Title
LEARNING PROGRAM EMBEDDING FROM UNLABELED SOURCE CODE

Author
Ahmed, Toufique

Publication Date
2023

Peer reviewed|Thesis/dissertation

LEARNING PROGRAM EMBEDDING FROM UNLABELED SOURCE CODE

By

TOUFIQUE AHMED
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Premkumar Devanbu, Chair

Vladimir Filkov

Kenji Sagae

Committee in Charge

2023

Contents

Abstract

Machine-learning models can reach very high performance with *supervised* training, where they learn from labeled data. However, supervised training requires annotating data with desired output labels, which can be a difficult and time-consuming task. Meanwhile, advancements in deep learning models and technology have made it possible to train very large models, which was not feasible a few years ago. Although training such big models requires a substantial amount of supervised data, models can overcome this limitation by first learning from un-labeled data. Pre-trained language models enable us to achieve state-of-the-art performance from large-scale models with limited supervised data. During the pre-training stage, models are exposed to unlabeled data, and their weights can be adjusted using self-supervised tasks such as filling masks or spans, de-noising artificially induced noises, or simple auto-regressive token generation. These tasks help the model learn token distributions and the context of the programming language. After acquiring this knowledge, the model can be easily fine-tuned or used even without any fine-tuning for a specific task.

This thesis began with work on foundation models, which are pre-trained on simple tasks like mask-filling and de-noising, and then are fine-tuned for task-specific applications. We show how to effectively apply pre-trained language models in SE tasks, including traditional ones such as code correction and novel ones such as decompiling binaries. We also investigate the effectiveness of multilingual training and demonstrate how knowledge can be transferred from one language to another, thereby improving the model's performance on three tasks: code summarization, code search, and method name prediction. We also investigated what foundation models learn during the pre-training stage. It is evident that learning syntactical distribution is relatively easier and can be done using tasks such as MLM. However, it is unclear whether the model learns semantic distribution and is robust with respect to meaning-preserving transformations. In our work, we found that the model learns semantics and is robust to meaning-preserving transformations.

More recently, Large language models (LLMs), with a few billion to 540 billion parameters, trained on billions of tokens, have become available. They are proficient at few-shot learning, where just a few examples provided at the time of querying are sufficient. These models have largely eliminated the need for a fine-tuning stage and can perform even with a few training samples

or none at all. It's worth noting that during zero-shot and few-shot learning, we don't change the parameters of the model; we just use the prompt to divert the model to condition its text generation in a more desirable manner. However, the model's performance heavily relies on the prompt, and prompt engineering for different tasks has become a focus of attention for numerous researchers. In this work, we demonstrate how to automatically semantically augment prompts for code summarization, achieving state-of-the-art performance on this task.

Acknowledgments

To begin with, I would like to express my gratitude to my advisor, Professor Premkumar Devanbu, for his invaluable guidance and mentorship that have played a significant role in shaping me as a researcher. Throughout my PhD journey, he introduced me to various research domains, and his unwavering support has motivated me to explore different topics over the past five years. Besides his profound intellect, Prem possesses exceptional qualities of compassion, humility, and kindness that make working with him a privilege. Choosing to pursue my academic journey at Davis under his mentorship has undoubtedly been one of the best decisions I have ever made.

At the beginning of my PhD journey, I was fortunate to have Vincent Hellendoorn and Casey Casalnuovo as my lab mates. Their invaluable support and plethora of ideas helped me tremendously. I owe Vincent a special thanks for his guidance in familiarizing myself with deep learning models and enabling me to contribute to the research community at an early stage. Furthermore, I would like to express my gratitude to my other lab mates, including Kevin, David, Claudio, and others, who came later and kept me inspired with their ideas. I would also like to acknowledge Anand Sawant, a postdoc in our lab, whose collaboration was one of the best experiences of my research career. Few people have the remarkable understanding and competence that Vincent and Anand possess when it comes to working with complex systems. Their expertise and insights have been immensely valuable to me.

Thanks Prem for introducing me to and connecting me with so many talented researchers in our community. I would also like to extend my appreciation to my collaborators and mentors, including Vladimir Filkov, Tom Zimmermann, Baishakhi Ray, Christian Bird, Earl Barr, Kenji Sagae, Bogdan Vasilescu, Arie van Dreusen, Saikat Chakraborty, Emily Morgan and many others. Their constant inspiration, valuable ideas, unwavering support, and kind words have been instrumental in my research journey, and I am honored to have known them during my PhD.

Lastly, I want to express my gratitude to my parents, elder sister, and my wife for their relentless support. Without their encouragement and backing, I would never have been able to accomplish my PhD.

CHAPTER 1

Introduction

The introduction of Large Language Models (LLMs) has revolutionized pre-trained language models. LLMs are decoder-only models that have been trained using an autoregressive next token prediction task. These models have an enormous capacity (ranging from a few billion to 540 billion parameters) and are trained with significantly more data than prior pre-trained models. The combination of their capacity and extensive training has made LLMs incredibly powerful, to the point where they no longer require fine-tuning. LLMs, such as GPT-3 [**37**, **43**], are few-shot learners, and various prompt engineering techniques can significantly boost their performance.

LLMs however, are the latest development in a long history of using language models. In the early days, n-gram models were the SOTA in language modeling. With neural networks becoming available, the earliest models for deep were Recursive Neural Networks; these were augmented to include attention [**18**] and memory [**82**]; but these architectures were limited because of the need for "back propagation through time" which limited efficiency and paralellizability Transformer networks solved this problem, introducing multi-layer, multi-head attention, which afforded much higher levels of parallelization, and an unprecedented level of scalability: as GPUs became faster (and cheaper to operate), and VRAM capacities increased, transformer models provide an as-yet unbounded capacity to scale up in parameter count and performance. With this apparently unlimited level of architectural- and device-afforded scalability, the the research community then shifted its attention to better methods of training, starting with 2-stage training in foundation models, then to prompt-engineering in large language models, and then more sophisticated methods such as reinforcement-learning with human feedback. The work in this thesis began in the early days of transformer models, and has introduced a range of innovations (and innovative applications) in software engineering, tracking the capabilities of neural models.

One of the early development was foundation models, which we begin with. Foundation models (*e.g.*,BERT [**56**], GPT [**161**], CodeBERT [**64**], GraphCodeBERT [**69**]) are currently achieving best-in-class performance for a wide range of tasks in both natural language and code [**32**]. The models work in 2 stages, first "pre-training" to learn statistics of language (or code) construction from very large-scale corpora in a self-supervised fashion, and then using smaller labeled datasets to "fine-tune" for specific tasks. Training a high-capacity deep learning model is challenging because it requires a vast amount of data. Having sufficient labeled data required for traditional supervised learning is nearly impossible for big models. Foundation models provide a way of training such models with the unlabeled data in a self-supervised way in the pre-training stage. It is worth noting that the term LLM gained popularity following the emergence of GPT-3 models. While some publications include pre-trained models such as BERT under the LLM umbrella, for the purposes of this dissertation, we will only refer to generative models with billions of parameters as LLM.

Though foundation models generally perform quite well, it is not trivial to apply them in Software Engineering. Firstly, in Chapter 2, getting data for pre-training and fine-tuning can sometimes be challenging. For example, recovering library function usage in optimized binaries requires building many C projects, which is cumbersome. We present a pipeline to build a significant number of C projects and generate data from them for both pre-training and fine-tuning. We show that pre-training can help achieve best-in-class performance in recovering library function usage in optimized binaries. This work was published in IEEE Transactions on Software Engineering [**7**].

Secondly, code sequences are longer than NLP sequences in most cases. For example, a simple java program can be more than 400 tokens. Prior works [**71**,**171**] apply Neural Machine Translation (NMT) model to fix student program. NMT models are designed to handle much shorter sequences in NLP. Therefore, applying a single NMT model can not fix a program with more than 400-500 tokens. In Chapter 3, we propose a pipeline consisting of 3 deep-learning models that can fix very long programs and achieve state-of-the-art performance. This work was also published in IEEE Transactions on Software Engineering [**9**].

There are several models pre-trained, especially for programming languages. These models are pre-trained with a multilingual dataset like CodeSearchNet [**89**], which consists of six programming languages (*i.e.*, Ruby, JS, Java, Go, PHP, and Python). Though multilingual pre-training is a well-accepted idea in Software Engineering, multilingual fine-tuning has not been investigated yet. Researchers in the NLP area have reported that multilingual training is beneficial for low-resource languages [**53**, **72**, **165**, **186**]. Several papers show that multilingual-trained models show better performance [**117**, **185**] and are more practical to deploy [**17**]. For some programming languages (*e.g.,* Ruby) labeled data is less abundant; in others (*e.g.,* JavaScript), the available data may be more focused on some application domains and thus less diverse. We find evidence that multilingual training data (across different languages) can be used to amplify performance. We study this for three different tasks: code summarization, code retrieval, and function naming. We note that this data-augmenting approach is broadly compatible with different tasks, languages, and machine-learning models in Chapter 4. In the following sections, we briefly introduce the ideas and contributions we discuss in the dissertation.This work was accepted at 44th International Conference on Software Engineering [**4**].

Although foundation models are achieving high-levels of performance in many tasks, it is still unclear what type of knowledge these models gather during pre-training. Do they learn semantics, or just the syntax of the program? Understanding the model's capabilities can guide us to create more robust and better models. With this aim, we conducted several meaning-preserving transformations and observed how the model reacted. Note that semantic inconsistency is hard to resolve with just syntactical proficiency. We found that these models can respond to queries that involve semantic understanding, even if the model is not explicitly trained to do so. In Chapter 5, we will present our detailed results and observations. A submission is being prepared for CACM.

In this dissertation, we also applied LLMs to the code summarization task and discovered that combining BM25 retrieved samples with semantic augmentation of prompts can help improve the model's performance. In the following sections, we introduce automatic semantic augmentation of prompt in LLM. In Chapter 6, we will discuss this in more detail. A submission is being prepared for the 46th International Conference on Software Engineering [**10**].

In the following sections, we provide an overview of each of the main thrusts in each of the chapters of the dissertation, beginning with our work reverse engineering library function usage from binaries (without source code, of course).

1.1. Recovering Usages of Library Function

In their seminal work, Chikofsky and Cross [**46**], define *Software Reverse Engineering* as "the process of analyzing a subject system to (1) identify the system's components and their interrelationships and (2) create representations of the system in another form or at a higher level of abstraction". Understanding the behavior of software binaries generated by potentially untrusted parties have many motivations: such binaries may incorporate *e.g.*, stolen intellectual property (such as patented or protected algorithms or data), unauthorized access to system resources, or malicious behavior of various sorts. The ability to reverse engineer binaries would make it more difficult to conceal potential bad behavior, and thus act as a deterrent, and this would enhance public confidence in, and overall free exchange of, software technology.

Reverse engineering of an *optimized* binary is quite challenging, since compilers substantially transform source code structures to create the binary. This is done primarily to improve run-time performance; however, some compilers support deliberate obfuscation of source code details, to protect IP, or for security reasons. The resulting binary could be stripped of all information such as variable names, code comments, and user-defined types. Binaries compiled using `gcc` can be optimized in the interest of run-time performance benefits, (even if compilation *per se* takes longer). Optimizations include function inlining, array vectorization, and loop unrolling. This dramatically alters the code at the assembly level, making it substantially more challenging to decompile the binary successfully.

Two tools are industry standard for reverse engineering: Hexrays IDA Pro [**25**] and Ghidra [**147**]. These tools incorporate two distinct functionalities: a *disassembler* (convert binary to assembly code), and a *decompiler* (convert assembly code to a higher-order representation, similar to C code), and a variety of data flow analysis tools. Both tools can handle binaries that have been compiled on a variety of architectures (such as x86/64 or ARM64).

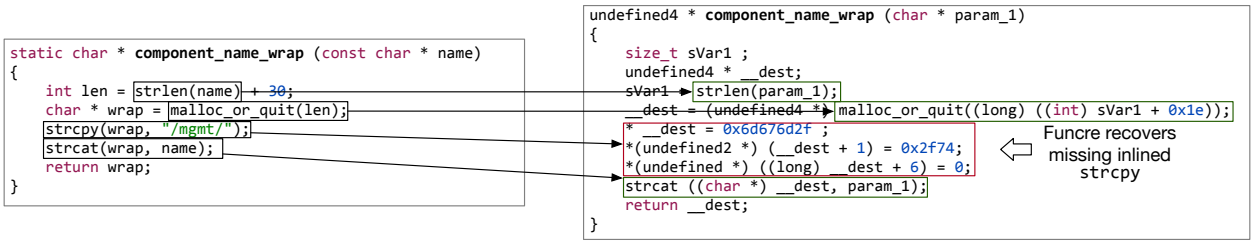


FIGURE 1.1. Finding inline functions in real-world[1]decompiled version of original C source code

In the field of security, quite a bit of work has focused on understanding the behavior of malicious applications by examining their library API calls [**90**, **140**, **173**, **188**, **199**, **217**, **218**, **219**, **220**]. The intuition behind this is that calls made to library APIs (such as Windows DLLs) can capture the important underlying semantics of the malware's attacking behaviour [**216**]. However, uncovering API calls is particularly hard as the compiler might have mangled the inlined body of the *called* function together with the code at the *calling* site in complex ways. Both Ghidra and Hexrays have specially engineered functionality for recovering calls to library functions . These recovery functions are considered very important by the developers of these tools and are explicitly documented and advertised.

Ghidra and Hexrays use a pattern-based approach to recover library calls. They rely on a pattern database, which must be *manually maintained* to include a pattern (or patterns) for each possible inlined function, each possible optimization level, compiler type, platform, *etc.* We adopt a more scalable, data-driven approach and propose to *learn* to identify those functions that are most often used in the data. To that end, we develop our tool, FUNCRE, which finds inlined library function invocations even in binaries compiled with higher optimization levels. As an illustration, in Figure 1.1, we present a sample of original source code (from GitHub) and its Ghidra output for compilation with Os. From the decompiled version, it's evident that recovering `strlen`, `malloc_or_quit` and, `strcat` is possible for Ghidra, but the `strcpy` gets inlined with copying of pointers to stack addresses after optimization. The pattern-database approach used by Ghidra works well for the three of the four functions; but Ghidra fails to recover the latter, (*strcpy*), because the more complex pattern required is not available in its database. The precision of Ghidra is 1.0 for this example, but the recall is lower (0.75). We note that our tool, FUNCRE, *starts with*,

and *builds upon* the output of Ghidra decompiler. It works on top of Ghidra-decompiled source code output; it tries to recover *only those* functions missed by Ghidra, and effectively improves upon it, and outperforms all existing approaches. Details are presented in chapter 2.

1.2. Fixing Student Program

Syntax errors are easy to make, and will cause compiles to fail. Novices find syntax errors challenging [**182**]. Studies have documented the challenges faced by novices in various languages [**93**, **116**, **141**]. Novices make a wide range of syntax mistakes [**93**], some of which are quite subtle; time that might otherwise be spent on useful pedagogy on problem-solving and logic is spent helping students deal with such errors. Unfortunately the error messages provided by compilers are often not helpful; novices struggle to interpret the messages, and sometimes even experts do! [**114**]. Consider for example, the real program example in figure 1.2, where a novice student just replaced a "*" with an "x" on line 8. None of the big 3 IDEs (VSCode, IntelliJ, or Eclipse) provide a direct diagnostic for this very understandable error. A lot of time can be spent on such errors [**55**], and researchers have called out for more attention to help novices [**114**] deal with errors, specifically syntax errors. While *semantic errors* (bug-patching) have received quite some attention, syntax errors have attracted less interest.

```
1 import java.util.Scanner;
2 public class Multiplication
3 {
4   public static void main(String[] args){
5     Scanner sc = new Scanner(System.in);
6     int a = sc.nextInt();
7     int b = sc.nextInt();
8     int res = a x b;
9     System.out.println("The result is: " + res);
10  }
11 }
```

FIGURE 1.2. Incorrect student code sample

Existing approaches have struggle to deal with programs longer than 2-300 tokens, and often fail to make good use of language-specific compiler errors [**5**, **71**, **214**] In addition they have not adequately exploited the tremendous capacity of current DL models to learn (in a self-supervised fashion) the statistics of *very large* amounts of unlabeled sequential data. Modern pre-training approaches such as RoBERTa can ingest vast corpora of sequential data (*e.g.,* a billion tokens from GitHub-hosted code) and learn the patterns of syntax, identifier usage patterns, arithmetic

expressions, method call patterns *etc.* These patterns are automatically learned and represented as high-dimensional vector embeddings of tokens, without requiring any human effort to label the data. These embeddings, however, have been shown to substantially improve performance when used as pre-set embeddings in other networks that can be "fine-tuned" with smaller amounts of human-labeled data.

In our research, by using the diagnostics from a compiler, and exploiting the ability to pre-train embeddings with high capacity RoBERTa model, we build a tool, SynShine, which improves substantially on the state-of-the-art in automated syntax repair in Java. Details are presented in chapter 3.

1.3. Multilingual Training

To motivate multilingual training, we begin by introducing the *code summarization* task, which we use to motivate multilingual training.

Developers often rely heavily on comments, to gain a quick (even if approximate) understanding of the specification and design of code they are working on. An actual example of a comment is shown in Figure 1.3. Such comments help a developer gain a quick mental preview of *what* the proximate code does, and *how* it might go about it; this helps the developer know what to look for in the code. Knowing that such comments are useful to others (or even later to oneself) incentivizes developers to create comments that explain the code; however the resulting redundancy (*viz.,* code that does something, and some nearby English text that describes just what the code does), with the same concept expressed in two languages results in a bit of extra work for the original coder. This extra work, of creating aligned comments explaining the code, can be fruitfully viewed [**67**] as a task related to *natural language translation* (NLT) (*e.g.,* translating English to German). The mature & powerful technology of NLT becomes applicable for comment synthesis; ML approaches developed for the former can be used for the latter. An effective comment synthesizer could help developers: by saving them the trouble of writing comments; and perhaps even be used on-demand in the IDE to create descriptions of selected bits of code.

Comment synthesis is now an active research area, including many projects such as CodeNN [**91**], DeepCom [**83**], Astattgru [**122**], CodeBERT [**64**], Rencos [**223**], SecNN [**126**], PLBART [**1**], CoTexT [**154**], ProphetNet-X [**157**], NCS [**2**], Code2seq [**15**], Re^{2}Com [**206**], and many more [**66**, **76**, **77**, **84**, **85**, **120**, **121**, **124**, **138**, **139**, **197**, **200**, **202**, **205**, **213**, **221**]. All these approaches rely on datasets of aligned code-comment pairs. Typically, these datasets are then used to train complex deep learning models to model a probabilistic distribution of the form $p(comments \mid code)$; one can sample from these (usually generative) models to create candidate comments for a given a piece of code. Given a dataset of code-comment pairs in a specific language, *e.g.,* Java, or Python, or PHP, or Ruby, one can train models to translate code in *that* language to comments. The quality of the translation will depend largely upon the inductive power of the model, and quality and diversity of the code-comment dataset.

```
1 //Returns the text content of
2 //this node and its descendants.
3 public String getTextContent() {
4     StringBuilder sb=new StringBuilder(getChildNodesCount()+1);
5     appendTextContent(sb);
6     return sb.toString();
7 }
```

FIGURE 1.3. *Example for code comment generation task*

Of late, given the power of GPUs, and the capacity of the models, the limitations largely arise from dataset quality and diversity, especially in languages for which limited, or rather specialized data is available. For instance, CodeXGLUE [**134**] dataset consists of six languages (*i.e.*, Ruby, Java, JavaScript, Go, Php, Python). Most languages have well over 100,000 training examples, covering a wide set of application domains. Some languages, particularly Ruby and Javascript, have far fewer examples, and cover a narrower range of application domains. As a result, state-of-the-art models perform less well for these two languages. This is a well-known problem for natural language translation: while training data for language pairs like $English \leftrightarrow French$ is abundant, resources may be lacking for less-used languages like *Quechua* or *Badaga*. In such cases, a common technique is adapt ML models to learn useful statistics from abundant data in other, perhaps related languages [**144**]. This works well when languages often have similar grammars, and share common word etymologies.

We propose an analogous approach to improve the diversity and quality of training data for software-engineering tasks, exploiting an interesting property of source code that human beings write. It's generally agreed that variable names help code comprehension [**118**]. Developers know this, and typically choose descriptive variable names (reflective of code logic and purpose) regardless of the language they are coding in. Thus, one could expect that developers coding *the same functionality, using similar algorithms, even in different languages, will use similar variable names.* This suggests that machine-learning approaches could sometimes leverage corpora in different programming languages. This work a) shows that this expectation actually has a sound empirical basis, and then b) demonstrates that this approach in fact works not just for code summarization, but also for several other tasks. Technical details are presented in full in a later, in chapter 4.

1.4. Towards Understanding What Code Language Models Learned

Pre-trained language models (PLMs) such as BERT [**56**], GPT-3 [**36**], and PaLM [**47**] are powerful processors of natural language, exhibiting prowess at various tasks, such as question answering and joke explanations [**36**, **47**, **56**]. Their skill is amplified with just a few examples (in a few shot setting, or even natural language instructions without labeled data [**148**]). PLMs also appear to be surprisingly adept at transfer learning, despite relatively simple pre-training objectives. Previous research aimed at understanding the linguistic capabilities of PLMs shows that they encode syntactic, semantic, and world knowledge [**168**]. However, while some authors suggest that PLMs understand language and learn meaning, others have argued that they do not in fact understand the meaning of natural language [**27**].

This question (concerning semantics captured by PLMs for code) arises, because, as with natural language tasks, neural *code* language models perform quite well on code-related tasks: code summarization, code retrieval, code generation, defect detection, and others [**43**, **64**, **68**, **102**]. Indeed, these models achieved state-of-the-art in most software-engineering related tasks [**133**] and some are incorporated into the most widely-used programming tools such as Github CoPilot[2]. Given their impressive performance, it's natural to wonder how much PLMs understand about code. We

[2] https://github.blog/2022-03-29-github-copilot-now-available-for-visual-studio-2022/

examine specifically whether a PLM's apparent "understanding" of code is just capturing distributions of lexical and syntactic token frequencies and co-occurrence patterns, or if PLMs are actually capturing distributions at a deeper level, *viz.,* of the computational semantics of code: thus in some sense they "know" the meaning of the code, not just superficial form. Our results provide some evidence suggesting that they are. This is somewhat surprising, especially considering some recent results suggesting that language models respond to "knowledge probes" in ways very sensitive to changes in form [**226**].

Our approach is largely automated, inspired by concepts *Metamorphic Testing*. Most of the current PLMs are pre-trained to reconstruct partially-obscured programs, *in the "original, natural" forms that humans wrote them*. However, because of the formal semantics of programs, there are different syntactic forms of the *very same* computational meaning. If PLMs are robustly learning distributions over computational *semantics*, rather than *lexical* or *syntactic* patterns, then they should also be able to correctly reconstruct other forms of programs that *have the same meaning*. Details are presented in Chapter 5.

1.5. Automatic Semantic Augmentation of Language Model Prompts

Large language models (LLMs) often outperform smaller, custom-trained models on several tasks, especially when prompted with a few shots or examples. LLMs are pre-trained on a masking or de-noising task whereby a vast amount of labeled data naturally exists. LLMs exhibit surprising emergent behaviour as their training data and number of parameters are scaled up. They do well at many important tasks; so well that it is unclear whether sufficient task-specific data can be gathered to train a customized model to achieve the few shot (or even zero-shot) performance of modern LLMs. LLMs are ushering in a new era, where prompt engineering, to carefully condition the input to an LLM to tailor its massive, but generic capacity, to specific tasks, will become a new style of programming, placing new demands on software engineers.

We propose *Automatic Semantic Augmentation of Prompts* (ASAP), a new methodology for constructing prompts for software engineering tasks. ASAP methodology rests on an analogy: an effective prompt an LLM for a task is similar to what a developer thinks about when manually performing that task. In other words, we posit that prompting an LLM with the questions a

developer asks about the code or the syntactic and semantic facts they hold in mind when manually performing a task will increase the LLM's performance on that task. We illustrate this methodology on code summarization. This task takes code, usually a function, and summarizes it using natural language; such summaries can support code understanding to facilitate requirements traceability and maintenance. ASAP aims to augment prompts using semantic analysis. Motivated by the observation that developers make use of properties of code such as parameter names, local variable names, methods called, and data-flow, we propose augmenting the prompt with semantic facts automatically extracted from the source code. These facts are added to the few-shots in the prompt, along with the desired comment output, thus providing the language model with some selected — via *BM25* [**166**] — relevant, illustrations of how these extracted facts might help construct a good summary. Finally, the model is provided with the target code and facts extracted therefrom, and asked to emit a summary. Concretely, these facts include the fully qualified name of function, the parameter names, and its data flow graph. These facts are presented to the LLM as separate, identified, fields within the few-shot examples. We evaluated the benefits of this approach on the high-quality (carefully de-duplicated, multi-project) CodeSearchNet dataset. Details are presented in chapter 6.

Finally, Chapter 7 concludes the dissertation with some future directions.

CHAPTER 2

Learning to Find Usages of Library Functions in Optimized Binaries

Much software, whether beneficent or malevolent, is distributed only as binaries, sans source code. Absent source code, understanding binaries' behavior can be quite challenging, especially when compiled under higher levels of compiler optimization. These optimizations can transform comprehensible, "natural" source constructions into something entirely unrecognizable. Reverse engineering binaries, especially those suspected of being malevolent or guilty of intellectual property theft, are important and time-consuming tasks. There is a great deal of interest in tools to "decompile" binaries back into more natural source code to aid reverse engineering. Decompilation involves several desirable steps, including recreating source-language constructions, variable names, and perhaps even comments. One central step in creating binaries is optimizing function calls, using steps such as inlining. Recovering these (possibly inlined) function calls from optimized binaries is an essential task that most state-of-the-art decompiler tools try to do but do not perform very well. In this chapter, we evaluate a supervised learning approach to the problem of recovering optimized function calls. We leverage open-source software and develop an automated labeling scheme to generate a reasonably large dataset of binaries labeled with actual function usages. We augment this large but limited labeled dataset with a pre-training step, which learns the decompiled code statistics from a much larger unlabeled dataset. Thus augmented, our learned labeling model can be combined with an existing decompilation tool, Ghidra, to achieve substantially improved performance in function call recovery, especially at higher levels of optimization.

2.1. Background

2.1.1. Importance of Function Calls in Binary Comprehension. Eisenbarth *et al.* [**60**] and Schultz *et al.* [**173**] argue that identifying library function calls made within a binary are key

to comprehending its behavior. Substantial prior research in the field of binary analysis has focused on this problem.

Much of the effort to understand binaries is to identify malware. Schultz *et al.* [**173**] use the calls made to Windows system APIs to understand if a binary has malicious intentions. Ye *et al.* [**220**] found that reverse engineers can identify malware in a binary based on the co-occurrence of six calls made to a specific kernel API.

A barrier to static analysis techniques is that sometimes binaries can be optimized and/or obfuscated. To overcome this, researchers have used dynamic analysis to understand the APIs being accessed by a binary. Hunt and Brubacher [**88**] and Willems *et al.* [**210**] attempt to detect calls made to Windows system APIs by instrumenting the system libraries. Bayer *et al.* [**23**] and Song *et al.* [**181**] emulate the Windows runtime and recover the Windows system API calls. In comparison to static analysis, dynamic analysis is limited by test set coverage, as well as by dynamic cloaking (malware could disguise its behavior when it knows it being surveilled, *e.g.,* in a VM).

Given how important library/API calls are to reverse engineers' understanding of the semantics of a binary, it is pivotal that these calls are recovered by disassembler and decompiler. However, optimizing compilers can inline many library calls, thereby making them hard to recover, even by state-of-the-art tooling; improving library function recovery is an important problem.

2.1.2. Disassembler vs Decompiler. Binaries are created in two stages: (1) source code is pre-processed and compiled into machine code and (2) the machine code is linked with all supporting code such as libraries and system calls to create the executable binary. Similarly, the process of reverse engineering of a binary comprises of two stages: (1) the binary is "disassembled" into assembler code and (2) the assembler code is converted into a higher-order representation which is close to the original source code.

Disassemblers such as Ghidra, Binary Ninja, IDA Pro, and gdb perform the first stage of reverse engineering. Since the machine instructions in a binary generally have a one-to-one mapping with the assembly instructions for most platforms, disassembly *per se* is relatively straightforward.

Next, *decompilers* such as Ghidra, Hex-rays, and Snowman transform the machine code produced by the disassembler into a more legible representation referred to as pseudo-C code. Decompilers produce code that is ostensibly more compact and readable than that produced by a

disassembler. They also often recover the original function boundaries (*i.e.*, where function *bodies* start and end), control flow structure, and primitive type information. This makes decompilers a better tool for a reverse engineer as the effort and knowledge required to understand pseudo-C code is less than that of reading assembler code.

2.1.3. Library Function Recovery. Given its importance, we focus on the recovery of function calls from binaries. The two main reverse engineering tools - Hex-rays IDA Pro (commercially available with a license cost of approximately $10,000[1]) and Ghidra (open source and maintained by the National Security Agency) - have dedicated solutions targeted just for recovering function calls. The developers of Hexrays acknowledge the importance of function recovery by stating that: "Sometimes, the knowledge of the class of a library function can considerably ease the analysis of a program. This knowledge might be extremely helpful in discarding useless information." [**24**]

Both tools can identify function calls. They maintain a database of function signatures at the byte level (assembler code). They recover the function by checking each sequence or block of operations in the disassembled code against the signatures in the database.

We observe that majority of the previous research work in this field is based on call graph matching which has been designed to be robust to statement interleaving due to compiled optimization. These approaches are static in nature and try to go beyond the offerings of Ghidra and Hex-rays.

Qiu *et al.* [**159**, **160**] implement a static approach to recover inlined library functions by extracting execution dependence graphs for a library function and then matching this in decompiled code to recover. This work reports deal with inlined functions in optimized binaries, however, the evaluation lacks a performance breakdown by optimization level. Furthermore, only precision numbers are reported on a small subset of inlined string library functions, and the overall performance is not compared to Ghidra or Hex-Rays.

BinShape by Shirani *et al.* [**179**] also uses graph features for function identification. However, they do not assess the efficacy of their approach against inlined functions. "impact of inlined

[1]Estimate based on the cost of base IDA Pro disassembler license and the cost adding three platform-specific Hexrays decompilers

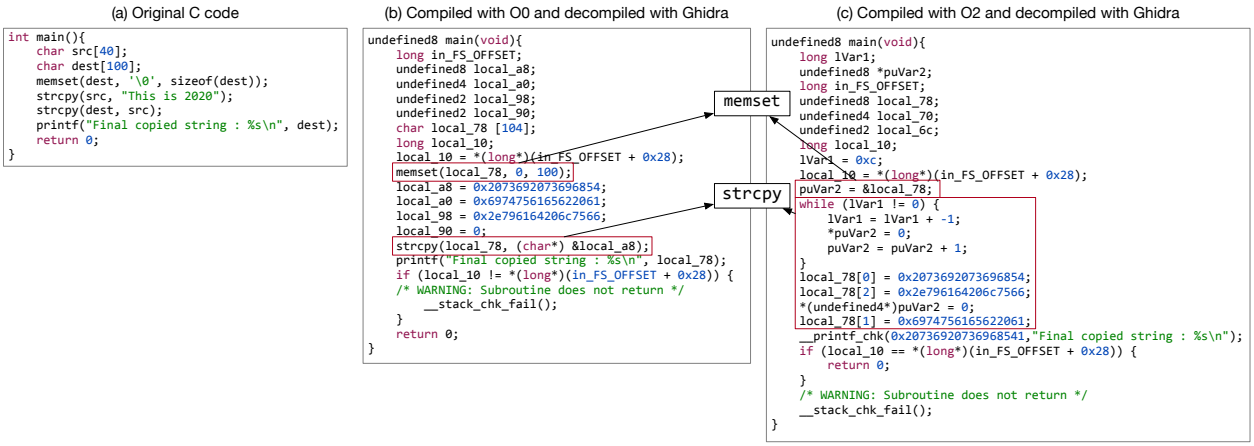


FIGURE 2.1. Comparison between original source code (a), Ghidra output for compilation with O0 (b) and Ghidra output for compilation with O2 (c)

functions" were not scrutinized. We are the first to attempt this task with a neural model and compare this to the SOTA tools such as Ghidra and HexRays.

There have been a couple of neural approaches to recovering function names but not inlined functions. He *et al.* [**78**] present a tool called Debin that is based on a combination of a decision-tree-based classification algorithm and a probabilistic graph model that attempts to recover function invocations and other symbols in obfuscated code. David *et al.* [**54**] encode the control flow graphs of invocation sites and try to recover the correct invocation name using LSTM's and Transformers. Neither approach explicitly deals with inlined library functions nor present any results broken down by optimization level.

The function recovery in these tools has a major flaw: they aren't good at recovering functions from binaries that have been compiled with optimizations. In C and C++, there are six optimization levels (in increasing order of complexity: O0, O1, Os, O2, O3, and, Of). Code compiled with O0 is the most basic: compilation is fast, and no optimization is done. Starting from O1, more and more optimizations are introduced, and the code structure changes substantially. At Of, the most optimized level, the compiler does not even guarantee correctness. Hex-rays IDA Pro and Ghidra work better with code that has been compiled using the O0 or O1 optimization levels, since the code structure is largely preserved.

In the toy example seen in Figure 2.1(a), we see that the source code of a file is written in C. The function depicted invokes `memset` and then `strcpy` (twice). When we compile this file with no optimizations (the O0 flag) and then decompile it using Ghidra (output seen in Figure 2.1(b)), we see that the decompiler can recover called functions and can create a generally good representation of the original C code. Note, however, that it "fuses" the chained string copy invocations. When we compile with a higher optimization level such as O2 and then decompile the file, we see the result in Figure 2.1(c). The performance of the decompiler degrades, as the binary gets more optimized: some library function uses are no longer recovered. In the figure, we highlight the parts of the function relating to the library function calls, whose implementation has been inlined. In this example, we do see that three other function calls are recovered, however, this could be due to the fact that they were never inlined or that Ghidra does a good job of recovering them even if they were inlined.

We want to clarify that like [**20**, **152**, **178**, **198**] we do not target the function boundary identification task. Ghidra & Hexrays already do this at 90%+ accuracy. They do much worse at the recovery of inlined library functions; This task is a challenge for the heuristic method used by Ghidra & Hexrays, especially at higher optimization levels, as acknowledged by the developers [**25**]; by leveraging powerful neural models (explained in Section 2.3), FUNCRE can improve these tools.

A decompiler can recover most of the semantics of the original code, however, it has a hard time recovering variable names, struct types, exact data flow, code comments, and inlined library functions. Most of this information is lost in the compilation - decompilation loop, *e.g.*, in Figure 2.1 the decompiler adds a lot of new local variables each with an ambiguous name.

State-of-the-art approaches to improving decompilation employ machine learning (ML) techniques. Katz *et al.* [**111**] propose to decompile disassembled code by using a Neural Machine Translation (NMT) model. Their approach currently works at the statement level and seeks to recover natural C code for each block of disassembled code. Lacomis *et al.* [**115**] use an encoder-decoder model to recover variable names. Their approach only targets code compiled with O0 and not on higher optimizations.

We see that ML approaches to improving decompilation are limited. We hypothesize that an ML-based approach will work well for the task of library function recovery because ML can detect patterns in highly unstructured data.

2.2. Approach/Methodology

2.2.1. Research Questions. Our approach to improving library function recovery builds on top of the pseudo-C code produced by the Ghidra decompiler. Using large volumes of source-available projects and some careful, automated instrumentation, we develop a *supervised learning* approach to find library functions that other tools are unable to find. Our first RQ considers the effectiveness of our approach *i.e.*, how effective is FUNCRE at recovering library function invocations not recovered by Ghidra.

RQ1a: *How effective is our supervised learning-based approach in recovering library function usage?*

In Natural Language Processing (NLP), it is now well-established that pre-training a high-capacity model using self-supervision for an artificial task (*e.g.*, predicting a deleted token, or the following sentence) improves performance on practical tasks like question-answering. Pre-training forces the layers of the model to learn position-dependent embeddings of tokens that efficiently capture the statistics of token co-occurrence in very large corpora. These embeddings are a very useful, transferable representation of a token and its context [**56**, **129**] which substantially improves performance on other tasks. By using them as an initial embedding within a (possibly different) task-specific network and "fine-tuning" using data labeled specifically for that task, much higher performance can be achieved, even if the task-specific labeled data is limited. The benefits of this approach have also been reported for code [**64**, **103**]. We examine whether pre-training over large corpora of decompiled pseudo-C can be helpful for our task of recovering library function invocations not recovered by Ghidra.

RQ1b: *How much does pre-training with RoBERTa help with library function usage recovery?*

C and C++ binaries can be compiled with a variety of optimizations (see Section 2.1.3). Most disassemblers and decompilers can handle code with no optimizations. In line with that, past research that uses a deep learning (DL) model also targets code compiled with no optimizations.

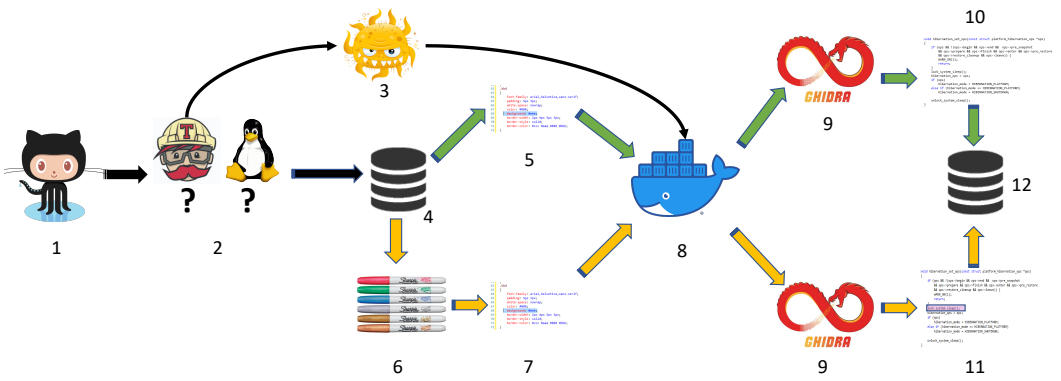


FIGURE 2.2. Training data pipeline. We mine projects from GitHub (1); after filtering (2) the ones enabled for Travis, and certain Operating systems, the projects are gathered in a source dataset (4). We then adapt the publicly available BugSwarm toolset (3) to mine Docker containers (8) for building. We indelibly instrument (6) the library function invocations in the source to get marked source code (7). The raw (5) and marked (7) sources are built using the Docker containers (8); we then use Ghidra (9) to decompile matched pairs (10,11) of marked and unmarked decompiled sources, which are gathered into our labeled dataset (12).

However, in our work, we target higher optimization levels as well. We assess the performance of our model on five optimization levels:

RQ2: *How does optimization level affect the performance of* FUNCRE*?*

With machine-learning approaches, the training data can strongly influence the test results. The model might perform better on library functions more prevalent in training data.

RQ3: *How does the popularity of library methods influence test results?*

Finally, we assess whether our model outperforms current tools when it comes to retrieving library functions in decompiled code:

RQ4: *How does* FUNCRE *perform in relation to state-of-the-art approaches?*

A key requirement for using supervised machine learning for library function recovery is the creation of a curated, labeled dataset where the occurrence of in-lined functions within decompiled binaries is labeled. There is currently no such dataset of labeled decompiled C/C++ binaries, and we have had to create our own. This presented several challenges.

(1) *Large scale, diverse data.* We need a broadly representative, large dataset that captures relevant statistics of current coding styles, library/API usages, compiler settings, and platforms.

(2) *Reproducible Builds.* To create binaries with *labeled* inlined library functions we need to suitably instrument the source to insert labels, and then reproduce the build procedures of a large, diverse set of projects. Build procedures are notoriously brittle, with many tricky dependencies, and so challenging to reproduce [**189**].

(3) *Indelible labels.* Because optimizing compilers make substantial changes to the logic of the code, our approach to creating binaries where the original inlined library functions could be labeled in a way that endures after optimization & decompilation is a tricky business.

We employ a multi-stage project selection and build process to meet these challenges (an overview of which can be seen in Figure 2.2) as elucidated below:

Large-scale, Diverse data: The focus of this work is to recover library functions from C-based binaries. Since modern deep-learning models are "data-hungry", we need the largest possible corpus of built binaries aligned with its original C source code. We sourced data from GitHub. Our selection criteria for projects is as follows:

(1) *Projects under active development.* We determine a project's activity by checking for commits in the previous six months (as of April 2020). This helps ensure that selected projects are representative of current coding styles and API usages.

(2) *Projects using Travis as their CI build platform.* We select those with public build histories and logs (hosted on `travis.org`) so that we can replicate the builds.

(3) *Projects with available Build Containers.* We filter out projects that declare in their Travis configuration (.travis.yml file) that their platform requirement is either Mac OS/X or Linux version 12.04. Travis does not provide open-source Docker containers for either build platform, thus making a build irreproducible.

Our initial selection comprised 10,000 actively developed C-based projects. After filtering for Travis-based projects and then on their build distribution, we are left with 2,634 projects.

Reproducible Builds: Successfully re-building projects at scale requires the downloading of each project's dependencies and ensuring that the correct build platform is used. This is a frustrating, failure-prone process under the best of circumstances. These problems are exacerbated when building C projects as there is no standard dependency management system comparable to those in languages such as Java (Maven or Gradle), Python (pip) and, C# (NuGet).

All 2,634 projects in our filtered set of GitHub-based C projects use Travis for continuous integration and require one of three Linux distributions: Trusty (14.04), Xenial (16.04), or Bionic (18.04). Travis CI builds each project in a clean docker container: it first installs all required dependencies and then invokes build and test scripts. We aim to recreate this process.

Fortunately, we were able to leverage the open-source BugSwarm [**189**] infrastructure. BugSwarm was originally developed to reproduce buildable pairs of buggy and fixed versions of large, real-world systems. To ensure reliable builds and reproducible failures, the BugSwarm pipeline builds pairs of commits and tests them five times. For our purposes, we do not need pairs; we just need reproducible, test-passing (not failing), singleton builds. BugSwarm is able to identify the right Docker container that a project uses, download the project into the container, install dependencies and build the project using its scripts and the Travis configuration of the project. We only need this part of the pipeline that can build just the latest commit of the code. We downloaded the source-available BugSwarm [**38**] project and adapted it for our purposes. First, BugSwarm currently does not support Travis builds for languages other than Java and Python. We augment BugSwarm's capability to deal with C-based projects. Second, we refactored the BugSwarm code to retain only a single, buildable version from the latest version of active projects. This adapted version of BugSwarm called "BuildSwarm " (which we will make available upon publication of this work [**31**]) works as follows.

(1) For each project, we use the Travis API, to download a list of public Travis builds; along with each build, we also download its details, such as build configuration, date of the build, and associated jobs for the build.

(2) From this list of builds, we select the latest passing build. Each build might have more than one job associated with it [**58**] For this build, we select the first job that suits our criteria (1) the job fits our OS criteria (see above), (2) the job does not require Docker as

a service (some projects require child Docker containers for testing, a scenario we cannot reproduce) to build the project and (3) the job needs either a `gcc` or `clang` compiler.

(3) For the selected job, we create the Travis build script that can replicate the entire build procedure, using the Travis build utility [**48**].

(4) From the downloaded log for the job, we parse the Docker image that was used by Travis. Travis releases most of their docker images on docker hub [**49**]. We use the same image as our base image to build the project and add the build script to it by composing a docker image.

(5) Once this docker image is built, we run the docker build script (generated earlier) on the project inside a docker container. This build script downloads the dependencies builds the code to produce binaries.

(6) If the project builds successfully in the container, we release the docker image to Dockerhub [**30**], and retain that image tag so that the image can be reused; we also collect the pair of the C source file and its object file.

Disassembling and decompiling a binary For each project that we can re-build, we need to decompile its binary *i.e.*, convert the executable into a form of pseudo-C code. Section 2.1.2 explains the process of disassembling and decompiling the binary to recover the pseudo-C code.

The two main tools for disassembling and decompiling a binary are Ghidra and Hexrays IDA Pro. We select Ghidra as our base tool, as it is open source and is freely available; however, we also baseline an evaluation set against both tools, for the specific task of identifying inlined library functions.

Ghidra can disassemble and decompile an executable (.exe file). This entails separating the executable into smaller object files (.o files). Each of these object files is then disassembled by delinking the libraries that they depend on, and then the resulting assembler code is decompiled. In our case, we directly operate on the object files and not on the full executable. This is because we have a one-to-one mapping between the object file and its corresponding C source file. This results in us creating a dataset with source code, binary code, and decompiled code triplets.

Indelible Labels: Our machine-learner must learn to associate patterns within the decompiled code with specific inlined library functions. To provide a supervisory signal for the learner, we must

identify and *locate* a in-lined functions within the Pseudo C code produced by the decompiler; this is non-trivial. It's difficult to align the original C source code with the decompiled Pseudo C, since optimization, followed by the decompilation process, can mangle the original code beyond recognition. Thus, recovering a one-to-one mapping between the original code and the decompiled code is virtually impossible (especially when compiled using higher optimization levels).

To create our dataset of decompiled code with labeled locations for inlined library functions, we need to inject some form of *robust label* into the original C source, that would survive despite optimization transformations and be inserted into the binary; this could then be recovered by a decompiler. We refer to this system of annotating inlined library functions in the binary as *indelible labeling*, or "marking" for short.

The process of marking starts by injecting a marker header in each function in a C project and each function from the libraries used by the project. For our purposes, we wish to train a learner to learn to identify *just* those functions *not* identified by current decompilers. To find these, we first compile and then decompile source files. For those inlined library functions which are *not recovered* by the decompiler, we must insert a marker that indicates the name of the function and its location within the decompiled Pseudo C code. The marking process must meet several requirements: the injected marker must not interfere with the compilation and decompilation process (no errors should be triggered). Second, there must be no inadvertent changes to the final compiled binary that would differentiate the marked binary from the unmarked: if the resulting decompiled marked Pseudo C differs too much from the original Pseudo C, the training signal for our learner would be confused, and the marked inline library function would not be reliably recoverable by the learner. Third, the injected marker must be resistant to change or removal by the compiler's optimization process. For example, if we were to insert a marker, as a code fragment, whose results were not used elsewhere in the code, for example, something naive like:

```
char *marker1 = "function_inlined:  printf()";
```

then the compiler might, for example, find that `marker1` is not used elsewhere, and just simply remove the marking statement, thus robbing us of a data label. We tried several approaches to this problem:

(1) `printf`: Injecting a statement that prints the name of the function being inlined. We found that a `printf` statement could at times change the structure of the Ghidra output. For lower optimization levels, the code does not change much; however, for higher optimization levels, the nature of control structures can change *e.g.*, a `while` loop is replaced with a `do while` loop.

(2) `puts`: Similar to `printf` the `puts` can print details about the inlined function. While the distortion of the decompiled binary is less than that of `printf`, we do notice that the ordering of statements can be changed, especially for longer function bodies.

(3) `Global array marker`: We can inject a global character pointer array (String array) in a source code file and assign it to the array for each function call. Since the array is global, the compiler will not discard it since modifying or discarding such an assignment may change the program semantics.

```
bVar12 = 0;
lVar9 = 0x12;
local_30 = *(ulong*) (in_FS_OFFSET + 0x28);

psVar10 = (sigset_t*) (&local_1f8 + 8);
while (lVar9 != ) {
    lVar9 = lVar9 + -1;
    psVar10->_val[0] = 0;
    psVar10 = (sigset_t *)(psVar10 > _val + 1);
}
```

(a) Without marker

```
bVar12 = 0;
lVar9 = 0x12;
local_30 = *(ulong*) (in_FS_OFFSET + 0x28);
marker21._11 = "function inlined: memset";
psVar10 = (sigset_t*) (&local_1f8 + 8);
while (lVar9 != ) {
    lVar9 = lVar9 + -1;
    psVar10->_val[0] = 0;
    psVar10 = (sigset_t *)(psVar10 > _val + 1);
}
```

(b) With marker

FIGURE 2.3. Marker survives -O2 optimization level without inducing any change in the code

Of these approaches, we chose global arrays to inject markers (Figure 2.3 depicts one example of an injected marker in decompiled code) in the source code. In comparison to the `printf` and `puts` approaches, the decompiled code obtained from Ghidra is not distorted. This might be due to Ghidra having a harder time in recovering library function calls in the correct position as opposed to array assignment. Furthermore, this tactic ensures that the compiler does not optimize the array access by vectorizing it, which would be the case for linear assignment. The global array we inject is a constant character pointer array of size 2000. We declare a global array and assign each marker

to a different position of the array. We note that the *actual value* in the array is not important for labeling; it's the assignment statement itself that constitutes the label.

In each file, we inject a uniquely identified global array, and this helps avoid a compilation conflict. This is necessary because, during compilation, the compiler merges different files (*e.g.*, header files merged into the declaring C file), which might result inadvertently inserting multiple declarations of the same array in one file. For each function call, we assign a marker to a unique position of the array with the name of the function as seen in Figure 2.3.

If we mark all function calls, we might mark some recovered by Ghidra. Since the learner does not need to recover these, we don't mark them in the code. To remove these markers, for each function definition in the decompiled code, we compare the decompiled function definition bodies with their respective function definition in the original C code. In some cases, function calls from the original C code that are inlined during compilation might be found by the decompiler and indicated as such in the decompiled code. For those function calls that Ghidra recognizes, we remove the marker from the decompiled code; for the rest, we leave the marker as they are an indication of which function call has been inlined and where it has been inlined.

Identifying target functions: For this research, we would like to design an approach that is global *i.e.*, that works on every function that has been inlined. However, for a deep learning-based approach to work, the model has to see one or more examples of an inlined function at training time, allowing it to learn an appropriate representation of each inlined function.

Using our dataset of C projects, we select a set of library functions that could be inlined in the code. We determine the most popular library function calls made by parsing all function calls from the entire dataset of 10,000 projects. To understand which function has been called in a file, we use SrcML [**51**] to build and resolve the AST and recover the exact function binding.

After parsing all 10,000 projects, we obtain a list of the top 1,000 popularly invoked library functions that we can potentially target. From this 1,000, we filter out those that are never invoked in the 2,634 Travis-based projects, resulting in 724 potential target functions.

2.2.2. Final Dataset Details. We build and obtain binaries from 1,185 (out of 2,634) projects. Many (1,449) projects would not build. For others (726), we cannot find an exact alignment between the source code and the object files. This is because the compiler merges several source

files into a single object file, thus confusing file boundaries. In such cases, it is difficult to find the alignment between decompiled pseudo-C, and the original source code, to allow the labeling of the pseudo-C with the requisite inlined functions. However, this decompiled code is still valuable, and we keep it in our dataset for (RoBERTa) pre-training purposes (as described in the next section).

(A) Cross-file train-test file distribution

OPT-Level	Train Set	Validation Set	Test Set
O1	9	0	29
Os	2851	88	197
O2	2710	74	196
O3	2591	94	144
Of	482	14	150
Overall	8643	270	716

(B) Cross-project train-test file distribution

OPT-Level	Train Set	Validation Set	Test Set
O1	37	2	NA
Os	2840	92	195
O2	2686	91	207
O3	2517	96	215
Of	627	19	NA
Overall	8707	300	617

TABLE 2.1. File-level distribution of the dataset used to train and test FUNCRE

For the other 459 projects, we split the files into training, validation, and test sets in two different settings (file level breakdown presented in Table 2.1): (1) cross-file where files from the same project can be present in the train, test or validation set and (2) cross-project where all the files from a single project are in one of the train, validation or test sets. In the cross-project setting we do not have enough projects and files for the test set for O1 and Of and thus all evaluation in this setting is done on just three settings. In the cross-file setting, our training set consists of the bodies of 391,967 function definitions spanning 8,643 files and in the cross-project setting we have 401,923 function definitions and 8,707 files. These function bodies are labeled with markers indicating any inlined functions not recovered by Ghidra and used to construct pairs as indicated in Figure 2.2 in our dataset.

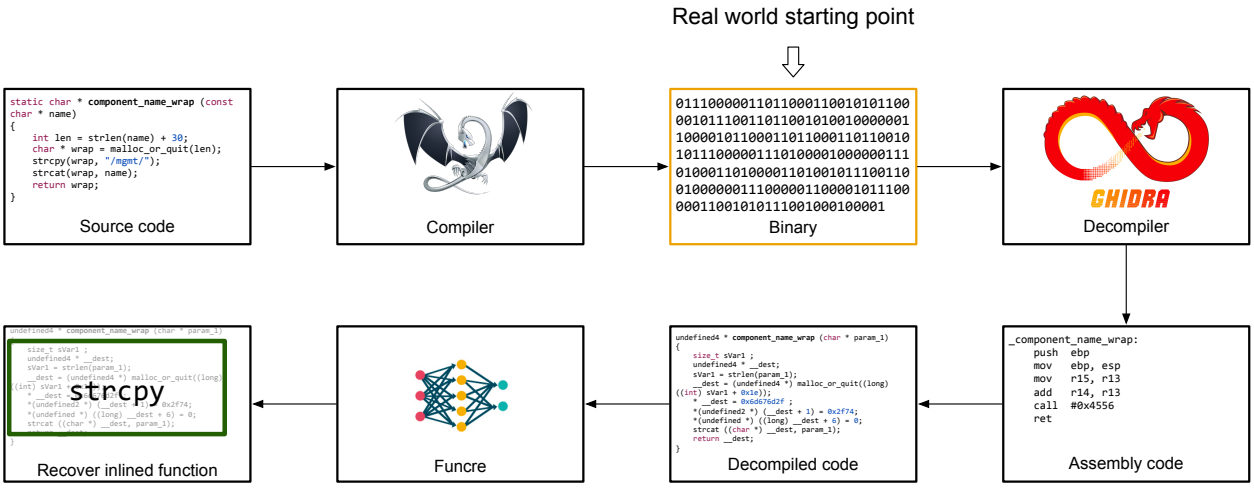


FIGURE 2.4. Working of Funcre. Funcre works on the decompiled output from Ghidra. In a real-world scenario, we start with an external binary. For training and testing purposes we create our own binaries using real-world source code obtained from GitHub.

2.3. Creating FUNCRE

The expected use-case scenario of FUNCRE is shown in Figure 2.4. To get FUNCRE working, we made several engineering decisions concerning the use of machine-learning techniques. First, we had to select a suitable deep-learning approach. Second, we had to develop an approach to train our models. Finally, we had to design an evaluation methodology to gauge the value added by FUNCRE.

2.3.1. Model Selection. We claim that the task of recovering library function invocations from decompiled pseudo C code resembles text classification in Natural Language Processing (NLP). This intuition's essence: function invocations, especially if inlined, can span multiple lines in decompiled pseudo-C code; some of these lines may contain some pattern that indicates the presence of an invoked library function. We hypothesize that such patterns of pseudo-C code, reflecting the presence of library function invocations, can be learned by a machine learning model given sufficient data and model capacity. In addition, our goal is to *build on top* of the available tools that already recover atleast some function invocations, thus providing greater value to reverse engineers.

Potential approaches: We consider two approaches which are the current state of the art in NLP: *Transformers*, and *Masked Language models with fine-tuning*.

Transformers. The Transformer [**193**] model has proven to be very effective for NLP tasks. It is a sequence-to-sequence architecture (*i.e.*, it transforms a given input sequence into another sequence) consisting of an encoder and decoder. The encoder reduces an input sequence into a high dimensional vector, which is fed to the decoder, which outputs another sequence. Older sequence-to-sequence models use a RNN (Recurrent Neural Network) for the encoder and the decoder. Rather than recurrence, Transformers use a multi-head attention architecture along with feed-forward layers. Attention is a mechanism whereby for each token in an input sequence, a model chooses another token therein to "attend to", *viz*, weight in its decision making. A transformer can attend to many tokens in the input sequence, to produce an embedding of an input sequence, using "multi-head" attention, which improves capacity and parallelism beyond RNN (including LSTM and GRU) approaches [**193**].

Masked Language Model with fine-tuning. BERT-based (Bidirectional Encoder Representations from Transformers) [**56**] masked language models (MLM) leverage self-supervised "pre-training". BERT learns a representation for an input sequence. A BERT model is pre-trained on large unlabeled corpora, using self-supervision, and then fine-tuned for a specific task using standard supervision (*viz.,* explicitly labeled data). This set-up has been shown to outperform traditional Transformer based approaches in NLP. For pre-training, two self-supervised tasks are used: first, it learns to predict masked tokens in the input sequence (typically 15% of the tokens are masked), and second, learning to predict the next sentence following an input sentence (NSP). This pre-trained model is then fine-tuned for a downstream supervised task such as sequence tagging. Currently, it is more common to use RoBERTa (A Robustly Optimized BERT Pretraining Approach) [**129**] to train a MLM. RoBERTa differs from BERT, with a few changes such as dynamic masking and not doing NSP, but achieves better performance. This setup of pre-training and fine-tuning achieves SOTA results for downstream SE tasks such as variable misuse detection and function-comment mismatch detection [**64**, **103**]. We reuse Huggingface's open-source implementation of RoBERTa [**87**].

In our setting, labeled training data (for the fine-tuning stage) is somewhat limited, because: (1) we limit the task of function-invocation recovery to only a select set of library functions (see Section 2.2.1), (2) the number of markers that we are successfully able to insert in the code and recover after the compilation - decompilation loop (see Section 2.2.1) is limited, (3) only projects

that are compiled with a higher optimization level (O2, O3, Os, Of and in rare cases O1) can contain a function invocation that is not already recovered by Ghidra (see Section 2.1.3). At lower optimization levels, the function calls are not inlined and are more easily found by Ghidra and Hexrays; we don't need to learn patterns for the invocations that are recovered already. As a result of these restrictions, we have a labeled dataset that is smaller than ideal for these powerful data-hungry transformer models. Thus, the fine-tuning approach (pre-training MLM using RoBERTa and then fine-tuning on our downstream task using the labeled dataset) is well-suited.

For pre-training, we have available a dataset with 1.2B tokens of Pseudo-C files produced by Ghidra, *sans any markers*. We omit markers here to preclude any chances of inadvertent "leaking" knowledge relevant to the final task. Pre-training does learn robust, well-generalized representations of the statistics of decompiled pseudo-C, which enables FUNCRE to quickly patterns that reflect library function invocations, from a few labeled examples. We use the standard pre-training configuration, *viz.,* "RoBERTa base". This configuration has 12 attention layers, 768 hidden dimensions, and 12 self-attention heads in each layer resulting in a model of 125M parameters. We tokenize the code by using a Byte Level BPE (Byte Pair Encoding) Tokenizer. We limit the vocabulary size to 25,000 and keep tokens with a minimum frequency of 20. We train the MLM model on two NVIDIA Titan RTX GPUs for 2.5 epochs with a batch size of 40 sequences. This pre-training process takes three days and achieved a final perplexity of 1.22 when predicting masked tokens. This corresponds to a rather low cross-entropy loss of around 0.36 bits.

This suggests that the BERT model is learning a very powerful model of token co-occurrence statistics in the Pseudo-C using the enormous (1.2B token) pretraining data. For comparison, the original RoBERTa paper (for natural language) reported a pre-training final perplexity as low as 3.68 (cross-entropy about 1.8 bits); the significantly higher perplexity for natural language is consistent with prior studies [**40**].

We end the pre-training once both the training and evaluation loss stops declining further.

Choosing a Context Window Size Before finalizing our model, we needed to address two design issues. (1) We need to determine whether the pre-trained RoBERTa model provides any advantage over simply training a state-of-the-art transformer model directly on task. (2) We need to select a context window-size (in terms of the number of lines the model needs to look at it) that can

Context Length	Models					
	RoBERTa-base			Transformer		
	Top 1 Acc. in %	**Top 5 Acc. in %**	**Top 10 Acc. in %**	**Top 1 Acc. in %**	**Top 5 Acc. in %**	**Top 10 Acc. in %**
±3	75.73	88.33	91.44	64.93	84.56	89.28
±5	80.38	91.64	95.02	71.71	90.12	94.14
+10	78.28	91.25	93.45	71.92	89.20	93.38
−10	56.43	76.64	83.17	49.00	73.17	81.38
±10	**80.41**	**92.56**	**95.21**	69.37	89.41	93.13

TABLE 2.2. Performance of RoBERTa and Transformer models on development set at different context size

capture the signature of an in-lined method. Inlined library functions may span multiple lines. If this context window is too narrow, then a pattern capturing a library function invocation (especially if inlined) may not fit in entirely. If it is too big, then it will compromise our ability to locate the method more precisely. Our marking approach indicates the start of the function; however, it does not indicate how many lines it spans. We need to determine what the size of the context window (relative to the start position) must be for a model to effectively learn the representation of a function.

Finalizing our Design: We evaluated our design choices on our validation set, using a straw-man task. We train a vanilla Transformer model end-to-end on the labeled training dataset. For the RoBERTa-based model, we reuse the pre-trained MLM mentioned earlier and then fine-tune it on our labeled dataset. We train the models by feeding them input sequences where the function invocation marker is in the middle surrounded by a context window (in terms of the number of lines) of varying sizes. This form of training does not parallel a real-world setting where the location of function invocation is typically unknown; however, to choose a model architecture, this approach is reasonable.

In Table 2.2 we see the performance of both the RoBERTa model and the Transformer model on five different context window sizes. We observe that the window size ±10 works best in the case of RoBERTa. We also notice that the context window sizes +10 and −10 do not work as well as a context window that spans both sides of a marker. This implies that the model requires both the preceding and succeeding lines of code to learn the representation of a function invocation. Both

RoBERTa and the Transformer model learn a fairly good representation of the invocations. With the Top 1 accuracy for RoBERTa reaching 80% and the Top 5 accuracy being 92.5% (both in the case of context window size ±10). Furthermore, we see that in Table 2.2 the RoBERTa model outperforms the Transformer model in every setting. This implies that by pre-training a MLM and then fine-tuning it on the task, we can achieve better performance.

We, therefore, chose the MLM architecture, and ±10 *context window size, for our more realistic training/evaluation regime.*

2.3.2. Final Training & Evaluation. Using the pre-trained MLM, now we fine-tune the MLM using the labeled dataset to create FUNCRE. In the earlier straw-man approach, which was used model selection, we assumed that the model would know the location of the library function invocation. In a real-world setting, the model must recover the invocation from the entirety of the decompiled code for a function definition, without knowledge of the specific location. Because of optimizations such as inlining and code movement, it's often not possible to know exactly where in the pseudo-C the function invocation should be located; we therefore relax the problem to locating the *calling function body* in which the invocation occurs. A correct recovery for us would therefore amount to recovering the correct invoked method, and the correct function body in which that method is invoked. Our recovery works by scanning a window over the a function body and looking for invocations within each window.

Scanning the Window: Based on the indication from the straw-man evaluation above, we employ a context window of ±10 size to both train and test our model. For function bodies that are over 20 lines long, we slide a context window forwards, one line at a time. In both training and test, each sliding window is labeled with the marker that occurs in that window. When there are multiple markers in a window, we simplify the labeling for the block by marking it with the first marker in lexical order. This line-by-line scanning does present a problem. Consider an inlined library function invocation (say `atoi`) that occurs at line 8 of a 30-line function. The corresponding marker will occur around line 8 in the first 20-line scanning window (starting at line 1 of the function) and repeat eight or more times as the 20-line scanning window moves forward, a line at a time. The learner may find an invoked function's signature in even more than eight successive windows.

Consequently, we adopt a sequential filtering heuristic to coalesce these repeating sequential labels, as described next.

Coalescing Sequential Labels: We use a simple, noise-tolerant run-length encoding heuristic to coalesce sequential blocks. This heuristic was tuned on the validation set without examining the test set.

(1) Given a sequence of predicted labels in long function, we remove sequentially inconsistent predictions, *viz*, labels that disagree with the five preceding *and* succeeding labels. This works well since in-lining is rare in shorter functions.

(2) We use run-length encoding on the sequence. Run-lengths are incremented leniently; our heuristic will treat up to 3 successive unlabeled windows as being the same as preceding label and succeeding label (if both preceding label and succeeding label are same). Thus if a, b, c are method labels and x is a "no method" label, the label sequence $aaxxaabbbbbxcxxxxcdc$ is encoded as $a^6b^5c^1c^3$. The second two x after the first two a are treated as a, so we get a total run length of 6 a; after the 5 b, the c is accepted, although it is inconsistent with the preceding b because it agrees with a following c; the d within the run of 2 c at the end is erased for being inconsistent.

(3) Next, we only retain function labels with a run-length of at least four as a true label. The above example then collapse to just a^6b^5. Finally, we divide the run-length by 20 and take the ceiling. This leaves us with just two labels, a and a following b. We collect the markers from the Ghidra output and compare the result. Finally, we add the result to the result achieved by Ghidra.

We note again that this heuristic was tuned **exclusively** *on the validation set*, to scrupulously avoid overfitting to the test set.

Data Imbalance: After dividing both the training and validation sets into windows containing 20 lines of code shifted by one line, we observe that our dataset is imbalanced: the unlabeled windows dominate. Since we have a pre-trained RoBERTa model that has learned the statistics of the unlabeled window, we re-balance the data by discarding some 65% of these blocks with no label. Even so, the no-label windows are about 80% of our training and development set. For the

fine-tuning stage, we employ the same tokenizer that was used for pre-training. We fine-tune our model over three epochs on six NVIDIA Titan RTX GPUs, taking a total of three hours.

2.4. Empirical Results

Our goal is to improve the performance of Ghidra in recovering inlined library functions. Ghidra already recovers some library functions; the combination of Ghidra with our model *should* improve performance. We begin with our evaluation metric and then dive into our results.

Evaluation Metric:

We remind the reader (as described in Section 2.3.2) a correct recovery for us is a library function invocation, together with the function body in which this invocation occurs. Thus *a single test instance is a specific library function invocation, together with the containing function; this is our target for recovery.* This task is performed by scanning candidate function bodies in 20-line blocks; we explain above Section 2.3.2 how the model decides if and what function invocations occur in these blocks. In the following we explain our evaluation criteria as it applies to the problem of *recovering library function invocations within function bodies*, *viz.* how we decide if a test instance results in a TP, FP, TN, FN, *etc*

- *Model predicts empty (*i.e.*, no invoked function) and true label is empty: this function body contains no library function invocations* we mark this as a true negative (TN). These are of limited value to the RE and extremely numerous! So we ignore these in our calculations (note that we do not report "accuracy", and do not count these in our precision calculation).
- *Model predicts 'Func1' and true label is 'Func1':* we count it as a true positive (TP). These are helpful to the RE.
- *Model predicts label 'Func1', and true label is empty:* we count a false positive (FP). In this case, our model is confused, and finds a library function invocation in the body of another function, where there isn't any such invocation. These create needless work for the RE.

- *Model predicts empty, and true label is 'Func1'* we count a false negative (FN). Our model failed to recover a library function invocation that did occur within a function body. These cases fail to provide useful information to the RE.
- *Model predicts label 'Func1' and true label 'Func2:* we score this as a false negative, FN, (for missing 'Func2'), and also a FP (for predicting 'Func1'). Our model not only failed to recover a function invocation, it also reported incorrectly that a different function was used than the actual one! These cases fail to report important information, *and* create needless work, and so is doubly penalized.

As can be seen from the above, TP+FN is the count of actual function invocations that *could* be recovered. Based on these counts for FP, FN, TP, and TN, we calculate the precision, recall, and F-score that our model achieves on the test set. Note again that we ignore the correctly labeled empties despite this being an instance of our model performing well (this lowers our precision from ∼0.90 to ∼0.60) since it is of limited value to the RE. All the evaluations are given below use these criteria.

(A) Cross-file train-test split

OPT-Level	TP	FP	FN	Prec.	Recall	F-score
O1	135	86	130	0.61	0.51	0.56
Os	752	366	867	0.67	0.46	0.55
O2	647	403	760	0.62	0.46	0.53
O3	736	437	955	0.63	0.44	0.51
Of	898	429	998	0.67	0.47	0.56
Overall	3168	1721	3710	0.64	0.46	0.54

(B) Cross-project train-test split

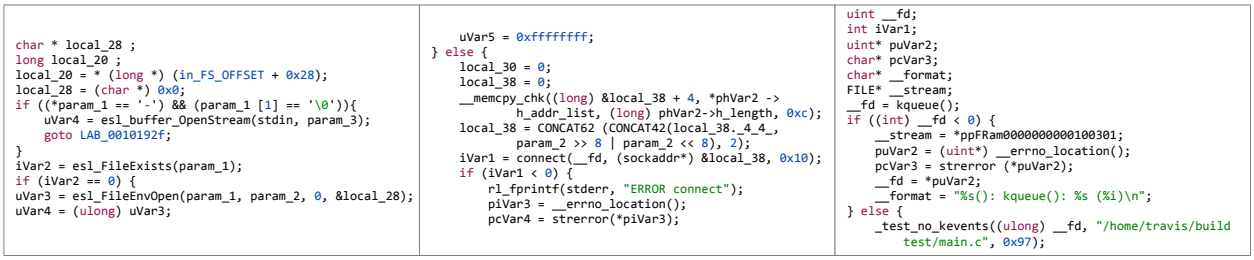
OPT-Level	TP	FP	FN	Prec.	Recall	F-score
O2	1004	417	923	0.70	0.52	0.60
O3	743	1192	1222	0.38	0.38	0.38
Os	927	641	919	0.59	0.50	0.52
Overrall	2674	2250	3064	0.54	0.46	0.50

TABLE 2.3. Performance of FUNCRE at various optimization levels

2.4.1. RQ1: Effectiveness of FUNCRE. Now, we start with the results for FUNCRE on the task of recovering library function invocations *not* recovered by Ghidra (we evaluate the recovery of *all* function invocations later). Our evaluation is based on two different dataset splits as elucidated

in Section 2.2.2. Table 2.3a presents our model's performance on the test set that has been split at file level. Overall test instances, FUNCRE achieves a precision of 0.64 and a recall of 0.46. We have more FNs than FPs, suggesting that the model errs on the side of caution *i.e.*, instead of inaccurately predicting the presence of a function, the model predicts that there is no function in place.

In Table 2.3b we present our results on a test that has been split at project level. With this test split we only have sufficient training and test data for three optimization levels - O2, O3, Os as for O1 and Of the data originates from a few projects thus rendering a project level split impossible. We observe that with this cross project split, the precision and recall at O2 increases in comparison with the cross-file split. We see that for Os we lose some precision but improve the recall and overall the F-score is slightly lower. However, for O3 we see a degradation in both precision and recall.



```
char * local_28 ;
long local_20 ;
local_20 = * (long *) (in_FS_OFFSET + 0x28);
local_28 = (char *) 0x0;
if ((*param_1 == '-') && (param_1 [1] == '\0')){
    uVar4 = esl_buffer_OpenStream(stdin, param_3);
    goto LAB_0010192f;
}
iVar2 = esl_FileExists(param_1);
if (iVar2 == 0) {
uVar3 = esl_FileEnvOpen(param_1, param_2, 0, &local_28);
uVar4 = (ulong) uVar3;
```

(a) Inlined function: `strcmp`

```
    uVar5 = 0xffffffff;
} else {
    local_30 = 0;
    local_38 = 0;
    __memcpy_chk((long) &local_38 + 4, *phVar2 ->
            h_addr_list, (long) phVar2->h_length, 0xc);
    local_38 = CONCAT62 (CONCAT42(local_38._4_4_,
            param_2 >> 8 | param_2 << 8), 2);
    iVar1 = connect(__fd, (sockaddr*) &local_38, 0x10);
    if (iVar1 < 0) {
        rl_fprintf(stderr, "ERROR connect");
        piVar3 = __errno_location();
        pcVar4 = strerror(*piVar3);
```

(b) Inlined function: `memset`

```
uint __fd;
int iVar1;
uint* puVar2;
char* pcVar3;
char* __format;
FILE* __stream;
__fd = kqueue();
if ((int) __fd < 0) {
    __stream = *ppFRam000000000100301;
    puVar2 = (uint*) __errno_location();
    pcVar3 = strerror (*puVar2);
    __fd = *puVar2;
    __format = "%s(): kqueue(): %s (%i)\n";
} else {
    _test_no_kevents((ulong) __fd, "/home/travis/build
        test/main.c", 0x97);
```

(c) Inlined function: `die`

FIGURE 2.5. Example code snippet from decompiled code containing an inlined library function

Since the data in our cross-file dataset is so imbalanced, we check if our model performs better than a random model or coin toss. We made several runs of a simulated model that uses only prior probability to predict invoked function. Not surprisingly, FUNCRE vastly outperforms a simulated model that guesses labels just based on priors: the f-score never rises above 0.003 despite hundreds of simulated runs.

Figure 2.5 shows three code snippets from our test set which contain the functions `memset`, `strcmp`, and `die` respectively. Despite the lack of obvious signs in the decompiled code of these invoked functions, our model can identify the functions correctly. This suggests that our model is learning a useful representation of the decompiled code and can recover (even inlined) function calls.

Table 2.4 presents a sample list of library functions recovered by Funcre. Several of these represent vulnerabilities, and/or malicious behavior; in general, labeling unidentified library function calls correctly, in decompiled code, represents useful information that is *simply not* otherwise available to the reverse engineer.

```
memset, fprintf, check, setjmp, match, snprintf, free,
fopen, wifexited, closesocket, xmalloc, htons, calloc,
testnext, malloc, open, localtime, wifsignaled, impossible,
fail, unlock, xstrdup, pixgetdata, verbose, validate, typeof,
getpid, strcasecmp, warnx, waitforsingleobject, getgid,
system, entercriticalsection, createevent, setsockopt, raise
crc32, leavecriticalsection, perror, chmod, report
```

TABLE 2.4. Functions recovered by FUNCRE.

Finding 1. Overall, the code representation learned by FUNCRE is powerful enough to recover 46% of library function calls, and errs on the side of caution (more FN than FP)

2.4.2. RQ2: Effect of Optimization Level. We look at the effect of optimization level on FUNCRE in only the cross-file setting as the cross-project setting has not been evaluated on two optimization levels. In Table 2.3a we examine model performance at varying optimization levels: higher optimization levels make the task harder. For the O1 level, we have reduced training data due to the low rate of function inlining that occurs; thus we see the poorest performance. Likewise, we have reduced training data for Of; however, we see that model's precision is higher in this case. This despite Of being the most complex optimization; we hypothesize that is because of inductive transference from the O3, O2, and Os, classes, where many similar optimizations may occur.

For Os, O2, and O3, the F-score ranges between 0.55 and 0.51. Additionally, in all three cases, the precision is higher than 0.60. As the optimization level increases, the precision drops slightly. However, the recall remains almost constant. This relatively stable performance suggests that the model can deal with the challenges posed by higher optimizations, where the decompiler typically struggles to create a helpful representation.

Finding 2. The performance of FUNCRE does not deteriorate significantly with the increasing complexity of compiler optimization.

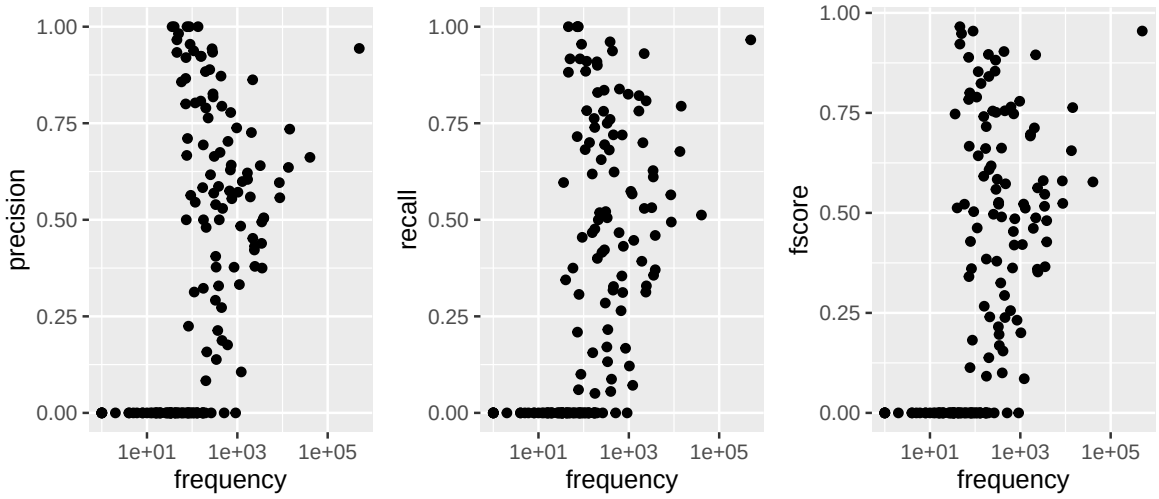


FIGURE 2.6. Does training set frequency affect model performance?

2.4.3. RQ3: Impact of the Popularity of Methods. How does performance vary with training sample frequency? Figure 2.6 plots the method frequency in the training set against precision, recall, and F-score in the cross-file setting (we omit the cross project setting here due to the relative lack of diversity of the test set). We can see that for methods that occur less than roughly 50 times in the training dataset, performance is generally quite low. These methods include `strtoi` (frequency is 1), `vprintf` (frequency is 20) and `rand` (frequency is 30). At intermediate frequencies, between 50 and 1500, performance is quite variable. There are some popular methods such as `offsetof` (frequency is 917) and `max` (frequency is 1,220) for whom the F-score remains quite low, near 0. However FUNCRE can perform well on functions such as `lseek` (frequency is 90) and `strndup` (frequency is 72) where the F-score is higher than 0.8. We conjecture that performance depends on other factors, *e.g.*, how varied the invocation (or inlined) code looks for each function.

At much higher frequencies, performance more reliably improves; methods such as `sscanf` (frequency is 2,186), `printf` (frequency is 14,437) and `assert` (frequency is 40,520) all show good performance. FUNCRE is also able to perform well on rares functions such as `lseek` (frequency is 90) and `strndup` (frequency is 72) where the F-score is higher than 0.8.

The overall Pearson correlation values of the frequency with precision is 0.14, with recall 0.16, and with F-score 0.17.

Finding 3. The performance of FUNCRE has a weak correlation with call frequency. The weak correlation arises from 3 distinct regions of performance: consistently poor performance at low frequencies, very variable in mid-ranges, and more reliably higher at higher frequencies.

(A) Cross-file train-test split

OPT. level	File Count	Total Functions	Tool	TP	FP	FN	Prec.	Recall	F-score	Unique Function Recovered
O1	29	867	Hex-Rays	456	103	407	0.81	0.53	0.64	50
			Ghidra	460	67	404	**0.87**	0.53	0.66	51
			Ghidra+FUNCRE	595	153	400	0.80	**0.59**	**0.68**	**54**
Os	197	2932	Hex-Rays	1796	1819	2665	0.50	0.40	0.44	100
			Ghidra	1632	1399	2983	0.54	0.36	0.43	89
			Ghidra+FUNCRE	2384	1765	3144	**0.57**	**0.43**	**0.49**	**108**
O2	196	1686	Hex-Rays	2234	1325	2764	0.62	0.44	0.52	131
			Ghidra	2012	934	3129	**0.68**	0.39	0.50	126
			Ghidra+FUNCRE	2659	1337	3256	0.67	**0.45**	**0.54**	**160**
O3	144	1747	Hex-Rays	2499	3330	3279	0.43	0.43	0.43	104
			Ghidra	2334	2191	3699	0.52	0.39	0.44	98
			Ghidra+FUNCRE	3070	2628	3950	**0.53**	**0.43**	**0.48**	**126**
Of	150	1579	Hex-Rays	1111	489	2840	**0.69**	0.28	0.40	78
			Ghidra	893	656	3282	0.58	0.21	0.31	75
			Ghidra+FUNCRE	1791	1085	3411	0.62	**0.34**	**0.44**	**89**

(B) Cross-project train-test split

OPT. level	File Count	Total Functions	Tool	TP	FP	FN	Prec.	Recall	F-score	Unique Function Recovered
O2	207	1250	IDA	2028	2028	3917	0.34	0.27	0.30	110
			Ghidra	1864	1593	3664	0.53	0.33	0.41	105
			Ghidra+Us	2868	2010	3633	**0.58**	**0.44**	**0.50**	**123**
O3	215	2496	IDA	1784	5031	5323	0.26	0.25	0.26	115
			Ghidra	1623	2030	3771	**0.44**	0.30	0.36	110
			Ghidra+Us	2366	3222	4346	0.42	0.35	**0.39**	**134**
Os	195	2687	IDA	2498	3967	5362	0.39	0.32	0.35	102
			Ghidra	2363	1280	3115	**0.64**	0.43	0.52	98
			Ghidra+Us	3290	1921	3242	0.63	0.50	**0.56**	**113**

TABLE 2.5. Comparison of Ghidra + FUNCRE, Ghidra and Hexrays

2.4.4. RQ4: Comparison to Existing Tools. Next, we compare our overall performance to the state-of-the-art, Ghidra, and Hex-rays. In this evaluation, we consider recovery of *all* function invocations. How well do the available tools identify function invocations, whether inlined or not? Since FUNCRE works on top of Ghidra, we augment Ghidra's results with ours to measure if (and how much) the function recovery of Ghidra is improved.

To assess the TP and FP rates for function recovery with Ghidra and Hex-rays, we compare the decompiled code against the original source code to see how many functions are recovered correctly. We run this analysis on our test set in both the cross-file and cross-project setting. Compared to Table 2.3a, we are missing the results for Hex-rays on 14 files in our test set, because the version of Hex-rays at our disposal cannot process 32-bit binaries[2], so we just omitted them from this comparison, to be generous to Hex-rays.

In Table 2.5a we see that Ghidra + FUNCRE has the best f-score for all optimizations (precision declines somewhat for O1 and Of, and marginally for O2). Our tool does not degrade the recall or F-score for Ghidra; instead, it enhances it enough to outperform Hex-rays. We also see that Ghidra + FUNCRE recovers the most library function calls, and in all cases, also the most *unique* functions: *e.g.*, at the O2 level, Ghidra + FUNCRE recovers 29 more unique functions than Hex-rays, while achieving also getting the highest F-score.

All outputs of FUNCRE, Hex-ray, and Ghidra are multisets. If one `EnterCriticalSection` and 3 `sprintf` are actually inlined, Ghidra may recover partially (e.g, one `sprintf` is recovered), and we combine those with our output. To combine outputs, we take a multiset union (which could boost both TP and FP, and reduce FN; note that both function name and count matter to measure Recall/Precision/F1). Table 2.3a reports on JUST the MARKED functions (we evaluate on the recovery of 2 sprintf and 1 EnterCriticalSection) recovered per optimization level by Funcre ALONE.

We repeat the same analysis in our cross project setting as well and present the results in Table 2.5b. We see that in contrast to the cross file setting, the precision and recall is down in all three scenarios across all three optimization levels. Despite this downturn, Ghidra + FUNCRE outperforms plain Ghidra and Hex Rays in terms of F-score in all the cases. We do notice that the

[2]A commercial license for the 32 bit version of Hexrays is available for additional purchase

precision and recall for O2 and O3 are lower than in the cross file setting, however, for Os there is an increase. Overall, in the cross project setting we see a drop in performance in comparison to the cross file setting, but even in this setting FUNCRE shows that it outperforms the competition.

We find it noteworthy that the Hex-ray's FP count is this high given that the Hex-rays developers explicitly designed their FLIRT signature system to be cautious and never introduced a false positive (see Section 2.1.3).

To investigate further, we examine a random subset of 10 function definitions containing one or more FP library calls for each of the three tools. We observe that in the majority (seven for Hex-rays, seven for Ghidra, and five out of ten for Ghidra + FUNCRE) of the cases, the FP are correctly marked as FP *i.e.*, the tool incorrectly recovers the wrong function based on a comparison with the original source. In the remaining cases, we find that the function call is transitively inlined from a function definition or macro from another file and due to the limitation in our detection strategy (these cases are near impossible to detect due to the absence of a system-level call graph) are marked as FP.

Out of 724 popularly used library functions present in the projects we target , only 365 are inlined depending on optimization level (O1 has minimal inlining). Only 168 occur in our test set, and Funcre recovers 93. Significantly, we improve 19% over Ghidra and 10% over Hex-rays; for the challenging case of O2, we improve by 22% over Hex-rays (second highest in Table 2.5a). Note that we correctly recover more inlined-functions; our recall improves over Hex-rays e.g., for Of by over 20%, finding 680 more instances of inlined-functions. While improving recall, Funcre leaves precision about the same or improved (Table 2.5a). Our FP rate *is not higher* than current tooling.

Finding 4. FUNCRE enhances the performance of Ghidra to the extent that it outperforms Hex-rays.

2.5. Threats to Validity

Generalizability. We collect code and build binaries from GitHub projects; these may not yield binaries that are typically reverse-engineered. Furthermore, we only target binaries built against three versions of Linux on an x86 architecture; performance on other platforms may vary.

Internal Validity. We mark predictions as true only when exactly matched with the expected label. However, in some cases, the decompilers recover functions such as `printf_chk` and `assert_fail` rather than `printf` and `assert`. One could argue that the prediction is correct in such cases, but we still mark it as incorrect. This impacts measured scores equally for all tools (FUNCRE, Ghidra, and Hex-rays). This is a non-trivial issue with no easy resolution: *e.g.,* not requiring an exact match also risks biases & errors.

Certain false positives in the function recovery for all three tools also originate from the fact that function definitions or macros from the same project might have been inlined into the function that we analyze. Inlined calls transitively inlined in the function under consideration might be recovered by all three tools; however, we have no way of knowing whether the recovery is a true positive. We do look at this transitive inlining for one step *i.e.*, for the first function declaration inside the same file that is inlined. However, we do not construct a system-level call graph which might adversely impact the false positive rate reported for all three tools.

Practical Applicability. When both Ghidra and Hex-rays recover a function call, they can place it in the function definition body as an actual function call and not a collection of statements. FUNCRE can only recover the list of inlined functions per function declaration body. However, as seen in Section 2.4.4 we observe that both Ghidra and Hex-rays have false positives when it comes to function recovery, furthermore, the evaluation strategy employed in this work does not know whether the location where the function is recovered is correct or whether the parameters that are passed to the function call are correct. We do recover some functions that current tools cannot; still, marking the exact position of the inlined is function is much harder because the compilation-decompilation loop can move the location of a marker. Knowing exactly which lines correspond to an inlined function is also non-trivial without a more advanced representation of the code.

2.6. Contributions

We have described an approach to improve inlined library function recovery in comparison with state-of-the-art tools (Ghidra and Hex-rays IDA Pro). Our main contributions are:

(1) We created a technique to build C-based projects on a large scale. Using this pipeline, we build and release the Docker containers for 1,185 C projects. We also created an annotated

set of real projects (first of its kind), which indicates functions inlined by compilers. Our data & tooling will be released.

(2) We show that MLM pre-training, on 1.2 billion tokens of decompiled code improves task-performance for inlined library function recovery. We will release the trained MLM for reuse in other RE tasks, such as name recovery.

(3) We improve upon Ghidra and Hex-rays on library function recovery, both in terms of f-score and unique functions recovered. This suggests that modern machine-learning methods have value for this task.

(4) There has been less attention in prior research work (on binary analysis) towards highly optimized binaries. Our work considers all optimization settings, including the highest (Of). This suggests that our research has greater relevance in broader settings than prior work.

CHAPTER 3

SynShine: improved fixing of Syntax Errors

Beginning programmers struggle with the complex grammar of modern programming languages like Java, and make lot of syntax errors. The diagnostic syntax error messages from compilers and IDEs are sometimes useful, but often the messages are cryptic and puzzling. Students could be helped, and instructors' time saved, by automated repair suggestions when dealing with syntax errors. Large samples of student errors and fixes are now available, offering the possibility of data-driven machine-learning approaches to help students fix syntax errors. Current machine-learning approaches do a reasonable job fixing syntax errors in shorter programs, but don't work as well even for moderately longer programs. We introduce SynShine, a machine-learning based tool that substantially improves on the state-of-the-art, by learning to use compiler diagnostics, employing a very large neural model that leverages unsupervised pre-training, and relying on multi-label classification rather than autoregressive synthesis to generate the (repaired) output. We describe SynShine's architecture in detail, and provide a detailed evaluation. We have built SynShine into a free, open-source version of Visual Studio Code; we make all our source code and models freely available.

3.1. Background & Motivation

Problem-solving, motivation & engagement, and difficulties in learning the syntax of programming language are three fundamental challenges in introductory programming courses [**142**]. The dropout and failure rates are still high in introductory programming courses even after applying advanced methods and tools [**28**, **204**]. Helping novices with programming syntax can prevent novices to get demotivated [**142**] at the beginning of the learning process. In this paper, we aim to help novice programmers by automatically suggesting repairs for syntax errors. Consider the program in Fig. 3.1, which is an actual example our dataset of novice programs with errors [**35**].

Note the use of “x” instead of “*” on line 8. Many school maths texts use “x” for multiply, so this an understandable error.

In an introductory programming course, a novice may make this error by force of habit, and then find it quite challenging to fix the problem. Most popular IDEs (Eclipse, IntelliJ, Visual Studio Code) have trouble fixing this; however, our approach, which feeds a `Javac`-based error diagnostic, into a multi-stage repair engine that combines unsupervised pre-training, with fine-tuning, can resolve this.

```
1 import java.util.Scanner;
2 public class Multiplication
3 {
4   public static void main(String[] args){
5     Scanner sc = new Scanner(System.in);
6     int a = sc.nextInt();
7     int b = sc.nextInt();
8     int res = a x b;
9     System.out.println("The result is: " + res);
10  }
11 }
```

FIGURE 3.1. Incorrect novice code sample

Researchers have been interested in compiler diagnostics or syntax error messages for over half a century [**26**]. Barik et al. reported [**22**] that the difficulties programmers face while reading or understanding error messages are comparable to the difficulty of reading source code. Understanding Java error messages is quite challenging for two reasons; i) the same error produces different diagnostics depending on the context, and ii) the compiler may produce the same diagnostic for different errors [**22**]. Though prior works [**107**, **172**] addressed fixing errors in novice programs, DeepFix [**71**] was the first to apply deep learning to fix errors. DeepFix considers code repair as Neural Machine Translation (NMT) and uses an encoder-decoder based deep learning model to fix errors in C programs. Though initially aimed at semantic bugs, the approach also works for syntax errors. This approach was limited by the use of RNN (recursive neural network) seq2seq models—the RNN architecture is challenged by longer inputs, and outputs; also since the back-propagation through time (for the recursive elements) is not easily parallelized, it’s challenging to exploit larger datasets and additional processors. These became nagging problems in NLP; initial efforts with basic attention mechanisms [**135**] were supplanted by powerful multilayer models with multiple attention heads to avoid recursive elements altogether [**193**], yielding high-capacity, eminently parallelizable *transformer* models. Certain errors, such as the ones relating to block nesting, statement

delimitation (with "`;`") *etc.* involve long-range syntax dependencies, and require attending to very long contexts, which transformers can do better; still, even these models fail when the dependencies become much longer.

Ahmed *et al.* [**5**], developed *BF+FF*, using a multi-layer, multi-head transformer approach, to address the limitations of traditional seq2seq models. In addition, *BF+FF* used a two-stage pipeline, with the first stage addressing long-range block nesting errors, even ones beyond the range of transformers (BlockFix) and the second stage addressing shorter-range errors (FragFix). Using the Blackbox [**35**] dataset, they demonstrated that their approach substantially improved over prior work on the same dataset [**171**] (which used language models). *BF+FF* had important limitations, noted in their paper; it didn't take advantage of error localization and diagnosis provided by compilers; it also didn't effectively address errors in identifiers. Indeed, none of the existing approaches dealt effectively with identifiers, since they had to limit vocabulary. Deep learning models are challenged by large vocabularies, which require very large embedding and softmax layers. (See [**108**] details). We use BPE [**108**] to address this issue.

By addressing these limitations, we were able to achieve very substantial improvements on the state of the art for fixing Java programs. Ahmed *et al.*[1] and Gupta *et al.*[2] provided extensive source-available replication packages which enabled us to provide a detailed comparison (See § 6.3).

3.2. Methodology

Previous work had various limitations: longer programs were difficult to repair; error messages from compilers were not used; vocabulary limitations in DeepFix and design choices in *BF+FF* limited the ability to address errors in identifier usage. SynShine directly addresses these issues, and achieves substantial improvements. We use a multi-stage pipeline which incorporates the Java programming language compiler (`Javac`), along with three learned DL neural networks (DNN). The first DNN model is directly based on the BlockFix stage provided by *BF+FF*; this resolves (the potentially long-range dependent) nesting errors in the program. In the second stage, SynShine departs from *BF+FF*. *BF+FF* uses the fixed nesting structure from BlockFix to split the program into lines, and then just tries to fix *every line*; this leads to a lot of incorrect fixes. Deepfix and

[1] `https://zenodo.org/record/4420845`
[2] `https://bitbucket.org/iiscseal/deepfix/src/master/`

Santos *et al.* also try to fix the entire program. The second stage (LineFix) in SynShine uses the line-location of the error, as detected by the standard `Javac` compiler, together with the actual error message, and generates relevant fixes for delimiters, operators, and keywords; it also flags potential locations for errors in identifier usage; these locations are sent to the third & final stage, UnkFix. The UnkFix DNN model uses a RoBERTa-MLM to correct any identifiers that flagged as potentially wrong by LineFix.

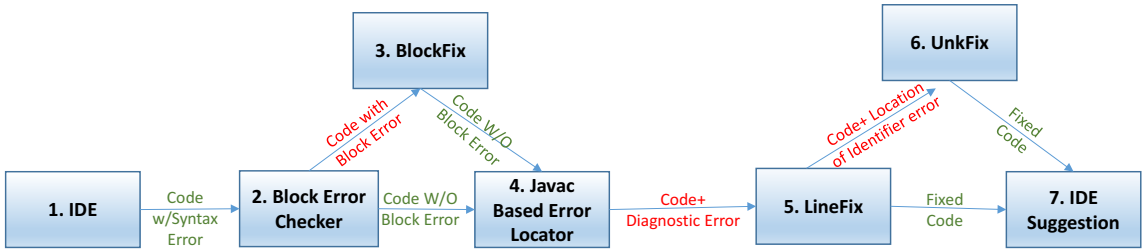


FIGURE 3.2. Overall architecture of the SynShine tool.

```
public class PrintVsPrintln
{
    public static void ( String args [ ] )
    {
        System.out.print(3+);
        System.out.println(4+5);
    }
}
```

(a) Program with syntax error

```
a_50.java:3: error: <identifier> expected
    public static void ( String args [ ] )
                       ^
a_50.java:5: error: illegal start of expression
        System.out.print(3+);
                         ^
2 errors
```

(b) javac error message before fixing

```
a_50.java:1: error: class PrintVsPrintln is public,
should be declared in a file named PrintVsPrintln.java
public class PrintVsPrintln
       ^
1 error
```

(c) javac error message after fixing

FIGURE 3.3. Locating erroneous line using `Javac`

3.2.1. Overall Architecture. Fig. 3.2 shows the architecture of our approach. When the IDE flags an error (step 1) we first pass the program through a block-nesting error checker (2), which is a simple pushdown automaton, that checks the program's nesting structure. If block-related issue is found, it's sent from (2) to BlockFix (3) a transformer model (as provided in the open-source *BF+FF* implementation [**5**]) for repair. In either case, the code, hopefully now free of block-nesting errors, is sent to step 4, where we try to locate the erroneous line using `Javac`.

We identify the line that `Javac` associates with the syntax error, and pass it on to LineFix (step 5) with the error message. In some cases, LineFix can fix it directly; in others, it passes a token position to UnkFix (6), primarily to fix errors in identifier usage. Finally, the fixed code is returned as a suggestion to the IDE (7).

We separate the line-level repairs into LineFix and UnkFix to eke out more functions out of deep-learning model capacity. LineFix outputs one of 154 possible editing commands, to insert/delete/substitute delimiters, keywords, operators, or identifiers. We limit its output vocabulary to 154; the full list is made available in supplementary materials. This limitation improves performance, but results in more "unknown" fixes, as described further below (§ 3.2.4). These unknowns are resolved by the final DNN model, UnkFix. UnkFix uses a high-capacity masked-language model to suggest a fix (usually an identifier being renamed or inserted) given a location. In combination, these elements allow us to substantially surpass the state-of-the-art.

3.2.2. `javac` Errors: Promises and Perils. While novices often find compiler error messages unhelpful [**114**], our own experience suggests that they do help experienced developers! This suggests that with sufficient training data, machine-learning models could learn something about how to fix syntax errors, from compiler syntax-error diagnostics. Older machine-learning-based approaches had not leveraged these diagnostics [**5**, **71**, **171**]. Recently, DrRepair [**214**] uses these diagnostics for fixing C programs; SynShine also uses them.

`Javac` flags syntactically incorrect programs with diagnostic errors; though the messages *are not precise*, they are sometimes useful. Fig. 3.3 (a) presents an actual novice program with two syntactic errors (missing "*main*" and unwanted operator "*+*"). The `Javac` compiler reports those two errors for the given program 3.3 (b). Although these error messages are unhelpful, `Javac` does in this case finger the actual lines with errors. Line-level syntax error localization can be helpful, if the program is long. DeepFix, for example, can not fix longer programs; it relies on seq2seq translation methods, and so has trouble with inputs longer than a few 100's of tokens. *BF+FF* resolves this problem by trying to fix every line in the program using its FragFix second stage; this approach does induce a fair number of false positives. `Javac` promises more accurate location, which could reduce this risk.

There is a potential issue with using `Javac`, arising mainly from the constraints of our novice error Blackbox dataset. `Javac` generates some error categories which cannot be fixed by editing the program directly. These errors arise for example, from file-naming conventions and incomplete typing environments. For example, class name & filename mismatch errors, and missing class definition errors are shown in Fig. 3.3 (c). The Blackbox dataset (also used by Santos *et al.* [**171**] and Ahmed *et al.* [**5**]) only includes programs with errors and their associated fix; it does not include the complete programming environment. SynShine only deals with errors that can be fixed by *directly editing the* Java *source*; we ignore the others. This is a decision also made by all the other papers that deal with syntax error correction [**5**, **71**, **171**]; we do, however, make use of compiler diagnostics for Java, and do manage to fix a much larger portion of the errors in the Blackbox dataset than prior work, as seen in Table 3.2. Therefore, to remove the errors we don't consider from our training set, we simply wrote a wrapper around `Javac`, to retain just those errors that can be fixed by editing the source [3]. However, it is important to note here that these "unfixable" errors in our dataset are counted in the denominator when we report our final success rate; in other words, these errors excluded from training are counted against SynShine and other tools as failures, and are not ignored in our reported performance.

3.2.3. Recovering Block Structure: BlockFix**.** Errors involving imbalanced curly braces are prevalent in novice programs, and are hard to resolve because of the long distance between the pair of braces. Ahmed *et al.* [**5**] report that block nesting errors consist of around 20-25% of all syntactic errors in novice programs [**5**]. They incorporate a component, BlockFix, for fixing block-nesting errors. BlockFix uses a transformer-based machine-translation model to locate & fix block-nesting errors; the translation model is trained on synthetic data with artificially generated nesting errors, and the corresponding fix. It works with an abstracted version of the code without statements, identifiers, and types to fix errors in nesting structure. In SynShine, we simply adopt the BlockFix component from the implementation made available by Ahmed *et al.*'s replication package.

Ahmed *et al.* abstracted out all the identifiers, constants, expressions, and delimiters, retraining just the curly braces and keywords (see Fig. 3.4). They then introduce structure-related syntax

[3]Most common ignored errors relate to "file and class name mismatch" and "undeclared identifiers".

corruptions, by adding or dropping the curly braces at randomly chosen positions; and then teaching the model to recover the original abstracted version from the corrupted model. BlockFix model learns to fix such errors by training on many such abstracted, corrupted pairs. After fixing the nesting error, the abstracted tokens are replaced with the original ones, and the program is passed to the following stages for further processing.

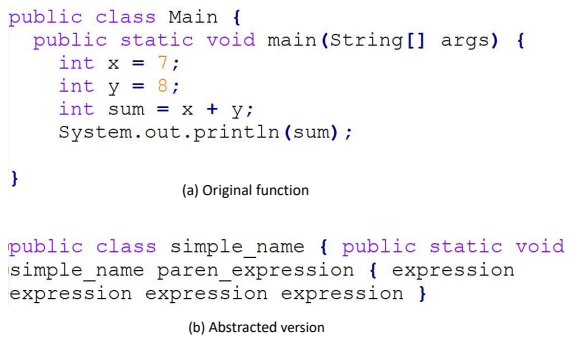


FIGURE 3.4. Abstracting source code for recovering block Structure.

We found that `Javac` works quite well in localizing the error (at least the buggy line and finding the line is sufficient for our approach) *if the program is free of nesting errors.* This is why we apply BlockFix, *before* running `Javac` to localize and diagnose the error.

3.2.4. Fixing Line Error: LineFix. LineFix uses a RoBERTa based pre-training + fine-tuning approach. RoBERTa derives from BERT, which uses unlabeled text data to pre-train deep bidirectional representations of text by jointly conditioning on both left and right context in all layers of a deep transformer model [**56**] to perform simple, self-supervised tasks like filling in masked tokens. This model and training method effectively captures the statistics of token co-occurrences in very large corpora within the layers of the transformer model. This pre-trained model learns excellent vector representations of code patterns in the higher layers of the transformer; these learned vector representations can be "fine-tuned" with just one additional output layer for specific tasks, and achieves state-of-the-art performance. For pre-training, BERT uses two tasks: fill in masked out tokens using the context (also known as Masked language modeling, or "MLM") and predict the next sentence given the previous one (the "NSP" task). Liu *et al.*'s RoBERTa (Robustly

Optimized BERT Pretraining Approach) dominates BERT's performance [**129**]. Liu *et al.* drop the NSP objective but dynamically change the masking pattern used in the MLM of BERT models.

Pre-training + fine-tuning also works very well indeed for code. One can gather millions of unlabeled code tokens from open-source projects, conduct pre-training, and then fine-tune the model with a limited amount of labeled data to achieve state-of-the-art performance in different software engineering applications [**29**, **64**, **104**, **225**] (albeit not yet for code syntax repair). Since we are working on novice code correction and our objective does not involve any relation between two programs, such as Question Answering (QA) and Natural Language Inference (NLI), training on NSP is not beneficial. Furthermore, using a dynamic masking pattern to the training data helps the model achieve better performance in downstream tasks. Therefore, We use RoBERTa for pre-training and fine-tuning of the model.

Why pre-training? As explained in the papers on BERT [**56**] and RoBERTA [**129**], for natural language, and the very recent, but rapidly growing body of literature using pre-training for code [**1**, **6**, **29**, **64**, **69**, **97**, **104**, **133**, **139**, **157**, **170**, **225**], pre-training is a way to exploit enormous volumes of data in a self-supervised fashion to learn the statistics of token sequences, and capture patterns in a position-dependent vector notation. For our purposes, these pre-trained models are automatically ingesting patterns of syntax and identifier usage from vast quantities of source code (around a billion tokens) and bringing all this knowledge implicitly to bear to the task of fixing errors in syntax and identifier usage.

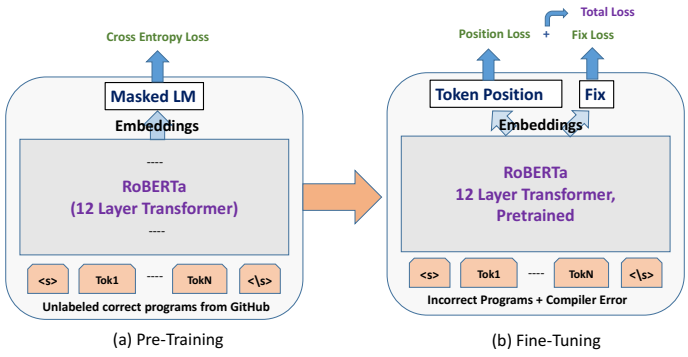


FIGURE 3.5. Pre-training and fine-tuning using RoBERTa.

Pre-training To generate the dataset for pre-training, we collected 5000 most starred Java projects from GitHub (since our end-goal is to correct Java syntax errors). We tokenized the files, yielding 1.2 billion tokens for the pre-training. For the MLM pre-training over code, we randomly select 15% of tokens, and replace with a unique token `mask`. The loss here is the cross-entropy of the original masked token. Of the 15% selected tokens, 80% are replaced with a specific marker `mask`, 10% are left unchanged, and a randomly selected token replaces the remaining 10%. This training method follows the standard RoBERTa protocol.

The architecture is as shown in Fig. 3.5. The main RoBERTa model is in the central grey box, labeled "RoBERTa" in 3.5 (a) and 3.5 (b). The left side is the architecture when RoBERTA is being pre-trained; the last layer on top is the MLM, implemented as a softmax layer taking the RoBERTa embeddings as input, and produces an output token, The entire model is trained using cross-entropy loss. Our RoBERTa architecture consists of 12 attention layers, 768 hidden dimensions, and 12 self-attention heads in each layer. We applied Byte Level BPE (Byte Pair Encoding) tokenizer [**108**] limiting the sub-token vocabulary size to 25K.

We trained the MLM model using cross-entropy loss on two NVIDIA Titan RTX GPUs for five epochs with a batch size of 44 sequences and learning rate $5e-5$. When pre-training completed, our MLM model achieved a final loss corresponding to a perplexity of 1.46, (cross-entropy 0.546 bits) which is rather low; RoBERTa for natural language yields final losses around 3.68-4.0 perplexity (1.88 to 2 bits).

Fine-tuning The fine-tuning step here is to train LineFix, a model that accepts an incorrect input line from a novice program, (the line flagged by `Javac` as containing a syntax error) together with the text of the error itself, and then generates a set of locations and edit commands, using multi-label classification layers, as explained below.

For fine-tuning and then for evaluation, we used realistic novice programs with syntax errors and human-produced fixed versions. We used the exact dataset used by Santos *et al.* [**171**] and Ahmed *et al.* [**5**] from the Blackbox [**35**] repository. This dataset contains 1.7M pairs, of erroneous and fixed programs. Both Santos *et al.* and Ahmed *et al.* primarily report their performance on programs with a single token error because a single edit can fix a large fraction of the programs (around 57%). Therefore, for a fair comparison, we also initially focused our evaluation on single

token errors and broke down our performance by token-length, as done by Ahmed *et al.* We selected a test set of 100K samples, with samples stratified by length, from the full dataset for the evaluation. We divided the test dataset into ten token-length ranges (lengths of 1-100, 101-200, ... , and 900-1000 tokens), with each range having around 10K examples. We prepare our fine-tuning dataset from the remaining examples.

Since BlockFix handles long-range block-nesting errors, the LineFix stage is focused on those errors unrelated to nesting. We discarded the programs with imbalanced curly braces from the training set, and after tokenization, we found around 540K examples to train the model. We used `Javac` (discussed in Section 3.2.2) to localize the error. The input to the model then is the buggy line indicated by `Javac`, appended with a special separator token (denoted `<SEP>`) followed by the error message from `Javac`. Altogether, the maximum input is 150 sub-tokens, which captures virtually all the input lines flagged as erroneous in our dataset. From this, the pre-trained RoBERTa model calculates positional embeddings for each subtoken; however, as with many RoBERTa-based classification tasks, we use just the embedding of the first token.

The desired output is the matching edits required to create the fixed version, as explained next.

To make a complete fix, the model should produce one or more locations, and one or more "fix", *viz* edit commands. The fix has two parts: i) the *type* of fix (insertion, deletion, or substitute?) ii) the *content* of the fix (is it a specific keyword, delimiter, or any other token?). When the *type* is a deletion, there is no *content* required: if the model identifies the buggy token at position x and recommends deletion, we just drop that token. For substitute operation, if the location is x and the edit command is $substitute \rightarrow y$, we will replace the token at position x with the token y. For insertion, if the command for position x is $insert \rightarrow y$, we will add the suggested y token at the $x + 1$ position. For insertion at the start of the line, we use a special token. For example, consider the following buggy line from Fig. 3.3 (a).

```
public static void (String args[])
```

To fix this missing "main", LineFix should output the location "3" and the fix "$insert \rightarrow unk$" ("main" is an identifier). This "unk" will be coverted to "main" with another model. We will discuss it in Section 3.2.5.

Our model's final layer consists of two distinct multi-label classification output layers, one which outputs one or more locations, another which outputs one or more fixes. The input to both these output layers, as explained above, is the RoBERTa embedding of the first token of the input. From this input, the two separate multi-label classification output layers calculate the position(s), and fix(es). Since most (99%) of the erroneous lines are 100 tokens are less, we output one or more positions (1-100) from the first output layer, and, from the second output layer we generate one or more of 154 distinct possible fixes. We remind the reader that a multi-label classification task involves generating an output vector of class probabilities, where the classes are non-exclusive. A single input might generate one or more class labels. In our case, we take all class labels in the output vector scoring above 0.5 as an assigned label. If none of the classes are assigned a probability above 0.5, we just take the highest probability class label. In almost all cases, we have only one fix per line, so one position and one edit command are expected; however, in rare cases, more than one position *and* more than one edit command could be generated. In the former case, we just apply the edit command at that position; in the latter case, which occurs very rarely, we try all combinations and return the first edit combination that compiles. A somewhat more common case (for example with multiple missing delimiters, like ")"), we get one edit command like *insert* $\rightarrow$) and multiple locations, in which case, we just apply the same edit at all locations.

There are reasons for our choice of multi-label classification, rather than simply synthesizing the fixed output. Prior approaches [**5**, **71**, **171**] used autoregressive[4] code generation to synthesize repairs. Given the sizeable vocabularies in code, many complex dependencies must be accounted for when generating code tokens conditional on previous tokens, the original input tokens, and the compiler error. We simplify the problem into a multi-label classification task here; all that is required is to identify the token positions(s) of the error, and the applicable edit commands. In the vast majority of cases, there is usually only a single change required per line). This allows the model to learn, and rapidly reduce training loss and perform well under test. In addition, the multi-labeling approach (rather than auto-regressive generation also allows us to handle repairs that require multiple fixes on the same line (example below, Fig 3.6). It's important to note that a single line can contain several token locations with errors, and distinct edit commands at each

[4]Autoregressive generation conditions the generation of each token on previously generated tokens, and is used in machine-translation approaches.

```
-System.out.println(*);
+System.out.println("*");
```

FIGURE 3.6. Example requiring two edits to fix

position. Limiting the size of the set of possible fixes to 154 will limit the ability to fix identifier names; this is handled by including fix commands that insert and substitute to *unk* in the output vocabulary of LineFix; these fixes are handled by a component is called UNKFIX, which is described in § 3.2.5. Note that dealing with multiple fixes on different lines is easily manageable. If there are multiple positions, all with the same fix (like Fig. 3.6), one can just perform that fix at all the positions. However, for multiple positions and multiple fixes one needs to try all combinations until the `Javac` accepts with no errors. We did not incorporate that to our code, because:

(1) Trying all possible combinations will slow down the entire process.
(2) Two different errors in a line (even in a file) is very rare. In the Blackbox data repository, for example, the majority of files contain just a single syntactical error.

The standard way to train multi-label classification layers is with binary cross-entropy loss (with logits), which is what we use for our fine-tuning. Since both the output layers are closely related to each other, we fine-tuned them simultaneously for 5 epochs. We collected the loss from each layer and added them to define the batch's final loss, and updated the model accordingly. Note that the same pre-trained model parameters (from Fig. 3.5 (a)) are used to initialize these; during fine-tuning, all parameters in all layers are modified (Fig. 3.5). We use the Huggingface open-source implementation of RoBERTa [**87**] for both pre-training and fine-tuning.

Utilizing Compiler Diagnostics during Fine-tuning Apart from localizing the erroneous line, the compiler warning can boost the performance of the fine-tuning model. As an input sequence to the model, we tried two versions, *i.e.,* with the warning, without warning. We observed a small but significant improvement in line-level code fixing (detailed in Section 3.3.2). Consider the following code snippet from the Blackbox dataset. The variable "bmr" is declared twice, and the second declaration is invalid. Though the `Javac` localizes the error correctly, it is really hard for the model to resolve this without any hint. Our model fails to fix this one when trained without the compiler message. However, with the compiler error message, our RoBERTa-based fine-tuned model can

solve errors like this one by deleting the token "double". This particular example is fixable with a modern IDE; however, it serves as a good illustration of how our model can use error messages. We remind the reader that in general we can handle numerous examples that IDEs cannot. Several typical examples are included in the supplemental file https://bit.ly/3CMM0TP.

```
double bmr;
/* some additional irrelevant lines */
boolean isMale = male == 'M';
if(isMale)
double bmr = ((9.5 * wgt) + (5.0 * hgt)
               + (6.7 * age) + 66.47);
```

Without Warning:

```
double bmr = ((9.5 * wgt) + (5.0 * hgt) + (6.7 * age) +
66.47);
```

With Warning:

```
double bmr = ((9.5 * wgt) + (5.0 * hgt) + (6.7 * age) +
66.47); <SEP> variable declaration not allowed here
```

LineFix works best with small sequences. Java is inherently verbose, and so sequence lengths are often beyond the model's capacity. Compiler diagnostics help us in two ways. Primarily, it helps us localize the error, and secondly, the message (even if imprecise) helps deep learning models fix the error. This claim is supported by a study (Yasunaga *et al.* [**214**]).

3.2.5. Recovering Unknown Tokens: UnkFix. Recall that LineFix output is restricted to 154 distinct fixes in the fine-tuning model. To deal with edits (inserts or substitutes of identifiers, constants *etc.*) outside of the limited vocabulary of edits, have an "escape" mechanism. Out of these 154, we included two unique outputs *insert* $\rightarrow$ *unk* and *substitute* $\rightarrow$ *unk* to cover other changes. To precisely identify these "unk" tokens, we use UnkFix, which reuses the masked-language model (MLM) we obtained during pre-training. This masked language model can recover the *unk* tokens if sufficient context is given. After getting the position information, we can collect sufficient tokens from the previous and following lines to fill the input buffer, and ask the pre-trained model to

unmask the *unk*. Applying this approach, we could fix several *unk*-related program errors like the following ones where the LineFix predicts *insert* → *unk* and *substitute* → *unk* for "Item" and "Integer", and then the MLM is able to locate them correctly.

```
-public void takeItem (item ) {
+public void takeItem (Item item ) {

-float number = float.parseInt(text);
+float number = Integer.parseInt(text);
```

Note that though we designed UnkFix primarily for identifiers, it can potentially handle other tokens, including values.

3.2.6. Integrating SYNSHINE into VSCode. To make SynShine more broadly accessible, we have made it available within a popular IDE. We have initially chosen VSCode since it's widely available, free for students[5], and well-documented; in the future, we will incorporate SynShine into other IDEs. The source code for the integration is available in our replication package. A demo video is viewable: https://youtu.be/AR1nd2PJczU.

In this VSCode integration, we desired fast response times, and wanted to avoid the requirement for a GPU, since many novices may not have a GPU. So for the SynShine deep learning model, we just used CPU floating point operations; to avoid having to reload the (very large) model for each repair request, we wrapped the SynShine model within a "correction" server, which services HTTP requests from the IDE.

The IDE triggers a request to SynShine when the user requests a fix suggestion. When SynShine is triggered, VSCode looks for the active text editor and extracts the (erroneous) code content from there. After getting the content, VSCode sends an HTTP request to the code correction server. Models are pre-loaded in the correction server, so that it can immediately service requests. In this server, the code goes through our proposed pipeline presented in Fig. 3.2, and the code returns to the editor after finishing all the steps. Now we have two versions of the code, i.e.,

[5] https://visualstudio.microsoft.com/students/

the buggy code and the corrected version. We highlight the difference and present both versions to the user and allow them to accept or reject the solution.

Note that the demo presented on the link mentioned above was captured on a machine without any GPU. We observe that SynShine can operate on a CPU and is quite fast at generating the solution even though the models were trained on GPUs. Just to get a sense of the delay, we randomly chose 200 erroneous programs of various lengths from our dataset, and measured the response time (time from the "SynShine" button press to the time the fixed code is received back). The average response time is 0.88 seconds (standard deviation 0.49s, maximum 2.2s). While this by no means instantaneous, we can still provide a fix for a syntax error virtually always within a second or two, potentially saving the novice and instructor's time. Our approach to integrating SynShine into VSCode thus arguably attenuates the need for expensive GPUs, and facilitates the use of the deep learning model in CPU-only machines. The CPU we used for the experiment is "AMD Ryzen 7 2700X". The code correction server occupies 1.765 GB of the memory.

SynShine's response time is significantly lower than the time needed by a programmer to fix the program. Brown and Altadmri divided the mistakes that occurred in the Blackbox repository into 18 different classes, where 11 of them are syntactical errors [**34**]. The programmers take 13-1000 seconds (median) to fix the mistakes [**34**]. Our model, on the other hand, takes less than a second on average to process the files and suggest a fix.

3.3. Evaluation & Results

In our evaluation, we compare our work with several baselines: Santos *et al.*, DeepFix , *BF+FF*, and SequenceR. The original DeepFix [**71**] used a GRU based RNN encoder-decoder translation model, which takes an entire program (with syntax error) as input, and produces a fix. For baselining their *BF+FF* tool, Ahmed *et al.* used two versions of DeepFix, one ("short") trained on error-fix pairs upto 400 tokens long and another ("long") trained on error-fix pairs upto 800 tokens long. Another approach, SequenceR [**45**] has reported success in fixing *semantic* errors, when provided with fault localization; it is also adaptable for syntax errors. SequenceR differs from DeepFix in a few ways: it uses a separate fault localizer, and also incorporates a copy mechanism. We describe the intricacies in full detail later. Ahmed *et al.*'s *BF+FF* program used a 2-stage

transformer-based lenient parser, as described above. Our approach combines several techniques: pre-training, compiler-based reporting, and fine-tuning with novice data.

Below, we present summary top-1 accuracy results, evaluated over a random sample of 100,000 examples of length upto 1000 tokens, with single-token errors, taken from the Blackbox dataset. The detailed result is presented in Table 3.2. We follow the lead of the first paper in the area [**171**] in this table, reporting performance for single-token errors, which constitute 57% of the data in Blackbox. We report the numbers for more complex errors below. As can be seen, SYNSHINE

Santos *et al.* [171]	**DeepFix (short)**	**DeepFix (long)**	**SequenceR**	*BF+FF*	SYNSHINE
46.00%	63.25%	62.14%	56.89%	56.91%	**74.89%**

TABLE 3.1. *Summary Results: Santos* et al. *performance is as reported by them; we measured the others*

achieves a substantial performance boost, over all the prior approaches, elevating the performance further and providing us with the motivation to build it into a popular IDE to make it more widely available. Here below, we evaluate the performance in more detail, comparing SYNSHINE with the closer competitors (we exclude Santos *et al.* from this comparison) and also examine the contributions of our various stages to the significant overall improvement. We begin with an evaluation of the effect of program length on performance, then we consider the effect of the various components of SYNSHINE. Finally, we breakdown the performance of SYNSHINE in repairing various categories of syntax errors.

3.3.1. Fixing shorter & longer programs. Table 3.2 baselines the relative performance of SYNSHINE against prior work, broken down by length, in categories. The rows are different length ranges of programs. The second column is the fraction of the Blackbox programs falling in this length range. The next several columns are are baselines from prior work: first two are DeepFix (short) trained on shorter error-fix pairs (upto 400 tokens long), DeepFix (long) trained on pairs up to 800 tokens long. The next two are SequenceR, trained on all pairs in the training set, and *BF+FF*, trained exactly provided in Ahmed *et al.*'s scripts. Finally, on the last column we have our results from SYNSHINE; the 3 columns to the right of the SYNSHINE column represent the contributions of our 3 components. As can be seen our overall performance exceeds the performance

Token Range	Percent of Overall Data	DeepFix (short)	DeepFix (long)	SequenceR	*BF+FF*	SynShine By BlockFix	SynShine By LineFix	SynShine By UnkFix	SynShine Total
1-100	31.01%	76.71%	73.72%	59.21%	65.16%	21.01%	58.86%	2.41%	**82.28%**
101-200	29.43%	69.98%	67.15%	57.21%	60.24%	17.53%	58.98%	1.96%	**78.47%**
201-300	15.25%	63.27%	60.29%	55.40%	54.47%	14.35%	56.00%	1.93%	**72.28%**
301-400	8.56%	53.71%	54.02%	54.64%	50.01%	10.18%	54.45%	1.89%	**66.52%**
401-500	5.51%	42.17%	45.47%	54.54%	46.19%	7.71%	54.00%	1.88%	**63.59%**
501-600	3.63%	32.84%	39.78%	54.47%	42.81%	5.95%	53.83%	2.19%	**61.97%**
601-700	2.17%	23.76%	33.02%	54.35%	38.07%	3.80%	53.62%	2.10%	**59.52%**
701-800	1.90%	17.10%	26.57%	53.78%	35.35%	3.04%	51.98%	2.65%	**57.67%**
801-900	1.34%	11.43%	22.88%	55.56%	32.24%	2.20%	52.84%	2.19%	**57.23%**
901-1000	1.19%	8.80%	17.94%	53.87%	29.62%	1.27%	51.63%	2.10%	**55.00%**
Overall		63.25%	62.14%	56.89%	56.91%	15.56%	57.22%	2.11%	**74.89%**

TABLE 3.2. *Baselining* SYNSHINE *against prior work on syntax error correction. SequenceR was provided with* `Javac` *localization.*

of all the others in every length category, and on the entire sample significantly improves on all of them. Before we examine the numbers in detail, we first present some relevant details on how we measured them.

All evaluations were done on a very large, randomly chosen, representative sample of 100,000 error-fix pairs from Blackbox that were not seen during training by any of the models. The percentages shown in the second column, and the overall performance numbers (all numbers are *top-1 accuracy*) are thus robust estimates of actual performance on programs up to 1000 tokens long, which constitute around 95% of the Blackbox data. An additional evaluation on a random sample of the entire dataset is reported below. DeepFix (short), DeepFix (long), and *BF+FF* were all trained and evaluated using the scripts made available in the replication package of Ahmed *et al.* [**5**] and Gupta *et al.* [**71**].

SequenceR [**45**] had to be retrained for syntax error correction: Chen *et al.* originally developed SequenceR for fixing *semantic* bugs, viz., test failures. It uses the OpenNMT translation framework [**112**] and thus had to be trained using bug-fix pairs. SequenceR assumes that the precise location of the bug was known via fault-localization; the training pairs consisted of a) the buggy region of code, bracketed within `<start_bug>` ... `<end_bug>` markers, augmented with sufficient context (preceding and succeeding tokens) to make up 1000 tokens of input b) and the corresponding fix, which is the region including the changed code, upto a maximum of 100 tokens; longer fix regions will fail (this almost never happens in our setting). They used an RNN sequence-to-sequence encoder-decoder model that uses LSTM for the recurrent nodes, and incorporates a copy

mechanism to enable the model to generate specific local variables, *etc.* in fixes. We used the code provided by Chen *et al*, and trained the model using Blackbox data; we used the `Javac` compiler to find the error location, and created training/test pairs using the `Javac` indicated location (with context), together with the corresponding novice fix. In our case, since most novices' programs are shorter than 1000 tokens, we provided the entire novice program as context. Once SequenceR is trained, it can generate fixes, given the novice program with error, with location indicated as above. However, SequenceR cannot insert or delete entire lines, so it cannot fix many nesting errors (for example, by inserting or deleting a line with a single "{" or "}" delimiter).

Our overall accuracy ranges between 55% to 82%, and always outperforms DeepFix long (18%-74%), and short (9%-77%), SequenceR (54%-59%) and *BF+FF* (29%-65%). Both SequenceR and SynShine benefit from the error location provided by `Javac`. By improving on prior work at every range, on the entire representative 100,000 sample, SynShine achieves significant gains in *overall performance* (bottom line) over the state of the art. Two factors contribute to this improvement: i) javac-based error localization and ii) robustness of LineFix and UnkFix. javac-based error localization enables a more selective LineFix+UnkFix to the most likely errorful code, thus reducing false positives; Ahmed *et al.*'s *BF+FF* attempts corrections throughout the program, resulting in more mistaken corrections. The robustness of LineFix and UnkFix is really boosted by the pre-training + Fine-tuning strategy; we explore the relative benefits of this step further below.

Table 3.2, in columns under the SynShine header, also shows relative contributions of the components of SynShine. First stage is BlockFix borrowed from *BF+FF*. About 20%-25% programs, regardless of length have nesting errors. BlockFix's accuracy decreases with program length, and we observe that the contribution of BlockFix is low after 700 tokens. However, for the other 75% to 80% programs without nesting errors, LineFix & UnkFix perform pretty consistently. Finally, we note that 1-1000 tokens cover about 95% of the overall data. To observe the performance of SynShine on the overall distribution, including programs over 1000 tokens long, we test it on 5000 random samples. We found that our model can repair 75.36% of the programs, and as before, comfortably exceeds performance of prior tools. Note that if the BlockFix model has already fixed the curly braces and there is no other error, `Javac` will not produce any error message, and LineFix will not process that. Note that we always compare the end-to-end tokens

of the reference and the model's proposed sequence; if needless "over" fixes are applied, that will be counted as wrong. Moreover, none of the fixes are credited twice. If the model is fixed by UnkFix, it alone receives credit; we did not count it in the LineFix column. Likewise, we credited a sample in the LineFix column, if it is completely fixed by LineFix and does not receive any help from UnkFix.

We also applied our model on files that required 2 and 3 edits to fix the program and observed 29.4% and 14.4% accuracy, which is much higher than the reported accuracy by Ahmed et al. (19% and 9%). Finally, we note that Ahmed *et al.* report on a blended strategy where shorter uncompilable programs could be sent to DeepFix and longer ones to *BF+FF*, thus obtaining better performance than either at all lengths. A similar strategy could be employed here, blending SynShine with other models, trying all the proposed solutions, and picking the ones that compile. However we didn't implement this approach: we just integrated SynShine into VSCode since it performs quite well at all lengths on its own, and avoids the need to load and run many models, and try repeated compiles.

3.3.2. LineFix: **The Role of Compiler Errors.** SynShine differs from both versions of DeepFix, and SequenceR, because it's multistage; it differs from *BF+FF* mainly because of the two new components, LineFix and UnkFix. We simply reused the BlockFix component made available by Ahmed *et al.* [6], and find performance very similar to that reported by them for this component. The improvements reported in Table 3.2 clearly arise from our two new components. We now focus in on LineFix and evaluate how it contributes to overall performance. LineFix's task is to take an input line flagged as a relevant syntax error (by `Javac`), together with the actual error, and then output a position, and an editing hint (insert, substitute, delete). LineFix improves upon the FragFix stage of *BF+FF* in two ways: first, it uses pretraining+finetuning, and second, it also takes the syntax error message from `Javac` as an additional input. The value of pre-training has been extensively documented for code-related tasks [**1**,**6**,**29**,**64**,**69**,**97**,**104**,**133**,**139**,**157**,**170**,**225**], so we focus here on the effect of providing compiler errors. Note again that LineFix has two tasks: *Localize* the token to be replaced, and and output an editing command with the correct *Fix*. We evaluate the impact of compiler warnings using 10,000 randomly chosen erroneous lines, of various

[6] https://zenodo.org/record/4420845

lengths, each taken with and without the compiler syntax error messages. Since we're evaluating fixing capability on single erroneous lines, rather than entire programs, the numbers reported below are higher than in Table 3.2.

With compiler error?	Localization F-Score	Fix F-Score	Complete Correction (Location+Fix) Accuracy
No	90.75%	92.41%	86.71%
Yes	93.58%	93.18%	89.39%

TABLE 3.3. Impact of using compiler error

Table 3.3 presents the impact of using the syntax error message in our tool.

We gain around 2.7% improvement in overall accuracy using the compiler error message. We also see improvements on both Localization and Fix f-scores by providing the compiler message along with the the erroneous line (row 1 & 2). The improvement is more for the Localization than for the Fix. We tested the statistical significance of all differences, using Binomial difference of proportions test on a trial sample of 10,000; we then corrected the p-values using Benjamini-Hochberg. The improvements observed when using compiler error message for overall accuracy and fix location f-score are *highly* significant ($p < 1e-9$); however, the f-score for the fix *per se* are only significantly improved ($0.01 < p < 0.05$). This suggests that the compiler error message is of highly significant help in providing our model with information required to locate the precise token that needs to be edited, and somewhat less so to identify the precise edit that is required. It is *very important* to note however, that the `Javac` compiler is of crucial help in locating the line where the error is located. This above study also shows that the actual error message *per se* helps our model locate the *token* within that line that needs to be edited.

We present an illustrative example of how compiler error messages help. Sometimes the compiler warnings are very precise, *e.g.,* when semicolons or other punctuations are to be inserted. In such cases, it may appear that the task is quite simple, and the model is simply "translating" the error into a fix. We sampled 50 programs and observed how many of them can be fixed just by reading the comments. We observed that in roughly 60% cases, the `Javac` warning is not that helpful, and

the model learns to respond in fairly nuanced ways to address the error. Consider the following repair that LineFix correctly achieves.

```
-return s == reverse ( String s ) ;
+return s == reverse ( s ) ;
```

`Javac` *per se* not helpful: it produces an error message suggesting to insert "`)`" after "`String`". LineFix learns to ignore such messages, and instead correctly omits the token "String". Therefore, the model is not just "translating" the message from `Javac` into a fix; The high capacity of the model, enriched by pre-training and fine-tuning, is deployed to leverage the often incorrect, imprecise message from `Javac` into a good fix. Depending on the error, it can resolve a very imprecise message from `Javac`. Indeed, quite often the same error message from `Javac` can lead the model to provide very different (correct) fixes.

3.3.3. When SynShine Fails, and When it Works. We now examine in further detail the cases where SynShine works correctly, and where it does not. To be conservative, we have defined as a "failure" any fix *not exactly the same* as the one recorded in the Blackbox dataset; note that a) the fix recorded in Blackbox is created by an actual human user, and also b) the recorded fixes always compile without error. We start with an examination of the cases where SynShine *fails* to produce a correct fix, as per our conservative definition, and then examine in detail the *diversity* of fixes that it does provide.

Fix Failures Despite our over-conservative definition of "failure", sometimes SynShine can generate a solution that differs from the user-intended solution but is still compilable with our javac-based compiler. In some cases, the solution is even semantically correct. As an illustration, in Table 3.6, examples 1, 2 & 3 are fixes generated by SynShine that not only compile without error, but are also semantically correct. By contrast, the last example in Table 3.6 is not semantically correct but compilable. Ideally, we'd like to characterize how often SynShine finds fixes that are not only compilable, but also semantically correct. The *compilability* of a fix that differs from the user's fix recorded in Blackbox can be determined automatically, and at scale (by just compiling!) and we report it below; however, the *semantic correctness* of a fix that differs from a user's fix requires

Length	Overall Compilability of fixes	Fixes Exactly Matching Blackbox	Compilability for non-matching cases
1-100	90.18%	82.28%	44.58%
101-200	86.13%	78.47%	35.58%
201-300	79.33%	72.28%	25.43%
301-400	73.35%	66.52%	20.40%
401-500	70.14%	63.59%	17.99%
501-600	67.83%	61.97%	15.41%
601-700	65.92%	59.52%	15.81%
701-800	64.00%	57.67%	14.96%
801-900	63.32%	57.23%	14.24%
901-1000	60.76%	55.00%	13.00%

TABLE 3.4. Compilability of SYNSHINE

Category	Prevalence of Error Category	Fix Accuracy (in %)
Keyword	5.04%	70.64%
Operator	5.87%	77.73%
Delimiter	80.37%	81.60%
Other	8.72%	60.94%

TABLE 3.5. Performance of SYNSHINE over diverse error categories

manual examination, and is not practical to do at a large scale. We try to characterize these to some extent by examining a small sample.

Seq No	Buggy Line	Model	Original Fixed
1	int i = ((int) (Math . random ( ) * 3) ;	int i = ((int) (Math . random ( ) * 3)) ;	int i = (int) (Math . random ( ) * 3) ;
2	int userInt_1 , int userInt_2 ;	int userInt_1 ; int userInt_2 ;	int userInt_1 , userInt_2 ;
3	System . out . print ((" Hello, world. ") ;	System . out . print (" Hello, world. ") ;	System . out . print ((" Hello, world. ")) ;
4	System . out . println (" sum = " + (sum +)) ;	System . out . println (" sum = " + (sum)) ;	System . out . println (" sum = " + (sum + 5)) ;

TABLE 3.6. Examples showing the compilability of the model

Table 3.4 presents the overall compilability of the solutions. The second column is the overall *compilability* of the generated fix. This is calculated as the fraction of the number of attempted fixes, that actually results in a successful compilation. The third column is the proportion of fixes that we deem correct, based on *exact* match with the fix recorded in Blackbox (the numbers will

match shown in the rightmost column of Table 3.2). As can be seen, we record many compilable cases as incorrect. The last column in Table 3.4 shows the proportion of apparent failures that are actually compilable: as an illustration, for programs up to 100 tokens long, about 45% of the cases that we record as an incorrect fix, in fact compile correctly. Depending on length, between 13% and 45% of the fixes we classify as failures are actually compilable. Table 3.6, examples 1,2,3,4 are exactly such fixes.

Now what proportion of these "compilable failures" are actually semantically correct? To get a (very) rough estimate of this, we did a small manual study. We randomly collect 50 cases where the model generates a compilable fix, that fails to match the user fix recorded in Blackbox. We found that about 18% of programs are semantically correct.

To summarize: even in our very conservative evaluation, SynShine produces the same fixes as recorded by a human in a sizable fraction (roughly 75%) of errors in our novice dataset; an examination of SynShine's failures suggests that it could possibly be helpful in some additional cases.

Fix Diversity What kinds of errors does SynShine fix? In our dataset, about 80% of the errors are related to delimiters, and even solving only those would make a significant dent. However, the novices make syntax errors in using keywords, operators, identifiers, and numbers; sometimes they introduce illegal spaces, declarations, characters, *etc.* We examined how SynShine performs with respect to different types of errors. For convenience, we divided the error into four major categories-keywords (all Java keywords), delimiters (e.g., semicolon, comma, parentheses, braces, brackets), operators (all Java operators), and others (identifiers, literals, and anything that falls outside the first three categories). To do categorization, we followed two rules. Errors that required substitutes or inserts belonged to the category of the substituted or inserted token; errors that required deletion belonged to the category of the deleted token. Thus if an error required a semicolon to be inserted, it was in the "delimiter" category; if an error required an extra "if" keyword to be deleted, it was in the "keyword" category.

We randomly sampled a 5K test dataset, and determined the error category prevalence in this dataset; see Table 3.5, first column, for the prevalence of errors in various categories. Delimiter errors dominate, and thus our model learns to fix those best (81.6% accuracy); however, it performs

well in other categories (60%-78% accuracy). The take-away from this analysis is that SynShine performs reasonably well at a wide range of syntax errors.

3.4. Related Work

The most closely related works are DeepFix [**71**], *BF+FF* [**5**], and Santos *et al.* [**171**] which we have discussed above. We also discussed SequenceR [**45**]. We have compared SynShine to all of these.

Gupta et al. [**70**] applied reinforcement learning to a very similar dataset like DeepFix [**71**]. It utilizes total count of compiler errors as a part of the reward mechanism. However, RLAssist [**70**] shows only a very minor improvement over DeepFix [**71**], and also it takes the whole program as input. Therefore, we did not re-implement RLAssist [**70**]. Though RLAssist [**70**] looks into compiler errors but it does not directly uses the error messages as we do. DeepDelta [**143**] is another approach that fixes compiler errors but mostly identifier name-related errors, not syntax errors. DeepDelta [**143**] was developed and tested on code from professional developers at Google. The authors also assume that precise knowledge of the location will be given to the program. Yasunaga *et al.* [**214**, **215**] introduce two compiler-dependent approaches to fix C program: DrRepair that utilizes C compiler warnings with a graph-based self-supervised approach, and BIFI that applies two models "critic" and "fixer" to fix the programs. A tool for the C programming language, Tracer, abstracts the code and uses a seq2seq model on the source code abstractions that are later concretized [**11**].

All the DNN based Automatic Program Repair (APR) tasks have a fault localization step [**45**, **57**, **125**, **136**, **192**], and these tools' performance depends a lot on the fault-localizer. Semantic code correction is an inherently difficult problem, and syntax correction can be considered as a subset of semantic code correction problems. None of the previous syntax correction tools has compared their work with these tools because previous syntax correction tools did not depend on any fault localizer. Some of the APR tools [**42**, **119**, **125**, **131**] expects syntactically correct programs and those approaches are not applicable for syntactical code correction. For our purposes the most directly compatible recent APR tool was "SequenceR" [**45**] which reported good performance, and also fixes errors at the line level; it was readily adapted to using the `Javac` to locate the line

to be fixed, so we chose it for comparison. Pradel et al. also detect specific types of bugs (e.g., accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations) but in syntactically correct code [**156**].

Brown *et al.* used BlueJ IDE to collect the data in Blackbox repository [**35**] In this paper, we did a case study on the performance of the popular IDEs (e.g., Eclipse, IntelliJ, VSCode, BlueJ) in fixing novice programs. We compare repair hints from Eclipse JDT Core Compiler for Java (ECJ) (used in both Eclipse and VSCode) and javac (used by IntelliJ and BlueJ). That is, both Eclipse and VSCode present the same error messages, and IntelliJ and BlueJ present the same error messages. Four IDEs, but ultimately, only two compilers. SynShine improves upon repair hints from both compilers. Therefore, we primarily focus on Eclipse and IntelliJ for the case study. We chose VSCode because it is popular, well-documented, available free for students, and is easy to extend. We were able to integrate SynShine into VSCode without any major difficulties.

3.5. Conclusion

We have described SynShine, a machine-learning based tool to fix syntax errors in programs. SynShine leverages RoBERTa pre-training, uses compiler errors (both location and message), and generates fixes using multi-label classification, rather than autoregressive generation, to achieve substantial improvements in fixing syntax errors. Our evaluation shows substantial improvements in fixing rates over the previous best results reported by *BF+FF*, and other tools, at all program lengths. Our evaluations suggest that the the use of compilers to locate the precise line provides a big advantage; our evaluations also suggest that the compiler error message *per se* may be helpful in locating the precise token within the line that needs to be repaired. We have built SynShine into the VSCode IDE, and have found that even without a GPU, the SynShine-enhanced VSCode can fix syntax errors fairly quickly, often in less than second. We have made all the source-code and data available, to the extent allowable under UK Law applicable to the BlackBox dataset. SynShine can fix errors that IDEs (Eclipse, IntelliJ, and VSCode) cannot. In the supplementary materials (https://bit.ly/3CMM0TP) we show several real-world examples of student-made errors that cannot be fixed by any of these IDEs, but can be fixed by SynShine. The supplementary materials also include the list of edits (FixList.pdf) that LineFix can generate. Finally, the entire

source for our SynShine, including the VSCode extension, is made available anonymously at https://doi.org/10.5281/zenodo.4563241.

CHAPTER 4

Multilingual Training for Software Engineering

Well-trained machine-learning models, which leverage large amounts of open-source software data, have now become an interesting approach to automating many software engineering tasks. Several SE tasks have all been subject to this approach, with performance gradually improving over the past several years with better models and training methods. More, and more diverse, *clean, labeled* data is better for training; but constructing good-quality datasets is time-consuming and challenging. Ways of augmenting the volume and diversity of clean, labeled data generally have wide applicability. For some languages (*e.g.,* Ruby) labeled data is less abundant; in others (*e.g.,* JavaScript) the available data maybe more focused on some application domains, and thus less diverse. As a way around such data bottlenecks, we present evidence suggesting that human-written code in different languages (which performs the same function), is rather similar, and particularly preserving of identifier naming patterns; we further present evidence suggesting that identifiers are a very important element of training data for software engineering tasks. We leverage this rather fortuitous phenomenon to find evidence that available *multilingual* training data (across different languages) can be used to amplify performance. We study this for 3 different tasks: code summarization, code retrieval, and function naming. We note that this data-augmenting approach is broadly compatible with different tasks, languages, and machine-learning models.

4.1. Background & Motivation

We now present some motivating evidence suggesting the value of multilingual training data for deep-learning applications to software tasks. We begin the argument focused on code summarization.

Deep learning models have been widely applied to code summarization, with papers reporting substantial gains in performance over recent years [**1**, **2**, **15**, **64**, **66**, **76**, **77**, **83**, **84**, **85**, **91**, **120**, **121**, **122**, **124**, **126**, **139**, **154**, **157**, **197**, **200**, **202**, **205**, **206**, **213**, **221**, **223**]. We focus here on *what*

information in the code ML models leverage for summarization (while we use summarization to motivate the approach, we evaluate later on 3 different tasks). Does every token in the program under consideration matter, for the code summarization task? Or, are the function and variable names used in the programs most important? Since identifiers carry much information about the program, this may be a reasonable assumption.

Considering the content words[1] in the example in Figure 1.3 there are four major terms (*i.e.*, Returns, text content, node, and descendants) used in the summary. The first 3 directly occur as tokens or subtokens in the code. Though the word "descendants" is missing in the program, high capacity neural models like BERT [**56**] can learn to statistically connect, *e.g.,* "descendant" with the identifier subtoken "child". This suggests that, perhaps, comments are recoverable primarily from identifiers. If this is so, and identifiers matter more for comments than the exact syntax of the programming language, that may actually be very good news indeed. If developers choose identifiers in the same way across different languages (*viz.,* problem-dependent, rather than language dependent) perhaps we can improve the diversity and quality of dataset by pooling training set across may languages. Pooled data sets may allow us to fine-tune using multilingual data, and improve performance, especially for low-resource languages (*e.g.*, Ruby and JavaScript from CodeXGLUE [**134**]). Since this is a core theoretical background for our work, we start off with two basic research questions to empirically gauge the possibility and promise of multilingual fine-tuning.

RQ1 What role do identifiers play in for code summarization?

RQ2 Do programs that solve the same problem in different languages tend to use similar identifier names?

4.1.1. RQ1: Role Played by Identifiers. We first examine the importance of *identifiers* for code summarization; specifically, we compare the relative value of identifier tokens and other tokens. We use the CodeXGLUE dataset and pre-trained CodeBERT embeddings for the task [**64**]. We begin with a brief backgrounder on CodeBERT [**64**] & BERT [**56**].

[1] "Content" words in linguistics, are words that carry meaning, as contrasted with *function* words, such as prepositions, pronouns, and conjunctions, which denote grammatical relationships. See `https://en.wikipedia.org/wiki/Content_word`. In code, we consider function words to be keywords, operators and punctuations, and content words to be identifiers (functions, variables, types, *etc*)

CodeBERT uses the pre-training + fine-tuning strategy of BERT, RoBERTa *etc* [**56**,**129**]. This approach begins with a self-supervised "pre-training" step, to learn textual patterns from a large, unlabeled, corpus using just the content; in the next step, "fine-tuning", task-specific *labeled* data is used to provide task-related supervised training. This approach is known to achieve state-of-the-art performance in both natural language processing, and software-related tasks [**6**, **9**, **29**, **64**, **69**, **97**, **99**, **103**, **139**, **225**].

We study the effect of identifiers in several steps. For the pre-training step, we start with the available CodeBERT model, which is pre-trained on a large, multilingual corpus of code. For the fine-tuning step, for this task, we use the CodeXGLUE benchmark dataset (see table 4.4 for languages and dataset sizes); we start with the original set of code-comment pairs, and apply two different treatments to create overall three different fine-tuning training datasets–1) base case leaving code as is, 2) a treatment to emphasize identifiers, and 3) a treatment to de-emphasize them. First, to emphasize identifiers we abstract out the program's keywords, separators, and operators by replacing those with three generic tokens (*i.e.*, "key", "sep", and "opt"), thus forcing the model (during fine-tuning) to rely more on the identifiers, for the task. Next, to assess the importance of keywords, separators, and operators, we abstract out the identifiers with a generic token "id". We fine-tune the model separately after each of these abstraction steps, thus yielding 3 fine-tuned models: the baseline, keyword-abstracted, and identifier-abstracted. We compare the results (smoothed BLEU-4) across all three.

If a fine-tuned model's performance is relatively unaffected by an abstraction, one may infer that the model relies less on the abstracted tokens. We perform these experiments with two languages with low-resource (*i.e.*, Ruby and JavaScript, See table 4.4) and two languages with high-resource (*i.e.*, Java and Python). We train, validate, and test with the same dataset in each case. For each test instance, we have one value from the complete program and another one from each of the two abstracted versions. We compared these values, using two distinct pair-wise Wilcoxon tests: 1) Alternative Hypothesis (AH): complete program > identifier de-emphasis & 2) AH: complete program ¿ identifier emphasis. We also perform the same test with the keyword-abstracted and identifier-abstracted versions (AH: identifier emphasis > identifier de-emphasis).

Dataset	Complete Program	Abstracting keyword, operator, separator			Abstracting identifiers		
	BLEU-4	BLEU-4	Effect Size	p-value (adjusted)	BLEU-4	Effect Size	p-value (adjusted)
Ruby	**12.53**	11.57	-0.028	0.008	7.94	-0.238	<0.001
JavaScript	**13.86**	13.06	-0.033	<0.001	9.06	-0.175	<0.001
Java	**18.72**	18.72	-0.002	0.344	11.41	-0.254	0
Python	**18.25**	18.10	-0.010	<0.001	11.68	-0.288	0

TABLE 4.1. *Role played by identifiers*

The data (table 4.1) suggests that abstracting the keyword, separator, and operator has a smaller impact on the performance: the BLEU-4 scores are rather similar (with *effect size* ranging from 0.002 to 0.033) to those from the unabstracted code. On the other hand, when de-emphasizing identifiers, the performance drops more, with effect sizes 5x-100x larger. We find similar results while comparing the emphasizing and de-emphasizing identifiers versions (omitted for brevity).

Language		Training					
		Ruby	JavaScript	Java	Go	PHP	Python
Testing	Ruby	*12.53*	11.84	**13.42**	12.32	**13.84**	**14.09**
	JavaScript	11.98	*13.86*	**14.16**	12.55	**13.90**	**14.09**
	Java	13.38	14.57	***18.72***	14.20	16.27	16.20
	Go	11.68	11.24	13.61	***18.15***	12.70	13.53
	PHP	17.52	19.95	22.11	18.67	***25.48***	21.65
	Python	14.10	14.44	16.77	14.92	16.41	***18.25***

TABLE 4.2. *Intra and inter language training and testing*

The results in table 4.1 suggests that syntax is less relevant that identifier names. In all the prior works, the training and testing were done in the same language. Since syntax is less important, could we train and test with different languages? The CodeXGLUE dataset enables just such an experiment. Using six different languages, we apply a CodeBERT model fine-tuned in each language, to a test set in another language. Table 4.2 shows that for high-resource languages (*i.e.*, Java, go, PHP, and Python), we achieve the best result (diagonal) when training and test data are from the same language. However, the performance does not degrade to a very large extent when

trained with one language and tested on a different one. Surprisingly we observe that for Ruby and JavaScript, we actually *achieve higher performance while trained with Java, PHP, and Python than the language itself.* That indicates that code summarization is not completely dependent on syntax (perhaps it relies more on identifier similarity, which we shall explore next)

Finding 1. Code summarization sometimes appears to train quite well with data sets from other languages, even if the syntax is different.

4.1.2. RQ2: Identifier Similarity Across Languages. Here, we evaluate RQ2: given a problem, do developers choose similar, descriptive identifiers, regardless of the programming language? Based on the findings in the previous section: if identifiers were indeed used in similar ways, perhaps code-comment pairs *from any programming language* could help train a code summarization model, *for any other language*. As an example, Figure 4.1 presents that all the "indexOf" functions implemented in Java, PHP and JavaScript use very similar identifiers "needle" and "haystack".

Quantitatively evaluating this hypothesis requires multiple implementations of the same problem in different programming languages, where we could compare identifier names. Luckily, RosettaCode provides just such a dataset. RosettaCode currently consists of 1,110 tasks, 305 draft tasks and includes 838 languages[2]. We collect the mined data[3] and study the same six languages (*i.e.*, Ruby, JavaScript, Java, Go, PHP, and Python) in the CodeXGLUE dataset. We get 15 cross-language pairs from six languages and measure identifier similarity between pairs of programs which solve the same problem in each language (*e.g.,* programs for graph diameter problem in Java and Ruby). For baselining, we also compare with a random pair (solving different problems) for the same two languages (*e.g.* graph diameter in Java, and SHA-hashing in Ruby). Fortunately, we found sufficient sample sizes for all our language pairs in RosettaCode. For example, for Java & Python we find 544 matched program pairs solving the same problem in both languages. We then take the 544 Java programs and randomly pair them with 544 other Python programs. Therefore, we have two groups of programs (*i.e.*, same program implemented in different languages and different programs implemented in different languages), and we check the similarity level between the two groups. Note that *size-unrestricted* random pairing may yield misleading results. Suppose

[2]Last Accessed August, 2021
[3]https://github.com/acmeism/RosettaCodeData

we have a Java & Python program *matched* pair with 100 Java subtokens and 40 Python subtokens. Now, if we replace the matched python program with a random, bigger program (*e.g.*, 500 subtokens), we may have more chance of finding matched identifiers. Therefore, while choosing the random program, we try to ensure it has a similar length to the program it is replacing in the pair. We randomly select a program having the subtoken counts within a 5% length range (*e.g.*, 38-42 subtokens for a 40 subtoken program) of the removed one. Fortunately, in *99.25%* cases, we get at least one example within the 5% range. On the remaining instances, we select the program with the nearest subtoken count.

We measure identifier similarity thus:

(1) Remove all keywords, operators, and separators from the programs.

(2) Break all CamelCase and snake_case identifiers and keep only one copy of each sub token.

(3) Discard too-small programs with less than 5 sub-tokens.

(4) Calculate the mean Szymkiewicz-Simpson coefficient (overlap coefficient) [**194**] for both groups (*i.e.*, same program pair and random pair) of programs.

(5) Repeat this process across all 15 language pairs, for all program pairs.

Table 4.3 shows the common program pairs have 89%-235% *additional* **identifier** *overlap* compared to random program pairs. We compare the matched and random pair overlaps using the non-parametric Wilcoxon signed-rank test (AH: random has less overlap than matched). We observe that the null hypothesis is rejected, and Szymkiewicz-Simpson Overlap coefficient[4] is significantly higher for the common program pairs in all the cases. That indicates programs solving the same problem (even in different languages) are much more likely to use the same or similar identifier names.

We also calculate each pair's Jaccard index [**92**] (similarity coefficient) and find 112%-309% more similarity between common pairs than random ones, thus, giving essentially the same result. However, we prefer to report the detailed result using the overlap coefficient because Jaccard index can be affected by the differing verbosity of languages. For example, on average, Java, Python, and Ruby programs in RosettaCode have 29.45, 17.93, and 17.63 identifier subtokens. Java has higher

[4]This is a measure of similarity like the Jaccard index; we use it here since sometimes the sizes of the programs are quite different. It's calculated as $\frac{|X \cap Y|}{min(|X|,|Y|)}$.

Language pair	#of common programs	Overlap coefficient for random programs	Overlap coefficient for common programs	Overlap coefficient increased in %	Effect Size	p-value (adjusted)
Java & Python	544	0.10	**0.32**	+210.67%	0.747	<0.001
Java & Ruby	532	0.11	**0.31**	+174.97%	0.751	<0.001
Java & Javascript	411	0.13	**0.36**	+188.17%	0.774	<0.001
Java & Go	602	0.19	**0.36**	+89.24%	0.641	<0.001
Java & PHP	282	0.08	**0.28**	+235.01%	0.740	<0.001
Python & Ruby	538	0.11	**0.35**	+228.89%	0.780	<0.001
Python & Javascript	377	0.12	**0.34**	+190.09%	0.728	<0.001
Python & Go	601	0.13	**0.31**	+133.06%	0.664	<0.001
Python & PHP	267	0.09	**0.29**	+214.32%	0.679	<0.001
Ruby & Javascript	370	0.13	**0.35**	+167.02%	0.751	<0.001
Ruby & Go	571	0.12	**0.28**	+133.47%	0.724	<0.001
Ruby & PHP	262	0.09	**0.28**	+205.32%	0.716	<0.001
Javascript & Go	418	0.14	**0.29**	+110.96%	0.635	<0.001
Javascript & PHP	236	0.11	**0.29**	+175.03%	0.678	<0.001
Go & PHP	293	0.10	**0.23**	+121.25%	0.562	<0.001
Overall	6304	0.12	**0.31**	+158.94%	0.697	0

TABLE 4.3. *Cross-language identifier similarity, when functionality is preserved*

subtokens compared to Python and Ruby because of the import statements, package naming etc. Therefore, Jaccard index between Java and Python will be lower than that of Python and Ruby even if the programs use very similar identifiers.

Finding 2. For a given problem, developers are likely to choose similar identifiers, even if coding in different languages.

In this section, we have presented evidence suggesting that a) identifiers are important for code summarization, that b) cross-language training is promising, and also that c) identifiers tend to be used in similar ways across languages. Taken together, these findings present a strong argument to try multilingual fine-tuning for SE tasks. Note that it is already well established that multi-lingual pre-training is helpful, and most BERT-style SE pre-trained models are multilingual [**1**, **64**, **154**, **157**]. However, pre-training data are unsupervised and easy to collect. Preparing clean data for the supervised fine-tuning phase requires more time and attention. In this paper, our aim is to prove that multilingual training is not only effective in pre-training stage but also in fine-tuning stage for SE models, which is already found to be beneficial for *natural language* models [**186**].

```java

public static int indexOf(ByteBuf needle,
    ByteBuf haystack) {
  // TODO: maybe use Boyer Moore for efficiency
    .
  int attempts = haystack.readableBytes() -
    needle.readableBytes() + 1;
  for (int i = 0; i < attempts; i++) {
    if (equals(needle, needle.readerIndex(),
      haystack, haystack.readerIndex() + i,
      needle.readableBytes())) {
      return haystack.readerIndex() + i;
    }
  }
  return -1;
}
```

(a) *Java*

```php

public static function indexOf(string $haystack
    , string $needle,
   int $offset=0):int
{
  $pos=self::strpos($haystack, $needle, $offset
    );
  return is_int($pos)?$pos:-1;
}
```

(b) *PHP*

```javascript
function indexOf(haystack, needle) {
  if (typeof haystack==='string')
    return haystack.indexOf(needle);
  for (let i=0, j=0, l=haystack.length, n=
    needle.length; i<l; i++) {
    if (haystack[i]===needle[j]) {
      j++;
      if (j===n) return i-j+1;
    }
    else {
      j=0;
    }
  }
  return -1;
}
```

(c) *JavaScript*

FIGURE 4.1. *Usage of similar identifiers (*e.g.*, needle, haystack) in "indexOf" function in different programming languages*

4.2. Benchmark Datasets and Tasks

We evaluate the benefits of multilingual training in the context of several tasks, and associated datasets. In this section, we discuss the models and tasks used for our experiments.

4.2.1. The Models. For our study of multilingual training, we adopt the BERT, or "foundation model" paradigm. Foundation models [**37**, **52**, **56**, **129**, **163**] have two stages: i) unsupervised pre-training with corpora at vast scale and ii) fine-tuning with a smaller volume of supervised data for the actual task. Foundation models currently hold state-of-the-art performance for a great many NLP tasks. BERT [**56**] style models have also been adapted for code, pre-trained on a huge, multilingual, corpora, and made available: CodeBERT and GraphCodeBERT are both freely available: both source code and pre-trained model parameters. While these models for code have thus far generally been fine-tuned monolingually, they provide an excellent platform for training experiments like ours, to measure the gains of multilingual fine-tuning. CodeBERT & GraphCodeBERT use a multi-layer bidirectional Transformer-based [**193**] architecture, and it is exactly as same as the RoBERTa [**129**], with 125M parameters; we explain them further below.

Pre-training The *CodeBERT* [**64**] dataset, has two parts: a matched-pairs part with 2.1M pairs of function and associated comment (NL-PL pairs) and 6.4M samples with just code. The code includes several programming languages. It was created by Hussain *et al.* [**89**]. CodeBERT model is pre-trained with two objectives (*i.e.*, Masked Language Modeling and Replaced Token Detection) on both parts. Mask language Modeling (MLM) is a widely applied and effective [**56**, **129**] training objective where a certain number of (15%) tokens are masked out, and the model is asked to find those tokens. For CodeBERT training, Feng *et al.* apply this first objective only to bimodal data [**64**]. The second objective, Replaced Token Detection (RTD) [**50**], is a binary classification problem that is applied to both unimodal and bimodal data. Two data generators (*i.e.*, NL and PL) generate plausible alternatives for a set of randomly masked positions, and a discriminator is trained to determine whether a word is the original one or not. We note that CodeBERT pre-training is all about representation-learning: by learning to perform the task well, the model learns a good way to *encode* the text, which is helpful during the next, fine-tuning stage. The pre-training

took about 12 hours on a machine with 16 NVIDIA V100 cards, and would have taken us very much longer, so we were grateful to be able to just download the estimated parameters.

Pre-training GraphCodeBERT GraphCodeBERT augments source-code with data flow, during pre-training. It uses a simple data flow graph (DFG) encoding a *where-the-value-comes-from* relation between variables [**69**]. The DFG nodes are variable occurrences, edges are value flow. GraphCodeBERT pretraining learns a joint representation of 1) the DFG structure, 2) DFG alignment with source code, and 3) the source code token sequences. GraphCodeBERT is therefore pre-trained with three training objectives (*i.e.*, Edge Prediction, Node Alignment, and MLM) on 2.3M functions (PL-NL pairs) from CodeSearchNet [**89**] dataset. For details see [**69**].

The pre-training+fine-tuning approach relies on VERY high capacity models, and are pre-trained over a large, multilingual corpus. Thus, even before fine-tuning, the models already know a lot about each language. Thus, fine-tuning on many languages should not negatively impact what the model knows about any one language. Thus we find that multilingual fine-tuning improves on monolingual fine-tuning in most cases. We believe our proposed approach would still consider the context surrounding the individual programming language even after multilingual training because these models have sufficient capacity to do so.

We now describe our tasks: in each, we describe the task, the dataset, and the multilingual fine-tuning approach (if applicable).

4.2.2. Code Summarization. *The Task:* as described earlier, the goal is to generate a NL summary given code in some PL.

The Dataset: There are several different code summarization datasets; we chose CodeXGLUE[5] [**134**], for two main reasons:

(1) CodeXGLUE is carefully de-duplicated [**176**]. Prior datasets like TL-CodeSum [**85**] have duplicates [**176**] in training, testing, and validation partitions. Duplication can inflate measured performance [**13**, **176**].

(2) We need a multilingual dataset to prove the effectiveness of multilingual fine-tuning. None of the existing datasets [**85**, **122**] is multilingual.

[5]CodeSearchNet [**89**] dataset is a standard benchmark, which has been incorporated into CodeXGLUE

Table 4.4 presents the number of training, testing and validation instances for each language. in CodeXGLUE.

Programming language	Training	Dev	Test	Candidate codes*
Ruby	24,927	1,400	1,261	4,360
JavaScript	58,025	3,885	3,291	13,981
Java	164,923	5,183	10,955	40,347
Go	167,288	7,325	8,122	28,120
PHP	241,241	12,982	14,014	52,660
Python	251,820	13,914	14,918	43,827

*Candidate codes are only used for code retrieval task

TABLE 4.4. *CodeXGLUE dataset*

Model & Fine-tuning Feng *et al.* use a transformer-based encode-decoder architecture for the code summarization task [**64**]. The encoder is all ready well-trained in the pre-training stage; for fine-tuning, the encoder is primed with weights from pre-training. Now, the transformer model is given the input code token sequence and asked to generate the comment, as in the Neural Machine Translation (NMT) problem. We fine-tune using the CodeXGLUE paired samples. During fine-tuning, the decoder is trained auto-regressively, using next-token cross-entropy loss. Feng *et al.* use smooth BLEU-4 [**128**] for the evaluations of the models. Subsequently, We replace the pre-trained CodeBERT with pre-trained GraphCodeBERT in the encoder while evaluating the effectiveness of multilingual fine-tuning with GraphCodeBERT.

Why baseline with CodeBERT for code summarization? Feng *et al.* compare CodeBERT with other popular encoder-decoder based (*e.g.*, LSTM [**184**], Transformer [**193**], RoBERTa [**129**]) models; CodeBERT handily beats all of them [**64**]. Thus, CodeBERT is a good baseline to measure the value of multilingual finetuning. CodeBERT also does very well on prior datasets: using smoothed Sentence BLEU-4, we found that CodeBERT reaches 44.89 on TL-Codesum [**85**], and 32.92 on Funcom [**122**][6]. TL-Codesum has high degree of duplicates; we found that Funcom also does, but just in the comments. CodeXGLUE has very little duplication, which makes it more challenging,

[6]As reported in [**67**, **176**], measurement approaches vary across papers, and these numbers may differ from prior results: we use smoothed sentence BLEU-4 everywhere in our paper.

and also more reliable. Note that GraphCodeBERT *does not report* any performance on the code summarization task, and so we had to measure it.

4.2.3. Code Search. *The Task* Given a natural language query, find the semantically closest code sample from a large set of candidates. Vector-based information retrieval methods can be used here along with BERT-style encoders. CodeBERT was shown to perform quite well; the best published performance is reported by GraphCodeBERT [**69**] (CodeBERT augmented with graph representations). We study the value of multilingual fine-tuning for both CodeBERT and GraphCodeBERT (pre-training of both models was discussed earlier in Section 4.2.1).

The Dataset: Guo *et al.* adapt the same CodeSearchNet [**89**] dataset, with some additional data for candidate codes [**69**]. Note that it is basically the same dataset we used for code summarization except the candidate codes.

Model & Fine-tuning We use Guo *et al.*'s GraphCodeBERT model, which at the time of submission is the best performing model with code and parameters available, and so is fine-tunable. The fine-tuning data is code (PL) matched with (NL) comments, from CodeXGLUE. The pre-trained GraphCodeBERT embedding vector is calculated for each PL and NL part. During fine-tuning, Guo *et al.* take a minibatch of (say n) NL query vector, along with n (correct answers) PL answer vectors. n^2 dot products are calculated; the embedding vectors are then full-stack trained to give "1" normalized dot product for the matches, and "0" for the mis-matches. For the actual retrieval, GraphCodeBERT calculates the vector embedding of a given query, and simply retrieves candidates ranked by the dot-product distance from the query vector.

4.2.4. Method Name Prediction. *The Task* as introduced by Allamanis *et al.* [**14**] as the "extreme summarization" problem, the task is to predict the function name given the body.

The Dataset: We adapt the CodeXGLUE dataset by extracting the function name and asking the model to find the name given the function body. Following [**14**], the function names are broken into subtokens using BPE [**175**] (we've used BPE tokenization for all tasks). This problem then becomes very similar to code summarization.

Model & Fine-tuning Previously Code2Seq [**15**] and Code2Vec [**16**] have worked on this problem. All prior works [**14**, **15**, **16**] use a mono-lingual datasets, which are not suitable for our experiment.

We use the same model we used for summarization, except we now learn to sequentially generate the method name, subtoken by subtoken. We use F1-score for the evaluation. For example, the function name "createLocal" is broken into two sub tokens (*i.e.*, create and Local), and the model predicts only "create". Hence, the precision, recall, and F1-score are 1.0, 0.5, and 0.66, respectively.

4.3. Results

In this section, we evaluate multilingual fine-tuning for the baselines for the tasks enumerated above.

4.3.1. Code Summarization. We apply multilingual fine-tuning on the CodeXGLUE dataset. We first *replicate* the summarization task by (monolingually) fine-tuning the available pre-trained CodeBERT model for six languages[7]. We replicate the fine-tuning stage for 2 reasons:

(1) We want to account for any hardware or environmental bias (*e.g.*, we have a different set of GPUs than the original paper. We fine-tune with NVIDIA TITAN RTX, while Feng *et al.* [**64**] use NVIDIA Tesla V100).

(2) We use a pairwise two-sample statistical test (as described in [**169**], it is more precise than just comparing test-set summary statistics) to gauge differences. This requires a performance measurement for each test sample, which the repository did not include.

Our BLEU-4 numbers for monolingual training were close to reported numbers, with some differences; but we do obtain the same overall score (17.83) (table 4.5, leftmost 2 columns).

We use the same, per-language test sets to compare monolingual and multilingual fine-tuning. The validation set, however, is a single multilingual one combining all the monolingual validation sets. Table 4.5 shows that multilingual fine-tuning improves performance, even for high-resource languages (with more than 100K training instances). With CodeBERT, multilingual fine-tuning gains 2.5%-17.5% over monolingual fine-tuning, for all languages, yielding a 6.90% overall improvement (4.48% weighted improvement)[8]. With the more advanced GraphCodeBERT, we see smaller gains, although the relative gains span a wide range.

[7]We use the publicly available CodeBERT implementation and dataset, `https://github.com/microsoft/CodeXGLUE/tree/main/Code-Text/code-to-text`

[8]The CodeBERT paper simply averages the BLEU across languages to report the "overall" number; our *weighted average* weights each BLEU by the number of samples in that language.

Language	CodeBERT (reported)	CodeBERT (re-trained)	*Polyglot*CodeBERT	Improvement	Effect Size	p-value (adjusted)	GraphCodeBERT	*Polyglot*GraphCodeBERT	Improvement	Effect Size	p-value (adjusted)
Ruby	12.16	12.53	14.75	+17.72%	0.055	<0.001	12.62	**14.95**	+18.46%	0.055	<0.001
JS	14.90	13.86	**15.80**	+14.00%	0.016	<0.001	14.79	15.79	+6.76%	0.016	0.014
Java	17.65	18.72	**20.11**	+7.43%	0.016	<0.001	19.22	19.91	+3.59%	0.016	<0.001
Go	18.07	18.15	18.77	+3.42%	0.010	<0.001	18.40	**18.92**	+2.83%	0.010	<0.001
PHP	25.16	25.48	**26.23**	+2.94%	0.012	<0.001	25.45	26.15	+2.75%	0.012	<0.001
Python	**19.06**	18.25	18.71	+2.52%	0.022	<0.001	18.02	18.90	+4.88%	0.022	<0.001
Overall	17.83	17.83	19.06	+6.90%	0.016	<0.001	18.08	**19.10**	+5.64%	0.016	<0.001
Overall (weighted)	Not Reported	19.85	20.74	+4.48%			19.98	**20.76**	+3.90%		

*Evaluation criteria followed by CodeXGLUE [**134**] and CodeBERT [**64**]

TABLE 4.5. *Effectiveness of multi-lingual fine-tuning for code summarization task. Note that p-values are B-H corrected*

We use a one-sided (AH: monolingual ¡ multilingual) pairwise Wilcoxon signed-rank test (thus avoiding the corpus-level measurement pitfalls noted in [**169**]). Null hypothesis is rejected for all six languages, for CodeBERT. For GraphCodeBERT, it's rejected overall, and for every language; except for Javascript, where the p-value is 0.014 (all after B-H correction).

Thus our measurement indicates that multilingual fine-tuning provides a statistically significant improvement over monolingual training. We find rather low effect sizes using Cliff's Delta [**137**]. While we report the effect size for the sake of completeness, this is not a major concern: we note that *all gains are statistically highly significant.* We also emphasize that even the minor improvements provided here by multilingual training (which is broadly compatible with a range of settings) constitute a relevant and potentially widely useful result. Roy *et al* [**169**] have previously noted that small gains in BLEU-4 may not be perceptible to humans as increased text quality; nevertheless, we note that natural language translation (which is now widely used) attained high performance levels based on decades of incremental progress; this result and others below provide evidence that multilingual training could be an important step in the progress towards more useful automated tools. Finally, we note that BLEU-4 gains are higher for low-resource language (*e.g.*, 17.7% for Ruby), and lower for high-resource languages (*e.g.*, 2.5% for Python), as expected.

Comparing Multi-lingual CodeBERT with Other Models Code summarization is widely studied—there are many models for this task; our specific focus here is to understand if multilingual fine-tuning provides benefits, using a high-quality token-sequence model and dataset. So we focus comparisons on the papers which report performance on CodeXGLUE dataset, and use a token-sequence inductive bias: comparing against all models is beyond the scope of this paper. We compare multilingual CodeBERT (*Polyglot*CodeBERT) and GraphCodeBERT (*Polyglot*GraphCodeBERT)

with other models that have been published in peer-reviewed venues; among them, four apply pre-training strategies [**1**, **64**, **129**, **157**]. We achieve the best overall performance (table 4.6), outperforming all the models, and for four specific languages (*i.e.*, Ruby, Java, Go and PHP).

There is one other system, CoTexT [**154**] which claims (in an unpublished, non-peer-reviewed report) better performance than us for just Python [**154**], but is worse overall. We will include it for comparison once it is published in a peer-reviewed venue.

This table also provides evidence supporting the effectiveness of multilingual fine-tuning.

4.3.2. Code Search. We study the gains from multilingual fine-tuning using two pre-trained models (*i.e.*,CodeBERT & GraphCodeBERT). We multilingually fine-tune both models using the publicly available code & dataset [9]. As we did for code summarization, we re-trained the baseline models, to get performance numbers for each case in the test set (to enable pairwise two-sample testing). We use the same test sets for both monolingual and multilingual training to evaluate our approach. During the training, GraphCodeBERT uses a matrix of dimension $|query| * |candidate\_codes|$. We could not use the full merged validation set (as we did for the code summarization task) because that makes the query and candidate code sets too large; the resulting matrix could not fit on our GPU server. We used a down-sampled validation set comprising six monolingual validation sets with 10K query and 50K candidate codes each. However, we did not face any issue while testing because we did not merge the test sets.

[9]https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT/codesearch

Models	Overall	Ruby	JavaScript	Go	Python	Java	PHP
*Polyglot*GraphCodeBERT	**19.10**	**14.95**	15.79	**18.92**	18.90	19.91	26.15
*Polyglot*CodeBERT	19.06	14.75	15.80	18.77	18.71	**20.11**	**26.23**
ProphetNet-X [**157**]	18.54	14.37	**16.60**	18.43	17.87	19.39	24.57
PLBART [**1**]	18.32	14.11	15.56	18.91	**19.30**	18.45	23.58
GraphCodeBERT [**69**]	18.08	12.62	14.79	18.40	18.02	19.22	25.45
CodeBERT [**64**]	17.83	12.16	14.90	18.07	19.06	17.65	25.16
RoBERTa [**129**]	16.57	11.17	11.90	17.72	18.14	16.47	24.02
Transformer [**193**]	15.56	11.18	11.59	16.38	15.81	16.26	22.12
Seq2Seq [**184**]	14.32	9.64	10.21	13.98	15.93	15.09	21.08

Table 4.6. *Comparison to existing models, on CodeXGLUE dataset*

We report both the published values, and our replication; we need the replication to measure pairwise gains. Though CodeBERT and GraphCodeBERT both work on sequence of code tokens, GraphCodeBERT creates a rudimentary data-flow graph, once it's told the programming language.

Table 4.7 shows that multilingual fine-tuning improves the mean reciprocal rank for all languages except Go with CodeBERT. The improvement for Ruby, JavaScript, and Java are statistically significant. We found similar results for GraphCodeBERT exhibiting improvement for Ruby, JavaScript, Java, and Python; but with GraphCodeBERT both Go and PHP showed performance declines. However, overall, both showed statistically signficant improvements (p ¡ 0.001); but the improvement for GraphCodeBERT (1.54%) is lower than CodeBERT (2.74%). Finally, we note that our numbers for CodeBERT differ from the performance reported for on the CodeXGLUE leaderboard. This is because CodeXGLUE benchmark uses only Python, and is based on a restricted setting where identifier names are left out. CodeXGLUE team argues that this abstraction enables them to stress-test the generalization ability of a model; however, here we consider an unmodified setting where someone gives an natural language query and wishes to find "natural" code with variable names intact.

Language	CodeBERT (published) [69]	CodeBERT (re-trained)	*Polyglot*CodeBERT	Improvement	Effect Size	p-value (adjusted)	GraphCodeBERT (published) [69]	GraphCodeBERT (re-trained)	*Polyglot*GraphCodeBERT	Improvement	Effect Size	p-value (adjusted)
Ruby	0.679	0.677	0.732	+8.12%	0.072	<0.001	0.703	0.708	**0.738**	+4.24%	0.039	<0.001
JavaScript	0.620	0.616	0.643	+4.38%	0.034	<0.001	0.644	0.644	**0.660**	+2.48%	0.019	0.004
Java	0.676	0.676	0.697	+3.11%	0.026	<0.001	0.691	0.693	**0.710**	+2.45%	0.022	<0.001
Go	0.882	0.885	0.885	0%	-0.003	0.550	**0.897**	0.894	0.894	0%	-0.002	0.724
PHP	0.628	0.629	0.635	+0.95%	0.009	0.003	**0.649**	0.648	0.646	-0.31%	-0.002	0.904
Python	0.672	0.676	0.678	+0.30%	0.004	0.050	0.692	0.692	**0.695**	+0.43%	0.005	0.300
Overall*	0.693	0.693	0.712	+2.74%	0.013	<0.001	0.713	0.713	**0.724**	+1.54%	0.007	<0.001
Overall (weighted)	Not Reported	0.692	0.702	+1.42%			Not Reported	0.709	**0.715**	+0.80%		

*Evaluation criteria followed by GraphCodeBERT [**69**]

TABLE 4.7. *Effectiveness of multi-lingual fine-tuning for code search task. Note that p-values are BH-corrected*

4.3.3. Method Name Prediction. As for the previous two tasks, we try multilingual fine-tuning for method name prediction for CodeBERT. Here, too, we find evidence supporting the conclusion that multilingual training provides improvement for all the languages (Table 4.8). Non-parametric pairwise improvements are significant for Ruby, JavaScript, and Java. We also note observe relatively greater effect size for Ruby and JavaScript. Note that we achieve highest improvement for JavaScript because many functions therein are anonymous lambdas, since these functions have no names, they are not useful, and this diminishes available the JavaScript training

set relative to other tasks (lambdas still have summaries, and can be used for other tasks). Therefore, multilingual fine-tuning increases the dataset diversity and boosts JavaScript method name prediction performance.

Language	CodeBERT			*Polyglot*CodeBERT			F-Score	Effect	p-value
	Precision	Recall	F-Score	Precision	Recall	F-Score	Improvement	Size	(adjusted)
Ruby	0.44	0.40	0.41	0.53	0.49	**0.49**	20.59%	0.112	<0.001
JavaScript	0.30	0.24	0.26	0.45	0.40	**0.41**	59.00%	0.215	<0.001
Java	0.54	0.51	0.51	0.56	0.52	**0.52**	2.22%	0.016	<0.001
Go	0.54	0.52	0.52	0.56	0.53	0.52	1.67%	0.015	0.004
PHP	0.56	0.53	0.52	0.57	0.53	**0.53**	1.30%	0.009	0.004
Python	0.49	0.45	0.45	0.50	0.45	**0.46**	1.60%	0.011	0.002
Overall	0. 48	0.44	0.44	0.53	0.49	**0.49**	10.09%	0.024	<0.001
Overall (weighted)	0. 52	0.48	0.48	0.54	0.50	**0.50**	3.37%		

TABLE 4.8. *Effectiveness of multi-lingual fine-tuning for method naming task. Note that p-values are adjusted using Benjamini-Hochberg*

4.3.4. Two Illustrative Examples. We used the same dataset for all tasks; for illustration, we show (Table 4.9) two test instances where all the tasks show improved performance from multilingual fine-tuning. In code summarization task, the monolingual fine-tuning scores 25 BLEU-4 in Example 1. CodeBERT produces a semantically wrong comment where multilingual fine-tuning generates the semantically correct solution. Note that the BLEU-4 is 84 for the second example because of the missing period in the gold standard (BLEU-4 is case-insensitive). Multilingual fine-tuning also helps the code search problem by increasing the MRR from 0.33 (Rank:3) to 1.00 (Rank:1). We also observe performance improvement from the method name prediction task. The gold standard consists of two sub tokens (*i.e.*, set and Values), and mono-lingual fine-tuning generates three (*i.e.*, set, Array, and Value), one of them is exact match. On the other hand, multilingual fine-tuning removes the extra "Array" subtoken and produces two subtokens(*i.e.*, set and Value) resulting in the F-score 0.50. We observe a similar result in example 2. Note that like BLEU-4, our method name prediction metric is also case-insensitive.

Finding 3. Multilingual fine-tuning is likely to increase diversity and help the models perform better than those trained with smaller mono-lingual datasets, especially for low-resource languages, irrespective of the task.

4.4. Interpreting results, and Threats

In this section we consider several issues that are relevant to the observed performance of multilingual training, such as model choice, dataset duplication, performance metrics, generalization, and different training strategies for the models.

4.4.1. Does Multilingual Fine-tuning Help with Other Models? There are several models, including CoTexT [**154**], ProphetNet-X [**157**], and PLBART [**1**] which report higher performance than CodeBERT [**64**] model for the code summarization task. The models for all these tasks were fine-tuned using monolingual datasets, so we might expect that multilingual fine-tuning should improve performance. These experiments would require a substantial investment of compute energy and is left for future work. We focused on CodeBERT (and also GraphCodeBERT on some tasks). We did some preliminary experiments with multilingual fine-tuning on PLBART. In our preliminary study, did see the same gains for low-resource language (Ruby, 5% gain). However, we found a 0.55% overall loss, which is inconsistent with what we observe with *Polyglot*CodeBERT (6.90% overall improvement) & *Polyglot*GraphCodeBERT (5.64% overall improvement). More study is needed.

Finding 4. Multilingual fine-tuning could benefit a broad range of models. We find gains for CodeBERT and GraphCodeBERT, but more data is required for other models.

4.4.2. Threats: Risk of Data Duplication? Data duplication can lead to poor-quality estimates of performance, especially when data is duplicated across training & test; even duplication just within test data risks higher variance in the estimates. Allamanis finds that performance metrics are highly inflated when test data has duplicates, and advocates de-duplicating datasets, for more robust results [**13**]. Shi *et al.* also discusses the impact of duplication in code summarization task [**176**].

Sadly, there is a large amount of copied code on GitHUB [**132**]; inattentively combining different datasets harvested from GitHUB can lead to undesirable levels of duplication in the merged dataset. Fortnuately, CodeXGLUE is atually a *carefully de-duplicated* dataset; performance estimates therein are thus more robust. Combining *multilingual* data is unlikely to introduce the same

kind of exact duplication in the dataset, because of syntax differences; There is a possibility of cross-language clones [**153**]; the study of this is left for future work.

Finding 5. Combining multilingual datasets is unlikely to cause exact duplication, because of syntax differences. More study is needed to study the effect of cross-language clones.

4.4.3. Threats: Other Metrics? Following CodeXGLUE benchmark recommendation, we evaluate the code summarization task with smooth sentence BLEU-4 [**128**] throughout this paper. However, other recognized metrics are are available (*e.g.*, ROUGE-L [**127**], METEOR [**19**]). Prior works [**67**, **169**, **176**] provide a careful analysis of the metrics, baselines, evaluations for code summarization task. Table 4.10 shows ROUGE-L and METEOR data; we find that multilingual fine-tuning increases the overall performance by 4.89% and 5.61% in ROUGE-L and METEOR, respectively. As with BLEU-4, we find that multilingual fine-tuning shows similar performance gains with these metrics. We find 0.3%-14.1% improvement in ROUGE-L and 1.2%-22.5% gains in METEOR (except for PHP, were we see a 0.17% *decline*, not statistically significant). We also see that Python shows the smallest improvement, not as strongly statistically significant. These metrics also indicate strong gains from multilingual training for low-resource and narrow-domain languages (*i.e.*, Ruby and JavaScript).

Finding 6. We observe performance improvement in all code summarization metrics with multilingual fine-tuning.

4.4.4. Monolingual Minibatches? or Multilingual? While training deep neural networks with stochastic gradient descent, gradients (multivariate derivatives of loss *w.r.t* learnable parameters) are estimated over *mini-batches*, rather than calculating loss gradients over the entire training set; these estimates are used to adjust the weights in the network. Better choices of mini-batches could improve convergence behavior. With multilingual training, a natural question arises: is it better to sequentially interperse *monolingual* mini-batches (*e.g.,* first a Java minibatch, then Ruby minibatch and so on, before going back to Java?) or should we make each minibatch *per se* multilingual?

In the previous experiments, we had randomly sort the dataset; hence, our mini-batches are also multilingual. So we deliberately tried sequentially monolingual minibatching during multilingual fine-tuning. We find that sequentially monolingual minibatch training appears to somewhat degrade

performance: we observe the overall performance goes down by 1.05%. However, the change is not statistically significant for any language. We omit the actual numerical details, for space reasons, since we didn't find any strong results in either direction.

Finding 7. We don't find any clear difference between multilingual mini batches and (interspersed) monolingual mini batches.

4.4.5. Multilingual Model as Pre-trained Model. Our findings provide evidence supporting the claim that a multilingual fine-tuned model is effective for code summarization task, which outperforms all the models trained with monolingual datasets. Could this this improved multilingual model further benefit from a secondary, *monolingual* fine-tuning exercise, where it receives specialized fine-tuning for each language separately? To evaluate this intriguing and promising idea, we load the model with the weights from multilingual fine-tuning, and fine-tune it, again, for each individual language. Table 4.11 shows that We found some minor performance improvement for JavaScript and Python. However, the performance goes down for other languages. We do not find evidence that a secondary, monolingual fine-tuning is helpful; further work is needed to understand why, and perhaps develop other ways this idea might yield further improvement.

Finding 8. We don't find evidence that applying a secondary, mono-lingual fine-tuning provides benefits for all languages.

4.5. Related work

Code Summarization: Code summarization has recently been a hot topic. More than 30 papers have been published in the last five years that follow some form of encoder-decoder architecture [**169**]. Several works [**67**, **169**, **176**] discuss the evaluations, metrics, and baselining. Roy *et al.* show that metric improvements of less than 2 points do not guarantee systematic improvements in summarization and are not reliable as proxies of human evaluation [**169**]. We find more than 2 points of improvement for Ruby and almost 2 points of improvement for JavaScript. We observe less than 2 points in other languages. It should also be noted that we *don't* use the corpus-level metrics which Roy *et al.* show is problematic; we use pairwise comparisons on the test-sets. Finally, we note that progress in both code & NLP occurs in small steps over decades, and innovations

(especially ones that could cumulate with others) such as ours can be an important part of research community's long-term pursuit of practically relevant performance improvements.

Pre-trained models [**1**, **64**, **129**, **154**, **157**] are proven to be more effective than prior models. Different pre-trained models are trained with the different pre-trained objectives even though fine-tuning steps are almost similar for all the models. As discussed earlier in Section 4.2.1, CodeBERT is an encoder model, pre-trained with MLM and Repace Token Detection objectives. Unlike CodeBERT, PLBART [**1**] is an encoder-decoder model which is trained as a denoising auto-encoder. Though all the models are pre-trained with different training objectives, there is one thing common among all the models: using Transformers as core architecture.

Parvez *et al.* very recently present an approach that augments training data using relevant code or summaries retrieved from a database (*e.g.*, GitHub, Stack Overflow) [**151**]. They apply this approach on monolingual Java and Python datasets from CodeXGLUE and claim gains over *Polyglot*CodeBERT & *Polyglot*GraphCodeBERT for code summarization. *Prima facie*, multilingual fine-tuning is complementary to their approach; this needs to be studied.

Code Retrieval and Method Name Prediction: Code retrieval is also getting attention recently. There are multiple datasets for this task. CodeXGLUE introduces a monolingual python dataset (taken initially from CodeSearchNet) abstracting the function names and variables. Guo *et al.* modify the multilingual CodeSearchNet dataset and achieve state-of-the-art performance on this task. However, using multilingual training, we show that both CodeBERT and GraphCodeBERT can be improved. There is one other very recent paper, CLSEBERT [**201**] which reports (in an unpublished, non-peer-reviewed report) better performance than us in all languages except Ruby. We could not show the effectiveness of multilingual training on CLSEBERT because the authors have not released the code implementation yet. Note that like code summarization, we focus only on the work using CodeSearchNet multilingual dataset.

CodeSearchNet dataset can be easily adapted to method name prediction task. Several earlier works address method name prediction, in a Java-only such as Code2Seq [**15**], Allamanis [**14**]. They all use a conventional single-stage machine-learning approach (no pre-training + fine-tuning). Our goal here is to simply demonstrate that multilingual fine-tuning improves upon monolingual fine-tuning for the method-naming task, so we demonstrate using CodeBERT. Our numbers are roughly

comparable with previously reported results, but we cannot make a precise comparison because of differences in subtokenization, and because our datasets are multilingual whereas previous work has largely been monolingual. We are simply arguing here our data suggests that multilingual fine-tuning is broadly beneficial in different tasks.

It would certainly be interesting to use same-domain data for fine-tuning. For example, summarizing methods in Android apps might work better if trained on Android app corpora; however curated, domain-specific datasets for each domain are needed, and may not always be possible, depending on the domain. However, cross-language data is already available, and we show that it does indeed help improve performance! The effect of domain-specific corpora is left for future work.

4.6. Conclusion

We began this chapter with three synergistic observations: *First,* when solving the *same problem*, even in different programming languages, programmers are more likely to use similar identifiers (than when solving *different* problems). *Second,* identifiers appear to be relatively much more important than syntax markers when training machine-learning models to perform code summarization. *Third,* we find that quite often a model trained in one programming language achieves surprisingly good performance on a test set in a different language, sometimes even surpassing a model trained on the same language as the test set! Taken together, these findings suggest that pooling data across languages, thus creating *multilingual* training sets, could improve performance for any language, particularly perhaps languages with limited resources, as has been found in Natural-language processing [**53**, **72**, **165**, **186**]. We test this theory, using two BERT-style models, CodeBERT, and GraphCodeBERT, with encouraging results.

Foundation models [**32**] are currently achieving best-in-class performance for a wide range of tasks in both natural language and code. The models work in 2 stages, first "pre-training" to learn statistics of language (or code) construction from very large-scale corpora in a self-supervised fashion, and then using smaller labeled datasets to "fine-tune" for specific tasks. We adopt the multilingual CodeXGLUE dataset, and the pre-trained CodeBERT and GraphCodeBERT models, and study the value of multilingual fine-tuning for a variety of tasks. We find evidence suggesting

that multilingual fine-tuning is broadly beneficial in many settings. Our findings suggest that multilingual training could provide added value in broad set of settings, and merits further study.

Example:1

//set the values from an Array

public void setValues* (Array arr) {

//we omit intermediate lines to fit in the paper

//original code here

}

Code Summarization

Models & comments	BLEU-4
Gold: set the values from an Array	NA
CodeBERT: Sets the values of the array .	25
*Polyglot*CodeBERT: Set the values from an array .	84

Code Search

Models	MRR
GraphCodeBERT	0.33
*Polyglot*GraphCodeBERT	1.00

Method Name Prediction

Models & method name	Sub tokens	F-Score
Gold: setValues	set Values	NA
CodeBERT: setArrayValue	set Array Value	0.40
*Polyglot*CodeBERT: setValue	set Value	0.50

Example:2

//Registers set injection point .

public void registerPetiteSetInjectionPoint* (final String beanName, final String property) {

//we omit intermediate lines to fit in the paper

//original code here

}

Code Summarization

Models & comments	BLEU-4
Gold: Registers set injection point .	NA
CodeBERT: Register a set of set InjectionPoint .	19
*Polyglot*CodeBERT: Register a set injection point .	60

Code Search

Models	MRR
GraphCodeBERT	0.50
*Polyglot*GraphCodeBERT	1.00

Method Name Prediction

Models & method name	Sub tokens	F-Score
Gold: registerPetiteSetInjectionPoint	register Pet ite Set In jection Point	NA
CodeBERT: addPropertyInjectionPoint	add Property In jection Point	0.50
*Polyglot*CodeBERT: setPropertyInjectionPoint	set Property In jection Point	0.57

**"registerPetiteSetInjectionPoint" & "setValues" tokens are abstracted for method name prediction task*

Table 4.9. *Examples exhibiting the effectiveness of multilingual training*

Language	ROUGE-L				METEOR			
	*Polyglot*CodeBERT	Improve.*	Effect Size	p-value (adjusted)	*Polyglot*CodeBERT	Improve.*	Effect Size	p-value (adjusted)
Ruby	24.36	+14.10%	0.087	<0.001	21.96	+22.54%	0.125	<0.001
JavaScript	24.30	+7.05%	0.022	<0.001	21.59	+11.40%	0.030	<0.001
Java	34.89	+3.32%	0.020	<0.001	31.73	+4.41%	0.020	<0.001
Go	37.36	+2.69%	0.024	<0.001	30.28	+3.73%	0.023	<0.001
PHP	38.81	+0.34%	-8.65E-05	0.508	35.52	-0.17%	-0.003	0.779
Python	32.86	+1.86%	0.015	<0.001	27.75	+1.24%	0.004	0.033
Overall	32.10	+4.89%	0.016	<0.001	28.14	+5.61%	0.013	<0.001
Overall (weighted)	34.82	+2.24%			30.52	+2.59%		

*Improvement reported over CodeBERT

TABLE 4.10. *Performance improvement in ROUGE-L and METEOR for code summarization task*

Language	*Polyglot*CodeBERT	*Polyglot*CodeBERT as pre-training	Improvement	Effect Size	p-value (adjusted)
Ruby	**14.75**	14.58	-1.15%	-0.016	0.303
JS	15.80	**16.47**	+4.24%	0.024	<0.001
Java	**20.11**	19.81	-1.49%	-0.003	0.303
Go	**18.77**	17.97	-4.26%	-0.012	<0.001
Php	**26.23**	25.52	-2.71%	-0.017	<0.001
Python	18.71	**18.83**	+0.64%	0.010	<0.001
Overall	**19.06**	18.86	-1.05%	-0.003	0.005
Overall (weighted)	**20.74**	20.43	-1.47%		

TABLE 4.11. *Multilingual model as pre-trained model*

CHAPTER 5

Towards Understanding What Code Language Models Learned

Pre-trained language models are effective in a variety of natural language tasks, but their capabilities fall short of fully learning meaning or understanding language. To understand the extent to which language models can learn some form of meaning, we investigate their ability to capture semantics beyond superficial frequency and co-occurrence. In contrast to previous research on probing models for linguistic features, we study pre-trained models in a setting that allows for objective and straightforward evaluation of a model's ability to learn semantics. In this paper, we examine whether models capture the semantics of code, which is precisely and formally defined. Through experiments involving the manipulation of code fragments, we show that code pre-trained models learn a robust representation of the *computational semantics* of code that goes beyond superficial features of form alone.

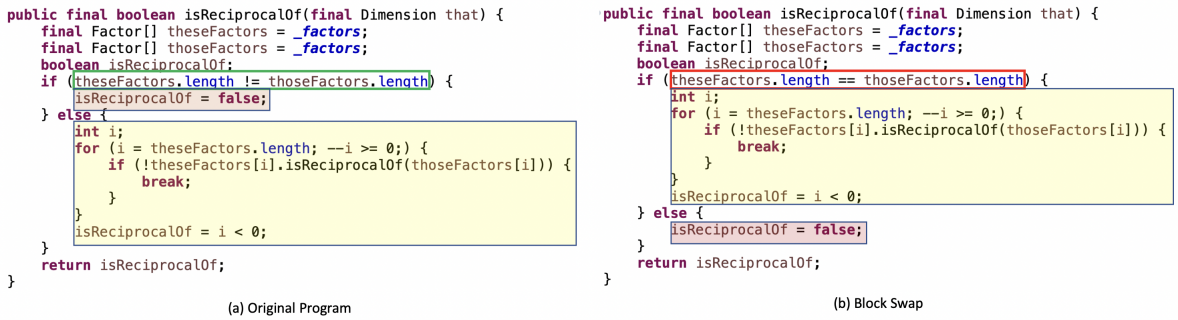


FIGURE 5.1. Semantically identical forms after transformation: Block Swap

5.1. Related Work

Language models are capable of completing various tasks such as reading comprehension [**47**, **208**]. However, model performance is lacks robustness, and so it's unclear to what extent the performance of PLMs can be characterized as understanding of the input. For example, PLMs are known for not properly handling negation and other changes to the intent of the input [**62**, **110**].

Moreover, adversarial triggers [**100**,**195**] can determine the model output independent of the actual context. This suggests that PLMs lack understanding of the "meaning" of input texts. Like NLP, researchers in SE domain have also become interested in studying what pre-train models learn. Karmakar and Robbes designed four probing tasks (probing for surface-level, syntactic, structural, and semantic information) [**109**]. They used them to show how probes can be used to observe different code properties. Troshin and Chirkova introduced a new set of more complex probing tasks [**190**]. They also consider a more comprehensive range of pre-trained models and investigate different pretraining objectives, model sizes, and the effect of fine-tuning. Unlike these two works, we restrict ourselves from fine-tuning and directly communicating with the pre-trained models (via API) to investigate the robustness of the models' prediction.

Different from previous probing work in understanding what knowledge PLMs capture by either correlating models with known information [**187**, **211**] or extracting information from the model [**61**, **155**], we focus on studying the extent to which PLMs, for code, understand inputs. Specifically, instead of polluting the input or introducing implicit diagnosing targets [**168**], we design experiments to show whether the apparent understanding exhibited by PLMs is limited to mimicry at a superficial (lexical of syntactic) level, or whether these models capture a deeper notion of semantics.

Like the BERT-style models, the generative autoregressive models (*e.g.*, GPT-2 [**162**], GPT-3 [**36**]) may also learn the semantics of the program. However, our approach applies only to the models trained with mask language modeling objectives. BERT Model still in wide use, since VLLMS not widely available our experimental method is not suitable to study how autoregressive models model distributions over semantics.

It should be noted that various works have utilized transformations that preserve meaning to train and comprehend the behavior of models. For instance, Jain *et al.* [**95**] utilized an automated source-to-source compiler to generate functionally similar variants of a program as data and pre-trained a neural network to identify them among many non-equivalent distractors. Henke *et al.* [**80**] and colleagues employed sequences of parametric, semantics-preserving program transformations in adversarial training to develop models that can resist such adversaries. Chakraborty *et al.* [**41**] introduced unnatural forms of code using six classes of semantic-preserving transformations in

NatGen and compelled the model to generate more natural original programs written by developers in the pre-training stage. Wan *et al.* [**196**] conducted a thorough structural analysis to interpret pre-trained language models for source code, such as CodeBERT and GraphCodeBERT, from three perspectives, including attention analysis, probing on word embedding, and syntax tree induction. They discovered that integrating the syntax structure of code into the pre-training process may lead to better code representations.

The objective of our study is not to enhance the performance of the models, but rather to comprehend the knowledge that the model acquires via certain irrelevant self-supervised training. For instance, BERT models are trained to unmask randomly selected tokens, which does not contribute to the model's resilience to meaning-preserving transformations.

5.2. Methodology

This section will cover the methodology we used in conducting our experiments.

5.2.1. Models' accuracy with meaning preserving transformation. Unlike natural language, code affords *feasibly automatable* meaning-preserving transforms: in several settings, code can be rewritten into lexically and syntactically different forms, while exactly preserving meaning. The formal semantics of code allows, *e.g.*, compilers to manipulate code to change its form, without altering the computational meaning thereof[1]. In other words, the compiler for a language $\mathcal{L}$ has a built-in, robust conception of the meaning of code in language $\mathcal{L}$. This robust conception allows compilers (during optimization) to extensively modify the *form* of code, without changing its *meaning*. Thus, here we're asking the question: *can a PLM, trained purely on lexical fill-in-the-blanks language modeling tasks, capture semantics in a similar manner to compilers, which are explicitly programmed to capture programming language semantics, via meaning-preserving transforms?* Here are the two different types of meaning-preserving transformations for our experiments.

5.2.1.1. *Block Swap.* One example can be seen in Figure 5.1, where the original java `if` statement can be rewritten into the semantically identical form by flipping the `then` and `else` blocks, and also the condition. The forms are different, while the *semantics* remains unchanged. A PLM that learns a robust representation of such computational semantics should perform its primary

[1]This property enables effective *metamorphic* testing [**44**] of compilers.

completion task correctly, regardless of the lexical or syntactic form of this computation, especially if the computation is very common. In the case of a BERT-like PLM, the primary task is masked language modeling (MLM): *viz.,* fill in a masked token. Thus in the example of the original program in Figure 5.1, if the comparison operator != is masked, a well-trained PLM should guess it correctly if it can determine that one specific computation is preferred over others resulting from alternatives that are equally valid from a syntactic perspective. This preference could well be a reflection of frequency. Now, if the PLM were calculating a *robust* representation of the computation, if the operator were masked in the transformed (block swap), *semantically identical* form, it should guess the == token.

5.2.1.2. *Operand Swap.* Figure 5.2 presents another type of meaning-preserving transformation. Unlike block swaps, we did not change the order of the statements; instead, we switch the positions of the operands used for binary operators. If we swap the positions of "a" and "b" for this expression, $a > b$, we have to update the operator ">" to "<" to preserve the meaning.

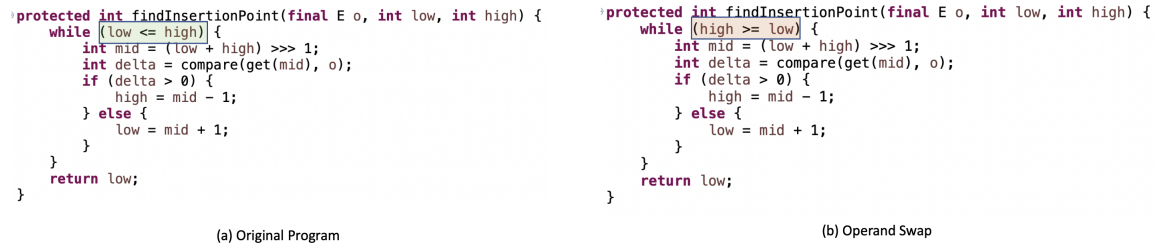


FIGURE 5.2. Semantically identical forms after transformation: Operand Swap

A variety of such transforms are possible: more generally, assume a token (operator, identifier, etc) ω in a program P. When this program form P is transformed into a semantically equivalent form P^T, let us assume that the ω token in P becomes a token ω^T in P^T. Given a PLM $\mathcal{M}$, we check that when ω is masked out in P, $\mathcal{M}$ is able to correctly guess it; in addition, if $\mathcal{M}$ were able to robustly capture the computational semantics of P, it should also recover ω^T in P^T, were that token masked.

5.2.2. Robustness of Model Representation. In this section, we evaluate factors affecting how robustly the model appears to be learning the computational meaning of the code. We consider various potential factors that might be relevant, including: the influence of variable naming; the

```
if (i < n) {
    balance = balance * (1 + interest);
}
else {
    return balance;
}
```
(a) Original Program

```
if (i >= n) {
    return balance;
}
else {
    balance = balance * (1 + interest);
}
```
(b) Equivalent Program

```
if (i < n) {
    return balance;
}
else {
    balance = balance * (1 + interest);
}
```
(c) Non-Equivalent Program

FIGURE 5.3. Semantically equivalent and non-equivalent pairs of program snippets. We found that the embedding vector distance between the equivalent pairs is lower than the nonequivalent pairs.

code context that allows the model to learn the meaning; and the effect of refactoring conditions. Finally, we examine the geometry of the internal embedding space, to check if similarity in meaning translates to closeness in vector space.

5.2.2.1. *Consistent Variable Renaming.* Apart from block swap and operand swap, we also evaluated the performance of the GraphCodeBERT model with consistent variable renaming. Renaming the locally declared variables does not change the program's semantics but certainly leads to a lexically very different program. Figure 5.4 presents an example where we replace the "reference" and "ctx" variables with var1 and var2 and perform the block swap and operand swap evaluation following the approach we mentioned in Section 5.3.3.3. We only change the locally declared variables, because renaming other variables may change the semantics on the program; our goal here is to evaluate whether PLMs can reconstruct programs of the same meaning. We discuss our findings in Section 5.3.3.3.

```
InternalContext enterContext1 ( ) {
    Object [ ] reference = localContext . get ( ) ;
    if ( reference == null ) {
        reference = new Object [ 1 ] ;
        localContext . set ( reference ) ;
    }
    InternalContext ctx = ( InternalContext ) reference [ 0 ] ;
    if ( ctx == null ) {
        reference [ 0 ] = ctx = new InternalContext ( options , reference ) ;
    }
    else
    {
        ctx . enter ( ) ;
    } return ctx ;
}
```
(a) Original Program

```
InternalContext enterContext ( ) {
    Object [ ] var1 = localContext . get ( ) ;
    if ( var1 == null ) {
        var1 = new Object [ 1 ] ;
        localContext . set ( var1 ) ;
    }
    InternalContext var2 = ( InternalContext ) var1 [ 0 ] ;
    if ( var2 == null ) {
        var1 [ 0 ] = var2 = new InternalContext ( var3 , var1 ) ;
    }
    else {
        var2 . enter ( ) ;
    }
    return var2 ;
}
```
(b) obfuscated Program

FIGURE 5.4. Impact of variable renaming on the performance of GraphCodeBERT model in block swap transformation. Results show that the model's performance does not degrade that much with obfuscation

5.2.2.2. *Context length and direction.* Language models use embeddings of context to make their predictions. If predictions are robust to changes in form, which part of the context matters in determining the operators? To determine the operator in an- `if` statement, a human developer

needs to see the following code (or code written after the conditional statement) to select the right operator. The code *before* the `if` statement should have much less influence. Is this also true for the model? In this study, we gradually increase the context on both sides (before and after the conditional statements) and observe the models' output. We compare it with results we get only using the after context. Of course, it's possible that previous tokens may also help maintain the program semantics (*e.g.*, method name, identifiers)

5.2.2.3. *Refactoring of Conditional Statement.* We consider a *Refactoring* transformation, to see if this affects the performance of the model[2]. Figure 5.5 presents an example showing how we refactored the conditional statement, by introducing a boolean variable. Note that it is not trivial to change the position of the conditional statement one has to ensure that such a refactoring step, leaves the semantics the same. We declared a boolean variable condition, assigned the conditional statement to the variable, and replaced the statement with that variable. After transformation, we applied a block swap operation on the function and evaluated the model on the original and block-swapped versions.

5.2.2.4. *Distance in Embedding Space.* We also study how program transformations affect the representation space. Specifically, we create two versions of programs from the original function. One preserves computational meaning (i.e., block swap), and the other one is superficially more similar (smaller edit distance) but is semantically non-equivalent (See example in Table 5). We take the embedding of the CLS token in the last layer as the program representation and report the euclidean distance between the two swap versions and the original. If PLMs are capturing meaning robustly in representation space, we might expect that *semantically* identical programs are closer in representation space. We discuss the result in Section 5.3.3.5.

5.2.3. Confidence *vs.* Robustness. PLMs, like other language models, have varying degrees of confidence when they predict a token. This can be measured using negative log-likelihood, which is analagous to entropy[3]. The more confident a model is on a prediction, the higher the probability, and the lower this value. We were interested to understand how *confidence*, measured as negative log-likelihood, which we informally refer to as entropy below affects the correctness of it's prediction.

[2]We thank Aryaz Eghbali of the Uni. of Stuttgart, for this suggestion.

[3]Entropy is the expected value of negative log-likelihood, typically estimated by averaging negative log-likehood over a test sample.

Transformation	Model	Accuracy			Entropy	
		Original	Transformed	Both	Original	Transformed
Block Swap	RoBERTa (NLP)	48.12%	29.01%	1.88%	1.43	1.78
	CodeBERT	84.64%	63.65%	59.04%	0.55	1.43
	GraphCodeBERT	87.97%	68.34%	65.19%	0.45	1.39
Operand Swap	RoBERTa (NLP)	52.67%	13.31%	10.77%	1.21	1.75
	CodeBERT	93.30%	87.18%	86.02%	0.24	0.48
	GraphCodeBERT	95.18%	91.38%	89.85%	0.17	0.35

TABLE 5.1. Performance of model prediction on the original and transformed programs. Results show that the PLMs can still predict the operators accurately on both block and operand swap despite higher entropy. This indicates that the model relies on semantic meaning rather than simple frequency (signified by entropy).

It's not always the case that higher confidence results in more correctness; Zhou et al [**226**] for example, report inverse relationships between *confidence*, and *more accuracy* in answers on Q&A datasets generated by GPT-3. A well-calibrated model should be *more right, when it's more confident*. To evaluate this, we consider four cases:

(1) ORTR (original right & transformed right): The model successfully predicts both the original and transformed operator.

(2) ORTW (original right & transformed right): The model successfully predicts the original operator but fails to predict the transformed operator.

(3) OWTR (original wrong & transformed right): The model successfully predicts the transformed operator but fails to predict the original operator.

(4) OWTW (original wrong & transformed wrong): The model fails to predict the original and transformed operator.

The model's robustness gradually decreases from top to bottom in the above list. For example, if the model can achieve a higher number of examples in ORTR category, that indicates that the model is more robust to meaning-preserving transformation. We observed the entropy in each category and examined if the entropy of the operator connected to the robustness of the model.

5.3. Experiments and Results

We now present the experimental setup we used, and some of our results and analysis.

5.3.1. Dataset and experiments. We implemented two transforms in our experiments to check the robustness property, `Block Swap`, and `Operand Swap`. For block swap, we parsed programs into abstract syntax tree (AST)[4] form, and swapped the "then-else" clause nodes in the tree; to preserve meaning, the conditional operator therein will also have to be changed. Similarly, for operand swap, we swap the variable names of the statement clause. We apply operand swap to all conditional statements (*e.g.,* in both loops and conditionals), while we only apply transformation to if-else statements with block swap. We show examples in Figure 5.1 and 5.2 for illustration.

We use the Java code in the test set of CodeSearchNet [**89**] for our experiments; CodeSearchNet is a properly de-duplicated dataset following the approach proposed by Allamanis [ref]. The training split of the dataset was used to pre-train the GraphCodeBERT model. To prevent exact data leakage from training to testing, we only use the test split of the CodeSearchNet dataset for our semantic transformations. Note that we also apply several transformations (variable renaming and refactoring of the conditional statement) to the program. The resulting code is un-natural, and that reduces the likelihood of the exact test data being present in the original pre-training dataset (as demonstrated by Casalnuovo *et al* [**39**]). We gather 1,425 examples for block swap and 9,377 examples for operand swap. To guarantee meaning preserving transformation, we did not include any condition where both operands make function call.

Specifically, we choose the programs where there are conditions that we can apply block or operand swap. For all the experiments, the task is to predict the masked operator (i.e., ==, ! =, <, >, <=, >=).

We experiment with one LM trained on natural language (RoBERTa [**130**]) and two PLMs trained on code (CodeBERT [**64**] and GraphCodeBERT [**68**]). In particular, CodeBERT adapts MLM and replaced token detection (RTD) as the training objective, while GraphCodeBERT also considers simplified dataflow (including edge prediction and node alignment) in addition to MLM. We report the model prediction accuracy before and after the transformation on Java, as well as the accuracy when the model predicts both forms correctly.

5.3.2. Models under Consideration. Widely-used pre-trained models, including BERT, RoBERTa, and GPT, are based on Transformer models and trained with various pre-training

[4]We use `https://github.com/tree-sitter/tree-sitter`

objectives. The pre-trained models go through two stages, where the model is pre-trained on a large, unlabeled corpus before being fine-tuned on a smaller labeled dataset for specific tasks. This allows for training high-capacity deep-learning models despite the limited availability of labeled data. The following subsections will briefly overview models used in the experiments. Note that we could evaluate the models with enabled “”fill-mask”” API.

5.3.2.1. *RoBERTa.* BERT was the first model to introduce pre-training as a strategy, which surpassed traditional Transformer models. It used two pre-training techniques: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). MLM involves randomly masking out 15% of the tokens and asking the model to predict them, while NSP trains the model to predict the next sentence following an input sentence. RoBERTa, proposed by Liu et al. [**130**], improved on BERT's performance by implementing a few changes, such as dynamic masking and dropping NSP. As a result, it achieved better results and is used as the NLP baseline model.

5.3.2.2. *CodeBERT.* CodeBERT [**64**] is similar in structure to the RoBERTa model and utilizes two pre-training objectives: MLM and Replaced Token Detection (RTD). RTD involves two data generators, NL and PL, generating possible replacements for a set of randomly masked positions, which the model is then trained to classify as either the original word or a replacement. CodeBERT was pre-trained on the CodeSearchNet dataset. Note that we could not use the original CodeBERT model because that model is not easily programmable. We used an alternative CodeBERT version “CodeBERT-mlm”, pre-trained with only MLM objective.

5.3.2.3. *GraphCodeBERT.* GraphCodeBERT [**68**] enhances source code with data flow during the pre-training process. It uses a simple data flow graph (DFG) to represent the relation between variables in terms of where their values come from. The DFG nodes are variable occurrences, while the edges represent value flow. GraphCodeBERT is pre-trained with three objectives (Edge Prediction, Node Alignment, and MLM) on 2.3 million functions (PL-NL pairs) from the CodeSearchNet dataset. It learns a joint representation of the DFG structure, DFG alignment with source code, and the source code token sequences. The pre-training and fine-tuning approach utilizes high-capacity models that are pre-trained over a large, multilingual corpus. Hence, the models already have extensive knowledge of each language even before fine-tuning.

5.3.3. Results and Analysis. In this section, we will analyze the outcomes of our experiments and discuss our findings.

5.3.3.1. *Accuracy of Block Swap and Operand Swap.* Table 5.1 shows results for operator prediction accuracy. CodeBERT and GraphCodeBERT achieve high accuracy on the original programs ($> 80\%$) and relatively high for the transformed programs ($> 60\%$) for both the block swap and operand swap, compared to a random baseline (which does a multinomial guess over operators based on frequency in the training set) of 16.67%. This suggests that the PLMs are robustly learning a realistic distribution over the *semantics* of the code, and can thus correctly perform the pre-training tasks, even when working with different forms of a program (while maintaining the same computational meaning). The higher accuracy on operand swap over block swap can be due to that other conditions might be easier to predict over if statements. Moreover, the smaller difference between the original and transformed forms indicates that PLMs are not confused by the edit distance of transformation, while the context window might be important in preserving semantics (analyzed below). The poor results of RoBERTa, which is not pre-trained specifically on code data are noteworthy; the results suggest that it lacks robustness in capturing semantics, perhaps it hasn't "seen" a lot of code.

5.3.3.2. *Impact of Context Length and Direction.* In Section 5.2.2.2, we discuss the context length and direction used by the model for prediction. Table 5.2 examines how model prediction accuracy (and entropy) vary with context length. We evaluate the performance considering the next ten tokens only after the target operator (e.g., $+10$) or together with the previous ten tokens (e.g., ± 10). Results suggest that previous tokens (such as function names and identifiers) also help maintaining the program semantics, but the majority improvement is from considering longer context after the target token for both block swap and operand swap. Note that as the amount of context increases, the entropy decreases, indicating greater model confidence. For Block swap, using the 50 following tokens we achieve 82.05% accuracy while adding 50 more preceding tokens increases the accuracy to 85.13%. Therefore, we can conclude that the preceding token has minimal impact on the models' accuracy. The importance of the following tokens to accuracy gives an additional signal that the model is utilizing the correct context for determining the correct missing operator in both transformed and original programs.

Transformation	Context	Original	Transformed	Both	Original (Entropy)	Transformed (Entropy)
Block-swap	±10	66.61%	47.96%	31.35%	1.10	1.85
	+10	67.25%	46.14%	30.71%	1.18	2.03
	±30	81.77%	60.66%	52.80%	0.66	1.52
	+30	79.52%	58.34%	50.07%	0.77	1.58
	±50	85.13%	64.10%	58.34%	0.53	1.47
	+50	82.05%	61.99%	54.63%	0.66	1.54
	Complete	87.97%	68.34%	65.19%	0.45	1.39
Operand-swap	±10	85.05%	81.70%	78.08%	0.51	0.65
	+10	78.88%	73.83%	69.32%	0.69	0.98
	±30	92.86%	89.49%	87.46%	0.25	0.41
	+30	88.19%	84.39%	80.47%	0.38	0.57
	±50	93.93%	90.14%	88.44%	0.21	0.39
	+50	89.21%	85.60%	81.89%	0.36	0.54
	Complete	95.18%	91.38%	89.85%	0.17	0.35

TABLE 5.2. Impact of context length and direction on the performance of GraphCodeBERT model in block swap and operand swap transformations. Results show that the next few tokens (e.g., +50) is critical to model performance, while considering previous tokens (e.g., +-50) provides complementary information. "complete" considers all tokens in the program.

5.3.3.3. *Impact of Consistent Variable Renaming.* Earlier in Section 5.2.2.1, we discussed the use of consistent variable renaming to evaluate the models' accuracy and robustness against meaning preserving transformations. Though the number of samples where the model successfully predicts the original and transformed operators decrease a bit (65.19% vs. 64.64% for block swap and 89.85% vs. 86.11% for operand swap), the performance and entropy of the model are very much similar to what we observed with the original version of the program. Therefore, variable renaming does not impact GraphCodeBERT's performance on both the original and transformed version of the code; the model is able to robustly capture useful statistics over semantics of programs, even after variable renaming.

The variable names we use are unhelpful and unnatural; professional developers would avoid writing such programs. The model is unlikely to have seen the transformed program in the pre-training dataset. Even so, the model fairly reliably predicts the operators, suggesting that the

Transformation	Model	Accuracy			Entropy	
		Original	Transformed	Both	Original	Transformed
Bock Swap	RoBERTa	50.04%	27.4%	1.87%	1.48	1.81
	CodeBERT	84.88%	62.33%	58.41%	0.56	1.41
	GraphCodeBERT	88.12%	67.89%	64.64%	0.46	1.39
Operand Swap	RoBERTa	52.53%	14.07%	11.44%	0.38	1.45
	CodeBERT	92.91%	83.28%	81.72%	0.26	0.71
	GraphCodeBERT	94.65%	87.54%	86.11%	0.20	0.55

TABLE 5.3. Impact of variable renaming on the performance of the models in block swap transformation. Results show that the model's performance does not degrade that much with variable renaming.

Model	Accuracy			Entropy	
	Original	Transformed	Both	Original	Transformed
RoBERTa (NLP)	13.77	7.78	0.7	2.64	2.39
CodeBERT-mlm	68.52	51.06	41.83	1.13	1.56
GraphCodeBERT	77.41	58.16	52.52	0.88	1.45

TABLE 5.4. Impact of refactoring conditional statement on model performance *in re* block-swap transformation. Model performance decreases somewhat with refactoring, but is still quite good.

model robustly learns the semantics of the program and isn't just repeating something it has seen during pre-training.

```
public final boolean isReciprocalOf(final Dimension that) {
    final Factor[] theseFactors = _factors;
    final Factor[] thoseFactors = _factors;
    boolean isReciprocalOf;
    if (theseFactors.length != thoseFactors.length) {
        isReciprocalOf = false;
    } else {
        int i;
        for (i = theseFactors.length; --i >= 0;) {
            if (!theseFactors[i].isReciprocalOf(thoseFactors[i])) {
                break;
            }
        }
        isReciprocalOf = i < 0;
    }
    return isReciprocalOf;
}
```

(a) Original Program

```
public final boolean isReciprocalOf(final Dimension that) {
    final Factor[] theseFactors = _factors;
    final Factor[] thoseFactors = _factors;
    boolean isReciprocalOf;
    boolean condition = ( theseFactors.length == thoseFactors.length ) ;
    if (condition) {
        isReciprocalOf = false;
    } else {
        int i;
        for (i = theseFactors.length; --i >= 0;) {
            if (!theseFactors[i].isReciprocalOf(thoseFactors[i])) {
                break;
            }
        }
        isReciprocalOf = i < 0;
    }
    return isReciprocalOf;
}
```

(b) Transformed

FIGURE 5.5. Impact of refactoring on the performance of GraphCodeBERT model in block swap transformation. Results show that the model's performance does not degrade that much with such operation

5.3.3.4. *How Do the Models When Condition Expressions are Refactored?* Table 5.4 shows that the model's accuracy decreases for all the models. For GraphCodeBERT, the accuracy goes down for both the original and block-swapped versions, and the model's accuracy to correctly guess both

Model	distance between non-equivalent swap	distance between equivalent swap	p-value
CodeBERT	2.51e-5	**1.80e-5**	<0.001
GraphCodeBERT	9.09e-5	**7.63e-5**	<0.001

TABLE 5.5. The impact of semantically meaning preserving transformations in the embedding space reported by the averaged cosine similarity between semantically equivalent swap and non-equivalent swap. The significant difference suggests that PLMs learn a robust representation beyond superficial features. p-value is calculated for one-sided pairwise Wilcoxon set [**209**].

operators goes down from 65.19% to 52.52%. However, 52.52% is significantly higher than random guessing. Note that refactoring the conditional statement, substantially increases the entropy of the operator, and (as shown below) the models' robustness depends on this measure.

5.3.3.5. *Semantics versus Training Frequency.* To examine whether the PLMs predict the operands by merely memorizing token frequency, rather than more robustly relying on the functional semantics (which is an important component of meaning), we report the entropy (negative log-likelihood) of the masked operands. Entropy measures the unlikelihood of a predicted token, which is influenced by its surrounding context. Table 5.1 presents the log-likelihood of the original operator and transformed operator and the log-likelihoodof the predicted operator increases significantly after the transformation (for example, doubling from 0.17 to 0.35 for GraphCodeBERT on Operand Swap). Even so the accuracy is robustly preserved (from 95.18% to 91.38%). This suggests that rather than minimizing the superficial (lexical) entropy by memorizing familiar syntax, PLMs learn to encode the computational meaning.

We also examined the geometric effect of program transformation in the vector representation space. Specifically, we create two versions of programs from the original function. One preserves computational meaning (*i.e.,* block swap), and the other one is superficially more similar (smaller edit distance) but is semantically non-equivalent (See example in Table 5.3). We take the embedding of the CLS token in the last layer as the program representation, and report the cosine distance between the two swap versions and the original. Table 5.5 shows that the distance between semantics-preserving swap is significantly smaller than that for non-preserving swap. The results suggest that PLMs assign similar representation to programs of similar meanings, rather than similar surface forms. It also justifies for the relatively high accuracy in operator prediction

after form transformation. Note that we can increase the syntactic similarity but keep the program semantically inconsistent. In those scenarios, the models may prefer the syntactically similar one. However, evaluating all possible combinations is beyond the scope of the study.

Since the transformed programs do not occur in the training data and in fact are rather different from real code scripts, our findings suggest that code PLMs learn a robust representation of computational meaning, beyond superficial features.

Transformation	Model	Transformed?	Entropy				Count			
			ORTR	ORTW	OWTR	OWTW	ORTR	ORTW	OWTR	OWTW
Block Swap	RoBERTa (NLP)	No	0.99	0.63	1.96	2.38	22	542	318	290
		YES	0.94	2.46	0.66	1.76				
	CodeBERT-mlm	No	0.06	0.41	2.58	2.67	692	300	54	126
		YES	0.10	3.88	0.41	3.31				
	GraphCodeBERT	No	0.05	0.31	2.74	2.99	764	267	37	104
		YES	0.08	4.29	0.48	3.84				
Operand Swap	RoBERTa (NLP)	No	0.42	0.57	2.03	1.96	931	3619	219	3868
		YES	1.03	2.20	1.37	1.55				
	CodeBERT-mlm	No	0.05	0.31	1.99	2.79	7431	629	100	477
		YES	0.09	3.10	0.72	3.19				
	GraphCodeBERT	No	0.03	0.34	2.02	2.95	7762	460	132	283
		YES	0.06	3.28	0.67	3.48				

TABLE 5.6. The model's success on both original and transformed operators can be predicted from the entropy of the original and transformed operators.

5.3.4. Log-likelihood Indicates Robustness. As discussed in Section 5.2.3, we consider four categories of samples (*i.e.*, ORTR, ORTW, OWTR, and OWTW); in each, we measure the average entropy of the masked operator. In all cases, lower values of operator's cross-entropy is associated with better performance of the model; In the (best performing) ORTR category, the entropy for the CodeBERT and GraphCodeBERT models ranges from 0.05 to 0.10 for both the original and transformed version of the operators in both transformations; most examples for the code-specialized models above are in this category. Compared to ORTR, for other categories, the entropy is much higher. Note that the (non-code) NLP model (*i.e.*, RoBERTa) yields higher log-likelihood even for ORTR category. Also for ORTW and OWTR category the code models relatively show low log-likelihood for the right prediction and high log-likelihood for the wrong prediction.

5.3.5. Error analysis. Table 5.7 shows model performance on the original and block swapped programs for each individual operator. We observe that the model achieves the highest F-score for

Operator	Is block swapped?	TP	FP	FN	Precision	Recall	F-score
“==”	No	547	50	19	0.92	0.97	0.94
	Yes	314	171	7	0.65	0.98	0.78
“!=”	No	301	15	20	0.95	0.94	0.94
	Yes	431	13	175	0.97	0.71	0.82
“<”	No	66	23	29	0.74	0.69	0.71
	Yes	18	67	8	0.21	0.69	0.32
“<=”	No	2	11	23	0.15	0.08	0.1
	Yes	9	2	90	0.82	0.09	0.16
“>”	No	57	25	42	0.7	0.58	0.63
	Yes	9	107	16	0.08	0.36	0.13
“>=”	No	18	16	8	0.53	0.69	0.6
	Yes	20	10	75	0.67	0.21	0.32
Overall	No	991	140	141	0.88	0.88	0.88
	Yes	801	370	371	0.68	0.68	0.68

TABLE 5.7. Operator-wise performance of GraphCodeBERT in the block-swapped program.

Operator	Is operand swapped?	TP	FP	FN	Precision	Recall	F-score
“==”	No	4081	90	116	0.98	0.97	0.97
	Yes	4119	226	78	0.95	0.98	0.96
“!=”	No	2910	62	74	0.98	0.98	0.98
	Yes	2901	94	83	0.97	0.97	0.97
“<”	No	761	80	67	0.9	0.92	0.91
	Yes	184	207	125	0.85	0.67	0.75
“<=”	No	51	46	37	0.53	0.58	0.55
	Yes	70	65	104	0.52	0.4	0.45
“>”	No	237	86	73	0.73	0.76	0.74
	Yes	558	97	270	0.85	0.67	0.75
“>=”	No	126	46	48	0.73	0.72	0.72
	Yes	10	25	78	0.29	0.11	0.16
Overall	No	8166	410	415	0.95	0.95	0.95
	Yes	7842	714	738	0.92	0.91	0.91

TABLE 5.8. Operator-wise performance of GraphCodeBERT in the operand-swapped program.

“==” (0.94) , which is also the most frequent operator in the original code. However, after transformation, the model can still predict its corresponding “!=” correctly (0.82) despite the higher frequency distribution of “==” in the training data. Meanwhile, results show a large performance

drop after the transformation for "<" and ">", where the model is confused by whether to add the equal condition (e.g., the model may predict ">" while the ground truth is ">=" after the transformation for "<") . This can also be justified by the relatively high F-score after transformation on the "<=" and ">=" operators, where the frequency distributions are relatively similar. We observed similar results with operand swap operations (Table 5.8). Apart from >= and <=s, for all the operators the F-score is more than 0.73.

5.4. Limitations

Although our experiments and analysis suggest that pre-trained language models can learn a semantic representation which robust against some meaning-preserving transforms, we acknowledge that our findings may only apply to code PLMs rather than PLMs for natural language, and that the semantics of code is different from the semantics of natural language, although parallels exist. Despite that defining "meaning understanding" is complicated in natural language, it is non-trivial to construct an equivalent experiment to our setting. We plan to expand our experiments to natural language inspired by pragmatics and psycholinguistic diagnostics [**62**,**150**] as our future work. Also, our proposed approach can only investigate the mask language model. The recent large language models (*e.g.*, GPT-3) are decoder only, and whether that auto-regressive generative model learns semantics can not be judged by our approach.

5.5. Conclusion

Unlike previous probing research using linear classifiers, we study how much pre-trained language models encode semantics for understanding beyond frequency and co-occurrence. We designed experiments in a restricted setting based on the precise and formal quality of code and found that the models learn code semantics. We also observed that semantically equivalent program pairs have the minimum distance in the embedding space compared to syntactically more similar but semantically inequivalent program pairs. Our experiments with restricted context, variable renaming, and repositioning the conditional statement strengthen our claim about the model's semantic understanding because such transformations made the program more unnatural. Still, the model can efficiently predict the original and transformed operators correctly even though the model is not pre-trained with the exact token sequence.

CHAPTER 6

Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)

Large Language Models (LLM) are a new class of computation engines, "programmed" via prompt engineering. Researchers are still learning how to best "program" these LLMs to help developers. We start with the intuition that developers tend to consciously and unconsciously have a collection of semantics facts in mind when working on coding tasks. Mostly these are shallow, simple facts arising from a quick read. For a function, examples of facts might include parameter and local variable names, return expressions, simple pre- and post-conditions, and basic control and data flow, etc.

One might assume that the powerful multi-layer architecture of transformer-style LLMs makes them inherently capable of doing this simple level of "code analysis" and extracting such information, *implicitly*, while processing code: but are they, really? If they aren't, could explicitly adding this information help? Our goal here is to investigate this question, using the *code summarization task* and evaluate whether automatically augmenting an LLM's prompt with semantic facts *explicitly*, actually helps.

Prior work shows that LLM performance on code summarization benefits from few-shot samples drawn either from the same-project or from examples found via information retrieval methods (such as BM25 [**166**]). While summarization performance has steadily increased since the early days, there is still room for improvement: LLM performance on code summarization still lags its performance on natural-language tasks like translation and text summarization.

We find that adding semantic facts actually does help! This approach improves performance in several different settings suggested by prior work, including for two different Large Language Models. In most cases, improvement nears or exceeds 2 BLEU; for the PHP language in the

challenging CodeSearchNet dataset, this augmentation actually yields performance surpassing 30 BLEU[1].

6.1. Background & Motivation

Large Language Models (LLM) are transforming software engineering: these LLMs define a new class of computation engines that require a new form of programming, called prompt engineering. We first contextualise ASAP, our contribution to prompt engineering. Finally, we discuss our choice of code summarisation as the exemplary problem to demonstrate ASAP's effectiveness.

6.1.1. The LLM Tsunami Hits SE. LLMs are now widely used in Software Engineering for many different problems: code generation [**43**,**94**], testing [**106**,**123**], mutation generation [**21**], program repair [**63**, **98**, **101**, **145**], incident management [**8**], and even code summarization [**3**]. Clearly, tools built on top of pre-trained LLM are advancing the state of the art. Beyond their raw performance at many tasks, two key factors govern the growing dominance of pretrained LLM, both centered on cost. First, training one's own large model, or even extensively fine-tuning a pre-trained LLM, requires expensive hardware. Second, generating a supervised dataset for many important software engineering tasks is difficult and time-consuming. Often, neither academic nor small companies can afford these costs.

There are some smaller models swimming against the LLM riptide specifically designed for code that have gained popularity, *e.g.*, Polycoder [**212**] or Codegen [**146**]. Despite these counterpoints, we focus on LLM rather than small models, because, while small models can be fine-tuned, they don't do very well at few-shotting, which brings the advantage of being able to use just small amounts of available data. The few-shot approach is key because it brings into reach many problems, like code summarization, for which collecting sufficient, high-quality, project- or domain-specific training data to train even small models from scratch is challenging.

With few-shot learning, we do not actually change the parameters of the model. Instead, we present a few problem instances along with solutions to a model and ask it to generate the answer for the last instance, which we do not provide with a solution. When it works, few-shotting allows us to automate even purely manual problems, since generating a few samples is relatively easy. In

[1]Scores of 30-40 BLEU are considered "Good" to "Understandable" for natural language translation, see `https://cloud.google.com/translate/automl/docs/evaluate`

this paper, we experiment with the code-davinci-002 model [**43**]. We discuss models in more detail in subsection 6.2.2.

6.1.2. Prompt Engineering. Reasoning is a mental process that involves using evidence, logical thinking, and arguments to make judgments or arrive at conclusions. It is an essential component of intellectual activities like decision-making, problem-solving, and critical thinking [**86**, **158**]. In the field of natural language processing (NLP), several attempts have been made to develop models that can reason about specific scenarios and improve performance. Approaches like "Chain of thought" [**207**] and "step-by-step" [**113**] require generating intermediate results ("lemmas") and utilizing them in the task at hand. Such approaches appear to work on simpler problems like school math problems even without providing them with "lemmas" , because, for these problems, models are powerful enough to generate their own "lemmas"; in some cases just adding "let's think step by step" seems sufficient (See Kojima et al. [**113**]).

We tried an enhanced version of the "step-by-step" prompt, with few-shots, on code summarization. We found that the model under-performed (getting about 20.25 BLEU-4), lower even than our vanilla BM25 baseline (24.97 BLEU-4). With zero-shot Kojima-style "step by step" prompt, the models peform even worse. To induce the model to generate steps, and finally a summary, we framed the problem as chain of thought, *and* included few-shot samples containing both intermediate steps ("lemmas") and final comments. The reasoning is that, on the (usually challenging) code-related tasks, models need to explicitly be given intermediate "lemmas" derived from code, to be able to reason effectively about most software engineering tasks, which tend more complex and varied than school maths.

Fortunately, software engineering is a very mature research area; well-engineered tools for code analysis are available. In this paper, we aim to derive "lemmas" directly using code analysis tools, rather than expecting the models to (perhaps implicitly) derive them, during on-task performance. We directly embed this information into the prompt provided to the model, and evaluate the benefits. The information we derive and add are based on our own intuitions about the kinds of "lemmas" that developers consciously or unconsciously consider as they seek to understand and summarize code.

We do find that providing such information to models improves scores at the summarization task. It is, of course, possible that LLMs could derive this information themselves given more computation during training and inference. Nonetheless, it is simple, quick and easy to just build them into a prompt, using robust, fast analysis tools. It is worth reminding the reader that most work involving large language models (LLMs) usually uses some form of prompt engineering to boost performance. In this paper, we show that the ASAP approach, which augments prompts with code analysis products, can help models summarize code even better than previous prompting approaches.

6.1.3. Summarizing Code. Well-documented code is much easier to maintain; so experienced developers do make significant efforts to add textual summaries to code. However, outdated or misleading summary comments can occur due to the continuous evolution of projects [**33**, **65**]. Automated code summarization is thus a well-motivated task, which has attracted a great deal of attention; and considerable progress (albeit incremental, over many years) has been made. Initially, template-based approaches were popular [**59**,**73**,**74**,**167**,**183**]; however, creating a list of templates with good coverage is very challenging. Later, researchers focused on the retrieval-based (IR) approach [**59**,**73**,**74**,**167**], where existing code (with a summary) is retrieved based on similarity-based metrics. However, this promising approach only worked if a similar code-comment pair could be found in the available pool.

In recent years, the field of Natural Language Processing (NLP) has undergone a revolution with the introduction of neural models. The similarity of code summarization to Neural Machine Translation (NMT), led to research that adapted NMT approach for code summarization. Neural models are now widely used for code summarization, and numerous studies have been conducted in this area [**2**,**85**,**91**,**122**]. Some studies have combined previous approaches, such as template-based and retrieval-based approaches, using neural models [**223**], and have reported promising results. Such neural methods for NLP have vastly improved, due to the Transformer architectural style.

Until recently, pre-trained language models such as CodeBERT [**64**] and CodeT5 [**203**] performed best for code summarization. However, Large Language Models (LLMs) can outperform smaller pre-trained models in many problems; indeed, it is quite rare anymore for pre-trained models to outperform LLMs. Ahmed and Devanbu [**3**] report that LLMs can outperform pre-trained

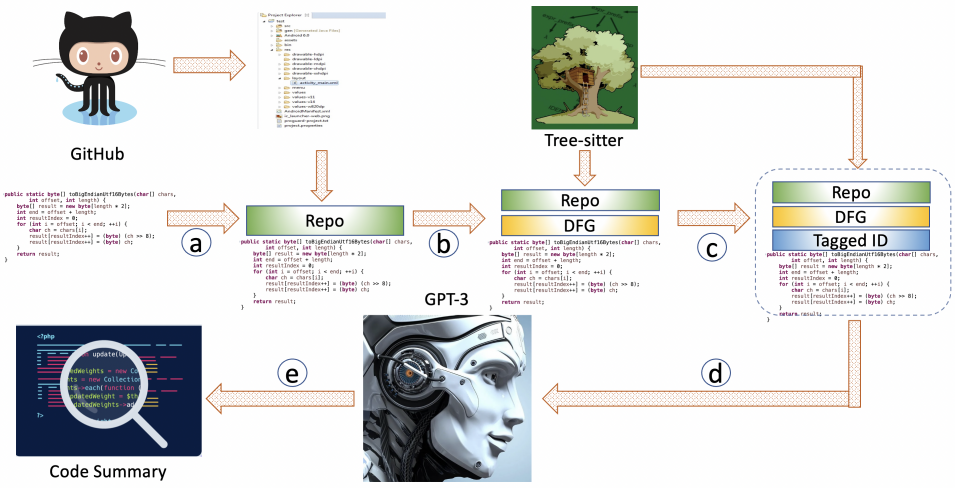


FIGURE 6.1. Different steps of prompt enhancement.

language models with a simple prompt consisting of just a few samples already in the same project; this work illustrates the promise of careful construction of prompt structures (*c.f.* "prompt engineering"). In this paper, we introduce ASAP, the general approach of Automatic Augmentation of Prompts with Semantic information. We emphasize, again, that progress in code summarization has been incremental, as in the field of NMT, where practical, usable translation systems took decades to emerge; while progress has been faster for code summarization, more advances are still needed, and we contribute our work to this long-term enterprise.

6.2. Dataset & Methodology

In this section, we will discuss the dataset, models, and methodology of our approach.

6.2.1. Dataset. We use the CodeSearchNet [**89**] dataset for our experiments: it is a carefully de-duplicated, multi-project dataset, which allows cross-project testing. De-duplication is key: Code duplication in machine learning models can deceptively inflate performance metrics by as much as 100% when measured on duplicated code datasets, compared to de-duplicated datasets [**13**, **132**, **177**].

It is part of the CodeXGLUE [**133**] benchmark, which comprises 14 datasets for 10 software engineering tasks. Many models have been evaluated on this dataset. The CodeXGLUE dataset contains thousands of samples from six different programming languages (*i.e.*, Java, Python,

Language	#of Training Samples	#of Test Samples
Java	164,923	250
Python	251,820	250
Ruby	24,927	250
JavaScript	58,025	250
Go	167,288	250
PHP	241,241	250

TABLE 6.1. Number of training and test samples.

JavaScript, Ruby, Go, PHP). However, we did not use the entire test dataset; instead, we selected 250 samples uniformly at random from each language. Since the original dataset is cross-project and we sampled it uniformly, our subsample remains cross-project. Additionally, we utilized a subset of this dataset for same-project few-shotting, following Ahmed and Devanbu [**3**]; This approach sorts the same-project data by creation date, using git blame, and selects only temporally earlier samples for the few-shot samples; this prevents any data leakage from the future to the past. We will delve deeper into this same-project dataset in Section 6.3.3.

As mentioned earlier, we don't use any parameter-changing training on the model; we just insert a few samples selected from the training subset into the few-shot prompt. Table 6.1 lists the count of training & test samples used for our experiments.

6.2.2. The Models. In earlier work, transformer-based pre-trained language models offered significant gains, in both NLP and software engineering. Pre-trained language models can be divided into three categories: encoder-only, encoder-decoder, and decoder-only models. While encoder-decoder models have initially shown success on many tasks, decoder-only LLMs are now more scaleable and effective for numerous tasks.

Encoder-Decoder model. BERT was one of the earliest pre-trained language models [**56**]; it was pre-trained using two self-supervised tasks: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). Later, RoBERTa [**129**] was introduced with some minor modifications to BERT, using only the MLM model, and it performs even better than BERT. CodeBERT [**64**] and GraphCodeBERT [**69**] introduced these ideas to Software Engineering, solving more complex problems and trained with very similar pre-training objectives. Note that although CodeBERT and

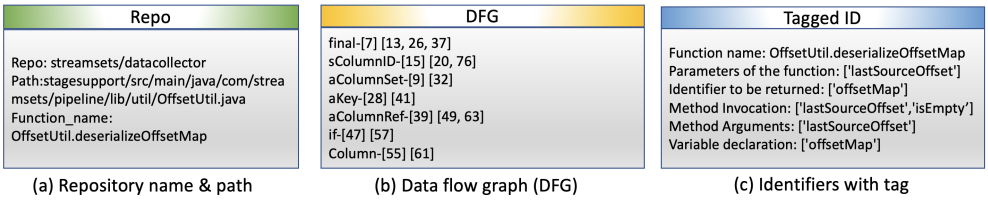


FIGURE 6.2. Different components of our enhanced prompt.

GraphCodeBERT are encoder-only models, they can be applied to code summarization after fine-tuning with an uninitialized decoder. Ahmed & Devanbu report that polyglot models, which are fine-tuned with multilingual data, outperform their monolingual counterparts [**4**]. They also report that identifiers play a critical role in code summarization tasks. PLBART [**1**] and CodeT5 [**203**] also include pre-trained *decoders* and are reported to work well for code summarization tasks. More recently, very large scale (decoder-only) auto-regressive LLMs (with 175B+ parameters) have been found to be successful at code summarization with few-shot learning, without any explicit training. In the next section, we will briefly introduce the two OpenAI models we considered for our experiments.

Decoder-only model. In generative pre-training, the task is to auto-regressively predict the next token given the previous tokens moving from earlier to later. This unidirectional auto-regressive training prevents the model from pooling information from future tokens. The newer generative models such as GPT [**161**], GPT-2 [**162**] and GPT-3 [**37**], are also trained in this way, but they have more parameters, and are trained on much larger datasets. Current Large language models, such as GPT-3, have around (or more than) 175B parameters. These powerful models perform so well, with few-shot prompting, that they have diminished the focus on task-specific parameter-adjustment via fine-tuning.

Codex is a GPT-3 variant, specifically trained on code and natural language comments. The Codex family consists of two versions: Codex-Cushman, which is smaller, with 12B parameters, and Codex-davinci, the largest, with 175B parameters. The Codex model is widely used, for various tasks. Our experiments mostly target the code-davinci model, particularly code-davinci-002, which excels at translating natural language to code [**43**] and supports code completion as well as code insertion[2]. A new variant, GPT-3.5 Turbo, is now available; unlike Codex, GPT-3.5 models can

[2]https://openai.com/

understand and generate both natural language and code. Although optimized for chat, GPT-3.5 Turbo also performs well on traditional completion tasks. We evaluate the effectiveness of our prompt enhancement in code summarization tasks using the GPT-3.5 Turbo model.

6.2.3. Retrieving Samples from Training Data. As previously discussed, few-shot learning can be quite effective, when used with models at the scale of GPT-3. We prompt the model with a small number of $< problem, solution >$ pairs, and ask it to solve a new problem. However, carefully selecting samples for few-shot learning can be very useful. Nashid *et al.* discovered that retrieval-based prompt selection is helpful for problems such as assertion generation and program repair [**145**]. Following their findings, we use the *BM25* IR algorithm to select relevant samples from the training set, for few-shot prompting. *BM25* [**166**] is a frequency-based retrieval method which improves upon TF-IDF [**164**]. We noted a substantial improvement over fixed samples in few-shot learning, as detailed in Section 6.3.1. Nashid et al. compare several retrieval methods, and we use *BM25*, which they found to work best.

6.2.4. Automatic Semantic Augmentation of Prompts (ASAP). This section presents different prompt enhancement strategies and our final ASAP pipeline (See Figure 6.2). ASAP is not tied to these analyses. Developers can easily add others as discussed at the close of this section.

Repository Name & Path. Augmenting prompts with domain-specific information can improve LLM performance on various tasks. Prior work suggests that prompts comprising information from the *same repository* can enhance performance in code generation tasks [**180**]. Even basic repository-level information, such as the repository name and the complete path to the repository, provides additional context. For example, repository names like "tony19/logback-android", "apache/parquet- mr", and "ngageoint/geo- package-android" all connect a function to a specific domain (*e.g.*, android, apache, geo-location), which can enhance the understanding of the target code to be summarized. Figure 6.2 (a) presents an example of how we enhance the prompt with repository-level information. Similar to the repository name, the path to the function can also contribute to the model.

Tagged Identifiers. Prior work suggests that pre-trained language models find greater value in identifiers, rather than code structure, when generating code summaries [**4**]. However, identifiers

do play specific roles in code. Local variables, function names, parameters, global variables etc., all play different roles in the meaning and purpose of the method in which they occur; a developer reading the code is certainly aware of the roles of identifier, simply by identifying the scope and use. Thus, providing the roles of the identifiers within the prompt might help the model better "understand" the function. We use tree-sitter to traverse the AST of the function and gather identifiers with their roles. Figure 6.2 (c) presents a sample example showing how we enhanced the prompt of the function with tagged identifiers. Although the model has access to the token sequence of the code, and thus also all the identifiers, these in a classified form might a) save the model some compute effort, and b) help focus the model's conditioned prompt generation better.

Data Flow Graph (DFG). Guo *et al.* introduced the GraphcodeBERT model, which uses data flow graphs (DFG) instead of syntactic-level structures like abstract syntax trees (ASTs) in the pre-training stage [**69**]. They conjectured that data flow presents a semantic-level structure of code that encodes the relationship of "where-the-value-comes-from" between variables. GraphcodeBERT outperformed the CodeBERT [**64**] model in various software engineering (SE) tasks. We incorporate this DFG information into the prompt; we conjecture that this provides the model a better semantic understanding of the examples. Figure 6.2 (b) presents a sample showing the Data Flow Graph (DFG) we used for our experiments. Each line contains an identifier with its index and the index of the identifiers to which that particular data flows. Note that unlike repo and tagged identifiers, the data flow graph can be very long, making it inconvenient to add the complete data flow to the prompt. In the case of long prompts, we only kept the first 30 lines of the DFG in the prompt. In addition to identifiers, the DFG also provides a better understanding of the importance of identifiers in the function.

Use Case & Completion Pipeline. To deploy ASAP, we envision realising it as a function, which we eponymously name *ASAP*, that takes a function definition as input. *ASAP* must be equipped with program analyses, and given an LLM to query and told where to find that LLM's training data. A configuration file specifies these inputs. Once configured, a developer can invoke it on a function definition. Once invoked, *ASAP* first feds the function definition to BM25 over the LLM's training data to get a result set of exemplars, which, in our context, are relevant function definition with function header comments. It then applies program analyses to its input and the exemplars

found by BM25. It constructs code summarization prompt from the results of those analyses, its BM25 queries and the input function definition. *ASAP* then queries an LLM with that prompt, and returns the natural language summarisation. A developer would apply *ASAP* to a function definition and use its output as the function's header comment for documentation.

ASAP's configuration file specifies the program analyses it applies to an input function definition and its exemplars. By default, *ASAP*'s come configured with analyses that extract repository info, tag identifiers, construct DFGs. These analyses are independent and label their additions separately. For example, Figure 6.2 (b) show the output of the DFG analysis in *ASAP*'s constructed prompt.

These few shot examples, are augmented and inserted into the prompt: the code, repository info, tagged identifiers, the DFG, and the desired (Gold) summary are all included in each few-shot. For the test example, naturally, the desired summary is omitted; ASAP thus provides the LLM with a lot of additional information. In prior work using "chain of thought" [**207**] or "step by step" [**113**] reasoning, no such information is given to the model, and we simply help it organize its reasoning about the sample with some instructions. Here, rather than having the model do its own reasoning, we are providing it externally using a simple program analysis tool, since we can get very precise information from very efficient analysis tools. Each few-shot example includes source code, derived information, and conclusion (summary), thus providing exemplary "chains of thought" for the model to implicitly use when generating the desired target summary. Figure 6.1 presents the overall pipeline of our approach that we apply to each sample.

Next we describe how we evaluate this pipeline.

6.2.5. Experimental Setup & Evaluation Criteria. Our primary model is OpenAI's code-davinci-002. We access the beta version, through the web service API. Based on the rate limits, while also desiring robust estimates of performance, we chose to use 250 samples per experimental treatment (one treatment for each language, each few-shot selection approach, with ASAP, without ASAP etc.). While higher sample sizes would be nice, we had sufficient statistical power to get interpretable results. Each 250-sample trial still took 2 to 5 hours, presumably varying with OpenAI's load factors. We includes waiting periods between attempts, following OpenAI's recommendations. To obtain well-defined answers from the model, we found it necessary to set the temperature to

Language	CodeBERT	GraphCodeBERT	Polyglot CodeBERT	Polyglot GraphcodeBERT	CodeT5	Few-shot (random)	Few-shot with BM25	Gain (%) over random few-shot
Java	19.28	19.32	20.13	19.65	20.86	21.14	**24.97**	18.12%
Python	18.48	17.87	18.54	18.03	20.87	21.31	**22.43**	5.26%

TABLE 6.2. Performance of encoder-decoder and few-shot models on Java and Python code summarization.

0, for all our experiments. The model is designed to allow a window of approximately 4K tokens; this limits the number of few-shot samples. For our experiments, we used 3 shots. However, for up to 2% of the randomly chosen samples in each experiment, we didn't get good results; either the prompt didn't fit into the model window, or the model mysteriously generated an empty string. In cases where the prompt as constructed with 3 samples was too long, we automatically reduce the number of shots. When empty summaries were emitted, we resolved this by increasing the number of shots. This is simple repeated procedure can be incorporated into automated summarization tools with a modest overhead.

6.3. Results

We now report the benefits of ASAP-enhanced prompts, for code summarization, in different settings and using various metrics. We do find evidence of overall performance gain, in studies for six languages. However, for other detailed analyses, we focused primarily on Java and Python, because of OpenAI API rate limits.

6.3.1. Encoder-decoders & Few-shot Learning. Following a lot of prior work, we use CodeXGLUE [**133**] as a benchmark dataset to evaluate our approach of ASAP-enhanced few-shots. Of course, using better samples improves few-shot performance. Prior work suggests that IR methods can find better samples for few-shot prompting, for tasks such as program repair [**145**] and code generation [**94**]. In Table 6.2, we observe that this is also true for code summarization. We observed an improvement of 3.83 (18.12%) and 1.12 (5.26%) in BLEU-4 score for Java and Python, respectively, simply by using *BM25* as a few-shot sample selection mechanism. Since *BM25* was already used in prior paper (albeit for other tasks) [**145**], we consider this *BM25*-based few-shot learning for code summarization as just a baseline (not a contribution *per se*) in this paper.

6.3.2. Prompt Enhanced Few-shot Learning. We now focus on the effect of our proposed ASAP semantic prompt-enhancement. Table 6.3 shows the element-wise and overall improvements

achieved after combining all the prompting elements for all six programming languages. We observed BLEU improvements ranging from 1.85 (7.41%) to 5.42 (20.93%). Regarding magnitude of improvement: Roy *et al.* suggest that a measured improvement of more than two BLEU may be perceptible to humans [**169**]; in most cases, we see such an improvement. We also noticed that all three components (i.e., *Repo*sitory Information., *DFG* Data Flow Graph, *Id*entifiers) help the model achieve better performance in all six languages, as we combined these components individually with *BM25*. However, for Python, the best performing combination doesn't require all three components: just Repo. information gives the best results. In most cases, incorporating Repo. helps a lot, in comparison to other components.

To measure the significance of our contributions, we performed a pairwise one-sided Wilcoxon signed-rank test and found statistical significance in all cases, for our final prompt when compared with vanilla BM25 few-shot learning, even after adjusting for false discovery risk.

Language	BM25	BM25+repo	BM25+id	BM25+DFG	ASAP	Comparing with BM25	
						Gain (%) over BM25	p-value
Java	24.97	25.68	25.37	25.40	**26.82**	+7.41%	<0.01
Python	22.43	**24.93**	23.67	21.79	24.56	+9.50%	<0.01
Ruby	17.40	21.16	20.66	18.50	**21.19**	+21.78%	<0.01
JavaScript	23.36	25.61	24.82	24.16	**26.04**	+11.47%	<0.01
Go	21.62	24.05	22.64	23.19	**24.31**	+12.44%	<0.01
PHP	25.90	30.37	27.35	26.30	**31.32**	+20.93%	<0.01

TABLE 6.3. Performance of prompt enhanced comment generation with code-davinci-002 model. p-values are calculated applying one-sided pair-wise Wilcoxon signed-rank test and B-H corrected.

Language	Project Name	#of training sample	#of test sample	Cross-project			Same-project		
				BM25	ASAP	p-value	BM25	ASAP	p-value
Java	wildfly/wildfly	14	100	24.05	**24.77**	<0.01	17.86	18.27	<0.01
	orientechnologies/orientdb	10	100	25.54	**27.23**		19.43	20.24	
	ngageoint/geopackage-android	11	100	29.33	42.84		45.48	**46.21**	
	RestComm/jain-slee	12	100	17.04	19.06		17.99	**19.61**	
Python	apache/airflow	12	100	20.39	20.37		20.36	**20.72**	
	tensorflow/probability	18	100	**21.36**	21.18		20.30	20.86	
	h2oai/h2o-3	14	100	19.50	**20.72**		18.75	19.81	
	chaoss/grimoirelab-perceval	14	100	25.23	29.23		32.75	**38.23**	

TABLE 6.4. Performance of prompt enhanced comment generation with code-davinci-002 model on same project data and p-values are calculated applying one-sided pair-wise Wilcoxon signed-rank test after combining the data from all projects.

6.3.3. Same Project Code Summarization. Few-shot learning is especially salient for software engineering, because of the availability of pre-existing project-specific artifacts; such artifacts

can provide useful statistical context that generative models can condition upon. For example, developers often use project-specific names for identifiers, APIs, *etc*; there are also coding patterns that are specific to each project's requirements [**79**,**81**,**191**]. These practices are closely tied to the project domain's concepts, algorithms, and data. Experienced developers have prior knowledge of domain-specificities, and so can comprehend code better & faster. Naturally, these details can also provide helpful hints for machine learning models. However, project-specific data can be limited, *e.g.* in the beginning stages of a project. The capacity of LLMs to leverage even just a few shots is very helpful in such settings.

To see if our prompt enhancement idea helps in project-specific code summarization, we evaluated our approach on the dataset supplied by Ahmed and Devanbu [**3**]. Due to rate limits, we reduced the number of test samples to 100 for each of the four Java and Python projects. When working with the same project, one must split data with care, to avoid leakage from future samples (where desired outputs may already exist) to past ones. Therefore, we sorted the samples by creation dates in this dataset. After generating the dataset, we applied our approach to evaulate the performance in same project setting. We also compared our results with a cross-project setup, where we retrieved samples from the complete cross-project training set, similar to the setting used in Section 6.3.2.

Table 6.4 displays the results of our approach for project code summarization. We found that for 4 projects, cross-project few-shot learning yielded the best performance, while for another 4 projects, same-project few-shot learning was most effective. We noted that Ahmed & Devanbu didn't use IR to select few-shot samples and consistently achieved better results with same-project few-shot learning [**3**]. IR does find relevant examples in the large samples available for Java & Python, and we get good results. We analyzed 16 pairs of average BLEU-4from 8 projects, considering both cross-project and same-project scenarios. Our prompt-enhanced few-shot learning outperformed vanilla BM25 retrieved few-shot learning in 14 cases (87.5%). This suggests that ASAP prompt enhancement is helpful across projects. Since we have too-few samples for a per-project test, we combined all the samples to perform the statistical test. ASAP performs significantly better in both cross-project and same-project settings.

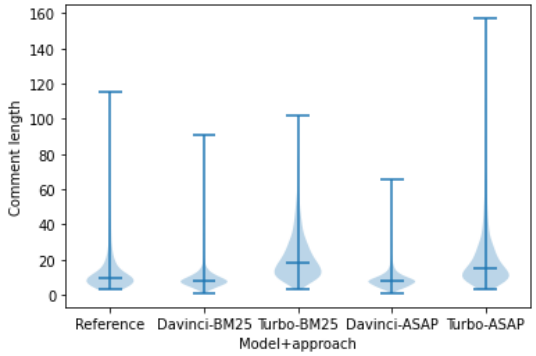


FIGURE 6.3. Length (number of tokens) distribution of the reference comment and model outputs

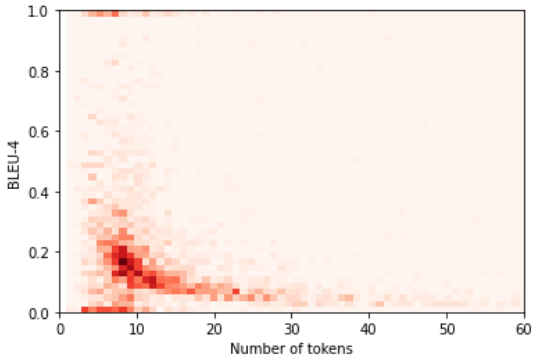


FIGURE 6.4. Length (number of tokens) vs BLEU-4 for ASAP.

6.3.4. Is ASAP Model-agnostic? Thus far, we presented the results for the code-davinci-002 models. We also applied semantic prompt augmentation to another model, gpt-3.5-turbo, which is primarily a chat model but can also be used for code analysis. Our findings are in Table 6.5. Gpt-3.5-turbo doesn't do as well as the code-davinci-002 model at code summarization. Gpt-3.5-turbo generates verbose comments that stylistically differ from comments written by developers, and the few-shot examples in the prompt. Figure 6.3 illustrates how comments generated by gpt-3.5-turbo are more verbose (longer) than both "gold" comments and those generated by code-davinci-002. Longer comments tend to achieve lower BLEU-4 (Figure 6.4). It may be possible to prompt-engineer the turbo model into generating program comments that are as natural, and brief as human-produced comments; this is left for future work.

Nevertheless, we found that our prompt-enhanced few-shot learning approach improved the performance of the gpt-3.5-turbo model by 2.47% to 25.74%. We also conducted pairwise one-sided

Language	BM25	ASAP	Gain (%)	p-value
Java	15.37	**17.55**	+14.18%	<0.01
Python	12.18	**14.18**	+16.42%	<0.01
Ruby	9.09	**11.43**	+25.74%	<0.01
JavaScript	10.51	**10.76**	+2.47%	0.25
Go	13.48	**15.66**	+16.17%	<0.01
PHP	13.56	**16.86**	+24.33%	<0.01

TABLE 6.5. Performance of GPT-3.5-turbo on code summarization. p-values are calculated applying one-sided pair-wise Wilcoxon signed-rank test and B-H corrected.

Wilcoxon signed rank tests, and the statistical significance of our findings (except JavaScript) suggests that the ASAP approach is applicable across LLMs.

6.3.5. Performance on Other Metrics. Proper performance evaluation of Neural Machine Translation (NMT), or code summarization models, is an often debated issue [**67**]! Despite the availability of several metrics, researchers have yet to reach a consensus on a standard metric for this problem. BLEU [**128**, **149**] is the most widely used metric, but there are various versions of BLEU that can yield significantly different results. For our experiments, we primarily use BLEU-CN. There are deep-learning-based metrics available for code summarization, such as BERTScore [**75**, **224**], BLEURT [**174**], NUBIA [**105**]. However, these metrics are also limited and computationally expensive. Recently, Shi et al. found that BLEU-DC better reflects human perception [**177**], so we also report the performance of our models on BLEU-DC and two other popular metrics: ROUGE-L [**127**] and METEOR [**19**].

Our results, in Table 6.9, demonstrate that ASAP improves performance on all three metrics (except for gpt-3.5-turbo model's performance on JavaScript) indicating its effectiveness across different metrics. Furthermore, we conducted pairwise one-sided Wilcoxon signed-rank tests and found that the majority of languages (for Davinci and Turbo models) exhibited statistically significant improvement with BLUE-DC and ROUGE-L. However, we did not observe significant differences with METEOR, even though we noticed improved performance with ASAP in 11 out of 12 comparisons. It's worth noting that we had only 250 language samples for each language, so it's not

Language	Prompt component	BLEU-4
Java	ALL	**26.82**
	-Repo.	25.73
	-Id	26.16
	-DFG	26.25
Python	ALL	**24.56**
	-Repo.	24.25
	-Id	24.31
	-DFG	24.25

TABLE 6.6. Ablation study.

unexpected to see some cases where we didn't observe significance. To evaluate the overall impact of ASAP, we combined the dataset from all languages for both Davinci and Turbo models (3000 samples) and performed the same test. This time, *we achieved statistical significance (p-value ¡ 0.01) for all three metrics, a finding that strongly supports the effectiveness of ASAP.*

6.3.6. Ablation Study. The primary aim of an ablation study is to identify the crucial components or features that are essential for the proper functioning of the system or model and to understand their impact on its overall behavior or performance. In our study, we removed one component of the enhanced prompt and observed its performance. We found that Repo. was the most significant contributing component to the model's performance (Table 6.6) especially for Java. However, tagged identifier and DFG also made some contribution, and the best results were obtained when we combined all three components in the prompt.

6.3.7. Two Illustrative Examples. Table 6.7 shows the results of our experiments with enhanced prompting on two samples. We computed the difference between the BLEU score of the vanilla *BM25* model and our prompt-enhanced BLEU score, and sorted the samples in descending order of improvement. We observed that in several samples, the prompted enhanced version was able to recover new information that was crucial for the summary. For instance, in the first example, the baseline model failed to generate the term "element-wise". However, our prompted enhanced version was able to capture this important concept, resulting in a higher BLEU-4 score of 0.74 compared to the baseline score of 0.39. Similarly, in the second example, the baseline model was not able to recognize the function as a standalone process, leading to a low BLEU score of

Example 1

```
def round(input_a, name: nil)
    check_allowed_types(input_a, TensorStream::Ops::FLOATING_POINT_TYPES)
    _op(:round, input_a, name: name)
end
```

Gold & model output	Comment	BLEU-4
Gold	Rounds the values of a tensor to the nearest integer element - wise	NA
BM25	Round a tensor to the nearest integer	0.39
ASAP	Rounds the values of a tensor to the nearest integer, element-wise.	0.74

Example 2

```
public static void main(final String[] args)
{
    loadPropertiesFiles(args);
    final ShutdownSignalBarrier barrier = new ShutdownSignalBarrier();
    final MediaDriver.Context ctx = new MediaDriver.Context();
    ctx.terminationHook(barrier::signal);
    try (MediaDriver ignore = MediaDriver.launch(ctx))
    {
        barrier.await();
        System.out.println("Shutdown Driver...");
    }
}
```

Gold & model output	Comment	BLEU-4
Gold	Start Media Driver as a stand - alone process .	NA
BM25	Main method that starts the CLR Bridge from Java .	0.10
ASAP	Main method for running Media Driver as a standalone process.	0.33

TABLE 6.7. Examples showing the effectiveness of prompt enhancement.

0.10. However, our proposed approach successfully identified the function as a standalone process, resulting in a higher BLEU score of 0.33.

6.4. Discussion

In this section, we discuss several issues that are relevant to our prompt component & design choices.

6.4.1. Does the Model Memorize the Path? Among the three components, it was found that the repository information has a greater impact on the model's performance. However, it is important to note that the training dataset for the code-davinci-002 model is not accessible, which means that the model may have memorized the file path during pre-training. Therefore, when we provide the path to the function, the model may use the memorized information and perform better based on prior exposure.

To investigate this further, we modified the path representation from absolute to a list. We took the repository name and path, split the tokens at "/", and presented the model with a list of tokens. The main idea behind this approach is to dissolve the original representation and present the model with something that it did not encounter during pre-training. If the model is not memorizing, its performance should not be impacted. We observed that the differences between both versions were very small. For Java, we gained 0.24 BLEU but, for Python, we lost 0.04 with tokenized paths. This indicates a reduced chance of the model memorizing the path to the function.

6.4.2. Is the Identifier Tag Necessary? In this paper, we assign roles to the identifiers and tag them as *Function Name, Parameters, Identifier* etc. in the prompt (See fig. 6.2 part c). But does this explicit tagging actually help performance? To investigate this, we compare the model's performance when provided with a plain, "tag-free" list of identifiers. We observed that the tagged identifiers lead to better performance for both Java and Python than a simple tag-free list of identifiers. Our performance metric BLEU-4 increased by 0.41 and 1.22 for Java and Python, respectively. Therefore, providing explicit semantic information does indeed contribute to better model performance.

6.4.3. What's Better: More Shots, or ASAP?. Although Large Language Models (LLMs) have billions of parameters, their window sizes are still limited. For example, code-davinci-002 and gpt-3.5-turbo support only around 4k tokens. Therefore, the length of our prompt is constrained by the model's capacity. Augmentation does indeed gobble up some of the available prompt length

Language	Prompt Enhanced		Vanilla BM25	
	#of shots	BLEU-4	#of shots	BLEU-4
			3	24.97
Java	3	**26.82**	4	24.82
			5	25.75
			3	22.43
Python	3	**24.56**	4	21.57
			5	22.18

TABLE 6.8. Comparing with higher-shots Vanilla BM25.

Language	Model	BLUE-DC				ROUGE-L				METEOR			
		BM25	ASAP	Gain (%)	p-value	BM25	ASAP	Gain (%)	p-value	BM25	ASAP	Gain (%)	p-value
Java	code-davinci-002	17.36	18.32	+5.53%	<0.01	38.62	40.74	+5.49%	0.01	37.09	38.19	+2.97%	0.24
	gpt-3.5-turbo	9.98	11.32	+13.43%	0.02	27.81	29.38	+5.65%	0.01	33.48	34.83	+4.03%	0.10
Python	code-davinci-002	12.84	14.43	+12.38%	<0.01	35.70	38.16	+6.89%	<0.01	33.47	35.35	+5.62%	0.10
	gpt-3.5-turbo	6.54	7.19	+9.94%	0.02	23.89	25.63	+7.28%	<0.01	28.86	29.88	+3.53%	0.19
Ruby	code-davinci-002	9.16	11.95	+30.45%	<0.01	28.45	32.80	+15.29%	<0.01	28.84	32.67	+13.28%	0.01
	gpt-3.5-turbo	4.63	5.82	+25.70%	<0.01	19.53	21.77	+11.47%	<0.01	26.32	27.28	+3.65%	0.18
JavaScript	code-davinci-002	15.12	16.82	+11.24%	0.02	31.97	33.80	+5.72%	0.02	31.72	32.67	+2.99%	0.17
	gpt-3.5-turbo	5.49	5.28	-3.83%	0.21	20.28	19.41	-4.29%	0.49	26.36	24.96	-5.31%	0.91
Go	code-davinci-002	16.06	18.01	+12.14%	0.03	40.21	41.70	+3.71%	0.06	35.18	35.81	+1.79%	0.48
	gpt-3.5-turbo	9.80	11.34	+15.71%	<0.01	29.20	31.44	+7.67%	<0.01	31.39	33.42	+6.47%	0.10
PHP	code-davinci-002	14.92	18.16	+21.71%	<0.01	37.72	41.09	+8.93%	<0.01	39.29	41.08	+4.56%	0.10
	gpt-3.5-turbo	6.96	8.76	+25.86%	<0.01	24.64	27.84	+12.99%	<0.01	32.03	33.75	+5.37%	0.10

TABLE 6.9. The effectiveness of ASAP in popular code summarization metrics. p-values are calculated applying one-sided pair-wise Wilcoxon signed-rank test and B-H corrected.

budget! Thus we have two design options: 1) use fewer, but Automatically Semantically Augmented samples in the prompt or 2) use more few-shot samples without augmentation. To investigate this, we also tried using 4 and 5 shots (instead of 3) for Java and Python with the code-davinci-002 model. However, Table 6.8 shows that higher shots using BM25 does not necessarily lead to better performance. With higher shots, there is a chance of introducing unrelated samples, which can hurt the model instead of helping it.

Only for Java, we observed better performance with 5 shots compared to our baseline model. However, our proposed technique with just 3-shots still outperforms using BM25 with 5 shots. It's worth noting that the context window of the model is increasing day by day, and the upcoming GPT-4 model will allow us to have up to 32K tokens[3]. Therefore, the length limit might not be an issue in the near future. However,our studies suggest that Automated Semantic Augmentation will still be a beneficial way to use available prompt length budget.

[3]https://platform.openai.com/docs/models/gpt-4

6.5. Related work

6.5.1. Code Summarization. Deep learning models have advanced the state-of-the-art in SE tasks such as code summarization. The LSTM model for code summarization was first introduced by Iyer *et al.* [**91**], leveraging early conceptions of attention [**18**]. Transformer [**193**] models have substantially impacted software engineering. Pre-trained transformer-based models such as CodeBERT [**64**], PLBART [**1**], and CodeT5 [**203**] have been extensively used on the CodeXGLUE [**133**] code summarization dataset, resulting in significant improvements. However, there is a caveat to using pre-trained language models: although these models perform well, extensive fine-tuning is required, which can be data-hungry & time-consuming. Additionally, separate models had to be trained for different languages, making it an expensive process. To reduce the number of models required, multilingual fine-tuning has been suggested, to maintain or improve performance while reducing the number of models to one [**4**]. However, this approach did not reduce the training time or the need for labeled data.

LLMs, or large language models, are much larger than earlier pre-trained models, and are trained on much bigger datasets with a simple training objective — auto-regressive next-token prediction [**37**]. A huge benefit of such models is that they perform quite well on tasks even without fine-tuning. Just prompting the model with different questions, while showing the model a few examples of how to solve the problem, is sufficient. Few-shot learning has already been applied to code summarization, and has been found to be beneficial [**3**].

6.5.2. Other Datasets. There are several datasets available for code summarization, in addition to the CodeXGLUE [**133**] dataset. TL-CodeSum [**85**] is a relatively smaller dataset, with around 87K samples, but it has duplicates and data from the same projects are spread across training, test, and validation sets, which may result in higher performance. Funcom [**122**] is a dedicated dataset with 2.1 million Java functions, but there are some repetitions on the comment side. CodeXGLUE (derived from CodeSearchNet) is a diverse, multilingual dataset that presents a challenge for models. Even well-trained initial models like CodeBERT struggle on this benchmark dataset, and performance is particularly poor for languages with fewer training samples. Note that there have been hundreds of models introduced for the code summarization problem, ranging from

template matching to few-shot learning. These models use different representations and sources of information to perform well in code summarization. However, comparing or discussing all of these models is beyond the scope of this work. We note, however, that our numbers represent a new high-point on the widely used CodeXGlue benchmark for code summarization; we refer the reader to `https://microsoft.github.io/CodeXGLUE/` for a quick look at the leader-board. Our samples are smaller (N=250), but the estimates, and estimated improvements, are statistically robust.

6.5.3. LLMs in Software Engineering. Although LLMs are not yet so widely used for code summarization, they are extensively used for code generation [**43**, **146**, **212**] and program repair [**63**, **98**, **101**]. The goal of models like Codex is to reduce the burden on developers by automatically generating code or completing lines. Several models such as Polycoder [**212**] and Codegen [**146**] perform reasonably well, and due to their few-shot learning or prompting, they can be applied to a wide set of problems. However, Code-davinci-002 model generally performs well than those models and allows us to fit our augmented prompts into a bigger window.

Jain et al. proposed supplementing LLM operation with subsequent processing steps based on program analysis and synthesis techniques to improve performance in program snippet generation [**94**]. Bareiß *et al.* showed the effectiveness of few-shot learning in code mutation, test oracle generation from natural language documentation, and test case generation tasks [**21**]. CODAMOSA [**123**], an LLM-based approach, conducts search-based software testing until its coverage improvements stall, then asks the LLM to provide example test cases for functions that are not covered. By using these examples, CODAMOSA helps redirect search-based software testing to more useful areas of the search space. Jiang et al. evaluated the effectiveness of LLMs for the program repair problem [**98**].

Retrieving and appending a set of training samples has been found to be beneficial for multiple semantic parsing tasks in NLP, even without using LLM [**222**]. One limitation of this approach is that performance can be constrained by the availability of similar examples. Nashid et al. followed a similar approach and significantly improved performance in code repair and assertion generation with the help of LLM [**145**]. However, none of the above works has attempted to automatically semantically augment the prompt. Note that it is still too early to comment on the full capabilities

of these large language models. Our findings suggest that Augmenting the input with semantic hints helps on the code summarization task; we hope that this type of prompt augmentation will prove useful for other tasks as well.

6.6. Threats & Limitations

A major concern when working with large language models is the potential for test data exposure during training. However, it is not possible to confirm this since the training dataset is not accessible. Thus there is a risk that the model may overfit, but examining its ability to solve a diverse set of problems does not indicate overfitting. Additionally, the model's performance with random few-shotting suggests that overfitting is not a major issue since the numbers are low. As we incorporate relevant information, the model's performance improves with the amount and quality of information. If the model had memorized the summaries, it would have achieved a much higher BLEU-4 without requiring the augmented prompt based on the given token sequence.

Another concern is our smaller test dataset, which we had to reduce due to API rate limits. Despite this limitation, we achieved statistical significance for each language with the Davinci model on the BLEU-4 metric, demonstrating ASAP's effectiveness. ASAP also performs well with other metrics and achieved statistical significance if we combine all the samples from different languages and models.

Fine-tuning the LLMs may yield even better results than our augmented prompting approach, but it is costly to train such a model for a specific task. With just a few samples and augmented prompts, we could easily outperform all the fine-tuned models trained with thousands of samples. We will leave the fine-tuning part for future research.

6.7. Conclusion

In this paper, we explored the idea of *Automatic Semantic Augmentation of Prompts*, whereby we propose to enhance few-shot samples in LLM prompts, by adding tagged facts automatically derived by semantic analysis. This based on an intuition that human developers often will need to scan the code to implicitly extract such facts in the process of code comprehension leading to writing a good summary. While it is concievable that LLMs can implicitly infer such facts for themselves, we conjectured that adding these facts in a formatted style to the samples and target will help

the LLM organize it's "chain of thought" as it seeks to construct a summary. We evaluated this idea a challenging, de-duplicated, well-curated CodeSearchNet dataset, and found that Automated Semantic Augmentation of Prompts is helpful in the preponderance of settings where we tried it, beyond the state of the art in few-shot prompting technique; our estimates suggest it can surpass state-of-the-art.

CHAPTER 7

Conclusion & Future Research Direction

In this chapter, we will summarize the dissertation with some discussion of possible future research directions.

7.1. Pre-trained language model: where to go from here?

Pre-trained language models have undergone numerous changes since their inception. Prior to LLM, pre-trained language models were typically smaller in size (110-220M parameters), and were easily trainable using a few medium-end GPUs. In this dissertation, we mainly focused on discussing the applicability and robustness of these models, with the exception of chapter 6. The most widely used pre-training objectives that played an important role in pre-training were masking of tokens or spans. Syntactic [**1**] and semantic denoising [**41**] were also successful for various software engineering tasks. It should be noted that although researchers proposed several models, the performance improvements of newer models were not massive compared to previous models, and in many cases, the improvements did not have any statistical significance. With the introduction of LLM, traditional pre-trained language models are receiving less attention; however, the value and importance of such models cannot be ignored. These models can be easily trained for new domains and can be applied with minimal resources.

Various pre-training objectives have been identified as the primary driving force behind the success of pre-training models. As of now, the Transformer architecture remains the most popular and widely used. Until we develop better architectures than Transformer, pre-training objectives will remain the major factor affecting performance. Incorporating new pre-training objectives is relatively easier than introducing new architecture. However, to achieve even better results, we may have to go beyond masking or spanning objectives and incorporate more contextual information into the models to help them better understand the problem and improve their performance.

7.2. Emergence of LLMs and its Implications

Large Language Models (LLMs) are decoder-only models that have had a significant influence on both the NLP and SE domains. LLMs are large-scale models that are pre-trained with over 175 billion parameters and work very well with prompt engineering and few-shot learning. They eliminate the need for supervised fine-tuning with datasets of hundreds of thousands of samples. These models have proven useful in various tasks, including code summarization, test case generation, and code generation. Furthermore, these models are not only adept at old tasks, but they also offer an opportunity to work on new problems, as we do not need as much labeled data to begin with.

While large language models (LLMs) have significantly influenced both the NLP and SE domains, there are still some challenges associated with them. One such challenge is that it can be difficult to train such models with limited resources, especially in academia. The easiest way to access the models is through APIs, but this can be costly. Another challenge is that the dataset used to pre-train the models may not be accessible, making it difficult to measure the generality of the model. Some results may be inflated because the model may have already seen the data. That might be the case for smaller pre-trained models also. However, in the majority of cases, the dataset is accessible for these models. Additionally, even if the LLM has seen the data in the pre-training stage, the models were trained with different pre-training objectives, which may prevent them from simply memorizing the results. The fact that they perform well on a diverse set of data indicates the robustness of these models.

New large language models (LLMs) are being introduced almost every month, and it can be difficult to keep up with and fully understand each one. Despite variations in performance and size, most of these models are trained with the same objective: autoregressive next token prediction. However, recent research has shown that fine-tuning LLMs with reinforcement learning, using feedback from human annotators, can improve their performance even further. This approach involves providing the model with feedback in the form of rewards for generating high-quality text or penalizing it for generating low-quality text. With this reinforcement learning approach, LLMs can learn to generate more accurate and natural-sounding text that better aligns with human preferences.

7.3. AI Safety and Use Cases

The emergence of LLMs comes with a set of new tools like copilot, which shows promising results in program generation tasks like token completion, line completion, or even function completion. It comes with a new concern regarding the safety of these models. Unlike the prior model, it does not require intensive human intervention and can generate programs that can be executed without any edits.

In this software-centric ecosystem, it is not wise to assume that these models are safe. The models are trained on buggy programs, even with programs that can introduce security vulnerabilities. It is already found [**96**], the model tend to generate buggy program if it is more "natural" than the fixed one. Furthermore, these high capacity models can memorize and regurgitate training data, thus increasing the worry that proprietary information accidentally leaked into training data (Meeting notes, internal discussions, other organizational secrets) can be exposed to such models with clever prompting [**12**].

Considering above scenario, the safety of such model will be the center of attention in near future. Safety-related modification may have to be applied to the model, and or to the model client, to ensure the safe application models. Also even though models are found to be useful, it is also important to gain the trust of the users and make them aware of the possible adverse impact of the model. The AI safety and use-cases of LLMs is slowly becoming the most widely reserach area in this era.

Although large language models (LLMs), even smaller pre-trained models, have revolutionized AI, they have also raised concerns. The interpretability, explainability, and robustness of the models still need to be studied to improve their safety. Despite these concerns, the rise of LLMs is inevitable due to their perceived ability to increase productivity.

Bibliography

[1] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, *Unified pre-training for program understanding and generation*, in Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Online, June 2021, Association for Computational Linguistics, pp. 2655–2668.

[2] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, *A transformer-based approach for source code summarization*, in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL), 2020.

[3] T. Ahmed and P. Devanbu, *Few-shot training llms for project-specific code-summarization*, in 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–5.

[4] ——, *Multilingual training for software engineering*, in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1443–1455.

[5] T. Ahmed, P. Devanbu, and V. Hellendoorn, *Learning lenient parsing & typing via indirect supervision*, Empirical Software Engineering Journal, (To Appear), (2021).

[6] T. Ahmed, P. Devanbu, and A. A. Sawant, *Learning to find usage of library functions in optimized binaries*, IEEE Transactions on Software Engineering, (2021).

[7] ——, *Learning to find usages of library functions in optimized binaries*, IEEE Transactions on Software Engineering, 48 (2021), pp. 3862–3876.

[8] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, *Recommending root-cause and mitigation steps for cloud incidents using large language models*, ICSE, (2023).

[9] T. Ahmed, N. R. Ledesma, and P. Devanbu, *Synshine: improved fixing of syntax errors*, 2022.

[10] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, *Improving few-shot prompts with relevant static analysis products*, arXiv preprint arXiv:2304.06815, (2023).

[11] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, *Compilation error repair: for the student programs, from the student programs*, in Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, 2018, pp. 78–87.

[12] A. Al-Kaswan, M. Izadi, and A. van Deursen, *Targeted attack on gpt-neo for the satml language model data extraction challenge*, arXiv preprint arXiv:2302.07735, (2023).

[13] M. Allamanis, *The adverse effects of code duplication in machine learning models of code*, in Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2019, pp. 143–153.

[14] M. Allamanis, H. Peng, and C. Sutton, *A convolutional attention network for extreme summarization of source code*, in International conference on machine learning, PMLR, 2016, pp. 2091–2100.

[15] U. Alon, S. Brody, O. Levy, and E. Yahav, *code2seq: Generating sequences from structured representations of code*, in International Conference on Learning Representations, 2019.

[16] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, *code2vec: Learning distributed representations of code*, Proceedings of the ACM on Programming Languages, 3 (2019), pp. 1–29.

[17] N. Arivazhagan, A. Bapna, O. Firat, D. Lepikhin, M. Johnson, M. Krikun, M. X. Chen, Y. Cao, G. Foster, C. Cherry, et al., *Massively multilingual neural machine translation in the wild: Findings and challenges*, arXiv preprint arXiv:1907.05019, (2019).

[18] D. Bahdanau, K. Cho, and Y. Bengio, *Neural machine translation by jointly learning to align and translate*, arXiv preprint arXiv:1409.0473, (2014).

[19] S. Banerjee and A. Lavie, *Meteor: An automatic metric for mt evaluation with improved correlation with human judgments*, in Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, 2005, pp. 65–72.

[20] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, {*BYTEWEIGHT*}*: Learning to recognize functions in binary code*, in 23rd {USENIX} Security Symposium ({USENIX} Security 14), 2014, pp. 845–860.

[21] P. Bareiss, B. Souza, M. d'Amorim, and M. Pradel, *Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code*, arXiv preprint arXiv:2206.01335, (2022).

[22] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, *Do developers read compiler error messages?*, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 575–585.

[23] U. Bayer, C. Kruegel, and E. Kirda, *TTAnalyze: A tool for analyzing malware*, na, 2006.

[24] H. B. BE, *Flirt signatures*. `https://www.hex-rays.com/products/ida/tech/flirt/in_depth/`, 2020. Last Accessed August 12th 2020.

[25] H. B. BE, *Hexrays ida pro*. `http://hex-rays.com/products/ida/`, 2020. Last Accessed August 2020.

[26] B. A. Becker, P. Denny, R. Pettit, D. Bouchard, D. J. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P.-M. Osera, et al., *Compiler error messages considered unhelpful: The landscape of text-based programming error message research*, in Proceedings of the working group reports on innovation and technology in computer science education, 2019, pp. 177–210.

[27] E. M. Bender and A. Koller, *Climbing towards NLU: On meaning, form, and understanding in the age of data*, in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, July 2020, Association for Computational Linguistics, pp. 5185–5198.

[28] J. Bennedsen and M. E. Caspersen, *Failure rates in introductory programming*, AcM SIGcSE Bulletin, 39 (2007), pp. 32–36.

[29] E. Biswas, M. E. Karabulut, L. Pollock, and K. Vijay-Shanker, *Achieving reliable sentiment analysis in the software engineering domain using bert*, in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 162–173.

[30] Blinded, *Docker containers created*. `https://hub.docker.com/r/binswarm/cbuilds/tags`, 2020. Last Accessed August 2020.

[31] ——, *Replication package for this work*. `https://doi.org/10.5281/zenodo.4007527`, 2020. Last Accessed August 2020.

[32] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al., *On the opportunities and risks of foundation models*, arXiv preprint arXiv:2108.07258, (2021).

[33] L. C. Briand, *Software documentation: how much is enough?*, in Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings., IEEE, 2003, pp. 13–15.

[34] N. C. Brown and A. Altadmri, *Novice java programming mistakes: Large-scale data vs. educator beliefs*, ACM Transactions on Computing Education (TOCE), 17 (2017), p. 7.

[35] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, *Blackbox: a large scale repository of novice programmers' activity*, in Proceedings of the 45th ACM technical symposium on Computer science education, ACM, 2014, pp. 223–228.

[36] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language models are few-shot learners*, in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds., vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901.

[37] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., *Language models are few-shot learners*, arXiv preprint arXiv:2005.14165, (2020).

[38] BugSwarm, *Bugswarm githubory repository*. `https://github.com/BugSwarm/bugswarm`, 2020. Last Accessed August 2020.

[39] C. Casalnuovo, K. Lee, H. Wang, P. Devanbu, and E. Morgan, *Do programmers prefer predictable expressions in code?*, Cognitive science, 44 (2020), p. e12921.

[40] C. Casalnuovo, K. Sagae, and P. Devanbu, *Studying the difference between natural and programming language corpora*, Empirical Software Engineering, 24 (2019), pp. 1823–1868.

[41] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, *Natgen: generative pre-training by "naturalizing" source code*, in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 18–30.

[42] S. Chakraborty, M. Allamanis, and B. Ray, *Tree2tree neural translation model for learning source code changes*, arXiv preprint arXiv:1810.00314, (2018).

[43] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., *Evaluating large language models trained on code*, arXiv preprint arXiv:2107.03374, (2021).

[44] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, *Metamorphic testing: A review of challenges and opportunities*, ACM Computing Surveys (CSUR), 51 (2018), pp. 1–27.

[45] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, *Sequencer: Sequence-to-sequence learning for end-to-end program repair*, IEEE Transactions on Software Engineering, (2019).

[46] E. J. Chikofsky and J. H. Cross, *Reverse engineering and design recovery: A taxonomy*, IEEE software, 7 (1990), pp. 13–17.

[47] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, *Palm: Scaling language modeling with pathways*, 2022.

[48] T. CI, *Travis build utility*. `https://github.com/travis-ci/travis-build`, 2020. Last Accessed August 2020.

[49] ——, *Travis dockerhub repository*. `https://hub.docker.com/u/travisci`, 2020. Last Accessed August 2020.

[50] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, *Electra: Pre-training text encoders as discriminators rather than generators*, arXiv preprint arXiv:2003.10555, (2020).

[51] M. L. Collard, M. J. Decker, and J. I. Maletic, *Lightweight transformation and fact extraction with the srcml toolkit*, in 2011 IEEE 11th international working conference on source code analysis and manipulation, IEEE, 2011, pp. 173–184.

[52] A. Conneau and G. Lample, *Cross-lingual language model pretraining*, Advances in Neural Information Processing Systems, 32 (2019), pp. 7059–7069.

[53] R. Dabre, C. Chu, and A. Kunchukuttan, *A survey of multilingual neural machine translation*, ACM Computing Surveys (CSUR), 53 (2020), pp. 1–38.

[54] Y. David, U. Alon, and E. Yahav, *Neural reverse engineering of stripped binaries using augmented control flow graphs*, Proceedings of the ACM on Programming Languages, 4 (2020), pp. 1–28.

[55] P. Denny, A. Luxton-Reilly, and E. Tempero, *All syntax errors are not equal*, in Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, 2012, pp. 75–80.

[56] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, arXiv preprint arXiv:1810.04805, (2018).

[57] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, *Patching as translation: the data and the metaphor*, 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), (2020).

[58] Docker, *Docker job matrix configuration*. `https://docs.travis-ci.com/user/build-matrix`, 2020. Last Accessed August 2020.

[59] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, *Evaluating source code summarization techniques: Replication and expansion*, in 2013 21st International Conference on Program Comprehension (ICPC), IEEE, 2013, pp. 13–22.

[60] T. Eisenbarth, R. Koschke, and D. Simon, *Aiding program comprehension by static and dynamic feature analysis*, in Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, IEEE, 2001, pp. 602–611.

[61] Y. Elazar, S. Ravfogel, A. Jacovi, and Y. Goldberg, *Amnesic probing: Behavioral explanation with amnesic counterfactuals*, Transactions of the Association for Computational Linguistics, 9 (2021), pp. 160–175.

[62] A. Ettinger, *What BERT is not: Lessons from a new suite of psycholinguistic diagnostics for language models*, Transactions of the Association for Computational Linguistics, 8 (2020), pp. 34–48.

[63] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, *Automated repair of programs from large language models*, ICSE, 2022.

[64] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., *Codebert: A pre-trained model for programming and natural languages*, arXiv preprint arXiv:2002.08155, (2020).

[65] A. Forward and T. C. Lethbridge, *The relevance of software documentation, tools and technologies: a survey*, in Proceedings of the 2002 ACM symposium on Document engineering, 2002, pp. 26–33.

[66] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, *Code structure guided transformer for source code summarization*, arXiv preprint arXiv:2104.09340, (2021).

[67] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, *Code to comment ?translation?: Data, metrics, baselining & evaluation*, in 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2020, pp. 746–757.

[68] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, *Graphcode{bert}: Pre-training code representations with data flow*, in International Conference on Learning Representations, 2021.

[69] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., *Graphcodebert: Pre-training code representations with data flow*, in International Conference on Learning Representations, 2020.

[70] R. Gupta, A. Kanade, and S. Shevade, *Deep reinforcement learning for programming language correction*, arXiv preprint arXiv:1801.10467, (2018).

[71] R. Gupta, S. Pal, A. Kanade, and S. Shevade, *Deepfix: Fixing common c language errors by deep learning*, in Thirty-First AAAI Conference on Artificial Intelligence, 2017.

[72] T.-L. Ha, J. Niehues, and A. Waibel, *Toward multilingual neural machine translation with universal encoder and decoder*, arXiv preprint arXiv:1611.04798, (2016).

[73] S. Haiduc, J. Aponte, and A. Marcus, *Supporting program comprehension with source code summarization*, in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, 2010, pp. 223–226.

[74] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, *On the use of automated text summarization techniques for summarizing source code*, in 2010 17th Working Conference on Reverse Engineering, IEEE, 2010, pp. 35–44.

[75] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, *Semantic similarity metrics for evaluating source code summarization*, in Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 36–47.

[76] S. Haque, A. LeClair, L. Wu, and C. McMillan, *Improved automatic summarization of subroutines via attention to file context*, in Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 300–310.

[77] M. Hasan, T. Muttaqueen, A. A. Ishtiaq, K. S. Mehrab, M. Haque, M. Anjum, T. Hasan, W. U. Ahmad, A. Iqbal, and R. Shahriyar, *Codesc: A large code-description parallel dataset*, arXiv preprint arXiv:2105.14220, (2021).

[78] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, *Debin: Predicting debug information in stripped binaries*, in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 1667–1680.

[79] V. J. Hellendoorn and P. Devanbu, *Are deep neural networks the best choice for modeling source code?*, in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 763–773.

[80] J. Henke, G. Ramakrishnan, Z. Wang, A. Albarghouth, S. Jha, and T. Reps, *Semantic robustness of models of source code*, in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 526–537.

[81] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, *On the naturalness of software*, in 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 837–847.

[82] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, Neural computation, 9 (1997), pp. 1735–1780.

[83] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, *Deep code comment generation*, in 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), IEEE, 2018, pp. 200–20010.

[84] ———, *Deep code comment generation with hybrid lexical and syntactical information*, Empirical Software Engineering, 25 (2020), pp. 2179–2217.

[85] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, *Summarizing source code with transferred api knowledge*, (2018).

[86] J. Huang and K. C.-C. Chang, *Towards reasoning in large language models: A survey*, arXiv preprint arXiv:2212.10403, (2022).

[87] Huggingface, *Huggingface transformers*. `https://github.com/huggingface/transformers`, 2020. Last Accessed August 2020.

[88] G. Hunt and D. Brubacher, *Detours: Binaryinterception ofwin 3 2 functions*, in 3rd usenix windows nt symposium, 1999.

[89] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, *Codesearchnet challenge: Evaluating the state of semantic code search*, arXiv preprint arXiv:1909.09436, (2019).

[90] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, *Classification of malware based on integrated static and dynamic features*, Journal of Network and Computer Applications, 36 (2013), pp. 646–656.

[91] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, *Summarizing source code using a neural attention model*, in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2073–2083.

[92] P. Jaccard, *Étude comparative de la distribution florale dans une portion des alpes et des jura*, Bull Soc Vaudoise Sci Nat, 37 (1901), pp. 547–579.

[93] J. Jackson, M. Cobb, and C. Carver, *Identifying top java errors for novice programmers*, in Proceedings Frontiers in Education 35th annual conference, IEEE, 2005, pp. T4C–T4C.

[94] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, *Jigsaw: Large language models meet program synthesis*, in Proceedings, 44th ICSE, 2022, pp. 1219–1231.

[95] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, *Contrastive code representation learning*, arXiv preprint arXiv:2007.04973, (2020).

[96] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, *Large language models and simple, stupid bugs*, arXiv preprint arXiv:2303.11455, (2023).

[97] K. Jesse, P. T. Devanbu, and T. Ahmed, *Learning type annotation: is big data enough?*, in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1483–1486.

[98] N. Jiang, K. Liu, T. Lutellier, and L. Tan, *Impact of code language models on automated program repair*, ICSE, (2023).

[99] N. Jiang, T. Lutellier, and L. Tan, *Cure: Code-aware neural machine translation for automatic program repair*, in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 1161–1173.

[100] D. Jin, Z. Jin, J. T. Zhou, and P. Szolovits, *Is BERT really robust? A strong baseline for natural language attack on text classification and entailment*, in The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, AAAI Press, 2020, pp. 8018–8025.

[101] H. Joshi, J. Cambronero, S. Gulwani, V. Le, I. Radicek, and G. Verbruggen, *Repair is nearly generation: Multilingual program repair with llms*, arXiv preprint arXiv:2208.11640, (2022).

[102] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, *Learning and evaluating contextual embedding of source code*, in Proceedings of the 37th International Conference on Machine Learning, H. D. III and A. Singh, eds., vol. 119 of Proceedings of Machine Learning Research, PMLR, 13–18 Jul 2020, pp. 5110–5121.

[103] ———, *Learning and evaluating contextual embedding of source code*, in International Conference on Machine Learning, PMLR, 2020, pp. 5110–5121.

[104] ———, *Learning and evaluating contextual embedding of source code*, in International Conference on Machine Learning, PMLR, 2020, pp. 5110–5121.

[105] H. Kane, M. Y. Kocyigit, A. Abdalla, P. Ajanoh, and M. Coulibali, *Nubia: Neural based interchangeability assessor for text generation*, 2020.

[106] S. Kang, J. Yoon, and S. Yoo, *Large language models are few-shot testers: Exploring llm-based general bug reproduction*, ICSE, (2023).

[107] E. Kantorowitz and H. Laor, *Automatic generation of useful syntax error messages*, Software: Practice and Experience, 16 (1986), pp. 627–640.

[108] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, *Big code != big vocabulary: Open-vocabulary models for source code*, in International Conference on Software Engineering (ICSE), 2020.

[109] A. Karmakar and R. Robbes, *What do pre-trained code models know about code?*, in 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 1332–1336.

[110] N. Kassner and H. Schütze, *Negated and misprimed probes for pretrained language models: Birds can talk, but cannot fly*, in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, July 2020, Association for Computational Linguistics, pp. 7811–7818.

[111] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, *Towards neural decompilation*, arXiv preprint arXiv:1905.08325, (2019).

[112] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, *Opennmt: Open-source toolkit for neural machine translation*, arXiv preprint arXiv:1701.02810, (2017).

[113] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, *Large language models are zero-shot reasoners*, arXiv preprint arXiv:2205.11916, (2022).

[114] S. K. Kummerfeld and J. Kay, *The neglected battle fields of syntax errors*, in Proceedings of the fifth Australasian conference on Computing education-Volume 20, Citeseer, 2003, pp. 105–111.

[115] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, *Dire: A neural approach to decompiled identifier naming*, in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 628–639.

[116] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, *A study of the difficulties of novice programmers*, Acm SIGCSE bulletin, 37 (2005), pp. 14–18.

[117] S. M. Lakew, M. Cettolo, and M. Federico, *A comparison of transformer and recurrent neural networks on multilingual neural machine translation*, arXiv preprint arXiv:1806.06957, (2018).

[118] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, *Effective identifier names for comprehension and memory*, Innovations in Systems and Software Engineering, 3 (2007), pp. 303–318.

[119] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, *Genprog: A generic method for automatic software repair*, Ieee transactions on software engineering, 38 (2011), pp. 54–72.

[120] A. LeClair, A. Bansal, and C. McMillan, *Ensemble models for neural source code summarization of subroutines*, arXiv preprint arXiv:2107.11423, (2021).

[121] A. LeClair, S. Haque, L. Wu, and C. McMillan, *Improved code summarization via a graph neural network*, in Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 184–195.

[122] A. LeClair, S. Jiang, and C. McMillan, *A neural model for generating natural language summaries of program subroutines*, in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 795–806.

[123] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, *Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models*, in 45th International Conference on Software Engineering, ser. ICSE, 2023.

[124] B. Li, M. Yan, X. Xia, X. Hu, G. Li, and D. Lo, *Deepcommenter: a deep code comment generation tool with hybrid lexical and syntactical information*, in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1571–1575.

[125] Y. Li, S. Wang, and T. N. Nguyen, *Dlfix: Context-based code transformation learning for automated program repair*, in 2020 42th International Conference on Software Engineering (ICSE), 2020.

[126] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, and D. Yu, *Secnn: A semantic cnn parser for code comment generation*, Journal of Systems and Software, 181 (2021), p. 111036.

[127] C.-Y. Lin, *Rouge: A package for automatic evaluation of summaries*, in Text summarization branches out, 2004, pp. 74–81.

[128] C.-Y. Lin and F. J. Och, *Orange: a method for evaluating automatic evaluation metrics for machine translation*, in COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics, 2004, pp. 501–507.

[129] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, *Roberta: A robustly optimized bert pretraining approach*, arXiv preprint arXiv:1907.11692, (2019).

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

[131] F. Long and M. Rinard, *Automatic patch generation by learning correct code*, in Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp. 298–312.

[132] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, *Déjàvu: a map of code duplicates on github*, Proceedings of the ACM on Programming Languages, 1 (2017), pp. 1–28.

[133] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al., *Codexglue: A machine learning benchmark dataset for code understanding and generation*, arXiv preprint arXiv:2102.04664, (2021).

[134] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, *Codexglue: A machine learning benchmark dataset for code understanding and generation*, CoRR, abs/2102.04664 (2021).

[135] M.-T. Luong, H. Pham, and C. D. Manning, *Effective approaches to attention-based neural machine translation*, arXiv preprint arXiv:1508.04025, (2015).

[136] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, *Coconut: combining context-aware neural translation models using ensemble for program repair*, in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 101–114.

[137] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, *Cliff's delta calculator: A non-parametric effect size program for two groups of observations*, Universitas Psychologica, 10 (2011), pp. 545–555.

[138] J. Mahmud, F. Faisal, R. I. Arnob, A. Anastasopoulos, and K. Moran, *Code to comment translation: A comparative study on model effectiveness & errors*, arXiv preprint arXiv:2106.08415, (2021).

[139] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, *Studying the usage of text-to-text transfer transformer to support code-related tasks*, in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 336–347.

[140] M. M. Masud, J. Gao, L. Khan, J. Han, and B. Thuraisingham, *Mining concept-drifting data stream to detect peer to peer botnet traffic*, Univ. of Texas at Dallas, Tech. Report# UTDCS-05-08, (2008).

[141] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, *A multi-national, multi-institutional study of assessment of programming skills of first-year cs students*, in Working group reports from ITiCSE on Innovation and technology in computer science education, ACM, 2001, pp. 125–180.

[142] R. P. Medeiros, G. L. Ramalho, and T. P. Falcão, *A systematic literature review on teaching and learning introductory programming in higher education*, IEEE Transactions on Education, 62 (2018), pp. 77–90.

[143] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, *Deepdelta: learning to repair compilation errors*, (2019).

[144] P. Nakov and H. T. Ng, *Improved statistical machine translation for resource-poor languages using related resource-rich languages*, in Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, 2009, pp. 1358–1367.

[145] N. Nashid, M. Sintaha, and A. Mesbah, *Retrieval-based prompt selection for code-related few-shot learning*, in Proceedings, 45th ICSE, 2023.

[146] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, *Codegen: An open large language model for code with multi-turn program synthesis*, arXiv preprint arXiv:2203.13474, (2022).

[147] NS-A, *Ghidra*. `https://ghidra-sre.org/`, 2020. Last Accessed August 2020.

[148] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, *Training language models to follow instructions with human feedback*, 2022.

[149] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, *Bleu: a method for automatic evaluation of machine translation*, in Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 2002, pp. 311–318.

[150] A. Parrish, S. Schuster, A. Warstadt, O. Agha, S.-H. Lee, Z. Zhao, S. R. Bowman, and T. Linzen, *NOPE: A corpus of naturally-occurring presuppositions in English*, in Proceedings of the 25th Conference on Computational Natural Language Learning, Online, Nov. 2021, Association for Computational Linguistics, pp. 349–366.

[151] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, *Retrieval augmented code generation and summarization*, arXiv preprint arXiv:2108.11601, (2021).

[152] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, *Xda: Accurate, robust disassembly with transfer learning*, arXiv preprint arXiv:2010.00770, (2020).

[153] D. Perez and S. Chiba, *Cross-language clone detection by learning over abstract syntax trees*, in 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, 2019, pp. 518–528.

[154] L. Phan, H. Tran, D. Le, H. Nguyen, J. Anibal, A. Peltekian, and Y. Ye, *Cotext: Multi-task learning with code-text transformer*, arXiv preprint arXiv:2105.08645, (2021).

[155] T. Pimentel, J. Valvoda, R. Hall Maudslay, R. Zmigrod, A. Williams, and R. Cotterell, *Information-theoretic probing for linguistic structure*, in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, July 2020, Association for Computational Linguistics, pp. 4609–4622.

[156] M. Pradel and K. Sen, *Deepbugs: A learning approach to name-based bug detection*, Proceedings of the ACM on Programming Languages, 2 (2018), pp. 1–25.

[157] W. Qi, Y. Gong, Y. Yan, C. Xu, B. Yao, B. Zhou, B. Cheng, D. Jiang, J. Chen, R. Zhang, et al., *Prophetnet-x: Large-scale pre-training models for english, chinese, multi-lingual, dialog, and code generation*, arXiv preprint arXiv:2104.08006, (2021).

[158] S. Qiao, Y. Ou, N. Zhang, X. Chen, Y. Yao, S. Deng, C. Tan, F. Huang, and H. Chen, *Reasoning with language model prompting: A survey*, arXiv preprint arXiv:2212.09597, (2022).

[159] J. Qiu, X. Su, and P. Ma, *Library functions identification in binary code by using graph isomorphism testings*, in 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 261–270.

[160] ——, *Using reduced execution flow graph to identify library functions in binary code*, IEEE Transactions on Software Engineering, 42 (2015), pp. 187–202.

[161] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al., *Improving language understanding by generative pre-training*, (2018).

[162] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al., *Language models are unsupervised multitask learners*, OpenAI blog, 1 (2019), p. 9.

[163] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, *Exploring the limits of transfer learning with a unified text-to-text transformer*, arXiv preprint arXiv:1910.10683, (2019).

[164] J. Ramos et al., *Using tf-idf to determine word relevance in document queries*, in Proceedings of the first instructional conference on machine learning, vol. 242, Citeseer, 2003, pp. 29–48.

[165] S. Ranathunga, E.-S. A. Lee, M. P. Skenduli, R. Shekhar, M. Alam, and R. Kaur, *Neural machine translation for low-resource languages: A survey*, arXiv preprint arXiv:2106.15115, (2021).

[166] S. Robertson, H. Zaragoza, et al., *The probabilistic relevance framework: Bm25 and beyond*, Foundations and Trends® in Information Retrieval, 3 (2009), pp. 333–389.

[167] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, *Improving automated source code summarization via an eye-tracking study of programmers*, in Proceedings of the 36th international conference on Software engineering, 2014, pp. 390–401.

[168] A. Rogers, O. Kovaleva, and A. Rumshisky, *A primer in BERTology: What we know about how BERT works*, Transactions of the Association for Computational Linguistics, 8 (2020), pp. 842–866.

[169] D. Roy, S. Fakhoury, and V. Arnaoudova, *Reassessing automatic evaluation metrics for code summarization tasks*, in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1105–1116.

[170] B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, *Dobf: A deobfuscation pre-training objective for programming languages*, arXiv preprint arXiv:2102.07492, (2021).

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

[172] T. Schorsch, *Cap: an automated self-assessment tool to check pascal programs for syntax, logic and style errors*, ACM SIGCSE Bulletin, 27 (1995), pp. 168–172.

[173] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, *Data mining methods for detection of new malicious executables*, in Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001, IEEE, 2000, pp. 38–49.

[174] T. Sellam, D. Das, and A. P. Parikh, *Bleurt: Learning robust metrics for text generation*, arXiv preprint arXiv:2004.04696, (2020).

[175] R. Sennrich, B. Haddow, and A. Birch, *Neural machine translation of rare words with subword units*, arXiv preprint arXiv:1508.07909, (2015).

[176] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, *Neural code summarization: How far are we?*, arXiv preprint arXiv:2107.07112, (2021).

[177] ———, *On the evaluation of neural code summarization*, in Proceedings of the 44th International Conference on Software Engineering, 2023, pp. 1597–1608.

[178] E. C. R. Shin, D. Song, and R. Moazzezi, *Recognizing functions in binaries with neural networks*, in 24th {USENIX} Security Symposium ({USENIX} Security 15), 2015, pp. 611–626.

[179] P. Shirani, L. Wang, and M. Debbabi, *Binshape: Scalable and robust binary library function identification using function shape*, in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2017, pp. 301–324.

[180] D. Shrivastava, H. Larochelle, and D. Tarlow, *Repository-level prompt generation for large language models of code*, arXiv preprint arXiv:2206.12839, (2022).

[181] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, *Bitblaze: A new approach to computer security via binary analysis*, in International Conference on Information Systems Security, Springer, 2008, pp. 1–25.

[182] J. C. Spohrer and E. Soloway, *Novice mistakes: Are the folk wisdoms correct?*, Communications of the ACM, 29 (1986), pp. 624–632.

[183] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, *Towards automatically generating summary comments for java methods*, in Proceedings of the IEEE/ACM international conference on Automated software engineering, 2010, pp. 43–52.

[184] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to sequence learning with neural networks*, in Advances in neural information processing systems, 2014, pp. 3104–3112.

[185] X. Tan, Y. Ren, D. He, T. Qin, Z. Zhao, and T.-Y. Liu, *Multilingual neural machine translation with knowledge distillation*, arXiv preprint arXiv:1902.10461, (2019).

[186] Y. Tang, C. Tran, X. Li, P.-J. Chen, N. Goyal, V. Chaudhary, J. Gu, and A. Fan, *Multilingual translation with extensible multilingual pretraining and finetuning*, arXiv preprint arXiv:2008.00401, (2020).

[187] I. Tenney, P. Xia, B. Chen, A. Wang, A. Poliak, R. T. McCoy, N. Kim, B. Van Durme, S. R. Bowman, D. Das, et al., *What do you learn from context? probing for sentence structure in contextualized word representations*, arXiv preprint arXiv:1905.06316, (2019).

[188] R. Tian, R. Islam, L. Batten, and S. Versteeg, *Differentiating malware from cleanware using behavioural analysis*, in 2010 5th international conference on malicious and unwanted software, Ieee, 2010, pp. 23–30.

[189] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, *Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes*, in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 339–349.

[190] S. Troshin and N. Chirkova, *Probing pretrained models of source code*, arXiv preprint arXiv:2202.08975, (2022).

[191] Z. Tu, Z. Su, and P. Devanbu, *On the localness of software*, in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 269–280.

[192] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, *On learning meaningful code changes via neural machine translation*, in Proceedings of the 41st International Conference on Software Engineering, IEEE Press, 2019, pp. 25–36.

[193] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention is all you need*, in Advances in neural information processing systems, 2017, pp. 5998–6008.

[194] M. Vijaymeena and K. Kavitha, *A survey on similarity measures in text mining*, Machine Learning and Applications: An International Journal, 3 (2016), pp. 19–28.

[195] E. Wallace, S. Feng, N. Kandpal, M. Gardner, and S. Singh, *Universal adversarial triggers for attacking and analyzing NLP*, 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), Hong Kong, China, Nov. 2019, Association for Computational Linguistics, pp. 2153–2162.

[196] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, *What do they capture? a structural analysis of pre-trained language models for source code*, in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2377–2388.

[197] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, *Improving automatic source code summarization via deep reinforcement learning*, in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 397–407.

[198] S. Wang, P. Wang, and D. Wu, *Semantics-aware machine learning for function recognition in binary code*, in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 388–398.

[199] T.-Y. Wang, S.-J. Horng, M.-Y. Su, C.-H. Wu, P.-C. Wang, and W.-Z. Su, *A surveillance spyware detection system based on data mining methods*, in 2006 IEEE International Conference on Evolutionary Computation, IEEE, 2006, pp. 3236–3241.

[200] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, *Reinforcement-learning-guided source code summarization via hierarchical attention*, IEEE Transactions on software Engineering, (2020).

[201] X. Wang, Y. Wang, P. Zhou, M. Xiao, Y. Wang, L. Li, X. Liu, H. Wu, J. Liu, and X. Jiang, *Clsebert: Contrastive learning for syntax enhanced code pre-trained model*, arXiv preprint arXiv:2108.04556, (2021).

[202] Y. Wang, E. Shi, L. Du, X. Yang, Y. Hu, S. Han, H. Zhang, and D. Zhang, *Cocosum: Contextual code summarization with multi-relational graph neural network*, arXiv preprint arXiv:2107.01933, (2021).

[203] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, *Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation*, in Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[204] C. Watson and F. W. Li, *Failure rates in introductory programming revisited*, in Proceedings of the 2014 conference on Innovation & technology in computer science education, 2014, pp. 39–44.

[205] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, *Code generation as a dual task of code summarization*, arXiv preprint arXiv:1910.05923, (2019).

[206] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, *Retrieve and refine: exemplar-based neural comment generation*, in 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2020, pp. 349–360.

[207] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, *Chain of thought prompting elicits reasoning in large language models*, arXiv preprint arXiv:2201.11903, (2022).

[208] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou, *Chain of thought prompting elicits reasoning in large language models*, CoRR, abs/2201.11903 (2022).

[209] F. Wilcoxon, *Individual comparisons by ranking methods*, in Breakthroughs in statistics, Springer, 1992, pp. 196–202.

[210] C. Willems, T. Holz, and F. Freiling, *Toward automated dynamic malware analysis using cwsandbox*, IEEE Security & Privacy, 5 (2007), pp. 32–39.

[211] Z. Wu, Y. Chen, B. Kao, and Q. Liu, *Perturbed masking: Parameter-free probing for analyzing and interpreting BERT*, in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, July 2020, Association for Computational Linguistics, pp. 4166–4176.

[212] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, *A systematic evaluation of large language models of code*, in Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, 2022, pp. 1–10.

[213] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, and K. Liu, *Comformer: Code comment generation via transformer and fusion method-based hybrid code representation*, arXiv preprint arXiv:2107.03644, (2021).

[214] M. Yasunaga and P. Liang, *Graph-based, self-supervised program repair from diagnostic feedback*, in International Conference on Machine Learning, PMLR, 2020, pp. 10799–10808.

[215] ——, *Break-it-fix-it: Unsupervised learning for program repair*, arXiv preprint arXiv:2106.06600, (2021).

[216] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, *A survey on malware detection using data mining techniques*, ACM Computing Surveys (CSUR), 50 (2017), pp. 1–40.

[217] Y. Ye, T. Li, K. Huang, Q. Jiang, and Y. Chen, *Hierarchical associative classifier (hac) for malware detection from the large and imbalanced gray list*, Journal of Intelligent Information Systems, 35 (2010), pp. 1–20.

[218] Y. Ye, T. Li, Q. Jiang, Z. Han, and L. Wan, *Intelligent file scoring system for malware detection from the gray list*, in Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, 2009, pp. 1385–1394.

[219] Y. Ye, T. Li, Q. Jiang, and Y. Wang, *Cimds: adapting postprocessing techniques of associative classification for malware detection*, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 40 (2010), pp. 298–307.

[220] Y. Ye, D. Wang, T. Li, and D. Ye, *Imds: Intelligent malware detection system*, in Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, 2007, pp. 1043–1047.

[221] H. Yuchao, W. Moshi, W. Song, W. Junjie, and W. Qing, *Yet another combination of ir-and neural-based comment generation*, arXiv preprint arXiv:2107.12938, (2021).

[222] Y. Zemlyanskiy, M. de Jong, J. Ainslie, P. Pasupat, P. Shaw, L. Qiu, S. Sanghai, and F. Sha, *Generate-and-retrieve: use your predictions to improve retrieval for semantic parsing*, arXiv preprint arXiv:2209.14899, (2022).

[223] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, *Retrieval-based neural source code summarization*, in 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE, 2020, pp. 1385–1397.

[224] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, *Bertscore: Evaluating text generation with bert*, arXiv preprint arXiv:1904.09675, (2019).

[225] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, *Sentiment analysis for software engineering: How far can pre-trained transformer models go?*, in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 70–80.

[226] K. Zhou, D. Jurafsky, and T. Hashimoto, *Navigating the grey area: Expressions of overconfidence and uncertainty in language models*, arXiv preprint arXiv:2302.13439, (2023).