strangely, understanding and annotating programs were never HITs, even for the best, expert programmers. Any one of the following explanations possibly justifies this

observation.

- The time the human race has been using and inventing programming languages is much less than the time the human race has learned and acquired skills like natural language, vision, and speech. It is hence possible that there exist dedicated brain regions for processing language, vision, and speech while none exist for programming languages, thus requiring us more time to process programs.
- Experience could be another factor. A typical adult is exposed to many more years of language, vision, and speech than a programming language in their lifetime. Perhaps a child who is exposed to a programming language as its first language will process programs as easily as adults process natural languages.
- It is possible that the nature of programming models and environments offered by different languages contribute to the ease with which we understand programs. For instance, *visual learners* may find web mark-up languages simpler and more *natural* to reason about. Similarly, some find it easier to visualize and mentally manipulate rows and columns of data, thus finding languages like R, Matlab, and libraries like Numpy easier to understand. Some anecdotally find functional languages easier to comprehend than others.

While the explanations for the underlying processes were not clear, it was clear that in order to solve this puzzle, it was necessary to address the behavioral factors that underlie comprehension. Further, it suggested room for rigorously defining ideas like *visual learners* and *easier to reason about* in this context.

This observation also raised questions about the different code models proposed in the literature. The accuracy of code models at tasks such as summarization and predicting tokens were not as high as at tasks in language processing. Was this because training code models on tasks that were not HITs inherently harder? Understanding the brain and behavioral bases of comprehension would hopefully inform how we could improve training computational models to perform code reasoning tasks.

Around the same time, Siegmund et al. [2014] published their influential study on using fMRI to study programmers' brains. This provided additional support for the potential to study and establish the behavioral foundation for comprehending

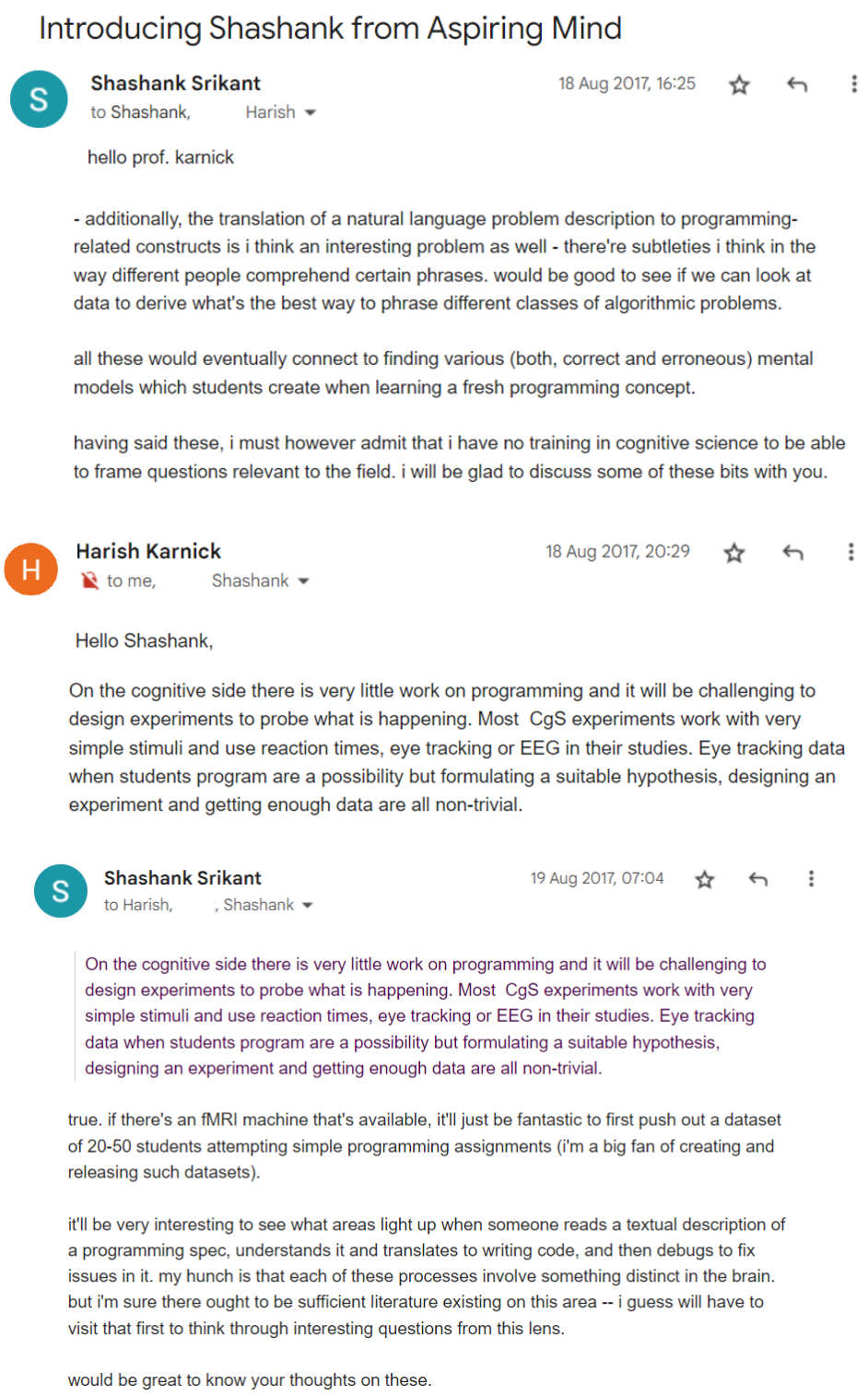


Figure 1-1: Excerpts from an email conversation with Prof. Harish Karnick, Emeritus Fellow, IIT Kanpur, on the possibility of studying the brain bases of programming, *circa* August 2017.

code. Shown in Figure 1-1 is my conversation on this topic with Prof. Harish Karnick, Emeritus Fellow at Indian Institute of Technology Kanpur (IITK)[1], just before I started graduate school.

1.2 Puzzle 2 - Programs and patterns

Around the same time as I was asking these questions, I came across a now prominent work on the naturalness of software by Hindle et al. [2016]. The authors analyzed code *in the wild*—available in public software projects, open-source repositories *etc.*, and showed the frequency distribution of the tokens and phrases (collection of tokens) appearing in code corpora followed a Zipf-like distribution. This frequency distribution is obtained by first computing the frequency of all the unique tokens across all programs appearing in a corpus, and then plotting these frequencies in a ranked (usually descending) order. Figure 1-2 shows an example of a Zipf-like distribution.

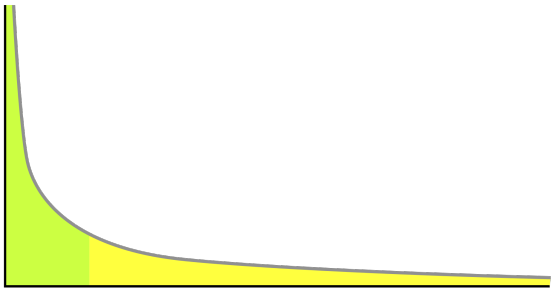


Figure 1-2: A Zipf-like distribution. For language, the X-axis represents unique words or phrases appearing in corpora of text, and the Y-axis the frequencies corresponding to each of those words/phrases occurring in the corpora. Image source: Wikimedia Commons

The authors found this distribution to hold irrespective of the programming language itself: the distribution of tokens in corpora across languages like C, Java, Python each showed a similar distribution. Word frequencies from corpora of text have long been shown to follow a Zipf-like distribution [Piantadosi, 2014]. Such a statistical regularity of words and tokens has been used in applications such as data compression [Schwartz, 1963], and cryptography [Boztas, 1999]. Hindle et al. [2016]

[1] https://iitk.ac.in/new/dr-harish-karnick

propose similar applications for code that can make use of such regularity, such as code auto-completion, code summarization, and more.

While the reason for the occurrence of this particular distribution is currently not well established, one popular account attributes it to a communicative optimization principle [Piantadosi, 2014]. According to this principle, the task of speaking and writing text is considered to have been evolved to optimally facilitate communication. The principle suggests any communication language will exhibit a statistical regularity that helps maximize its successful reception. In light of this principle, the results from Hindle et al. [2016], which had also been reported in previous studies [Shooman and Laemmel, 1977, Clark and Green, 1977, Chen, 1991], remain particularly puzzling. For one, the distributions are similar despite the many differences in the grammar and the execution semantics of programming languages and natural languages. Further, the communication recipient in the case of code is an assembly-level, register-based execution model. Why do humans then produce code with a Zipf-like token distribution when communicating with a machine that has been invented by humans?

Could it follow that our *minds*—our consciously aware perceptions and thoughts [Shiffrin et al., 2020]—use some mechanism which naturally constrains the way we express communicative thought? What is an analytic description of this mechanism? As a corollary, when understanding code, do our minds inherently expect this distribution? How we might is unclear. Bicknell and Levy [2012], Malmaud et al. [2020] propose a probabilistic framework to explain text understanding. Does a similar probabilistic account explain code understanding? Importantly, are code models encoding this probabilistic mechanism, thus giving them the ability to reason about the communicated intent in programs? These were the open questions motivating me, which I attempt to investigate in this thesis.

1.3 Reconciling these puzzles - Questions that arise

These two puzzles inform the theme of the questions I explore in this work. Reconciling the questions raised by the puzzles, I ask the following questions, which lie at the

intersection of computational models of code understanding and human behavioral bases of code understanding.

- **Thesis Question 1: Computational perspective.** To start with, what is a good framework to evaluate code models' understanding of programs. Code models are either *supervised*—trained to infer a specific task like code summarization, type inference, *etc.*; or *unsupervised*—language models trained on code corpora. Further, these models are typically trained either on *source code* or, as I propose, can be trained on *execution traces of programs*.
- **Thesis Question 2: Cognitive neuroscience perspective.** Similarly, what is a good framework to understand how code comprehension happens in our brains and minds. *Brain* refers to the neurons, cells, and chemicals that govern activities of an organism. *Mind* is often considered consciously aware perceptions and thoughts [Shiffrin et al., 2020].
- **Thesis Question 3: Bridging the two perspectives.** Is there any correspondence between the information encoded by code models and human brains when comprehending programs? Can computational models help in learning how our brains and minds comprehend programs? Can our brain and minds inform the better design of computational models?

In the following section, I describe how the chapters in this thesis correspond to these key research questions.

1.4 Thesis map

I develop multiple ideas to address each of the three questions introduced in Section 1.3. Each idea has been described as a separate chapter in this thesis. I motivate the relevance of each idea and summarize key results here. I also describe how these ideas contribute to addressing the three broader thesis questions.

Figure 1-3 summarizes the key themes of the different chapters in this thesis, and shows how they relate ontologically.

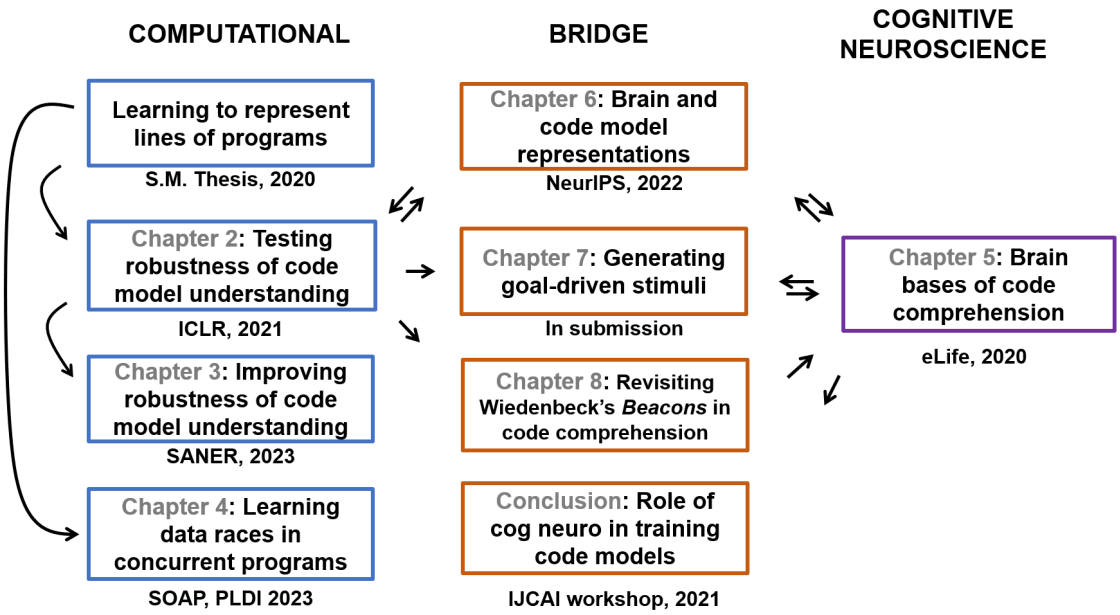


Figure 1-3: Thesis map. The three verticals correspond to the three broad thesis questions I introduce in Section 1.3. Each box describes the theme of one chapter in this thesis. The *bridge* chapters inform and are informed by ideas from chapters in both - the *computational* and *cognitive neuroscience* verticals. The arrows indicate these relationships.

Thesis Question 1. Computational perspective

Chapter 2. Testing the robustness of code model understanding using source code modifications.

I propose a principled method to test how well models trained on source code understand programs. The key idea is that humans' understanding of code is not affected by minor changes made to the code. We hold code models to the same test. For example, a model's output should not be affected by a variable being consistently renamed from `x` to `y`. The proposed method attempts to find such small changes which (a) do not alter the semantics of the original program, but (b) change the model's output. If such minor modifications are easy to find, it demonstrates the brittle *understanding* these models have of programs. I formulate finding such minor modifications as a first-order optimization problem. The optimization solves for two key components: which parts of the program to transform, and what transformations to use. I show that it is important to optimize both these aspects to generate the best candidate changes which are minimal and can flip a model's decision. Although I evaluate this

method on Python and Java programs, the proposed method is independent of the model (supervised or unsupervised), or the languages the models are trained on.

The details of this method are presented in chapter 2. It is, in full, a reprint of *Generating adversarial computer programs using optimized obfuscations.* Srikant, S., Liu, S., Mitrovska, T., Chang, S., Fan, Q., Zhang, G., and O'Reilly, U.M. (2021). ICLR 2021. [Srikant et al., 2021]

Chapter 3. Improving the robustness of code model understanding while retaining model accuracy

In this work, I propose improving the baseline understanding of code models that I test and measure in chapter 2.

I separately address two types of models - unsupervised code models (language models) and supervised, fine-tuned models. For unsupervised models, I provide a constrastive learning setup to learn those properties which humans can naturally reason about when comprehending code, such as invariance to variable names. For supervised models, I identify the existence of a *sweet-spot* in the frequency of updates made to the model parameters when being updated to learn the different human-like properties.

I show that these two solutions bring models closer to what humans can reason about when comprehending code.

I describe this work in Chapter 3. It is, in full, a reprint of *CLAWSAT: Towards Both Robust and Accurate Code Models.* Jia*, J., Srikant*, S., Mitrovska, T., Chang, S., Gan, C., Liu, S., and O'Reilly, U.M. (2023). SANER 2023 [Jia et al., 2022]. Jinghan Jia contributed equally with me as a primary author in this work.

Chapter 4. Learning code models to understand concurrent programs using program execution traces.

Of the several applications and developer tasks which code models can learn and assist, tasks that reason about concurrent programs have been studied the least [Allamanis et al., 2018a]. In this work, I describe how code models can learn to understand

concurrent programs, and specifically reason about data races.

I first propose a theoretical formulation for an ML model to learn data races. I discuss how operationalizing this idea is challenging. I then study the limits of neural networks architectures in learning and detecting data races from execution traces. I model events appearing in a program thread as a string of characters in a toy language that I design. Using such a language to denote threads, I study how well different ML models can be trained to detect the presence of specific substrings in the toy language that represent data races.

I then attempt to learn ML models on the execution traces of real concurrent programs. In my attempt, I learned the following severe limitations in prior work:

- No comprehensive dataset of concurrent programs exists in which data race conditions have been clearly labeled. Such a dataset is essential to get started with any ML-based approach.
- Data race detection algorithms proposed over the last four decades have not been evaluated on such comprehensive, labeled datasets. Instead, they have typically compared their performance to other prior algorithms and have reported only relative improvement. It is thus unclear how accurate these different algorithms are.

We develop *RaceInjector* to address this issue of a lack of a comprehensive dataset, which uses a Satisfiability Modulo Theories (SMT)-based solver to generate multiple possible traces which contain an injected data race. This generates a dataset of traces, wherein each trace is guaranteed to contain a data race. Such a dataset is suitable as a benchmark to rigorously evaluate other data race detection algorithms, and train ML models to detect data races.

I describe this work in Chapter 4. Section 4.2 describes the theoretical formulation. Section 4.3 refers to a thesis I mentored, authored by Teodor Rares Begu: *Modeling concurrency bugs using machine learning.* Rares Begu, T., Srikant, S., and O'Reilly, UM (2020). MIT SuperUROP Thesis [Rares Begu, 2020]. Section 4.4, in full, is a reprint of *RaceInjector: Injecting Races To Evaluate And Learn Dynamic Race Detection Algorithms.* Wang, M., Srikant, S., Samak, M., and O'Reilly, U.M. (2023) Wang et al. [2023].

How the chapters contribute to the computational perspective.

- **Chapter 2.** Humans can understand code despite simple changes made to it. Can models do the same? The method proposed in this work uses this idea, and serves as a practical baseline test of how well code models understand code.
Given how general our formulation is, we also show its application in generating English sentences that can elicit specific neural responses in the brain (details in Chapter 7).
- **Chapter 3.** This work identifies ways to fix the brittleness in code model understanding which the method from Chapter 2 identifies.
- **Chapter 4.** This work takes the first step towards training code models to comprehend and reason about concurrent programs. It specifically develops a way forward for designing data-driven data race detectors, which can potentially improve upon the heuristics that have been proposed over the last four decades. To the best of my knowledge, no previous attempts have been made in this regard.

Thesis Question 2. Cognitive neuroscience perspective

Chapter 5. In this chapter, I identify the regions of our brains involved in code comprehension. We consider two candidate brain systems—the Multiple Demand (MD) system and the Language system (LS). While the MD system is known to respond to stimuli involving general problem solving, math operations, and logic operations, the LS is known to be sensitive to language inputs alone. We establish whether reading and comprehending programs activates the LS or the MD system by using fMRI to study brain activity in participants reading code. We find that the LS does not consistently respond when comprehending programs, while the MD strongly does.

This chapter, in full, is a reprint of the arXiv report Srikant et al. [2023a]. The report is a rewrite of *Comprehension of computer code relies primarily on domain-general executive brain regions.* Ivanova, A. A., Srikant, S., Sueoka, Y., Kean, H. H., Dhamala, R., O'Reilly, U.M., Bers, M. U., and Fedorenko, E. (2020). Elife, 9:e58906

[Ivanova et al., 2020]. The chapter informs the results of the eLife work to a computer science audience; the original eLife work was written to primarily inform a cognitive neuroscience audience.

How the chapter contributes to the cognitive neuroscience perspective.

This work establishes the regions of the brain most responsible for code comprehension. Knowledge of the functional regions of the brain involved in code comprehension allows us to probe more into the nature of information represented (stored) in these brain regions.

Thesis Question 3. Bridging the two perspectives

Chapter 6. Mapping brain and model representations

In this work, I attempt to describe the nature of information encoded in the different brain regions identified in Section 1.4. This gives us an insight into the division of labor during code comprehension. For instance, token-related information can be encoded more prominently in Language system (LS), but, say, loops can be encoded in the Multiple Demand (MD) system. Similarly, numbers-related processing can happen in the MD while strings-related processing happen in the LS.

In addition to understanding this division of labor, our approach of decoding program-related information the first step at establishing the bases of the MD system. The MD system has typically been associated with *fluid intelligence*, but no description exists of *fluid intelligence*. Thus, the nature of operations performed in the MD system have not been rigorously described. Programs are a natural way to describe tasks that resemble *problem-solving* and *general intelligence* [Newell et al., 1958]. Newell et al. [1958] were the first to propose how tasks requiring some form of problem-solving can be described using computer programs. Thus, if information from programs are well encoded in the MD system (as opposed to the LS), it provides initial evidence to the bases of the MD system.

Additionally, I test whether information encoded in the brain can predict repre-

sentations (embeddings) learned by code models. A strong correspondence between the two representations—of brain regions and code models—would suggest that the learning objective used by code models to learn the parameters of the models resembles that employed by our brains, which thus result in the emergence of similar representations.

We show that the program-related information is encoded both in the MD and Language systems. Execution-related properties are more strongly encoded in the MD system. We find that representations from more complex models tend to align best with the MD system than the LS.

I describe this work in Chapter 6. It is, in full, a reprint of **Convergent representations of computer programs in human and artificial neural networks.** Srikant*, S., Lipkin*, B., Ivanova, A. A., Fedorenko, E., and O'Reilly, U.M. (2022). NeurIPS 2022 [Srikant et al., 2022]. Ben Lipkin contributed equally with me as the primary author of this work.

Chapter 7. Generating stimuli for cognitive neuroscience and psycholinguistics

Experiments in psycholinguistics and the cognitive neuroscience of language rely on linguistic stimuli (sentences) which either possess specific linguistic properties or which target specific cognitive processes. Such stimuli are generally assembled using manual or semi-manual methods, limiting their quality, quantity, and diversity.

I show how the method I propose in Chapter 2 can be reformulated to automate the generation of stimuli which target specific cognitive processes or possess desired linguistic properties while not being subject to experimenter biases which may arise from manual methods.

I describe this work in Chapter 7. It is, in full, a reprint of *GOLI: Goal-Optimized Linguistic Stimuli for Psycholinguistics and Cognitive Neuroscience.* Srikant, S., Tuckute, G., Liu, S., and O'Reilly, U.M. (2023) [Srikant et al., 2023b].

Chapter 8. What is *important* to programmers when comprehending code?

Soloway and Ehrlich [1984] and Wiedenbeck [1986] proposed the presence of *beacons* in programs: substrings in a program which programmers deem important to their understanding of the program. In this work, I verify whether common factors known to affect the comprehension of text such as surprisal and word length, and whether code model's representations can predict the behavioral finding by Wiedenbeck [1986]. The motivation is to determine the factors influencing programmer notions like "*important part of the code*", "*confusing part of the code*", and other such vaguely defined terms typically used by programmers.

I conduct a behavioral experiment in which I find the model's representations to be good predictors of the *importance* of a token in a program's overall comprehension, while the surprisal of a token as a signal is a weak predictor. I describe this work in Chapter 8.

How the chapters contribute to bridging the two perspective.

- **Chapter 6.**

– Our work is the first to describe the nature of program-related information encoded in the two brain regions most closely associated with code comprehension—the Multiple Demand system and the Language system.

– We take the first steps in describing the foundations of the MD system. Programs are a natural way to describe problem-solving tasks, which the MD system is believed to specialize in.

– We show a weak correspondence between the representations of a program in the brain and in code models. Future work may try to improve the architecture of current code models to improve this correspondence [Srikant and O'Reilly, 2021].

- **Chapter 7.** This work shows how experiment stimuli can be generated that satisfy diverse goals. This utility of this method was recently demonstrated in Tuckute et al. [2023], which establishes the ability to noninvasively control neural activity in higher-level cortical areas, like the language network.

While we do not demonstrate the generation of code stimuli in this work, the method can be used to similarly learn more about the sensitivity of the MD and Language system to the presence of specific code patterns.

- **Chapter 8.** I show how language models of code, when used as proxies of expert programmer knowledge, can help study different behavioral responses seen when understanding code.

1.5 Software

Software repositories relevant to the chapters presented in this thesis:

- **Chapter 2.** `https://github.com/ALFA-group/adversarial-code-generation`
- **Chapter 3.** `https://github.com/ALFA-group/CLAW-SAT`
- **Chapter 4.** `https://github.com/ALFA-group/RaceInjector-counterexamples`
- **Chapter 5.** `https://github.com/ALFA-group/neural-program-comprehension`
- **Chapter 6.** `https://github.com/ALFA-group/code-representations-ml-brain`
- **Chapter 7.** `https://github.com/alfa-group/goli`
- **Chapter 8.** `https://github.com/ALFA-group/beacons-in-code-comprehension`

Chapter 2

Testing the robustness of code model understanding using source code modifications

Preface. This chapter, in full, is a re-print of **Generating adversarial computer programs using optimized obfuscations.** Srikant, S., Liu, S., Mitrovska, T., Chang, S., Fan, Q., Zhang, G., and O'Reilly, U.M. (2021). ICLR 2021 [Srikant et al., 2021].

Refer to Section 1.4 to read more about the motivation behind this work, and how it connects to the rest of the chapters presented in this thesis.

2.1 Introduction

Machine learning (ML) models are increasingly being used for software engineering tasks. Applications such as refactoring programs, auto-completing them in editors, and synthesizing GUI code have benefited from ML models trained on large repositories of programs, sourced from popular websites like GitHub Allamanis et al. [2018c]. They have also been adopted to reason about and assess programs [Srikant and Aggarwal, 2014a, Si et al., 2018], find and fix bugs [Gupta et al., 2017, Pradel and Sen, 2018], detect malware and vulnerabilities in them [Li et al., 2018b, Zhou et al., 2019] *etc.*

thus complementing traditional program analysis tools. As these models continue to be adopted for such applications, it is important to understand how robust they are to *adversarial attacks*. Such attacks can have adverse consequences, particularly in settings such as security [Zhou et al., 2019] and compliance automation [Pedersen, 2010]. For example, an attacker could craft changes in malicious programs in a way which forces a model to incorrectly classify them as being benign, or make changes to pass off code which is licensed as open-source in an organization's proprietary code-base.

Adversarially perturbing a program should achieve two goals – a trained model should flip its decision when provided with the perturbed version of the program, and second, the perturbation should be *imperceivable*. Adversarial attacks have mainly been considered in image classification Goodfellow et al. [2014], Carlini and Wagner [2017], Madry et al. [2018b], where calculated minor changes made to pixels of an image are enough to satisfy the imperceptibility requirement. Such changes escape a human's attention by making the image look the same as before perturbing it, while modifying the underlying representation enough to flip a classifier's decision. However, programs demand a stricter imperceptibility requirement – not only should the changes avoid human attention, but the changed program should also importantly functionally behave the same as the unperturbed program.

Program obfuscations provide the agency to implement one such set of imperceivable changes in programs. Obfuscating computer programs have long been used as a way to avoid attempts at reverse-engineering them. They transform a program in a way that only hampers humans' comprehension of parts of the program, while retaining its original semantics and functionality. For example, one common obfuscation operation is to rename variables in an attempt to hide the program's intent from a reader. Renaming a variable `sum` in the program statement `int sum = 0` to `int xyz = 0` neither alters how a compiler analyzes this variable nor changes any computations or states in the program; it only hampers our understanding of this variable's role in the program. Modifying a very small number of such aspects of a program marginally affects how we comprehend it, thus providing a way to produce changes imperceivable

to both humans and a compiler. In this work, we view adversarial perturbations to programs as a special case of applying obfuscation transformations to them.

Having identified a set of candidate transformations which produce imperceivable changes, a specific subset needs to be chosen in a way which would make the transformed program adversarial. Recent attempts Yefet et al. [2019], Ramakrishnan et al. [2020], Bielik and Vechev [2020] which came closest to addressing this problem did not offer any rigorous formulation. They recommended using a variety of transformations without presenting any principled approach to selecting an optimal subset of transformations. We present a formulation which when solved provides the exact location to transform as well as a transformation to apply at the location. Figure 2-1 illustrates this. A randomly selected local-variable (`name`) when replaced by the name `virtualname`, which is generated by the state-of-the-art attack generation algorithm for programs Ramakrishnan et al. [2020], is unable to fool a program summarizer (which predicts `set item`) unless our proposed site optimization is applied. We provide a detailed comparison in Section 2.2. In our work, we make the following key contributions –

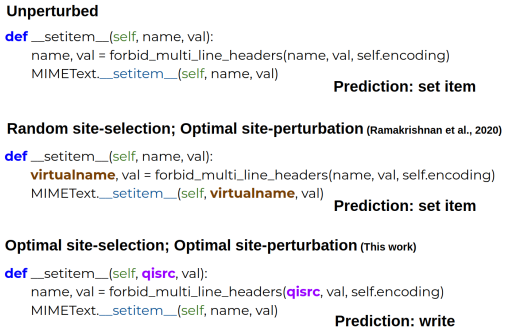


Figure 2-1: The advantage of our formulation when compared to the state-of-the-art.

- We identify two problems central to defining an adversarial program – identifying the sites in a program to apply perturbations on, and the specific perturbations to apply on the selected sites. These perturbations are involve replacing existing tokens or inserting new ones.
- We provide a general mathematical formulation of a perturbed program that models site locations and the perturbation choice for each location. It is independent of programming languages and the task on which a model is trained, while seamlessly modeling the application of multiple transformations to the program.
- We propose a set of first-order optimization algorithms to solve our proposed formulation efficiently, resulting in a differentiable generator for adversarial programs. We

further propose a randomized smoothing algorithm to achieve improved optimization performance.

- Our approach demonstrates a 1.5x increase in the attack success rate over the state-of-the-art attack generation algorithm Ramakrishnan et al. [2020] on large datasets of Python and Java programs.
- We further show that our formulation provides better robustness against adversarial attacks compared to the state-of-the-art when used in training an ML model.

2.2 Related Work

Due to a large body of literature on adversarial attacks in general, we focus on related works in the domain of computer programs. Wang and Christodorescu [2019], Quiring et al. [2019], Rabin et al. [2020], and Pierazzi et al. [2020] identify obfuscation transformations as potential adversarial examples. They do not, however, find an optimal set of transformations to deceive a downstream model. Liu et al. [2017] provide a stochastic optimization formulation to obfuscate programs optimally by maximizing its impact on an *obscurity language model* (OLM). However, they do not address the problem of adversarial robustness of ML models of programs, and their formulation is only to find the right sequence of transformations which increases their OLM's perplexity. They use an MCMC-based search to find the best sequence.

Yefet et al. [2019] propose perturbing programs by replacing local variables, and inserting print statements with replaceable string arguments. They find optimal replacements using a first-order optimization method, similar to Balog et al. [2016] and HotFlip Ebrahimi et al. [2017]. This is an improvement over Zhang et al. [2020], who use the Metropolis-Hastings algorithm to find an optimal replacement for variable names. Bielik and Vechev [2020] propose a robust training strategy which trains a model to abstain from deciding when uncertain if an input program is adversarially perturbed. The transformation space they consider is small, which they search through greedily. Moreover, their solution is designed to reason over a limited context of the program (predicting variable types), and is non-trivial to extend to applications such

as program summarization (explored in this work) which requires reasoning over an entire program.

Ramakrishnan et al. [2020] extend the work by Yefet et al. [2019] and is most relevant to what we propose in this work. They experiment with a larger set of transformations and propose a standard min-max formulation to adversarially train robust models. Their inner-maximizer, which generates adversarial programs, models multiple transformations applied to a program in contrast to Yefet et al. [2019]. However, they do not propose any principled way to solve the problem of choosing between multiple program transformations. They randomly select transformation operations to apply, and then randomly select locations in the program to apply those transformations on.

We instead show that optimizing for locations alone improves the attack performance. Further, we propose a joint optimization problem of finding the optimal location and optimal transformation, only the latter of which Ramakrishnan et al. [2020] (and Yefet et al. [2019]) address in a principled manner. Although formally unpublished at the time of preparing this work, we compare our experiments to Ramakrishnan et al. [2020], the state-of-the-art in evaluating and defending against adversarial attacks on models for programs, and contrast the advantages of our formulation.

2.3 Program Obfuscations as Adversarial Perturbations

In this section, we formalize program obfuscation operations, and show how generating adversarial programs can be cast as a constrained combinatorial optimization problem.

Program obfuscations. We view obfuscation transformations made to programs as adversarial perturbations which can affect a downstream ML/DL model like a malware classifier or a program summarizer. While a variety of such obfuscation transformations exist for programs in general (see section 2A, Liu et al. [2017]), we consider two

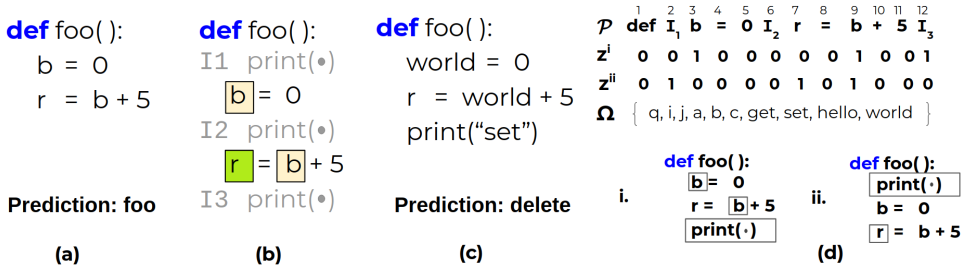


Figure 2-2: (a) A sample program $\mathcal{P}$ containing a function `foo` (b) $\mathcal{P}$ contains five *sites* which can be transformed - two replacesites corresponding to local variables b and r, and three insertsites at locations `I1`, `I2`, `I3`. Ω is a vocabulary of tokens which can be used for the transformations. (c) This is a *perturbed program* with the tokens `world` and `set` from Ω used to replace tokens `b` and at location `I3`. These transformations do not change the original functionality of $\mathcal{P}$, but cause an incorrect prediction `delete` (d) Examples of two site selection vectors $\mathbf{z}^{\text{i}}$, $\mathbf{z}^{\text{ii}}$ selecting different components. $\mathbf{z}_i = 1$ for a location i signifies that the ith token in $\mathcal{P}$ is selected to be optimally transformed. $\mathbf{z}^{\text{i}}$ corresponds to the perturbed program in (c).

broad classes – replaceand insert transformations. In replacetransformations, existing program constructs are replaced with variants which decrease readability. For example, replacing a variable's name, a function parameter's name, or an object field's name does not affect the semantics of the program in any way. These names in any program exclusively aid human comprehension, and thus serve as three replacetransformations. In inserttransformations, we insert new statements to the program which are unrelated to the code it is inserted around, thereby obfuscating its original intent. For example, including a print statement with an arbitrary string argument does not change the semantics of the program in any way.

Our goal hence is to introduce a systematic way to transform a program with insertor replacetransformations such that a trained model misclassifies a program $\mathcal{P}$ that it originally classified correctly.

***Site-selection* and *Site-perturbation* – Towards defining adversarial programs.** Before we formally define an adversarial program, we highlight the key factors which need to be considered in our formulation through the example program introduced in Figure 2-2.

Consider applying the following two obfuscation transformations on the example program $\mathcal{P}$ in Figure 2-2.a – replacing local variable names (a replacetransform), and

inserting `print` statements (an inserttransform). The two local variables `b` and `r` in $\mathcal{P}$ are potential candidates where the replace transform can be applied, while a `print` statement can potentially be inserted at the three locations `I1`, `I2`, `I3` (highlighted in Figure 2-2.b). We notate these choices in a program as *sites*– locations in a program where a unique transformation can be applied.

Thus, in order to *adversarially perturb* $\mathcal{P}$, we identify two important questions that need to be addressed. First, which sites in a program should be transformed? Of the n sites in a program, if we are allowed to choose at most k sites, which set of $\leq k$ sites would have the highest impact on the downstream model's performance? We identify this as the ***site-selectionproblem***, where the constraint k is the ***perturbation strength*** of an attacker. Second, what tokens should be inserted/replaced at the k selected sites? Once we pick k sites, we still have to determine the best choice of tokens to replace/insert at those sites which would have the highest impact on the downstream model. We refer to this as the ***site-perturbationproblem***.

Mathematical formulation. In what follows, we propose a general and rigorous formulation of adversarial programs. Let $\mathcal{P}$ denote a *benign program* which consists of a series of n tokens $\{\mathcal{P}_i\}_{i=1}^{n}$ in the source code domain. For example, the program in Figure 2-2.a, when read from top to bottom and left to right, forms a series of $n = 12$ tokens $\{$`def`, `b`, ..., `r`, `+`, `5`$\}$. We ignore white spaces and other delimiters when tokenizing. Each $\mathcal{P}_i \in \{0,1\}^{|\Omega|}$ here is considered a one-hot vector of length $|\Omega|$, where Ω is a vocabulary of tokens. Let $\mathcal{P}'$ define a *perturbed program* (with respect to $\mathcal{P}$) created by solving the *site-selection*and *site-perturbation*problems, which use the vocabulary Ω to find an optimal replacement. Since our formulation is agnostic to the type of transformation, *perturbation* in the remainder of this section refers to both replaceand inserttransforms. In our work, we use a shared vocabulary Ω to select transforms from both these classes. In practice, we can also assign a unique vocabulary to each transformation we define.

To formalize the *site-selection*problem, we introduce a vector of boolean variables $\mathbf{z} \in \{0,1\}^n$ to indicate whether or not a site is selected for perturbation. If $z_i = 1$ then the ith site (namely, $\mathcal{P}_i$) is perturbed. If there exist multiple occurrences of a token in

the program, then all such sites are marked 1. For example, in Figure 2-2.d, if the site corresponding to local variable b is selected, then both indices of its occurrences, z_3, z_9 are marked as 1 as shown in z^{i}. Moreover, the number of perturbed sites, namely, $\mathbf{1}^T\mathbf{z} \leq k$ provides a means of measuring *perturbation strength*. For example, $k = 1$ is the minimum perturbation possible, where only one site is allowed to be perturbed. To define *site-perturbation*, we introduce a one-hot vector $\mathbf{u}_i \in \{0,1\}^{|\Omega|}$ to encode the selection of a token from Ω which would serve as the insert/replace token for a chosen transformation at a chosen site. If the j^{th} entry $[\mathbf{u}_i]_j = 1$ and $z_i = 1$, then the jth token in Ω is used as the obfuscation transformation applied at the site i (namely, to perturb $\mathcal{P}_i$). We also have the constraint $\mathbf{1}^T\mathbf{u}_i = 1$, implying that only one perturbation is performed at $\mathcal{P}_i$. Let vector $\mathbf{u} \in \{0,1\}^{n\times|\Omega|}$ denote n different $\mathbf{u}_i$ vectors, one for each token i in $\mathcal{P}$.

Using the above formulations for *site-selection*, *site-perturbation*and *perturbation strength*, the *perturbed program* $\mathcal{P}'$ can then be defined as

$$\mathcal{P}' = (\mathbf{1} - \mathbf{z}) \cdot \mathcal{P} + \mathbf{z} \cdot \mathbf{u}, \text{ where } \mathbf{1}^T\mathbf{z} \leq k,\ \mathbf{z} \in \{0,1\}^n,\ \mathbf{1}^T\mathbf{u}_i = 1,\ \mathbf{u}_i \in \{0,1\}^{|\Omega|},\ \forall i, \quad (2.1)$$

where $\cdot$ denotes the element-column wise product.

The adversarial effect of $\mathcal{P}'$ is then measured by passing it as input to a downstream ML/DL model $\boldsymbol{\theta}$ and seeing if it successfully manages to fool it.

Generating a successful adversarial program is then formulated as the optimization problem,

$$\begin{aligned} \underset{\mathbf{z},\mathbf{u}}{\text{minimize}} \quad & \ell_{\text{attack}}((\mathbf{1} - \mathbf{z}) \cdot \mathcal{P} + \mathbf{z} \cdot \mathbf{u}; \mathcal{P}, \boldsymbol{\theta}) \\ \text{subject to} \quad & \text{constraints in (2.1)}, \end{aligned} \quad (2.2)$$

where ℓ_{attack} denotes an attack loss. In this work, we specify ℓ_{attack} as the cross-entropy loss on the predicted output evaluated at $\mathcal{P}'$ in an untargeted setting (namely, without specifying the prediction label targeted by an adversary) Ramakrishnan et al. [2020]. One can also consider other specifications of ℓ_{attack}, e.g., C&W untargeted and targeted attack losses Carlini and Wagner [2017].

2.4 Adversarial Program Generation via First-Order Optimization

Solving problem (2.2) is not trivial because of its combinatorial nature (namely, the presence of boolean variables), the presence of a bi-linear objective term (namely, $\mathbf{z} \cdot \mathbf{u}$), as well as the presence of multiple constraints. To address this, we present a projected gradient descent (PGD) based joint optimization solver (JO) and propose alternates which promise better empirical performance.

PGD as a joint optimization (JO) solver. PGD has been shown to be one of the most effective attack generation methods to fool image classification models Madry et al. [2018b]. Prior to applying PGD, we instantiate (2.2) into a feasible version by relaxing boolean constraints to their convex hulls,

$$\begin{array}{ll} \underset{\mathbf{z},\mathbf{u}}{\text{minimize}} & \ell_{\text{attack}}(\mathbf{z}, \mathbf{u}) \\ \text{subject to} & \mathbf{1}^T\mathbf{z} \leq k,\ \mathbf{z} \in [0,1]^n,\ \mathbf{1}^T\mathbf{u}_i = 1,\ \mathbf{u}_i \in [0,1]^{|\Omega|},\ \forall i, \end{array} \tag{2.3}$$

where for ease of notation, the attack loss in (2.2) is denoted by $\ell_{\text{attack}}(\mathbf{z}, \mathbf{u})$. The continuous relaxation of binary variables in (2.3) is a commonly used trick in combinatorial optimization to boost the stability of learning procedures in practice Boyd et al. [2004]. Once the continuous optimization problem (2.3) is solved, a hard thresholding operation or a randomized sampling method (which regards $\mathbf{z}$ and $\mathbf{u}$ as probability vectors with elements drawn from a Bernoulli distribution) can be called to map a continuous solution to its discrete domain Blum and Roli [2003]. We use the randomized sampling method in our experiments.

The PGD algorithm is then given by

$$\{\mathbf{z}^{(t)}, \mathbf{u}^{(t)}\} = \{\mathbf{z}^{(t-1)}, \mathbf{u}^{(t-1)}\} - \alpha\{\nabla_{\mathbf{z}}\ell_{\text{attack}}(\mathbf{z}^{(t-1)}, \mathbf{u}^{(t-1)}), \nabla_{\mathbf{u}}\ell_{\text{attack}}(\mathbf{z}^{(t-1)}, \mathbf{u}^{(t-1)})\} \tag{2.4}$$

$$\{\mathbf{z}^{(t)}, \mathbf{u}^{(t)}\} = \text{Proj}(\{\mathbf{z}^{(t)}, \mathbf{u}^{(t)}\}), \tag{2.5}$$

where t denotes PGD iterations, $\mathbf{z}^{(0)}$ and $\mathbf{u}^{(0)}$ are given initial points, $\alpha > 0$ is a learning rate, $\nabla_{\mathbf{z}}$ denotes the first-order derivative operation w.r.t. the variable $\mathbf{z}$, and

Proj represents the projection operation w.r.t. the constraints of (2.3).

The projection step involves solving for $\mathbf{z}$ and $\mathbf{u}_i$ simultaneously in a complex convex problem.

A key insight is that the complex projection problem can equivalently be *decomposed* into a sequence of sub-problems owing to the separability of the constraints w.r.t. $\mathbf{z}$ and $\{\mathbf{u}_i\}$. The two sub-problems are –

$$\begin{array}{ll} \underset{\mathbf{z}}{\text{minimize}} & \|\mathbf{z}-\mathbf{z}^{(t)}\|_2^2 \\ \text{subject to} & \mathbf{1}^T\mathbf{z} \le k,\ \mathbf{z}\in[0,1]^n, \end{array} \quad \text{and} \quad \begin{array}{ll} \underset{\mathbf{u}_i}{\text{minimize}} & \|\mathbf{u}_i-\mathbf{u}_i^{(t)}\|_2^2 \\ \text{subject to} & \mathbf{1}^T\mathbf{u}_i = 1,\ \mathbf{u}_i\in[0,1]^{|\Omega|}, \end{array} \quad \forall i. \tag{2.6}$$

The above subproblems w.r.t. $\mathbf{z}$ and $\mathbf{u}_i$ can optimally be solved by using a bisection method that finds the root of a scalar equation. We use this decomposition and solutions to design an alternating optimizer, which we discuss next.

Alternating optimization (AO) for fast attack generation. While JO provides an approach to solve the unified formulation in (2.2), it suffers from the problem of getting trapped at a poor local optima despite attaining stationarity Ghadimi et al. [2016]. We propose using AO Bezdek and Hathaway [2003] which allows the loss landscape to be explored more aggressively, thus leading to better empirical convergence and optimality (see Figure 2-4a).

AO solves problem (2.2) one variable at a time – first, by optimizing the site selection variable $\mathbf{z}$ keeping the site perturbation variable $\mathbf{u}$ fixed, and then optimizing $\mathbf{u}$ keeping $\mathbf{z}$ fixed. That is,

$$\mathbf{z}^{(t)} = \underset{\mathbf{1}^T\mathbf{z}\le k,\ \mathbf{z}\in[0,1]^n}{\arg\min}\ \ell_{\text{attack}}(\mathbf{z},\mathbf{u}^{(t-1)}) \quad \text{and} \quad \mathbf{u}_i^{(t)} = \underset{\mathbf{1}^T\mathbf{u}_i=1,\ \mathbf{u}_i\in[0,1]^{|\Omega|}}{\arg\min}\ \ell_{\text{attack}}(\mathbf{z}^{(t)},\mathbf{u})\ \forall i. \tag{2.7}$$

We can use PGD, as described in (2.6), to similarly solve each of $\mathbf{z}$ and $\mathbf{u}$ separately in the two alternating steps. Computationally, AO is expensive than JO by a factor of 2, since we need two iterations to cover all the variables which JO covers in a single iteration. However, in our experiments, we find AO to converge faster. The decoupling in AO also eases implementation, and provides the flexibility to set a different number

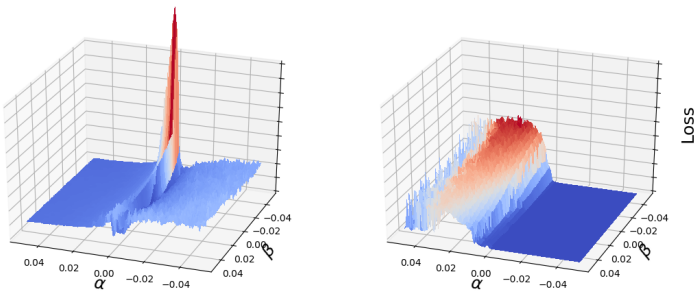
of iterations for the u-step and the z-step within one iteration of AO. We also remark that the AO setup in (2.7) can be specified in other forms, *e.g.* alternating direction method of multipliers (ADMM) Boyd et al. [2011]. However, such methods use an involved alternating scheme to solve problem (2.2). We defer evaluating these options to future work.

Randomized smoothing (RS) to improve generating adversarial programs. In our experiments, we noticed that the loss landscape of generating adversarial program is not smooth (Figure 2-3). This motivated us to explore surrogate loss functions which could smoothen it out. In our work, we employ a convolution-based RS technique Duchi et al. [2012] to circumvent the optimization difficulty induced by the non-smoothness of the attack loss ℓ_{attack}. We eventually obtain a smoothing loss ℓ_{smooth}:

$$\ell_{\text{smooth}}(\mathbf{z}, \mathbf{u}) = \mathbb{E}_{\boldsymbol{\xi},\boldsymbol{\tau}}[\ell_{\text{attack}}(\mathbf{z} + \mu\boldsymbol{\xi}, \mathbf{u} + \mu\boldsymbol{\tau})], \tag{2.8}$$

where $\boldsymbol{\xi}$ and $\boldsymbol{\tau}$ are random samples drawn from the uniform distribution within the unit Euclidean ball, and $\mu > 0$ is a small smoothing parameter (set to 0.01 in our experiments).

The rationale behind RS (2.8) is that the convolution of two functions (smooth probability density function and non-smooth attack loss) is at least as smooth as the smoothest of the two original functions. The advantage of such a formulation is that it is independent of the loss function, downstream model, and the optimization solver chosen for a problem. We evaluate RS



(a) **(b)**

Figure 2-3: The original loss landscape for a sample program (2-3a). Randomized smoothing produces a flatter and smoother loss landscape (2-3b). We plot the loss along the space determined by the vector $(\alpha.\text{sgn}(\nabla_x f(x)) + \beta.\text{Rademacher}(0.5))$ for α, $\beta \in [-0.05, 0.05]$ Engstrom et al. [2018]

on both AO and JO. In practice, we consider an empirical Monte Carlo approximation of (2.8), $\ell_{\text{smooth}}(\mathbf{z}, \mathbf{u}) = \sum_{j=1}^{m}[\ell_{\text{attack}}(\mathbf{z} + \mu\boldsymbol{\xi}_j, \mathbf{u} + \mu\boldsymbol{\tau}_j)]$. We set $m = 10$ in our experiments to save on computation time. We also find that smoothing the site perturbation variable u contributes the most to improving attack performance. We hence perturb only u to further save computation time.

2.5 Experiments & Results

We begin by discussing the following aspects of our experiment setup – the classification task, the dataset and model we evaluate on, and the evaluation metrics we use.

Task, Transformations, Dataset. We evaluate our formulation of generating optimal adversarial programs on the problem of program summarization, first introduced by Allamanis et al. [2016]. Summarizing a function in a program involves predicting its name, which is usually indicative of its intent. We use this benchmark to test whether our adversarially perturbed program, which retains the functionality of the original program, can force a trained summarizer to predict an incorrect function name. We evaluate this on a well maintained dataset of roughly 150K Python programsRaychev et al. [2016a] and 700K Java programs Alon et al. [2018a]. They are pre-processed into functions, and each function is provided as input to an ML model. The name of the function is omitted from the input. The ML model predicts a sequence of tokens as the function name. We evaluate our work on six transformations (4 replaceand 2 inserttransformations). The results and analysis that follow pertains to the case when any of these six transformations can be used as a valid perturbation, and the optimization selects which to pick and apply based on the *perturbation strength*k. This is the same setting employed in the baseline Ramakrishnan et al. [2020].

Model. We evaluate a trained SEQ2SEQ model. It takes program tokens as input, and generates a sequence of tokens representing its function name. We note that our formulation is independent of the learning model, and can be evaluated on any model for any task. The SEQ2SEQ model is trained and validated on 90% of the data while

tested on the remaining 10%. It is optimized using the cross-entropy loss function.

CODE2SEQ Alon et al. [2018a] is another model which has been evaluated on the task of program summarization. Its architecture is similar to that of SEQ2SEQ and contains two encoders - one which encodes tokens, while another which encodes AST paths. The model when trained only on tokens performs similar to a model trained on both tokens and paths (Table 3, Alon et al. [2018a]). Thus adversarial changes made to tokens, as accommodated by our formulation, should have a high impact on the model's output. Owing to the similarity in these architectures, and since our computational bench is in Pytorch while the original CODE2SEQ implementation is in TensorFlow, we defer evaluating the performance of our formulation on CODE2SEQ to future work.

Evaluation metrics. We report two metrics – Attack Success Rate (ASR) and F1-score. ASR is defined as the percentage of output tokens misclassified by the model on the perturbed input but correctly predicted on the unperturbed input, *i.e.* ASR $= \frac{\sum_{i,j} \mathbb{1}(\boldsymbol{\theta}(\mathbf{x}'_i) \neq y_{ij})}{\sum_{i,j} \mathbb{1}(\boldsymbol{\theta}(\mathbf{x}_i) = y_{ij})}$ for each token j in the expected output of sample i. Higher the ASR, better the attack. Unlike Ramakrishnan et al. [2020], we evaluate our method on those samples in the test-set which were fully, correctly classified by the model. Evaluating on such fully correctly classified samples provides direct evidence of the adversarial effect of the input perturbations (also the model's adversarial robustness) by excluding test samples that have originally been misclassified even without any perturbation. We successfully replicated results from Ramakrishnan et al. [2020] on the F1-score metric they use, and acknowledge the extensive care they have taken to ensure that their results are reproducible. As reported in Table 2 of Ramakrishnan et al. [2020], a trained SEQ2SEQ model has an F1-score of 34.3 evaluated on the entire dataset. We consider just those samples which were correctly classified. The F1-score corresponding to 'No attack' in Table 2.1 is hence 100. In all, we perturb 2800 programs in Python and 2300 programs in Java which are correctly classified.

Method	$k = 1$ site				$k = 5$ sites			
	ASR		F1		ASR		F1	
No attack	0.00		100.00		0.00		100.00	
Random replace	0.00		100.00		0.00		100.00	
BASELINE*	19.87		78.18		37.50		59.54	
AO	23.16	+3.29 ▲	74.78	-3.40 ▲	43.53	+6.03 ▲	53.75	-5.79 ▲
JO	23.32	+3.45 ▲	74.56	-3.62 ▲	41.95	+4.45 ▲	56.06	-3.48 ▲
AO + RS	30.25	+10.38 ▲	69.52	-8.66 ▲	51.68	+14.18 ▲	47.92	-11.62 ▲
JO + RS	23.95	+4.08 ▲	74.24	-3.94 ▲	48.70	+11.20 ▲	51.55	-7.99 ▲

Table 2.1: Our work solves two key problems to find optimal adversarial perturbations – *site-selection*and *site-perturbation*. The BASELINE method refers to Ramakrishnan et al. [2020]. The *perturbation strength*k is the maximum number of sites which an attacker can perturb. Higher the the Attack Success Rate (ASR), better the attack; the converse holds for F1 score. Our formulation (Eq. 2.2), solved using two methods – alternate optimization (AO) and joint optimization (JO), along with randomized smoothing (RS), shows a consistent improvement in generating adversarial programs. Differences in ASR, marked in blue, are relative to BASELINE.

2.5.1 Experiments

We evaluate our work in three ways – first, we evaluate the overall performance of the three approaches we propose – AO, JO, and their combination with RS, to find the best sites and perturbations for a given program. Second, we evaluate the sensitivity of two parameters which control our optimizers – the number of iterations they are evaluated on, and the *perturbation strength*(k) of an attacker. Third, we use our formulation to train an adversarially robust SEQ2SEQ model, and evaluate its performance against the attacks we propose.

Overall attack results. Table 2.1 summarizes our overall results. The first row corresponds to the samples not being perturbed at all. The ASR as expected is 0. The 'Random replace' in row 2 corresponds to both z and u being selected at random. This produces no change in the ASR, suggesting that while obfuscation transformations can potentially deceive ML models, any random transformation will have little effect. It is important to have some principled approach to selecting and applying these transformations.

Ramakrishnan et al. [2020] (referred to as BASELINE in Table 2.1) evaluated their work in two settings. In the first setting, they pick 1 site *at random* and optimally perturb it. They refer to this as $\mathcal{Q}_G^1$. We contrast this by selecting an optimal site

through our formulation. We use the same algorithm as theirs to optimally perturb the chosen site *i.e.* to solve for u. This allows us to ablate the effect of incorporating and solving the *site-selection*problem in our formulation. In the second related setting, they pick 5 sites at random and optimally perturb them, which they refer to as $\mathcal{Q}_G^5$. In our setup, $\mathcal{Q}_G^1$ and $\mathcal{Q}_G^5$ are equivalent to setting $k = 1$ and $k = 5$ respectively, and picking random sites in z instead of optimal ones. We run AO for 3 iterations, and JO for 10 iterations.

We find that our formulation consistently outperforms the baseline. For $k = 1$, where the attacker can perturb at most 1 site, both AO and JO provide a 3 point improvement in ASR, with JO marginally performing better than AO. Increasing k improves the ASR across all methods – the attacker now has multiple sites to transform. For $k = 5$, where the attacker has at most 5 sites to transform, we find AO to provide a 6 point improvement in ASR over the baseline, outperforming JO by 1.5 points.

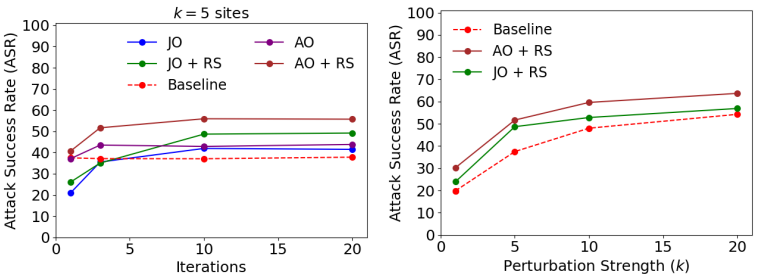
Smoothing the loss function has a marked effect. For $k = 1$, we find smoothing to provide a 10 point increase ($\sim 52\%$ improvement) in ASR over the baseline when applied to AO, while JO+RS provides a 4 point increase. Similarly, for $k = 5$, we find AO+RS to provide a 14 point increase ($\sim 38\%$ improvement), while JO+RS provides a 11 point increase, suggesting the utility of smoothing the landscape to aid optimization.

Overall, we find that accounting for site location in our formulation combined with having a smooth loss function to optimize improves the quality of the generated attacks by nearly 1.5 times over the state-of-the-art attack generation method for programs.

Effect of solver iterations and perturbation strength k. We evaluate the attack performance (ASR) of our proposed approaches against the number of iterations at $k = 5$ (Figure 2-4a). For comparison, we also present the performance of Baseline, which is not sensitive to the number of iterations (consistent with the empirical finding in Ramakrishnan et al. [2020]), implying its least optimality. Without smoothing, JO takes nearly 10 iterations to reach its local optimal value, whereas AO achieves it using only 3 iterations but with improved optimality (in terms of higher ASR than JO). This supports our hypothesis that AO allows for a much more aggressive exploration

of the loss landscape, proving to empirically outperform JO. With smoothing, we find both AO+RS and JO+RS perform better than AO and JO respectively across iterations. We thus recommend using AO+RS with 1-3 iterations as an attack generator to train models that are robust to such adversarial attacks.

In Figure 2-4b, we plot the performance of the best performing methods as we vary the attacker's *perturbation strength*(k). We make two key observations – First, allowing a few sites (< 5) to be perturbed is enough to achieve 80% of the best achievable attack rate. For example, under the AO+RS attack, the ASR is 50 when $k = 5$ and 60 when $k = 20$. From an attacker's perspective, this makes it convenient to effectively attack models of programs without being discovered. Second, we observe that the best performing methods we propose consistently outperform BASELINE across different k. The performance with BASELINE begins to converge only after $k = 10$ sites, suggesting the effectiveness of our attack.



(a) (b)

Figure 2-4: ASRs of our approaches and BASELINE against the number of optimization iterations (2-4a) and *perturbation strength* of an attacker (2-4b).

Improved adversarial training under proposed attack. Adversarial training (AT) Madry et al. [2018b] is a min-max optimization based training method to improve a model's adversarial robustness. In AT, an attack generator is used as the inner maximization oracle to produce perturbed training examples that are then used in the outer minimization for model training. Using AT, we investigate if our proposed attack generation method (AO+RS) yields an improvement in adversarial robustness (over BASELINE)

Train	**Attack** (ASR)		
	BASELINE	AO	AO+RS
No AT	19.87	23.16	30.25
BASELINE	17.99	18.87	19.11
AO+RS	**12.73**	**13.01**	**13.75**

Table 2.2: We employ an AT setup to train SEQ2SEQ with the attack formulation we propose. Lower the ASR, higher the robustness to adversarial attacks. Training under AO+RS attacks provides best robustness results.

when it is used to adversarially train the SEQ2SEQ model. We evaluate AT in three settings – 'No AT', corresponding to the regularly trained SEQ2SEQ model (the one used in all the experiments in Table 2.1), BASELINE - SEQ2SEQ trained under the attack by Ramakrishnan et al. [2020], and AO+RS - SEQ2SEQ trained under our AO+RS attack. We use three attackers on these models – BASELINE and two of our strongest attacks - AO and AO+RS. The row corresponding to 'No AT' is the same as the entries under $k = 1$ in Table 2.1. We find AT with BASELINE improves robustness by ∼11 points under AO+RS, our strongest attack. However, training with AO+RS provides an improvement of ∼16 points. This suggests AO+RS provides better robustness to models when used as an inner maximizer in an AT setting.

2.6 Conclusion

In this paper, we propose a general formulation which mathematically defines an adversarial attack on program source code. We model two key aspects in our formalism – location of the transformation, and the specific choice of transformation. We show that the best attack is generated when both these aspects are optimally chosen. Importantly, we identify that the joint optimization problem we set up which models these two aspects is decomposable via alternating optimization. The nature of decomposition enables us to easily and quickly generate adversarial programs. Moreover, we show that a randomized smoothing strategy can further help the optimizer to find better solutions. Eventually, we conduct extensive experiments from both attack and defense perspectives to demonstrate the improvement of our proposal over the state-of-the-art attack generation method.

Chapter 3

Improving the robustness of code model understanding while retaining model accuracy

Preface. This chapter, in full, is a re-print of **CLAWSAT: Towards Both Robust and Accurate Code Models.** Jia*, J., Srikant*, S., Mitrovska, T., Chang, S., Gan, C., Liu, S., and O'Reilly, U.M. (2023). SANER 2023 [Jia et al., 2022]. Jinghan contributed equally with me as the primary author of this work.

Refer to Section 1.4 to read more about the motivation behind this work, and how it connects to the rest of the chapters presented in this thesis.

3.1 Introduction

Recent progress in large language models for *computer programs* (*i.e.* code) suggests a growing interest in self-supervised learning (**SSL**) methods to learn code models–deep learning models that process and reason about code Kanade et al. [2020b], Chen et al. [2021a,c], Jain et al. [2021a,b]. In these models, a task-agnostic encoder is learned in a pre-training step, typically on an unlabeled corpus. The encoder is appended to another predictive model which is then fine-tuned for a specific downstream task. In particular, contrastive learning (**CL**) based self-supervision Chen et al. [2020], He

et al. [2020] has shown to improve the downstream performance of code reasoning tasks when compared to state-of-the-art task-specific supervised learning (**SL**) models Bui et al. [2021b], Ding et al. [2021], Jain et al. [2021b].

While CL offers to be a promising SSL approach, nearly all the existing works focus on improving the accuracy of code models. Yet, some very recent works Henkel et al. [2022], Yefet et al. [2020], Srikant et al. [2021] showed that trained supervised code models are vulnerable to **code obfuscation** transformations. These works propose **adversarial code**–changing a given code via obfuscation transformations. Such transformed programs retain the functionality of the original program but can fool a trained model at test time. Figure 3-2 shows an example of adversarial code achieved by code obfuscation (more details in Section 3.3). In software engineering, code obfuscation is a commonly-used method to hide code in software projects without altering their functionality Collberg and Thomborson [2002], Linn and Debray [2003], and is consequently a popular choice among malware composers Schrittwieser et al. [2016]. Thus, it is important to study how obfuscation-based adversarial code could affect code model representations learned by SSL. As an example, Schuster et al. Schuster et al. [2021] successfully demonstrate adversarial code attacks on a public code completion model pre-trained on GPT-2, a large language model of code.

Improving the robustness of ML models to adversarial code however comes at a cost–its accuracy (model generalization). Works in vision Goodfellow et al. [2015], Madry et al. [2018b] and text Miyato et al. [2016] have shown how adversarially trained models improve robustness at the cost of model accuracy. While some works Su et al. [2018], Tsipras et al. [2019] provide a theoretical framework for how the robustness of learned models is always at odds with its accuracy, this argument is mainly confined to the SL paradigm and vision applications, and thus remains uninvestigated in SSL for code. While there exist similarities between SSL in vision and code, the discrete and structured nature of inputs, and the additional constraints enforced on views (obfuscated codes) introduce a new set of challenges that have been unexplored by the vision community.

For code models, we ask: *Can pre-trained models be made robust to adversarial attacks? Is it possible to retain this 'pre-trained robustness' when fine-tuning on different tasks? And importantly, is it possible to improve on both the retained generalization and robustness during fine-tuning, thus challenging the popular view of having to trade-off robustness for accuracy gains?*

3.1.1 Overview of proposed approach

We offer two methods that help us improve not only the transfer of robustness from pre-trained models to downstream tasks, but also co-improve fine-tuned accuracy and robustness. The schematic overview of our proposal is shown in Figure 3-1. **First**, we propose a self-supervised pre-training method, contrastive learning with adversarial views (**CLAW**), which leverages adversarially-obfuscating codes as positive views of CL so as to enforce the robustness of learned code representations. We formulate and achieve CLAW through a bi-level optimization method. We show that the representations learned from these pre-trained models yield better robustness transfer to downstream tasks. **Second**, we propose staggered adversarial training (**SAT**) to preserve the robustness learned during pre-training while also learning task-specific generalization and robustness during fine-tuning. We show for the first time that the scheduler of adversarial code generation is adjustable and is a key to benefit both the generalization and robustness of code models.

3.1.2 Contributions

On one hand, we propose CLAW by extending the standard CL framework for code. As a baseline, we compare CLAW to the state-of-the-art standard CL framework for code models - CONTRACODE released by Jain et al. [2021b]. We find CLAW to 'retain' more robustness when compared to **CONTRACODE**. Further, we provide a detailed analysis of understanding this improved performance from the perspective of model interpretability and characteristics of its loss landscape. *On the other hand*, we integrate CLAW with SAT to achieve the eventually fine-tuned **CLAWSAT**

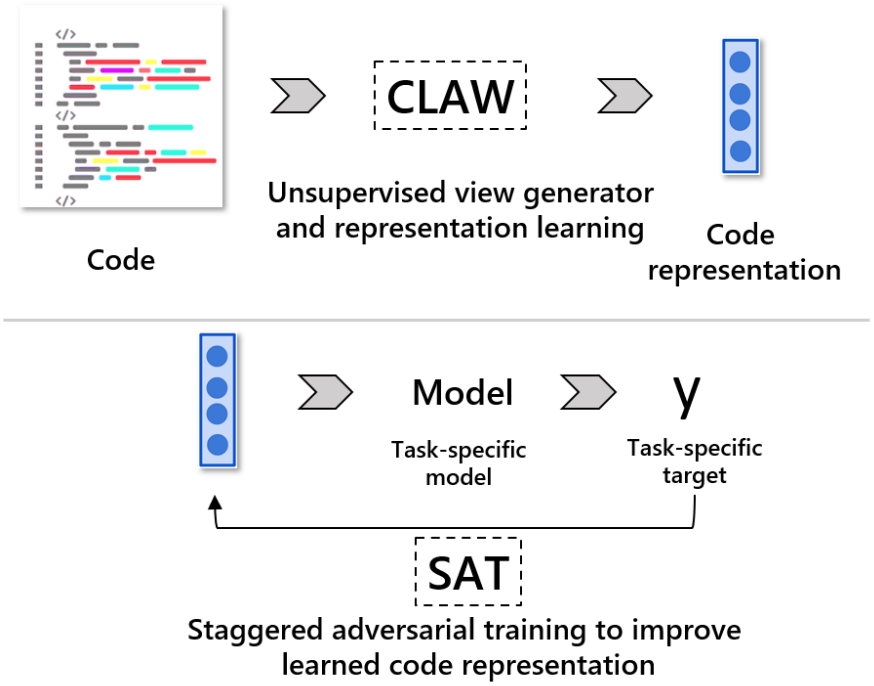


Figure 3-1: Schematic overview. We present **CLAW** - a contrastive learning-based unsupervised method which learns *adversarial views* of the input code to in turn learn accurate and robust representations of the code. We also present **SAT**, a refinement to the adversarial training algorithm proposed by Madry et al. Madry et al. [2018a] which helps retain the task-independent robustness and accuracy learned by CLAW while also learning task-specific accuracy and robustness. We show that CLAWSAT yields better accuracy and robustness when compared to state-of-the-art self-supervised learning models for code.

models. We evaluate three tasks–code summarization, code completion, and code clone detection, in two programming languages - Python and Java, and two different decoder models–LSTMs and transformers. We show that CLAWSAT outperforms CONTRACODE Jain et al. [2021b] by roughly 6% on the summarization task, 2% on the completion task, and 1% on the code clone task in accuracy, and by 9%, 3%, and 1% on robustness respectively. We study the effect of different attack strengths and attack transformations on this performance, and find it to be largely stable across different attack parameters. We will make our code and datasets available publicly, and have uploaded an anonymized copy of it with our submission for an evaluation of this work.

3.2 Related work

Due to a large body of literature on SSL and adversarial robustness, we focus our discussion on those relevant to code models and CL (contrastive learning).

3.2.1 SSL for code

CL-based SSL methods offer a distinct advantage by being able to signal explicit examples where the representations of two codes are expected to be similar. While the method itself is agnostic to the input representation, all the works in CL for code models work on code tokens directly. Existing works Jain et al. [2021b], Chen et al. [2021b], Bui et al. [2021a], Wang et al. [2022] show that CL models for code improve the generalization accuracy of fine-tuned models for different tasks when compared to other pre-training methods like masked language models. Each of these works uses semantics-preserving, random code transformations as positive views in its CL formulation. Such transformations help the pre-trained model learn the equivalence between representations of program elements which do not affect the executed output, such as the choice of variable names, the algorithmic 'approach' used to solve a problem, *etc.*

State-of-the-art CL-based representations generally provide an improvement in the range of 1%-5% points of F1/BLEU/accuracy scores when compared to their fully supervised counterparts and other pre-training methods like transformers, which is significant in the context of the code tasks they evaluate, providing clear evidence for the utility of SSL methods. While these methods improve the generalizability of task-specific models, *none of these works have studied the robustness of these models*, especially with a growing body of works showing the susceptibility of code models to adversarial attacks Srikant et al. [2021], Henkel et al. [2022], Yefet et al. [2020]. By contrast, the adversarial robustness of CL models for image classification has increasingly been studied by the vision community Fan et al. [2021], Jiang et al. [2020], Kim et al. [2020], Gowal et al. [2021]. These works have shown that CL has the potential to offer dual advantages of robustness and generalization. The fundamental

differences in image and code processing, including how adversarial perturbations are defined in these two domains, motivate us to ask if and how the advantages offered by CL can be realized for code models.

3.2.2 Adversarial robustness of code models: Attacks & defenses

Wang and Christodorescu [2019], Quiring et al. [2019], Rabin et al. [2020], Pierazzi et al. [2020] showed that obfuscation transformations made to code can serve as adversarial attacks on code models. Following these works, recent papers Yefet et al. [2020], Henkel et al. [2022] proposed perturbing programs by replacing local variables and inserting print statements with replaceable string arguments. They found optimal replacements using a first-order optimization method, similar to HotFlip Ebrahimi et al. [2017]. Srikant et al. [2021] framed the problem of attacking code models as a problem in combinatorial optimization, unifying the attempts made by prior works. Yefet et al. [2020], Henkel et al. [2022], Bielik and Vechev [2020] and Srikant et al. [2021] also proposed strategies to train code models against adversarial attacks. While Bielik and Vechev [2020] employed a novel formulation to decide if an input is adversarial, the other works employed the adversarial training strategy proposed by Madry et al. [2018b]. Recently,Yang et al. [2022] proposed a black-box attack method to generate adversarial attacks for code, which is different from the white-box setting used in Wang and Christodorescu [2019], Quiring et al. [2019], Rabin et al. [2020], Pierazzi et al. [2020]. In this paper, we only focused on the adversarial robustness of white-box attacks.

Work most relevant to ours. Our work comes closest to ContraCode, the system proposed by Jain et al. [2021b]. While they established the benefit of using CL-based unsupervised representation learning for code, the work neglects the interrelationship between pre-training and fine-tuning in the SSL paradigm, and the consequences of this relationship on both the robustness and generalization of the final model.

3.3 Preliminaries

We begin by providing a brief background on code models, code obfuscation transformations, and SSL-aided predictive modeling for code. We then motivate the problem of how to advance SSL for code models. We study this through the lenses of accuracy and robustness of the learned models.

3.3.1 Code and obfuscation transformations

Let $\mathcal{P}$ denote a *computer program* (*i.e.*, code) which consists of a series of n tokens $\{\mathcal{P}_i\}_{i=1}^{n}$ in the source code domain. For example, Figure 3-2 shows an example code $\mathcal{P}$. Given a vocabulary of tokens (denoted by Ω), each token can be regarded as a one-hot vector of length $|\Omega|$. Here we ignore white spaces and other delimiters when tokenizing.

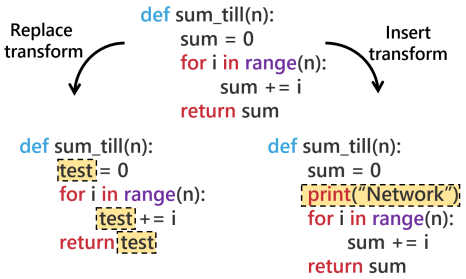


Figure 3-2: Two types of semantics-preserving transformations (obfuscations) can be made to a code to attack code models–replace - where existing code is modified at a *site*—location in the code, or insert - where new lines of code are inserted at a *site*. We select *sites* at random. The specific tokens used in these transformations (`test` and `"Network"` in the the example) can either be a random transformation $t_{\text{rand}}(\cdot)$–a randomly selected token from a pre-defined vocabulary, or can be an adversarial transformation $t_{\text{adv}}(\cdot)$, where the token is obtained from solving a first-order optimization designed to fool the model Srikant et al. [2021], Henkel et al. [2022], Yefet et al. [2020].

Let $t(\cdot)$ denote an *obfuscation transformation*, and $t(\mathcal{P})$ an obfuscated version of $\mathcal{P}$. $t(\mathcal{P})$ is semantically the same as $\mathcal{P}$ while possibly being different syntactically. Following the notations defined by Srikant et al. [2021] and Henkel et al. [2022], we refer to locations or tokens in a code which can be transformed as *sites*. We focus on *replace* and *insert* transformations, where either existing tokens in a source code are

replaced by another token, or new lines of code are inserted in the existing code. For example, in Figure 3-2, the replace transformation modifies the variable `sum` with `test`, while the insert transformation introduces a new line of code `print("Network")`.

Obfuscation transformations have been shown to serve as adversarial examples for code models (see Section 3.2). During an adversarial attack, these transformations are made with the goal to get the resulting transformed code to successfully fool a model's predictions. The transformations at any given *site* in a code, such as the tokens `test`, `"Network"` in Figure 3-2, can be obtained in two ways—by random transformations $t_{\text{rand}}(\cdot)$: they introduce a token sampled at random from Ω, or through adversarial transformations $t_{\text{adv}}(\cdot)$: they solve a first-order optimization problem such that the transformed code maximizes the chances of the model making an incorrect prediction.

Our goal then turns to improve not only *accuracy* (*i.e.* prediction accuracy of properties of code $\mathcal{P}$) but also *robustness* (in terms of prediction accuracy of properties of $t(\mathcal{P})$, obfuscated transformations of $\mathcal{P}$).

3.3.2 Problem statement

SSL typically includes two learning stages: *self-supervised pre-training* and *supervised fine-tuning*, where the former acquires deep representations of input data, and the latter uses these learned features to build a supervised predictor specific to a downstream task, *e.g.* code summarization Alon et al. [2018b] as considered in our experiments. In the pre-training phase, let $\boldsymbol{\theta}$ denote a feature-acquisition model (trained over unlabeled data), and $\ell(\boldsymbol{\theta})$ denote a pre-training loss, *e.g.* the normalized temperature-scaled cross-entropy (NT-Xent) loss used in CL Wu et al. [2018], Chen et al. [2020], He et al. [2020]. In the supervised fine-tuning phase, let $\boldsymbol{\theta}_{\text{ft}}$ denote the prediction head appended to the representation network $\boldsymbol{\theta}$, and $\ell_{\text{ft}}(\boldsymbol{\theta}_{\text{ft}} \circ \boldsymbol{\theta})$ denote a task-specific fine-tuning loss seen as a function of the entire model $\boldsymbol{\theta}_{\text{ft}} \circ \boldsymbol{\theta}$, where $\circ$ denotes model composition. Fine-tuning is performed over labeled data. The SSL pipeline can then

be summarized as

$$\begin{aligned} \text{Pre-training:} \quad & \boldsymbol{\theta}_{\text{pre}} = \arg\min_{\boldsymbol{\theta}} \ \ell(\boldsymbol{\theta}), \\ \text{(Full) Fine-tuning:} \quad & \underset{\boldsymbol{\theta}_{\text{ft}}, \boldsymbol{\theta}}{\text{minimize}} \ \ell_{\text{ft}}(\boldsymbol{\theta}_{\text{ft}} \circ \boldsymbol{\theta}), \\ & \text{with initialization } \boldsymbol{\theta} = \boldsymbol{\theta}_{\text{pre}}. \end{aligned} \tag{3.1}$$

We remark that if we fix $\boldsymbol{\theta} = \boldsymbol{\theta}_{\text{pre}}$ in (3.1) during fine-tuning, then the resulting scheme is called *partial fine-tuning* (**PF**), which only learns the prediction head $\boldsymbol{\theta}_{\text{ft}}$. Based on (3.1) for code, this work tackles the following research questions:

> *(Q1) How to design a self-supervised pre-training scheme to acquire $\boldsymbol{\theta}_{\text{pre}}$ that is robust to obfuscating codes?*
>
> *(Q2) How to design a supervised fine-tuning scheme that can not only preserve the generalization and robustness abilities gained from pre-training but also achieve new improvements via task-driven supervised learning?*

3.4 Method

In this section, we will study the above (Q1)-(Q2) in-depth. To address (Q1), we will develop a new pre-training method, termed CLAW, which integrates CL with adversarial views of codes. The rationale is that promoting the invariance of representations to possible adversarial candidates should then likely improve the robustness of models fine-tuned on these representations. To answer (Q2), we will a novel fine-tuning method, termed staggered adversarial training (SAT), which can balance the supervised fine-tuning with unsupervised pre-training. The rationale is that the supervised fine-tuning overrides pre-trained data representations and hardly retains the robustness and generalization benefits achieved during pre-training. We will show that the interplay between pre-training and fine-tuning should be carefully studied for robustness-generalization co-improvement in SSL for code.

3.4.1 CLAW: CL with adversarial codes

CL Chen et al. [2020], He et al. [2020] proposes to first construct 'positive' example pairs (*i.e.*, original data paired with its transformations or 'views'), and then maximize agreement between them while contrasting with the rest of the data (termed 'negatives'). In programming languages, code obfuscation transformations naturally serve as view generators of an input code. While we reuse the same set of transformations (that are applicable to Python and Java programs) employed by the prior work Jain et al. [2021b], we generate transformed views of the code differently. Jain et al. [2021b] use random views in their CL setup, which select and apply a transformation at random from the set of permissible transformations. We generate worst-case, optimization-based adversarial codes Henkel et al. [2022], Srikant et al. [2021] resulting in 'adversarial views'.

Infusing the original code, its random view, and its adversarial view into CL, we obtain a *three-view* positive tuple, denoted by $(\mathcal{P}, t_{\text{rand}}(\mathcal{P}), t_{\text{adv}}(\mathcal{P}))$. Since the

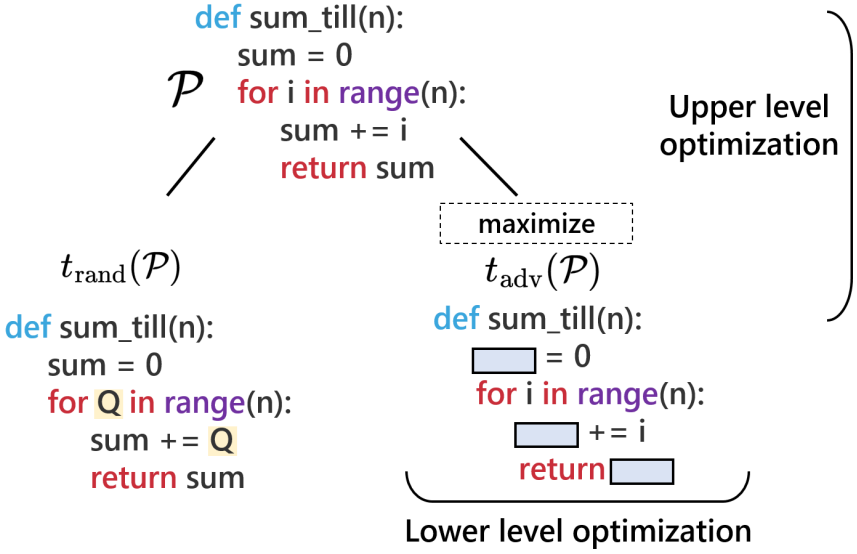


Figure 3-3: During pre-training, we propose **CLAW** containing two optimization problems: (1) to learn invariant code representations by minimizing the representation distances of a code ($\mathcal{P}$) from all its views ($t_{\text{rand}}(\mathcal{P})$, $t_{\text{adv}}(\mathcal{P})$) via CL, and (2) to generate an adversarial code $t_{\text{adv}}(\mathcal{P})$ ('hard' positive example) by maximizing its representation distance from $\mathcal{P}$. In the example, this requires solving for a replacement token at the randomly selected *site* marked as ▣.

generation of adversarial code (the right tokens for a given site) is in itself an additional optimization task, we leverage a bi-level optimization (BLO) framework Liu et al. [2021], Zhang et al. [2022] and define optimization problems at two levels: the upper-level problem aims to solve the multi-view CL, while the lower-level problem aims to solve adversarial code generation (see an illustration in Figure 3-3). This results in the following formulation for our proposed approach CLAW:

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{minimize}} \;\; & \mathbb{E}_{\mathcal{P},t_{\text{rand}}}[\ell_{\text{NT-Xent}}(\boldsymbol{\theta};\mathcal{P},t_{\text{rand}}(\mathcal{P}))]+ \\ & \underbrace{\mathbb{E}_{\mathcal{P}}[\ell_{\text{NT-Xent}}(\boldsymbol{\theta};t_{\text{rand}}(\mathcal{P}),t_{\text{adv}}(\mathcal{P}))]}_{\text{Upper level: Multi-view CL}} \\ \text{subject to} \;\; & t_{\text{adv}}(\mathcal{P}) = \underbrace{\arg\max_{\mathcal{P}'} \ell_{\text{NT-Xent}}(\boldsymbol{\theta};\mathcal{P},\mathcal{P}')}_{\text{Lower level: Adversarial code generation}} , \end{aligned} \tag{3.2}$$

where the three-view objective function is constructed by NT-Xent losses applied to two positive pairs $(\mathcal{P}, t_{\text{rand}}(\mathcal{P}))$ and $(t_{\text{rand}}(\mathcal{P}), t_{\text{adv}}(\mathcal{P}))$, respectively. The first positive pair is to gain the generalizable code representation by promoting the representation invariance across the original view $\mathcal{P}$ and its (benign) randomly-obfuscating view t_{rand}, as suggested in Jain et al. [2021b]. The second positive pair is to enforce the adversary-resilient code representation by promoting the representation invariance across the benign code t_{rand} and the adversarial code $t_{\text{adv}}(\mathcal{P})$. Given two codes $\mathcal{P}_1$ and $\mathcal{P}_2$, the specific form of $\ell_{\text{NT-Xent}}$ is given by as follows:

$$\ell_{\text{NT-Xent}} = -\tfrac{1}{2}\sum_{i=1}^{2} \log \frac{\exp\left(\text{sim}(\mathbf{z}_1(\boldsymbol{\theta}),\mathbf{z}_2(\boldsymbol{\theta}))/t\right)}{\sum_{k\in\mathcal{N}(i)},\exp\left(\text{sim}(\mathbf{z}_i(\boldsymbol{\theta}),\mathbf{z}_k(\boldsymbol{\theta}))/t\right)} \tag{3.3}$$

where $\mathbf{z}_i$ denotes the feature representation of the input code $\mathcal{P}_i$ achieved through the representation network $\boldsymbol{\theta}$, $\text{sim}(\mathbf{z}_i, \mathbf{z}_j)$ denotes the cosine similarity between two feature representations $\mathbf{z}_i$ and $\mathbf{z}_j$, $t > 0$ is a temperature parameter, and $\mathcal{N}(i)$ is the set of batch data except the data sample i Chen et al. [2020].

To solve the BLO problem (3.2), we apply an alternating optimization method Bezdek and Hathaway [2003], Liu et al. [2021]. Specifically, by fixing the representation network $\boldsymbol{\theta}$, the lower-level adversarial code generation is accomplished using first-

order gradient descent following Srikant et al. [2021], Henkel et al. [2022]. Given the generated adversarial code $t_{\text{adv}}(\mathcal{P})$, we then in turn solve the upper-level CL problem. The above procedure is alternatively executed for every data batch.

Adversarial view is beneficial to representation learning. To highlight the effectiveness of incorporating adversarial codes in CL at the pre-training phase, the rows 'ContraCode-PF' and 'Claw-PF' of Table 3.1 demonstrate a warm-up experiment by comparing the performance of the proposed pre-training method Claw with that of the baseline approach ContraCode Jain et al. [2021b]. To precisely characterize the effect of the learned representations on code model generalization and robustness, we partially fine-tune (PF) a seq2seq model on the downstream task of summarizing code (details in Section 3.5) in Python (**SummaryPy**) and Java (**SummaryJava**) by fixing the set of weights learned during pre-training. **Gen-F1** and **Rob-F1** are the generalization F1-scores and the robust F1-scores, *i.e.*, F1 scores of the model when attacked with adversarial codes. Partially fine-tuning these models allows us to study the sole contribution of the pre-training method in the learned robustness and generalization of the model. As we can see, the partially fine-tuned Claw model (termed Claw-PF) outperforms the baseline ContraCode-PF, evidenced by the substantial robustness improvement (4.38% increase in Rob-F1 in SummaryPy) as well as lossless or better generalization performance (1.58% increase in Gen-F1 scores on SummaryJava). It is worth noting that adversarial codes serve as 'hard' positive examples in the representation space (given by maximizing the representation distance between $\mathcal{P}$ and its perturbed variant $\mathcal{P}'$ in the lower optimization level of Claw, (3.2)). The benefit of hard positive examples in improving generalization has also been seen in vision Chuang et al. [2020], Wang et al. [2020a], Fan et al. [2021].

3.4.2 SAT: Staggered adversarial training for fine-tuning

As shown in the previous section, an appropriate pre-training method can improve the quality of learned deep representations, which help improve the robustness and accuracy of a code model. However, the state of the model present at the end of the pre-training phase may no longer hold after supervised fine-tuning. That is because

Model	Partial fine-tuning			
	SummaryPy		SummaryJava	
	Gen-F1	Rob-F1	Gen-F1	Rob-F1
ContraCode-PF	25.46	15.47	20.92	16.63
Claw-PF	25.45	19.05	22.50	17.14
	Full fine-tuning			
ContraCode-ST	36.28	28.97	41.37	33.01
Claw-ST	36.57	29.97	41.23	32.53
ContraCode-AT	32.80	32.39	38.67	35.91
Claw-AT	32.97	32.65	38.86	36.10

Table 3.1: Partially fine-tuned (PF) models show that Claw improves robustness. Standard training (ST) yields better generalization than adversarial training (AT) while the latter provides better robustness.

supervised learning (trained on labeled data vs. unlabeled data in representation learning) may significantly alter the characteristics of the learned representations. Thus, a desirable fine-tuning scheme should be able to yield accuracy and robustness improvements *complementary* to the representation benefits provided by pre-training. Towards this goal, we posit that fine-tuning should *not* be designed in a way which merely optimizes a single performance metric–either accuracy or robustness.

To justify this hypothesis, we consider two extreme cases during fine-tuning: *(i)* standard training (ST)-based FF, and *(ii)* adversarial training (AT)-based FF Madry et al. [2018b]. ST is essentially the same setup as fully supervised training with the only difference being in the set of initial parameters of the model. This setup optimizes improving a model's generalization ability. On the other hand, AT optimizes improving the model's adversarial robustness.

The *last four rows of Table 3.1* present the performance of these two extreme fine-tuning cases applied to the pre-trained models provided by ContraCode Jain et al. [2021b] and Claw, respectively. As we can see, when either ST or AT is used, different pre-training methods (ContraCode and Claw) lead to nearly the same generalization and robustness performance. This shows that fine-tuning, when aggressively optimizing one particular performance metric, could override the benefits

achieved during pre-training. To this end, we propose staggered AT (SAT), a hybrid of ST and AT by adjusting the time instances (in terms of epoch numbers) at which adversarial codes are generated (see Algorithm 1). SAT involves two key steps—

Algorithm 1: Staggered Adversarial Training (SAT)

Input:model $\mathcal{M} = \{\boldsymbol{\theta}_{\text{ft}}, \boldsymbol{\theta}\}$, attack frequency τ
for each epoch e **do**
    for each data batch $\mathcal{B}_i$ **do**
        1. Train $\mathcal{M}$ by updating $\boldsymbol{\theta}_{\text{ft}} \circ \boldsymbol{\theta}$
    if e mod $\tau = 0$ **then**
        for each data batch $\mathcal{B}_i$ **do**
            2. Attack $\mathcal{M}$ by finding adversarial codes $\mathcal{B'}_i$
            3. Retrain $\mathcal{M}$ on $\mathcal{B'}_i$

training a model $\mathcal{M} := \{\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{ft}}\}$ on a batch $\mathcal{B}$ of data (step 1), and attacking the learned model at a staggered frequency τ (steps 2-3). *Different from* AT, adversarial code is not generated in every data batch. Instead, in SAT, we propose reducing the frequency of adversarial learning. Accordingly, adversarial code generation occurs at the frequency of each epoch or at every few epochs. This ensures the model parameters retain as much of the attributes from pre-training while also learning task-specific generalization and robustness. In SAT, the model is finetuned using ($\mathcal{B}_i$ + $\mathcal{B}'_i$) where $\mathcal{B}'_i$ refers to the generated adversarial code corresponding to $\mathcal{B}_i$. Eventually, by combining the proposed pre-training scheme CLAW with the fine-tuning scheme SAT, we term the resulting SSL framework for code as CLAWSAT.

3.5 Experiment Setup

We describe the following aspects of our experiment setup - the task, dataset, and the details of the model.

Task, dataset, error metrics. We evaluate our algorithm on **four** tasks: **(1)** code summarization Alon et al. [2018b, 2019c], Allamanis et al. [2018d], Wang et al. [2020d], David et al. [2020], Jain et al. [2021b] in Python, **(2)** code summarization in Java (generates English description for given code snippet), **(3)** code completion in Python Lu et al. [2021a] (generates the next six tokens for a given code snippet), and **(4)**

	Model	SummaryPy		SummaryJava		CompletePy		CloneJava	
		Gen-F1	Rob-F1	Gen-F1	Rob-F1	Gen-F1	Rob-F1	Gen-F1	Rob-F1
M_1	Supervised learning	$33.33_{\pm 0.17}$	$26.16_{\pm 0.31}$	$38.42_{\pm 0.25}$	$29.89_{\pm 0.27}$	$56.72_{\pm 0.22}$	$53.89_{\pm 0.26}$	$67.20_{\pm 0.11}$	$64.35_{\pm 0.15}$
M_2	M_1-AT Henkel et al. [2022]	$33.03_{\pm 0.21}$	$32.20_{\pm 0.26}$	$37.81_{\pm 0.23}$	$34.86_{\pm 0.29}$	$55.40_{\pm 0.26}$	$55.34_{\pm 0.35}$	$66.12_{\pm 0.13}$	$65.84_{\pm 0.17}$
M_3	ContraCode Jain et al. [2021b]	$36.28_{\pm 0.18}$	$28.97_{\pm 0.27}$	$41.37_{\pm 0.14}$	$33.01_{\pm 0.25}$	$57.70_{\pm 0.23}$	$54.83_{\pm 0.31}$	$69.25_{\pm 0.09}$	$68.86_{\pm 0.13}$
M_4	Claw-ST	$36.57_{\pm 0.20}$	$29.97_{\pm 0.22}$	$41.23_{\pm 0.17}$	$33.53_{\pm 0.22}$	$57.65_{\pm 0.25}$	$54.75_{\pm 0.31}$	$69.63_{\pm 0.09}$	$68.95_{\pm 0.15}$
M_5	Claw-AT	$32.97_{\pm 0.15}$	$32.65_{\pm 0.17}$	$38.86_{\pm 0.23}$	$36.10_{\pm 0.31}$	$57.44_{\pm 0.26}$	$57.12_{\pm 0.29}$	$69.40_{\pm 0.16}$	$69.00_{\pm 0.14}$
M_6	**ClawSAT (ours)**	$\mathbf{42.12}_{\pm 0.19}$	$\mathbf{40.70}_{\pm 0.23}$	$\mathbf{41.77}_{\pm 0.27}$	$\mathbf{38.80}_{\pm 0.33}$	$\mathbf{58.80}_{\pm 0.24}$	$\mathbf{57.21}_{\pm 0.28}$	$\mathbf{69.73}_{\pm 0.10}$	$\mathbf{69.25}_{\pm 0.13}$

Table 3.2: **Overall performance of ClawSAT**: We evaluate our models in two settings: standard training (ST) and adversarial training (AT) by Madry et al. [2018b]. For each of the four tasks: code summarization: SummaryPy, SummaryJava, code completion: CompletePy, and code clone detection: CloneJava, we report the model's generalization F1-score (Gen-F1) and the robustness F1 (Rob-F1)–the generalization F1 when the model is adversarially attacked. M_2 corresponds to an adversarially trained (AT) version of the supervised model M_1, first introduced in Henkel et al. [2022]. M_4 and M_5 are two variants of ClawSAT (M_6): one integrates Claw with standard training (ST), and the other integrates Claw with the adversarial training (AT) Madry et al. [2018b]. The result $a_{\pm b}$ represents mean a and standard deviation b, calculated over 5 random trials.

code clone detection Wang et al. [2020c] in Java (classifies whether a pair of code snippets are clones of each other). For models evaluated in Python, we pre-train on the Py-CSN dataset Husain et al. [2019], containing $\sim$ 500K methods, and fine-tune on the Py150 dataset Raychev et al. [2016b], containing $\sim$ 200K methods. For Java, we pre-train on the Java-CSN dataset Husain et al. [2019] containing $\sim$ 600K and fine-tune on the Java-C2S Alon et al. [2018b] dataset containing $\sim$ 500K methods. We use the F1-score $\in [0, 100]$ to measure the performance of all our models, consistent with all the related works, including Jain et al. [2021b]. A higher value indicates that the model generalizes better to the task. While these F1-scores are correlated to BLEU scores, they directly account for token-wise mis-predictions. The F1 scores are computed following Allamanis et al. [2016], Alon et al. [2019b] Specifically, for each of our models, we reuse the two F1 scores: **Gen-F1** - the model's generalization performance on a task, and **Rob-F1** - the model's performance on the task when semantics-preserving, adversarially-transformed obfuscated codes are input to it; see details below.

Adversarial attacks, attack strength, code transformations. When attacking code models, we use the formulation by Srikant et al. [2021] to define the strength of

an attack. Specifically, selecting a larger number of *sites* in a code—locations or tokens in a code which can be *transformed* to produce an adversarial outcome—corresponds to a stronger adversarial attack, since this allows multiple changes to be made to the code. Also, we follow Srikant et al. [2021], Henkel et al. [2022] to specify the set of code transformations: replace (renaming local variables, renaming function parameters, renaming object fields, replacing boolean literals) and insert (inserting print statements, adding dead code).

In our setup, we can leverage the attack strength at three stages–during pre-training (using adversarially attacked code as views), when fine-tuning with adversarial training AT Madry et al. [2018b] or SAT, and when evaluating robustness on an unseen test set. Unless specified otherwise, we pre-train on one *site*, attack one *site* in each iteration of SAT, and attack one *site* during evaluation. In Table 3.6 ,we analyze the effect of varying the number of sites at each of these stages. We apply the first-order optimization method proposed in Henkel et al. [2022] to generate adversarial codes. To adversarially train these models during fine-tuning, we employ either AT Madry et al. [2018b] or our proposed SAT for code.

Baselines. We compare ClawSAT to **three** baselines. **(1)** A supervised model (model $\mathbf{M_1}$ in Table 3.2) - Pre-training has no effect on a fully supervised model. **(2)** Adversarially trained supervised model ($\mathbf{M_2}$) on top of M_1 - we use the AT setup first proposed by Henkel et al. [2022], which in turn employs the setup from Madry et al. [2018b]. Due to the characteristics of AT, we expect to see an improvement in its Rob-F1 as compared to $\mathbf{M}_1$ but a decrease in Gen-F1. **(3)** The ContraCode model ($\mathbf{M_3}$) from Jain et al. [2021b]. ContraCode reflects the state-of-the-art in pre-training methods as it outperforms other pre-training models like BERT-based models and GPT3-Codex. Hence, we do not compare ourselves again to other pre-training models.

Models. For the summarization and completion tasks, we experiment with two seq2seq architectures–LSTMs and transformers. For the detection task, we use a fully connected linear layer as a decoder. The decoders are trained to predict the task (generating English sentence summaries, generating code completions, flagging

code clones respectively) in both the fine-tuned and standard training settings. When fine-tuning, we use the learned encoders from the pre-trained models. For the summarization and completion tasks, we report all our results on the LSTM decoder (Table 3.2), and compare the performance of transformers in our ablation study. The LSTM encoder has 2 layers across all experiments. In the code summarization tasks, there exists another two-layer decoder added to the encoder to generate the summary of the programming language. In the code completion task, we also add a two-layer decoder to generate the code snippets. In the code clone detection task, we add one linear layer to map the data representations to the data labels. As for the transformer architecture, The transformer encoder has 6 layers and the transformer decoder has another 6 layers to generate the programming language summary.

Hyperparameter setup. We optimize model parameters using Adam with linear learning rate warm-up. For the bidirectional LSTM encoders, the maximum learning rate for ContraCode and Claw is 10^{-4}, and then is decayed accordingly. For the transformer-type encoder, the maximum learning rate is 10^{-4} for ContraCode and 10^{-5} for Claw. For different downstream tasks, we optimize the parameters using Adam with step-wise learning rate decay. The maximum learning rates of ST, AT, SAT are 10^{-3} on code summarization and code completion tasks, and 10^{-4} for code clone detection tasks. For downstream tasks using transformer, the learning rates of ST, AT, SAT are 10^{-4}. All of the downstream tasks are finetuned for 10 epochs with practical convergence, and we utilize a validation dataset to pick the best-performed model. All of the experiments are conducted on 4 Tesla V100 GPU with 16 GB memory.

3.6 Experiment Results

We summarize the overall performance of ClawSAT and follow that up with multiple additional analyses and ablations to better understand our model's performance.

3.6.1 Overall performance

We evaluate the different pre-training and fine-tuning strategies we consider in Section 3.4. The accuracy and robustness F1-scores of the different models are shown in Table 3.2. Models $\mathbf{M}_1$-$\mathbf{M}_3$ are the baselines described in the previous section. Model $\mathbf{M}_4$ pertains to using a CLAW encoder with standard training (ST) for the downstream task. Model $\mathbf{M}_5$ pertains to using a CLAW encoder with standard adversarial training (AT) for the downstream task. And finally, $\mathbf{M}_6$ pertains to using SAT for the downstream task with a CLAW encoder. We observe the following:

First, from the perspective of accuracy, Table 3.2 shows that our proposal CLAWSAT (model $\mathbf{M}_6$) outperforms all the baselines on GEN-F1. Particularly, CLAWSAT achieves nearly 8% accuracy improvement over CONTRACODE ($\mathbf{M}_3$) on SUMMARYPY. **Second**, we see that CLAWSAT can substantially help improve the robustness (measured by ROB-F1) of a downstream model. Compared to $\mathbf{M}_3$, we see an improvement of 14.7% for SUMMARYPY, and 5.8% for SUMMARYJAVA. Similarly, the gain in robustness is much more substantial in COMPLETEPY than in accuracy. CLONEJAVA is the simplest of the four tasks, and we see comparable accuracies across all the models we evaluate. **Additionally**, we adversarially train baselines as well (models $\mathbf{M}_2$ and $\mathbf{M}_5$ respectively) and compare their robustness scores to ours ($\mathbf{M}_6$). We make two observations–(a) As is expected with AT, we notice a drop in these models' accuracy–the GEN-F1 scores of $\mathbf{M}_5$ is lower than that of its standard-training counterpart $\mathbf{M}_4$ (the same trend is observed between $\mathbf{M}_2$ and $\mathbf{M}_1$ as well). (b) While AT-based fine-tuning provides an expected improvement in robustness over the other baselines that use ST-based fine-tuning, the ROB-F1 they achieve is still much lower than our model. This is because the robustness gain at the pre-training phase was overridden by AT at the fine-tuning phase.

In summary, **Table 3.2** shows that the proposed CLAWSAT allows us to learn task-specific accuracy and robustness while preserving these attributes learned during pre-training.

In what follows, we analyze the performance of CLAW and SAT separately from different perspectives.

3.6.2 Why is CLAW effective? A model landscape perspective

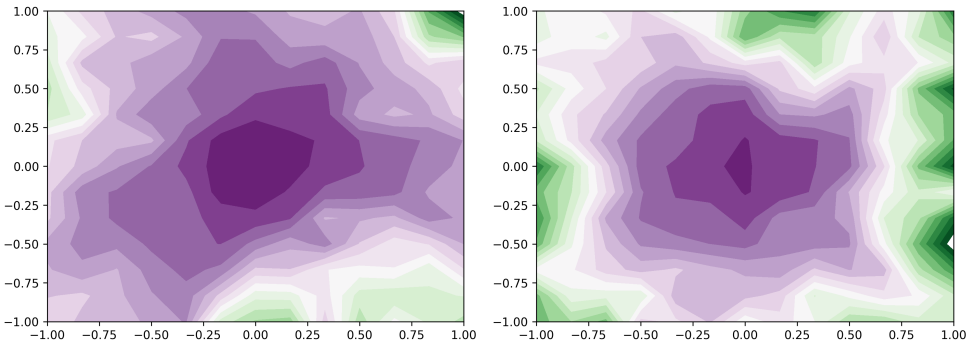


Figure 3-4: Loss landscapes: CLAW (L), and CONTRACODE (R), the **X** and **Y** axes represent the directional coefficients α and β in (3.4).

Liu *et. al.* Liu et al. [2020a] show that the generalization benefit of an approach in the'pre-training + fine-tuning' paradigm can be deduced by the flatness of the loss landscape of the pre-trained model. This would then let fine-tuning to force the optimization for fine-tuning to stay in a certain neighborhood of the pre-trained model of high quality.

To plot the loss landscapes, we follow the procedure from Li *et. al.* Li et al. [2018a] by plotting

$$f(\alpha, \beta) = \ell(\theta^\star + \alpha\delta + \beta\eta) \tag{3.4}$$

where δ and η are two random direction vectors in the parameters' subspace, and $\theta^\star$ is the parameters of a model. We average the supervised loss from the partially fine-tuned models from 640 randomly selected samples in our test set (out of 10000, 6.4%).

```
def _makeOne(self,discriminator=None
    ,family=None):
    from ..index import AllowedIndex
    index = AllowedIndex(discriminator,family=family)
    return index
```

(a) Sample

```
def _makeOne(self,repeat=None
    ,family=None):
    from ..index import AllowedIndex
    index = AllowedIndex(repeat,family=family)
    return index
```

(b) Adversarially perturbed version of sample (a1)

```
def _makeOne(self,discriminator=None
    family=None,action_Mode=None):
    from ..indexes import FieldIndex
    return FieldIndex(discriminator,family,action_mode)
```

(c) EBE similar to (a) - Claw

```
def _makeOne(self,discriminator=None
    family=None,action_Mode=None):
    from ..indexes import FieldIndex
    return FieldIndex(discriminator,family,action_mode)
```

(d) EBE similar to (a) - ContraCode

```
def _makeOne(self,discriminator=None
    family=None,action_Mode=None):
    from ..indexes import FieldIndex
    return FieldIndex(discriminator,family,action_mode)
```

(e) EBE similar to (b) - Claw

```
def buildIndex(self,l):
    index = self.mIndex()
    for strat, end, value in self.l:
        index.add(strat,end)
    return index
```

(f) EBE similar to (b) - ContraCode

Figure 3-5: Explanation-by-example to demonstrate the robustness benefits of Claw. **(a)** Sample program from the test set **(b)** Adversarially perturbed variant of the sample program. **(c-d)** Examples closest to the sample program (a) when using Claw and ContraCode. **(e-f)** Examples closest to the perturbed variant (b) when using Claw and ContraCode.

Results of **Figure 3-4** confirm the **flatness** of the loss landscape in Claw when compared to ContraCode, implying a better transfer of generalizability by Claw.

Next, we show another way to justify the flatness merit of CLAW's loss landscape. The key idea is to track the deviations of the weights of the pre-trained encoders in the fine-tuned setting, as inspired by Liu et al. [2020a]. Specifically, let $\boldsymbol{\theta}_{\text{pre}}$ and $\boldsymbol{\theta}'_{\text{pre}}$ denote the weights of the representation model $\boldsymbol{\theta}$ pre-trained by CLAW and the fine-tuned weights obtained by using different fine-tuning methods, respectively. It was shown in Liu et al. [2020a] that the generalization benefit of an approach in the'pre-training + fine-tuning' paradigm can be deduced by the deviation between the fine-tuned weights $\boldsymbol{\theta}'_{\text{pre}}$ and the pre-trained weights $\boldsymbol{\theta}_{\text{pre}}$. This would then let fine-tuning to force the optimization of $\boldsymbol{\theta}'_{\text{pre}}$ to stay in a certain neighborhood of $\boldsymbol{\theta}_{\text{pre}}$. We have already shown that CLAW will lead to a flatter loss landscape compared to CONTRACODEpreviously. Here we computed the Frobenius norm $||\boldsymbol{\theta}'_{\text{pre}} - \boldsymbol{\theta}_{\text{pre}}||_{\text{F}}$ as a proxy to justify the generalization benefit following Liu et al. [2020a].

$\boldsymbol{\theta}'_{\text{pre}}$	$\|\|\boldsymbol{\theta}'_{\text{pre}} - \boldsymbol{\theta}_{\text{pre}}\|\|_{\mathbf{F}}$
CLAW-ST	207.99
CLAW-AT	323.41
CLAWSAT	232.56
CONTRACODE-ST	218.14
CONTRACODE-AT	345.36
CONTRACODE-SAT	242.70

Table 3.3: Weight difference after finetuning based on different pretraining methods.

Table 3.3 summarizes the aforementioned weight characteristics for the code summarization task. As we can see, the weight deviation corresponding to CLAW is less than that associated with CONTRACODE given a finetuning method.

The results from **Table 3.3** suggest that an encoder pretrained using CLAW transfers better than that using CONTRACODE.

3.6.3 Interpretability of learned code representations

We evaluate the robustness benefit of CLAW through the lens of (input-level) model explanation. Following the observations from Jeyakumar *et. al.* Jeyakumar et al.

[2020] on probing models locally, we investigate CLAW and CONTRACODE using a training data-based model explanation method: explanation-by-example (EBE) Kim et al. [2016]. The core idea is to leverage train-time data to explain test-time data by matching their respective representations. If the pre-trained models are robust, adversarially perturbing the samples should not alter their representations and thus should be mapped to the same set of closest training examples that were found without perturbations.

Based on EBE, we sample 100 code snippets $\{\mathcal{P}_i^{\text{test}}\}_{i=1}^{100}$ at random from our test dataset, and find the closest samples $\{\mathcal{P}_{\text{CLAW}}^{\text{train}}\}$ and $\{\mathcal{P}_{\text{CONTRACODE}}^{\text{train}}\}$ in the training dataset using the EBE method, based on representations produced by $\boldsymbol{\theta}_{\text{CLAW}}$ and $\boldsymbol{\theta}_{\text{CONTRACODE}}$ respectively. We find that 68% of the representations from CLAW match their original codes in $\{\mathcal{P}_{\text{CLAW}}^{\text{train}}\}$ while 57% of CONTRACODE representations match their original codes in $\{\mathcal{P}_{\text{CONTRACODE}}^{\text{train}}\}$. **The above results** suggest that the learned representations by CLAW are more adversarially robust than CONTRACODE.

In what follows, we peer into the EBE method's results with an example below.

• **A sample program from the test-set:**

```
def __init__(self,helper_name):
    self.helper_name = helper_name
    self.cheeks = []
```

• **Adversarially perturbed variant of the sample program:** The adversarial attack algorithm replaces the method argument `helper` with `edges`.

```
def __init__(self,edges):
    self.helper_name = edges
    self.cheeks = []
```

• **EBE sample in the train-set closest to the sample program when using CONTRACODE or CLAW:** This is the example whose CONTRACODE or CLAW

representation (encoder trained by ContraCode or Claw) is closest to the representation of the sample program. We can find that they have the same functionality.

```
def __init__(self,name):
    self.name = name
    self.warning = []
```

• **EBE sample in the train-set closest to the perturbed variants of sample program from ContraCode:** We can observe that the closest program of the perturbed program in the training dataset is different from that of the original program. The new closest program has different functionality from the previous test sample program.

```
def setUp(self):
    super(ApiCallHandlerRegressionTest,self).setUp()
    self.checks = []
```

• **EBE sample in the training-set closest to the perturbed sample program from Claw:** This example pertains to the representation that is closest to the representation of the sample program computed by an encoder trained by Claw. We see that despite comparing it to a perturbed sample's representation, the example found by EBE corresponds to the unpertrubed sample program, suggesting the robustness of Claw over ContraCode.

```
def __init__(self,name):
    self.name = name
    self.warning = []
```

We also provide more examples in Figure 3-5. Figure 3-5.a shows a sample Python program and Figure 3-5.b shows its respective adversarially perturbed variant. The closest training programs in the training set mapped to the representations before perturbations are shown in Figures 3-5.c and 3-5.d; and those mapped to the

representations after perturbations are shown in Figures 3-5.e and 3-5.f.

As shown in **Figure 3-5**, we find that EBE consistently finds the same training examples for CLAW (Figure 3-5.c and Figure 3-5.e) irrespective of the adversarial perturbations made to the sample program, confirming its enhanced robustness.

3.6.4 SAT enables generalization-robustness sweet spot

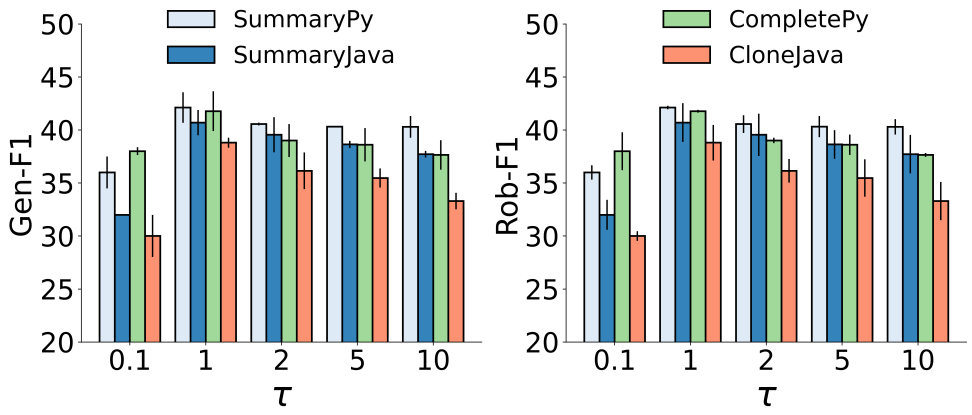


Figure 3-6: Effect of different update schedules (τ, see Algorithm 1) on GEN-F1 and ROB-F1.

Figure 3-6.(A) shows the results from our experiments on the code summarization task where we vary τ, the frequency of attacking the code model during SAT (see Algorithm 1).

We generate adversarial code tokens every τ^{th} epoch, where we vary τ from less than 1 (corresponds to an update occurring at every batch within an epoch; this is the AT algorithm from Madry et al. [2018b]) to 10. The X-axis shows this frequency. We plot both GEN-F1 (left) and ROB-F1 (right) of the adversarially trained model when varying n. Across the four tasks we evaluate in this work, We find that a sweet spot exists in a less frequent epoch-wise schedule, associated with CLAWSAT ($\mathbf{M}_6$, which corresponds to $\tau = 1$), which improves both GEN-F1 and ROB-F1 over $\mathbf{M}_5$ (which corresponds to $\tau = 0.1$).

Results of **Figure 3-6** validate our hypothesis of being able to retain the robustness learned during pre-training while updating the model just enough during fine-tuning to 'learn' new robustness while also learning the downstream task.

3.6.5 ClawSAT on a different architecture

		SummaryPy	
	Model	**Gen-F1**	**Rob-F1**
M_1	Supervised learning	$32.60_{\pm 0.14}$	$30.09_{\pm 0.21}$
M_2	M_1-AT Henkel et al. [2022]	$31.18_{\pm 0.13}$	$30.66_{\pm 0.23}$
M_3	ContraCode Jain et al. [2021b]	$34.93_{\pm 0.11}$	$32.86_{\pm 0.18}$
M_4	Claw-ST	$35.37_{\pm 0.14}$	$32.31_{\pm 0.20}$
M_5	Claw-AT	$34.23_{\pm 0.19}$	$33.34_{\pm 0.18}$
M_6	**ClawSAT (ours)**	$\mathbf{36.39}_{\pm 0.12}$	$\mathbf{35.53}_{\pm 0.24}$

Table 3.4: Overall performance of ClawSAT on transformer

We further evaluate SAT on a different model architecture. We consider the transformer architecture (6-layer encoder and 6-layer decoder following Jain et al. [2021b]), and observe similar results (see Table 3.4): ClawSAT offers the best accuracy and robustness.

Table 3.4 shows that ClawSAT performs well across multiple encoder architectures.

3.6.6 Extended study to integrate SAT with ContraCode

To further verify the effectiveness of SAT, we employ it in ContraCode—we modify their implementation to introduce a staggered adversarial training schedule. Table 3.5 tabulates its performance. We find that SAT also benefits ContraCode, but the gain is smaller than ClawSAT (M_6, Table 3.2). This justifies the complementary benefits of Claw and SAT.

Model	SummaryPy		SummaryJava		CompletePy		CloneJava	
	Gen-F1	Rob-F1	Gen-F1	Rob-F1	Gen-F1	Rob-F1	Gen-F1	Rob-F1
ContraCode Jain et al. [2021b]	$36.28_{\pm 0.18}$	$28.97_{\pm 0.27}$	$41.37_{\pm 0.14}$	$33.01_{\pm 0.25}$	$57.70_{\pm 0.23}$	$54.83_{\pm 0.31}$	$69.25_{\pm 0.09}$	$68.86_{\pm 0.13}$
ContraCode-AT	$35.88_{\pm 0.20}$	$31.29_{\pm 0.24}$	$38.67_{\pm 0.27}$	$35.91_{\pm 0.30}$	$57.21_{\pm 0.19}$	$56.80_{\pm 0.22}$	$69.20_{\pm 0.13}$	$68.88_{\pm 0.11}$
ContraCode-SAT	$41.01_{\pm 0.20}$	$39.80_{\pm 0.21}$	$41.27_{\pm 0.16}$	$38.14_{\pm 0.24}$	$58.04_{\pm 0.17}$	$57.01_{\pm 0.30}$	$69.47_{\pm 0.10}$	$69.08_{\pm 0.21}$

Table 3.5: Effectiveness of SAT on ContraCode

Table 3.5 shows the complementary benefits of Claw and SAT on the state-of-the-art SSL method ContraCode as well, demonstrating the effectiveness of the two model-independent techniques we introduce in this work.

3.6.7 Sensitivity of SAT to code transformation and attack strength types.

		Transformations (Gen-F1, Rob-F1)		
		Fine-tuning		
Pre-training		replace	insert	All
	replace	41.51, 39.84	40.49, 34.96	42.32, 40.75
	insert	41.71, 40.38	41.34, 35.24	41.71, 40.38
	All	42.66, 40.54	41.39, 34.91	42.12, 40.70

	Attack strength (Rob-F1)				
	1	**2**	**3**	**4**	**5**
ContraCode	28.97	28.29	27.32	26.24	25.14
ClawSAT	40.70	40.58	39.67	38.90	38.10

Table 3.6: **Performance of ClawSAT at different attack configurations**. We evaluate the sensitivity of our best performing model on (a) different transformation types used during pre-training and fine-tuning (SAT) (b) different attack strengths (number of *sites*) during evaluation.

We evaluate the sensitivity of our best-performing model on differing attack conditions. We consider two factors: (1) Transformation type: we study the effect of the two transformation types–replace and insert, and their combination. (2) Attack strength: we vary the number of *sites*—locations in the codes that can be adversarially transformed.

We summarize our results in Table 3.6. The values a, b in each cell correspond to Gen-F1 and Rob-F1 of ClawSAT respectively.

The results from **Table 3.6** suggest that when using transformations in pre-training or in fine-tuning, it is advisable to use a combination of both replace and insert transformations. When evaluating ClawSAT's robustness against stronger adversarial attacks, we find Rob-F1 consistently outperforms ContraCode in all the configurations we evaluate.

3.7 Conclusion & Discussion

In this work, we aim to achieve the twin goals of improved robustness and generalization in SSL for code, specifically in contrastive learning. We realize this by proposing two improvements–adversarial positive views in contrastive learning, and a staggered AT schedule during fine-tuning. We find that each of these proposals provides substantial improvements in both the generalization and robustness of downstream models; their combination, ClawSAT, provides the best overall performance. Given the growing adoption of SSL-based models for code-related tasks, we believe our work lays out a framework to gain a principled understanding into the working of these models. Future works should study this problem while attempting to also establish a theoretically sound foundation.

When compared to SSL for vision, it seems SSL for codes benefits from milder adversarial training during fine-tuning. Stronger attack methods for code might be needed to explore this phenomenon further. It will also be beneficial to understand how code models respond to perturbations, and to contrast it to our understanding of continuous data perturbations in vision.

Chapter 4

Training code models to understand concurrent programs using program execution traces

Preface. Section 4.3 refers to a thesis I mentored, authored by Teodor Rares Begu: **Modeling concurrency bugs using machine learning.** Rares Begu, T., Srikant, S., and O'Reilly, UM (2020). MIT SuperUROP Thesis [Rares Begu, 2020]. Teodor implemented and refined the idea that I proposed of using a toy language to simulate concurrent threads and evaluating different ML models.

Section 4.4, in full, is a re-print of **RaceInjector: Injecting Races To Evaluate And Learn Dynamic Race Detection Algorithms.** Wang, M., Srikant, S., Samak, M., and O'Reilly, U.M. (2023) [Wang et al., 2023]. The data-driven approach was developed substantially in collaboration with Michael and Malavika. Michael led the development of the system which eventually came to be *RaceInjector*.

Refer to Section 1.4 to read more about the motivation behind this work, and how it connects to the rest of the chapters presented in this thesis.

4.1 Introduction

Of the several applications and developer tasks which computational models of code can help with, tasks that reason about concurrent programs have been studied the least [Allamanis et al., 2018a]. I was inspired to study concurrent programs after having addressed a range of data race issues in my S.M thesis [Srikant, 2020]. In Srikant [2020], I modeled programs written in Solidity, a language used in Ethereum, a now popular distributed ledger. The goal of the work was to learn distributed representations of lines of Solidity programs. I tested whether these learned representations encoded the presence of errors and bugs introduced in any given line of Solidity code. The proposed recursive representations, similar to those learned by graph neural networks, were insufficient to detect errors in Solidity, many of which were concurrency-related. The recursive representations only considered static information from programs and failed to capture the dynamic interactions which resulted in data races. The inadequacy of my solution revealed to me the need for an improvement in machine learning models trained to reason about concurrent programs.

In this chapter, I propose ways to train models to reason about data races in concurrent programs, which effectively indicates a model's understanding of concurrent programs. I first provide some background on data races detection (Section 4.1.1). In Section 4.2, I propose a theoretical formulation to learn to detect data races. The proposed formulation learns to sample the best worst-case execution trace of a program to train the model to detect data races. I discuss how operationalizing this formulation is challenging.

In Section 4.3, I study how models simpler than the formulation described in Section 4.2 learn data races by training the models on simulated datasets which have specific, experimenter-controlled properties. The simulated data models events appearing in a program thread as a string of characters. For example, a program thread containing the three events `read(x)`, `write(x)`, `print(x)` is denoted as the string `rwp`. I then study how well different ML models can be trained to detect the presence of specific substrings which represent data races in such strings.

I then attempt to study traces from real concurrent programs. Soon after starting work on this problem, I learned the following severe limitations:

- A comprehensive dataset of concurrent programs in which data races have been clearly labeled does not exist. This impeded my ability to study how well different ML models can be trained to detect data races.
- Solutions that have been developed over the last four decades to detect data races have not been evaluated on such comprehensive, labeled datasets. These solutions typically compare themselves to other prior solutions and report relative improvement. Consequently, it is unclear how accurate these different solutions are. This observation undermined the reliability of the solutions that have been proposed till date.

In Section 4.4, I describe *RaceInjector*, our proposal to create a rich dataset of concurrent programs containing data race issues, such that the learning setup I propose in 4.3 can be applied to real-world, complex concurrent programs. I show how the dataset is able to generate examples of data races that previous solutions are unable to detect.

4.1.1 Background

	thread 1	thread 2
1		w(z)
2		acq(lock)
3		w(x)
4		rel(lock)
5		w(y)
6	w(y)	
7	acq(lock)	
8	w(x)	
9	rel(lock)	
10	w(z)	

Figure 4-1: Example of a *potential* data race on lines 1 and 10, an *observed* data race on lines 5 and 6, and a safe access on lines 3 and 8.

In this section, we provide a brief background on traces, data races, and the race

detection algorithms that we evaluate in this work.

Traces. We assume a sequential consistency memory model [Lamport, 1979], where a program trace is a sequence of events on executing a program. An event can be denoted as a tuple <`op`, `thread`, `loc`>, where `op` is the operation that is performed, `thread` is the thread which performed the operation, and `loc` is the file and line which performed the event. An `op` can be one of the following: `read(x)`, `write(x)`, `lock(L)` (thread has acquired a lock on `L`), `unlock(L)` (thread has released the lock on `L`), `fork(T)` (a thread has forked a new thread `T`), `join(T)` (a thread `T` has joined the current thread). Nondeterminism in the scheduler can cause one program to have many possible traces.

```
class Test {
       static int x;
       static int y;

       void inc1() {
            synchronized(lock){
                x++;
            }
       }

       void inc2() {
            synchronized(lock){
                y++;
            }
       }
   }

public static void main(String[] args) {
  Test test = new Test();
  fork { test.inc1(); }
  fork { test.inc2(); }
}
```

Figure 4-2: A simple program $\mathcal{P}$ with two threads and no pre-existing data races

Correct reordering of traces. A trace σ^* is said to be a *correct reordering* of trace σ if it has the following properties:

1. *Thread Ordering:* The order of intra-thread events remains the same in both σ and σ^*.

2. *Read-Write Consistency:* For every read event in σ and σ^*, the most recent write event to the variable that is read remains the same. This is to ensure that control flow will remain the same.

3. *Locking Semantics:* σ^* does not violate the semantics of synchronization events,

	thread 1	thread 2
1		acq(lock)
2		r(x)
3		w(x)
4		rel(lock)
5	acq(lock)	
6	r(y)	
7	w(y)	
8	rel(lock)	

(a) Original execution trace of $\mathcal{P}$ (Fig 4-2)

	thread 1	thread 2
1		acq(lock)
2		r(x)
3		w(x)
4		rel(lock)
5		write(z)
6	write(z)	
7	acq(lock)	
8	r(y)	
9	w(y)	
10	rel(lock)	

(b) Execution trace for Figure 4-2, with a trivial race injected.

	thread 1	thread 2
1	write(z)	
2	acq(lock)	
3	r(y)	
4	w(y)	
5	rel(lock)	
6		acq(lock)
7		r(x)
8		w(x)
9		rel(lock)
10		write(z)

(c) The trace after running a solver to move the racy events apart.

Figure 4-3: A demonstration of injecting a data race in an execution trace of the program in Figure 4-2

such as locks and unlocks.

Intuitively, σ^* is a correct reordering of σ if any program that produces σ can also produce σ^*.

Data races. A data race occurs when two threads access the same variable without any synchronization, where at least one of these accesses is a write. Data races can be classified as *observed* or *potential* data races. An *observed* data race is where a data race actively occurs in a trace, where two threads attempt to concurrently access a shared variable where at least one of the accesses is a write. Observed data races are

trivial to detect. A *potential* data race is where no data race is observed, but there exists another correct reordering of the trace where an observed data race could occur. Potential data races are much harder to detect. See Figure 4-1 for an example. Events 5 and 6 in red are an example of an observed race, where two conflicting events occur simultaneously. Events 1 and 10 in orange are an example of a potential race, where they do not occur consecutively in this trace, but could in another correct reordering. Events 3 and 8 are not racy due to synchronization mechanisms.

4.2 A theoretical formulation to learn data races

For the purpose of this discussion, I consider a program to consist of multiple functions which can be called concurrently. Consider a system where a program $\mathtt{P}$ consists of multiple functions $\mathtt{F}$, each of which can be executed serially to perform a series of operations. Let $\mathtt{E}(\cdot)$ be a function which *concurrently executes* a program $\mathtt{P}$, containing a set of functions $\mathtt{F}$, producing a concrete execution instance $\mathtt{I}_k(\mathtt{P})$. This instance is an ordered list of lines from the functions in $\mathtt{P}$ which get executed. Assume some random state generates this ordered list. For instance, for the functions $\mathtt{F}_1$ and $\mathtt{F}_2$ in Figure 4-4, $\mathtt{E}(\mathtt{F}_j)$ is trivially the sequence [$\mathtt{L}_{11}$, $\mathtt{L}_{21}$, $\mathtt{L}_{31}$, ..., $\mathtt{L}_{n_1 1}$] and [$\mathtt{L}_{12}$, $\mathtt{L}_{22}$, $\mathtt{L}_{32}$, ..., $\mathtt{L}_{n_2 2}$], where $\mathtt{L}_{ij}$ corresponds to line i from function $\mathtt{F}_j$, and n_j is the number of lines in $\mathtt{F}_j$. When called individually, these functions execute sequentially by design. When run together though, $\mathtt{E}(\mathtt{P})$, where $\mathtt{P}$ is the program containing functions $\{\mathtt{F}_1, \mathtt{F}_2\}$, can produce an execution instance containing *interleaved* lines from the two functions. Two such instances $\mathtt{I}_1(\mathtt{P})$ and $\mathtt{I}_2(\mathtt{P})$ have been shown in Fig 4-4. Let $\mathtt{I}(\mathtt{P})$ (without any subscript index k to denote a concrete instance) denote the set of such interleaved orders of execution instances possible in $\mathtt{E}(\mathtt{P})$. It is likely that some subset of execution orders in $\mathtt{I}(\mathtt{P})$ can lead the system of programs in $\mathtt{P}$ to crash.

A question central to the program analysis community is - **given a program $\mathtt{P}$ consisting of multiple functions, can we detect if it can possibly crash when executed concurrently by $\mathtt{E}$.**

I attempt to cast this problem in the framework of adversarial ML.

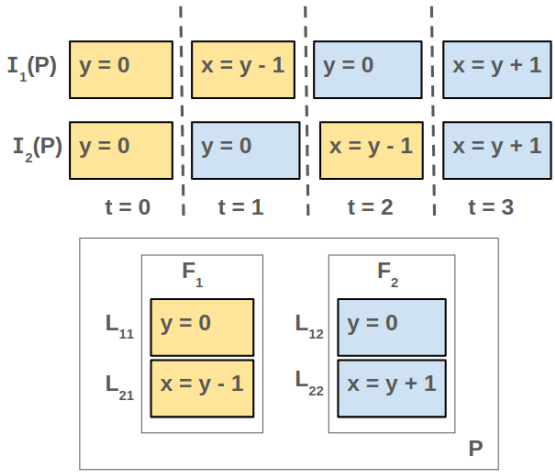


Figure 4-4: Program $\texttt{P}$ contains two functions $\texttt{F}_1, \texttt{F}_2$. The figure illustrates two possible instances of interleaving, $\texttt{I}_1(\texttt{P}), \texttt{I}_2(\texttt{P})$, that can occur in the lines of functions $\texttt{F}_1, \texttt{F}_2$ when concurrently executed. The value of $\texttt{x}$ when $\texttt{I}_1, \texttt{I}_2$ end executing at time $\texttt{t} = 3$ is 1 and 0 respectively. Here, $\texttt{I}(\texttt{P})$, the set of possible interleaved orders of executions, contains $4! = 24$ possible orderings of lines $\texttt{L}_{11}$,$\texttt{L}_{21}$,$\texttt{L}_{12}$,$\texttt{L}_{22}$

.

4.2.1 Problem formulation

I continue with the notation introduced in Section 4.1. Let $\boldsymbol{\theta}_{\texttt{L}}$ map each line $\texttt{L}_{\texttt{ij}}$ in $\texttt{F}_j$ to a continuous-valued representation $R_{ij} \in \mathbb{R}^d$ dimensions. Such representations can be obtained from various methods in the literature which model lines of source code [Alon et al., 2018a, Srikant et al., 2020]. Let $\boldsymbol{\theta}_{\texttt{P}}$ be weights that map a program $\texttt{P}$ containing functions $\texttt{F}$ to a label $\{0, 1\}$ denoting the presence of a data race issue. $\boldsymbol{\theta}_{\texttt{P}}$ in essence maps all the lines of a program to a representation of the entire program $\texttt{P}$, wherein each line's representation is mapped by $\boldsymbol{\theta}_{\texttt{L}}$.

Let $\texttt{I}_{\texttt{P}_i}$ denote the set of possible instances of line-orderings which can be generated when a program $\texttt{P}_i$ is executed by $\texttt{E}(\cdot)$. A data-point i in $\mathcal{D}$ corresponds to a program $\texttt{P}_i$ *concretized by any one instance* $\texttt{I}_k$ *in the set of instances* $\texttt{I}(\texttt{P}_i)$, and an associated label y_i. Specifically, what is provided at training time is just a randomly ordered list of lines of codes from the set of functions in $\texttt{P}$. Here, an instance $\texttt{I}_k$ is concretized by the representations of each line produced by the map $\boldsymbol{\theta}_{\texttt{L}}$.

The *true ordering*, if there exists one, of the lines of functions which leads the program to crash is unknown or latent. What the classifier ought to discover is an order-invariant pattern which signals the presence or absence of a data race issue. By training the classifier adversarially, where the adversary in this context can be considered to simply be an instance provided during inference which is not the *true ordering*, should help the classifier learn a signal which is invariant to the ordering.

The problem of detecting a data race issue thus reduces to the following objective function -

$$\underset{\boldsymbol{\theta}_{\mathtt{P}} \circ \boldsymbol{\theta}_{\mathtt{L}}}{\text{minimize}} \;\; \mathbb{E}_{(\mathbf{x},y)\in\mathcal{D}} \left[\underset{\mathbf{z}\in\mathcal{I}_{\mathbf{x}}}{\text{maximize}} \left(\ell(\mathbf{z}, y, \boldsymbol{\theta}_{\mathtt{P}} \circ \boldsymbol{\theta}_{\mathtt{L}}) \right) \right] \tag{4.1}$$

Here, $\mathbf{z}$ is an instance sampled from the set of instances of a program $\mathtt{P}$. The parameters of the model being optimized are denoted using the composition $\boldsymbol{\theta}_{\mathtt{P}} \circ \boldsymbol{\theta}_{\mathtt{L}}$, which suggests that these parameters need to jointly be optimized.

Intuitively, the inner maximization is training the classifier to learn the specific (worst-case) instance, among the combinatorial-space of instances of $\mathtt{P}$, which can correspond to the properties of data race issues. We minimize $\theta_{\mathtt{P}}$ such that it reduces the classification error of the $\{0, 1\}$ label. In this formulation, I assume only one *type of* data race issue is present in $\mathcal{D}$. Further, I assume that the presence of this type of issue in each instance i in $\mathcal{D}$ is not parameterized by any set of constraints $\mathtt{C}_i$ on $\mathtt{P}_i$.

4.2.2 Implementation challenges

While the formulation suggested in Eq 4.1 can potentially learn data race issues in traces, the following challenges make it infeasible the solution for real programs.

- **Representation of z.** The formulation relies on $\boldsymbol{\theta_L}$ finding a mapping of z, a line sampled from program P. In practice, mapping lines to a distributed representation in a way such that the representation faithfully encodes the information conveyed in a lines is a challenging learning task [Srikant, 2020]. Curiously, very few among recent works on code representation learning [Allamanis et al., 2018a] focus on learning representations of entire lines of a program. Thus, the inner problem of the objective

that is required to sample z such that the inner loss is maximized can be a challenging problem to solve.

- **Efficiently sampling $\mathbf{z} \in \mathcal{I}_{\mathbf{x}}$.** The inner problem requires different samples of $\mathbf{z}$, which in turn depends on the set $\mathcal{I}_{\mathbf{x}}$ to contain a large number of program instances and corresponding labels denoting the presence of a data race. The only way to construct a sizeable set $\mathcal{I}_{\mathbf{x}}$ to sample from is to execute the program `P` several times. This task of multiple executions and determining ground truth for the generated instances can be computationally expensive and infeasible. Note - this process requires ground truth labels to begin with. Every sample in set $\mathcal{I}_{\mathbf{x}}$ should have a corresponding label associated with it, making the construction of such a set a challenging bootstrapping problem. The larger goal to construct a data race detector using this learning objective is rendered moot if exist data race detectors which can efficiently label samples in $\mathcal{I}_{\mathbf{x}}$.

As a consequence of the challenges discussed in this section, the formulation proposed in Eq 4.1 can pragmatically be used when labeled datasets of execution traces are more easily accessible. Section 4.4 of this chapter directly addresses the challenge of assembling such labeled datasets.

4.3 Simulating data races to study the limits of ML models

4.3.1 Introduction

In the previous section, I showed how a a learning objective to detect data races is limited by the lack of an annotated dataset. To circumvent this limitation, I propose generating a dataset containing strings generated from a language I design. I show that the generated strings can simulate program execution traces, which allows us to study how well ML models can be trained on the problem of data race detection. Simulating data races also allows us to understand how much data we will need if we were to learn the problem of data race detection using ML models, including that referenced in objective Eq 4.1.

In this work, I do not study the objective proposed in Eq 4.1 on the synthetic data, and instead study much simpler ML model architectures. The simpler models allow for a richer understanding of the limits of learning objectives.

4.3.2 Simulating data races - A toy language

For the rest of this section, I focus on the problem of reading a variable that has not yet been written to. This is a classic synchronization issue which comes up in various data management systems, where a shared counter variable is either incremented or decremented by different functions [Zhivich and Cunningham, 2009, Boehm, 2012]. If a read operation of the shared variable happens *before* a write occurs to it, a segmentation fault occurs.

Synchronization is the standard methodology to prevent data races. Synchronization prevents two threads from performing operations concurrently. The most common mechanism for synchronization are locks. Only one thread can hold a lock at the same time. If thread F1 holds lock L, and thread T2 requests lock L, T2 must wait until T1 releases L in order to continue execution. This allows T1 and T2 to access the same memory location while ensuring there is no possible race, as it is not possible for them to access the variable at the same time due to the lock semantics.

Let us assume our setup to be the following - let each program $\mathtt{P}_i$ in $\mathcal{D}$ contain functions, $\mathtt{F}_{ji}$. Each function performs one operation in every line of its code. For a function containing k lines, let the notation $[a_i, a_2, a_3, ..., a_k]$ denote the *operation map* - the operations in each of its k lines. These operations can be represented using a continuous vector which captures what the specific operation is. See the following section for details.

For the remainder of this discussion, I consider the following notation for the possible set of operations -

1. **Read** (r) Denotes a read operation of a shared variable.
2. **Write** (w) Denotes a write operation to a shared variable.
3. **Check** (c) Denotes a check of existence on a shared variable. In Python, say, this check could look like

`if shared_var is not None: Do something`

`else: raise ValueError('No value found')`

4. **Up** (u) Denotes an *Up* operation of a semaphore.
5. **Down** (d) Denotes a *Down* operation of a semaphore.
6. **Compute** (.) Denotes any other arbitrary computation not relevant to the shared variable.

To illustrate this, if $\mathtt{F}_1, \mathtt{F}_2$ have the following operation maps $[...c..r...], [..w..]$, it means the following -

1. $\mathtt{F}_1$ has 10 lines of code and $\mathtt{F}_2$ has 5 lines of code.
2. $\mathtt{F}_1$ has a check (c) before a read operation (r) of a shared variable. There's one line of arbitrary computation (.) between the check and read operation.
3. $\mathtt{F}_2$ has a write (w) operation to a shared variable after two arbitrary computations (.).

4.3.3 Generalization properties which the generated dataset can test

To understand how well ML models generalize to different properties presented in data race detection, we generate the synthetic varying across multiple dimensions. Each of these dimensions poses a unique generalization challenge in data race detection that would the ML models would have to generalize to.

The different dimensions are:

- **Train-test split.** Random separation of the full dataset based on a given train-to-test ratio parameter. Such a split reflects the randomness of the generated execution traces of programs. It is infeasible to train models on all possible event orderings that can potentially appear.
- **Location-specific samples.** Separation of samples where meaningful operations appear only in the first half of a trace in the training set, while they appear only in the second half of a program in the test set. This is used to test a model's robustness to

variance in the position of operations. Even if a model can correctly classify a pattern that only occurs in the first half of a program during training, it should be able to generalize to correctly classify the same pattern if it were to occur in the second half of a program at test time.

• **Variable inter-operational distance.** The train and test set containing a varying number of no-op operations between meaningful operations. For example, the train set can contain samples with an inter-operational distance of 1 (e.g. `w.u`), while the test set can contain samples with an inter-operational distance of 2 or 0 (*e.g.* `w..u` and `wu` respectively). This variation requires the models to reason over large spans of characters to learn and infer the presence of data races. We should expect recurrent networks like LSTMs to outperform models with fixed and smaller spans like CNNs.

• **A combination of dimensions.** We combine the different variations described above to construct a training and test set that differ across multiple dimensions described above. This combination serves as a stronger test of generalization for the models.

4.3.4 Desirable capabilities of the learned models

We want our learned models to possess the following desired capabilities –

1. **Robustness to position variance.** If the model is provided two instances during training, say, $[r...]$, $[.r..]$, of crashes, can it generalize to predict that $[..r.]$, $[...r]$ also cause crashes.

2. **Robustness to inter-operational distance variance.** If the model is provided two instances, say, $[w....u]$, $[w..u]$ that correspond to correct behavior (say), can it generalize such that $[w.....u]$, $[wu]$ also are correct behavior. Such properties can hold across the two behaviors.

3. **Sensitivity to relative ordering of operations.** The model should be able to distinguish relative ordering of operations. For instance, the model should be able to associate $[w..u]$ with correct behavior whereas $[u..w]$ with a crash.

4.3.5 Experiments and Results - A summary

The experiments and results from Rares Begu [2020] have been summarized briefly here. For details, please refer to the thesis.

The aim of our experiments is to evaluate different ML models on their ability to generalize to the conditions described in Section 4.3.4 by constructing and training them on datasets with the properties described in Section 4.3.3.

Models. We evaluate three models: CNNs, LSTMs, and DeepSets [Zaheer et al., 2017]. DeepSets is a deep learning architecture designed to learn a set of objects, *i.e.* the model learns to permutation invariant representations of its inputs. In our case, DeepSets serves as a baseline to evaluate how well permutation-related constraints are learned. We use a feedforward network as a baseline.

Results summary. Our results confirm our hypotheses that only CNNs and LSTMs can generalize learning data race patterns when provided with simulated strings representing execution traces.

Across all of our dataset variations, CNN and LSTM-based models consistently achieve an accuracy of above 90% (with an overall accuracy across all experiments of 98.4% and 98.6% respectively), with feed-forward and DeepSets only achieving 78.2% and 78.1% respectively.

However, none of the models perfectly generalize to a combination of the conditions described in Section 4.3.3. The test accuracy drops roughly to 80% and 90% for CNNs and LSTMs respectively. This is particularly concerning since we should expect most realistic datasets of execution traces to mimic a combination of the properties described in Section 4.3.3. Future work should study the generalization of models when trained on such data configurations.

These results suggest that models that have a local view of the input (CNNs, LSTMs) have an advantage over those that do not (feed-forward, DeepSets). Further, models that can traditionally reason about recursive structures outperform fixed length maps, like those present in CNNs. Future work can explore whether other model architectures like gated recurrent units and graph neural networks can generalize to a

combination of the properties described in Section 4.3.3.

4.4 First steps towards learning data races: Creating a labeled dataset

Data race detection in concurrent programs using their execution traces, *i.e.* dynamic analysis, has been shown to be in NP-hard [Mathur et al., 2020]. Practical algorithms designed to detect data races hence rely either on heuristics [Lamport, 1978, Mathur et al., 2018, Smaragdakis et al., 2012, Kini et al., 2017, Mathur et al., 2021, Savage et al., 1997] or SMT-solvers [Wang et al., 2009, Kalhauge and Palsberg, 2018, Flanagan and Freund, 2009, Roemer et al., 2020, Huang et al., 2014b]. The goal of these algorithms is to start with a trace, and determine if two conflicting accesses in different threads to a shared variable can occur concurrently in an alternate execution of the program. Despite numerous such algorithms having been proposed over the last few decades, it is surprising that there exist no comprehensive benchmarks comprising industry-grade software projects which have races annotated in them—either annotated in the source code or in the traces—which would help rigorously evaluate these algorithms. Consequently, it is unclear what the true classification accuracy rates (true-positive, true-negative, *etc.*) of these algorithms are.

For instance, none of the larger benchmark datasets such as DaCapo [Blackburn et al., 2006], which have all been repeatedly and extensively used in the evaluation of multiple race detection algorithms cited above, have any ground-truth annotations.

One key reason for the lack of such datasets is the absence of sound, scalable methods to assemble them. A few prior works [Yuan et al., 2021, Lin et al., 2015, Jalbert et al., 2011, Gao et al., 2018] have released expert-annotated datasets containing races. However, they are too small to be effective. Jacontebe [Lin et al., 2015] contains a total of 19 data race bugs, RadBench has 10 bugs, while GoBench [Yuan et al., 2021] has 103 bugs for the Go language. With such small datasets, it is hard to evaluate if current algorithms commonly miss any race patterns.

Another approach to assembling such datasets has been to run SMT-based race detection algorithms on industry-grade software projects [Gao et al., 2018]. SMT-based race detection approaches, by their design, can provably detect true-positive data races. The detected true-positive data races are used as annotations and released as datasets. A major limitation of these approaches however is the relatively small number of constraints that SMT-solvers can solve at once. Longer, more complex real-world software produce longer execution traces, which in turn non-linearly increase the number of constraints which SMT-solvers have to solve. Detection algorithms typically circumvent this limitation by breaking the trace into fixed-length windows and solving each window as if they are independent of others [Huang et al., 2014a]. The assumption of independence of windows is not practical, thus limiting the number and quality of races that can be detected. Moreover, assembling a dataset using data races detected by known detection approaches limits the nature of races we can test other detection algorithms on. We further discuss other prior works in Section 4.4.3.

As a direct consequence of this absence of scalable methods to assemble comprehensive annotated datasets, we argue that the metrics that have been employed to evaluate any new race detection algorithm do not accurately represent their performance. Further, we argue that the lack of such annotated datasets has potentially stifled the state-of-the-art.

- **Evaluating race detection algorithms.** Presently, the efficacy of newly proposed race detection algorithms is primarily measured by the increase in the number of identified data races when compared to a previous algorithm. Among heuristics-based algorithms [Lamport, 1978, Mathur et al., 2018, Smaragdakis et al., 2012, Kini et al., 2017, Mathur et al., 2021, Savage et al., 1997], each new algorithm has successively proposed a set of rules which purportedly improves upon previous algorithms. The newer algorithms then demonstrate detecting races which the previous algorithms did not. The newly detected races are verified manually by experts. In this process, it is unclear how many true-positives and false-negatives each new set of rules introduce. When an algorithm reports finding N (say) new bugs which a previous algorithm had not found, it is unclear what N is relative to—the total number of bugs which either

of these algorithms are supposed to find in the first place. With surprisingly little attention paid to this essential metric, it is unclear what the state of progress in data race detection algorithms has been over the years.

Further, it is unclear whether an improvement proposed using a set of new rules results in detecting newer *classes* of data races, assuming there exist multiple semantic classes of use-cases in a program's execution behavior which manifest as race bugs. It is quite possible that a proposed improvement to an existing algorithm, while detecting a few new bugs, may not necessarily cover a significantly larger set of such semantic classes. An annotated dataset with a diverse set of bugs in them is the first step in establishing and quantifying the semantic classes a detection algorithm covers.

- **Training ML models.** We posit that it is possible for data-driven methods to replace the many heuristics which have been proposed over the years for race detection. Heuristics for race detection involve learning to draw edges between events of interest in a program's execution trace, and identifying cliques of connected or disconnected events. These cliques are then used to reason about and infer the existence of potential races. These heuristics typically guarantee soundness only for the first race they detect.

Given the inadequacy of existing guarantees, machine learning (ML) models provide a practical alternative. ML models have been shown to outperform expert-crafted heuristics when reasoning about graph-based data in the domains of compiler optimization [Cummins et al., 2021], network-graph analysis [Bowman et al., 2020], pointer analysis [Jeon et al., 2020], fault localization [Lou et al., 2021] and more. While such learned models will not be able to guarantee soundness even for the first detected race, they may well offer an improved performance in reasoning with trace-based information over expert-crafted heuristics. However, a key requirement to train any ML model is a sizeable, well annotated dataset.

Our solution–SMT-based race injection. We propose injecting races into existing concurrent software as an approach to scalably create comprehensive, annotated datasets. Specifically, we inject data races into an execution trace of a given program. We choose to inject into the execution trace rather than the source code because

injecting races into the program source does not guarantee the race manifesting into every execution trace of the program. This makes it difficult to evaluate race detection algorithms that employ dynamic analysis methods. We refer to the execution trace before injection as the **base trace**.

Adding two consecutive events that are conflicting (*e.g.* a read event immediately followed by a write event to the same variable without any synchronization mechanism) is the simplest way to inject a race into the base trace. More difficult is to inject conflicting events that *could possibly* occur consecutively in a different, random execution of the program. Our goal thus is to generate traces in which such conflicting events appear far apart in them, making them non-trivial to detect. To achieve this, we propose RaceInjector, which injects a trivial data race into any trace, and then uses an SMT-solver to find an alternate, valid reordering of the base trace where the conflicting events appear far apart. This approach is independent of how any race detection algorithm works and the program generating the trace into which the trivial race is injected. Our method importantly guarantees the injection of a race while maintaining the semantics of the base trace. Thus, RaceInjector generates traces with injected races appearing at random, valid locations, mimicking a thread scheduler scheduling a program containing a valid data race. To ensure RaceInjector generates data races that appear arbitrarily far apart in a trace, we propose a method which circumvents practical limitations of SMT-solvers while guaranteeing semantics of the base trace. We describe our approach in detail in Section 4.4.1. In this work, we generate a small dataset and demonstrate it on the research questions that it helps address. We also show how it can be easily extended to a comprehensive dataset. Among the few traces we generate, we find traces with data races that current state-of-the-art race detection methods fail to detect. This demonstrates one immediate utility of RaceInjector. A sample of these counterexamples can be found at `https://github.com/ALFA-group/RaceInjector-counterexamples`. In Section 4.4.2, we discuss other implications of RaceInjector.

4.4.1 Method

In this section, we describe how we inject synthetic data races into program traces. We begin with a motivating example.

Motivating example. The program $\mathcal{P}$ in Figure 4-2 reads and writes to two variables `x` and `y`. One of its possible execution traces is shown in Figure 4-3a. This is the **base trace** for our injection. Originally, this program does not contain a data race. We note that a thread switch occurs after event 6. We can trivially inject a race directly after the thread switch by adding two write events to the trace (lines 5-6, Figure 4-3b), resulting in an *observed* data race.

To invoke non-trivial reasoning to detect our injected data race, we propose using an SMT-solver to find a correct re-ordering of the events in a trace such that (a) the original trace's semantics hold, and (b) the inserted events are *moved apart* by some L events. One such alternate reordering can be seen in Figure 4-3c. Recall the definition of a data race: two events that access the same variable, at least one of which is a write, that occur in an unsynchronized manner. Our solution to generating these data races has three steps which we describe in detail: **Step 1:** Instrument and execute a program; collect base traces of relevant events. **Step 2:** Add a trivial data race to a base trace, and finally, **Step 3:** Use an SMT-solver to make the added race harder to detect.

Step 1. Trace collection. We start by logging a sequential trace of data accesses and thread synchronizations in a program. See Figure 4-3a for an example trace. Races are then injected into the collected *base traces* and analyzed. This decoupling of instrumentation and race injection allows for several instrumentation frameworks to be used. We use MCR [Huang, 2015] which instruments using the ASM framework [asm]; RoadRunner [Flanagan and Freund, 2010] which also instruments with ASM, and Calfuzzer [Joshi et al., 2009] which instruments using the SOOT compiler framework [Vallée-Rai et al., 1999]. SOOT and ASM allow the instrumentation frameworks to modify the bytecode and intercept relevant events as they occur during execution.

Step 2. Adding a trivial race. To modify a base trace to add a trivial data race,

we insert two new write events right where there is a context switch between threads. See Figure 4-3b for an example of modifying the base trace in Figure 4-3a. The writes are made to a new, dummy variable to ensure the semantics of the original program remains the same. We only inject one race into the base trace at a time before saving it.

Step 3. Using an SMT-solver to move apart the added data race. After having injected a trivial race comprising consecutive conflicting events, the goal is to then find an alternate valid interleaving where the race-events are farther apart.

We set up n SMT variables v, where each variable $v_j \in [1, n]$ corresponds to an event that appears in the base trace containing a total of n events. The value of v_j signifies the location index where the event should appear. In trace 4-3a for example, if v_1 corresponds to event `w(x)`, the assignment for v_1 corresponding to the trace would be 3, the location index `w(x)` appears in the trace. Similarly, if v_2 corresponds to event `w(y)`, the assignment of v_2 corresponding to the trace is 7. For a trace σ, we then formulate a constraint equation in a way that solving the constraints yields a valid assignment made to each v_j which results in an alternate trace σ^*.

Our constraints must ensure that the alternate trace is a correct reordering as defined in Section 4.1.1 (thread ordering, read-write consistency, locking semantics). These constraints have been commonly defined in race detection to find alternate reorderings [Wang et al., 2009, Said et al., 2011, Huang et al., 2014a].

Readers can refer to Said et al. [2011] for details. However, we introduce the following constraints in order to inject data races into base traces:

- **Distance between conflicting events.** We supply a hyperparameter `L` which constrains the distance between the inserted racy events.
- **Additional constraints.** We additionally ensure that the indices assigned to each v_j is positive, unique, and lies in the interval $[1, n]$.

We supply a conjunction of these constraints to an SMT-solver which produces an assignment to each v_j. These assignments correspond to a new, valid reordering of each event appearing in σ, thus resulting in a new trace σ^*. Further, σ^* contains the previously trivially injected race events now at least L events apart, and ensures the

Program	Base traces			RaceInjector-generated traces		
	#Inj. pts	Length	#Thrd	#Gen. traces	Avg race dist.	Max race dist.
ArrayList	207	677	27	207	$128_{\pm 111}$	558
TreeSet	130	756	22	130	$122_{\pm 115}$	526
LinkedList	1767	14937	451	160	$112_{\pm 124}$	851
Stack	2036	11372	451	100	$87_{\pm 74}$	458
Jigsaw	3394	97110	78	467	$693_{\pm 777}$	7396

Table 4.1: **Overview of RaceInjector results on a benchmark of programs.** Column 1 lists the different program benchmarks in which RaceInjector injects races. Columns 2,3,4 describe the base traces. The remaining columns describe the traces generated by RaceInjector. *Inj. pts.* refers to the number of injection points available in the base trace; *Thrd* the number of program threads.

Algorithm	ArrayList	TreeSet	Jigsaw	Stack	LinkedList	# Missed
HB (1979) [Lamport, 1978]	✔	✔	✔	✖	✖	60 (5.6%)
SHB (2018) [Mathur et al., 2018]	✔	✔	✔	✖	✖	64 (6%)
WCP (2017) [Kini et al., 2017]	✖	✔	✖	✖	✖	21 (2%)
SyncP (2020) [Mathur et al., 2021]	✔	✔	✔	✖	✖	22 (2%)

Table 4.2: **Counterexamples generated by RaceInjector.** A ✔ signifies there exists at least one trace among the RaceInjector-generated traces which is not detected by the corresponding algorithm. *# Missed* reports the number of traces the algorithm misses to detect (percentage mentioned within parenthesis).

same execution semantics as that of σ. We elide details of the symbolic encodings of these constraints for the sake of brevity.

Moving events arbitrarily apart in a trace. The number of constraints in the conjunction described above which generates σ^* is typically prohibitively large for SMT-solvers to solve. Our insight to circumvent this practical problem is to incrementally move the introduced conflicting events farther apart. We start with reordering the conflicting events (which initially appear consecutively when injected) and the events surrounding it in a window of fixed size. For the events in this window, we generate the constraints described above and run RaceInjector. We choose a window size in a way that the number of constraints does not overwhelm the solver. Once RaceInjector generates a reordering for the events in the window, we slide the window over by a fixed length and run RaceInjector again on the events that appear in the shifted window. We ensure the shifted window contains at least one of the two conflicting events we introduce, which will have been reordered from

their initial, consecutive indices. Running RaceInjector iteratively over smaller, fixed-length windows k times is computationally much more efficient than running the solver on a large number of events just once—the number of constraints tend to grow superlinearly with the number of events needed to reason about.

4.4.2 Results & Discussion

We demonstrate RaceInjector by using it with a suite of program benchmarks used in prior works to generate a sample of base traces containing data races (Section 4.4.2.1). Among the generated traces, we also find counterexamples which state-of-the-art race detection algorithms fail to detect (Section 4.4.2.2).

4.4.2.1 Generated dataset: Quantitative description

We employ RaceInjector to generate only a small, demonstrative dataset comprising ∼1000 total traces in this work. This is nonetheless sufficient to show the ease with which RaceInjector can be extended to generate a comprehensive dataset. We ran our experiments without any parallelization using Java version 11.0.18, on a CPU running Ubuntu 18.04 with 96 GB RAM.

Base traces. We run each of the five program benchmarks listed in Table 4.1 once on the testcases from `Calfuzzer` [Joshi et al., 2009]. This instruments and generates one execution trace each for each of the five programs. Table 4.1, columns 2, 3 and 4 document statistics of each program's base trace. Each program has a different number of threads (column 4). Consequently, each trace presents a different number of points of injection (column 2) to introduce a trivial race—these are points at which thread context-switches occur. We run RaceInjector on each program's trace (except for Jigsaw) for one hour, with a goal of injecting races into as many entry points as possible within the allotted one hour. Since Jigsaw is a significantly larger program than the rest, as seen by the length of an average trace generated (column 3), we run RaceInjector for ∼10 hours instead on the Jigsaw trace.

RaceInjector-generated traces. Running RaceInjector results in a total

of ∼1000 traces (sum of column 5). Columns 5, 6 and 7 in Table 4.1 document the statistics of the generated traces. Note again, these are all guaranteed to contain data races. The set of traces generated by RaceInjector for any one program will all have the same length (column 3) because each trace is just one possible valid reordering of the original. We find the number of traces (column 5) generated in one hour of running RaceInjector is roughly the same across the different program benchmarks (ignoring Jigsaw). In columns 6 and 7, we report the average distance and the maximum distance between the injected conflicting events in the traces generated by RaceInjector, which is measured by counting the number of events between them. We observe the average distances (column 6) to be significantly greater than zero, suggesting that the injected races, which are initially placed consecutively, end up significantly apart in the generated traces. From the standard deviations (subscripts in column 6), we see very high variance in the distance between injected races, suggesting the heterogeneity in the potential data races introduced by RaceInjector. In `ArrayList` and `TreeSet`, the maximum distance between the injected race events (column 7) nearly span the length of their base traces (column 3).

This demonstrates the flexibility offered by RaceInjector to generate any number of traces with guaranteed races that are varied in the locations they appear in. This is a desirable feature for a high-quality annotated dataset of such concurrent programs.

Discussion: Scaling the dataset. To assemble a comprehensive dataset, we recommend the following procedure: first, execute a program multiple times to obtain different base traces. Second, run RaceInjector on every possible point of injection in each base trace without constraining the time taken to complete this process. Both the steps are easy to parallelize. We plan to extend the race detection algorithms evaluated to include more recent tools such as RPT [Thokair et al., 2023], static race detection methods like Infer [Distefano et al., 2019], and SMT-based methods [Huang et al., 2014a].

Discussion: True-negative samples. In proposing RaceInjector, we only consider the problem of generating true-positive data races. In our larger goal to assemble a comprehensive dataset large enough to train machine learning models, we

would also need a method to assemble examples of true-negative cases in our dataset. As most pairs of conflicting accesses in software are not races, we can randomly sample such accesses, verify them using simple algorithms like HB, and label them as true-negatives. Machine learning algorithms are tolerant to a small number of mislabeled samples.

4.4.2.2 Counterexamples to SOTA algorithms

Table 4.2 shows that RaceInjector can easily produce counterexamples to state-of-the-art (SOTA) detection algorithms.

This is important because it reveals for the first time their sensitivity as well as guides future work to clearly define new rules and algorithms. We can potentially study the contribution of the different rules present in the heuristics of different algorithms in the decisions made by the algorithms. Thus, access to a comprehensive set of counterexamples can potentially empirically justify the different rules implemented by these algorithms, and can also point to equivalences between some of these rules. While we do not directly study the counterexamples, this is a compelling direction for future work.

We now analyze the counterexamples generated by RaceInjector that the different algorithms fail to detect. They are available at: `https://github.com/ALFA-group/RaceInjector-counterexamples`. The analysis that follows should be interpreted with caution because the number of counterexamples is relatively small (fewer than 100), which does not support statistical comparison tests. We will evaluate these claims rigorously on a larger dataset in future work. This analysis is instead indicative of the questions that can be studied.

SHB vs. HB. SHB guarantees soundness after the first detected data race it detects in exchange for detecting fewer races overall compared to HB. We should then expect SHB to detect fewer races on average, and conversely miss detecting more races. This is what we observe: SHB fails to detect 64 bugs RaceInjector generates (column 7, Table 4.2), 4 more than HB. That said, the total percentage of bugs that the algorithms fail to detect are roughly the same ($\sim$6%). We will, in the future, also

compare whether they fail on the same set of counterexamples.

SyncP vs. WCP. Despite SyncP following and improving upon WCP, we do not see a notable increase in its performance. Both fail to detect 2% of the generated races. On the other hand, both algorithms improve upon HB, so it is expected to see a improvement of ∼4% in the races they fail to detect when compared to HB.

`LinkedList` and `Stack`. We observe that none of the injected data races in `LinkedList` and `Stack` fail any algorithms (columns 5, 6, Table 4.2). Besides the low number of samples generated, a possible reason could be the large number of unsynchronized threads. These two programs have the largest number of threads relative to the length of their traces (Table 4.1). We suspect these threads mostly involve unsynchronized accesses, making the injected races relatively easy to detect as well. If in the future our hypothesis that the threads mostly involve unsynchronized accesses holds, we will filter out such injection points to reduce the number of trivially detectable races in our dataset.

Table 4.2 indicates that RaceInjector is able to generate data races which no SOTA method detects. This implies RaceInjector finds locations in a program trace which are complex to reason about. To finish, RaceInjector makes the widespread adoption of classification accuracy-related metrics (true-positive, false-positive, true-negative, false-negative) now possible when evaluating and comparing race detection algorithms.

4.4.3 Related work

Prior work closest to RaceInjector has mostly compiled known bugs that have been found over the years. Because these bugs have already been found, it is difficult to evaluate the capability of new approaches to detect new bugs. Additionally, these datasets are far too small to train a machine learning model, the largest being 985 races in Jbench [Gao et al., 2018]. JaConTeBe [Lin et al., 2015] is a benchmark of Java concurrency bugs, which scrapes past papers and aggregates a list of 47 distinct bugs along with their causes. GoBench [Yuan et al., 2021] is a dataset of 103 bugs in Go, scraped from Github. RADBench [Jalbert et al., 2011] is a dataset composed of

snapshots of open-source software projects with 10 total known bugs. Jbench [Gao et al., 2018] is a dataset of Java data races, aggregated from artifacts of existing race detection tools, and contains 985 unique data races. Jbench contains 6 real-world applications, and 42 custom testcases that were written during development of previous race detection tools. Typically, all these benchmarks are curated by either expensive manual analysis, or have been assembled using existing tools, which greatly limits their usefulness in evaluating and improving extant race detection algorithms while RaceInjector is fully automated. Additionally, since many of the samples have been curated using existing tools, a machine learning model trained on these samples will be unlikely to outperform the original tools used to find them.

Chapter 5

Program comprehension and the human brain

Preface. This chapter, in full, is a reprint of the arXiv report Srikant et al. [2023a]. The report is a re-write of **Comprehension of computer code relies primarily on domain-general executive brain regions.** Ivanova, A. A., Srikant, S., Sueoka, Y., Kean, H. H., Dhamala, R., O'Reilly, U.M., Bers, M. U., and Fedorenko, E. (2020). Elife, 9:e58906 [Ivanova et al., 2020]. The chapter informs the results of the eLife work to a computer science audience; the original eLife work was written to primarily inform the cognitive neuroscience community.

Refer to Section 1.4 to read more about the motivation behind this work, and how it connects to the rest of the chapters presented in this thesis.

5.1 Introduction

Reading and understanding computer programs (code) has been estimated to consume nearly 60% of a software professional's time [Xia et al., 2017]. Yet, we understand little of how we cognitively accomplish it, making this an open question in science. Extending seminal precedents [Siegmund et al., 2014, Floyd et al., 2017], we attempt in this work to study and establish the regions of the brain that are involved in comprehending computer code.

The recency of code comprehension as a cognitive skill suggests that brain regions which specialize in supporting other cognitive activities likely also support code comprehension. Given its association with logic and problem solving, code comprehension can arguably be handled by regions responsible for working memory and cognitive control, or those involved in math and logic. Similarly, code and natural language share many common properties. They possess similar syntactic and semantic structures, and hierarchically compose to convey meaningful information – in both code and text, tokens are associated to form statements, which are further associated to form an entire code or document, which results in meaning being associated with the artifact [Fedorenko et al., 2019]. Arguably, regions of the brain involved in processing language can support code comprehension.

Neuroimaging research is well positioned to address which regions are involved in code comprehension. Techniques such as functional magnetic resonance imaging (fMRI) measure brain activity when performing cognitive tasks like reading or hearing music. Brain regions whose functions have been well established, like language or music centers, responding to a new task, like code comprehension, can indicate the cognitive processes likely associated with that task [Mather et al., 2013].

In this work, we use fMRI to study how code-reading related tasks engage two known systems of brain regions – the Multiple Demand (MD) and Language systems (details in Section 5.3). While previous neuroimaging studies have also investigated brain regions involved in code comprehension, their results remain inconclusive. They provide evidence for activity in regions that roughly correspond to the MD system [Floyd et al., 2017, Huang et al., 2019, Siegmund et al., 2014, 2017, Liu et al., 2020b], as well as in regions resembling the Language system [Siegmund et al., 2014, 2017]. Importantly, these studies do not distinguish the act of code comprehension from other code-reading related activities like mentally simulating code. Further, most do not quantify brain responses, and compare them to responses to other tasks associated with working memory or language to meaningfully interpret their observations. We review these works in Sections 5.2 and contrast their design choices to ours in Section 5.4.

Our contributions in this work are twofold. First, we design novel experiments and introduce improved methods to identify brain regions involved in code comprehension. Second, we present a new set of results which adds to our current understanding of the cognitive bases of code comprehension. We summarize our design and method contributions below. See Section 5.6 for details on our results.

- We offer a clearer definition of code comprehension, and design experiment conditions to isolate and measure it.
- We use a state-of-the-art procedure to determine which known, well-characterized brain systems respond to code comprehension.
- We test our experiments in two programming languages - Python and ScratchJr, a programming system with a fully visual interface, on a group of 24 and 19 participants respectively. Using ScratchJr enables measuring the effect of text on code comprehension, and additionally helps validate the generalizability of our results. Prior studies have experimented only with one programming language.
- We ensure that the observations we make generalize to different code properties like control flow (sequential programs, loops, conditionals), or types of operations performed (string, math operations).
- We additionally investigate whether brain activity corresponding to code in the Language system is a result of descriptive variable names used in codes.
- We make our code, stimuli, and brain data publicly available for the community to reuse and extend. Link - `https://github.com/ALFA-group/neural-program-comprehension`

5.2 Related Work

The question of whether there exist specialized regions in the human brain which are exclusive to specific cognitive functions goes back to Paul Broca's investigations of language understanding in the 1850s [Henderson, 1986]. Advances in technology to accurately measure neural activity in the last three decades have revealed the existence of specialized regions for a variety of cognitive functions like language processing, face

recognition, navigation *etc.* [Kanwisher, 2010]

The use of neuroimaging techniques to study the cognitive responses to programming has gained momentum recently. Prior works have investigated the neural processes involved in debugging [Castelhano et al., 2019], variable tracking when reading programs [Ikutani and Uwano, 2014, Nakagawa et al., 2014], semantic cues or program layout [Fakhoury et al., 2018, Schröter et al., 2017], program generation [Krueger et al., 2020], manipulating data structures [Huang et al., 2019], biases in code review processes [Huang et al., 2020], and programming expertise [Floyd et al., 2017, Parnin et al., 2017, Ikutani et al., 2020].

Relevant to our scope are works which investigate regions of the brain involved in comprehending code (as opposed to writing code, or any other coding-related activity).

Siegmund *et. al.* Siegmund et al. [2014], an influential work which pointed the community's attention to this topic, investigate the question of which regions in the brain are involved in code comprehension. They present two sets of stimuli to 17 participants in an fMRI study. The first requires participants to read through snippets of code and determine their outputs. The second requires them to read code snippets with syntax errors and suggest fixes. The authors contrast activations from these two sets of stimuli, both of which correspond to code comprehension activity in the brain, to a baseline of no activity. They show parts of this contrast to lie in the Broca's region (language centers) as defined by Brodmann's areas [Brodmann, 1909].

Floyd *et. al.* Floyd et al. [2017] pose a different primary research question. They investigate, on a larger sample of 29 participants, whether it is possible to distinguish the act of program comprehension from English sentence comprehension using brain activity measurements. Their decoding experiments show that neural representations for code are unique and different from language. As a secondary result, they do comment on brain regions involved, and partially confirm Siegmund *et. al.*'s findings. In their design, they use a baseline contrast of an English comprehension task and two code-reading tasks.

Liu *et. al.* Liu et al. [2020b] very recently showed that code comprehension has very low overlap with the language centers of the brain, in line with the results we

present in this work. They present 17 expert programmers with two code-related tasks - the first is similar to Siegmund *et. al.*, where participants determine code output. The second requires participants to memorize what they call 'fake code' – code snippets with scrambled tokens in each line – and confirm the presence of a specific substring. They further administer math, logic, and language tasks to locate brain regions involved in these functions in every participant. They report an overlap of code activity with regions belonging to the MD system but not the language centers.

We pose the same question that Siegmund *et. al.* and Liu *et. al.* study. We differ though in our experiment design and workflow. We compare our design choices to these works in detail in Section 5.4. We shall see that this leads to a different set of conclusions than those of Siegmund *et. al.*

5.3 Background

We provide a brief background on fMRI studies, what is measured by such scanning machines, and the regions of the brain we investigate.

5.3.1 fMRI studies

Functional magnetic resonance imaging (fMRI) is typically used to identify regions of the brain which respond to any cognitive task (comprehending code, in our case). MRI machines can mark out and show brain responses in the order of a million voxels while sampling every few seconds [Glover, 2011]. A voxel is roughly the 3-dimensional equivalent of a pixel, and spans a few cubic millimeters of our brains.

When a brain region is involved in a cognitive task, blood flows into the region to aid its processing. An MRI machine measures this change in blood-flow, and reports BOLD (blood oxygen level dependent) values sampled at the machine's frequency. Following common practice, the parameters of a general linear model, fit to these time-varying values, are used as a metric for brain activity. We provide details in Section 5.5.

5.3.2 Regions of Interest (ROIs)

We investigate whether two well-studied systems of brain regions – the MD system and the Language system, which we know how to locate, are also activated when we comprehend code. A *region* (also referred to as *parcel*) here denotes a contiguous chunk of brain mass involved in a cognitive task. A *system* of regions (also referred to as a *network*) can comprise multiple disjoint (at the cortical level) regions, all involved in the same cognitive task.

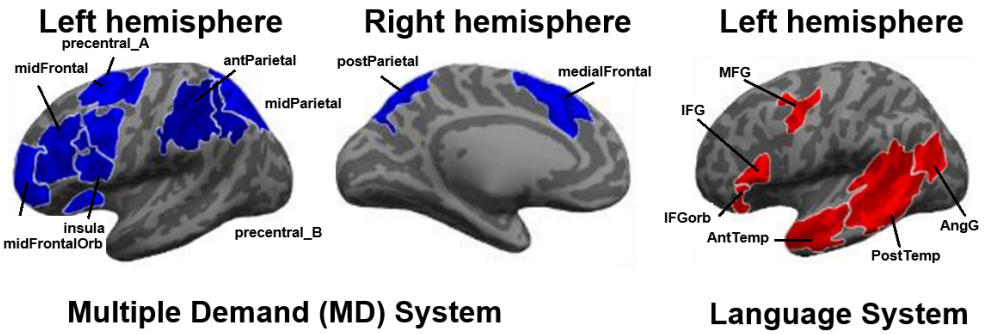


Figure 5-1: The Multiple demand (MD) system and Language system highlighted in a neurotypical adult brain. These two systems span multiple, closely situated regions in the brain, and have been established to have very different response profiles. What is conventionally referred to as Broca's region includes portions of both these systems [Fedorenko and Blank, 2020].

Multiple Demand (MD) system. Since programming conceivably involves arithmetic and general logic skills, we investigate whether the Multiple Demand system [Duncan, 2010], the most prominent system known to support these skills, is activated. Generally located in the prefrontal and parietal areas of the brain, this system of regions is known to be domain-agnostic, and is activated in a host of tasks requiring working memory and general problem solving skills, including math and logic [Duncan, 2010, Amalric and Dehaene, 2019].

Language system. Another possible candidate for processing code is the Language system. These regions have been identified to respond to both comprehension and production of language across modalities (written, speech, sign language), respond to typologically diverse languages ($>$ 50 languages, from across 10 language families),

form a functionally integrated system, reliably and robustly track linguistic stimuli, and have been shown to be causally important for language [Fedorenko et al., 2010a, Clark and Cummings, 2003, Blank and Fedorenko, 2017, Shain et al., 2019a, Mineroff et al., 2018, Blank et al., 2014].

Figure 6-1 shows approximate locations of these systems in a neurotypical adult brain. These systems have been consistently located roughly in the same parts of the brain across individuals [Fedorenko et al., 2010a, 2013]. While an ROI provides a set of broad regions observed to be involved in a cognitive task across individuals, we further locate *functional* ROIs (fROIs) – specific voxels within these broad regions which respond to working memory and language respectively *in an individual*. By doing this, we account for the exact anatomical locations of these voxels, which vary across individuals. This is one improved aspect of our experiment method over prior works. We provide details on fROIs in Section 5.4.4.

5.4 Experiment Design

We first provide a summary of our overall workflow. We follow that with details on three key components of our experiment design: **condition design** - the various design choices we consider in creating the code stimuli we show our participants, **fMRI tasks** - the tasks participants respond to in an MRI machine which enable measuring brain activities, and **processing fMRI data** - how we analyze participants' fMRI data and quantify the effect of *code comprehension*. In our description of these components, we also contrast how they differ from previous works.

5.4.1 Experiment workflow - An overview

The first step of our workflow is to frame hypotheses and design conditions which can test those hypotheses. These conditions inform the stimuli and tasks we present to human participants in an MRI machine. Our goal is to observe the effect reading code has on two regions of interest in our brains - the MD system and the Language system. We first determine which voxels (fROIs) belong to the MD and the Language systems

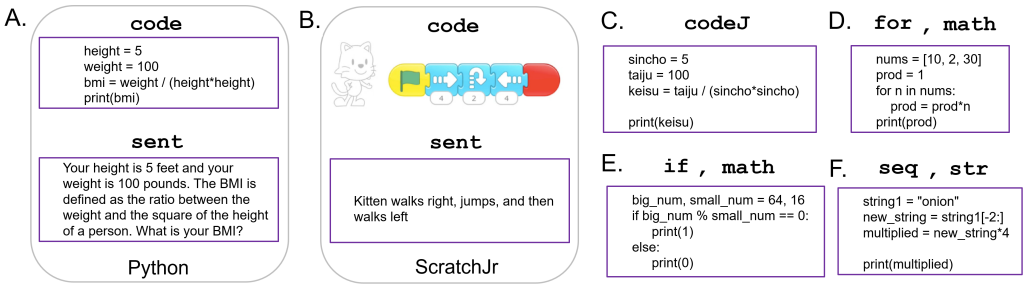


Figure 5-2: (A) A `code` condition stimulus in Python and its equivalent `sent` condition, which describes the `code` stimulus in words. `sent` controls for brain responses to *code simulation*. The difference in these conditions, `code`>`sent`, estimates *code comprehension*. (B) An example `code` and `sent` stimulus in ScratchJr, a programming system with a visual interface. ScratchJr allows to measure the effect of text in codes. (C) `codeJ` condition with Japanese variable names, which controls for the effect of meaningful variable names. (D, E, F) Conditions that measure the effect of control-flow properties (`for, if, seq`) and type of operations (`math, str`).

in each participant. We do this by getting participants to respond to *localizer tasks* – tasks which have been shown to consistently activate the two systems [Fedorenko et al., 2010a, Blank et al., 2014]. We then show participants stimuli corresponding to our own carefully designed code conditions, and we measure brain responses to these conditions within the identified fROIs. The goal of analyzing fMRI responses to our code conditions is to evaluate whether they activate the fROIs as much as the localizer tasks. If they do activate the regions as much, we infer that the fROIs are involved in processing code. For example, if comprehending code activates the Language system as much as comprehending English text (the localizer task for the Language system), we then conclude that the Language system is involved in processing code comprehension in addition to processing language comprehension.

5.4.2 Condition design

Brain activity measurements for a given condition (*e.g.* response to reading codes) can meaningfully be interpreted only relative to another condition (*e.g.* response to reading plain text), *i.e.* by *contrasting* two or more conditions. We describe the different conditions we design and contrast in our work, and discuss them in light of the design choices made by prior works.

Controlling for `non-codes`. The simplest condition pair to observe the effect of reading code is by contrasting `codes` with `non-codes` (notated as `code` > `non-code` in the cognitive neuroscience literature). Here, `non-codes` correspond to stimuli which participants can comprehend despite not being code-like. In our study, they correspond to statements in a natural language (English).

Our goal though is to push farther. We design conditions which help isolate the effect of other factors which might alternatively explain the activations we observe in different brain regions when understanding code.

Controlling for *code simulation*. Arguably, the task of reading code involves more than the act of *code comprehension*. To appreciate why, consider the different cognitive steps involved in reading and understanding code. On being presented code – 1) Retinal cells are activated by the presence of characters in a program 2) The visual system of our brain processes these characters. 3) Having recognized the characters, our brain interprets tokens present in the text. 4) Our brain groups tokens to recognize program statements, and eventually groups these statements to form a mental representation of the entire code, and understands its goal. 5) Our brain executes or simulates it to derive its final output.

In our work, we do not study the effects of reading code on the visual system (steps 1-2). We identify steps 3-4 as *code comprehension*, and step 5 as carrying out *code simulation*– which has also been referred to as *program tracing* [Soloway, 1986], and *processing code content* [Ivanova et al., 2020]. For example, comprehending the statement `x=10+20` refers to associating this statement with the notion '`x` stores the sum of numbers 10 and 20'. *Code simulation* in this case refers to mentally adding numbers 10 and 20 and realizing that `x` stores 30. We refer to steps 3-5 collectively as *code reading*.

Step 5 can potentially dominate brain measurements made when reading and understanding code. To factor out its effect, we offer the following insight – it is possible to describe code in different ways while retaining its *code simulation* operations. A code described in sentences or as a flow diagram does not alter its operations. Drawing on this insight, we design sentences whose content matches our

code conditions. We notate this condition as `sent` and the contrast as `code` > `sent`. See Figure 5-2.A. for an example. If the `code` condition measures *code comprehension* and *code simulation* as the dominant cognitive steps involved, the `sent` condition then arguably measures natural language (sentences) processing and *code simulation*. The difference in these two conditions `code` > `sent` thus allows us to isolate and measure the act of *code comprehension*.

Controlling for variable names. If *code comprehension* is indeed treated like language comprehension and the Language system is found to respond to it, it is reasonable to question whether the Language system responses are caused just by the presence of meaningful variable names and not other aspects of the code. We control for this possibility by replacing variable names with those which mean nothing in that context. The responses to such codes can then be attributed solely to *code comprehension* and not to the presence of meaningful English words in the code. In our work, we chose to rename variables with their Japanese equivalent names (written out in the English script) and administer it to participants with no knowledge of Japanese. We refer to this condition as `codeJ`. Figure 5-2.B shows the code in Figure 5-2.A instead with Japanese variable names. We also account for the effect meaningful string literals (*e.g.* `x="hello"`) or meaningful keywords (`for`, `if`) may have, by designing an equal number of stimuli without these artifacts (discussed in the following point).

Effect of control flow and operations. We additionally investigate whether brain activations to code are consistent across different code properties. This helps demonstrate the robustness of our observations to common variations possible in code. We test two such properties – control flow, and the types of operations. In control flow, we test each of loops (`for`), conditional statements (`if`), and sequential statements. See Figures 5-2.D, E, F for examples of each of these conditions. We test two types of operations – math and string. Figures 5-2.E, F show examples of `math` and str operations respectively. Every stimulus in these conditions has exactly one each of the three control structures, and one of the two data operations. This design also accounts for the presence of meaningful string literals and keywords by allowing us to observe brain activity corresponding to conditions that do not contain these artifacts

(`math`, `seq` respectively).

Effect of text in codes. We experiment with the conditions we describe above in two programming languages – Python and ScratchJr. ScratchJr is a programming system with a fully visual interface [Bers, 2018]. It is generally introduced to children as means to express themselves creatively, where the visual interface and intuitive drag-and-drop features representing different programming constructs enable them to code without relying on a language like English [Bers et al., 2019]. The very nature of this visual interface allows us rule out the influence of text on *code comprehension*. Figure 5-2.B shows an example. Further, using ScratchJr as a second programming language helps validate the generalizability of our findings. All prior works have evaluated their findings only in one programming language.

Design choices by prior works. Floyd *et. al.* also use the basic contrast `code` > `non-code`, but nothing more to isolate *code comprehension*. Siegmund *et. al.* instead contrast `code` > `code with syntax errors` (Section 5.2). Codes with syntax errors are still codes, and hence do not help differentiate activity in regions where non-codes (natural language) are known to be processed. Further, the `code-with-syntax-errors` condition likely measures aspects of *code comprehension*, *code simulation*, and perhaps other skills specific to debugging and finding such errors. Thus, their contrast does not fully isolate *code comprehension*. While Liu *et. al.* ensure their `code` stimuli generalize to loops and conditions, their primary contrast `code` > `fake code` also does not distinguish between *code comprehension* and *code simulation*. Their setup introduces the additional effect of memorizing `fake code` which involves multiple cognitive processes

Summary. To summarize, in our Python experiments, our overall experiment design is a $3 \times 3 \times 2$ study – 3 conditions - `code`, `sent`, `codeJ` (Japanese variable names), and within each of these three conditions, we further have 3 categories of control flow conditions, and 2 categories of operations-related conditions. Since many of these conditions are not applicable to ScratchJr (variable names, operation types), we evaluate only the critical `code > sent` condition in ScratchJr.

5.4.3 fMRI tasks

For each participant, in addition to presenting stimuli corresponding to code-related conditions in an MRI machine, we present two separate tasks to *localize* the two regions of interests in them. What is central to a localizer task is its ability to strongly activate a region of interest in every individual. It has been empirically established that reading semantically well-formed sentences in any natural language strongly activates the Language system, while performing spatial memory tasks strongly and distinctly activates the MD system [Fedorenko et al., 2013, 2010a]. We reuse these established localizer tasks in our work. We provide details in Section 5.5.

We now describe how we use this localization information when analyzing brain activity during code comprehension.

5.4.4 Locating fROIs and data analysis

We analyze brain data in the following five key steps. Our procedure follows the Group-constrained Subject-Specific (GSS) method of locating functional regions of interest (fROIs) that are activated consistently across individuals [Nieto-Castañón and Fedorenko, 2012].

1. Mapping to an exemplar brain structure. To normalize differences in brain anatomies, each participant's brain is spatially transformed to an exemplar brain structure like the Montreal Neurological Institute (MNI) template [Tzourio-Mazoyer et al., 2002]. These spatially transformed coordinates are used for subsequent analyses.

2. Selecting ROIs. Regions of interest (ROIs) mark out a set of broad regions observed to be involved in a cognitive task across individuals. For every participant, we use these regions as a starting point, and look for voxels within them which respond to a cognitive task. This helps avoid looking in regions which are not germane to the task. For example, reading code will understandably also activate the visual cortex, which is not of interest to our particular study.

In our work, we reuse a set of 20 MD parcels (10 in each hemisphere) and six Language parcels defined in prior works [Fedorenko et al., 2010a, 2013]. These parcels

have been curated by aggregating ∼200 participants' brain responses to spatial working memory and language tasks respectively. As an alternate, one could select ROIs from the parcels defined by Brodmann's areas [Brodmann, 1909], an atlas which maps regions of an exemplar's brain to cognitive functions.

3. Identifying fROIs. For every individual, a *functional* region of interest (fROI) refers to a collection of voxels within an ROI which respond to the cognitive task the ROI is involved in. Owing to differences in anatomies, the specific set of voxels which respond to a cognitive task (like spatial reasoning or language) varies across individuals. ROIs, aggregated from across individuals, help narrow down the search space to locate these specific voxels in every individual by pointing to a swath of regions known to respond to the task. fROIs in turn identify specific voxels *functionally involved* in the cognitive task. Localizer tasks (Section 5.4.3) for each system help identify these voxels. By the end of this step, we establish in each participant fROIs for the MD and the Language systems. We provide details in Appendix 6-1.

4. Aggregate activation data within a participant. We use the fROIs defined for the two systems in the previous step in all our remaining experiment conditions. Specifically, we measure the activations of our code conditions in the selected fROIs. At this stage, we have at least two sets of activation measurements for each voxel in an fROI – one corresponding to the localizer task, and the others corresponding to the different code-related conditions. For each fROI, we obtain a single response value per condition by averaging the responses of all voxels within the fROI.

5. Aggregate activation data across participants. For each system, we then evaluate whether the distribution of participant-level responses to the code conditions is comparable to that of the localizer task. If it is, we conclude that the system is involved in processing code conditions.

Multi-participant analysis without functional localizers. Among prior works, Liu *et. al.* alone use localizer tasks to find task-selective voxels in individual participants. However, they do not use ROIs (step 2 above) and instead perform a whole-brain analysis, and report overlaps as against measuring exact activations in fROIs. Their setup coupled with their ambiguous condition design (discussed in

Section 5.4.2) makes it hard to infer brain regions accurately.

In fMRI studies which do not use localizer information, as in the case of Siegmund *et. al.*, Floyd *et. al.*, and other works which have studied different aspects of programming, the primary difference is that ROIs are defined based on anatomy, and not on their function (*i.e.* how they respond to localizer tasks). Concretely, this difference arises in steps 3 and 4, where instead of aggregating activations within an fROI, activations are estimated in each voxel across the entire brain and aggregated across participants (also called the *group analysis* procedure). The location of such aggregated active voxels is then described using anatomical labels, such as Brodmann areas [Brodmann, 1909]. This method has broadly been referred to as *reverse inference* in neuroimaging studies [Poldrack, 2011].

The reverse inference method assumes that fROIs are spatially fixed among individuals and can be uniquely located in the exemplar brain structure. While reverse inference is not always a concern, especially when the regions are anatomically well separated and distinct (*e.g.* visual system vs. MD system), it has been shown to yield inaccurate estimates in the measurements of the closely situated MD and the Language systems [Brett et al., 2002, Amunts and Zilles, 2012, Fedorenko et al., 2012, Fedorenko and Blank, 2020]. What is referred to as the language region by Brodmann's areas (areas 44 and 45) in one individual can instead refer to functional regions belonging to the MD system in another individual, owing to differences in individual anatomies [Fedorenko et al., 2012, 2011]. The GSS approach of function-based ROI identification helps circumvent this potential cause for inaccuracy.

5.5 Experiment Procedure

We describe in brief our experiment procedure. We recruited 24 participants for Experiment 1 (Python) and 19 participants for Experiment 2 (ScratchJr), with no overlap between these groups. On the day of the scan, having provided consent, participants spent $1.5 - 2$ hours in the scanner. In Experiment 1, in the week of their scheduled fMRI scan, each participant additionally completed an assessment in

Python to evaluate their fluency in it.

Once in the scanner, a participant was presented with two localizer tasks, adopted from prior works [Fedorenko et al., 2013, 2010a], to locate the MD system and Language system respectively in their brain. The MD system localizer task is a working memory task, presented in two grades of difficulty - easy and hard. The Language system task has two conditions - sentence reading (SR), and non-word reading (NR). SR requires reading sentences which are structurally and semantically meaningful. NR requires reading sentences with pronounceable yet meaningless non-words (*e.g.* `BIZBY ACWORILLY BUSHU SNOOKI BILIBOP KUKEE`). These two conditions serve as references to measure other experiment conditions against – the Language system has been shown to respond strongly to SR while only minimally to NR.

Participants were also presented with coding tasks. The tasks shown were balanced between the three conditions - `codeE` (code with semantically meaningful variable names in English), `sent` (sentences describing programs, controlled for *code simulation*), and `codeJ` (code with Japanese variable names). Each participant saw 72 problems, 24 from each of the three conditions. Each of these set of 24 problems further had an equal number of control-flow and operations-related conditions. Any given participant saw only one of the three versions of a problem.

The data from the localizer scans was used to locate the fROIs in the MD and the Language systems in every participant. We fit a general linear model to the time series brain activation data generated as a response to our different tasks. The parameters of this model (β) are used as a metric for brain activity (BOLD) in all our analyses.

5.6 Results

We present our questions and their corresponding results here. In our results, we discuss the neural activations in different regions of the brain (Figures 5-3.A, 5-3.B). The x-axis in these plots corresponds to the different conditions participants responded to, and the y-axis represents activation strength (β values). Each dot in each bar corresponds to one participant's aggregate activity in the fROIs localized in them.

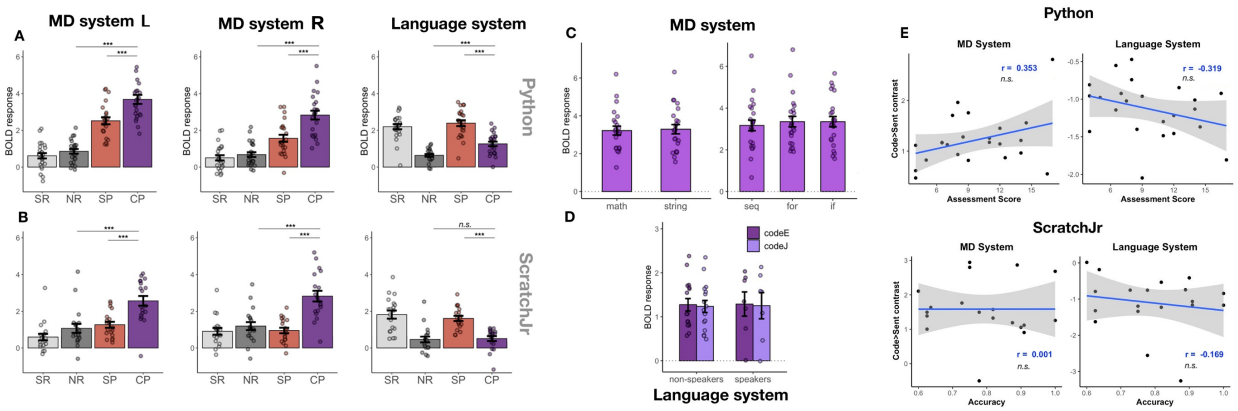


Figure 5-3: (A, B) Brain activations in the MD system left hemisphere (MD system L), MD system right hemisphere (MD system R), and the Language system. We measure responses to four conditions – codes (CP), sentences matching the code's operations (SP), Sentence reading (SR), and Non-words reading (NR). We experiment in Python (N=24) and ScratchJr (N=19). Each dot in the bars corresponds to aggregate data from one participant. *** indicates $p < 0.001$, n.s. - not significant (C) MD system responses to two code properties – operation type (math, string operations), and control-flow (sequential, loop (for), conditional (if)) (D) Language system responses to variable names in English (codeE) and Japanese (codeJ) (E) Correlation of responses in the MD and the Language systems to proficiency in Python (top) and ScratchJr (bottom).

When reporting results of a contrast between any two conditions, we measure the difference in the average β values ($\Delta\beta$) and compute its associated p-value.

RQ 1. Does *code reading* activate the Multiple Demand (MD) system?

Conditions contrasted. code, sentence reading, non-word reading

We begin by investigating whether reading code, which involves both *code comprehension* and *code simulation*, activates the MD system. We do this by comparing the activations of our primary code-related condition - code problems, to the localizer tasks for the Language system - sentence reading and non-word reading. We notate these conditions as CP, SR, and NR respectively in Figure 5-3.A, B. We evaluate two sets of fROIs in the MD system - one in each hemisphere of the brain (Figure 5-3.A, B., left and center plots). From the plots, we see both sentence reading and non-word reading, the language localizer conditions, have minimal activations in the MD system in both hemispheres. This is expected since the MD system is not sensitive to language tasks [Blank et al., 2014]. We find that code problems, which account for both *code*

comprehension and *code simulation*, activate fROIs in both hemispheres of the MD system consistently and significantly more than the baselines in both our experiments (Python: $\Delta\beta = 2.17$, $p < 0.001$; ScratchJr: $\Delta\beta = 1.23$, $p < 0.001$). This suggests that the MD system is involved in reading code.

We confirm whether these responses are consistent across code properties, which will establish its robustness to the variations possible in code. We test two properties – control-flow (sequential, `for`, `if`), and types of data manipulated in them (string, math operations) in Python. We observe strong responses regardless of the type of operations and control flow (Figure 5-3.C; y-axis is response to CP). We thus conclude that the responses in the MD system to code problems were not a result of any one particular type of problem, or mental operations related to a particular control flow.

This clearly identifies and establishes the role of the MD system in *code reading*. Prior works did not identify and study this system of regions.

RQ 1 result. Yes, *code reading* activates the MD system. Its responses are independent of the control-flow operations and types of data operations present in codes.

RQ 2. Does *code comprehension* activate the Multiple Demand (MD) system?
Conditions contrasted. `code`, `sent`, `sentence reading`, `non-word reading`

Since we find that *code reading* activates the MD system, we investigate whether these were responses to *code comprehension* or *code simulation*. To answer this, we study the effect of both our code-related conditions – code problems (CP), and sentence problems which match the code problems for their content (SP). We find that sentence problems, which measure only *code simulation* and not *code comprehension*, activate the MD system significantly greater than the language localizer baselines in both hemispheres only for Python (left: $\Delta\beta = 1.51$, $p < 0.001$; right: $\Delta\beta = 0.78$, $p < 0.001$). This activation is not significant for ScratchJr (left: $\Delta\beta = 0.09$, $p = 0.93$; right: $\Delta\beta = -0.40$, $p = 0.004$), suggesting that *code simulation* is not consistently supported by the MD. However, we find that code problems, which measure both

code comprehension and *code simulation*, strongly activate fROIs in both hemispheres. This is despite sentence problems taking slightly longer on average to respond to. We hence find that CP strongly activates the MD system and SP does not. This implies that the difference CP > SP, which measures *code comprehension*, strongly activates it. This is strong evidence for the MD system's consistent and robust activation to *code comprehension*, and shows it is not just a response to the underlying *code simulation* operations.

We investigate further for any hemispheric bias towards *code comprehension*. Previous works have shown that math and logic problems typically activate the MD system in the left-hemisphere of the brain [Amalric and Dehaene, 2016, 2019]. We did not find any such bias in Python (MD-L plot, Figure 5-3.A). In ScratchJr, we observe stronger responses in the right hemisphere ($\Delta\beta = 0.57$, $p < 0.001$; MD-R plot, 5-3.B), perhaps reflecting a known bias of the right-hemisphere towards visuo-spatial processing [Sheremata et al., 2010].

Follow up analyses of activity within individual regions within the MD system showed that 17 of the 20 fROIs in the Python experiment, and 14 of the 20 fROIs in the ScratchJr experiment responded significantly more strongly to code problems than to sentence problems. This demonstrates code processing is broadly distributed across the MD system and is not localized to a particular subset of regions within it. Within this activated subset, we evaluate whether any fROIs are *selective* to code problems in comparison to other cognitively demanding tasks which activate the MD system. We find none for ScratchJr, and three regions in the frontal lobe (precentral-A, precentral-B, midFrontal) which exhibit stronger responses to Python code problems than to the hard working memory localizer task for the MD system. However, the magnitude of code > sent in these regions ($\Delta\beta = 1.03, 0.95, 0.97$) was comparable to the mean magnitude across all MD system fROIs (average $\Delta\beta = 1.03$), suggesting that the high response was caused by the underlying *code simulation* rather than *code comprehension*. We conclude that *code comprehension* is broadly supported by the MD system, and no specific regions in the MD system are functionally specialized for it.

These new results further establish the role of the MD system in processing *code comprehension*, which we narrowly and clearly define in this work.

RQ 2 result. Yes, *code comprehension* consistently activates the MD system. Unlike math and logic, it activates fROIs in both the left and right hemispheres. In fact, no specific fROI within the MD system specializes for *code comprehension*, and it is instead broadly supported by the entire system.

RQ 3. Does *code reading* activate the Language system?

Conditions contrasted. `code`, `sent`, `sentence reading`, `non-word reading`

We investigate the Language system similarly for responses to our code conditions. Figures 5-3.A, B (rightmost plot) show the aggregate responses in the Language system to the two code conditions and the two language localizer conditions described above. As expected of the localizers, we find the activations of sentence reading to be significantly greater than non-word reading [Blank et al., 2014, Fedorenko et al., 2010a]. Among the code conditions, we find that sentence problems activate the Language system as much as the sentence localizer task in both Python and ScratchJr. This is again expected since sentence problems contain English sentences describing what the program does (Figure 5-2.A). However, the responses to code problems were weaker than responses to sentence problems in both experiments (Python: $\Delta\beta = 0.98$, $p < 0.001$, ScratchJr: $\Delta\beta = 0.99$, $p < 0.001$). This observation alone does not yield any insight on whether code activates the Language system, and we hence compare these activations to the localizer baseline non-word reading. Non-word reading is a lower bound for activity in the Language system; this is the activity seen in the Language system when it is not actively engaged in linguistic interpretation. Responses to the code condition were stronger than non-word reading only in the Python experiment ($\Delta\beta = 0.78$, $p < 0.001$) but not in the ScratchJr experiment ($\Delta\beta = 0.15$, $p = 0.29$), implying that code does not consistently activate the Language system.

The result from this principled investigation of the Language system is contrary to that of Siegmund *et. al.*, who report the involvement of the language system in

addition to other brain regions. We discuss this further in Section 5.7.

RQ 3 result. No, *code reading* does not consistently activate the Language system.

RQ 4. Do meaningful variable names affect the Language system's response to code?

Conditions contrasted. `codeE`, `codeJ`

Since we find that Python code activates the Language system but ScratchJr does not, we investigate whether this is a consequence of meaningful variable names present in codes. To study this effect, we had participants read half the Python code problems with semantically meaningful variable names in English (`codeE`) and the other half with Japanese variable names (`codeJ`), making them semantically meaningless; 18 of the 24 participants reported no knowledge of Japanese. In the Language system, we found no effect of meaningful variable names ($\Delta\beta$= 0.03, p = 0.84) (Figure 5-3.D, non-speakers), knowledge of Japanese ($\Delta\beta$= 0.03, p = 0.93) (Figure 5-3.D, speakers), nor any interaction between the two ($\Delta\beta$= 0.09, p = 0.71), suggesting that the Language system response was not affected by the presence of semantically meaningful variable names. This result is surprising since the Language system has been shown to be very sensitive to word meaning [Anderson et al., 2019]. A possible explanation is that participants do not fully engage with the words' meanings to solve problems.

RQ 4 result. Meaningful variable names do not affect the Language system's response to code.

RQ 5. Are there regions outside the MD system and Language system that respond to *code comprehension*?

To search for regions responsive to *code comprehension* outside the MD system and Language system, we perform a whole-brain Group-constrained Subject Specific

analysis. For both Python and ScratchJr, we search for brain areas with activations where code > sent. We then examine the response of such regions to code and sentence problems (cross-validated with held-out data), as well as to conditions from the two localizer experiments. In both experiments, the discovered regions spatially resembled the MD system. For Python, any region that responded to code also responded to the spatial working memory task (MD system localizer). In case of ScratchJr, some fROIs responded more strongly to code problems than to the spatial working memory task; these fROIs were located in early visual areas/ventral visual stream which likely responded to low-level visual properties of ScratchJr code (which contains colorful icons, objects, *etc.*). These whole-brain analyses demonstrate that the MD system responds robustly and consistently to *code comprehension*, confirming the results of the fROI-based analyses discussed in RQs 1 and 3. This further shows that fROI-based analyses did not miss any non-visual regions outside the boundaries of the MD and the Language systems that was activated by *code comprehension*.

RQ 5 result. We found no code-selective regions outside the MD and the Language systems.

RQ 6. Does expertise affect how the MD and the Language systems respond to *code comprehension*?

We study the role of expertise by correlating responses within each system with independently obtained proficiency scores for participants of Experiment 1 (a 1-hour Python assessment module) and with in-scanner accuracy scores for Experiment 2 participants. Figure 5-3.E plots the percentage proficiency scores (x-axis) against *code comprehension* (code > sent). No correlations were significant. However, due to a relatively low number of participants (N = 24 and N = 19, respectively), these results should be interpreted with caution.

RQ 6 result. We did not have enough data to observe the effect of expertise on the MD and the Language systems' responses to *code comprehension*.

5.7 Discussion

We present a new set of results which improves our understanding of the cognitive bases of program comprehension. We find that *code reading* (which we identify to include both *code comprehension* and *code simulation*) strongly activates only the MD system and not the Language system. Despite their anatomical proximity in the left-hemisphere of our brains, our work clearly distinguishes the roles of both these systems by means of functional localizers.

MD system results. We find that the MD system consistently processes *code reading*. We support our observations by showing these activations generalize across two code properties - control flow and data operations, suggesting that the system's response is robust to variations in code. We further learn that the MD system responds consistently to *code comprehension*. It also responds to *code simulation* strongly in Python, but we see only a weak evidence for it in ScratchJr, which needs to be investigated in future work. It is reasonable to expect the MD system to process *code reading*, since both *code comprehension* and *simulation* requires attention, working memory, inhibitory control, planning, and general flexible relational reasoning - cognitive processes long linked to the MD system [Duncan and Owen, 2000, Duncan, 2010]. This finding also supports Liu *et. al.*'s recent results [Liu et al., 2020b]. Since no other regions outside the MD system responded to codes, we posit this system stores code-specific knowledge in addition to processing it. This knowledge likely includes concepts specific to a programming language (*e.g.* the syntax marking an array in Java vs. Python) and concepts shared across languages (loops, conditions). Evidence from studies on processing mathematics and physics [Fischer et al., 2016, Amalric and Dehaene, 2019] has shown that the MD system can store some domain-specific representations in the long term, perhaps for evolutionarily late-emerging and late-acquired domains of

knowledge. In conclusion, we identify a known brain system, which had previously not been studied for its role in *code reading* tasks, to be involved in *code comprehension* specifically.

Language system results. We importantly establish in this work that *code reading* is *not* consistently processed by the Language system. This is a new finding, and adds to the results from Siegmund *et. al.* and Floyd *et. al.*, while confirming results from Liu *et. al.*. Siegmund *et. al.* report the involvement of the language centers in *code comprehension* by showing evidence of left-lateralized brain activity. While it is unclear whether their observations were technically from the Language system or the MD system, we suspect that they observed *code simulation* and not *code comprehension*. Additionally, and surprisingly, we find that the Language system is insensitive to the presence of meaningful variable names. More work is required to determine why the Language system showed some activity in response to Python code.

The Language system does not respond consistently to *code comprehension* despite numerous similarities between code and natural language. However, the lack of consistent Language system engagement in *code comprehension* does not mean that the mechanisms underlying language and code processing are completely different. It is possible that both the MD and the Language systems have similarly organized neural circuits that allow them to map a symbol to a concept. However, the fact that we observed code-related activity primarily in the MD system indicates that *code comprehension* does not activate the same neural circuits as language, and needs to use domain-general MD system circuits instead.

Having identified the regions activated when reading programs, we discuss how our results affect the programming languages (PL), software engineering (SE), and CS education (CS-Ed) communities.

Impact on the PL, SE, CS-Ed communities. To understand how our results can be applied specifically to *improving* how we can understand programs, we first establish the relationship between two cognitive activities engaging the same brain system (in our case - working memory tasks and *code comprehension* engaging the MD system). A few studies have claimed that for any two cognitive activities that share the same

brain resources, training one activity will lead to an improvement in the other [Jaeggi et al., 2008, Melby-Lervåg and Hulme, 2013]. For example, if language and music share and activate the same brain system, then tools and approaches used to engage and train one activity should be transferable to, and will lead to an improvement in the other. Since the effects of training and improving one's MD system are not well understood, it is unclear whether training on cognitively demanding non-coding tasks could improve our ability to read and understand programs.

Based on the opposite conclusions presented in Siegmund *et. al.*, Portnoff *et. al.* [Portnoff, 2018] and similar other works suggest adopting a "languages first" approach when teaching programming. Evidence from our work does not support this claim, and we caution against adopting practices that are used to teach natural languages for programming just based on conclusions from recent neuroimaging studies.

The Language system not being involved in *code comprehension* should not diminish the role of language in understanding programs. The use of poorly named variables has been shown to increase cognitive load [Fakhoury et al., 2019], and non-native English speakers have been found to often struggle with learning English-based programming languages [Guo, 2018]. Future work should reconcile this disparity, and aim to show how results from studies on cognition can aid understanding programs.

ML models for code. Recent advances in machine learning (ML) models trained on large corpora of programs have shown models' ability to perform tasks like renaming functions, bug detection, *etc.* [Allamanis et al., 2018a]. Deep networks learn a 'language model' of programs, and likely learn a generalized way to represent these programs. Do these model-learned representations correspond to the representations (activation data) in the different fROIs from our study? Such a correspondence may have far reaching implications. For instance, if we can probe and isolate specific weights or layers that encode loops and recursion in a recurrent model like seq2seq, code2seq, or GPT-3, the existence of a correspondence between representations may help locate the encodings of loops and recursion in our brains. Such a correspondence has recently been established between representations stored in the visual cortex and those learned by deep convolutional networks for image processing and recognition

[Yamins et al., 2014, Khaligh-Razavi and Kriegeskorte, 2014, Cadena et al., 2019]. This promises to be a compelling direction for future work.

5.8 Threats to validity

There are limitations to the results we report in this work. One possible threat is posed by the tasks we designed – do our programming tasks measure code comprehension and understanding? The programs we present in this study are short snippets of procedural code with limited program properties. They do not have complex control and data dependencies generally seen in production-grade programming projects. Properties like function calls, objects, types, complex data, and state changes in the program are not included either and should be studied in the future, building on the understanding of simpler snippets provided by our work. Further, we study a very specific activity related to programming – reading programs, and do not investigate other equally important aspects like designing solutions, selecting appropriate data structures, and writing programs.

In designing our code stimuli, having overtly informative variables names poses the risk of participants not going through all the lines of code presented to them, and instead just guessing what the code does by gleaning variable names. To avoid this, we constrain our variable names to be informative and natural (as it would appear in a real codebase) to the extent they do not reveal fully the intent of the entire code snippet. However, such a constraint does not appear in actual coding scenarios. Disparate stimuli are a source of possible confounds. If some conditions had disproportionately longer code than others, it would be unclear if any trend we saw in brain activations were a function of the condition, or such factors like code length. In an attempt at avoiding this, we ensured that the stimuli in our $3 \times 3 \times 2$ conditions had similar code lengths and overall response times. However, they are not all equal.

Expertise is also a potential confound which could affect the generalizability of the results we see. The majority of our participants were recruited from a university setting and had roughly 3-6 years of programming experience. While our participants'

experience level was largely homogeneous in our study, expertise could interact with brain functions associated with program comprehension, as it does with other cognitive functions [Agrawal et al., 2018, Hoenig et al., 2011, Gomez et al., 2019]. Accounting for the role of expertise would require running these experiments on a population with varying proficiencies.

Neuroimaging experiments generally risk lack of generalizability of results owing to low sample sizes. In our work, the relatively small sample sizes (N=24 for Experiment 1, and N=19 for Experiment 2) affect only our group analyses (of comparing aggregate information across participants). Although these sample sizes are the norm in the neuroimaging community, the robustness of our results are limited by the small number of participants. We see this as an opportunity for authors from similar neuroimaging studies to collaborate to analyze data across these works, which will also help amortize the high costs of carrying out this type of experiments.

Chapter 6

Convergent representations of computer programs in humans and code models

Preface. This chapter, in full, is a re-print of **Convergent representations of computer programs in human and artificial neural networks.** Srikant*, S., Lipkin*, B., Ivanova, A. A., Fedorenko, E., and O'Reilly, U.M. (2022). NeurIPS 2022 [Srikant et al., 2022]. Ben contributed equally with me as the primary author of this work.

Refer to Section 1.4 to read more about the motivation behind this work, and how it connects to the rest of the chapters presented in this thesis.

6.1 Introduction

Computer code comprehension is a complex task which recruits multiple cognitive processes—from syntactic parsing to mentally simulating programs. Despite the prevalence of this task, the representations of code processed in the human brain during code comprehension remain uninvestigated. Is it possible that common code properties and program semantics are faithfully represented in brain activity patterns when code is read and evaluated? A few prior works have used data derived from

functional magnetic resonance imaging (fMRI) and electroencephalography (EEG) to locate physical regions in the brain involved in code-related activities like code comprehension and debugging [Siegmund et al., 2017, Floyd et al., 2017, Peitek et al., 2018, Castelhano et al., 2019, Ivanova et al., 2020, Liu et al., 2020b, Ikutani et al., 2021, Peitek et al., 2021], code writing [Krueger et al., 2020, Karas et al., 2021], and data structure manipulation [Huang et al., 2019]. These studies have helped determine whether code-related activities join other activities supported by those brain regions, such as working memory (processed by a set of brain regions known as the Multiple Demand system) or language processing (processed by the Language system–another set of brain regions). While these results improve our understanding of the brain regions involved in code comprehension, it still remains unclear what specific code-related information these regions encode. For example, does the response in a region seen during code comprehension encode specific syntactic or semantic code properties? Do responses from multiple brain regions correspond to the same set of properties? Or, are different code properties encoded in different regions?

A possible approach and its limitation. One way to learn what information is encoded in the brain is to decode a code property of interest from recordings of brain signals when reading code (through fMRI or EEG). Being able to decode the property accurately from a specific region of the brain establishes that information related to that code property is faithfully represented in that brain region. A question central to such a decoding analysis is the choice of the target code property–what code properties should be investigated? We can hand-select a set of fundamental properties of code and test if they can be decoded. While helpful, such a set will not preclude other, more complex aspects of code possibly being encoded.

ML models of code as a tool to reverse engineer what is encoded. To address the limited scope of hand-selected properties, we look to machine learning (ML) models trained on code. Dubbed *code models*, they are trained on large corpora of code, in an unsupervised manner, to learn *ML model representations* of computer programs. Code models are increasingly being used in software engineering workflows [Allamanis et al., 2018a], and have been shown to perform well on tasks like code

summarization [Alon et al., 2019a], detecting variable misuse [Bichsel et al., 2016], and more recently, code auto-completion [Chen et al., 2021a]. These representations have been shown to encode and describe complex code properties [Bichsel et al., 2016, Allamanis et al., 2018d, Srikant et al., 2021]. Successfully decoding these code model representations from brain activity data then allows us to probe whether complex code properties are also encoded in the brain.

Besides serving as a tool to reverse engineer what is encoded in different brain regions, another distinct advantage of decoding code model representations is its potential to serve as a tool to reverse engineer our cognitive processes. It is currently unclear what mechanisms drive code comprehension. If we find one class of ML models (say, masked language models) to be more predictive than another (say, autoencoders), it is reasonable to suspect that our brains optimize objectives more similar to that of masked-LMs than that of autoencoders when comprehending code. As a corollary, a poor correspondence between the information encoded by our brains and ML models suggests the possibility of unexplored neural architectures and objectives which may better model our cognition, which in turn may outperform extant ML models. [Yamins et al., 2013] first showed how information encoded in our visual system resembles what convolutional neural networks learn when trained to recognize images. We attempt to establish a similar correspondence between code models and the human brain in comprehending code.

Why code comprehension? We focus on code comprehension in this work because very little of this important skill has been analyzed from a cognitive neuroscience perspective while steady advances are being made in training ML models to understand code and increase programmer productivity [Fedorenko et al., 2019, Hellendoorn and Sawant, 2021]. Extant ML models for understanding programming are direct adoptions of the state-of-the-art in language processing research. However, recent works in neuroimaging like Liu et al. [2020b] and Ivanova et al. [2020] suggest that code comprehension does not share the same neural bases as natural language comprehension. Do code models then mimic human cognition of programs? If not, can we consider other model architectures and training objectives which are directly inspired by results

from neuroscience? Our work provides early considerations towards these directions.

Further, characterizing the representations of different code properties in the human brain can inform us about the nature of human algorithmic problem solving more generally. We can possibly characterize signatures of brain activity corresponding to fundamental logic operators like iterative reasoning, conditional reasoning, retrieving calculations that were previously computed, *etc.* seen in individuals when solving carefully designed code comprehension tasks. Such signatures will then allow us to characterize other tasks which involve similar logic operators, but which are not naturally described as computer programs *e.g.* general logic reasoning, problem solving, and more generally, our ability to employ abstractions and form concepts [Rule et al., 2020].

Our setup. We introduce a framework to evaluate the code properties encoded in human brain representations of code by analyzing fMRI recordings of programmers comprehending Python programs. We present two means of proceeding— one, probing of brain region representations for specific code properties, and two, analyzing the mapping of these representations onto various code models with differing model complexity. We utilize the publicly available dataset from [Ivanova et al., 2020] for all our analyses as it offers high quality, granular brain response data on code comprehension stimuli controlled across multiple code properties. We train affine models on brain representations to predict hand-selected code properties which express syntactic and semantic behavior of programs. Similarly, we predict code representations obtained from a suite of code models. We investigate the effects that brain regions, the nature of code properties, and the complexity of the code models have on the accuracies of these prediction tasks.

Key findings. We explore two questions: (1) Do brain systems encode specific code properties? Are there differences in how well each brain system encodes each code property? (2) Do brain systems encode more complex properties of code, derived from the representations of code models? To answer these, we perform three critical comparisons: between code properties and code models, code models and brain systems, and brain systems and code models. We further provide a preliminary analysis into the

relationship between code models and the brain in the context of what properties code model representations encode. We find the Multiple Demand system and the Language system consistently encode both hand-selected code properties and data-derived code model representations, using features which cannot be explained by only low-level visual characteristics of the code. Within this set, the Multiple Demand system most effectively decodes runtime properties of code like the number of steps involved in a code's execution, while the Language system decodes syntax-related properties like the number of tokens in a program and the control structures present in it. These results improve our understanding of the functional organization of the two brain systems—MD and the Language systems—that we study, and provide initial evidence for incorporating the roles of these two distinct brain systems in the design of training objectives of code models.

We provide an open-source framework to replicate our experiments, and we release our data and analysis publicly. Link - `https://github.com/ALFA-group/code-representations-ml-brain`. This should enable authors from other neuroimaging studies or code model developers to collaborate and analyze data across these works, which will also help amortize the high costs of carrying out such experiments.

6.2 Related Work

Of the prior works that have investigated the neural bases of programming through fMRI and EEG techniques [Siegmund et al., 2017, Floyd et al., 2017, Peitek et al., 2018, Castelhano et al., 2019, Huang et al., 2019, Krueger et al., 2020, Ivanova et al., 2020, Liu et al., 2020b, Ikutani et al., 2021, Peitek et al., 2021] and through behavioral studies [Prat et al., 2020, Casalnuovo et al., 2020a, Crichton et al., 2021], the following probe brain recordings for program properties encoded in them.

Floyd et al. [2017] learn a linear model to successfully classify whether an observed brain activity corresponds to reading code or reading text. Ikutani et al. [2021] study expert programmers and show that it is possible to classify code into the four problem categories–math, search, sort, and string from the brain activations corresponding to

the code. Similarly, Liu et al. [2020b] classify whether a brain signal corresponds to code implementing an `if` condition or not. Peitek et al. [2021] analyze correlations between brain recordings of participants reading code and a set of code complexity metrics.

In testing for code properties, our work uses a similar methodology (a linear model trained on fMRI data), but we evaluate a larger set of static and dynamic code properties, often reflecting key programming constructs like control flow. Further, we perform these tests in the brain regions identified by Liu et al. [2020b] and Ivanova et al. [2020] as being responsive specifically to code comprehension, offering finer insight into the content of these specific regions' representations. In addition, we study representations generated by a suite of ML models with varying complexity and compare those learned representations to brain representations.

Brain representations have also been studied in domains like natural language, vision, and motor control. Among related works in natural language, a domain that resembles programming languages, Mitchell et al. [2008], Pallier et al. [2011], Brennan and Pylkkänen [2017], Jain and Huth [2018], Gauthier and Levy [2019], Schwartz et al. [2019], Wang et al. [2020b], Schrimpf et al. [2021a], Cao et al. [2021], Caucheteux et al. [2021], Toneva and Wehbe [2019] have studied brain representations of words and sentences by relating them to representations produced by language models. While the broader tools we use to investigate these representations, like multi-voxel pattern analysis (MVPA), are similar to some of these prior works, our focus is on properties specific to code and not natural language.

6.3 Background

We provide a brief background on fMRI signals as a proxy for brain representations and describe the brain systems that we probe in this work.

Measuring brain activity with fMRI. Functional magnetic resonance imaging (fMRI) is a brain imaging technique used to measure brain activity in specific brain regions. When a brain region is active, blood flows into the region to aid its processing.

An MRI machine measures this change in blood flow, and reports BOLD (blood oxygen level dependent) values sampled at the machine's frequency [Glover, 2011, usually 2 seconds]. The smallest unit of brain tissue for which BOLD signal is recorded is called a voxel (an equivalent of a 3D pixel); it comprises several cubic millimeters of brain tissue. For our analyses, we select subsets of voxels belonging to specific brain systems. Following common practice, the parameters of a general linear model, fit to time-varying BOLD values, are used as a measure of the overall activation in each voxel in response to a given input. It is the values of these parameters that, in concordance with common practices in the neuroscience community, can be accessed as brain representations.

Brain systems. A system of brain regions can span different areas of the brain but behaves as a holistic unit, showing similar patterns of engagement across a given cognitive task. We probe the following systems in our work: (a) **Multiple Demand (MD) system:** this system of regions is known to engage in cognitively demanding, domain agnostic tasks like problem solving, logic, and spatial memory tasks. Liu et al. [2020b] and Ivanova et al. [2020] reported that this system is active during code comprehension. (b) **Language system (LS):** this system responds during language production and comprehension across modalities (speech, text) and languages (across 11 language families, including American sign language). (c) **Visual system (VS):** these regions respond primarily to visual inputs. (d) **Auditory system:** these regions respond primarily to auditory inputs.

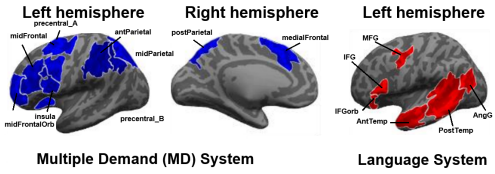


Figure 6-1: The approximate locations of MD and the Language systems in the human brain. The regions depicted are used as a starting point to functionally localize these systems in individual participants.

We probe the Visual system and Auditory system since they are involved in general perception. While we do not expect activity in Auditory system, we expect the Visual system to reflect low-level visual properties of the code (*e.g.* code length and indentation to reflect code-related properties.

6.4 Brain and Model Representations

We describe in this section the method we follow to gather representations of code in the brain (Section 6.4.1), evaluate the different code properties they encode (Section 6.4.2), and how we compare brain representations to those generated by ML models (Section 6.4.3).

6.4.1 Brain representations and decoding

We provide a summary of how we process activation signals in the brain elicited by code comprehension, to probe whether they encode any specific code properties.

Dataset. We use the publicly available brain recordings released as part of the study by Ivanova et al. [2020] (MIT license). It contains brain recordings of 24 participants, each of whom gave consent and is not personally identifiable according to Ivanova et al. [2020] protocol requirements, reading 72 programs from a set of 108 unique Python programs. The 72 programs were presented in 12 blocks of 6 programs each. These programs were 3-10 lines in length and contained simple Python constructs, such as lists, *for* loops and *if* statements. A whole program was presented at once, and the task required participants to read the code and mentally compute the expected output, press a button when done, and select one of four choices presented to them which matched their calculated output.

From dataset to brain representations. The original dataset contains 3D images of the brain of each participant. Each voxel value in these images is an estimate of the response strength in this voxel when a particular code (or sentence) problem is presented. To determine which brain systems contain information about particular code properties, we focus our analyses to four systems – MD, Language, Visual, and Auditory (Section 6.3). A vector of voxels' activation values in each brain system is then taken to constitute that system's *representation* of a computer program and serves as an input to all our analyses.

Analyzing brain representations of code. We probe brain representations from each participant separately. We do not average data across participants since the

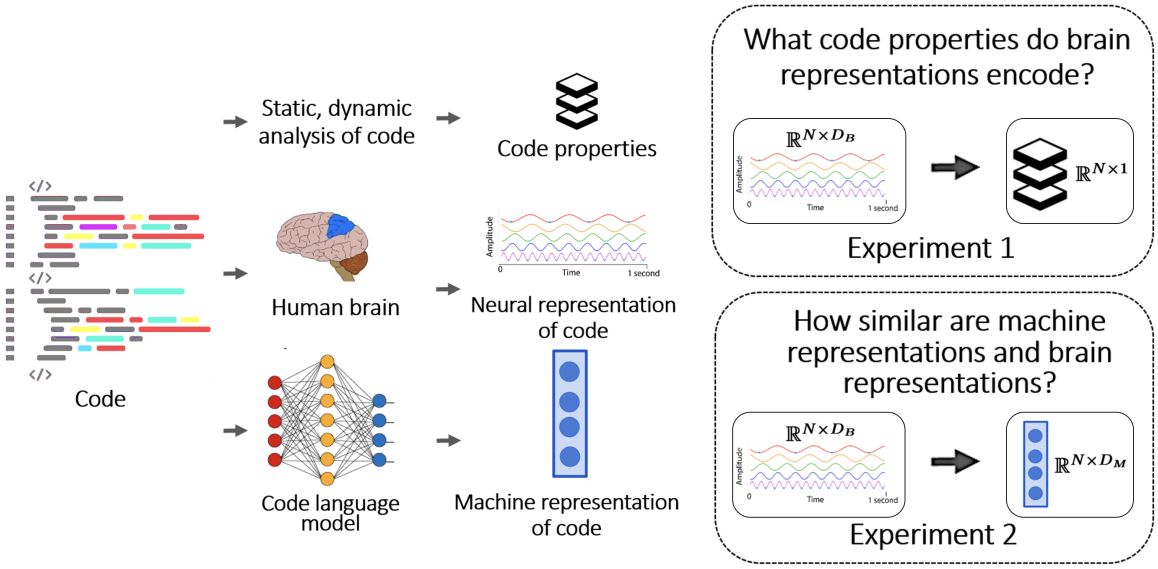


Figure 6-2: Overview. The goal of this work is to relate brain representations of code to (1) specific code properties and (2) representations of code produced by language models trained on code. In Experiment 1, we predict the different static and dynamic analysis metrics from the brain MRI recordings (each of dimension D_B) of 24 human subjects reading 72 unique Python programs (N) by training separate linear models for each subject and metric. In Experiment 2, we learn affine maps from brain representations to the corresponding representations generated by code language models (each of dimension D_M) on these 72 programs.

regions which respond to any task (comprehending code in our case) need not align anatomically. For each of two experiments—decoding different code properties, and mapping to code model representations—we train ridge regression/classification models which take as input normalized brain representations per participant. We hence learn 24 different regression models, for each code property or code model (one per participant), and then report the mean performance of these models across participants. This procedure is also referred to as multi-voxel pattern analysis (MVPA) [Norman et al., 2006]. Linear models are conventionally preferred for probes into brain representations since there has been evidence supporting the idea that other brain areas linearly map information from such brain representations [Kamitani and Tong, 2005, Kriegeskorte, 2011]. We choose a linear model primarily to control for over-fitting in light of the relatively small dataset.

A remark on data scarcity. For each participant, we train a linear regression/classification model with L2-regularization for on unique cross-validated leave-one-run-out folds of the 72 programs they attempted (or 48 programs when sentences are removed). On the order of 1000 voxels were selected from each brain region responding to any

given program per participant, thus resulting in a feature set of dimensions 72×1000. Such dimensions allow for strong statistical tests to support the robustness of the predictions made. As a baseline for the model predictions, we use the accuracy of a null permutation distribution generated from sampling 1000 random assignments of the labels.

6.4.2 Code properties

We attempt to decode the following code properties from brain representations. (a) **Code vs. sentences:** classify whether an input stimulus is a code or an English sentences describing a code (referred to as *sentences*). (b) **Variable language:** classify whether a program contains variable names written in English or Japanese (written in English characters). (c) **Control flow:** predict whether a program contains a loop (`for` loop), a branch (`if` condition), or has sequential instructions. (d) **Data type:** predict whether a program contains string or numeric operations. (e) **Static analysis:** predict static properties of a program like *token count*: number of tokens in the program, *node count*: number of AST nodes, *cyclomatic complexity*, and *Halstead difficulty*. The latter two metrics have been used by software engineering practitioners to quantify the complexity of code, and to quantify the difficulty a human would experience when comprehending code respectively. We defer predicting other advanced static analysis metrics such as tracking abstract interpretation joins, data flow analysis-related metrics, *etc.* to future work. (f) **Dynamic analysis:** predict information about a code's execution behavior like *runtime steps*: number of instructions executed in the program, and *bytecode ops*: number of bytecode operations executed in running the program.

Program length as a potential confound. Since the properties we examine can also potentially be differentiated using program length and other low-level code features, it is a potential confound in our experiments. We measured the inter-correlations of these properties, and their correlation to the number of tokens in the program (program length) We expectedly found the four *static analysis* properties to be highly correlated to each other and to *bytecode ops*. We hence use one representative metric

each from the two categories of properties for the rest of our analysis–*token count* for *static analysis*, and *runtime steps* for *dynamic analysis*. Importantly, the other properties we examine cannot be explained by program length alone, and therefore program length is not a confound in our experiments.

Mapping to code properties. The brain representations (Section 6.4.1) are mapped to each of the code properties by training a ridge regression (for *static analysis* and *dynamic analysis* properties; continuous values) or a classification model each for every participant-property pair. To evaluate model performance, we use classification accuracy when the predicted values are categorical (*e.g.* string vs. numeric data types), and the Pearson correlation coefficient when the predicted values are continuous (*e.g.* number of runtime steps). We choose Pearson correlation over RMSE, the canonical distance metric for continuous values, for its simplicity and interpretability. When testing for the significance of these predictions, we perform false discovery rate (FDR) correction for the number of brain systems tested and the number of properties tested.

6.4.3 Model representations and decoding.

We evaluate a bench of unsupervised language models, spanning from count-based language models to transformer neural networks [Vaswani et al., 2017]. These models were all trained on large (∼1M programs) Python datasets [Husain et al., 2019, Puri et al., 2021]. We use the output of the trained encoders (raw logits) in each of the neural network models as representations of the code input to the model. We vary the general complexity of these models to test whether that variation is meaningful in establishing the quality of brain to model fits. Model complexity here is the number of a model's learnable parameters. We evaluate the following models, ordered by their increasing model complexity: simple frequency-based language models—*bag-of-words*, *TF-IDF*; auto-encoder based unsupervised models—*seq2seq* [Sutskever et al., 2014], *CodeTransformer* [Zügner et al., 2021], *CodeBERT* [Feng et al., 2020], *CodeBERTa* [HuggingFace, 2020]; auto-regressive models with similar model complexity—*XLNet* [Yang et al., 2019], *CodeGPT* [Microsoft, 2021].

Baseline: Token projection model. We compare the results of the above models against an aggressive baseline (relative to the null-distribution labeling baseline), a *token projection model* provided by using unique Gaussian-distributed random vectors for the token embeddings in a vocabulary, and returning the sum of these token embeddings across a program. The resultant embedding is not transformed by any model or any weights–it instead serves as a proxy for the tokens that appear in the program. The results of this baseline model should be interpreted as the level of performance achievable from the presence of tokens alone with no semantic or syntactic information.

Mapping to code model representations. The brain representations (Section 6.4.1) are mapped to code representations by training another set of ridge regression models to learn an affine map, and a ranked accuracy metric is used to compare outputs. Ranked accuracy scores are commonly used in information retrieval where several elements in a range are similar to the correct one. In our case, the top-ranked prediction by the linear model indicates the closest fit (Euclidean distance) to the code model's representation. When reporting result significance, we perform false discovery rate (FDR) correction for the number of brain systems and the number of models. All experiments in this work were run on a single 8-core laptop in under an hour following setup.

6.5 Experiments & Results

Our experiments address two research questions:

- **Experiment 1.** How well do the different brain systems encode specific code properties? Do they encode the same properties?
- **Experiment 2.** Do brain systems encode additional code properties represented by computational language models of code?

6.5.1 Experiment 1 - How well do the different brain systems encode specific code properties? Do they encode the same properties?

We analyze the classification models and regressions trained on brain representations to predict each of the code properties described in Section 6.4.2. The results of our analyses are summarized in Figure 6-3. The classification and regression tests are marked on the x-axis of the left and right subplots respectively; the classification accuracy or Pearson correlation for each of the tasks is marked on the y-axes. We plot dynamic and static properties separately from the others because their baselines are different due to a difference in the similarity metric (classification accuracy vs. Pearson correlation). The baselines for the categorical code properties differ from each other due to variation in the number of target classes.

Auditory and Visual systems. The Auditory system and the Visual system serve as negative and baseline controls for the other systems. Since our code comprehension task is visual, we do not decode any meaningful information between programs from Auditory system, although we do observe decoding of *code vs sentences*, perhaps explained by previous work showing auditory cortex activation during silent sentence reading [Perrone-Bertolotti et al., 2012]. The Visual system serves as a baseline for low-level visual features of the code (the layout and indentation of the code, the presence of letters and alphabets in the programs, *etc.*). In Analysis 2, we show that the MD and the Language systems encode more than such visual features. The MD and the Language systems yield the following observations.

Analysis 1 - How accurately are different properties predicted by MD and LS? The MD and LS together decode all the properties well above chance barring variable language. The variable language finding is consistent with Ivanova et al. [2020], who show a lack of any significant difference in the aggregate neural activity between English and Japanese variables names—variable names seem to not be encoded any differently in the context of programs we study. To analyze the other results, we use paired two-sample t-tests ($p < 0.05$; FDR-corrected) and examine whether for a

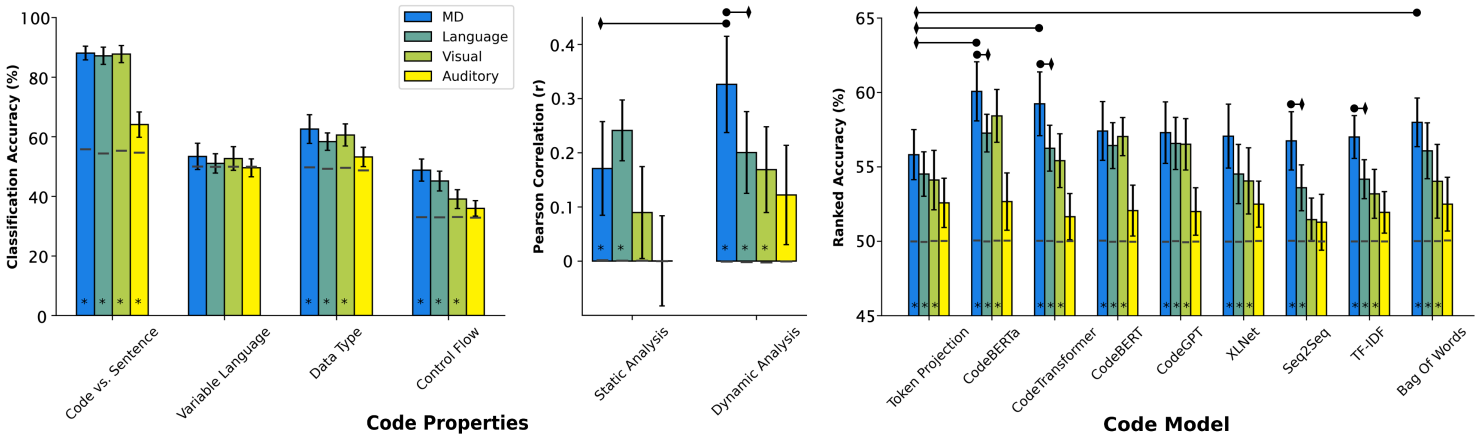


Figure 6-3: Affine models are learned on brain representations to predict each of the code properties described in Section 6.4.2, and a collection of code models described in 6.4.3, for each of the 24 participants. The mean decoding score across subjects is shown here, and error bars reflect the 95% confidence interval of the mean subject score. A solid line on each bar presents the empirical baseline for a null permutation distribution on shuffled labels. All decoding scores were compared to this permuted null distribution using a one-sample z-test, and the significance threshold was defined at $p < 0.001$; false-discovery-rate-corrected for the number of tests in each panel (FDR). Statistically significant results are denoted with a $*$, marked at the base of the bars. Additionally, $\bullet$-capped lines denote selected significant paired t-tests ($p < 0.05$; FDR).

given property, any one brain system decodes it significantly more effectively than another. We find that the MD system decodes the *dynamic analysis* property better than the Language system. We additionally test if any brain system has a preference for a specific code property over another. We find that the MD decodes the *dynamic analysis* property significantly better than the *static analysis* property. These findings establish the role of the MD system in encoding code-simulation and execution related information—an important of aspect of code comprehension.

Analysis 2 - Multi-system partial regression analysis. The decoding performance of the VS is comparable to that of the MD and LS (Figure 6-3). To assess the possibility that all three systems - MD, LS, and VS encode the same properties (all potentially related to low level program features), we employ a multi-system partial regression analysis. For each brain system, MD and LS (S_i), we train two models–one which decodes from VS, and another which decodes from S_i+VS. If the difference in the prediction accuracies between the two models is significant, we conclude that S_i encodes at least some information which is orthogonal to the information encoded by the Visual system. This method is similar to a variance-partitioning analysis

which is often employed in encoding models, *e.g.* [Deniz et al., 2019]. For all core properties, *control flow*, *data type*, *dynamic analysis*, and *static analysis*, we find the MD to encode information orthogonal to the VS. For *control flow* and *static analysis*, the LS also encodes information orthogonal to the VS. This suggests that low-level code properties are insufficient to explain the key results from Experiment 1. Other combinations in the regression model reveal that the MD encodes information orthogonal to the LS when predicting *code vs. sentences* and *dynamic analysis*.

Key learnings from Experiment 6.5.1. All core code properties are decodable from the representations of high-level brain systems, and this information is beyond that which can be explained from low-level visual information alone. Although no property is exclusively encoded in any one brain system, the MD system significantly encodes *dynamic analysis*-related properties–more than what the LS encodes, and more than *static analysis* properties. Similarly, we find evidence for the LS to also significantly encode *static analysis* and *control flow* related properties. These are new results on the nature of code properties different brain systems seem to process and encode. To explore properties which may not be specified by the set we investigate in this experiment, in the following section, we leverage code models as hypothesis-free proxy representations for code syntax and semantics, and see if any one system preferentially encodes a code model.

6.5.2 Experiment 2 - Do brain systems encode additional code properties encoded by computational language models of code?

We train ridge regression models with brain representations of programs from specific brain systems to predict code model representations of the same programs. The set of results from this experiment are summarized in Figure 6-3.

Auditory and Visual systems. Similar to Experiment 1, these systems perform as expected, with the Auditory system exhibiting the lowest decoding performance across code models, and the Visual system acting as a proxy for low-level information.

Analysis 1 - How well do brain representations in MD and LS map to code model representations? We find that the MD and LS map to all the models in our suite significantly more accurately than the null permutation baseline (Figure 6-3). Further, we find that the MD ranked accuracy is higher than LS for *CodeBERTa*, *CodeTransformer*, *seq2seq* and *TF-IDF* (differences evaluated using two-sample t-tests; $p < 0.05$; FDR-corrected.

Analysis 2 - The effect of model complexity on decoding to code models. We investigate the impact model complexity has on the performance of the mapping between brain and code representations.

We compare each of these code models against the *Token Projection* baseline model, which only encodes the presence of specific tokens, with no contextual or distributional information. We find that the MD system maps to all the models more accurately than the *Token Projection*, but this is not observed for the LS. In a set of paired two-tailed t-tests ($p < 0.05$; FDR-corrected), we find that the MD maps to three models: *CodeBERTa*, *CodeTransformer*, and *bag-of-words* significantly more accurately than to the *Token Projection* model. Curiously, since all but 3 of these mappings do not significantly surpass the model which can be explained using only token-level information, these data suggest that the brain signals we access primarily encode token-level information. To investigate this further, we analyze the correspondence between brain representations and code models in the context of the code properties they encode.

Analysis 3 - Code model and brain representations in the context of code properties. We first evaluate whether the different code properties we investigate in this work can be decoded from code model representations. We find that all the code properties are strongly encoded in all models. Since we have computed the mapping accuracies from code models $\rightarrow$ code properties (mentioned above), and from brain representations $\rightarrow$ code properties. we compute how well brain representation decoding results map to code model decoding results using Spearman rank correlation. We find two clusters in decoding performance across the code properties, each reflecting a distinct computational motif. We see perfect rank correspondence between the z-scores

of the decoding results for the MD and three transformer architectures: *CodeBERTa*, *CodeBERT*, and *CodeTransformer*, and between LS and two token-based models: *bag-of-words* and *Token Projection*.

Key learnings from Experiment 6.5.2. The MD decodes representations from four models of code significantly more accurately than LS, providing evidence that some aspects of code are more faithfully represented in the MD. A follow up partial regression analysis, as in 6.5.1, reveals that for most models, MD encodes information orthogonal to the LS, and each system encodes information above low-level VS.

Analyzing the correspondence between code models and brain representations based on the code properties they each encode reveals that the MD system maps preferentially to complex code models, encoding more than just token-level information. We discuss these results further in the following section.

6.6 Discussion

Through this study, we learn what computer program-related information can be decoded from the brain, and which brain systems primarily encode that information.

Brain representations and code properties. We show that the MD system preferentially encodes *dynamic analysis*-related properties when compared to other brain systems and other properties. Further, we find the Language system encodes syntax-related properties like *control flow* and *static analysis*. These findings are complementary to the results from Liu et al. [2020b] and Ivanova et al. [2020]– they show how the MD system is recruited in both mentally simulating code and comprehending code. They however do not find any consistent response in the Language system to either code simulation or comprehension. Our results show that despite not exhibiting significant average responses, these systems do encode code-specific properties, improving our understanding of these brain systems' functional organization.

Brain representations and code model representations. Another key contribution of our work, from Experiment 2, is demonstrating that it is possible to map

brain representations to representations learned by code models. In particular, we observe encoding of the properties represented by code models in the MD and LS, with 4 models more accurately mapped from MD. This is particularly noteworthy since these models, which are trained on source code symbols, can be mapped more faithfully from the representations of a network implicated in problem-solving than one associated with composition in languages.

We also considered model complexity as a relevant feature, and note that the MD and LS map to a combination of token projection embeddings almost as well as to complex models like *XLNet*. One plausible explanation for this surprising result is that the program stimuli are simple enough to allow the different properties evaluated in our work (*control flow*, *data type*, *etc.*) to be discerned from token level information alone (as validated in Experiment 1), which is likely why the *Token Projection* model also predicts these properties very well. Taken together, these data suggest that the information being decoded from brain activations in these two regions is driven at least by the information conveyed by tokens in the programs. This finding is notably consistent with work in the field of natural language which show swaths of cortex are predicted primarily by the token-level properties of sentence stimuli [Toneva et al., 2022].

While we see that information encoded in both the MD and the Language systems are driven by token-level information, a clearer trend emerges when evaluating brain-code model mapping in the context of specific code properties. We find that the MD shows decoding performance consistent with complex model architectures, suggesting that it may encode more than just token-related information, as complex code models capture contextual information as well. This is a new result, which along with Experiment 1, suggest distinct roles for the LS (purely token-level) and MD (more complex interactions) with respect to computer code comprehension and execution.

Way forward. Our findings have the potential to improve our understanding of the organization of the human brain, which can in turn lead to the design of better code models. In computer vision, results by Tschopp et al. [2018], Schrimpf et al. [2020] show how deep network architectures that mimic the visual system exhibit superior

image classification rates on image recognition tasks. Our findings prompt yet another reconsideration of the current design of ML models of code. Extant code model architectures do not explicitly model the Multiple Demand system in any way–they only model syntactic information and infer dependency information from program ASTs. Taking inspiration from the role of the Multiple Demand system we identified in this work, modeling dynamic runtime information as well as static code structure should be explored. See Srikant and O'Reilly [2021] for a related discussion.

We also provide initial results supporting the ability to decode specific and fundamental code-related primitives like control flow information. This sets us up to study other complex human cognitive processes which involve such primitives, but which are not naturally described in terms of code, like general problem solving, employing hierarchical and complex decision making strategies, *etc.* Such a study also promises to support the recent proposal of a child as a hacker [Rule et al., 2020].

Our work also promises to enhance *code prosthetics*–artificial interfaces that can help the physically challenged engage with programming environments. Such systems generally rely on designing and constructing brain decoders–models that convert brain activity to electrical impulses modulating external devices, which remains an open challenge. See the discussion in Nuyujukian et al. [2018] and Andersen et al. [2019] for details.

Limitations. The average program in any software project exhibits non-trivial control and data dependencies, object manipulation, function calls, types, and state changes. However, the programming tasks in Ivanova et al. [2020] are short snippets of procedural code with limited program properties. Responses to longer programs, with more complex properties and across multiple languages, should be studied on a larger number of participants in the future, in order to build on the trends provided by our work. Further, while there are multiple equally important aspects to programming like designing solutions, selecting appropriate data structures, and writing programs, here we have chosen to study only a single specific activity—code comprehension. Future work should explore these other aspects. Our results do not allow us to infer whether the MD and LS are driven by the same underlying features of code that are

used to discriminate between code properties and code models, so future work might consider an encoding analysis. Finally, as with all neural decoding analyses, extracting information from the mental states of participants should be done with caution to ensure it is not used for any exploitative purpose.

Chapter 7

Goal-optimized linguistic stimuli for psycholinguistics and cognitive neuroscience

Preface. This chapter, in full, is a re-print of **GOLI: Goal-Optimized Linguistic Stimuli for Psycholinguistics and Cognitive Neuroscience.** Srikant, S., Tuckute, G., Liu, S., and O'Reilly, U.M. Submitted to ACL 2023 [Srikant et al., 2023b]. The fMRI experiment (experiment 2) described in this work is described in full in [Tuckute et al., 2023].

Refer to Section 1.4 to read more about the motivation behind this work, and how it connects to the rest of the chapters presented in this thesis.

7.1 Introduction

Experiments in psycholinguistics and cognitive neuroscience of language aim to understand the representations and computations that support human comprehension and production abilities. In comprehension studies in particular, experiments record behavioral (eye-tracking and self-paced reading times) or neural outcomes (electroencephalogram, EEG, and functional magnetic resonance imaging, fMRI) while humans process carefully designed linguistic input [Lai et al., 2015, Shain et al., 2019b, Wehbe

	Specify goals?	Data-driven?	Automate?
Handcrafted	✓	✗	✗
Template-based	✓	✗	✓
Naturalistic corpora	✗	✓	✓
GOLI (this work)	✓✓	✓	✓

Table 7.1: GOLI automates generating stimuli which satisfy experimenter-supplied goals. It handles a broader set of goals than handcrafted and template-based methods while being data-driven.

et al., 2021, Heilbron et al., 2022]. Similar methods have been recently extended to probe how computational and language models process such linguistic input as well [Warstadt et al., 2019, Jeretic et al., 2020].

Linguistic stimuli (sentences) used in such experiments are typically hand-constructed [Martín-Loeches et al., 2012, Lai et al., 2015]. While handcrafting provides the experimenter with significant control over the goals of the constructed stimuli (*e.g.* the stimuli should adhere to grammatical rules such as subject-verb agreement or convey information about a particular topic), assembling a sizeable set of stimuli is time- and resource-intensive. Another important concern is the diversity of the resulting stimuli—they are generally limited by the experimenter's vocabulary and assumptions. Experimenters could easily be misguided by their top-down assumptions or an inaccurate formulation of the hypothesis being tested and use sets of words, sentence structures, or concepts that are biased in some way. Studies have shown how such biased stimuli have led to incorrect scientific conclusions [Chaves and Dery, 2018, Siegelman et al., 2019].

Another approach to constructing stimuli is to use templates [Warstadt et al., 2020]. An experimenter defines templates which structure and constrain stimuli. Stimuli are generated with a template by filling them with words sampled from different naturalistic vocabularies (*e.g.* of parts of speech). While this automates the process of creating stimuli and allows goal specification similar to handcrafting [Warstadt et al., 2019, Jeretic et al., 2020], the generated stimuli are still constrained to the experimenter's notions of a 'correct' template. Templates also often generate unnatural and incorrect sentences, which then need to be manually filtered out by the

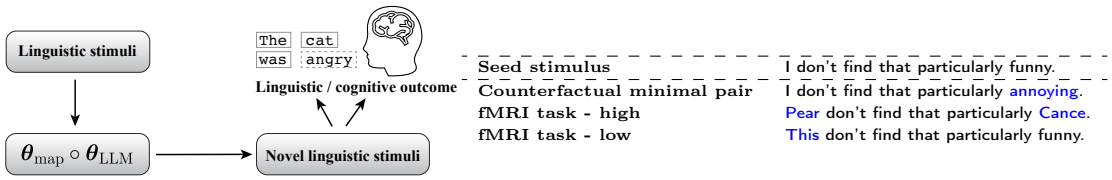


Figure 7-1: Overview. GOLI transforms a seed linguistic stimulus into a novel stimulus which either contains a desired linguistic property or elicits a desired cognitive outcome. It uses a language model ($\boldsymbol{\theta}_{\text{LLM}}$) to represent the seed sentence, a mapping model ($\boldsymbol{\theta}_{\text{map}}$) to map it to the desired property, and uses a gradient-based method to modify the seed sentence (propagates gradients through the composed model $\boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}$) into a novel one. The table (right) shows an example of stimuli generated from a seed stimulus for the three objectives we demonstrate in this work.

experimenter.

Yet another popular approach, which we refer to as the search-based method (SBM), involves randomly sampling from naturalistic text corpora [Kennedy et al., 2013, Nastase et al., 2021, Heilbron et al., 2022]. While this approach circumvents biases potentially introduced in handcrafted and template-guided stimuli, it has no efficient way to identify goals like targeted phenomena (*e.g.* sentiment polarity, surprisal, agreement, garden-path effects, licensing, gross syntactic expectation, center embedding, long-distance dependencies, and others mentioned in Marvin and Linzen [2018], Hu et al. [2020]) or infrequent phenomena [Bresnan and Kanerva, 1989, Losiewicz, 1992, Hoffmann, 2004, Ross, 2018, Turner, 2020] within the large corpora that need to be sampled to find suitable sentences. Prior work has manually edited such sampled texts and inserted linguistic properties of interest to create naturalistic stimuli [Futrell et al., 2020]. Further, sampling from corpora does not enable creating minimal pair stimuli–pairs of sentences that differ only in a very specific linguistic property. Minimal pairs are extensively used in psycholinguistic and language research to isolate causal attributes of behavior [Bemis and Pylkkänen, 2011, Kochari et al., 2018, Parrish and Pylkkänen, 2021]. Thus, minimal pairs, targeted and infrequent phenomena are mostly studied through stimuli that experimenters handcraft or create from templates.

It then seems that handcrafted and template-based stimuli offer significant control over the created stimuli, but may introduce undesirable experimenter-biases and are also time- and resource-expensive to create. On the other hand, automated sampling

from naturalistic corpora avoids experimenter-biases, but is not suited to test targeted or infrequent phenomena (Table 7.1). We propose a method that is data-driven, automated, efficient, and can fulfill a large set of experimenter goals which includes targeted and infrequent phenomena.

GOLI (Figure 7-1) is an automated approach to generate **g**oal-**o**ptimized **l**inguistic stimul**i**. GOLI starts from a seed sentence and modifies it until it satisfies experimenter-specified outcomes (linguistic or cognitive) by solving a gradient-based optimization formulation. Constraints on the generated stimuli can be easily enforced via the optimization formulation, providing the necessary control over stimuli that is typically offered by handcrafting and template-based approaches. In fact, we show that GOLI-generated stimuli can satisfy a broader set of goals than what handcrafting or template-use satisfies. GOLI is data-driven and is not bound to inductive biases of experimenters since it relies on data-driven computational models to transform the seed sentence and optimize it to achieve the desired goal.

We demonstrate GOLI on two deliberately different tasks: generation of minimal-pair counterfactuals and the generation of stimuli which predict specific responses in the human brain. These tasks differ in the nature of their outcomes, desired goals, and the constraints imposed on the generated sentences. Across these differences, we show that GOLI can successfully and easily model the various constraints posed by these tasks and efficiently generate novel stimuli, outperforming other methods currently used to prepare stimuli for such tasks. We will make all the code and data related to this work publicly available.

7.2 Problem description

In this section, we state the assumptions underlying GOLI. We introduce notation that we use in the rest of this work and then state the problem we solve.

GOLI assumes an experiment uses a set of linguistic stimuli to stimulate either a language property or a cognitive outcome. Further, it assumes the property or outcome is quantifiable, and a statistical model can predict its values corresponding

to an input linguistic stimulus. For example, if an experiment outcome measures logical inaccuracy (linguistic outcome) or reading times (cognitive outcome) in a sentence, then GOLI assumes a model which maps a random sentence stimulus $\mathcal{S}$ to a quantifiable measure of either the extent of inaccuracy (former) or the time taken to read a sentence (latter). We denote the mapping model as $\boldsymbol{\theta}_{\text{map}}$- the weights it is parameterized by, and the linguistic property or cognitive outcome as $y \in \mathbb{R}$. If the model does not initially exist, it can be learned as a preliminary step. We discuss in Section 7.5 the case where learning such a mapping model is not feasible.

Generally, $\boldsymbol{\theta}_{\text{map}}$ is trained to predict $y \in \mathbb{R}$ from representations $\boldsymbol{r} \in \mathbb{R}^d$ of the input stimulus $\mathcal{S}$. We use a large language model $\boldsymbol{\theta}_{\text{LLM}}$ without loss of generality, *i.e.* $\boldsymbol{r} = \boldsymbol{\theta}_{\text{LLM}}(\mathcal{S})$. Any alternate representation which is similarly differentiable can also be used, and this is yet another strength of GOLI.

$\mathcal{S} = \{\boldsymbol{x}_i\}_{i=1}^n$ denotes a sentence stimulus consisting of n tokens $\boldsymbol{x}_i \in \{0,1\}^{|V|}$, where $\boldsymbol{x}_i$ is a token from the set V, the vocabulary of permissible tokens which the LLM processes. We provide concrete examples describing the structure of $\boldsymbol{x}_i$ in Section 7.3. Following this notation, we have $y^{\text{pred}} = \boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}(\{\boldsymbol{x}_i\}_{i=1}^n)$, where $\circ$ denotes model composition, *i.e.* $\mathcal{S}$ is first input to $\boldsymbol{\theta}_{\text{LLM}}$, whose output is then input to $\boldsymbol{\theta}_{\text{map}}$. GOLI allows any number of and any kind of such models to be composed together.

We study the problem of generating a sentence $\mathcal{S}^{\text{gen}}$ by transforming $\mathcal{S}$ into $\mathcal{S}^{\text{gen}}$ in a way such that y^{pred}, the prediction of $\boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}$, is *close* to y^{desired}, an outcome specified by the experimenter.

7.3 Method

In this section, we motivate our method using an example. We then show how the problem of generating novel linguistic stimuli can be cast and solved as a problem in first-order (gradient-based) optimization. Consider the sentence:

`Running slow makes me very happy.`

which when input to the model $\mathcal{M} = \boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}$ predicts the sentiment $y^{\text{pred}} =$ positive ($\boldsymbol{\theta}_{\text{map}}$ is a binary sentiment classifier). Further, let the vocabulary V consist

of the tokens:

$$V = \left\{ \begin{array}{c} \texttt{Run}, \texttt{Sit}, \texttt{Stand}, \texttt{ing}, \texttt{ed}, \texttt{slow}, \texttt{happy}, \\ \\ \texttt{car}, \texttt{me}, \texttt{you}, \texttt{very}, \texttt{makes}, \texttt{well}, \cdot, ? \end{array} \right\} \tag{7.1}$$

The sentence has six space separated words with a terminating period symbol in it. Let's assume tokenizing this sentence generates the following eight tokens:

$$\mathcal{S} = \{\boldsymbol{x}_i\}_{i=1}^{8} = \{\text{Run}, \text{ing}, \text{slow}, \text{makes}, \text{me}, \text{very}, \text{happy}, \cdot\} \tag{7.2}$$

Further assume we desire a novel sentence whose prediction is $y^{\text{desired}} = \text{negative}$. The *generation* of sentences that we describe in this work involves modifying a subset of the eight tokens in $\mathcal{S}$ in a way that results in the model $\mathcal{M}$ predicting a value that is *closer to* y^{desired} *i.e.* is transformed to a negative sentiment sentence.

Two important questions need to be addressed to generate such sentences. First, which tokens or *sites* in the sentence should be modified? Of the n sites, if we are allowed to choose at most k sites, which set of $\leq k$ sites would best guide $\mathcal{M}$ to the desired prediction. We call this the **site selection problem**. Second, how should a token at a given site be modified, and what should the modified token be? We call this the **site perturbation problem**.

Site selection. The benefit of isolating site selection as a distinct sub-problem is it supports complex formulations, such as constraining and optimizing specific sites. For instance, site selection and site perturbation can be jointly optimized: an optimal site can depend on the optimal token found by the site perturbation sub-problem and vice versa.

We employ a simple site selection strategy in this work. To select k specific sites from the available n sites, we follow the gradient-based word importance method from Wallace et al. [2019]. The method first sorts the tokens $\boldsymbol{x}_i$ in decreasing order of the magnitude of the gradient on the output y with respect to $\boldsymbol{x}_i$. The top k magnitude tokens are selected as the sites to perturb, since they impact the output the most.

Site perturbation. At a given site, there are three operations which would modify the token: *replace* an existing token with another token, *insert* another token at the site–either before or after the token present at the site, or *retain* the token at the

site unaltered (this is equivalent to not selecting a site to carry out a modification operation).

We discuss only replace modifications, since deletion and insertion reduce to replace modifications. Deletion is replacing with an empty token, and an insertion is a replace modification applied to a dummy token inserted at a site.

A replacement token modification strategy requires a replacement token $\boldsymbol{u}_i \in \{0,1\}^{|V|}$ to be identified from the set of tokens V. For example, if the selected site for replacement in (7.2) is 3: `slow`, then a possible sentence could result from replacing `slow` with `car` (token 8 sampled from the vocabulary V in (7.1)), resulting in the previously unseen sentence: `Running car makes me very happy.` Increasing the number of sites to be perturbed results in a sentence that is very different from the original sentence. Similarly, inserting new tokens can introduce new words and phrases.

Selecting an appropriate token from a vocabulary is a combinatorially expensive problem: it takes $O(|V|^k)$ time to select tokens at k sites from the vocabulary V, since each site offers $|V|$ possible tokens to choose from. The aim is to thus tractably select a replacement token $\boldsymbol{u}_i$ at a site i such that the predicted activation y^{pred} matches y^{desired}. We set this up as a combinatorial optimization problem and solve for $\boldsymbol{u}_i$.

7.3.1 Solution formulation

Based on the site selection perturbation formulation, we formally define the described replacement operation.

For a sentence $\mathcal{S} = \{\boldsymbol{x}_i\}_{i=1}^{n}$and a set K of k site indices to perturb, wherein each site index j satisfies $(1 \leq j \leq n)$, we formalize site perturbation in the following way: we introduce a one-hot vector $\mathbf{u}_i \in \{0,1\}^{|V|}$ to encode the selection of a token from V which would serve as the replaced token for at a chosen site.

If the j^{th} entry $[\mathbf{u}_i]_j = 1$ and $i \in K$, then the j^{th} token in V is used as the modified token which will replace $\boldsymbol{x}_i$ at the site i. We also impose the constraint $\mathbf{1}^T\mathbf{u}_i = 1$, implying that only one perturbation is performed at $\boldsymbol{x}_i$.

Let vector $\mathbf{u} \in \{0,1\}^{k\times|V|}$ denote k different $\mathbf{u}_i$ vectors, one for each token $i \in K$,

where $|K| = k$. We then define a newly generated or transformed sentence $\mathcal{S}^{\text{gen}}$ as comprising tokens $\{\boldsymbol{x}_i^{\text{gen}}\}_{i=1}^n$, where each $\boldsymbol{x}_i^{\text{gen}}$ is defined as:

$$\boldsymbol{x}_i^{\text{gen}} = \begin{cases} \boldsymbol{u}_i, & \forall i \in K, \text{ where } \mathbf{1}^T \mathbf{u}_i = 1, \ \mathbf{u}_i \in \{0,1\}^{|V|} \\ \boldsymbol{x}_i, & \forall i \notin K \end{cases} \tag{7.3}$$

We solve the following objective to obtain $\boldsymbol{u}$:

$$\begin{aligned} \underset{\boldsymbol{u}}{\text{minimize}} \quad & \ell(\boldsymbol{u}; \boldsymbol{x}, \boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}) \\ \text{subject to} \quad & \text{constraints in (7.3)} \end{aligned} \tag{7.4}$$

where ℓ denotes an appropriate loss function which encodes the desired cognitive outcome. Algorithm 2 in Appendix 2 describes how GOLI solves this optimization problem.

Algorithm

Algorithm 2: GOLI: A gradient-based sentence transformation method

1: **Input:** Random $\mathcal{S} = \{\boldsymbol{x}_i\}_{i=1}^n$, model $\mathcal{M} = \boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}$; Learning rate α; Loss function ℓ; Perturbation iterations N; Number of sites to perturb k
2: ▷ **Site selection**
3: $\mathcal{T}$ = OrderByImportance($\{\boldsymbol{x}_i\}_{i=1}^n$)
4: ▷ From Wallace et al. [2019]; see Section 7.3
5:
6: ▷ **Site perturbation**
7: $\boldsymbol{u} = \boldsymbol{x}$
8: **for** j in N **do**
9: **for** $\boldsymbol{x}_i$ in $\mathcal{T}$ **do**
10: **if** $k > 0$ **then**
11: $\boldsymbol{u}_i^{\text{soft}}$ = softmax($\boldsymbol{u}_i$)
12: $\boldsymbol{u}_i$= Multinomial($\boldsymbol{u}_i^{\text{soft}}$)
13: $k = k - 1$
14: y^{pred}= $\mathcal{M}(\boldsymbol{u})$ ▷ Forward pass
15: $\nabla = \frac{\partial}{\partial \boldsymbol{u}} \ell(y^{\text{pred}})$ ▷ Backward pass
16: $\boldsymbol{u} = \boldsymbol{u} - \alpha.\nabla$
17: $\mathcal{S}^{\text{gen}}$= $\boldsymbol{u}$
18: **return** $\mathcal{S}^{\text{gen}}$

Summary. A backward pass (line 17) allows gradients with respect to the input $\boldsymbol{u}$ to be propagated from the loss function ℓ. The input is then modified in the direction of

these gradients (line 18), with the modified input passed to $\mathcal{M}$ (line 11-16).

To solve Eq (7.4) effectively, we relax $\mathbf{u}_i \in \{0,1\}^{|V|}$ to $\mathbf{u}_i \in [0,1]^{|V|}$. This continuous relaxation of binary variables is a commonly used trick in combinatorial optimization to boost the stability of learning procedures in practice [Boyd et al., 2004].

See Jang et al. [2017], Maddison et al. [2017] for details on how the softmax (line 11, Algorithm 2) aids in reparametrization of the `argmax` functionality in the categorical case. This is theoretically equivalent to the Gumbel softmax trick.

Once the continuous optimization problem Eq (7.4) is solved, a hard thresholding operation or a randomized sampling method can be used to map a continuous solution to its discrete domain. For the randomized sampling method, we consider $\mathbf{u}$ as probability vectors with elements drawn from a Multinomial distribution. A Multinomial distribution models selecting one of the $|V|$ classes when selecting a token from the vocabulary. We use the randomized sampling method in our experiments and follow the setup described in Algorithm 1 in Xu et al. [2019]. See also Xu et al. [2019] for a proof of convergence of the randomized sampling method.

When incorporating additional constraints, such as capitalizing the first word or ensuring the last word is a punctuation, we sample from a subset of u_i indices. Originally, $|u_i| = |V|$. To sample from a subset of the vocabulary, say capitalized letters, we identify the set of indices C in the vocabulary corresponding to capitalized letters. When sampling from u_i, we mask out all those indices not in C, and sample only from those present in C.

Incorporating additional constraints. A variety of constraints on $\mathcal{S}^{\text{gen}}$ can be imposed by using appropriate loss functions and vocabulary subsets to find candidate replacement tokens from. Section 7.4.1 discusses how a loss function can be modified to generate stimuli that are grammatically likely. Similarly, other site-specific constraints like capitalizing the first word of a sentence or the last token being a punctuation can be ensured by assigning different subsets of naturalistic vocabularies when solving the site perturbation problem.

7.4 Experiments & Results

We demonstrate and assess GOLI on two tasks—constructing minimal pairs of counterfactual sentences for sentiment analysis, and an fMRI-based targeted brain response task. These two tasks differ in the questions they ask, the architectures used to encode sentences (BERT vs. GPT2-XL), the outcome of $\boldsymbol{\theta}_{\text{map}}$ (sentiment-class classification vs. brain region response predictions), the set of constraints imposed on the generated sentences, and consequently the loss functions needed to generate sentences. We describe these details below.

7.4.1 Counterfactual minimal-pair task

Training-data augmentation with *counterfactuals* (CFs) has been proposed as a way to mitigate out-of-domain generalization of NLP models [Levesque et al., 2012, Kaushik et al., 2020]. Rooted in causal learning, a CF in the context of NLP models is designed to study the change in an NLP model's prediction following an intervention to its input text, generally implemented as minimal edits to the text. Such minimal changes to different input features help ascertain the causal role of these features in a model's prediction. Producing such CF stimuli though can be challenging, and resembles the process of developing minimal-pair stimuli in psycholinguistics experiments discussed in Section 7.1, Introduction.

Recent work however has explored automated generation algorithms for such CFs [Wang and Culotta, 2021, Yang et al., 2021, Howard et al., 2022]. Notably, Howard et al. [2022], the state-of-the-art, propose a system to generate CFs for sentiment analysis on the SST-2 IMDB movie reviews dataset [Socher et al., 2013]. The CF reviews they generate have the opposite sentiment as the original stimulus, while being *natural* in a way that would resemble CFs generated by human experts [Kaushik et al., 2020, Gardner et al., 2020]. We demonstrate how GOLI can be setup for this task by appropriately customizing the loss function and constraints in the formulation in Eq. (7.4).

Objective. We use a BERT-based sentiment classifier fine-tuned on the SST-2 task

(binary classification) as our mapping model, $\boldsymbol{\theta}_{\text{map}}$. In this case, BERT serves as $\boldsymbol{\theta}_{\text{LLM}}$. Our objective then is to generate modifications to a given sentiment review such that $\boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}$ flips its prediction on the modified sentence and the modified sentence is *close* to the original sentence.

Loss, Constraints. We use the standard binary cross entropy (Loss_{BCE}) as our loss function as it allows us to specify the desired class we want $\boldsymbol{\theta}_{\text{map}}$ to predict in the binary sentiment classification task:

$$\ell(\boldsymbol{u}) = \text{BCE}(\boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}(\boldsymbol{u}), y^{\text{desired}}) \tag{7.5}$$

where y^{desired} is 0 for the negative sentiment class and 1 for positive. An alternate loss function which we do not try and defer to future work is ensuring general fluency and grammaticality of the generated sentences [Goswamy et al., 2020] by introducing two additional loss terms:

$$\ell(\boldsymbol{u}) = \text{Loss}_{\text{BCE}} + \text{Loss}_{\text{BOW}} + \text{KL}(H(\boldsymbol{u}), H(\boldsymbol{x})) \tag{7.6}$$

where $\text{Loss}_{\text{BOW}} = -\log(\sum(p_i u_i))$ penalizes selecting a u_i whose bag-of-words probability p_i is low or unlikely, and $\text{KL}(\cdot)$ is the KL-divergence between the intermediate decoder representation $H(\cdot)$ of the modified input $\boldsymbol{u}$ and the unmodified, original input stimuli $\boldsymbol{x}$. The KL-term ensures the distribution of each generated token u_i is similar to the original token x_i. To ensure minimal pairs, we select a maximum of two sites to be modified in each original sentence.

Evaluation. We evaluate our generated stimuli against the CF-generation method introduced in Howard et al. [2022]. They work with a subset of the IMDB dataset (training set, N=8173). For each sentence in the training set, they generate a CF using the following complex setup: first, they provide a prompt (a part of the original stimulus) and a desired sentiment (positive or negative) to a pre-trained adaptation of the GPT-2 model (first proposed by Gururangan et al. [2020]). The model is optimized to generate reviews which complete the prompt and are of the desired sentiment polarity. Second, they use a constrained decoding algorithm (first proposed

	Test-set 1	Test-set 2	Test-set 3
GOLI	$\mathbf{93.37_{0.01}}$	$\mathbf{94.94_{0.53}}$	$\mathbf{92.14_{0.05}}$
NeuroCF-1g	$92.75_{0.03}$	$93.10_{0.06}$	$89.27_{0.04}$
NeuroCF-np	$93.10_{0.05}$	$\mathbf{94.74_{0.08}}$	$\mathbf{91.18_{1.17}}$
Expert-crafted	$92.63_{0.48}$	$97.34_{0.37}$	$95.22_{0.45}$

	MoverScore	Perplexity
GOLI	0.45	39.2
NeuroCF-1g	0.46	14.1
NeuroCF-np	**0.20**	**12.7**
Expert-crafted	0.70	19.3

Table 7.2

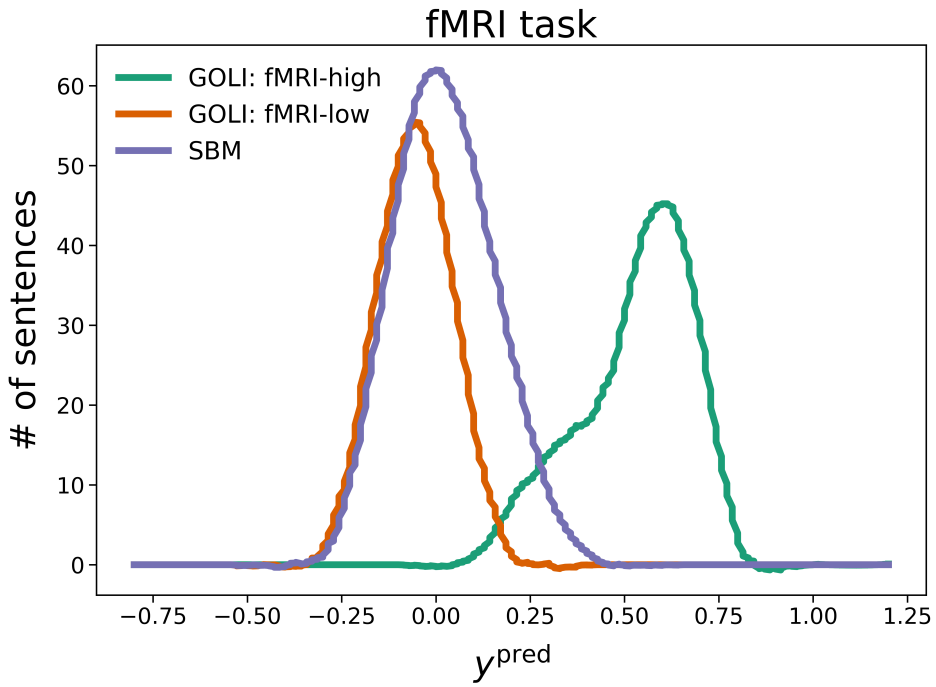


Figure 7-2

Results. Table 7.2: GOLI-generated counterfactual sentences vs. NeuroCF Howard et al. [2022] vs. expert-crafted CFs (serves as an upper bound; Kaushik et al. [2020]), augmented with training data to improve the robustness of a RoBERTa-based sentiment analysis classifier. *Top.* Accuracy (percent) on three unseen test sets. Std. dev. across 10 runs mentioned as subscripts. *Bottom.* MoverScore and Perplexity, two quality measures of the generated sentences. **Figure 7-2**: A histogram of y^{pred} from sentences sampled using a search-based method (SBM), and those generated by GOLI optimized on two objectives: fMRI-high and fMRI-low.

by Lu et al. [2021b]) with the adapted GPT-2 model to ensure the generated sentence remains a minimal pair to the given input sentence. They augment the generated CFs with the training set and fine-tune a RoBERTa-based classifier. They augment with CFs generated from two settings–**NeuroCF-1g**: where they provide just the first word in the original stimulus as a prompt to their system, and **NeuroCF-np**: which selects a subset of the original sentence as a prompt. They evaluate the fine-tuned model on three out-of-distribution test sets - **Test-set 1**: another subset from the IMDB dataset (N=2245), **Test-set 2**: a dataset from Kaushik et al. [2020] (N=488), **Test-set 3**: a dataset from Gardner et al. [2020] (N=488). Accuracy on the test set and descriptive metrics of the generated sentences (described below) serve as merit indicators for the effectiveness of the generated CFs.

To evaluate GOLI, we generate minimal-pairs for each sentence in their training set, and fine-tune and evaluate their RoBERTa model using an augmented dataset containing GOLI-generated sentences. We compare the accuracy of the RoBERTa model against the CFs generated by NeuroCF-1g and NeuroCF-np. Expert-crafted CFs generated by Kaushik et al. [2020] serves as an upper bound for performance in our evaluation.

Results. Table 7.2, *Top* shows the accuracy of the CF-augmented RoBERTa models on the three test sets, averaged over 10 random runs. Across the test sets, we see that GOLI consistently outperforms NeuroCF-1g, and is comparable in its performance to NeuroCF-np. We highlight that GOLI uses a very generic formulation for transforming an input sentence to meet a desired goal, and despite the generality, is capable of matching the performance of a bespoke solution like NeuroCF.

Further, to evaluate the quality of the generated sentences, Howard et al. [2022] use two metrics (Table 7.2, *Bottom*): MoverScore [Zhao et al., 2019] and perplexity. A MoverScore is computed between the generated counterfactual and the original sentence. A low score suggests the two sentences are similar. We see that GOLI generates sentences of similar MoverScores to NeuroCF-1g and comparable to NeuroCF-np. However, the average counterfactuals created by experts seem to be fairly farther off from their respective original sentences, suggesting room for automated stimuli

generation methods to improve.

Perplexity is a measure of how likely a given sentence is, which we evaluate on GPT-J, a domain-agnostic model. A lower score suggests a higher likelihood of the sentence. Table 7.2 shows that GOLI produces sentences with comparatively higher perplexity. This was expected since we do not incorporate felicity-related loss terms as described in Eq. (7.6). As seen in previous works *e.g.* Goswamy et al. [2020], a modified loss incorporating felicity should improve perplexity, making it comparable to NeuroCF. We defer this verification to future work.

7.4.2 fMRI task

A characterization of the sentences that activate the language network in the human brain [Fedorenko et al., 2010b] remains an open question. The fMRI task is to thus generate sentences that predict a desired brain response in the language network. To do so, we set up an fMRI experiment where we first collect brain responses of participants reading random sentences. We then fit a linear model $\boldsymbol{\theta}_{\text{map}}$ to predict these brain responses from LLM representations of the sentences. Given a trained $\boldsymbol{\theta}_{\text{map}}$, we use GOLI to generate novel sentences using a separate dataset of seed sentences.

Objectives. GOLI is provided two separate objectives: to generate sentences that predict high responses in the language network (fMRI-high; $y^{\text{pred}} \geq +0.4$) and to generate another set of sentences which predict low brain responses (fMRI-low; $y^{\text{pred}} \leq -0.3$).

Setup. We invited participants (N=5) to passively read a set of 1000 diverse, corpus-extracted 6-word sentences in an event-related design (referred henceforth as training set) We pre-process and select responses from language-selective areas of the brain. Voxels (3D pixels) from these areas were averaged within and across each participant to yield a scalar language network response value associated with each sentence stimulus. The range of brain responses values predicted by $\boldsymbol{\theta}_{\text{map}}$ on the training set across participants was $[-0.47, +0.54]$. These values represent z-scores of brain responses and hence are both positive and negative–they represent relative magnitudes of brain responses. Based on the training set, we interpret negative values ≤ -0.3 as a low

response and $\geq +0.4$ as high. A linear model $\boldsymbol{\theta}_{\text{map}} \in \mathbb{R}^d$ was learned to predict these average brain responses across participants from GPT2-XL representations $r \in \mathbb{R}^d$ of sentences ($d = 1600$).

Loss, Constraints. We model the fMRI-high and fMRI-low objectives with a squared-loss function:

$$\ell(\boldsymbol{u}) = (y^{\text{desired}} - \boldsymbol{\theta}_{\text{map}} \circ \boldsymbol{\theta}_{\text{LLM}}(\boldsymbol{u}))^2 \tag{7.7}$$

To generate sentences with high positive and high negative desired predicted responses, we set y^{desired} to $+1.2$ and -0.8 respectively, values slightly beyond the maximum and minimum predicted values seen on the training set.

The number of words in each sentence in $\mathcal{S}^{\text{gen}}$ was constrained to contain six space separated words, terminated by a punctuation, with the first word capitalized. These constraints ensure avoiding confounding effects of sentence length and unusual orthography (*e.g.* lack of capitalization, no end-of-sentence punctuation) in fMRI recordings.

Search-based method (SBM). We compare GOLI to a search-based approach which is routinely used to assemble language stimuli for such a task: exhaustively searching a large, unseen naturalistic corpus of text. Each sentence from such a corpus is individually tested against the desired goal. Unlike GOLI, the search-based method does not modify any sentences in the set of sentences it searches through–it just filters and selects those that achieve the desired goal. A key drawback of this method is that a prohibitively large corpus may then need to be sampled from should a small proportion of natural sentences meet the desired goals (fMRI-high, fMRI-low objectives for this task). Further, as discussed in Section 7.1, natural sentences may not necessarily meet the desired goals for this task—the brain may well be responsive to a very particular subset of sentences and sentence structure patterns. SBM over a naturalistic corpus then threatens the discovery of such patterns.

Evaluation criteria. We select 1500 sentences extracted from various, diverse text corpora (referred henceforth as test set) to evaluate SBM and GOLI. We demonstrate

the utility of GOLI over SBM along two dimensions: **sample efficiency:** the number of sentences needed in a corpus which when sampled results in the desired number of linguistic stimuli which satisfy the desired goal, and **solution diversity:** whether the sentences generated by GOLI achieves (or outperforms) the desired goal in both quality and quantity.

Results. Figures 7-2 summarizes our results. We plot the distribution of y^{pred}– predictions made by $\boldsymbol{\theta}_{\text{map}}$—on processing sentences produced by GOLI and by SBM on the test set (N=1500). We see that while most randomly sampled sentences (marked in blue, SBM) in the test set elicit average brain responses (around the z-score 0 of y^{pred}), 0.2% ($\frac{3}{1500}$) sentences elicit high brain responses ($y^{\text{pred}} \geq 0.4$). In sharp contrast, we find that GOLI, when optimized for fMRI-high (green curve in Figure 7-2), generates 80% ($\frac{838}{1049}$) high-response prediction sentences. We work with 1049 GOLI-generated sentences because 451 (1500–1049) of those failed the automated filters.

Comparing the two methods on the fMRI-low objective, we see 0.2% ($\frac{4}{1500}$) sentences in SBM elicit low brain responses ($y^{\text{pred}} \leq 0.3$). GOLI sentences optimized for fMRI-low (red curve, Figure 7-2) yield an interesting observation: despite the sentences being optimized to minimize their predictions, we find that, unlike in the fMRI-high objective, GOLI is unable to generate sentences that predict values significantly lower than those found on the test set. GOLI generates 0.8% ($\frac{8}{990}$) low-response sentences, although the overall average y^{pred} drops to -0.05 in fMRI-low, from $+0.02$ in the SBM setting.

These results suggest that in order to assemble a total of 500 high or low activity sentences (a reasonable estimate of the number of unique sentences needed in an fMRI experiment), one would have to significantly increase the number of sentences to sample from when using SBM, which increases the compute and data-needs to run such experiments. For the fMRI-high objective especially, we see that the number of sentences required to sample from may be significantly higher than 20x since we never see sentences greater than 0.40 on the training set, while GOLI reveals that perhaps high-response sentences are those that predict ≥ 0.65.

Further, we find that GOLI generates fairly unusual sentence structures for the fMRI-high objective (Fig 7-1), while generating more 'regular-looking' sentences for the fMRI-low objective. This is an interesting result which would not have been discovered had we sampled from regular text corpora via SBM. Collecting brain data for the GOLI-generated sentences, and analyzing the implication of these *unusual* fMRI-high sentences on the neuroscience of language-responsive brain regions is left for future work. We discuss this more in Section 7.5.

7.5 Discussion

We demonstrate the effectiveness of GOLI in generating stimuli in two distinct experiment settings. The minimal pairs task demonstrates how easily GOLI can be employed to generate a tightly constrained set of stimuli. It is infeasible to generate such stimuli pairs using either templates or by looking in naturalistic corpora.

In the fMRI task, GOLI helped generate stimuli that are predicted to elicit high or low responses in the language network in the human brain. Knowing which stimuli elicit maximal activity in neurons can provide useful insight into the representations and computations that brain areas perform [Hubel and Wiesel, 2009, Bashivan et al., 2019, Xiao and Kreiman, 2020]. The task—of predicting specific brain responses—is unique, and we demonstrate how such goals can successfully be encoded in GOLI, which even handcrafting does not support. We highlight the innovative use of $\boldsymbol{\theta}_{\text{map}}$ as a surrogate model to quantify and predict the goal, which GOLI then uses to guide stimuli generation. It is possible that the *unusual* fMRI-high sentences generated by GOLI are high on surprisal, since only a few words are abruptly modified in $\mathcal{S}^{\text{gen}}$. This hypothesis can be confirmed in a follow-up fMRI study where participants' brain responses to GOLI-generated sentences are compared to their responses to other meaningful sentences with one or two words randomly swapped out. We will investigate this in future work.

GOLI in other domains. GOLI can potentially be used to generate inputs in domains beyond language, such as tasks in memory [Barr et al., 2016], motor-control

[Srivastava et al., 2022], planning in robotics [Aznan et al., 2019], and AI [Chollet, 2019] or in engineering such as circuit design [Liu et al., 2018], processor design [Ritter and Hack, 2020] and electric machine design [Wang et al., 2017]. In each of these works, the authors attempt to generate hand-crafted stimuli or inputs in a discrete domain (similar to linguistic stimuli) that are required to satisfy a suite of constraints their respective problem domains pose.

Chapter 8

Modeling the presence of *beacons* in program comprehension

8.1 Introduction

The software engineering (SE) community has long tried to establish what makes a *program*—a one-page length computer program in the context of this work—easy to understand. This question has been addressed from three broad perspectives: empirical [Buse and Weimer, 2008, Zimmermann et al., 2010, Srikant and Aggarwal, 2014b, Scalabrino et al., 2017, Trockman et al., 2018], behavioral [Soloway and Ehrlich, 1984, Wiedenbeck, 1986, Letovsky, 1987, Casalnuovo et al., 2020b,a] and theoretical [Parnas, 1972, 1979, Koppel and Jackson, 2020].

Among them, a few behavioral studies have addressed finding important parts of a program during its comprehension. Soloway and Ehrlich [1984], Wiedenbeck [1986], Letovsky [1987] conjectured the presence of *beacons* and *schemas*. *Beacons* and *schemas* refer to 'important' substrings or patterns which convey the key ideas of a bug-free program's functionality. These works show the absence of such beacons or schemas makes it harder for programmers to comprehend a program. Wiedenbeck [1986], in particular, *defines* beacons as those snippets which experts attend to but novices do not, as novices are unable to appreciate the importance of such snippets. She bases this on her experiment in which she observes expert programmers to recall

from memory parts of codes which are very different from what novice programmers recall. Similarly, Soloway and Ehrlich [1984] and Letovsky [1987] propose the presence of *schemas* which aid code comprehension. Through controlled behavioral tests, they show how comprehension diminishes among expert programmers when programs diverge from their most expected schema.

While these works make an important intellectual contribution, they provide only weak empirical evidence for the presence of beacons. These studies were conducted on very few program samples and the samples were chosen in a way that the presence of beacons was apparent.

In the context of these prior works, the central questions we investigate are:

> Do humans consistently identify *beacons* in any program? What are the predictors of these *beacons*?

For possible predictors of beacons, we look to recent literature on text comprehension. Hahn and Keller [2018], Malmaud et al. [2020], Schrimpf et al. [2021b] recently showed that language model representations of sentences exhibited a close correspondence to behavioral responses to understanding those sentences (reading times, eye-gaze information). Further, Srikant et al. [2022] recently provided initial evidence for a correspondence between code model representations and the representations of programs encoded in the brains.

Building on these results, we explore whether representations of code learned by code models [Allal et al., 2023, Chowdhery et al., 2022, Brown et al., 2020] can predict the presence of beacons. In the context of our work, we use code models as proxies for expert programmers since they have been trained on extensive code corpora, and evaluate whether these models encode the presence of beacons.

Predicting the presence of such beacons can inform how our minds seek information in any program. Knowledge about the factors affecting our information-seeking can have important consequences—these factors can potentially be used to reduce the *mental load* [Crichton et al., 2021] of reading and understanding programs. It is possible that an increase in the number of beacons in a program increases confusion,

as it increases the number of things to attend to. Consequently, these factors can be used as objectives by generative models of code [Allal et al., 2023, Chowdhery et al., 2022, Brown et al., 2020] when generating and synthesizing programs—for example, they can be constrained to generate programs containing just one beacon in it.

In addition to using code model representations as predictors, we also investigate the surprisal of a token. Studies in language comprehension have shown the surprisal of a word to be a strong predictor of comprehension difficulty [Bicknell and Levy, 2010]. Beacons may correlate strongly to those parts of the program which are difficult to comprehend.

Section 8.2 provides details on the experiment setup and Section 8.3 discusses the results from our experiments. Related work is discussed in Section 8.4.

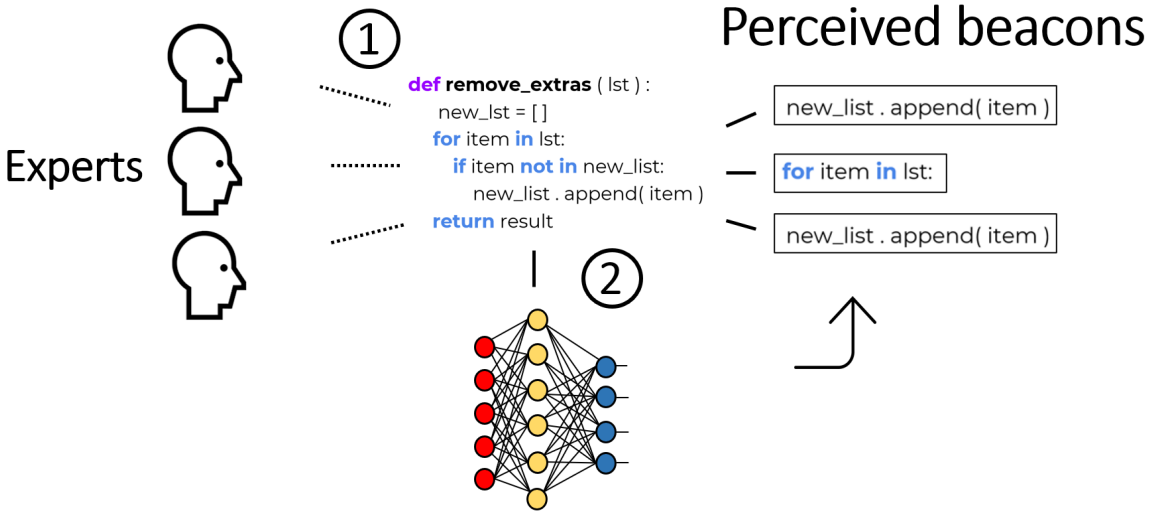


Figure 8-1: Overview of experiment setup. 1. Programs are first shown to experts. Each expert marks out the beacons they perceive. We define the token response rate (TRR) for each token in a program as the ratio of the number of raters who rated the token as a potential beacon to the total number of raters. 2. A code model is then provided the same program. The code model representations for each token is correlated with the TRR for that token.

8.2 Experiment Setup

We describe the dataset we use to collect responses from experts, the code model we use to predict beacons, and the behavioral experiment in which experts mark out

beacons in programs. Figure 3-1 provides an overview of the experiment setup.

Dataset. We sample a total of 10 programs at random from two sources: Kanade et al. [2020a] and Lin et al. [2017]. The dataset by Kanade et al. [2020a] contains de-duplicated programs sourced from software projects that have been on Github. Each program in the sampled set contains between 14 to 34 tokens (they can be described in 4-20 lines of Python code. They can be viewed at `https://github.com/ALFA-group/beacons-in-code-comprehension/blob/main/data/codes.xlsx`).

Code representation. We experimented with code models available on Hugging face, and selected Santacoder [Allal et al., 2023]. Santacoder is a 1.1B parameter model trained on the Python, Java, and JavaScript subset of The Stack (v1.1) [Kocetkov et al., 2022], a 6.4 TB dataset of permissively licensed source code in 358 programming languages. The main model uses Multi Query Attention [Shazeer, 2019], was trained using near-deduplication and comment-to-code ratio as filtering criteria and using the Fill-in-the-Middle objective (a variant of the cloze task). In our experiments, we use the activations of the model's final layer as representations for each token in a program. The dimensions of the representations $\in \mathbb{R}^{2048}$. Future work should evaluate the sensitivity of the encoded information in other layers of the model.

Behavioral responses. We enlisted programmers to read and understand programs and mark tokens in programs they deemed beacons. The aim was to have multiple programmers evaluate every program in our dataset and for each programmer, record those tokens they considered to be beacons. If beacons exist, we should observe a consensus among the marked tokens.

Programmers who were recruited for the study (henceforth referred to as experts $\mathtt{E}_i$) had at least four years of experience in Python. Thirteen programmers were selected, out of which ten programmers completed the surveys. The results in this study are based on the responses from the ten experts.

We used Qualtrics for our surveys[1]. One of the response formats which Qualtrics provides, which was suitable for our study, allows participants to mark out individual tokens in text. Each screen in our survey presented one program in a non-editable

[1]https://www.qualtrics.com/

textbox. Experts could select multiple, space-separated tokens in these programs. They were provided with the following instruction:

> You will be required to highlight the most important "snippet" in the code shown to you. A "snippet" means a substring that is crucial to help you comprehend what the function ‘foo‘ is doing. If there are multiple such "key" snippets, highlight them all.

The survey is available at this URL: `https://mit.co1.qualtrics.com/jfe/form/SV_bIpZfY2rWDlb4cC`

Tokenization. Qualtrics tokenizes based on whitespaces, which experts can select as being a beacon. Santacoder on the other hand uses a byte-pair encoding (BPE) tokenizer. Outcomes from the two tokenization methods should be aligned carefully. For example, `set(arr[k+m])` can be one Qualtrics token. The BPE-tokens for this token are the following seven tokens: `{, set, arr, [, k, +, m, ], }`. In our work, we consider the model's representation of a Qualtrics token to be the mean of the representations of the corresponding BPE tokens [Malmaud et al., 2020].

Alternate model representation. As a baseline, we use GPT-4 [Bubeck et al., 2023] to test how well the model representations can predict experts' judgement scores of beacons. We get GPT-4 to predict the beacons in a program using few-shot prompting. To predict the beacons of a program, we supply the remaining programs from the dataset and their corresponding beacons as prompts. This serves as an alternate, weakly supervised model representation since neither do we explicitly define what beacons are, nor do we define other concepts like tokens when prompting the model.

8.3 Results

8.3.1 RQ 1. Do humans consistently identify beacons?

To determine whether there exists a consensus in humans recognizing beacons in programs, we measure the inter-expert correlation of the 10 experts on the 8 programming problems they were shown. The inter-expert correlation is obtained by correlating

the judgement scores of each expert with the average of all other experts, and then avearging those correlations for all experts to get a single value. This normalized average inter-expert consensus across the ten experts and eight programming problems was 0.58 (also reported in Table 8.1, row 1). We will return to discussing this in RQ 2 in the context of other predictors of beacons.

We also study the distribution of the judgement scores of experts for each token in a program. For each token, we define its *Token Response Rate* (TRR) as TRR = $\frac{\sum_i \mathbb{1}(\mathtt{E}_i=1)}{\sum_i \mathbb{1}(\mathtt{E}_i)}$ for each expert $\mathtt{E}_i$. Each expert selects a token as either being important or not. The plots in Figure 8-2 show the distribution of TRR scores in the eight programs. The X-axis in each sub-figure shows all the unique tokens in the program that was marked important by at least one expert. The Y-axis shows the TRR corresponding to each such token.

The histograms of TRRs in 8-2 suggest that longer programs (larger total number of tokens) have a wider spread in judgement scores as compared to shorter programs, suggesting that it is easier to identify beacons in shorter programs than longer ones. The effect of program length needs to be investigated in future work. Despite the wider spread however, all the distributions show few tokens appearing in each program that have high TRRs (> 0.7). This suggests the consistent presence of a few tokens across programs which seem to be prioritized more than others in their perceived contribution to understanding those programs.

Discussion: Going beyond TRR. We note here that the TRR could possibly be a restrictive metric to measure the consensus between experts. In every program, we find the beacons marked out by the experts span multiple program tokens. For a beacon spanning a line or a set of tokens, the boundaries marked out by experts can differ by a few tokens or even lines, on either side of the intended beacon. In such cases, it is unfair to predict beacons at the level of a token using code model representations or other predictors. So, it then seems reasonable to have a metric which measures clusters of tokens or lines of code, instead of a metric measuring every token itself. That way, it should be sufficient if the predictors can predict tokens within any given cluster, without needing to predict to any one token in the cluster

specifically.

> Yes, experts identify the same set of beacons in a given program with an inter-expert agreement (across 10 experts) of $r = 0.58$

8.3.2 RQ 2. What are the predictors of beacons?

We evaluate different predictors of beacons. We test two predictors: code model representations and token surprisal, a well established predictor of reading times of text [Bicknell and Levy, 2012]. Table 8.1 reports the correlations of the predictors with each token's response rate (TRR). For each token, the TRR is the average of all the experts' judgement scores. The normalized inter-expert agreement (described in RQ 1) is reported in row 1. Since the model's representation $\in \mathbb{R}^{2048}$, we report the most correlating feature of the model. Similarly, likelihood of a token (which is the inverse of token surprisal) is computed as the log likelihood of observing a token conditioned on observing all the tokens preceding it. The model is trained to infer these likelihoods.

We report correlations in two settings: across programs, and within programs. In the former, we correlate the model representation and likelihood for each token with the TRR of that token across all programs in our dataset, totaling 228 tokens. In the latter, we calculate correlations for each program separately, and average the correlations across the 8 programs (average tokens per program = 25.3; std = 12.9). It is justifiable to treat every token independently because for any one token, its model representations and likelihood already accounts for the information in the tokens preceding it.

Table 8.1 shows the inter-expert correlation is 0.58 on average for each program, while across programs, the inter-expert correlation of 0.48. The inter-expert correlation serves as a ceiling for the predictions we should expect from other predictors.

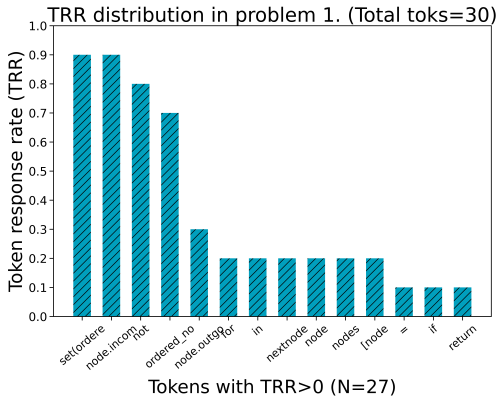
Compared to the inter-expert correlations, we find the model's best representation feature predicts beacons with an average correlation of 0.72 per program. These

```
def foo(nodes):
    ordered_nodes = [node for node in nodes if not node.incoming_nodes]

    for node in ordered_nodes:
        for nextnode in node.outgoing_nodes:
            if set(ordered_nodes).issuperset(nextnode.incoming_nodes) and nextnode not in ordered_nodes:
                ordered_nodes.append(nextnode)

    return ordered_nodes
```

Problem 1 - Expert annotations of beacons

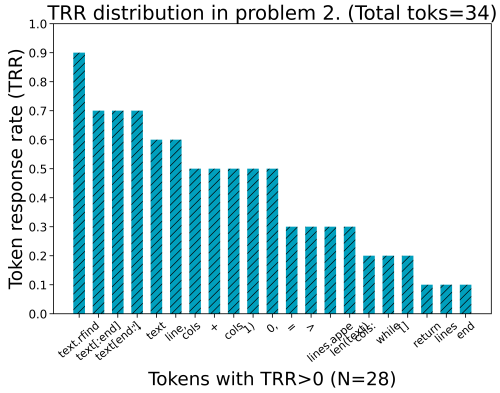


Token response rate for Problem 1

```
def foo(text, cols):
    lines = []
    while len(text) > cols:
        end = text.rfind(' ', 0, cols + 1)
        if end == -1:
            end = cols
        line, text = text[:end], text[end:]
        lines.append(line)

    lines.append(text)
    return lines
```

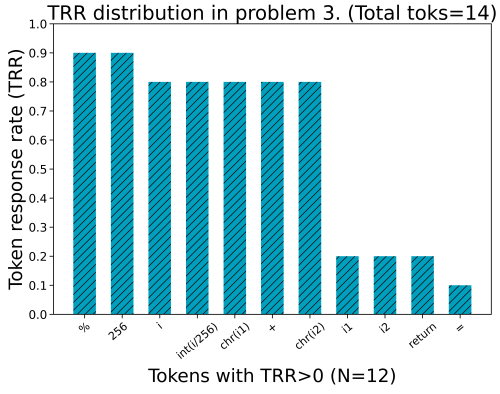
Problem 2 - Expert annotations of beacons



Token response rate for Problem 2

```
def foo(i):
    i1 = i % 256
    i2 = int(i/256)
    return chr(i1) + chr(i2)
```

Problem 3 - Expert annotations of beacons



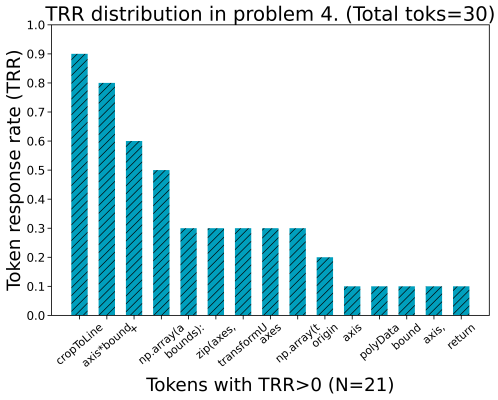
Token response rate for Problem 3

```python
def foo(polyData, transform, bounds):
    origin = np.array(transform.GetPosition())
    axes = transformUtils.getAxesFromTransform(transform)

    for axis, bound in zip(axes, bounds):
        axis = np.array(axis)/np.linalg.norm(axis)
        polyData = cropToLineSegment(polyData, origin + axis*bound[0], origin + axis*bound[1])

    return polyData
```

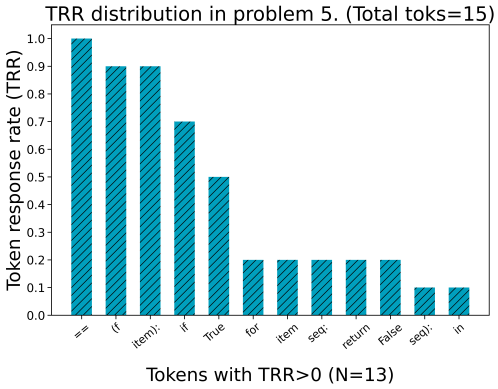
Problem 4 - Expert annotations of beacons



Token response rate for Problem 4

```python
def foo(f, seq):
    for item in seq:
        if (f == item):
            return True
    return False
```

Problem 5 - Expert annotations of beacons

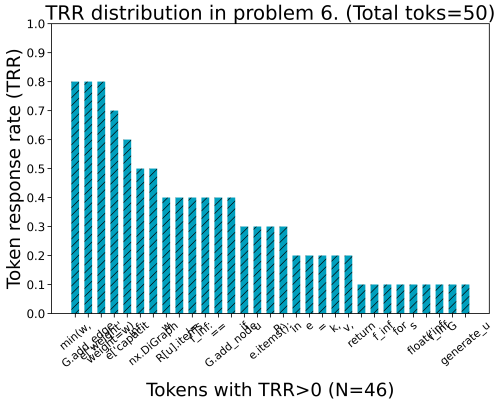


Token response rate for Problem 5

```python
def foo(R):
    s = generate_unique_node()
    G = nx.DiGraph()
    G.add_nodes_from(R)

    inf = R.graph['inf']
    f_inf = float('inf')
    for u in R:
        for v, e in R[u].items():
            w = f_inf
            for k, e in e.items():
                if e['capacity'] == inf:
                    w = min(w, e['weight'])
            if w != f_inf:
                G.add_edge(u, v, weight=w)
    return G
```

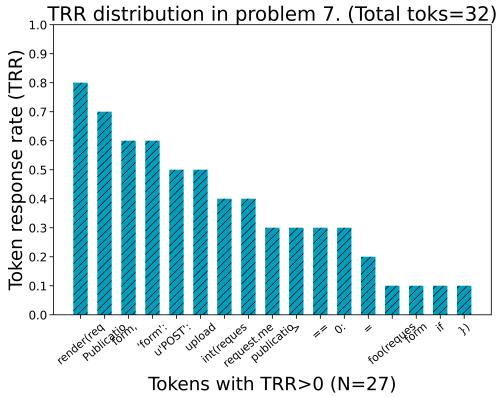
Problem 6 - Expert annotations of beacons



Token response rate for Problem 6

```
def foo(request):
    if request.method == u'POST':
        form = None
        publication_id = int(request.POST['publication_id'])
        if publication_id > 0:
            upload = Publication.objects.get(publication_id=publication_id)
            form = PublicationForm(instance=upload)
        else:
            form = PublicationForm()
        return render(request, 'publisher/my_publication/modal.html',{
            'form': form,
        })
```

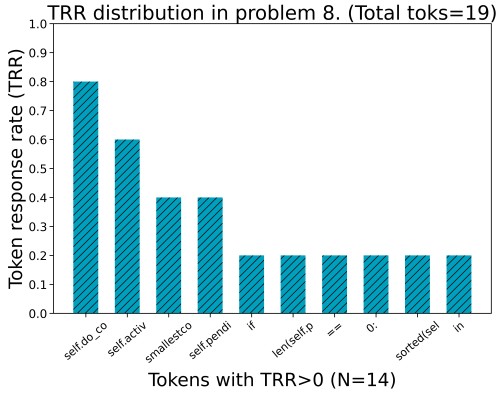
Problem 7 - Expert annotations of beacons



Token response rate for Problem 7

```
def foo(self):
    if len(self.pendingcommands) == 0:
        return
    smallestcommandnumber = sorted(self.pendingcommands.keys())[0]
    if smallestcommandnumber in self.pendingcommands:
        if self.active:
            self.do_command_propose_from_pending(smallestcommandnumber)
        else:
            self.do_command_prepare_from_pending(smallestcommandnumber)
```

Problem 8 - Expert annotations of beacons



Token response rate for Problem 8

Figure 8-2: Behavioral responses. The responses by the ten experts on each of the eight problems in our dataset. The left panels shows the problems as seen by the expert. The color gradients pertain to the token response rate (TRR): darker the shade of green, closer the TRR is to 1. The right panel shows the token-wise distribution of expert responses.

encouraging results suggest that the model representation does encode factors that affect the perception of beacons. Model correlations outperforming inter-expert correlations (0.72 vs. 0.58) on an average in programs is perhaps indicative of the relatively low number of tokens in each program (average N = 25.3). Future studies should investigate a larger number of programs, including longer programs in the dataset.

The likelihood, and thus the surprisal, of tokens does not seem to correlate with the presence of beacons. This suggests the factors driving our ability to perceive beacons are likely different than the factors affecting comprehension reading times [Bicknell and Levy, 2012], and consequently, comprehension difficulty. Future work should investigate the factors that are encoded in the model representations which correlate with beacons.

Predictor	**Correlation (r) Across programs**	**Correlation (r) Program-wise**
Human (inter-expert)	0.48	0.58
Model	0.38	0.72
Likelihood of token	-0.02	-0.07
GPT-4 (few-shot)	0.36	0.71

Table 8.1: **Predicting beacons.** The table reports Pearson correlations (r) between different predictors and human judgement scores of beacons. The correlations are computed both across all tokens appearing in the eight problems (N=228) and an average of the correlations in each of the eight programs in our dataset (average tokens per program = 25.3; std = 12.9). *Human* refers to the normalized inter-expert agreement.

Discussion: More baselines. The model representations' correlations should be interpreted in the context of multiple baselines. Baselines can be used to better understand both the inputs and outputs of our experiment design. The inputs are model representations, whose effects can be studied further. Different model architectures can be explored to see if their model representations affect the predictability of beacons. Other baselines controlling for features like the use of types, data structures, operations, and different syntax-features can be investigated as well.

To study the effect of the output, we introduce an alternate set of judgement scores—

those produced by GPT-4 [Bubeck et al., 2023]. GPT-4 was prompted to identify beacons when given a program. We see that model representations are able to predict GPT-4 identified beacons almost as well as those identified by experts. This may suggest that there could be other surface-level features which the model representations may be correlating with. To test this further, future work can investigate GPT-4's predictions of beacons by prompting it with programs whose lines are scrambled. GPT-4 should be unable to identify any meaningful lines when presented with scrambled lines. However, despite the scrambling, if GPT-4 predictions continue to correlate highly with model representations, it would confirm the central role of token-level, syntax-related features in predicting beacons.

The criterion validity of our current design has not been established. While we intend for participants to identify beacons, participants may likely be interpreting it to represent some other feature such as confusion, difficulty in reading, *etc.* To ascertain this, the experiment design can include programs containing bugs, which are delivered at random to the participants: participants identifying bugs as beacons will test participants' understanding of the concept of *importance* and *beacons*.

Discussion: Is every token not a beacons? One question that arises from this study is—should not every token be marked important, and considered a beacon? What even justifies some token being perceived to be more important than another? We address this from the lens of information theory. Several prior works have shown that the distribution of the different tokens that appear in programs are not uniform [Piantadosi, 2014, Shooman and Laemmel, 1977, Clark and Green, 1977, Chen, 1991]. One explanation offered for this observation is that our language production is optimized for communication, thus resulting in a distribution of tokens where some are more informative than others. The inclusion of non-informative tokens helps communicate information in a noisy-channel setting [Gibson et al., 2017, Ryskin et al., 2018].

Perceiving beacons also supports this general observation. If there inherently are some tokens which perceivably convey more information then others, then the experiment proposed in this work help discovering them. We go a step further in our

experiments by also attempting to explain this perceptual phenomenon (beacons) by establishing their predictors.

These results can be analyzed from the lens of computational models as well, and how language models of code are trained. If the model representations are predictive of the beacons that expert programmers identify, which we show to be the case, then it is possible that the neural mechanism involved in code comprehension is similar to the objectives employed in training code models. It is possible that beacons are an artefact of objectives such as masked language model training. To establish this, future work should look at models trained on various objectives to see if there exists any effect of the objectives on the predictions of beacons.

8.4 Related work

We provide a brief overview of relevant works in empirical and behavioral SE which have addressed the problem of code comprehension:

Comprehensibility is generally quantified as either judgement scores provided by human programmers or the number of bugs or fixes identified in the program during code-review recorded on platforms such as GitHub. While Trockman et al. [2018] identify properties with some weak correlation to comprehensibility, all other works fail to identify any determinants.

Empirical SE. The empirical SE community has attempted to predict the comprehensibility of code by its *properties* [Buse and Weimer, 2008, Zimmermann et al., 2010, Scalabrino et al., 2017]. These properties are descriptive statistics which serve as proxy representations of the code. Examples of such properties are—counts describing the syntax tree of the program such as the number of tokens in the code, the number of specific dependencies, *etc.* A mapping model $\boldsymbol{\theta}_{\text{map}}$ is learned to predict comprehensibility from these properties. Works in empirical SE that have studied this question all conclude by showing the inability of such properties in predicting comprehensibility. Further, these properties describe code-level behavior, and are incapable of identifying specific *snippets* (substrings) of a program which causally affect comprehensibility.

Behavioral SE. The behavioral SE community has stayed clear of using inherent program properties like dependence graphs, syntax trees, *etc.* as predictors of comprehensibility. Instead, they have attempted to predict comprehensibility behaviorally by positing the presence of *beacons* and *schemas* [Soloway and Ehrlich, 1984, Wiedenbeck, 1986, Letovsky, 1987]. *Beacons* refer to 'important' snippets which convey a bug-free code's functionality. Wiedenbeck [1986] shows how expert programmers consistently tend to identify these crucial parts of a code while novice programmers do not when tasked to recall the functionality of a code. In doing so, Wiedenbeck [1986] *defines* beacons as those snippets which experts tend to recall but novices are unable to appreciate the importance of, and hence do not tend to recall. We propose a computational mechanism which does not inherently depend on discovering what experts and novices believe through elaborate behavioral experiments.

Theoretical perspectives. Parnas [1972, 1979], Koppel and Jackson [2020] offer insights into the foundations of concepts like modularity and dependence which in turn directly affect our ability to understand software systems. While the theoretical underpinnings are essential for the better design of software, and attempt to unify different software design choices, it is not sufficient. We conjecture that irrespective of the principles guiding a software's design, there exists patterns in how humans seek information from any program. Our work attempts to establish and understand the drivers of such human behavior.

Probabilistic accounts of language reading. Chater and Manning [2006], Bicknell and Levy [2010], Armeni et al. [2017], Malmaud et al. [2020], Schrimpf et al. [2021b] provide probabilistic accounts of reading. Our work is closest to that of Schrimpf et al. [2021b], Malmaud et al. [2020] where behavioral information such as reading times and eye gaze information is predicted from model representations. See Ma and Jazayeri [2014] for probabilistic accounts of uncertainty in other tasks like vision and motor responses.

Chapter 9

Conclusion

I started this thesis with the following three questions:

- **Thesis Question 1: Computational perspective.** What is a good framework to evaluate code models' understanding of programs?
- **Thesis Question 2: Cognitive neuroscience perspective.** What is a good framework to understand how code comprehension happens in our brains and minds?
- **Thesis Question 3: Bridging the two perspectives.** Can computational models help in learning how our brains and minds comprehend programs? Can our brain and minds inform the better design of computational models?

My work contributes to improving each of these three perspectives in the following ways.

Computational perspective. Chapter 2 builds on the observation that humans can understand code despite simple changes made to it. I ask whether code models do the same? The method proposed in this work implements this idea, and serves as a practical baseline test of how well code models understand code. Given how general our formulation is, I show its application in generating English sentences that can elicit specific neural responses in the brain (details in Chapter 7). **Chapter 3** follows up on the brittleness of models understanding identified in Chapter 2, and proposes ways to fix the brittleness.

Chapter 4 presents a different perspective on the topic of code models. This chapter presents the first step towards training code models to comprehend and reason

about concurrent programs. It specifically develops a way forward for designing data-driven data race detectors, which can potentially improve upon the heuristics that have been proposed over the last four decades. The chapter demonstrates a strong case for how code models have to be thought of differently than NLP models. The task of understanding concurrent behavior is unique to programs.

Cognitive neuroscience perspective. Chapter 5 studies two candidate brain regions most likely responsible for code comprehension—the Multiple Demand system and the Language system. Knowledge of the functional regions of the brain involved in code comprehension allows us to probe more into the nature of information represented (stored) in these brain regions. This study presents an example of a group analysis, where the neural responses in the MD and language systems are compared across the total number of participants in the fMRI study. We show that on average, the MD system responds more consistently during code comprehension than the language system.

Bridging the two perspectives. We follow up on the group analysis on the neuroimaging work with an individual analysis of the information stored in these brain systems, presented in Chapter 6. This work presents a few firsts. We take the first steps in describing the foundations of the MD system. Programs are a natural way to describe problem-solving tasks, which the MD system is believed to specialize in. We then decode the presence of code properties in the different regions. To test for code properties which may not be enumerable, we propose using code model representations. We show a weak correspondence between the representations of a program in the brain and in code models.

Chapter 7 contributes to the bridging the two perspectives by directly demonstrating the method used in Chapter 2 to generate experiment stimuli that satisfy diverse goals. Such stimuli were recently used in Tuckute et al. [2023] to noninvasively control neural activity in higher-level cortical areas, like the language network. While we do not demonstrate the generation of code stimuli directly in this work, the method can be used to similarly learn more about the sensitivity of the MD and Language system to the presence of specific code patterns.

In **Chapter 8**, I show how language models of code, when used as proxies of expert programmer knowledge, can help study different behavioral responses seen when understanding code. This is a use-case for how code models can directly inform and improve our understanding of the cognitive bases of code understanding.

9.1 Future work

The thesis motivates a number of directions for future work that I discuss below.

9.1.1 The role of cognitive neuroscience: path ahead

The results from the cognitive neuroscience perspective presented in this thesis primarily improve our understanding of the brain bases of code comprehension. We learn of the different brain systems involved in comprehension, and establish the role of the Multiple Demand system in comprehension (Chapter 5), and also learn of the nature of programming concepts encoded in these systems (Chapter 6). While the results from these chapters improve our current understanding of the neural bases of comprehension, they are insufficient in explaining our behavior when reading and understanding code. The brain bases of understanding limits us to ask questions about the anatomy and neuroscience of information processing, while the many behavioral responses to code, which cognitive psychology helps studying, remain addressed. For instance, the behavioral responses to code understanding such as those I show in Chapter 8 will be essential to learn more about the limits of reasoning about code. Such responses will also provide us with information to propose models of our behavior, which hopefully can also inform the design of code models.

In my view, the way forward for understanding more about code comprehension is by using methods in cognitive psychology—measuring and studying behavioral responses to carefully designed experiments in code comprehension. These methods can potentially address unresolved questions such as the role of expertise in code comprehension and whether the semantics of different families of programming languages require different *mental models* to reason about, *e.g.*, determining the

skills needed to reason about functional languages, web-based languages (Javascript, PHP), numerical languages (Matlab, R, GNU Octave), or distributed and parallel languages (Go, OpenMP)—all of which anecdotally have required different *kinds* of reasoning. Addressing such questions can directly lead to the better design of user-first programming languages.

Prior works in cognitive psychology have hypothesized the different processes involved in comprehension. For example, Figure 9-1d isplays an excerpt from Letovsky [1987], where the authors describe the mental processes involved in comprehension. Unfortunately, these models have not undergone empirical validation. However, by utilizing the methods available today in cognitive psychology and machine learning, we can now revisit such inquiries and empirically establish these hypotheses. Moreover, the abundance of software artifacts and data enhances the feasibility of conducting such studies.

As an additional benefit, methods in cognitive psychology generally do not require extensive equipment and expertise like those needed for setting up neuroimaging experiments such as fMRI.

Further, this thesis addresses questions only in code comprehension. Future work should utilize cognitive psychology to address equally important questions in other activities related to programming, including code writing, code debugging, and software design.

9.1.2 Applying results from neuroimaging studies to CS education and pedagogy

To understand how the neuroimaging results from this thesis can be applied specifically to *improving* how we can understand programs, we first establish the relationship between two cognitive activities engaging the same brain system (in our case - working memory tasks and *code comprehension* engaging the MD system). A few studies have claimed that for any two cognitive activities that share the same brain resources, training one activity will lead to an improvement in the other [Jaeggi et al., 2008,

5.2.4. Summary. To summarize, we have identified several processes and knowledge types that are suggested by the conjecture data. These include

Plausible slot filling to integrate new code objects into prior expectations;

Abduction to hypothesize plausible explanations for code objects;

Planning to hypothesize plausible implementations for known goals;

Symbolic evaluation to determine what code does;

Discourse rules to drawn what conjectures from meaningful names, and from the parametrization of routines;

Generic plans to encode efficiency knowledge and to support vague initial descriptions of implementation relations;

Endorsement rules for assigning belief status to assertions.

Figure 9-1: An excerpt from Letovsky [1987] in which the authors describes the mental processes involved in comprehension. Unfortunately, such hypotheses have not yet been empirically validated.

Melby-Lervåg and Hulme, 2013]. For example, if language and music share and activate the same brain system, then tools and approaches used to engage and train one activity should be transferable to, and will lead to an improvement in the other. Since the effects of training and improving one's MD system are not well understood, it is unclear whether training on cognitively demanding non-coding tasks could improve our ability to read and understand programs.

The effect of training and improving abilities that by studying other abilities which activate the same brain systems should be explored in future work.

9.1.3 Establishing human performance for the better design of code models

Pavlick and Kwiatkowski [2019] analyze human performance on inference tasks in language. They study responses to the textual entailment (RTE) task, which expects conclusions to be drawn about the world on the basis of limited information expressed in natural language. For example, the sentence *Three dogs on a sidewalk* being true implies that the sentence *There is more than one dog here* is true. They perform this study on 50 human subjects, wherein each subject is presented 100 such entailment sentence pairs and is required to respond with one of either *entailment*, *neutral*, or *contradiction* for each pair. The authors show that humans consistently disagree on this task, and report a multi-modal distribution in their responses. Further, and importantly, they find that the uncertainty expressed by humans is not captured by state of the art inference models like BERT fine-tuned on this RTE task.

Studies such as those by Pavlick and Kwiatkowski [2019] suggest a partial understanding of our own capabilities and limitations on inference tasks. While recent advances in probing such language models for various properties Hewitt and Liang [2019], Voita and Titov [2020] is a step in the right direction in understanding these models better, they focus primarily on interpreting information learned by these black-box language models. Ambiguity faced by humans during inference is currently neither explained nor modeled in such models. The same learning applies to code

models as well.

There is a need to learn and acknowledge such gaps in our own abilities, and use such results to motivate the design of better 'general-purpose' language models of code. It is unclear though how such reconciliation can be operationalized. One observation is that neuro-symbolic systems are trained by integrating external knowledge sources to a learning model. It is possible that with the right choice of external knowledge sources, such as formal logic, relational reasoning, *etc.*, this integration might result in a performance similar to ours.

We see these ambiguities as litmus tests for any system – be it fully neural, or neuro-symbolic, in explaining human-like cognition. Establishing such ambiguities, and having neuro-symbolic models replicate them will be a worthy initial challenge. Encoding these ambiguities and integrating them in the design of neuro-symbolic models can be another challenge to follow it.

9.1.4 A case for separate architectures?

Diachek et al. [2020] investigate the neural regions involved in language comprehension. The authors find that the MD system does not consistently response to language comprehension tasks, while the language system is. Despite language understanding seemingly requiring the application of logic and symbolic manipulation – cognitive functions generally associated with the MD system, the lack of significant activity in the MD system challenges our intuition of how we cognitively process language. However, Diachek et al. [2020] suggest their results do not rule out the influence of the MD system in our ability to understand and infer language. The authors suggest that the MD system could likely be recruited for language production tasks, and in comprehension/inference tasks in everyday *noisy channel* conditions.

On the other hand, results from 5 on the neural bases of program comprehension tasks suggest that program comprehension and simulation, which typically entail computational and symbolic manipulation, strongly activate only the MD system and not the language system.

These two results when read together seem to suggest that there is a need for two

distinct architectures to support the broad range of reasoning and inference tasks in language that we engage in – one which models the MD system and the other the language system. We raise the question of whether evidence from how we anatomically process and infer language-related tasks can directly transfer to the design of neuro-symbolic computational models. It is reasonable to conceive a neurosymbolic model comprising two sub-models – one trained exclusively on symbolic and logic-related reasoning, and the other resembling a language model.

A step towards redesigning model architectures can be to first incorporate behavioral information as learning objectives. For instance, the work in Chapter 8 and other prior works like **?** provide evidence for various perceptual and behavioral limitations we possess when reading and understanding code. Gathering enough data on such behavioral responses enables ML models to learn these objectives in addition to learning other language model-related objectives. Such an ensemble of objectives can be an easy way to ensure our biases are incorporated into the code models we train and use.

9.1.5 Probing code models

Probing models that have been trained on code corpora for the concepts they learn is another challenging area of future work that is motivated by the results from Chapter 2. Chapter 3 provides solutions to address the limitations of code models demonstrated in Chapter 2, by suggesting improvements to the training architecture of code models using methods like contrastive learning and adversarial training. These methods, however, are unable to address the question of whether the models imbibe and learn specific concepts.

Recent work in probing natural language models have demonstrated promising directions of exploration to address such questions [Andreas and Klein, 2017, Hewitt and Manning, 2019, Voita and Titov, 2020, Hernandez and Andreas, 2021]. Future work should establish paradigms to similarly probe code models for code concept understanding.

Bibliography

Asm bytecode analysis framework. URL https://asm.ow2.io/. [Page 108.]

Varun Aggarwal, Shashank Srikant, and Harsh Nisar. Ameo 2015: A dataset comprising amcat test scores, biodata details and employment outcomes of job seekers. In *Proceedings of the 3rd IKDD Conference on Data Science, 2016*, CODS '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342179. doi: 10.1145/2888451.2892037. URL https://doi.org/10.1145/2888451.2892037. [Page 29.]

Aakash Agrawal, KVS Hari, and SP Arun. How does reading expertise influence letter representations in the brain? an fmri study. *Journal of Vision*, 18(10):1161–1161, 2018. [Page 142.]

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023. [Pages 182, 183, and 184.]

Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216, 2013. doi: 10.1109/MSR.2013.6624029. [Page 30.]

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100, 2016. [Pages 56 and 77.]

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018a. [Pages 37, 92, 98, 140, and 144.]

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51 (4), jul 2018b. ISSN 0360-0300. doi: 10.1145/3212695. URL https://doi.org/10.1145/3212695. [Page 30.]

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018c. [Page 45.]

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018d. URL https://openreview.net/forum?id=BJOFETxR-. [Pages 76 and 145.]

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018a. [Pages 56, 57, and 97.]

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 53(4): 404–419, 2018b. [Pages 70, 76, and 77.]

Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019a. URL https://openreview.net/forum?id=H1gKYo09tX. [Page 145.]

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019b. [Page 77.]

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019c. doi: 10.1145/3290353. URL https://doi.org/10.1145/3290353. [Page 76.]

Marie Amalric and Stanislas Dehaene. Origins of the brain networks for advanced mathematics in expert mathematicians. *Proceedings of the National Academy of Sciences*, 113(18):4909–4917, 2016. [Page 134.]

Marie Amalric and Stanislas Dehaene. A distinct cortical network for mathematical knowledge in the human brain. *NeuroImage*, 189:19–31, 2019. [Pages 122, 134, and 138.]

Katrin Amunts and Karl Zilles. Architecture and organizational principles of broca's region. *Trends in cognitive sciences*, 16(8):418–426, 2012. [Page 130.]

Richard A Andersen, Tyson Aflalo, and Spencer Kellis. From thought to action: The brain–machine interface in posterior parietal cortex. *Proceedings of the National Academy of Sciences*, 116(52):26274–26279, 2019. [Page 161.]

Andrew James Anderson, Edmund C Lalor, Feng Lin, Jeffrey R Binder, Leonardo Fernandino, Colin J Humphries, Lisa L Conant, Rajeev DS Raizada, Scott Grimm, and Xixi Wang. Multiple regions of a cortical network commonly encode the meaning of words in multiple grammatical positions of read sentences. *Cerebral cortex*, 29(6): 2396–2411, 2019. [Page 136.]

Jacob Andreas and Dan Klein. Analogs of linguistic structure in deep representations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2893–2897, Copenhagen, Denmark, September 2017. Association

for Computational Linguistics. doi: 10.18653/v1/D17-1311. URL https://aclanthology.org/D17-1311. [Page 202.]

Kristijan Armeni, Roel M Willems, and Stefan L Frank. Probabilistic language models in cognitive neuroscience: Promises and pitfalls. *Neuroscience & Biobehavioral Reviews*, 83:579–588, 2017. [Page 194.]

Nik Khadijah Nik Aznan, Jason D Connolly, Noura Al Moubayed, and Toby P Breckon. Using variable natural environment brain-computer interface stimuli for real-time humanoid robot navigation. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 4889–4895. IEEE, 2019. [Page 180.]

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016. [Page 48.]

Rachel Barr, Alecia Moser, Sylvia Rusnak, Laura Zimmermann, Kelly Dickerson, Herietta Lee, and Peter Gerhardstein. The impact of memory load and perceptual cues on puzzle learning by 24-month olds. *Developmental Psychobiology*, 58(7): 817–828, 2016. [Page 179.]

Pouya Bashivan, Kohitij Kar, and James J. DiCarlo. Neural population control via deep image synthesis. *Science*, 364, 2019. [Page 179.]

Douglas Knox Bemis and Liina Pylkkänen. Simple composition: A magnetoencephalography investigation into the comprehension of minimal linguistic phrases. *The Journal of Neuroscience*, 31:2801 – 2814, 2011. [Page 165.]

Marina U Bers, Carina González-González, and Mª Belén Armas-Torres. Coding as a playground: Promoting positive learning experiences in childhood classrooms. *Computers & Education*, 138:130–145, 2019. [Page 127.]

Marina Umaschi Bers. Coding, playgrounds and literacy in early childhood education: The development of kibo robotics and scratchjr. In *2018 IEEE global engineering education conference (EDUCON)*, pages 2094–2102. IEEE, 2018. [Page 127.]

James C Bezdek and Richard J Hathaway. Convergence of alternating optimization. *Neural, Parallel & Scientific Computations*, 11(4):351–368, 2003. [Pages 54 and 73.]

Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355, 2016. [Page 145.]

Klinton Bicknell and Roger Levy. A rational model of eye movement control in reading. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, page 1168–1178, USA, 2010. Association for Computational Linguistics. [Pages 183 and 194.]

Klinton Bicknell and Roger Levy. Word predictability and frequency effects in a rational model of reading. In *Proceedings of the 34th Annual Meeting of the Cognitive Science Society*, page 126–131, Sapporo, Japan, 2012. [Pages 34, 187, and 191.]

Pavol Bielik and Martin Vechev. Adversarial robustness for code. *arXiv preprint arXiv:2002.04694*, 2020. [Pages 47, 48, and 68.]

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933484. doi: 10.1145/1167473.1167488. URL https://doi.org/10.1145/1167473.1167488. [Page 104.]

Idan Blank, Nancy Kanwisher, and Evelina Fedorenko. A functional dissociation between language and multiple-demand systems revealed in patterns of bold signal fluctuations. *Journal of neurophysiology*, 112(5):1105–1118, 2014. [Pages 123, 124, 132, and 135.]

Idan A Blank and Evelina Fedorenko. Domain-general brain regions do not track linguistic input as closely as language-selective regions. *Journal of Neuroscience*, 37 (41):9999–10011, 2017. [Page 123.]

Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3): 268–308, 2003. [Page 53.]

Hans-J Boehm. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 9–14, 2012. [Page 100.]

Benjamin Bowman, Craig Laprade, Yuede Ji, and H Howie Huang. Detecting lateral movement in enterprise computer networks with unsupervised graph ai. In *RAID*, pages 257–268, 2020. [Page 106.]

Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004. [Pages 53 and 171.]

Stephen Boyd, Neal Parikh, and Eric Chu. *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc, 2011. [Page 55.]

Serdar Boztas. Entropies, guessing, and cryptography. *Department of Mathematics, Royal Melbourne Institute of Technology, Tech. Rep*, 6:2–3, 1999. [Page 33.]

Jonathan R Brennan and Liina Pylkkänen. Meg evidence for incremental sentence composition in the anterior temporal lobe. *Cognitive science*, 41:1515–1531, 2017. [Page 148.]

Joan Bresnan and Jonni M Kanerva. Locative inversion in chichewa: A case study of factorization in grammar. *Linguistic inquiry*, pages 1–50, 1989. [Page 165.]

Matthew Brett, Ingrid S Johnsrude, and Adrian M Owen. The problem of functional localization in the human brain. *Nature reviews neuroscience*, 3(3):243–249, 2002. [Page 130.]

Korbinian Brodmann. *Vergleichende Lokalisationslehre der Grosshirnrinde in ihren Prinzipien dargestellt auf Grund des Zellenbaues*. Barth, 1909. [Pages 120, 129, and 130.]

Tom B. Brown, 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. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL `https://arxiv.org/abs/2005.14165`. [Pages 182 and 183.]

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023. [Pages 185 and 192.]

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. *Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations*, page 511–521. Association for Computing Machinery, New York, NY, USA, 2021a. ISBN 9781450380379. URL `https://doi.org/10.1145/3404835.3462840`. [Page 67.]

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521, 2021b. [Page 64.]

Raymond PL Buse and Westley R Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130, 2008. [Pages 181 and 193.]

Santiago A Cadena, George H Denfield, Edgar Y Walker, Leon A Gatys, Andreas S Tolias, Matthias Bethge, and Alexander S Ecker. Deep convolutional models improve predictions of macaque v1 responses to natural images. *PLoS computational biology*, 15(4):e1006897, 2019. [Page 141.]

Lu Cao, Dandan Huang, Yue Zhang, Xiaowei Jiang, and Yanan Chen. Brain decoding using fnirs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14): 12602–12611, May 2021. [Page 148.]

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017. [Pages 46 and 52.]

Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. Do programmers prefer predictable expressions in code? *Cognitive Science*, 44(12): e12921, 2020a. [Pages 147 and 181.]

Casey Casalnuovo, E Morgan, and P Devanbu. Does surprisal predict code comprehension difficulty. In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society*. Cognitive Science Society Toronto, Canada, 2020b. [Page 181.]

Joao Castelhano, Isabel C Duarte, Carlos Ferreira, Joao Duraes, Henrique Madeira, and Miguel Castelo-Branco. The role of the insula in intuitive expert bug detection in computer code: an fmri study. *Brain imaging and behavior*, 13(3):623–637, 2019. [Pages 120, 144, and 147.]

Charlotte Caucheteux, Alexandre Gramfort, and Jean-Remi King. Decomposing lexical and compositional syntax and semantics with deep language models. In *International Conference on Machine Learning*, pages 1336–1348. PMLR, 2021. [Page 148.]

Nick Chater and Christopher D Manning. Probabilistic models of language processing and acquisition. *Trends in cognitive sciences*, 10(7):335–344, 2006. [Page 194.]

Rui Pedro Chaves and Jeruen E. Dery. Frequency effects in subject islands. *Journal of Linguistics*, 55:475 – 521, 2018. [Page 164.]

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021a. [Pages 63 and 145.]

Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. Varclr: Variable semantic representation pre-training via contrastive learning, 2021b. [Page 67.]

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020. [Pages 63, 70, 72, and 73.]

Yehong Chen. Zipf's law in natural languages, programming languages, and command languages : the simon-yule approach. *International Journal of Systems Science*, 22: 2299–2312, 1991. [Pages 34 and 192.]

Zimin Chen, Vincent Josua Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. PLUR: A unifying, graph-based view of program learning, understanding, and repair. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021c. URL `https://openreview.net/forum?id=GEm4o9A6Jfb`. [Page 63.]

François Chollet. On the measure of intelligence. *CoRR*, abs/1911.01547, 2019. URL `http://arxiv.org/abs/1911.01547`. [Page 180.]

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022. [Pages 182 and 183.]

Ching-Yao Chuang, Joshua Robinson, Lin Yen-Chen, Antonio Torralba, and Stefanie Jegelka. Debiased contrastive learning. *arXiv preprint arXiv:2007.00224*, 2020. [Page 74.]

David Glenn Clark and Jeffrey L Cummings. Aphasia. In *Neurological Disorders*, pages 265–275. Elsevier, 2003. [Page 123.]

Douglas W Clark and C Cordell Green. An empirical study of list structure in lisp. *Communications of the ACM*, 20(2):78–87, 1977. [Pages 34 and 192.]

Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746, 2002. [Page 64.]

Will Crichton, Maneesh Agrawala, and Pat Hanrahan. The role of working memory in program tracing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380966. doi: 10.1145/3411764.3445257. URL https://doi.org/10.1145/3411764.3445257. [Pages 147 and 182.]

Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/cummins21a.html. [Page 106.]

Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020. [Page 76.]

Fatma Deniz, Anwar O Nunez-Elizalde, Alexander G Huth, and Jack L Gallant. The representation of semantic information across human cerebral cortex during listening versus reading is invariant to stimulus modality. *The journal of neuroscience.*, 39, 2019. [Page 157.]

Evgeniia Diachek, Idan Blank, Matthew Siegelman, Josef Affourtit, and Evelina Fedorenko. The domain-general multiple demand (md) network does not support core aspects of language comprehension: a large-scale fmri investigation. *Journal of Neuroscience*, 40(23):4536–4550, 2020. [Page 201.]

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. Contrastive learning for source code with structural and functional properties. *CoRR*, abs/2110.03868, 2021. URL https://arxiv.org/abs/2110.03868. [Page 64.]

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70, 2019. [Page 112.]

John C Duchi, Peter L Bartlett, and Martin J Wainwright. Randomized smoothing for stochastic optimization. *SIAM Journal on Optimization*, 22(2):674–701, 2012. [Page 55.]

John Duncan. The multiple-demand (md) system of the primate brain: mental programs for intelligent behaviour. *Trends in cognitive sciences*, 14(4):172–179, 2010. [Pages 122 and 138.]

John Duncan and Adrian M Owen. Common regions of the human frontal lobe recruited by diverse cognitive demands. *Trends in neurosciences*, 23(10):475–483, 2000. [Page 138.]

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751*, 2017. [Pages 48 and 68.]

Logan Engstrom, Andrew Ilyas, and Anish Athalye. Evaluating and understanding the robustness of adversarial logit pairing. *arXiv preprint arXiv:1807.10272*, 2018. [Pages 18 and 55.]

Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 286–28610. IEEE, 2018. [Page 120.]

Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering*, pages 1–39, 2019. [Page 140.]

Lijie Fan, Sijia Liu, Pin-Yu Chen, Gaoyuan Zhang, and Chuang Gan. When does contrastive learning preserve adversarial robustness from pretraining to finetuning? In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL `https://openreview.net/forum?id=70kOIgjKhbA`. [Pages 67 and 74.]

Evelina Fedorenko and Idan A Blank. Broca's area is not a natural kind. *Trends in cognitive sciences*, 24(4):270–284, 2020. [Pages 20, 122, and 130.]

Evelina Fedorenko, Po-Jang Hsieh, Alfonso Nieto-Castañón, Susan Whitfield-Gabrieli, and Nancy Kanwisher. New method for fmri investigations of language: defining rois functionally in individual subjects. *Journal of neurophysiology*, 104(2):1177–1194, 2010a. [Pages 123, 124, 128, 131, and 135.]

Evelina Fedorenko, Po-Jang Hsieh, Alfonso Nieto-Castanon, Susan L. Whitfield-Gabrieli, and Nancy G. Kanwisher. New method for fmri investigations of language: defining rois functionally in individual subjects. *Journal of neurophysiology*, 104 2: 1177–94, 2010b. [Page 176.]

Evelina Fedorenko, Michael K Behr, and Nancy Kanwisher. Functional specificity for high-level linguistic processing in the human brain. *Proceedings of the National Academy of Sciences*, 108(39):16428–16433, 2011. [Page 130.]

Evelina Fedorenko, John Duncan, and Nancy Kanwisher. Language-selective and domain-general regions lie side by side within broca's area. *Current Biology*, 22(21): 2059–2062, 2012. [Page 130.]

Evelina Fedorenko, John Duncan, and Nancy Kanwisher. Broad domain generality in focal regions of frontal and parietal cortex. *Proceedings of the National Academy of Sciences*, 110(41):16616–16621, 2013. [Pages 123, 128, and 131.]

Evelina Fedorenko, Anna Ivanova, Riva Dhamala, and Marina Umaschi Bers. The language of programming: a cognitive perspective. *Trends in cognitive sciences*, 23 (7):525–528, 2019. [Pages 118 and 145.]

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. [Page 153.]

Jason Fischer, John G Mikhael, Joshua B Tenenbaum, and Nancy Kanwisher. Functional neuroanatomy of intuitive physical inference. *Proceedings of the national academy of sciences*, 113(34):E5072–E5081, 2016. [Page 138.]

Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 121–133, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542490. URL https://doi.org/10.1145/1542476.1542490. [Page 104.]

Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, page 1–8, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300827. doi: 10.1145/1806672.1806674. URL https://doi.org/10.1145/1806672.1806674. [Page 108.]

Benjamin Floyd, Tyler Santander, and Westley Weimer. Decoding the representation of code in the brain: An fmri study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186. IEEE, 2017. [Pages 117, 118, 120, 144, and 147.]

Richard Futrell, Edward Gibson, Harry J. Tily, Idan Asher Blank, Anastasia Vishnevetsky, Steven T. Piantadosi, and Evelina Fedorenko. The natural stories corpus: a reading-time corpus of english texts containing rare syntactic constructions. *Language Resources and Evaluation*, 55:63 – 77, 2020. [Page 165.]

Jian Gao, Xin Yang, Yu Jiang, Han Liu, Weiliang Ying, and Xian Zhang. Jbench: A dataset of data races for concurrency testing. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 6–9, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196451. URL https://doi.org/10.1145/3196398.3196451. [Pages 104, 105, 114, and 115.]

Matt Gardner, Yoav Artzi, Victoria Basmov, Jonathan Berant, Ben Bogin, Sihao Chen, Pradeep Dasigi, Dheeru Dua, Yanai Elazar, Ananth Gottumukkala, Nitish Gupta, Hannaneh Hajishirzi, Gabriel Ilharco, Daniel Khashabi, Kevin Lin, Jiangming Liu, Nelson F. Liu, Phoebe Mulcaire, Qiang Ning, Sameer Singh, Noah A. Smith, Sanjay Subramanian, Reut Tsarfaty, Eric Wallace, Ally Zhang, and Ben Zhou. Evaluating models' local decision boundaries via contrast sets. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1307–1323, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.117. URL https://aclanthology.org/2020.findings-emnlp.117. [Pages 172 and 175.]

Jon Gauthier and Roger Levy. Linking artificial and human neural representations of language. *arXiv preprint arXiv:1910.01244*, 2019. [Page 148.]

Saeed Ghadimi, Guanghui Lan, and Hongchao Zhang. Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization. *Mathematical Programming*, 155(1-2):267–305, 2016. [Page 54.]

Edward Gibson, Caitlin Tan, Richard Futrell, Kyle Mahowald, Lars Konieczny, Barbara Hemforth, and Evelina Fedorenko. Don't underestimate the benefits of being misunderstood. *Psychological science*, 28(6):703–712, 2017. [Page 192.]

Gary H Glover. Overview of functional magnetic resonance imaging. *Neurosurgery Clinics*, 22(2):133–139, 2011. [Pages 121 and 149.]

Jesse Gomez, Michael Barnett, and Kalanit Grill-Spector. Extensive childhood experience with pokémon suggests eccentricity drives organization of visual cortex. *Nature human behaviour*, 3(6):611–624, 2019. [Page 142.]

Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *International Conference on Learning Representations*, arXiv preprint arXiv:1412.6572, 2015. [Page 64.]

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014. [Page 46.]

Tushar Goswamy, Ishika Singh, Ahsan Barkati, and Ashutosh Modi. Adapting a language model for controlled affective text generation. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 2787–2801, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics. doi: 10.18653/v1/2020.coling-main.251. URL https://aclanthology.org/2020.coling-main.251. [Pages 173 and 176.]

Sven Gowal, Po-Sen Huang, Aaron van den Oord, Timothy Mann, and Pushmeet Kohli. Self-supervised adversarial robustness for the low-label, high-data regime. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=bgQek2O63w. [Page 67.]

Philip J Guo. Non-native english speakers learning computer programming: Barriers, desires, and design opportunities. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–14, 2018. [Page 140.]

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 1345–1351, 2017. [Page 45.]

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8342–8360, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.740. URL https://aclanthology.org/2020.acl-main.740. [Page 173.]

Michael Hahn and Frank Keller. Modeling task effects in human reading with neural attention. *CoRR*, abs/1808.00054, 2018. URL http://arxiv.org/abs/1808.00054. [Page 182.]

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9729–9738, 2020. [Pages 63, 70, and 72.]

Micha Heilbron, Kristijan Armeni, Jan-Mathijs Schoffelen, Peter Hagoort, and Floris P. de Lange. A hierarchy of linguistic predictions during natural language comprehension. *Proceedings of the National Academy of Sciences*, 119(32):e2201968119, 2022. doi: 10.1073/pnas.2201968119. URL https://www.pnas.org/doi/abs/10.1073/pnas.2201968119. [Pages 164 and 165.]

Vincent J Hellendoorn and Anand Ashok Sawant. The growing cost of deep learning for source code. *Communications of the ACM*, 65(1):31–33, 2021. [Page 145.]

Victor W Henderson. Paul broca's less heralded contributions to aphasia research: Historical perspective and contemporary relevance. *Archives of Neurology*, 43(6): 609–612, 1986. [Page 119.]

Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 526–537, 2022. doi: 10.1109/SANER53432.2022.00070. [Pages 19, 26, 64, 67, 68, 69, 72, 74, 77, 78, and 87.]

Evan Hernandez and Jacob Andreas. The low-dimensional linear geometry of contextualized word representations. In *Proceedings of the 25th Conference on Computational Natural Language Learning*, pages 82–93, Online, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.conll-1.7. URL https://aclanthology.org/2021.conll-1.7. [Page 202.]

John Hewitt and Percy Liang. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2733–2743, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1275. URL https://www.aclweb.org/anthology/D19-1275. [Page 200.]

John Hewitt and Christopher D. Manning. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1419. URL https://aclanthology.org/N19-1419. [Page 202.]

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016. [Pages 33 and 34.]

Klaus Hoenig, Cornelia Müller, Bärbel Herrnberger, Eun-Jin Sim, Manfred Spitzer, Günter Ehret, and Markus Kiefer. Neuroplasticity of semantic representations for musical instruments in professional musicians. *NeuroImage*, 56(3):1714–1725, 2011. [Page 142.]

Sebastian Hoffmann. Are low-frequency complex prepositions. *Corpus approaches to grammaticalization in English*, 13:171, 2004. [Page 165.]

Phillip Howard, Gadi Singer, Vasudev Lal, Yejin Choi, and Swabha Swayamdipta. Neurocounterfactuals: Beyond minimal-edit counterfactuals for richer data augmentation. *arXiv preprint arXiv:2210.12365*, 2022. [Pages 172, 173, 174, and 175.]

Jennifer Hu, Jon Gauthier, Peng Qian, Ethan Wilcox, and Roger Levy. A systematic assessment of syntactic generalization in neural language models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1725–1744, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.158. URL https://aclanthology.org/2020.acl-main.158. [Page 165.]

Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. *SIGPLAN Not.*, 50(6):165–174, jun 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737975. URL https://doi.org/10.1145/2813885.2737975. [Page 108.]

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. *SIGPLAN Not.*, 49(6):337–348, jun 2014a. ISSN 0362-1340. doi: 10.1145/2666356.2594315. URL https://doi.org/10.1145/2666356.2594315. [Pages 105, 109, and 112.]

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN*

conference on programming language design and implementation, pages 337–348, 2014b. [Page 104.]

Yu Huang, Xinyu Liu, Ryan Krueger, Tyler Santander, Xiaosu Hu, Kevin Leach, and Westley Weimer. Distilling neural representations of data structure manipulation using fmri and fnirs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 396–407. IEEE, 2019. [Pages 118, 120, 144, and 147.]

Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. Biases and differences in code review using medical imaging and eye-tracking: genders, humans, and machines. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 456–468, 2020. [Page 120.]

David H. Hubel and Torsten N. Wiesel. Republication of the journal of physiology (1959) 148, 574-591: Receptive fields of single neurones in the cat's striate cortex. 1959. *The Journal of physiology*, 587 Pt 12:2721–32, 2009. [Page 179.]

HuggingFace. Codeberta - a roberta-like model trained on the codesearchnet dataset from github. *HuggingFace*, 2020. URL `https://huggingface.co/huggingface/CodeBERTa-small-v1`. [Page 153.]

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019. URL `http://arxiv.org/abs/1909.09436`. [Pages 77 and 153.]

Yoshiharu Ikutani and Hidetake Uwano. Brain activity measurement during program comprehension with nirs. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6. IEEE, 2014. [Page 120.]

Yoshiharu Ikutani, Takatomi Kubo, Satoshi Nishida, Hideaki Hata, Kenichi Matsumoto, Kazushi Ikeda, and Shinji Nishimoto. Expert programmers have fine-tuned cortical representations of source code. *Eneuro*, 2020. [Page 120.]

Yoshiharu Ikutani, Takatomi Kubo, Satoshi Nishida, Hideaki Hata, Kenichi Matsumoto, Kazushi Ikeda, and Shinji Nishimoto. Expert programmers have fine-tuned cortical representations of source code. *Eneuro*, 8(1), 2021. [Pages 144 and 147.]

Anna A Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H Kean, Riva Dhamala, Una-May O'Reilly, Marina U Bers, and Evelina Fedorenko. Comprehension of computer code relies primarily on domain-general executive brain regions. *Elife*, 9: e58906, 2020. [Pages 6, 40, 117, 125, 144, 145, 146, 147, 148, 149, 150, 155, 159, and 161.]

Susanne M Jaeggi, Martin Buschkuehl, John Jonides, and Walter J Perrig. Improving fluid intelligence with training on working memory. *Proceedings of the National Academy of Sciences*, 105(19):6829–6833, 2008. [Pages 140 and 198.]

Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. *CoRR*, abs/2112.02969, 2021a. URL https://arxiv.org/abs/2112.02969. [Page 63.]

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5954–5971, Online and Punta Cana, Dominican Republic, November 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.482. URL https://aclanthology.org/2021.emnlp-main.482. [Pages 63, 64, 65, 66, 67, 68, 72, 73, 74, 75, 76, 77, 78, 87, and 88.]

Shailee Jain and Alexander Huth. Incorporating context into language encoding models for fmri. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. [Page 148.]

Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. RADBench: A concurrency bug benchmark suite. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*, Berkeley, CA, May 2011. USENIX Association. URL https://www.usenix.org/conference/hotpar-11/radbench-concurrency-bug-benchmark-suite. [Pages 104 and 114.]

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=rkE3y85ee. [Page 171.]

Minseok Jeon, Myungho Lee, and Hakjoo Oh. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020. [Page 106.]

Paloma Jeretic, Alex Warstadt, Suvrat Bhooshan, and Adina Williams. Are natural language inference models IMPPRESsive? Learning IMPlicature and PRESupposition. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8690–8705, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.768. URL https://aclanthology.org/2020.acl-main.768. [Page 164.]

Jeya Vikranth Jeyakumar, Joseph Noor, Yu-Hsi Cheng, Luis Garcia, and Mani Srivastava. How can i explain this to you? an empirical study of deep neural network explanation methods. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 4211–4222. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/2c29d89cc56cdb191c60db2f0bae796b-Paper.pdf. [Page 83.]

Jinghan Jia, Shashank Srikant, Tamara Mitrovska, Chuang Gan, Shiyu Chang, Sijia Liu, and Una-May O'Reilly. Clawsat: Towards both robust and accurate code models. *arXiv preprint arXiv:2211.11711*, 2022. [Pages 6, 37, and 63.]

Ziyu Jiang, Tianlong Chen, Ting Chen, and Zhangyang Wang. Robust pre-training by adversarial contrastive learning. *arXiv preprint arXiv:2010.13337*, 2020. [Page 67.]

Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 675–681, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02658-4. [Pages 108 and 111.]

Christian Gram Kalhauge and Jens Palsberg. Sound deadlock prediction. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi: 10.1145/3276516. URL `https://doi.org/10.1145/3276516`. [Page 104.]

Yukiyasu Kamitani and Frank Tong. Decoding the visual and subjective contents of the human brain. *Nature neuroscience*, 8(5):679–685, 2005. [Page 151.]

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020a. [Page 184.]

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020b. [Page 63.]

Nancy Kanwisher. Functional specificity in the human brain: a window into the functional architecture of the mind. *Proceedings of the National Academy of Sciences*, 107(25):11163–11170, 2010. [Page 120.]

Zachary Karas, Andrew Jahn, Westley Weimer, and Yu Huang. Connecting the dots: rethinking the relationship between code and prose writing with functional connectivity. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 767–779, 2021. [Page 144.]

Divyansh Kaushik, Eduard Hovy, and Zachary Lipton. Learning the difference that makes a difference with counterfactually-augmented data. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=Sklgs0NFvr`. [Pages 172, 174, and 175.]

Alan Kennedy, Joël Pynte, Wayne S. Murray, and Shirley-Anne S. Paul. Frequency and predictability effects in the dundee corpus: An eye movement analysis. *Quarterly Journal of Experimental Psychology*, 66:601 – 618, 2013. [Page 165.]

Seyed-Mahdi Khaligh-Razavi and Nikolaus Kriegeskorte. Deep supervised, but not unsupervised, models may explain it cortical representation. *PLoS computational biology*, 10(11):e1003915, 2014. [Page 141.]

Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. Examples are not enough, learn to criticize! criticism for interpretability. *Advances in neural information processing systems*, 29, 2016. [Page 84.]

Minseon Kim, Jihoon Tack, and Sung Ju Hwang. Adversarial self-supervised contrastive learning. *arXiv preprint arXiv:2006.07589*, 2020. [Page 67.]

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. *SIGPLAN Not.*, 52(6):157–170, jun 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062374. URL https://doi.org/10.1145/3140587.3062374. [Pages 104, 105, and 110.]

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022. [Page 184.]

Arnold R Kochari, Ashley Glen Lewis, Jan-Mathijs Schoffelen, and Herbert Schriefers. Semantic and syntactic composition of minimal adjective-noun phrases in dutch: An meg study. *Neuropsychologia*, 155, 2018. [Page 165.]

James Koppel and Daniel Jackson. Demystifying dependence. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 48–64, 2020. [Pages 181 and 194.]

Nikolaus Kriegeskorte. Pattern-information analysis: from stimulus decoding to computational-model testing. *Neuroimage*, 56(2):411–421, 2011. [Page 151.]

Ryan Krueger, Yu Huang, Xinyu Liu, Tyler Santander, Westley Weimer, and Kevin Leach. Neurological divide: an fmri study of prose and code writing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 678–690. IEEE, 2020. [Pages 120, 144, and 147.]

Vicky Tzuyin Lai, Roel M. Willems, and Peter Hagoort. Feel between the lines: Implied emotion in sentence comprehension. *Journal of Cognitive Neuroscience*, 27: 1528–1541, 2015. [Pages 163 and 164.]

Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439. [Page 94.]

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL https://doi.org/10.1145/359545.359563. [Pages 104, 105, and 110.]

Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987. [Pages 23, 181, 182, 194, 198, and 199.]

Hector Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *Thirteenth international conference on the principles of knowledge representation and reasoning*, 2012. [Page 172.]

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31, 2018a. [Page 81.]

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018b. [Page 45.]

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017. [Page 184.]

Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. Jacontebe: A benchmark suite of real-world java concurrency bugs (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189, 2015. doi: 10.1109/ASE.2015.87. [Pages 104 and 114.]

Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003. [Page 64.]

Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. Stochastic optimization of program obfuscation. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE, 2017. [Pages 48 and 49.]

Hong Liu, Mingsheng Long, Jianmin Wang, and Michael I. Jordan. Towards understanding the transferability of deep representations, 2020a. URL `https://openreview.net/forum?id=BylKL1SKvr`. [Pages 81 and 83.]

Risheng Liu, Jiaxin Gao, Jin Zhang, Deyu Meng, and Zhouchen Lin. Investigating bi-level optimization for learning and vision from a unified perspective: A survey and beyond. *arXiv preprint arXiv:2101.11517*, 2021. [Page 73.]

Xiaoping Liu, Jihong Ren, Wendem Beyene, Simon Ku, Chin Hong Heah, and Sherman Hsu. Design optimization and accurate extraction of on-die decoupling capacitors for high-performance applications. In *2018 IEEE 68th Electronic Components and Technology Conference (ECTC)*, pages 1712–1719. IEEE, 2018. [Page 180.]

Yun-Fei Liu, Judy Kim, Colin Wilson, and Marina Bedny. Computer code comprehension shares neural resources with formal logical inference in the fronto-parietal network. *Elife*, 9:e59340, 2020b. [Pages 118, 120, 138, 144, 145, 147, 148, 149, and 159.]

Beth L Losiewicz. *The effect of frequency on linguistic morphology*. The University of Texas at Austin, 1992. [Page 165.]

Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021. [Page 106.]

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021a. [Page 76.]

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. NeuroLogic decoding: (un)supervised neural text generation with predicate logic constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4288–4299, Online, June 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.339. URL https://aclanthology.org/2021.naacl-main.339. [Page 175.]

Wei Ji Ma and Mehrdad Jazayeri. Neural coding of uncertainty and probability. *Annual review of neuroscience*, 37:205–220, 2014. [Page 194.]

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=S1jE5L5gl. [Page 171.]

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *2018 ICLR*, arXiv preprint arXiv:1706.06083, 2018a. [Pages 18 and 66.]

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018b. [Pages 26, 46, 53, 60, 64, 68, 75, 77, 78, and 86.]

Jonathan Malmaud, Roger Levy, and Yevgeni Berzak. Bridging information-seeking human gaze and machine reading comprehension. In *CoNLL*. Association for Computational Linguistics, 2020. [Pages 34, 182, 185, and 194.]

Manuel Martín-Loeches, Anabel Fernández, Annekathrin Schacht, Werner Sommer, Pilar Casado, Laura Jiménez-Ortega, and Sabela Fondevila. The influence of emotional words on sentence processing: Electrophysiological and behavioral evidence. *Neuropsychologia*, 50:3262–3272, 2012. [Page 164.]

Rebecca Marvin and Tal Linzen. Targeted syntactic evaluation of language models. *arXiv preprint arXiv:1808.09031*, 2018. [Page 165.]

Mara Mather, John T Cacioppo, and Nancy Kanwisher. How fmri can inform cognitive theories. *Perspectives on Psychological Science*, 8(1):108–113, 2013. [Page 118.]

Umang Mathur, Dileep Kini, and Mahesh Viswanathan. What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018. [Pages 104, 105, and 110.]

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. The complexity of dynamic data race prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, page 713–727, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371049. doi: 10.1145/3373718.3394783. URL https://doi.org/10.1145/3373718.3394783. [Page 104.]

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. Optimal prediction of synchronization-preserving races. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434317. URL https://doi.org/10.1145/3434317. [Pages 104, 105, and 110.]

Monica Melby-Lervåg and Charles Hulme. Is working memory training effective? a meta-analytic review. *Developmental psychology*, 49(2):270, 2013. [Pages 140 and 200.]

Microsoft. Codegpt-small-py. *HuggingFace*, 2021. URL https://huggingface.co/microsoft/CodeGPT-small-py. [Page 153.]

Zachary Mineroff, Idan Asher Blank, Kyle Mahowald, and Evelina Fedorenko. A robust dissociation among the language, multiple demand, and default mode networks: evidence from inter-region correlations in effect size. *Neuropsychologia*, 119:501–511, 2018. [Page 123.]

Tom M Mitchell, Svetlana V Shinkareva, Andrew Carlson, Kai-Min Chang, Vicente L Malave, Robert A Mason, and Marcel Adam Just. Predicting human brain activity associated with the meanings of nouns. *science*, 320(5880):1191–1195, 2008. [Page 148.]

Takeru Miyato, Andrew M Dai, and Ian Goodfellow. Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725*, 2016. [Page 64.]

Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M German. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: A controlled experiment. In *Companion proceedings of the 36th international conference on software engineering*, pages 448–451, 2014. [Page 120.]

Samuel A. Nastase, Yun-Fei Liu, Hanna Hillman, Asieh Zadbood, Liat Hasenfratz, Neggin Keshavarzian, Janice Chen, Christopher John Honey, Yaara Yeshurun, Mor Regev, Mai Nguyen, Claire H. C. Chang, Christopher A. Baldassano, Olga Lositsky, Erez Simony, Michael A. Chow, Yuan Chang Leong, Paula P. Brooks, Emily T. Micciche, Gina Choe, Ariel Goldstein, Tamara Vanderwal, Yaroslav O. Halchenko, Kenneth A. Norman, and Uri Hasson. The "narratives" fmri dataset for evaluating models of naturalistic language comprehension. *Scientific Data*, 8, 2021. [Page 165.]

Allen Newell, John Calman Shaw, and Herbert A Simon. Elements of a theory of human problem solving. *Psychological review*, 65(3):151, 1958. [Page 40.]

Alfonso Nieto-Castañón and Evelina Fedorenko. Subject-specific functional localizers increase sensitivity and functional resolution of multi-subject analyses. *Neuroimage*, 63(3):1646–1669, 2012. [Page 128.]

Kenneth A Norman, Sean M Polyn, Greg J Detre, and James V Haxby. Beyond mind-reading: multi-voxel pattern analysis of fmri data. *Trends in cognitive sciences*, 10(9):424–430, 2006. [Page 151.]

Paul Nuyujukian, Jose Albites Sanabria, Jad Saab, Chethan Pandarinath, Beata Jarosiewicz, Christine H Blabe, Brian Franco, Stephen T Mernoff, Emad N Eskandar, John D Simeral, et al. Cortical control of a tablet computer by people with paralysis. *PloS one*, 13(11):e0204566, 2018. [Page 161.]

Christophe Pallier, Anne-Dominique Devauchelle, and Stanislas Dehaene. Cortical representation of the constituent structure of sentences. *Proceedings of the National Academy of Sciences*, 108(6):2522–2527, 2011. [Page 148.]

Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. Running experiments on amazon mechanical turk. *Judgment and Decision making*, 5(5):411–419, 2010. [Page 30.]

David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. [Pages 181 and 194.]

David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE transactions on software engineering*, (2):128–138, 1979. [Pages 181 and 194.]

Chris Parnin, Janet Siegmund, and Norman Peitek. On the nature of programmer expertise. In *Ppig*, page 16, 2017. [Page 120.]

Alicia Parrish and Liina Pylkkänen. Conceptual combination in the latl with and without syntactic composition. *Neurobiology of Language*, 3:46–66, 2021. [Page 165.]

Ellie Pavlick and Tom Kwiatkowski. Inherent disagreements in human textual inferences. *Transactions of the Association for Computational Linguistics*, 7:677–694, 2019. [Page 200.]

Palle Martin Pedersen. Methods and systems for identifying an area of interest in protectable content, September 14 2010. US Patent 7,797,245. [Page 46.]

Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C Hofmeister, and André Brechmann. Simultaneous measurement of program comprehension with fmri and eye tracking: A case study. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018. [Pages 144 and 147.]

Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fmri study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 524–536. IEEE, 2021. [Pages 144, 147, and 148.]

Marcela Perrone-Bertolotti, Jan Kujala, Juan R. Vidal, Carlos M. Hamame, Tomas Ossandon, Olivier Bertrand, Lorella Minotti, Philippe Kahane, Karim Jerbi, and Jean-Philippe Lachaux. How silent is silent reading? intracerebral evidence for top-down activation of temporal voice areas during reading. *Journal of Neuroscience*, 32(49):17554–17562, 2012. [Page 155.]

Steven T Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic bulletin & review*, 21:1112–1130, 2014. [Pages 33, 34, and 192.]

Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. IEEE, 2020. [Pages 48 and 68.]

Russell A Poldrack. Inferring mental states from neuroimaging data: from reverse inference to large-scale decoding. *Neuron*, 72(5):692–697, 2011. [Page 130.]

Scott R Portnoff. The introductory computer programming course is first and foremost a language course. *ACM Inroads*, 9(2):34–52, 2018. [Page 140.]

Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018. [Page 45.]

Chantel S Prat, Tara M Madhyastha, Malayka J Mottarella, and Chu-Hsuan Kuo. Relating natural language aptitude to individual differences in learning programming languages. *Scientific reports*, 10(1):1–10, 2020. [Page 147.]

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021. [Page 153.]

Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading authorship attribution of source code using adversarial learning. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 479–496, 2019. [Pages 48 and 68.]

Md Rabin, Rafiqul Islam, and Mohammad Amin Alipour. Evaluation of generalizability of neural program analyzers under semantic-preserving transformations. *arXiv preprint arXiv:2004.07313*, 2020. [Pages 48 and 68.]

Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043*, 2020. [Pages 25, 47, 48, 49, 52, 56, 57, 58, 59, and 61.]

Teodor Rares Begu. Modelling concurrency bugs using machine learning. Master's thesis, Imperial College London, 2020. [Pages 6, 38, 91, and 103.]

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015. [Page 30.]

Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016a. [Page 56.]

Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747, oct 2016b. ISSN 0362-1340. doi: 10.1145/3022671.2984041. URL `https://doi.org/10.1145/3022671.2984041`. [Page 77.]

Fabian Ritter and Sebastian Hack. Pmevo: portable inference of port mappings for out-of-order processors by evolutionary optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 608–622, 2020. [Page 180.]

Jake Roemer, Kaan Genç, and Michael D Bond. Smarttrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 747–762, 2020. [Page 104.]

Daniel Ross. Small corpora and low-frequency phenomena: try and beyond contemporary, standard english. *Corpus*, (18), 2018. [Page 165.]

Joshua S Rule, Joshua B Tenenbaum, and Steven T Piantadosi. The child as hacker. *Trends in cognitive sciences*, 24(11):900–915, 2020. [Pages 146 and 161.]

Rachel Ryskin, Richard Futrell, Swathi Kiran, and Edward Gibson. Comprehenders model the nature of noise in the environment. *Cognition*, 181:141–150, 2018. [Page 192.]

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating data race witnesses by an smt-based analysis. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 313–327, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-20398-5. [Page 109.]

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997. [Pages 104 and 105.]

Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427. IEEE, 2017. [Pages 181 and 193.]

Martin Schrimpf, Jonas Kubilius, Ha Hong, Najib J. Majaj, Rishi Rajalingham, Elias B. Issa, Kohitij Kar, Pouya Bashivan, Jonathan Prescott-Roy, Franziska Geiger, Kailyn Schmidt, Daniel L. K. Yamins, and James J. DiCarlo. Brain-score: Which artificial neural network for object recognition is most brain-like? *bioRxiv*, 2020. [Page 160.]

Martin Schrimpf, Idan Asher Blank, Greta Tuckute, Carina Kauf, Eghbal A. Hosseini, Nancy Kanwisher, Joshua B. Tenenbaum, and Evelina Fedorenko. The neural architecture of language: Integrative modeling converges on predictive processing. *Proceedings of the National Academy of Sciences*, 118(45), 2021a. [Page 148.]

Martin Schrimpf, Idan Asher Blank, Greta Tuckute, Carina Kauf, Eghbal A. Hosseini, Nancy Kanwisher, Joshua B. Tenenbaum, and Evelina Fedorenko. The neural architecture of language: Integrative modeling converges on predictive processing. *Proceedings of the National Academy of Sciences*, 118(45):e2105646118, 2021b. doi: 10.1073/pnas.2105646118. URL `https://www.pnas.org/doi/abs/10.1073/pnas.2105646118`. [Pages 182 and 194.]

Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, 49(1):1–37, 2016. [Page 64.]

Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 308–311. IEEE, 2017. [Page 120.]

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021. [Page 64.]

Dan Schwartz, Mariya Toneva, and Leila Wehbe. Inducing brain-relevant bias in natural language processing models. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. [Page 148.]

Eugene S Schwartz. A dictionary for minimum redundancy encoding. *Journal of the ACM (JACM)*, 10(4):413–439, 1963. [Page 33.]

Cory Shain, Idan Blank, Marten van Schijndel, Evelina Fedorenko, and William Schuler. fmri reveals language-specific predictive coding during naturalistic sentence comprehension. *Neuropsychologia*, 2019a. [Page 123.]

Cory Shain, Idan Asher Blank, Marten van Schijndel, Evelina Fedorenko, and William Schuler. fmri reveals language-specific predictive coding during naturalistic sentence comprehension. *Neuropsychologia*, 138, 2019b. [Page 163.]

Vinay Shashidhar, Nishant Pandey, and Varun Aggarwal. Automatic spontaneous speech grading: A novel feature derivation technique using the crowd. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1085–1094, Beijing, China, July 2015a. Association for Computational Linguistics. doi: 10.3115/v1/P15-1105. URL https://aclanthology.org/P15-1105. [Page 29.]

Vinay Shashidhar, Nishant Pandey, and Varun Aggarwal. Spoken english grading: Machine learning with crowd intelligence. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 2089–2097, New York, NY, USA, 2015b. Association for Computing Machinery. ISBN 9781450336642. doi: 10.1145/2783258.2788595. URL https://doi.org/10.1145/2783258.2788595. [Page 29.]

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *CoRR*, abs/1911.02150, 2019. URL http://arxiv.org/abs/1911.02150. [Page 184.]

Summer L Sheremata, Katherine C Bettencourt, and David C Somers. Hemispheric asymmetry in visuotopic posterior parietal cortex emerges with visual short-term memory load. *Journal of Neuroscience*, 30(38):12581–12588, 2010. [Page 134.]

Richard M. Shiffrin, Danielle S. Bassett, Nikolaus Kriegeskorte, and Joshua B. Tenenbaum. The brain produces mind by modeling. *Proceedings of the National Academy of Sciences*, 117(47):29299–29301, 2020. doi: 10.1073/pnas.1912340117. URL https://www.pnas.org/doi/abs/10.1073/pnas.1912340117. [Pages 34 and 35.]

M Shooman and A Laemmel. Statistical theory of computer programs information content and complexity. In *COMPCON'77*, pages 341–342. IEEE Computer Society, 1977. [Pages 34 and 192.]

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Neural Information Processing Systems*, 2018. [Page 45.]

Matthew Siegelman, Idan A Blank, Zachary Mineroff, and Evelina Fedorenko. An attempt to conceptually replicate the dissociation between syntax and semantics during sentence comprehension. *Neuroscience*, 413:219–229, 2019. [Page 164.]

Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pages 378–389, 2014. [Pages 31, 117, 118, and 120.]

Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150, 2017. [Pages 118, 144, and 147.]

Bhanu Pratap Singh and Varun Aggarwal. Apps to measure motor skills of vocational workers. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, page 340–350, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344616. doi: 10.1145/2971648.2971739. URL https://doi.org/10.1145/2971648.2971739. [Page 29.]

Gursimran Singh, Shashank Srikant, and Varun Aggarwal. Question independent grading using machine learning: The case of computer program grading. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 263–272, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939696. URL https://doi.org/10.1145/2939672.2939696. [Page 30.]

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. *Sound Predictive Race Detection in Polynomial Time*, page 387–400. Association for Computing Machinery, New York, NY, USA, 2012. ISBN 9781450310833. URL https://doi.org/10.1145/2103656.2103702. [Pages 104 and 105.]

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL https://aclanthology.org/D13-1170. [Page 172.]

Elliot Soloway. Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986. [Page 125.]

Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on software engineering*, (5):595–609, 1984. [Pages 42, 181, 182, and 194.]

Shashank Srikant. Vulcan: classifying vulnerabilities in solidity smart contracts using dependency-based deep program representations. Master's thesis, Massachusetts Institute of Technology, 2020. [Pages 6, 92, and 98.]

Shashank Srikant and Varun Aggarwal. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 1887–1896, New York, NY, USA, 2014a. Association for Computing Machinery. ISBN 9781450329569. doi: 10.1145/2623330.2623377. URL https://doi.org/10.1145/2623330.2623377. [Pages 30 and 45.]

Shashank Srikant and Varun Aggarwal. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1887–1896, 2014b. [Page 181.]

Shashank Srikant and Una-May O'Reilly. Can cognitive neuroscience inform neuro-symbolic inference models? In *Is Neuro-Symbolic SOTA still a myth for Natural Language Inference? The first workshop*, 2021. URL https://openreview.net/forum?id=iXv7fYSQ54. [Pages 42 and 161.]

Shashank Srikant, Rohit Takhar, Vishal Venugopal, and Varun Aggarwal. Skill evaluation. *Commun. ACM*, 62(11):60–61, oct 2019. ISSN 0001-0782. doi: 10.1145/3355268. URL https://doi.org/10.1145/3355268. [Page 29.]

Shashank Srikant, Nicolas Lesimple, and Una-May O'Reilly. Dependency-based neural representations for classifying lines of programs. *arXiv preprint arXiv:2004.10166*, 2020. [Page 97.]

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=PH5PH9ZO_4. [Pages 6, 19, 37, 45, 64, 67, 68, 69, 72, 74, 77, 78, and 145.]

Shashank Srikant, Ben Lipkin, Anna Ivanova, Evelina Fedorenko, and Una-May O' Reilly. Convergent representations of computer programs in human and artificial neural networks. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 18834–18849. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/77b5aaf2826c95c98e5eb4ab830073de-Paper-Conference.pdf. [Pages 7, 41, 143, and 182.]

Shashank Srikant, Anna A. Ivanova, Yotaro Sueoka, Hope H. Kean, Riva Dhamala, Evelina Fedorenko, Marina U. Bers, and Una-May O'Reilly. Program comprehension does not primarily rely on the language centers of the human brain, 2023a. [Pages 6, 39, and 117.]

Shashank Srikant, Greta Tuckute, Sijia Liu, and Una-May O'Reilly. Goli: Goal-optimized linguistic stimuli for psycholinguistics and cognitive neuroscience. In submission, 2023b. [Pages 7, 41, and 163.]

Megha Srivastava, Erdem Biyik, Suvir Mirchandani, Noah Goodman, and Dorsa Sadigh. Assistive teaching of motor control tasks to humans. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=k3MX8EK6Zf. [Page 180.]

Steven E Stemler, Varun Aggarwal, and Siddharth Nithyanand. Knowing what not to do is a critical job skill: evidence from 10 different scoring methods. *International Journal of Selection and Assessment*, 24(3):229–245, 2016. [Page 29.]

Dong Su, Huan Zhang, Hongge Chen, Jinfeng Yi, Pin-Yu Chen, and Yupeng Gao. Is robustness the cost of accuracy?–a comprehensive study on the robustness of 18 deep image classification models. *arXiv preprint arXiv:1808.01688*, 2018. [Page 64.]

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf. [Page 153.]

Rohit Takhar and Varun Aggarwal. Grading uncompilable programs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):9389–9396, Jul. 2019. doi: 10.1609/aaai.v33i01.33019389. URL https://ojs.aaai.org/index.php/AAAI/article/view/4987. [Page 30.]

Mosaad Al Thokair, Minjian Zhang, Umang Mathur, and Mahesh Viswanathan. Dynamic race detection with o(1) samples. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571238. URL https://doi.org/10.1145/3571238. [Page 112.]

Mariya Toneva and Leila Wehbe. Interpreting and improving natural-language processing (in machines) with natural language-processing (in the brain). In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/749a8e6c231831ef7756db230b4359c8-Paper.pdf. [Page 148.]

Mariya Toneva, Tom M. Mitchell, and Leila Wehbe. Combining computational controls with natural text reveals new aspects of meaning composition. *bioRxiv*, 2022. [Page 160.]

Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. "automatically assessing code understandability" reanalyzed: Combined metrics matter. In *International Conference on Mining Software Repositories*, MSR, pages 314–318. ACM, 2018. doi: https://doi.org/10.1145/3196398.3196441. [Pages 181 and 193.]

Fabian David Tschopp, Michael B. Reiser, and Srinivas C. Turaga. A connectome based hexagonal lattice convolutional network model of the drosophila visual system, 2018. [Page 160.]

Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=SyxAb30cY7. [Page 64.]

Greta Tuckute, Aalok Sathe, Shashank Srikant, Maya Taliaferro, Mingye Wang, Martin Schrimpf, Kendrick Kay, and Evelina Fedorenko. Driving and suppressing the human language network using large language models. *bioRxiv*, 2023. doi: 10.1101/2023.04.16.537080. URL https://www.biorxiv.org/content/early/2023/04/16/2023.04.16.537080. [Pages 42, 163, and 196.]

Chris Turner. Towards a new pedagogical approach to some and any based on large-scale corpus analysis. 2020. [Page 165.]

Nathalie Tzourio-Mazoyer, Brigitte Landeau, Dimitri Papathanassiou, Fabrice Crivello, Olivier Etard, Nicolas Delcroix, Bernard Mazoyer, and Marc Joliot. Automated anatomical labeling of activations in spm using a macroscopic anatomical parcellation of the mni mri single-subject brain. *Neuroimage*, 15(1):273–289, 2002. [Page 128.]

Abhishek Unnam, Rohit Takhar, and Varun Aggarwal. Grading emails and generating feedback. *International Educational Data Mining Society*, 2019. [Page 29.]

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999. [Page 108.]

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. [Page 153.]

Elena Voita and Ivan Titov. Information-theoretic probing with minimum description length. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 183–196, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.14. URL https://aclanthology.org/2020.emnlp-main.14. [Pages 200 and 202.]

Eric Wallace, Jens Tuyls, Junlin Wang, Sanjay Subramanian, Matt Gardner, and Sameer Singh. AllenNLP interpret: A framework for explaining predictions of NLP models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural*

Language Processing (EMNLP-IJCNLP): System Demonstrations, pages 7–12, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-3002. URL https://aclanthology.org/D19-3002. [Pages 168 and 170.]

Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*, pages 256–272. Springer, 2009. [Pages 104 and 109.]

Feng Wang, Huaping Liu, Di Guo, and Fuchun Sun. Unsupervised representation learning by invariancepropagation. *arXiv preprint arXiv:2010.11694*, 2020a. [Page 74.]

Ke Wang and Mihai Christodorescu. Coset: A benchmark for evaluating neural program embeddings. *arXiv preprint arXiv:1905.11445*, 2019. [Pages 48 and 68.]

Michael Wang, Shashank Srikant, Malavika Samak, and Una-May O'Reilly. Raceinjector: Injecting races to evaluate and learn dynamic race detection algorithms. State Of the Art in Program Analysis (SOAP), Workshop at PLDI, 2023. [Pages 6, 38, and 91.]

Shaonan Wang, Jiajun Zhang, Nan Lin, and Chengqing Zong. Probing brain activation patterns by dissociating semantics and syntax in sentences. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):9201–9208, Apr. 2020b. doi: 10.1609/aaai.v34i05.6457. [Page 148.]

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020c. [Page 77.]

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. AAAI, 2022. [Page 67.]

Yawei Wang, Giacomo Bacco, and Nicola Bianchi. Geometry analysis and optimization of pm-assisted reluctance motors. *IEEE Transactions on Industry Applications*, 53 (5):4338–4347, 2017. doi: 10.1109/TIA.2017.2702111. [Page 180.]

Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. Learning semantic program embeddings with graph interval neural network. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020d. [Page 76.]

Zhao Wang and Aron Culotta. Robustness to spurious correlations in text classification via automatically generated counterfactuals. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(16):14024–14031, May 2021. doi: 10.1609/aaai.v35i16.17651. [Page 172.]

Alex Warstadt, Yu Cao, Ioana Grosu, Wei Peng, Hagen Blix, Yining Nie, Anna Alsop, Shikha Bordia, Haokun Liu, Alicia Parrish, Sheng-Fu Wang, Jason Phang, Anhad Mohananey, Phu Mon Htut, Paloma Jeretic, and Samuel R. Bowman. Investigating BERT's knowledge of language: Five analysis methods with NPIs. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2877–2887, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1286. URL https://aclanthology.org/D19-1286. [Page 164.]

Alex Warstadt, Alicia Parrish, Haokun Liu, Anhad Mohananey, Wei Peng, Sheng-Fu Wang, and Samuel R. Bowman. BLiMP: The benchmark of linguistic minimal pairs for English. *Transactions of the Association for Computational Linguistics*, 8: 377–392, 2020. doi: 10.1162/tacl_a_00321. URL https://aclanthology.org/2020.tacl-1.25. [Page 164.]

Leila Wehbe, Idan Asher Blank, Cory Shain, Richard Futrell, Roger Levy, Titus von der Malsburg, Nathaniel Smith, Edward Gibson, and Evelina Fedorenko. Incremental language comprehension difficulty predicts activity in the language network but not the multiple demand network. *Cerebral Cortex*, 31(9):4006–4023, 2021. [Page 163.]

Susan Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697–709, 1986. [Pages 42, 181, and 194.]

Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3733–3742, 2018. [Page 70.]

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017. [Page 117.]

Will Xiao and Gabriel Kreiman. Xdream: Finding preferred stimuli for visual neurons using generative networks and gradient-free optimization. *PLoS Computational Biology*, 16, 2020. [Page 179.]

Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. Topology attack and defense for graph neural networks: An optimization perspective. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 3961–3967. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/550. URL https://doi.org/10.24963/ijcai.2019/550. [Page 171.]

Daniel L Yamins, Ha Hong, Charles Cadieu, and James J DiCarlo. Hierarchical modular optimization of convolutional networks achieves representations similar to macaque it and human ventral stream. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani,

and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper/2013/file/9a1756fd0c741126d7bbd4b692ccbd91-Paper.pdf. [Page 145.]

Daniel LK Yamins, Ha Hong, Charles F Cadieu, Ethan A Solomon, Darren Seibert, and James J DiCarlo. Performance-optimized hierarchical models predict neural responses in higher visual cortex. *Proceedings of the national academy of sciences*, 111(23):8619–8624, 2014. [Page 141.]

Linyi Yang, Jiazheng Li, Padraig Cunningham, Yue Zhang, Barry Smyth, and Ruihai Dong. Exploring the efficacy of automatically generated counterfactuals for sentiment analysis. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 306–316, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.26. URL https://aclanthology.org/2021.acl-long.26. [Page 172.]

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf. [Page 153.]

Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493, 2022. [Page 68.]

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *arXiv preprint arXiv:1910.07517*, 2019. [Pages 47, 48, and 49.]

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020. [Pages 19, 64, 67, 68, and 69.]

Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. Gobench: A benchmark suite of real-world go concurrency bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 187–199, 2021. doi: 10.1109/CGO51591.2021.9370317. [Pages 104 and 114.]

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf. [Page 103.]

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176, 2020. [Page 48.]

Yihua Zhang, Guanhua Zhang, Prashant Khanduri, Mingyi Hong, Shiyu Chang, and Sijia Liu. Revisiting and advancing fast adversarial training through the lens of bi-level optimization. In *International Conference on Machine Learning*, pages 26693–26712, 2022. [Page 73.]

Wei Zhao, Maxime Peyrard, Fei Liu, Yang Gao, Christian M. Meyer, and Steffen Eger. MoverScore: Text generation evaluating with contextualized embeddings and earth mover distance. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 563–578, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1053. URL https://aclanthology.org/D19-1053. [Page 175.]

Michael Zhivich and Robert K Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009. [Page 100.]

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pages 10197–10207, 2019. [Pages 45 and 46.]

Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third international conference on software testing, verification and validation*, pages 421–428. IEEE, 2010. [Pages 181 and 193.]

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations (ICLR)*, 2021. [Page 153.]