Modeling Source Code For Developers

By

KEVIN R. JESSE
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 T. Devanbu, Chair

Vladimir Filkov

Kenji Sagae

Committee in Charge

2023

Abstract

Software Engineering practices are changing in an age of artificial intelligence. While the core activities of design, develop, maintain, test and evaluate remain, the methods used in these activities are evolving. The prevalence of generative programming models has the potential to reconstitute the duties of a software engineer. Widely adopted models like Copilot and Bard are IDE-based pair-programming assistants that *create* code from virtually any input: contextual code, natural language, specifications, input output pairs, etc. The way developers interact with these models will redefine some core ideas of software engineering. These models empower virtually anyone, of varying coding proficiency, to create software. Models with the capacity to code will surely manage to inherit software design and analysis capabilities [**186**], albeit for now, with specific training or prompting.

Naturally, one wonders how language modeling, or more specifically the modeling of source code and its features, will impact developers. Researchers often conjecture on the varying degree of influence these methods will have, but certainly, these tools will support developers in new and existing tasks: code completion, bug and vulnerability detection, code summarization, type annotation, and more are already prominent use cases. One can envision a world where software developers delegate portions of their work to machine learning pipelines, such as unit testing and vulnerability testing of their code; how much of that code they actually write is up for debate as well. Developers will likely automate portions of their work flow but simultaneously gain new tasks and responsibilities. These tasks might include passing automatic code reviews that detect code smells, place code comments automatically, and detect refactorings; maybe using models from [**56**], [**133**], [**59**]. These capabilities come from modeling source code and its features directly by distilling down meaningful representations for the task at hand.

This thesis explores learning meaningful representations from code through a variety of applications for developer supporting tools. The first application is a type-prediction model using representations learned with masked-language-modeling. While effective, we find that the off-the-shelf model fails at an aspect of modeling source code, namely the use of local user-defined types. The next application modifies the model learned representations with one characterized by an objective function capturing how developers *actually* use types. Along this body of work, the next

two chapters present a type inference dataset for the community and a framework for new machine learning models with a Visual Studio plugin. This thesis concludes with a study of large language models on single statement bug introduction and proposes avoidance strategies. Finally I present some future work to improve these models. By reading this thesis, I hope the reader has a few takeaways:

(1) Machine learning is an essential tool for capturing code and its meta data. Models trained on code and its features are capable of generalizing and improving old and new processes.

(2) The data that models train on is not perfect, and the resulting models often inherit biases towards vulnerable and buggy code; researchers must evaluate the risk vs. reward with broadly trained models.

(3) The objectives optimized for software models may not align with our goals; models that incorporate human feedback may ultimately align better to our values and understanding of code.

(4) Large language models are powerful tools for software engineering, but they're only part of the picture. Models that learn data and control flow, project and file meta data, local and global scope semantics, and information associated with code traces, are better informed on the source code it consumes *and* produces.

This thesis attempts to quantify the utility of off-the-shelf LLMs like BERT, the misalignment of LLM representations to human derived representations of coding constructs, and the present risks of using LLM predictions at face value. Hopefully, in each case, the chapters leave you optimistic that many of the aforementioned concerns can be minimized, or mitigated with just a bit of ingenuity.

Acknowledgments

The PhD is a journey. I have met very smart and successful people that have completed a PhD and many that were not able to finish. The PhD process is a cycle of self improvement in technical and soft skills; both of which I find I am still improving. With my time as a PhD student coming to a close, I reflect on what I have learned and the people who graciously guided me. Frankly, when I started this process, I viewed the PhD as a way to further my passion and depth of knowledge for machine learning, but ultimately the PhD expanded my breadth of skills including communication via writing, speaking, and listening. Listening to others' ideas and infusing their intellectual curiosity with my own has led to, by far, my best work in the PhD. I am grateful for the people who took the time to educate me by listening to my ideas, guiding me in them, and letting me explore them until whatever end they led me to.

First and foremost, I need to thank my advisor Prem Devanbu. I feel quite lucky to have been mentored by Prem. Prem is incredibly well versed in one of the fastest paced research fields and is a beacon of innovation and ingenuity. Each meeting with Prem is filled with novel ideas, some even redefining how we think of the problem. Prem's breadth of knowledge is multidisciplinary and I often remark at the connections he makes between various problems in a variety of domains, thus enabling us to tackle many problems from multiple perspectives. I have really enjoyed this aspect of our advisor-student relationship; and I hope it continues in some fashion after my PhD. Along this vein, Prem gave me academic freedom to pursue problems that I found merit in; again, thank you for allowing me to put *my* stamp on the software engineering community. As I find myself presenting two papers at ICSE/MSR this May, I cannot help to think of where this started. Prem, thank you for taking me as a student and invigorating my passion for research; without your mentorship I would not be completing the PhD.

My first project in Prem's research group was focused around type inference and Vincent Hellendoorn's neural type inference model. Unfortunately for Vincent, he sat next to an eager but novice researcher that had endless questions. Vincent, thank you for answering those questions, even at times, when you were too busy. Since then, I have worked with undergraduates and I am always reminded of your patience and care. I value your expertise and opinion greatly. Thank you.

I have had the pleasure of working with a few special folks too. Anand Sawant is a collaborator and friend. I am thankful for our conversations on research and industry. Throughout the pandemic we met regularly over Zoom and now in person with a craft cocktail or beer; I look forward to many more conversations both on research and personal growth. Additionally, I must thank my collaborators in academia, Emily Morgan, Toufique Ahmed, David Gros, and Claudio Spiess. Thank you for taking part in my research, listening to my ideas, and helping me focus them into contributions to the field. Each of you helped me hone an idea into a contribution for the field. In industry, I especially thank Gokce Keskin formerly at Intel AI and Amazon, for pushing me in every meeting, where each meeting was in fact a qualifying exam; you taught me to be thorough in my evidence and reasoning. Thank you Christoph Kuhmuench for guiding me through our work at Siemens.

Lastly, I must thank those in the research community that has guided my work more generally. I have received feedback in a variety of ways by Vincent Hellendoorn, Romain Robbes, Miltos Allamanis, Chris Bird, Charles Sutton, and Amin Alipour. Small or large, the kind critique of ideas, the welcoming nature of the AI4SE community, and the openness of the people subscribe me to this industry for the foreseeable future. I look forward to future conversations and collaborations with all that I previously mentioned.

I chose to do a PhD so I could have a seat at the table in a research community. I realize now, that seat was always open for those that wish to contribute to the conversation. I value the PhD differently today than when I started. When I started, it was more about the accomplishment of having the highest education academia could offer and for many the PhD is just a degree. Today, I know the PhD is a standing your committee bestows, such that, one is humbly recognized as a peer in the pursuit of purposeful, ethical, and sound research; I am honored to have this distinction from my committee.

Finally, I look back to a conversations I had with my now wife Danielle when I was just thinking of applying for a PhD. Thank you for giving me the confidence to pursue my passion. Thank you for always being supportive and helping me navigate the difficult times in this journey; for that, I owe you a lifetime of love and support. As I conclude this part of my life, I am excited about the possibilities we will explore together in our respective careers. I have learned so much from you over

these years and I am so excited to have you as a partner forever. Having your love is greater than any pursuit or accomplishment, and I am eternally grateful. Finally, we joke about changing our letterhead, but if you know us, you know we won't. Sorry I took so long, Doc.

To my parents, Frank and Judy, who are without fail, the most supportive people I have ever known. From an early age, they encourage us to dream big and pursue our dreams without hesitation; and to my sister Lauren, now becoming a physician assistant. And to my extended family Gabriela, Darrell, and Emily, whom I have known since I was a young man, thank you for your patience, love, and guidance. To my close friends, thank you for your support $\sim$.

Let's get on with it before ChatGPT writes this thesis without me!

Chapter 1

Finding Meaningful Representations of Software

Software permeates all avenues of modern society, including retail, energy, healthcare, space, automotive, construction and more. Due to the integration of software in our societal fabric, there is an increasing demand to improve software development. Improvements may focus on the **process** of creating software (*e.g.* rapid prototyping and better programming environments) or the **product** with improving reliability and maintainability. With respect to the product, code is often reviewed by automatic tools developed by software engineering researchers and programmers. Automatic tools vary in their sophistication from observing patterns in individual statements and declarations, to complete analysis of programs. The analysis of code can result in simple results such as linting to leveraging mathematically derived formal methods to prove properties of a program.

Formal methods, are attractive because of the provability of program behavior, facts, and properties. Static program analysis has led to impactful advancements in the sub-fields of program verification [**244**], type checking [**30**], program minification [**49**], refactoring [**71**], and more. The rigor of formal proofs means that if well-specified guarantees exists in code, and as long as the proof holds, then the guarantees hold. In the medical, nuclear, aviation, and automotive industries, such guarantees are useful. Static analysis and formal models ultimately fall short on the chaos and complexity of real systems. For example, exhaustively testing a simple program that adds

only two 32-bit inputs would take hundreds of years. Moreover, with the communicative intent of developers, identifier and function names provide rich semantic clues, i.e `age`, and can be reasonably approximated and verified without previously seen distributions of values; age doesn't exceed 125 years. As we see, the formal approach to program analysis fails to capture many attitudes of the developer and the code itself. Until the last decade, software analysis largely avoided the text of the code in favor of various formal representations. The availability of "big-data" in the form of public software repositories and the recent evolution of machine learning methodologies inspires a greater analysis of statistical properties of *natural* code.

The rise of publicly available open-source software in the last decade provides an invaluable resource of "big-data". The scale of data is massive with over 128 million repositories on Github in addition to many other alternative hosting platforms. The repositories are rich with features and expose not just the code, but meta data about the code including bug-fixes, commits, api changes, and more. The availability of large scale data suggests that statistical based approaches can be effective in learning the communicative intent of humans and translate into many "downstream" applications. We ask, *"What statistical properties, analysed over representative code corpora, can elucidate meaningful representations of software?"* and *"How can we use them to help developers produce better code?"*. We are motivated by the various applications these representations can be implemented into; we will speak of representations align appropriately to developers' use of type inference and applications that are derived from such models.

In recent years, the software engineering community has seen a rapid departure from formal approaches with 69% (and growing) of publications in SE conference proceedings employing deep learning techniques [**241**]. Such research, including the use of machine learning, is often exploiting software repositories as a form of data. The conventions of popular coding languages and the semantics and syntax of code make way to the *naturalness of code*. The **naturalness hypothesis** states,

> *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.* [**8**]

Similar to how natural language is complex and expressive, programming languages are complex and powerful, however, both are largely regular and predictable. This is because developers are writing code with the intent of communicating clearly what the code should do and using the language as it is intended to be used. Code specifically requires understanding and problem solving so code exhibits larger regularity and repetitiveness than natural language due to code *reuse* [**64**]. The first empirical evidence of the naturalness hypothesis was in early works by Hindle et al. [**81**] where models designed for natural language were surprisingly effective on code; they also found that models could represent code with less bits! Following, other works found that the regularity of code could indicate when code was *irregular* or likely buggy [**180**]. Many methods and applications have emerged from the naturalness hypothesis in tandem with the rapid advance in natural language processing, computer vision, and other related fields. Such applications range from program analysis and synthesis, debugging, information retrieval, specification understanding, recommendation systems, type inference, and more [**8**]. The rapid application of machine learning to software engineering isn't all that surprising given the success on natural language and yet challenges arise.

While code is similar to natural language and includes forms of human communication exploitable by the naturalness hypothesis, code differs from natural language in ways that inhibit the same effective application of machine learning methods. Code has a formal syntax and semantics which allow machines to understand and execute the code when formed properly. Many Natural Language Processing (NLP) techniques applied to code do not adhere to the formal constraints of code or require methods to filter solutions with correct *local* syntax [**139**, **172**].

On the other hand, neural networks are effective learners of regular semantics and most formal syntax [**17**, **228**]. While code can be characterized as significantly less ambiguous than natural language, ambiguity still exists in parse tree structures, polymorphism, aliasing, and more. This means often or not, machine learners will have to model code accounting for ambiguity and noise. For tasks like type inference where variables/function types are dynamically assigned, attempts to type from static code will result in degrees of ambiguity. Techniques for resolving ambiguity can include incorporating information from other sources: traces, data flow, control flow, and common types in frequently included libraries to name a few. Ambiguity can be resolved from sources "outside" of the typical channels of code. Natural language through comments or functions

can describe a developers intent or desired goal. Machines can likewise use natural language to *summarize* the code into digestible segments for developers and use generated natural comments to help the developer verify code functionality [**87**]. Both the generation of code and natural language requires models to understand ambiguity in one domain (natural language) and transfer into an unambiguous target language (code) or vice versa. Finally, ambiguity can arise from human error in practical applications such as type inference with types that were mistakenly not imported; it is relevant to recommend the most likely type (potentially ambiguous) *and* the missing import statement. The duality of natural language and code further complicates the challenge of finding representation that is suitable for either *or both*.

Researchers have several ways to represent code with various architectures and learning techniques. Source code representations can differ greatly by the code semantics we intend to capture; there are general purpose representations learned from LLMs and specific representations crafted by an (neural) architects' understanding of the problem *viz.* inductive bias. Code in its simplest form, tokens, are typically used as input to neural networks to learn probabilistic language modeling; additionally input can be supplemented with ASTs or other code specific features [**98**, **199**]. Representations of code can range from traditional n-grams [**82**], bag-of-words, single/bidirectional Recurrent Neural Networks (RNN) [**78**], Hierarchical Neural Networks (HNNs) [**149**], Graph Neural Networks (GNNs) [**10**, **11**], Transformers [**62**], Hybrids [**80**], attention-based flow [**72**] and various tree-based encoding [**98**, **163**]. All of these representations extract meaningful features from code for specific purposes e.g. data flow for type information. We emphasize that extracting meaningful representations of code is part process in ML (improved machine learning techniques) [**222**], and part realization of adaptations that distill valuable code semantics in SE [**163**].

In this work, we try to improve code representations specifically by including valuable information that is inherent to code. Derivative representations of code currently exists in graph representations (hyperedges), attention-based data-flow edges, and AST information. Following this trend, aspects of code that have yet to be well integrated into static code representations are static/dynamic features like data value distributions, code traces, code structure, and more. The pursuit of *extractable* code features or external data is well worth the effort [**72**] and should be explored deeper. Regular features of programming languages change with new versions and often extend beyond the code

itself; e.g. meta information about projects, patches, code reviews, and more. With a combination of improved methodologies for distilling machine and human-centric representations of code, we hope to provide better insights and solutions for developers. In summary, the thesis of this work is goal-oriented towards better developer-aligned representations of source code. *Developer alignment* means that neural networks understand code in a manner that is aligned to developers' goals. Developer alignment issues exists even in code completion tools where the code *should* not introduce security vulnerabilities and bugs. In this work we see that off the shelf models, including some of the most complete LLMs like OpenAI's Codex, have inherited developer misalignment by introducing single statement bugs quite frequently. This is just one of several scenarios this work discusses.

Outline

In Chapter 2 we provide the reader with AI4SE and ML fundamentals that are the underpinnings for all the discussed research papers.

Towards the objective of modeling source code effectively for developers, we outline several completed works at a high level. In Chapter 3, we examine the task of type inference in JavaScript/TypeScript where variable, function and parameter types are inferred. This is helpful for several reasons, imagine, a code editor that guides the developer in the proper use of types. The flexibility of types in JavaScript/TypeScript means there are an innumerable set of types *viz.* an open-vocabulary of types. Typing is one particular aspect of code that is rather complex. In JavaScript variables can assume types dynamically and inferring these types during development often requires summarizing key language principles such as data-flow. Neural network designers found that explicitly defining relationships related to the task, while sometimes abstract, often improved model performance. This led to a growing field of neural networks that derived equally complicated inductive biases to capture said features [**9**, **230**]. We found that a simple inductive bias actually led to greater performance through large scale self-supervision; the learned regularities in pre-trained models could be transferred to type inference! This chapter titled *Learning Type Annotation: Is Big Data Enough?* [**95**] was presented at ESEC/FSE 2021. However, upon greater scrutiny of the models' performance, we found an aspect of typing that the model is particularly poor at: user-defined types.

Chapter 4 builds on the previous chapter by reorienting the model's learned representations with the way developers *actually* use type inference for user-defined types. We use an effective technique to align class and type declarations where the use of these user-defined types elegantly bypasses model limitations like the closed-vocabulary dilemma. The technique is widely applicable to novel types and conceivably any new type definition can be digested by the model and used in inference. The technique is evaluated with never-before-seen types which demonstrates the alignment is effective even for newly crafted types! In congruence with our thesis of modeling source code effectively for developers, we augmented the models' internal representations to be more *developer aligned*, simply put, aligned to the way developers interpret and use types. This chapter was accepted to Transactions on Software Engineering (TSE) titled *Learning to Predict User-Defined Types* and will be featured at ICSE 2023 in the journal first track.

The next two chapters, Chapter 5 and Chapter 6 are research efforts that put forward a model agnostic dataset for learning type inference and a corresponding framework to host such models. Prior to these works, type inference models were often trained and evaluated in an adhoc manner. Comparing two type inference models was often dependent on fleeting factors like project availability on GitHub. To alleviate this problem, we made a dataset with over 9 million type annotations with an automatic evaluation script available on the popular CodeXGLUE website. CodeXGLUE is a leaderboard for code intelligent tasks managed by Microsoft. Part of this contribution was the dataset *and* the mining scripts to update the dataset frequently; coding languages are evolving and the types developers use will change over time. The ManyTypes4TypeScript dataset also contributes state-of-the-art encoder models like CodeBERT and GraphCodeBERT. This chapter titled, *ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference* was accepted at MSR 2022.

The central theme of the ManyTypes4TypeScript work is that the dataset and benchmark is model agnostic. Neural network designers are free to adapt to new ML advancements and apply designs that work well for code. In AI4SE research, models are often benchmarked and left for engineers to integrate into a product; sometimes gamechanging models like Typilus [**11**] and LambdaNet [**230**] are never used beyond research. To address this, we proposed a framework for Visual Studio that does the heavy lifting of type inference; the code manipulation. The framework,

FlexType, finds the location for type inference and parses the AST for sequence-based models. FlexType uses type inference models to generate predictions and presents the predictions to the developer. FlexType is an effective framework for both JavaScript and TypeScript. The paper, *FlexType: A Plug-and-Play Framework for Type Inference Models* was presented at ASE 2022. The demo can be viewed here[1].

In Chapter 7, we evaluate popular code completion tools including OpenAI's Codex to see how often they introduce single statement bugs. Single statement bugs are tricky to find but often easy to solve. We also know that in a growing age of automatic coding, the AI engine might introduce large swaths of code that can riddled with bugs. This chapter explores how prone LLM code completions engines are at introducing these bugs, how pervasive they are, and *if* there is anything a developer can do while using the tool to avoid such bugs. *Spoiler*, there is!

In the final chapter of the thesis we discuss the future of these tools and current sub-fields dedicated to improving them. It is my belief that the field will progress down the path of integrated LLMs for code development, and key improvements will not be driven by parameter count and sheer model size. Instead, these models will be improve by *developer alignment* in a broad set of ways. *Developer alignment* means that models will adapt to the developer (maybe personalization), and fit inference around the code base at hand: code reviews, other developers, product feedback, and more. Currently, promising directions are done with reinforcement learning on human feedback, and conditioning through prompting based on desirable and undesirable outcomes [**76**] (secure versus insecure code). Other techniques, some not yet discovered, will be better than others and AI4SE researchers must uncover and evaluated these alignment techniques; frankly, to defend against threat vectors and unintended consequences. LLMs are wonderful retrieval tools, and must be guided to our interests, values, and goals.

[1] `https://youtu.be/4dPV05BWA8A`

The following publications correspond to the following chapters.

Chapter 3

Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed.

"Learning type annotation: is big data enough?" Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021.

Chapter 4

Kevin Jesse, Premkumar T. Devanbu, and Anand Sawant.

"Learning To Predict User-Defined Types". IEEE Transactions on Software Engineering, April 2023, pp. 1508-1522, vol. 49

45th International Conference on Software Engineering Journal First. 2023

Chapter 5

Kevin Jesse, Premkumar T. Devanbu

"ManyTypes4TypeScript: a comprehensive TypeScript dataset for sequence-based type inference" MSR '22: Proceedings of the 19th International Conference on Mining Software Repositories. 2022.

Chapter 6

Sivani Voruganti, *Kevin Jesse*, Premkumar T. Devanbu

"FlexType: A Plug-and-Play Framework for Type Inference Models" ASE '22: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering

Chapter 7

Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, Emily Morgan

"Large Language Models and Simple, Stupid Bugs" Proceedings of the 20th International Conference on Mining Software Repositories. 2023.

Chapter 2

Background

In this section, we introduce some key principles and theorems that guide the neural modeling of software and help find meaningful representations of code. This section starts with language modeling fundamentals both explicit and implicit. We follow with emerging innovative techniques of modeling corpora, using context, with and without pre-training. We discuss learning techniques that go beyond modeling language by aligning and dissociating representations of elements. Finally we discuss advanced modeling in the context of software engineering as well as the aspects of software that distinguishes it from natural language text.

2.1 Language Modeling Fundamentals

Language modeling is a fundamental task of natural language processing (NLP). In its simplest form it consists of modeling the probability of a word following or given a series of other words called a *context*. Language models assign a probability or score to either a phrase, sentence, word, or character depending on the granularity of the modeling objective. A language model that is representative of the target language should score utterances highly if they were more likely to occur in the training corpus. A high score should mean the sequence of words or characters is *natural* to a native speaker (or writer). The inverse holds true that any *unnatural* sentences results in a lower expectation. In order for a language model to properly predict the natural sequence of words, the language model predictions must be scored in a manner in which they appear naturally. Code models are analogous to language models in that natural code exhibits regular statistical properties

that can be captured by corpus-based language models [**82**]; in fact code reuse facilitates a higher global regularity and language keywords lends to a high level of local regularity. A naturalness survey [**8**] by Allamanis et al. maps existing sub-fields of applying language modeling to source code. Language modeling is not mutually exclusive to code or natural language. Domains of **causal modeling** (migration, code search/synthesis, code completion, obfuscation, information extraction), **representational modeling** (naming, code search, program analysis, typing, traceability, comment prediction, bug detection, and more), and **pattern mining** (defect prediction, knowledge base mining, clone detection, idiom mining, etc) depend on the joint modeling of natural language and code. In the next section, we discuss the formal definitions for causal modeling which frequently is at the core of modeling code.

2.1.1 Causal Language Modeling

The basis for causal modeling is to score a fragment of tokens t for a source snippet S such that the conditional probability yields,

$$P(S) = \prod_{i=1}^{N} P(t_i | t_0, \ldots, t_{i-1}) \tag{2.1}$$

where N is the length of the code snippet S. Equation 2.1 calculates per-token conditional probabilities with the chain rule to yield a single probability for the code snippet S. Through the chain rule, each token's conditional probability depends on previous tokens. This auto-regressive process is causal and can be used to model sequences given a particular *backwards context*. Due to varying orders of magnitude of probabilities and the production of multiplying probabilities of arbitrary lengths, the product of probabilities is often represented by a sum of logarithmic values. The theoretic measure of information is *entropy*[1]

$$H_{\mathcal{M}}(S) = -\frac{1}{|S|} log_2 p_{\mathcal{M}}(S) = -\frac{1}{|S|} \sum log_2 p_{\mathcal{M}}(t_i | c) \tag{2.2}$$

where c represents the aforementioned causal context. Entropy can be explained as the average number of bits to encode samples of probability distribution P; *perplexity* is often an alternative

[1]Typically *cross-entropy* and is evaluated between the expected distribution and the distribution learned by the model *viz.* Kullback-Leiber divergence.

measure seen in NLP literature. Notice that in Equation 2.1, the context includes all previous tokens in the sequence. As corpora get large enough, learning the context given all other tokens is intractable. In order to make learning feasible there are several effective solutions; we can often model windows of code as natural language rarely has far reaching references. We highlight that code does not necessarily follow this assumption as function calls and variables can span entire lengths of files. However, following the conventions of natural language, we will discuss common methodologies for making language modeling tractable over large corpora. In the following two sections, we discuss briefly and summarize explicit and implicit language modeling.

2.2 Explicit Language Models

In this section we will discuss the most common explicit language modeling technique, n-grams. The intuition of the n-gram model is that we are approximating the history of the prior probabilities with just the last few words. In a bigram model, we are approximating the conditional probability of the current word given all other words $P(t_i|t_{1:i-1})$ by using on the conditional probability of only the previous word $P(t_i|t_{i-1})$. This assumption we are making about the approximating all word probabilities through the current probabilistic value of the immediate past word is called a Markov assumption. The bigram is often expanded to larger probabilistic windows of trigram and 4-gram.

In order to estimate n-gram probability one will use maximum likelihood estimation (MLE). MLE of a n-gram is the probability of the particular token given previous tokens (n-gram) normalized by the number of (n-1)grams that share the previous tokens. We can think of this as normalizing the occurrences counts of, say, bigrams by the number of all unigram occurrences of that word [**99**]. Formally,

$$P_{\mathcal{M}}(t_i|\theta) = \frac{count(t_i, D)}{\sum count(t_i, D)} \tag{2.3}$$

where t_i is the first word in the bigram, D is other words in the document. As we can see, the sum equates to the probability normalized by the unigram probability. The relative frequency of words are stored in a table and retrieved when counting the maximum likelihood of a sequence. However a n-gram models come with some caveats.

The maximum likelihood estimator will often under estimate the probability of unseen words in the document. Smoothing practices add a non-zero probability to the unseen words to improve the accuracy of the probability estimation.

$$P_\delta(t_i|\theta) = \frac{count(t_i, D) + \lambda}{\sum count(t_i, D) + \lambda|V|} \tag{2.4}$$

Thus, words that are not seen during evaluation have a value λ associated to the count. Typically λ=1.

Another common practice is to *backoff* and use less context effectively reducing the size of the n-gram. While this means less information, it generally means there is an associated probability for that lesser context n-gram. In order to combine various length n-grams, we can use simple weighting techniques fitted uniformly or with fitted with a held-out dataset:

$$P(t_i|t_{i-2}t_{i-1}) = \lambda_1 P(t_i) + \lambda_2 P(t_i|t_{i-1}) + \lambda_3 P(t_i|t_{i-2}t_{i-1}) \tag{2.5}$$

where λs sum to 1. The field of smoothing language models and backing off is very well studied [**99**]. With the widespread availability of billion word corpora, neural networks and compute, the NLP community has put great focus on modeling language implicitly with neural networks.

2.3 Implicit Language Models

Implicit language modeling with neural networks bypasses some of the difficult caveats in explicit language modeling: storing actual n-gram frequencies and richer representations of words with high-dimension real valued vectors in contrast to raw frequencies. Rather than relying on n-gram frequency counts, neural networks can embed words and pass high dimensional embeddings into a neural network predicting the next word and implicitly model the co-occurrence patterns of large corpora. Typically the next word is predicted with a dense layer to the language model vocabulary normalize with a softmax distribution:

$$P_{\mathcal{M}}(t_i) = softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{k} exp(z_j)} \tag{2.6}$$

where k is the size of the vocabulary $|V|$. The output of the softmax is a probability distribution across all words in the vocabulary. Recall that the KL-divergence is a measure of how the model probability distribution is different from the second reference probability distribution. We can calculate the *cross-entropy* loss and optimize the parameters in the network using gradient-descent techniques resulting in a minimum where our model has learned the reference probability distribution. These models have proven to be powerful in language [**53**] and source code [**8**, **52**, **146**, **233**]. Recently, implicit modeling has grown to multi-billion parameter models [**200**] and are getting larger. In the next section we will discuss household model architectures, including the transformer architecture used in all three applications.

2.4 Language Modeling with Neural Networks

2.4.1 Large Corpora Word Vectors

Embedding works like GloVe [**165**], ELMo [**168**], Word2Vec [**147**], and FastText [**27**] learn semantic relations of text from large corpora to improve representations fed into neural networks traditionally initialized with random parameters. Word2Vec shifted the embedding paradigm from one-hot vectors, count vectors, and tf-idf vectors to vectors where syntactic and semantic relationships are defined specifically with continuous bag of words (CBOW) and Skip-Gram architectures. GloVe vectors combined meaningful global and local contexts by constructing ratios of relations between different combinations of words; in theory mimicking calculations between various n-grams across the corpora. ELMo embeddings are crafted by language modeling corpora with two biLSTMs and concatenating each LSTM to form an embedding. FastText breaks down words into n-grams, similar to subtokenization, so that more words can be expressed in the vocabulary by composition. These methodologies are effective for randomly initialized neural networks as they provide semantic and syntatic clues in the input. These embedding techniques have been used with code [**12**], however, with the advent of large scale pre-training, we find that transformer-based pre-trained embeddings

are equally informative for code and appear in the three presented works. In the following sections we look into two frequent architectures that leverage pre-trained embeddings and the pre-training process for highly parameterized transformers.

2.4.2 Recurrent Neural Networks

Recurrent Neural Networks or RNNs are generally considered a class of neural networks that are dependent on the output of the previous computation. Formally they can be represented by the formulas:

$$h_t = RNN_{enc}(x_t, h_{t-1}) \tag{2.7}$$

$$s_t = RNN_{dec}(y_t, s_{t-1}) \tag{2.8}$$

where h_t is the encoder hidden state at time step t for the input token embedding x_t. The decoder hidden state s_t is found by combining the previous output s_{t-1} and the input token embedding y_t.

They are frequently used for sequence data as their representation encodes the entire sequence. Due to their recurrent nature and the back-propagation of entropy through long inputs, traditional RNNs suffer from vanishing [**84**] and exploding gradients. The ReLU activation function helps maintain reasonable gradients by bounding the output for better gradient propagation, having sparse activation, and being scale invariant. Still the ability to model long term sequences with RNNs was not reachable; LSTMs and GRUs are sophisticated recurrent cells that improve modeling long sequences. Typically RNN_{enc} and RNN_{dec} are either LSTMs or GRUs.

2.4.3 Long Short-Term Memory

The LSTM cell has an input, output and forget gate that allows the network to decide how much of the past should be remembered in the hidden state. The forget gate uses the sigmoid function and looks at the previous state and content input and outputs a 0 to forget and a 1 to remember for the recurrent cell state. This allows the network to forget the context it had previously seen. The input gate also uses a sigmoid to determine which values from the input are added to the current

state. The sigmoid decides which values to let in and the tanh function gives weight to the input. Formally they are characterized by the following formulas:

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\hat{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \hat{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

with initial values for cell state hidden state is 0. Specifically, x_t is the input vector, f_t is the forget gate activation, i_t is the update gate activation, o_t is the output gate activation, h_t is the LSTM output vector, $\hat{c}_t$ is the input cell activation, c_t is the current cell state, and W, U, and b are the weight matrices and a bias vector. The controlled manipulation of the networks cells allow the network to model a diverse set of inputs better and preserve input longer in the cell state.

2.4.4 Gated Recurrent Unit

Like LSTMs, Gated Recurrent Unit networks (GRU) have become a standard in recurrent neural networks. The GRU solves the vanishing gradient problem of a standard RNN with update and reset gates that choose what information should be passed to the output; similar to how LSTMs use Forget and Input gates. The update gate receives a sum of the previous hidden state and input and decides *which* information will pass into the final state. The reset gate determines how much information the model should forget; it accomplishes this by squishing the combined input and previous hidden state with a sigmoid and recombines the salient values with the hidden state. The now salient hidden state features are recombined with the input and scaled with a tanh to indicate what combined input and hidden state values are important. Finally the output is calculated with

the input, hidden state, and update gate. Formally the equations are as follows with $t = 0$, $h_0 = 0$

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$

$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$

$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t$$

where x_t is the input vector, h_t is the output vector, $\hat{h}_t$ is a candidate activation vector, z_t is the update gate vector, r_t is the reset gate vector, and W, U b are parameter matrices and bias vector.

GRUs are able to store and filter the information using the update and reset gates and can pass selectively choosen information to pass to future time steps. While GRUs (and LSTMs) help improve sequential neural modeling with complex cell logic, they are unable to randomly access input (contextualized input) at each step making long term computations difficult. In the next section we will discuss the innovation of transformers and the pre-training techniques that followed.

2.4.5 Transformers

Transformers were designed to handle sequential sequences much like RNNs. However transformers do not necessarily process information in order and must be accompanied with a *positional encoding* vector to incorporate word sequence. The transformer *sees* the sequence as a whole and can identify contexts that confer meaning for that particular word. In the sense that skip-grams identify relevant immediate contexts for words, in transformers, the words are attempting to determine which contexts are relevant to themselves with the availability of a global context window. Transformers seek to elegantly bypass the vanishing gradient problem with attention.

Attention mechanisms were introduced in RNN architectures [**18**] but found widespread adoption in Vaswani et al. [**222**]. When added to RNNs prior to a feed-forward layer, the attention mechanisms increased performance. Vaswani et al. revealed that attention was powerful on its own and that recurrent processing was not necessary. Specifically, the attention used in [**222**] as scaled dot-product attention. The scaled dot product attention formula is as follows,

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.9}$$

where the transformer learns three weight matrices W_Q, W_K, W_V. W_Q the query weights, W_K the key weights, and W_V the value weights are multiplied with the input word embedding x_i to produce vectors $q_i = x_i W_Q$, $k_i = x_i W_K$, $v_i = x_i W_V$. The attention weights are calculated using the query and key values where the attention weight a_{i_j} is the attention from token i to token j (directional, square matrices). This allows the attention to be directional and non-symmetric. We can think of the attention vector as an alignment of tokens projecting in similar directions. The values are then divided by the square root dimension of the key vectors, $\sqrt{d_k}$ which improves gradient stability. The softmax normalizes the values into a vector that sums to 1 and is applied to the transformation of the input.

The dimensions of the input in which the transformer can attend to can be doesn't have to be static. There might be regularities that happen between tokens of different dimensions i.e channels of information. By defining multiple heads to attend to varying definitions of relevance in the embeddings, the transformer can focus on simultaneous correlations across the window. The dimension of W_Q, W_K, W_V is called an attention head and the dimensions are determined by the number of dimensions divided equally by the number of attention heads. The attention heads have been demonstrated to point to meaningful correlations like direct objects [**46**]. Additionally the attention heads can be computed in parallel as they are later concatenated together for the final feed-forward layers. Due to the $\mathcal{O}(n^2)$ computation cost of attention, there is a large field dedicated to reducing the computational cost for longer sequences [**21**, **42**, **43**, **83**, **112**, **128**, **212**, **227**, **238**]. Now we look into another growing field of pre-training transformers [**31**, **55**, **134**, **175**, **176**, **177**, **246**] which many have been adopted for code [**3**, **62**, **137**, **228**].

2.4.6 Pre-training Objectives

Language model pre-training has been effective in many natural language tasks [**50**, **55**, **85**, **168**, **175**] and has been successful in code [**62**, **102**, **228**]. A common theme is that the NLP and code models are *pre-trained* to model complex characteristics of token use, syntax and semantics, and how

these use vary across various linguistic, and formal (code), contexts. Some embedding approaches, such as [**168**], are capable of modeling varying contexts. For some tasks, transformers replaced RNNs, typically using pre-training input embeddings, due to absolute performance improvements. Transformers could be pre-trained with the causal language modeling task and seem to grow with no limit [**175**] in layers and internal dimensions. BERT [**55**] introduced a learning objective called masked language modeling (MLM), where the transformer could model language with both forward and backward directions. This bidirectional encoding boosts the models performance as it could be trained end-to-end with the MLM objective and then fine-tuned on a plethora of representation based tasks. We exploit this this network and the pre-training objective function for language modeling JavaScript and adjust the model for sequence tagging in TypeScript. In another work we test variations of widely adopted pre-trained networks for semantic properties using probes. Let's dive into the masked-language-model objective.

2.4.7 Masked-Language-Modeling (MLM)

Recall in Equation 2.1 the context for modeling token t_i was all previous tokens $t_0, \ldots, t_{i-1}$ and later a context window $t_{i-l}, \ldots, t_{i-1}$ where l is the window length. In both, the context is causal in that the model cannot predict token t_i with any *future* tokens $t_{i+1}, \ldots, t_n$ where $n = |S|$ *viz.* the length of the sequence. The mask-language-modeling objective trains the model with both *forward* and *backward* context in order for the model to be contextually aware of tokens on either side of the sequence. This is particularly useful for tasks that are **not** causal such as token tagging or sequence classification. With masked-language modeling, in contrast to Peters et al. [**168**], the MLM is training deep bidiectional representations rather than shallow concatentations of independently trained left-to-right and right-to-left LMs. The training procedure follows below.

The MLM objective masks, within the input sequence, a percentage of the tokens at random. The model's objective is to use the surrounding context to predict the masked tokens; this is primarily how the network learns language context. The masked tokens, if sub-tokens, are all masked so that a single word is entirely masked. If not whole-word-masked, parts of sub-word tokens would make the prediction task too easy and force the model to bias towards figuring out a singular missing subtoken with the immediate partial context. The network has a language modeling classification

head with an output softmax over the vocabulary. In BERT, the tokens are masked 15% of the time and the language modeling predictions are done over the masked words only. This is in contrast to auto-encoders that recreate the entire input [**223**]. After pre-training, the network can be fine-tuned on a specific task or the embeddings from the hidden state, the *context vector*, can be used in a fashion akin to GloVe, ELMo, FastText, and Word2Vec.

While BERT is incredibly useful as an encoder, it cannot be used in a generative fashion like a language model: it has seen the *right* context it would be predicting. T5 [**177**] introduced a way to use left and right context in an encoder to generate corrupt spans in a sequence with a transformer-based encoder-decoder. These spans can be generated with traditional tools like beam search. We use a version of this work when generating function calls in Chapter 5. CodeT5 [**228**] has the additive benefit of being pre-trained on large corpora of code thus resulting in a diverse and often syntactically correct set of predictions. We follow this section with alternative representations of code, in contrast to simple token representation, that illustrate the ways code differs from text. We highlight that each of the previously discussed methodologies for representing text do not consider code specific attributes such as the changing values of words *viz.* variables. The next section discusses some advanced code modeling techniques and closes with how code presents several challenges.

2.5 Advanced Modeling

The differences between text and code has piqued the interest of both machine learning and software engineering communities. Improving our representations of code for machine learning models facilitates improved performance universally. Several successful approaches have thrived by incorporating project and task specific information. Allamanis et al. [**9**] uses graphs to encode syntax nodes in the AST and syntax tokens. The work also captures control and data through the program by connecting hyperedges depicting specific relationships such as last read/write and dependency information like computed from. [**9**] demonstrated that long distance dependencies like return statements, typically difficult with transformers, can be encoded as a hyper edge to the graph representation. The gated graph neural network is then optimized in a series of message passing steps. Hellendoorn et al. [**80**] introduces a hybrid of graphs and transformers with two architectures

designed to capture the local and global relations in source code. Graph-sandwich models alternate running RNN token values for nodes in terminals in the graph and then pass those nodes back to the RNN alternating between graph and token representations. The second hybrid model Graph Relational Embeddings Attention Transformer uses a graph fundamentally for structural bias but then passes $\vec{e}_{ij}$ embeddings through a linear transformation as a bias to the traditional "relative" position bias term in Shaw et al. [**195**]. Other methods encode more common structural terms like ASTs into RNNs [**247**] and transformers [**110**, **163**, **199**]. In summary, many applications have benefited from machine learning models that better fit to code properties and permit the flexibility of defining specific code properties [**57**, **202**, **221**].

2.6 LLMs, Prompting, and RLHF

LLMs

Code assistant tools like OpenAI's Copilot use large language models [**40**] pretrained on massive corpora of open source projects. While excellent in code synthesis tasks, these models are vulnerable by the code they train on; they can inherit the same vulnerabilities and defects [**16**, **51**, **160**, **245**]. Open source code has bugs and vulnerabilities and can even be subject to exploitation by prompt engineering. The wide adoption of these models has enabled developers to new processes and ideas while posing significant risks [**76**]. Studies have shown that developers may not fully understand the code from LLMs and introduce bugs and vulnerabilities unknowingly. Worse, since they did not write the code, finding the bugs and vulnerabilities is more challenging and time consuming [**219**]. Chapter 7 discusses a growing body of work on this subject. For now, the key takeaway is this: Models that complete the most likely token by optimizing MLM-like functions are unaware of the greater implications of *some* completions. *Developer aligned* general purpose models *must* prioritize "implication-free" code; implication free code is free of bugs, vulnerabilities, code smells, etc. With LLMs exploding in popularity, a large focus is on conditioning these models for better outcomes by aligning them to human values and positive outcomes like secure code [**76**]. Some of these alignment approaches gaining traction are: prompting [**201**, **226**], prefix-tuning [**76**, **124**], and RLHF [**19**, **156**, **237**].

Prompting and Prefix Tuning

LLM task performance depends greatly on the quality of prompts used to steer the model. The effort of humans experimenting and engineering effective prompts is called *prompt engineering*. At the time of this writing, there are job listings for prompt engineers [**174**]. Brown et al. [**31**] showed back in 2020 that prompt design or "priming" is surprisingly effective at modulating a LLM in zero shot, one shot and few shot scenarios. Since Brown et al. , Shin et al. [**198**] proposed Autoprompt, a search algorithm over the discrete space of words guided to discover an optimal prompt from training data. Li and Liang et al. [**124**] proposed prefix tuning, a method that adds prefix activations prepended to each layer of the encoder stack; then only these parameters are tuned. Lester et al. [**120**] demonstrates prompt tuning via continuous vectors is an effective mechanism for improving performance on specific tasks. Lester finds promising results by freezing the pretrained network and allowing only k tunable tokens be prepended to the input text. The "soft prompt" nomenclature comes from the nondiscrete nature of the words they are optimizing. A similar paper for code was published by Wang et al. [**226**] where various hard and soft prompting techniques were tested on code intelligence tasks. Shrivastava et al. [**201**] use repository level data for prompt generation to improve 36% over Codex and further trained a model to automatically produce such effective prompts. For a complete survey on prompting techniques, we refer the reader to the paper by Liu et al. [**131**]. Adaptations for code is a very active sub-field.

Reinforcement Learning and human feedback

Some of the earliest work in deep reinforcement learning with human feedback (RLHF) dates back to 2017 [**44**, **140**]. Since then, the sub-field has exploded due to it's demonstrable impact on large language models. Since then there have been many notable papers using human feedback for RL-agents and LMs. Prior to 2022, Ziegler et al. [**250**] introduced using RL finetuning on language models and extended early work to pretrained models and KL regularization. Stiennon et al. [**205**] used human preferences (ranking) on high quality data and found that a ranking strategy produced summaries on par with human reference summaries. OpenAI experimented with the RLHF enabled models recursively for book summarization in a hierarchical fashion [**236**]. In 2022 alone there was significant focus on RLHF for LLMs. Bai et al. [**19**] and Anthropic released a paper on using RL

with AI Feedback (RLAIF) for harmlessness and helpfulness. Recently ChatGPT and GPT-4 [**155**] are using feedback from humans, but amount of RL is unclear from the technical report. In code, recent works like Chen et al. [**39**] improve code generation by using natural language feedback and unit tests. Xia and Zhang [**237**] use conversational APR with reinforcement learning to improve the performance of code generation. These lines of work will continue to grow especially as techniques mature in NLP and matriculate into AI4SE.

It is my hope that the previous section helped familiarize yourself, the reader, with the current standing of the field. This concludes the background needed to grasp the future chapters. In the first completed work, we discuss a shift in machine learning towards large scale pretrained transformer models, and the first to do so in type inference.

Chapter 3

Application 1a: Type Inference with Pretrained Models

3.1 Preface

Modern programming languages usually have a type system which determines the way variables, parameter, function return types should be interpreted. Some languages like Java and C require the developer to explicitly annotate each variable with its *static* type; this is often for memory allocation i.e `char`, `int`, `float`, and `double`. In many popular programming languages there are no such requirements and it is up to the run-time interpreter to determine the correctness. Developers enjoy *dynamic* typing because it is great for prototyping and simplifies development. However a lack of type constraints introduces bugs, difficult to solve dynamic run-time errors, and harms maintainability [**65**]. Learning to infer these types without a programmer extends the static type checkers, reduces the "type annotation tax", and can be used to find existing type errors [**65**]. Models for type inference must learn semantics of coding, through exploitation of the naturalness hypothesis, and the programs existing types *viz.* the existing formal constraints. Recently, inductive biases have gotten quite complex for code tasks with nuanced graph hyperedges in attempts to gain an edge in program understanding. In this chapter, we discuss how simple inductive biases and pretraining perform remarkably better than complex graphs [**230**] and formal constraints [**157**] for

type inference. This chapter is based on the published work *Learning Type Inference: Is Big Data Enough?* appearing at the *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Visions & Reflections Track* which I lead as the main author with Premkumar Devanbu and Toufique Ahmed serving in an advisory capacity.

3.2 Summary

TypeScript is a widely used optionally-typed language where developers can adopt "pay as you go" typing: they can add types as desired, and benefit from static typing. The "type annotation tax" or manual effort required to annotate new or existing TypeScript can be reduced by a variety of automatic methods. Probabilistic machine-learning (ML) approaches work quite well. ML approaches use different inductive biases, ranging from simple token sequences to complex graphical neural network (GNN) models capturing syntax and semantic relations. More sophisticated inductive biases are hand-engineered to exploit the formal nature of software. Rather than deploying fancy inductive biases for code, can we just use "big data" to learn natural patterns relevant to typing? We find evidence suggesting that this is the case. We present **TypeBERT**, demonstrating that even with simple token-sequence inductive bias used in BERT-style models and enough data, type-annotation performance of the most sophisticated models can be surpassed.

3.3 Introduction

Gradual typing [**45**, **203**, **224**] is gaining popularity, in programming languages like Python and JavaScript. Developers can *incrementally* type-annotate identifiers to better document, check, and maintain code [**113**]. Type annotation promotes error-detection, [**66**, **158**] while enabling more optimizations, and better IDE support. However, with type declarations existing in various library packages and project-specific locations, migrating dynamically typed software to gradually-typed paradigms is a non-trivial task, often requiring considerable human effort.

TypeScript *transpiles* type-annotated code into JavaScript (JS) which provides the benefits of typing wherever traditional JS is used [**24**]. A lot of TypeScript annotated code is available; this raises the opportunity to train *probabilistic type annotators* to help with type annotation. This idea of training a type annotator using data from manually annotated code, has been widely

applied [**78**, **142**, **183**, **230**]; except for Raychev *et al* [**183**], most use deep-learning methods. Each probabilistic annotator features a specific choice of representation, viz., *inductive bias,* that characterizes what and how they learn. Inductive biases are important, consequential, and well-studied. But do more complex inductive biases help? Do they perform better?

Recently, in NLP [**55**] and code [**62**, **102**], an alternative paradigm has emerged, to the ongoing quest for better inductive-biases: *let high-capacity models learn representations on their own* which capture the deeper statistical structure of the data, directly from very large dataset, using a form of self-supervision. In the case of NLP, Devlin *et al* [**55**] exploit giga-token textual corpora to construct a vector representation of token sequence patterns, by learning to reconstruct artificially masked-out tokens. This approach elegantly bypasses the debates on inductive-bias engineering, and simply lets high-capacity neural models *autonomously learn the statistical structure of the data* via simple, giga-scale self-supervision.

Autonomous representation-learning (aka pre-training) has been used for code. Feng et al. [**62**] used pre-training to improve performance on code-natural language bi-modal datasets (*e.g.* code with comments) and Kanade et al. [**102**] used pre-training to help with retrieval-like tasks. Type annotation is of particular interest: types are a subtle semantic property of code; one might reasonably expect that complex inductive biases leveraging syntax & semantics would be very helpful. Prior work has indeed heavily leveraged increasingly sophisticated inductive biases, with better and better results. But are these really necessary? Can models learn good enough representations *on their own?* This motivates our RQs.

RQ1: Does BERT-style pre-training work for type inference, and how does the performance compare with models that use sophisticated, custom-designed inductive biases?

Pre-training helps our TypeBERT reach 89.51% accuracy on common (top-100) types compared to the state-of-the-art LambdaNet (66.9% for the same types). Furthermore, despite the limits of a closed type vocabulary, TypeBERT does surprisingly well on user defined types. Overall, TypeBERT achieved an overall accuracy of 71.12% to LambdaNet's 64.2% across both common and user-defined

types.

RQ2: Qualitatively, what cues does TypeBERT appear to use for its inferences, and what inferences does it make?

TypeBERT seems to use multiple features for type inferences. Like TypeWriter [**171**], it appears to leverage names of identifiers; like LambdaNet *etc.* [**183**, **230**] it appears to use data and control-flow information. Using these cues, TypeBERT predicts types with specificity, *i.e.* `tf.Tensor` rather than `Tensor`. Overall, our qualitative analysis suggests that TypeBERT implicitly learns complex inductive biases like data/control flow, even without explicit graph representations.

Our models and datasets are publicly available[1].

3.4 Related Work

LambdaNet [**230**] (like other recent works) used sophisticated inductive biases [**10**, **11**, **68**] to achieve state-of-the-art (SOTA) type inference, improving substantially upon earlier approaches like Hellendoorn [**78**] and Malik et al. [**142**]. LambdaNet uses graph neural networks (GNN) to model abstract dependency graphs, derived by analysis of the code. Typilus [**11**] also uses GNNs, but includes vector embeddings to allow an open type vocabulary (for Python). Pradel [**171**] combines a probabilistic guessing component with a typechecker that verifies the proposed annotations. OPTTyper [**157**] achieves performance close to LambdaNet, by optimizing formal type constraints that are first "slackened" into numerical constraints; however OPTTyper is limited to the top 100 most frequent types ignoring the challenge of annotating user defined types.

Related to this work are highly parametrized pre-trained transformer models like CodeBert [**62**] and PLBART [**3**]. These models have successfully achieved SOTA on code-related tasks by pre-training on large code corpora and fine-tuning on the specific tasks. CodeBert's effectiveness suggests that self-supervised pre-training followed by fine-tuning may also work for type inference. Our approach differs in that our pre-training is mono-lingual; we don't use any natural language, and is

[1] `https://github.com/TypeBERT/TypeBERT`

pretrained on a type-free dialect (JavaScript) of the target language (TypeScript). Our goal was also to evaluate if pre-training could learn representations powerful enough for type inference.

3.5 TypeBERT

TypeBERT uses pre-training to learn JavaScript syntax and semantics by modeling token co-occurrences.

3.5.1 *Pre-Training TypeBERT*

Pre-Training Corpus

TypeBERT is pre-trained on a large corpus of JavaScript. We collected the most-starred 25,000 Github JavaScript projects, using the GraphQL[2]. To avoid bias, we remove duplicate snippets using Allamanis's method [**7**]. We remove non-code related entities like block comments and copyright blocks. The corpus is tokenized with a SentencePiece model [**117**] with a vocabulary of 16k subtokens. Tokenizing with Byte Pair Encoding (BPE) [**193**] or with a unigram language model like SentencePiece [**116**] is a common approach to manage large code vocabularies [**103**].

Architecture

TypeBERT uses the same architecture as BERT_{large} [**55**]. TypeBERT has 24 layers of encoder with model dimension of 1024 and 16 attention heads (340M parameters). Finally, we add an output classification layer for the type inference task (after pre-training).

Noise functions

Pre-training is self-supervised; the task is reconstructing noised-up text sequences. By training on this task, the model learns prevalent syntactic and semantic forms. TypeBERT follows BERT [**55**] where "noising" consists of randomly masking, replacing, or retaining sub-tokens. We uniformly sample (sub)tokens with a 15% probability and perform noising. Noising operations are weighted thus: 80% are masked, 10% are replaced with a random token, 10% are left alone. We allow up to 20 noise operations per token sequence. The noise function performs whole-word masking (*viz.,* all

[2] `https://graphql.org`

subtokens of a particular word are all masked if one subtoken is selected) so as to not provide too easy hints to the model. TypeBERT is jointly pre-trained with a next sentence prediction (NSP) task which is to predict if a code sequence b follows another code sequence a. For this, we select in-order or random pairs (in equal proportion) and train the model to label them correctly.

Input/Output Format

The input format for the pre-training step is two concatenated, randomly sampled, code sequences with a separator token [*CLS*], a_1, a_2, ...,a_n , [*SEP*], b_1, b_2, ..., b_n, [*SEP*]. Sometimes the a and b code sequences are contiguous, sometimes not; the NSP task is to distinguish these cases. For the random masking, any a_i or b_i may be noised. The [*CLS*] token's embedding is used as an aggregated sequence representation for tasks at the sequence level. As in BERT [**55**] the two sequences are separated by a [*SEP*] token.

Optimization

We train TypeBERT on 6 Nvidia Titan RTX GPUs for 200K steps. We use Adam with polynomial weight decay starting at 5e-5 and 10K warm-up. We use a dropout of 0.1 on hidden and attention layers. Pre-training takes about 160 hours (6.33 days) and was done using a modified version of Tensorflow's Model Garden. This pre-training is a one-time cost, followed by on-task fine-tuning.

3.5.2 *Fine-Tuning TypeBERT*

Type Inference Dataset

We collected 20,860 most-starred Github TypeScript projects (Table 3.1). This code contains human-annotated types within variable, parameter, function and method declarations. These types range from frequent types like `int` and `string` to library and user-defined types like `tf.Tensor` and `CoffeeFlavor`. We use LambdaNet's type parser to process type annotations, gathering both human annotated and compiler-inferred types. Following LambdaNet [**230**] and OPTTyper, we only use the human annotated locations for evaluation but include the compiler-inferred types in training data. Furthermore, as in prior work, we ignore locations with the uninformative "`any`" type. Of LambdaNet's full 300 project dataset, 275 could be found on Github. From these, we sample 60

TABLE 3.1. Type Annotation Datasets

Dataset	Projects	Files
TypeBERT	20,860	1,473,418
LambdaNet | OPTTyper	275	91,228

* LambdaNet / OptTyper parsed projects and files for comparison.

projects for testing, and just add the other 215 to our training set. This results in a total of 20,384 projects for training, 416 for validation (2%) and 60 projects for test. We report results on these 60 projects from the original LambdaNet/OPTTyper dataset.

Type Inference

We add a dense, softmax layer for the most frequent 40k TypeScript types and the `UNK` type for all types greater than rank 40k. The type vocabulary is closed, and restricted to these 40,001 types. TypeBERT does not handle a open vocabulary, but `UNK` occurs <8% in the test set see Figure 3.1. TypeBERT is fine-tuned on our data set consisting of > 2 million type annotations. We use de-duplication [**7**] to avoid risk of leakage from training to test.

3.6 Evaluation Metrics

We report Top 1 Accuracy (Exact Match) and Top 5 Accuracy (correct prediction in top 5 guesses) for several subsets of types exactly as with previous works.

User-Defined Types

User-defined types are type labels corresponding to class, enum, or type interface, where the type was defined within the same project scope (as in [**230**]). A class defined within the project would be considered a user-defined type; an imported library would not.

Top 100 Types

Top 100 types are highly frequent types (such as `int` and `string`) that are not user-defined, and

TABLE 3.2. Accuracy Comparison across Various Sets of Types.

Model	Top 1 Acc %				Top 5 Acc %			
	User Def	Other	Top 100	Overall	User Def	Other	Top 100	Overall
LambdaNet [**230**]	**53.4**	N/R	66.9	64.2	**77.7**	N/R	86.2	**84.5**
OPTTyper [**157**]	N/R	N/R	76	N/R	N/R	N/R	N/R	N/R
TypeBERT	41.40	50.49	**89.51**	**71.12**	55.02	70.34	**98.51**	81.88

* `UNK` (OOV) annotations are always counted incorrect for TypeBERT. Overall includes User Def and Top 100 and reported directly from [**230**] and [**157**]. N/R → Not Reported in the original paper. Allamanis [**7**] deduplication method applied on train and test sets for TypeBERT results.

are within the top 100 ranks [**157**, **230**].

Other Types

Other types are types occurrences that do not occur within the top 100 most frequent types and are not user defined. Examples would be library functions like `tf.Tensor4D`. This set of locations *were ignored* in previous works [**157**, **230**] and are not included in their reported results. We consider them, and report it separately.

Overall

We calculate an overall weighted average of user-defined type occurrences and top 100; this is calculated exactly as in [**230**], and is strictly comparable (See Table 4.3).

Unknown (OOV)

Unknown types are type occurrences which are types with rank > 40,000. We score occurrences of `UNK` (<8%) as incorrect predictions. `UNK` locations are comprised of a mixture of user-defined and other non user-defined types. Counting `UNK` occurrences as wrong for user definition, other, and overall is conservative but appropriate.

3.7 Results

RQ1: Comparing TypeBERT's type inference accuracy to SOTA.

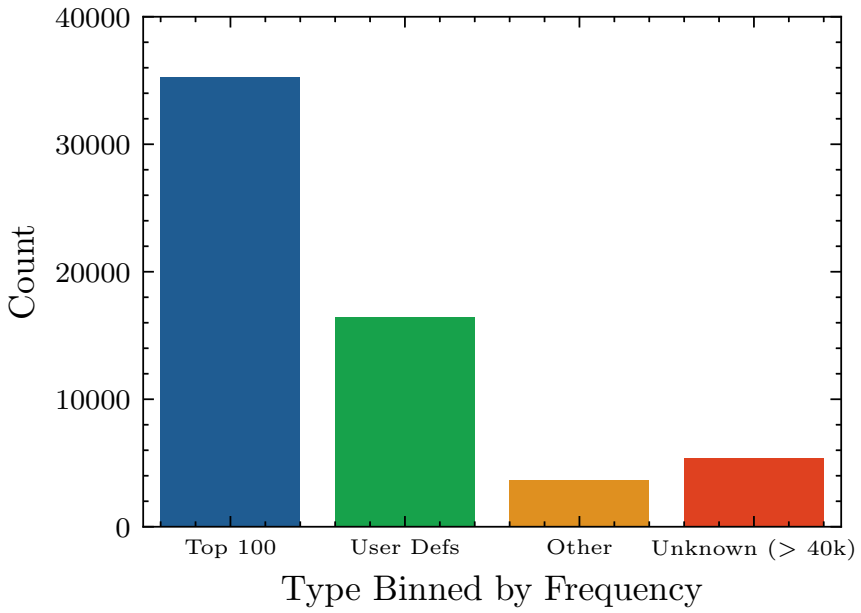


FIGURE 3.1. Frequency of types in bins. Types that exceed the Top 40,000 are marked `UNK` and scored incorrect in metrics.

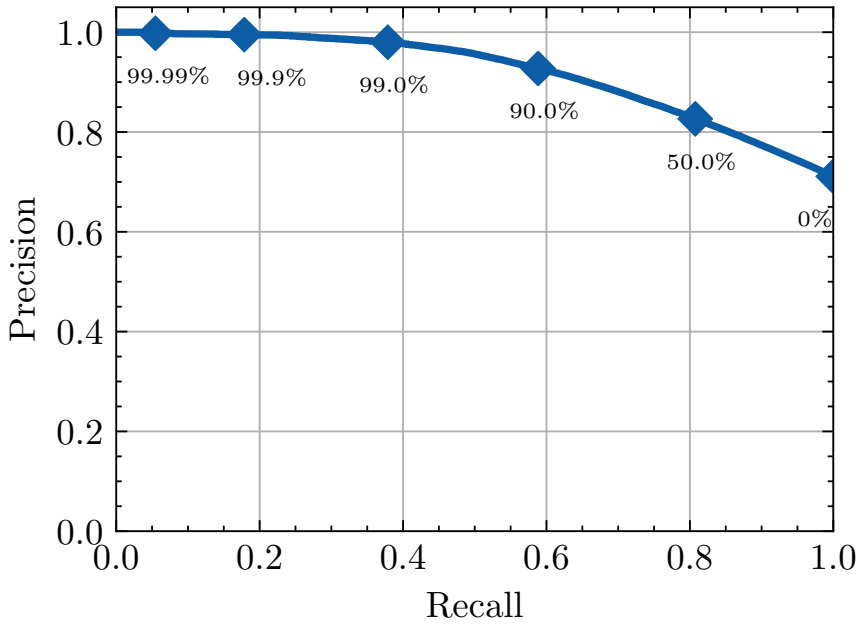


FIGURE 3.2. Recall vs. Precision of TypeBERT on test data subject to probability thresholds reflecting the models confidence.

A: https://github.com/justadudewhohacks/face-api.js/blob/master/src/faceRecognitionNet/convLayer.ts

```
function convLayer(
  x: tf.Tensor4D, params: ConvLayerParams, strides: array,
withRelu: boolean, padding: string
): tf.Tensor4D {
  const { filters, bias } = params.conv;
  let out = tf.conv2d(x, filters, strides, padding);
  out = tf.add(out, bias);
  out = scale(out, params.scale);
  return withRelu ? tf.relu(out) : out
}

export function conv(x, params) {
  return convLayer(x, params, [1, 1], true)
}
```

TypeBert

convLayer		**x**		**params**	
tf.Tensor4D	84.36%	tf.Tensor4D	99.12%	UNK	55.22%
Tensor	2.73%	tf.Tensor2D	0.39%	ConvParams	0.78%
tf.Tensor2D	2.37%	TensorInfo	0.07%	object	0.44%
object	2.30%	Tensor4D	0.05%	SeparableConvParams	0.20%
Tensor4D	2.21%	tf.Tensor3D	0.03%	Params	0.13%

strides		**withRelu**		**padding**	
array	98.93%	boolean	99.9886%	string	98.911%
Array	0.39%	Boolean	0.0095%	UNK	0.356%
ReadonlyArray	0.27%	string	0.0008%	PaddingMode	0.016%
number	0.15%	number	0.0005%	number	0.013%
Set	0.07%	UNK	0.0003%	String	0.010%

LambdaNet

strides		**withRelu**	
Function	12.45%	MtcnnBox	9.64%
String	10.14%	BoundingBox	5.44%
Number	8.44%	ComputeAllFaceDescriptorsTask	5.34%
NetInput	6.68%	FaceLandmarkNet	5.31%
Object	5.70%	DetectAllFaceLandmarksTask	4.86%

FIGURE 3.3. Qualitative evaluation of TypeBERT and LambdaNet.

Table 4.3 shows results on the test set of projects in LambdaNet/OPTTyper dataset. LambdaNet [**230**] serves as our baseline because it is evaluated on both the top-100 most frequent types and on user-defined types. Its use of a sophisticated graph inductive bias makes it a good contrast for our

pre-training/fine-tuning approach, with a very simple token-sequence basis. TypeBERT betters LambdaNet on the top-100 (89.51% Top 1 vs 66.9%), despite the disadvantage of a much simpler inductive bias. This suggests that large scale pre-training helps TypeBERT autonomously learn nuanced, rich contextual representations that rival LambdaNet's complex hand-engineered hyper-edges. LambdaNet's does excel on user defined types; still, TypeBERT (even without any mechanism for user definitions) achieves a higher Top 1 overall accuracy (71.3% vs 64.2%). TypeBERT's Top 5 accuracy on the top 100 types (98.5%), compared to LambdaNet's (86.2%) is significantly better demonstrating more relevant predictions across a set of five recommendations; for developers this means more relevant choices to choose from.

While TypeBERT demonstrates high Top 1 and Top 5 accuracy, it also performs well with high confidence. Figure 3.2 shows the trade-off in precision and recall when varying the confidence threshold. Precision exceeds 92.72% with a threshold of 90% with a recall of 58%. At a threshold of 99%, precision exceeds 98% with a recall rate of 38%. TypeBERT could add ca. 22,000 of the ca. 58,000 annotations across the 60 test projects with very high precision. TypeBERT is quite fast: on a single Nvidia Titan RTX can perform type inference on 16,384 locations in a batch of 64 sequences of length 256 in just 1.28s or .02s per sequence. LambdaNet, we note, requires a dependency hypergraph: computing which correctly is limited by missing dependencies, libraries, and type definitions; thus it cannot perform accurately at such locations. This is not a problem for TypeBERT. It's important to note that TypeBERT performs creditably on "Other" types, (50.5% for Top 1, 70.3% Top 5); this category is not dealt with by LambdaNet. Finally, we note that OPTTyper works only for for top-100 types, and we improve upon it as well.

TypeBERT improves Top-1 and overall performance **22.61%** and **7.1%** respectively over LambdaNet.

RQ2: Qualitative analysis of TypeBERT.

What evidence does TypeBERT use to make type predictions?

We examine this with an example. Figure 3.3 shows a function signature (top), with correct annotations; inferences from TypeBERT and LambdaNet below (correct inferences shown in yellow). This file imports `tfjs-core` with `import * as tf`. Thus syntactically correct types

from `@tensorflow/tfjs-core` must have a prefix of "`tf.`". TypeBERT recognizes this context and correctly infers `Tensor4D` with the appropriate prefix i.e `tf.Tensor4D`. TypeBERT maintains consistency in this example and types variable `x` as `tf.Tensor4D`; the same type and appropriate prefix. As another example, to infer boolean type for `withRelu`, TypeBERT appears to take cues from the return statement; to get array type for `strides`, it appears to be using the call to `convLayer` within `conv`. These control and data-flow oriented semantic cues are being learned implicitly from lexical sequences. LambdaNet fails to infer the return value type and cannot provide type recommendations for 4 other locations; this is likely the result of types existing outside of LambdaNet's prediction space both top 100 and its pointer mechanism.

TypeBERT shows consistency across types in the same context, extracts clues from surrounding context, and demonstrates some data-flow oriented semantic clues.

What kinds of inferences does TypeBERT make?

A characteristic of TypeBERT's top-k type guesses for an annotation are lexical and semantic similarities (Figure 3.3). This is due to the contextual usage of similar types i.e `tf.Tensor4D` and `tf.Tensor2D` and an alignment of meaning representation i.e `array` and `Set`. While TypeBERT seems highly confident when it is correct, the other alternatives tend to be relevant, and sometimes even partially-correct *e.g.* `Array` (.39%) and `Boolean` (.0095%) for `array` and `boolean`.

Finally, TypeBERT is strongly confident when the answer is outside its closed vocabulary (UNK). This confident `UNK` prediction has value: in future work, we hope to use open vocabulary mechanisms such as pointer networks or metric similarity functions to search for a better answer in such cases. This example (Figure 3.3) is typical; most often, TypeBERT's offers correct inferences with high confidence and with highly similar alternatives.

TypeBERT recommends types of meaning indicating that partially-correct types are often used in the same context.

3.8 Conclusion

We present a "big-data" alternative to the type inference problem: we use a pre-trained BERT-style model rather than custom-engineering a complex, specialized inductive bias and training dataset.

TypeBERT uses the simplest inductive bias: considering code as a sequence of tokens. The lack of input structure is overcome by increased learning capacity of the BERT-style approach. TypeBERT leverages pre-training on 25,000 JavaScript projects, and fine-tuning on 20,800 TypeScript projects. We find that TypeBERT is competitive with SOTA approaches which use much fancier inductive biases. It infers the exact type in common locations almost 90% of the time beating the SOTA models by a significant margin. Additionally, TypeBERT's Top 1 accuracy overall is better than the the SOTA, at 71.12%. Our findings suggest that TypeBERT *implicitly* learns the statistics of the semantic relationships, relevant to typing, that are made explicit in the graph-based static analysis products (*e.g.,* those used by LambdaNet). It is intriguing to contemplate that generic, automated methods can utilize additional model capacity to "learn" to do some sort of static analysis.

Chapter 4

Application 1b: User-Defined Types with Multi-Task Learning

4.1 Preface

The widespread tendency to neologism in code vocabulary (thanks to the prolific invention of new, locally-specific identifiers) compared to natural language is a non-trivial challenge in adopting state-of-the-art neural models to code related tasks. In representing source code input, probabilistic [**116**], frequency [**193**], and combinations of both [**117**] subword tokenization has ameliorated the exploding vocabulary issue [**103**]; additionally some have been proven to be more optimal [**29**]. Types, however, are introduced, like other code variables are, at the developers discretion. This newly introduced vocabulary may be different from that of natural language. This poses a problem for discrete type distributions.

While TypeBERT in Chapter 3, improved existing type benchmarks, it has a fixed categorical softmax layer. This means it can only select one type from 40,0001 types. A bounded, limited typeset cannot handle all types. We address this issue with DiverseTyper, a type inference model capable of modeling local and global user-defined types in addition to TypeBERT's common types. This chapter is based on a journal paper featured in Transactions on Software Engineering titled *Learning To Predict User-Defined Types*, which I lead as the main author with Premkumar Devanbu

and Anand Sawant in an advisory capacity. This work has recently been added to the journal first track at International Conference on Software Engineering (ICSE) 2023.

4.2 Summary

TypeScript permits a wide range of types including developer defined class names and type interfaces. These developer defined types, termed *user-defined types*, can be written within the realm of language naming conventions. The set of user-defined types is boundless and existing bounded type guessing approaches are an imperfect solution. Existing works either under perform in user-defined types or ignore user-defined types altogether. This work leverages a BERT-style pre-trained model, with multi-task learning objectives, to learn how to type user-defined classes and interfaces. Thus we present DiverseTyper, a solution that explores the diverse set of user-defined types by uniquely aligning classes and interfaces declarations to the places in which they are used. DiverseTyper surpasses all existing works including those that model user-defined types.

4.3 Introduction

Gradual typing is gaining popularity particularly in dynamically typed languages like JavaScript and Python. Typing and type-checking can find common kinds of data-misuse in programs by checking that variables, expressions, functions and modules are used in a consistent fashion. *Type systems* can verify the type safety of the program in different ways: most languages verify types either statically at compilation time, dynamically at run-time, or some combination of both. Developers have come to appreciate the benefits of type checking at run-time; benefits include faster prototyping and more flexible use of variables [**74**, **206**]. These advantages come at a cost because the resulting program has less known type associations prior to running [**135**]. Running a program with fewer type verified variables and functions results in an increased probability of uncaught type errors [**67**, **181**].

To address these concerns, gradual type systems [**30**, **203**] were proposed; they provide developers an attractive balance between static and dynamic typing. Developers can gradually add type annotations to a program, as they see fit. TypeScript is a gradually typed version of the JavaScript programming language that is gaining traction. TypeScript can be used anywhere JavaScript is used

because the type checker enforces type rules prior to *transpiling* into JavaScript. Thus, code bases can still run on the highly popular frameworks JavaScript runs on and enforce some type rules.

Unfortunately this approach also has disadvantages. The *optionality* of gradually typed languages is a double-edged sword wherein the convenience of not typing variables and functions may result in type errors that can be caught prior to deployment if properly labeled [**67**]. Consequently, researchers have been determined to develop tools that adequately help developers label types especially when it can prevents bugs.

The abundance of typed source code (in gradually typed languages) from repository sites like GitHub[1] enables researchers to use machine learning methodologies to infer types in dynamic languages [**78**, **95**, **142**, **157**, **172**, **182**]. Advancements in neural networks are helpful for software engineering tasks, including type inference [**10**,**78**,**95**,**149**]. These approaches adapt machine learning architectures as best as possible but neglect particular aspects of programming languages that are consequential to the problem. Type-inference is traditionally framed as a bounded classification task because of the natural alignment with fixed categorical classification losses in machine learning. However, types have unbounded vocabulary, as do variable and function names; so it is desirable to accommodate an *open* type vocabulary. Thus, modeling types with a bounded classification layer is overly restrictive; the model's performance is limited by an upper bound.

We approach type inference with an unbounded vocabulary very much in mind. We further hypothesize that user-defined type *declarations* contain important information that can help probabilistic machine learning methods to infer type annotations. Our implemented model, DiverseTyper, leverages two principles: *large-scale pre-training* and *deep similarity learning*.

The first principal idea, pre-training, is the practice of teaching models the form of languages by enforcing auto-encoding objectives like masked-language modeling [**54**]. Pre-trained models are ideal for efficiently encoding programming features like user-defined classes or type interfaces. The second principal idea, deep similarity learning, is used to align or associate two encodings, for example, a class declaration and the use of the declaration as a type. Our hypothesis is that such a relationship can be learned for an unbounded set of user-defined types, thus removing this artificial restriction that exists in previous methods.

[1] `https://github.com`

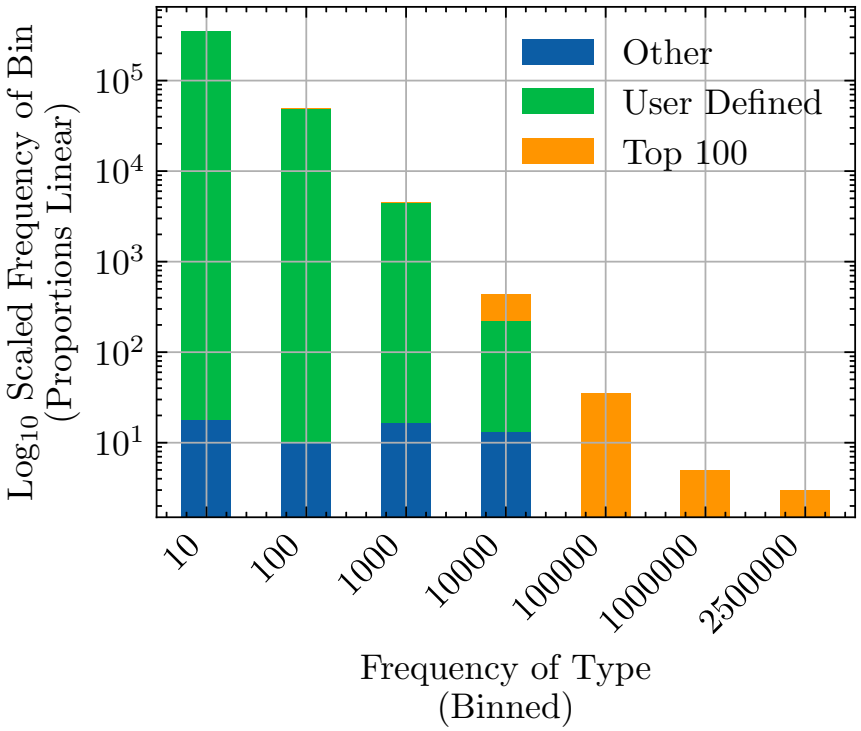


FIGURE 4.1. A type is binned by how often it's used in code in the dataset (x-axis). The histogram of bins are scaled by Log_{10} (y-axis) to see all bins. The ratios between Other, User-Defined, and Top 100 (color-coded) are linearly scaled for simplicity. For example, of the types that are used 10 times or less (first column), 77,820 are Other, and 266,882 are User-defined (22%/78%). On a log_{10} scale, the total (344,702) is between 10^5 and 10^6.

In addition to user-defined types, it is common to use *native* and *library* types. Native types like `number` or `string` and library types like `ArrayBuffer` do not have declarations, and are very frequent [**11**, **78**, **95**, **149**, **157**, **230**]. Thus the typing inference task has two seemingly orthogonal sub-tasks: learning common-types within a bounded vocabulary and aligning user-defined types to existing class and type declarations. We theorize that each task guides a neural model to learn divergent *type representations* which presents a challenge. A model learning a type representation for common-types in categorical form is equivalent to partitioning or folding embeddings across a fixed space. Additionally, a model learning a type representation that aligns declarations and annotations means clustering types into manifolds of separability. So we ask, can the model learn to selectively pick types that should be partitioned (common-types) versus clustered (user-defined types). The answer is yes! This is realized with a specially crafted training signal that balances the

representation learning of common-types and user-defined types. The resulting type inference model, DiverseTyper, can predict common and library types, while also supporting *new* types defined using class and interface declarations. This model is also capable of predicting user-defined and rare types, even if the type definitions were not seen during training. DiverseTyper is effective globally across all types but especially in the most difficult user-defined types because it diverges from previous machine learning (ML) assumptions and aligns with how developers annotate custom types. This work's contributions are,

Contributions

(1) A type inference model that adopts large scale pre-training to type-inference of common and user-defined types. DiverseTyper's adoption of pre-training helps it align new type declaration to uses of that declaration.

(2) A novel multi-task uncertainty learning approach that combines type inference classification (cross entropy) and type similarity (semi-hard triplet loss) where loss scaling is learned end-to-end.

(3) Improve type inference from state of the art approaches by 8.59% overall by improving user-defined type inference 30.16%. User-defined type inference is significantly harder than common-type inference due to its long-tailed distribution.

DiverseTyper can be found at our public GitHub[2].

In the following sections we discuss the challenges of user defined types, the underlying intuition behind our approach, and why the approach of DiverseTyper is positioned better than previously developed approaches.

4.4 Challenges of User-Defined Types

Software programs introduce new vocabulary at a higher rate than natural language [**104**], due to new identifier names, functions, classes, enums, structs, *etc.* Types also feature large vocabularies, and thus (like variable, function names, *etc*) are a challenge for models with finite type vocabularies. Figure 4.1 shows that most unique types occur less than 10 times, typical of a long-tailed distribution. The figure shows the proportions of the top-100 types, user-defined types, and other (library) types;

[2] `https://github.com/diversetyper/diversetyper`

```tsx
// types.tsx
export type CodeSandboxImport = {...};
export type CodeSandboxLanguage = 'js' | 'ts';
```

```tsx
//createPackageJson.ts
import * as _ from 'lodash';
import { CodeSandboxImport, CodeSandboxLanguage } from './types';

const name = 'fluent-ui-example';
const description = 'An exported example from Fluent UI React, https://aka.ms/fluent-ui/';

function createDependencies(code: string, imports: Record<string, CodeSandboxImport>) {...}

//DiverseTyper Top Guesses: CodeSandboxLanguage, string, Language, RemoteDebugLanguage, CloseReason
export const createPackageJson = (
    mainFilename: string,
    code: string,
    language: CodeSandboxLanguage,
    imports: Record<string, CodeSandboxImport>,
) => ({...});

//TypeBert Top Guesses: Language, string, UNK, ILanguage, LanguageCode
export const createPackageJson = (
    mainFilename: string,
    code: string,
    language: Language,
    imports: Record<string, CodeSandboxImport>,
) => ({...});

//LambdaNet Top Guesses: Unknown
export const createPackageJson = (
    mainFilename: string,
    code: string,
    language: any,
    imports: Record<string, CodeSandboxImport>,
) => ({...});
```

FIGURE 4.2. Code snippet from `microsoft/fluentui` GitHub repository. `CodeSandboxLanguage` is type defined in `types.tsx`. The type is also imported in `createPackageJson.ts`. However, both TypeBERT and LambdaNet fail to properly annotate the correct user-defined class. DIVERSETYPER references the type properly despite the type `CodeSandboxLanguage` is used only once in all repositories, *viz.*, *rare* and *infrequent*.

it's clear that most types constituting the long tail are in fact *user-defined* types. This is because user-defined types typically occur just within the scope of the project that defines them, and rarely

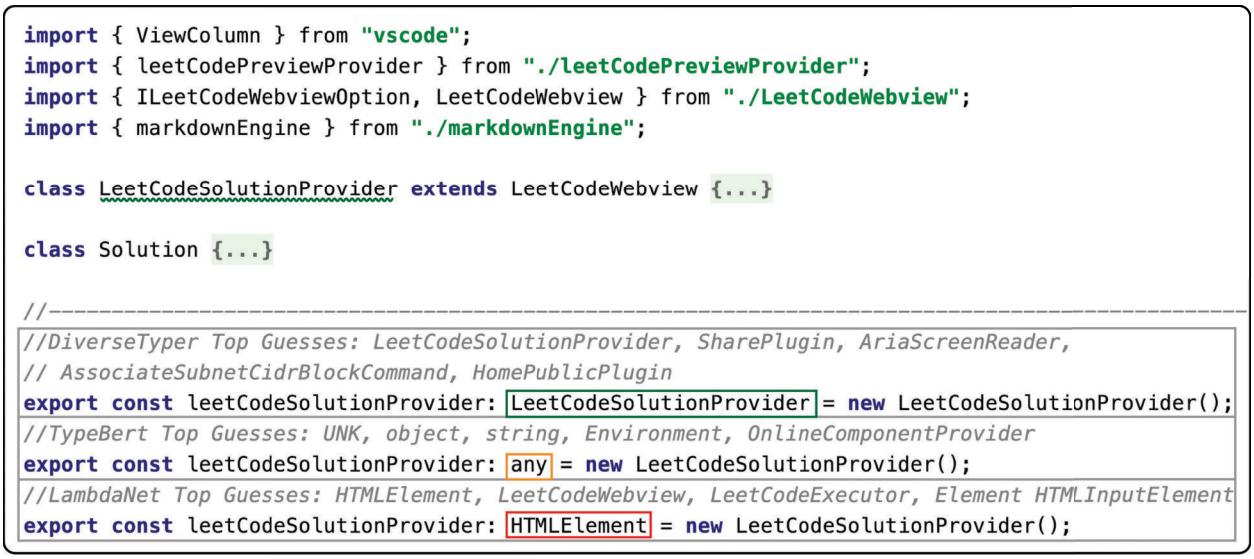


FIGURE 4.3. Code snippet from `LeetCode-OpenSource/vscode-leetcode` GitHub repository. The class `LeetCodeSolutionProvider` is declared in the same file that the class annotation exists in. Both TypeBERT and LambdaNet do not properly type this annotation. The user-defined class exceeds the bounded type vocabulary of TypeBERT so the best annotation it can do is `any`. LambdaNet seems to reference other LeetCode classes but annotates the class instance with `HTMLElement`. DIVERSETYPER gets the annotation correct.

exist elsewhere. It's evident from Figure 4.1 that type inference approaches that model a finite type vocabulary ignore a lot of types.

Close inspection user-defined type annotations (and their respective declarations) reveals that the declarations are often co-located in the same file or exist nearby (See Figure 4.2). In the example of Figure 4.2, representative of many user-defined type annotations, the compiler cannot deduce the corresponding type `CodeSandBoxLanguage` because of type ambiguity. This annotation ultimately resolves to a string and a variety of variables also are of string type. The task of correctly labeling types becomes challenging when types appear ambiguous to the compiler, say different type declarations with the same underlying string type. However, developers have a grounded common sense rooted in their experiences programming and familiarity with the language. A developer would observe the context around the type and gain familiarity with how the type should typically be used. The developer would likely see that the variable `createPackageJson` has an attribute `code` and `language` with a function `createDependencies` taking a code string, and record object that includes a `CodeSandbox` object. The developer would correlate that the words "dependencies"

and "package" often requires certain keywords like "imports" in the context of JavaScript. Lastly, any word of "sandbox" would allow the developer to narrow down the correct type even if other syntactically correct and semantically similar types exist; if `ProductionLanguage = 'js'` exists as another user-defined type, this would be syntactically correct . While no model has the same abilities to reason logically as a developer would with common sense knowledge, DiverseTyper, is designed to follow the same clues probabilistically which differs from previous approaches. In order to demonstrate the effectiveness of this approach over previous approaches and highlight our contribution across the user-defined type space, we will first discuss how the model encodes a type declaration, and then uses these powerful encodings to type variables in the main body of the code; no other approach does this, thus falling short across this diverse domain of types.

4.5 Why Other Typing Models Fall Short

The aim of DiverseTyper is try to reach a performance level closer to that of human developers. The model's understanding parallels developers by utilizing deep pretrained embeddings to encode a user-defined type; the pretraining practice is called Masked-Language-Modeling or MLM. This pretraining approach processes large swaths of raw source code available on GitHub [**3**, **4**, **62**, **72**, **102**] to learn representations (neural encodings) which capture common usage patterns. The model can use its neural encoding of source code to determine commonalities with other code tokens. With pretraining, the model in example Figure 4.2, will be guided to the words "language" and "sandbox" when guessing `CodeSandboxLanguage`. The words "language" and "sandbox" occur frequently in the context of the type `CodeSandboxLanguage` and not other types, which make these words highly indicative of this type. DiverseTyper digests any class or interface declaration and stores the embedding of the class or interface declaration as a type. DiverseTyper's novelty is that it can handle class and interface *declarations* as opposed to previous approaches that rely on learning from explicit type *annotations* already present in the code.

There are two popular approaches to recover types: (1) Models such as LambdaNet [**230**] will save the names of user-defined types and allow prediction to those names (using a pointer network). LambdaNet must determine the correct typing on very sparse occurrences of that type; as shown earlier, user-defined types are less frequent across a global set of projects. There is no

TABLE 4.1. Comparison between various learning-based type inference models

Model	Model Architecture	Type Vocabulary	User Definition Mechanism	Pre-Trained
DeepTyper [**78**]	biRNN	10,000	✗	✗
NL2Type [**142**]	LSTM	1,000	✗	✗
TypeWriter [**172**]	HNN	1,000	✗	✗
OptTyper [**157**]	LSTM	100	✗	✗
LambdaNet [**230**]	GNN	Unbounded	✗	✗
Typilus [**11**]	GNN	Unbounded	✗	✗
Type4Py [**149**]	HNN	Unbounded	✗	✗
TypeBERT [**95**]	Transformer	40,000	✗	✓
DiverseTyper	Transformer	Unbounded	✓	✓

pretrained embeddings involved so the parameterization of the model comes from sparsely learned co-occurrences of said infrequent types. (2) Other approaches, like Typilus and Type4Py, also do not benefit from pretraining and can only reference a user-defined type if appears as a previous *type annotation* (not *declaration*) in its training data. Our intuition is that models like Typilus and Type4Py cannot type as well as a human developer because it cannot observe new type declarations and understand nuances between such types without optimizing on them in a one-shot manner. To accommodate a new class declaration in Typilus and Type4Py, the model must rely either on previously seen type annotations of the *same exact type*, rather than directly computing an embedding from a declaration, and aligning that new embedding to valid type locations; this is what DIVERSETYPER does. When DIVERSETYPER digests class and interface declarations, it can use the pre-training basis to discriminate types by attributes and methods, thus deeming a type incompatible or compatible with the annotation location. With the type declarations at the disposal of the typing model, the model is able to reference a more diverse set of types and an improved performance is expected.

We are not aware of any existing approach that achieves our levels of performance for such a diverse set of types. Earlier works like DeepTyper [**78**] and JSNice [**182**], use a limited type vocabulary and focus only on common-types. More recent Python approaches Typilus [**11**] and Type4Py [**149**], expand the type vocabulary to include all types seen in training. This expansion, to all types seen in training, is an improvement; but even these approaches face a performance ceiling on new types. A TypeScript approach called LambdaNet [**230**], expands the typeset to all visible project types with a scoring mechanism; this approach most in line with our proposal, and

we do a careful comparison. A more recent approach, TypeBERT, discussed in Chapter 3, does not increase the type vocabulary, but demonstrates that BERT-style pre-training on JavaScript corpora boosts type prediction because the model learns token co-occurrence statistics relevant to typing. TypeBERT, like other fixed type vocabulary models [**78**, **157**, **172**, **182**], ignores new types. If developers were to use these tools in practice, the models will underperform on new types. Since newly defined types are common to projects per Figure 4.1, and are often key to good software design, our machine learning architecture is better-aligned to modern software development paradigms. Whenever a developer defines a new, project-specific class or type, DiverseTyper also encodes these classes and type interfaces with *pre-trained* vectors. DiverseTyper employs those representations in the type suggestion process. We explain in the following section how the pre-trained vectors improve a model's ability to capture the learnable and relevant features in code, and the benefits for machine learning based type-inference approaches.

4.6 DiverseTyper

In this section, we first introduce general pre-training for types, the training elements of DiverseTyper, followed by the inference mechanisms DiverseTyper.

4.6.1 Pre-training For Types

Pre-trained transformer models for code such as CodeBert [**62**], CuBert [**102**], PLBart [**3**], and TypeBERT [**95**] achieve state-of-the-art (SOTA) results on code-related tasks by pre-training on large code corpora followed by fine-tuning weights on a specific task. Pre-training on large corpora is compute-intensive, which is often performed at large, resource-rich organizations that can afford the cost of training [**22**, **191**]. Our work amortizes the expensive cost of pre-training by initializing DiverseTyper core weights with the pre-trained weights from TypeBERT [**95**]. The pre-trained model takes in a sequence and outputs a contextual vector for each input token. The code token's context determines the vector. The complete body of a class or type declaration provides rich information such as attributes and internal functions. This rich information can guide a type inference model to link the uses of the class or interface to its declaration; this approach is novel in this work.

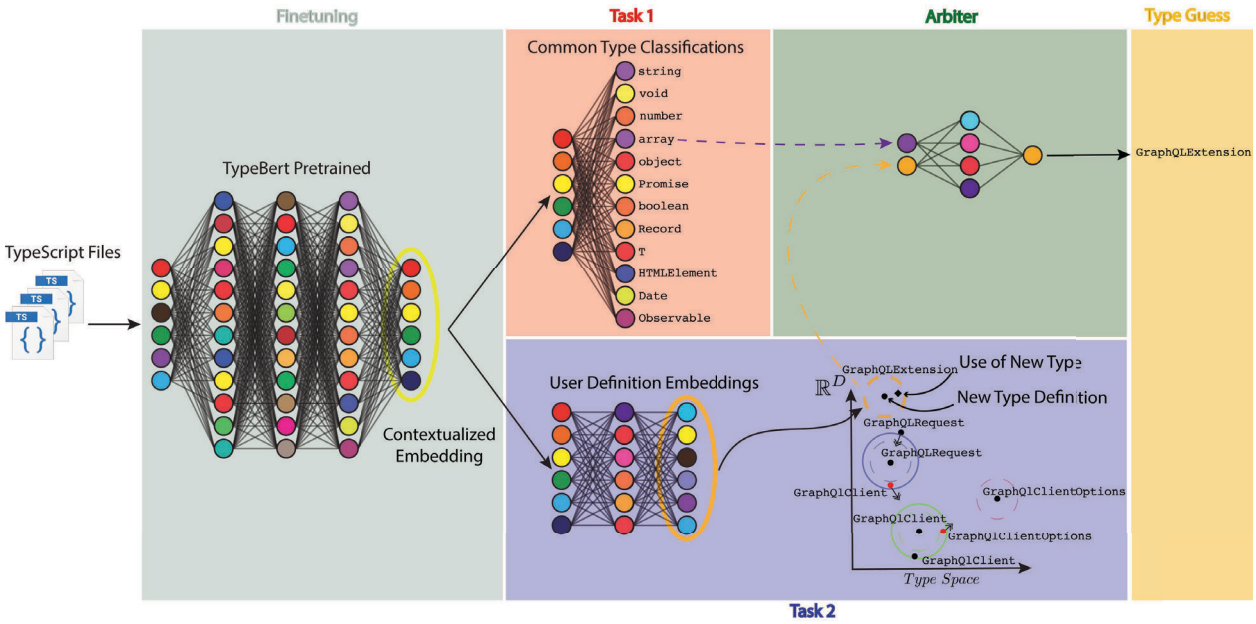


FIGURE 4.4. Overview of DIVERSETYPER. *Training*: DiverseTyper is trained end-to-end with two tasks Task1 and Task 2. Task 1: a classification layer is trained with a cross-entropy loss on the target types. Task 2: An alignment of user-defined types and the use of types with a triplet loss. A red dot indicates a type annotation that is incorrectly positioned closest to a different type. The model learns to correct this and incrementally shift the embedding to the nearest same labeled type. Type declarations, coexist in the same type space with normal type uses. *Inference*: DiverseTyper: A deep transformer, outputs a context embedding (yellow circle) which is projected into a common type guess (Task 1) and a user-defined type embedding (orange circle) corresponding to a user defined type-space (Task 2). To convert the user-defined type embedding to a type, a k nearest neighbor (kNN) search returns the nearest neighboring types. Arbiter: An independently trained multi-layered perceptron (MLP) decides which type prediction is better between the common type and the user-defined type.

To take advantage of these useful representations, DIVERSETYPER is initialized with the published TypeBERT weights, input width (256), and sub-tokenized input vocabulary trained with SentencePiece [**117**]. Tokenizing the code, with Byte Pair Encoding (BPE) [**194**] or unigram language modeling [**117**] is common to manage large input code vocabularies [**104**]. Tokenizing is not used in the type vocabulary because the model needs to output valid types. With the pre-trained weights and tokenized input inherited from TypeBERT [**95**], we are ready to define training procedure for DIVERSETYPER.

4.6.2 Training

The DiverseTyper approach leverages several key components. The first component is the **context vector** provided by the BERT-style model pre-trained on code (yellow circle in Figure 4.4. For instance, for a particular sequence of code s, each t^{th} source token s_t, has a context vector h_t existing in $\mathbb{R}^d$ where d is a dimension determined by the neural model architecture. This context vector, h_t, is the vector we propose is capable of representing common-types and user-defined types, when fine-tuned under the proper loss function and training procedure.

The second component is the **loss function** which is required in order to transform the aforementioned context vector. The loss function determines what the model learns and how efficiently it is learned; typically this is comprised with (sub) losses focused on the optimization of a particular objective. To learn common-types, the model defines a categorical learning signal where it learns to associate each token with a common type. For simplicity, we term this signal as **Task 1**. To learn user-defined types, the model uses a deep similarity learning signal we call **Task 2**. **Task 2** transforms the context vector into a new user-defined type vector (orange circle Figure 4.4), termed h_s, which can be used to compare new declarations with individual uses of these declarations. The aforementioned transformation into a user-defined type vector, h_s, requires DiverseTyper to use additional hidden layers shown in **Task 2** of Figure 4.4. We add additional layers to allow the context vector, h_t, to contort into a new representation that might differ greatly from **Task 1**, but be more suitable for **Task 2**; these layers introduce additional degrees of freedom that can be trained using the above mentioned loss function. As previously mentioned, a representation might become more or less suitable for one task over the other during the training procedure which means the model should turn up or down the amount of feedback from each task. With two losses of varying degrees of importance at a particular moment in training, the model must judiciously combine the losses in a manner that reflects the model's confidence in them. Another term for this is *uncertainty*.

While there exists many weighing strategies [**70**], we find using the inverse variance of a loss is a suitable weighing term, i.e., $\frac{1}{\text{variance}}$ *viz.* $\frac{1}{\text{uncertainty}}$. When the uncertainty is high, the model will weigh the signal less and vice versa when the uncertainty is low. The next sections will describe

these three components in detail. We first describe the two losses and consequentially how we weigh them with uncertainty.

Task 1: Classifying common-types. The first task is based on the model's ability to classify commonly occurring types; these types are often self-evident through simple expressions containing string manipulation or mathematical operations. At a high level, the machine learning model is given a sequence s and returns embeddings for each token s_t. The embedding is used as input to the classifier to produce types for each token, given the token can assume a type. With a type associated to each variable, parameter, and function code token, the model can check the predicted type with the ground truth and learn a distribution that matches the ground truth distribution. This is a popular, yet effective way to classify a large bulk of type annotations but is poor in predicting a large breadth of types. In the next paragraphs we discuss the details of the common type classifier and motivate an alternative for user-defined types.

In order to learn the ground truth type distribution, machine learning models, like DiverseTyper, require a loss or feedback from the various tasks they are trying to solve. To get a loss value for **Task 1**, DiverseTyper must calculate the following for each type annotation. Per source token s_t and a corresponding type annotation τ, DiverseTyper passes the hidden state h_t (yellow circle in Figure 4.4) to a classification layer (**Task 1** in Figure 4.4). This classification layer is defined as a probability distribution formed from the linear combination of the hidden state h_t and a learned type representation r_τ. The linear combination produces *logits* or log-odds that can be mapped to a probability distribution with the softmax equation (seen in Figure 4.5). This leads to a probability associated with each type. In machine learning terminology, this is defined below,

$$P_{s_t}(\tau) = \frac{\exp(h_t^T r_\tau + b_\tau)}{\sum_{\tau\prime}^{\mathcal{T}} \exp(h_t^T r_{\tau\prime} + b_{\tau\prime})} \tag{4.1}$$

where $P_{s_t}(\tau) \in (0, 1)$ and τ is $\in \mathcal{T}$, which is the finite set of known types. Equation 4.1 is a classic equation in machine learning for predicting probabilities from a finite set of classes. During training, these probabilities are often incorrect and must be adjusted to the underlying true probability. This is accomplished with the types' true labels.

With the probability across common types, we seek to maximize the expected probability $P_{s_t}(\tau)$ over the training set by minimizing the corresponding loss $\mathcal{L}_{\text{CLASS}}$. We use a standard classification

(cross-entropy) loss:

$$\mathcal{L}_{\text{CLASS}}(s_t, \tau) = -\sum_{\tau\prime}^{\mathcal{T}} y_{s_t} log(P_{s_t}(\tau)) \tag{4.2}$$

where y_{s_t} is the ground truth type for the source code token s_t. By optimizing this signal, the model can learn to adjust it's internal parameters for common types according to their true distribution in code. Figure 4.5, illustrates how Equation 4.1 normalizes a neural network output to probabilities.

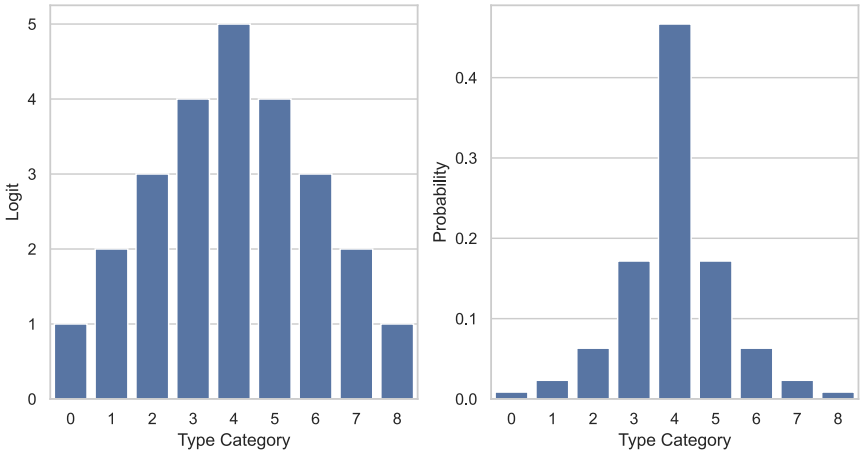


FIGURE 4.5. The softmax equation, Equation 4.1, forces model outputs into probabilities. Left: Raw values from the network regarding the log likelihood of a category. Right: Equation 4.1 forces the distribution into a probability distribution with a cumulative sum of 1.

However, infrequent and user-defined types are not represented by the model's fixed type set (shown in the red block of **Task 1** in Figure 4.4), and cannot be learned as previously described. To reiterate, this is because the model's output is finite. We address this issue with **Task 2**, capable of learning infrequent and user-defined types (blue block in Figure 4.4).

Task 2: Learning User-Defined Types. Probabilistic type inference approaches perform best when a sample of type annotations are reflective of the population. This is very difficult for user-defined types, like class declarations and type interfaces, which typically occur infrequently within the scope of a single project. This is a major reason why data driven approaches fail on user-defined types. On the contrary, if this problem is mapped into a matching task from declarations to uses of those type, then there are examples in practically every project. Despite class declarations and corresponding uses being different code entities, according to the contextual embedding, we

can establish an alignment task and adjust those distant embeddings to be translated into nearby embeddings. Thus, DiverseTyper can leverage rare types into many good training examples of matching across thousands of projects irrespective of the sparsity of individual types across the entire corpora. In the following paragraphs, we examine how the aforementioned similarity is defined and learned by DiverseTyper.

To learn infrequent and user-defined types, the model uses deep similarity learning [**73**] to associate type declarations with the respective annotation. In machine learning, we can define similarity in many arbitrary ways as similarity is a subjective measure. We use a loss that compares embeddings to other embeddings with respect to the embeddings' labels (again could be subjectively assigned). If two embeddings correspond to the same item, according to the label, and the distance between the embeddings is large, then the similarity would be low when it should be high; this can be corrected with the model producing *better* embeddings. To elucidate a more formal concept of similarity, we first introduce the notion of a *triplet*, the fundamental building block to **Task 2**.

Similarity for types is a relative measure defined by grouping same and different types. For a particular type x_t, the model finds a type x_t^+ with the same label, and different type x_t^-. Together these three elements define a triplet as (x_t, x_t^+, x_t^-). A distinct property of a triplet is that there is a notion of similarity between the reference point or *anchor* x_t, the *positive* point x_t^+, and *negative* point x_t^-. In Task 2 of Figure 4.4, the black points are same labeled types with the *anchor* being the focal point of the circle. The red dots are *negative* points where the label is different than the anchor. The red points should be moved closer to the center of the correct point through the optimization of a training loss. In Figure 4.4 the negative examples have arrows to demonstrate the direction the model is moving the points in order to correct the prediction.

To use the triplet (x_t, x_t^+, x_t^-) for learning user-defined types, DiverseTyper randomly constructs triplets from embeddings it produces (orange circle in Figure 4.4). In **Task 2** of Figure 4.4, the types anchor and positive will be annotations that are both the same, i.e. `GraphQlClient` and another annotation of `GraphQlClient`, with a negative annotation of a different label, i.e, `GraphQlClientOptions`. The novelty of DiverseTyper is that the declarations, i.e, `class GraphQlClient() {...}` are valid positive annotations, despite the differing pretrained embeddings indicating they

are different entities; we override this model assumption by dictating a new notion of similarity; how a developer interprets these code entities.

By selecting our triplet in this manner, indiscriminate of declaration and annotation, DIVERSETYPER learns an optimal representation of user-defined types and has the capability of using any declaration for type inference irregardless of if the model has seen it before. The unique combination of pre-trained vectors with type clustering is what makes our model perform so well for never before seen types.

More formally the goal is a final representation where the same labeled types and differently labeled types are separated by a margin or distance m. Using the aforementioned notation, *anchor*, *positive* (+), and *negative* (-) to represent the labels of similarity and h_s for the embedding of an i$^{\text{th}}$ type location,

$$\left\|h_{\mathrm{s}_i} - h^+_{\mathrm{s}_i}\right\| + m < \left\|h_{\mathrm{s}_i} - h^-_{\mathrm{s}_i}\right\| \tag{4.3}$$

$\forall\{h_{\mathrm{s}_i}, h^+_{\mathrm{s}_i}, h^-_{\mathrm{s}_i}\} \in \mathbb{T}$ where $\mathbb{T}$ is the set of all possible triplets in the mini-batch, or iteration of training. The notation h_{s_i}, $h^+_{\mathrm{s}_i}$, $h^-_{\mathrm{s}_i}$, are the same anchor, positive, and negative versions (as used above) but referring to the embedding h_s (orange circle in Figure 4.4). The embeddings are considered positives and negatives by their respective type label; the same type is positive and the different type label is a negative. With all three representations, h_{s_i}, $h^+_{\mathrm{s}_i}$, $h^-_{\mathrm{s}_i}$, the triplet loss for a mini-batch with B examples is defined as

$$\mathcal{L}_{\text{TRIPLET}}(h_{\mathrm{s}}, h^+_{\mathrm{s}}, h^-_{\mathrm{s}}) = \sum_{i}^{B} \left[\left\|h_{\mathrm{s}_i} - h^+_{\mathrm{s}_i}\right\| - \left\|h_{\mathrm{s}_i} - h^-_{\mathrm{s}_i}\right\| + m\right]_+$$

This formula simply means, that the model incurs a loss when the distance between the anchor and negative point (different labels) is less than the distance between the anchor and positive point (same labels); a violation of the type space. The margin is added so that the loss occurs even if the negative point is within the extra boundary. $\mathcal{L}_{\text{TRIPLET}}$, (Equation 4.6.2) rewards an embedding $h^+_{s_i}$ that is closer to the anchor h_{s_i} and penalizes $h^-_{s_i}$ that exists within the margin m. Equation 4.6.2 calculates the loss across all possible triplets $\mathbb{T}$ in the mini-batch B. Calculating the loss across all points is less ideal, computationally, as many points easily result in a 0 loss, i.e, they are correctly situated. Realistically, only a few triplets provide a valuable loss; namely ones where the similarity

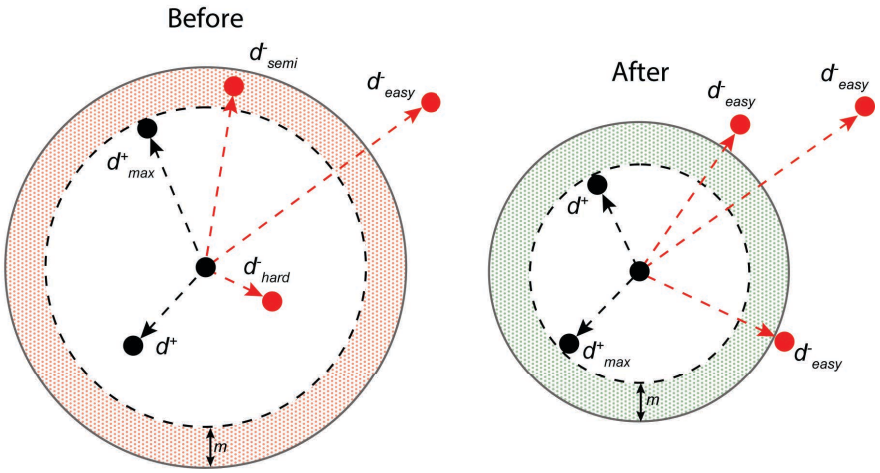


FIGURE 4.6. Illustration of triplet loss with semi-hard negatives. The center is an anchor type surrounded by the same types (black dots) and exist within d^+_{max} where d is the L_2 distance between points. The distance d^+_{max} defines a neighborhood shown with the visualization of a circle. Differently labeled types (red dots) can exist within d^+_{max} (hard negative), $d^+_{max} + m$ (semi-hard negative), and greater than the neighborhood d^- (easy negatives). By optimizing the triplet loss in the left circle, the model adjusts the embeddings so a lower loss occurs. This is accomplished by moving + points (the same types) closer to the center and moving − points (different types) away, further than the margin (dotted boundary). The final result (right circle) is the optimized type space where $\forall$ − points, $d^- > d^+_{max} + m$.

notion is violated and the loss significant. To adjust for the mentioned inefficiency, we use a triplet mining technique called semi-hard negative mining [**192**] that helps find the most valuable triplets and optimize those.

Figure 4.6 demonstrates the selection of semi-hard negatives denoted in the red margin. The green circle in Figure 4.6 is a converged similarity representation where training is complete. Practically, until perfect convergence occurs between all types, there will always occur a decreasing loss.

Subsequent of above, we find that $\mathcal{L}_{\text{Triplet}}$ converges faster than $\mathcal{L}_{\text{Class}}$ due to the relative simplicity of aligning embeddings, but has an increased variance that reduces the effectiveness of $\mathcal{L}_{\text{Class}}$. This trade off means excellent performance of user-defined types with some degradation of the common classifier. We can counter this effect quite significantly by judiciously combining the errors from the loss functions such that each loss is optimized as best as possible. We explore this is the following section.

TABLE 4.2. Building an Arbiter for Metric and Probability Type-spaces

Method	User $\text{Def}_{\text{embedding}}$	$\text{kNN}_{\text{Similarity} \lhd}$	$\text{kNN}_{\text{User Def}}$	Class	$\text{Class}_{\text{Labels}}$	Attention	Data %	Accuracy %
Sort							0%	90.92
Neural Network	✓						10%	82.76
Neural Network	✓	✓					10%	83.43
Neural Network	✓	✓	✓	✓	✓		10%	91.67
Neural Network	✓	✓	✓	✓	✓	✓	10%	91.72
Neural Network		✓	✓	✓	✓		10%	91.96
Neural Network		✓	✓	✓	✓	✓	10%	91.20
Neural Network		✓	✓	✓	✓		50%	**94.14**
Neural Network		✓	✓	✓	✓	✓	50%	**93.54**

The highest performing arbiter has a configuration consisting of a neural network with inputs of user-defined type labels and similarity scores, common-type labels and probability scores, and 50% of the training data.

An Optimal Balance of Losses. As described above, the model learns different losses **Task 1** and **Task 2**: common types for **Task 1**, and user-defined types for **Task 2**. In practice, deep multi-task learning models have claimed improvements in performance by sharing representations, in our case, the pre-trained vector in yellow in Figure 4.4 between both tasks [**108**]. The ideal contribution from each task is not known *a priori* and typically requires searching for a good weighing strategy. In lieu of searching for the perfect strategy with trial and error techniques, like a grid-search, an alternative learnable weighing technique can be used; the model can learn the best weighing scheme as it trains (learning to learn as the model is learning). The learnable weighing technique can be described as learning to estimate the uncertainty of the loss [**108**] from the two typing tasks. A recent empirical survey [**70**] for optimal multi-task weighing strategies demonstrated that uncertainty losses [**108**, **126**] performed best. We follow Kendall et al. [**108**] approach of combining a discrete output (categorical) and continuous output (similarity). The combined loss follows,

$$\mathcal{L} = \frac{1}{\sigma^2_{\text{Class}}}\mathcal{L}_{\text{Class}} + \frac{1}{2\sigma^2_{\text{Triplet}}}\mathcal{L}_{\text{Triplet}} + \log \sigma_{\text{Class}} + \log \sigma_{\text{Triplet}} \tag{4.4}$$

where σ represents the standard deviation. σ^2_{Class} and $\sigma^2_{\text{Triplet}}$ represent the cumulative learned variance (uncertainty) per task. For learning stability, the model learns $\log \sigma^2$ rather than regressing on σ^2. For more details on this, please refer to Kendall et al. [**108**].

We can interpret Equation 4.4 as a combination of the losses with weights for each one. Higher σ values will decrease the impact of the loss signal from the corresponding task, and smaller σ values increase it. Finally, the regularizing terms, $\log \sigma_{\text{Class}}$ and $\log \sigma_{\text{Triplet}}$, penalize the model when the scale of the σ values are too large. The loss asymptotically goes to zero as both sigmas approach

infinity, and while the the model would have a zero loss, it wouldn't learn either task! In summary, the loss from Equation 4.4 can be viewed simply as a *learned* weighted loss with some bias, i.e.,

$$\mathcal{L} = \omega_1 \mathcal{L}_{\text{Class}} + \omega_2 \mathcal{L}_{\text{Triplet}} + b \tag{4.5}$$

A major benefit of a *learned* weighting strategy, like Equation 4.5, is that the final weights are automatically determined by the model over the data; this is clearly preferable to hand-engineering the weights in each problem setting.

The derived (combined) loss, Equation 4.5, allows DiverseTyper to focus on both aspects of type prediction **jointly**: the optimization of common type classifications and the clustering of rare and user-defined types. This work to our knowledge, is the first to apply an effective, general multi-tasking approach to type prediction; this approach may also benefit other SE settings that performance on two tasks must be effectively balanced.

In order to use a trained DiverseTyper, we must define its inference methods.

4.6.3 Inference

This section introduces how DiverseTyper makes either: (1) a common type prediction or (2) a user-defined type prediction. This is done with an *arbiter*; which (like a human arbiter) settles "disputes" between the common-type and user-defined type predictions, as we now explain.

Common Type or User-Defined Type? During inference, DiverseTyper outputs a common-type guess (purple arrow in Figure 4.4) and a user-defined type embedding (orange arrow in Figure 4.4). The common-type guessing mechanism is the classification layer that outputs common-types from the fixed set of types with probabilities per **Task 1**. The user-defined type guessing mechanism is the closest neighbor lookup using the user-defined type embedding first defined in **Task 2**.

The neighborhood lookup is an efficient k-nearest-neighbor (kNN) search algorithm[3] across all training set declarations and uses of those declarations. DiverseTyper adds the testing set declarations only because the declarations are available at the time the corresponding type can be predicted; the model has never seen these test declarations before. If the model is successful at never before seen types, these new declarations will be embedded near relevant usages. Finally,

[3]https://github.com/spotify/annoy

the search returns the distance which $\in (0, 1]$ with 0 being an exact match. When calculating the similarity, we can take $1 -$ distance.

We note that the similarity measure is not a probability measure, looks nothing like Figure 4.5, and thus the two are not comparable unless some mapping is applied. This presents a quandary: given two incommensurate measures, how would an arbiter resolve a "dispute" when different type labels are offered by the two? In the following section, we discuss how a special mapping can be baked into a neural network, automatically picking the best type.

TABLE 4.3. Multi-Task Type Annotation Datasets

Approach	**Type Inference Dataset**			**User Definition Dataset**		
	Projects	Files	Annotations	Projects	Files	Annotations (Definitions/Usage)
TypeBERT [**95**]	20,860	1,474,418	12,920,988	✗	✗	✗
DiverseTyper	20,860	1,474,418	12,920,988	14,309	225,551	3,204,180 (50% / 50%)

DIVERSETYPER uses two joint learning objectives to learn common-type and user-defined type associations. We provide no additional type inference data to demonstrate the effectiveness of the supplemental objective. The true ratio of definitions to usage is 32% / 68% but we over-sample definitions such that each batch has both a definition and its corresponding use.

Arbiter. The arbiter first obtains a list of common-types and their probabilities from the common-type guesser. Next, a k nearest neighbors search of user-defined types is performed, returning a second list of user-defined types and the respective similarities. The arbiter combines unique types from each, sorts them, and returns L_{mixed}, a set of mixed types. However, sometimes both type prediction mechanisms present similar scores, so, how to choose the very best type?

As an initial step, we compare our approach with a baseline "sorting" approach despite the two different metrics: probability and similarity. This approach consists of combining both sets and sorting irrespective of type metric. To our surprise, this simple baseline performed better than expected, with an accuracy of 90.92%. In a later analysis of the type spaces in Section 4.8.2, it can be inferred that (extremely) low distance points often yield the correct prediction, comfortably overriding an incorrectly predicted common-type's probability. Likewise, the probabilistic predictions are highly confident when correct in the BERT-family models and thus can easily override the distance of an incorrect user-defined type. The baseline is effective in most cases but there are some

scenarios where the correct answer is enigmatic. This is where we find performance of a specifically trained neural network arbiter beats the simple sorting baseline.

We tried several designs for the neural network in Table 4.2. We manipulate the network's access to various inputs: the similarity embedding, similarity distance, user-defined type label, common-type probability, common-type label, attention, and amount of data trained on. From the above ablations, we find the best performing arbiter uses the top 5 common-type prediction probabilities and user-defined type similarity scores with the respective type labels. We create a dataset with the described inputs in Table 4.2. The output label is 0 if the common-type mechanism gets the answer correct and 1 if the user-defined type mechanism gets the solution correct. The arbiter's neural network is trained on a holdout portion of the training data while the remainder is used to learn the kNN search. The trained model is selected by performance on a validation set and evaluated on the test set per Table 4.2. The trained model was very good as a binary classifier picking between the two type predictors (common vs. user-defined type). As seen in Figure 4.7, this classifier has a strong receiver operating characteristic (ROC) curve with an area under the curve (AUC) of .93. This ROC and AUC demonstrates that the classifier is effective at arbitrating the two type recommendation mechanisms. The next section examines the performance of DiverseTyper with several research questions (RQs).

4.7 Quantitative Evaluation

In this section we present the dataset DiverseTyper is trained and evaluated on, metrics for type inference evaluation, and our baselines for DiverseTyper.

Then we answer the following research questions:

RQ1: How effective is DiverseTyper compared to baseline approaches?

RQ2: Can DiverseTyper predict user-defined types?

RQ3: How does DiverseTyper perform on previously unseen types?

4.7.1 Dataset

We use the same 20,860 projects collected in TypeBERT [**95**] for the type inference dataset and maintain the same data splits between train, test, and validation. This dataset contains

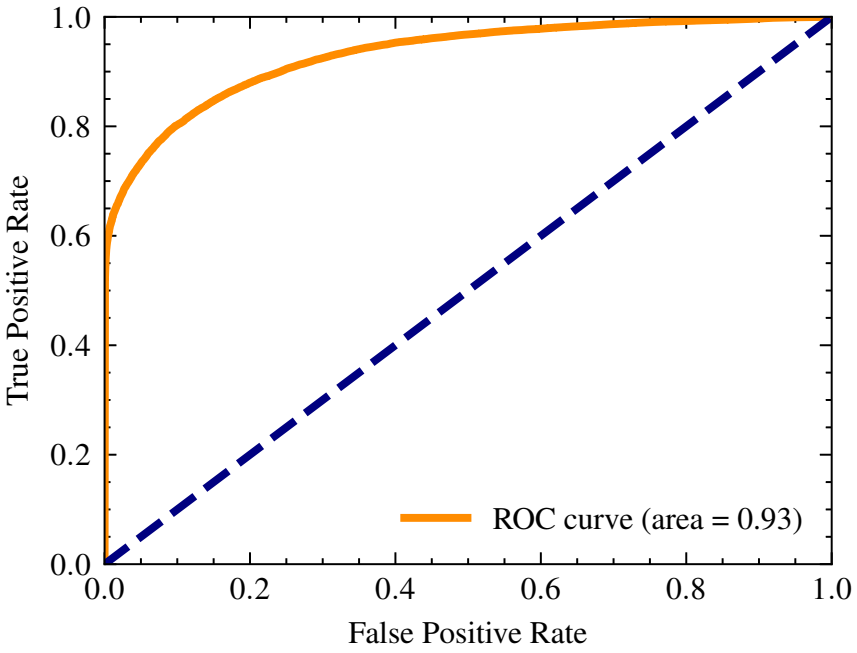


FIGURE 4.7. Receiver Operating Characteristic (ROC) curve of the Arbiter. Area Under the Curve (AUC) is .93 which means it is an excellent classifier.

human-annotated types on variables, parameters, functions and method declarations. Types range from common-types, e.g., `number` and `string` to library and user-defined types like `dynamodb` and `Point`. This dataset does not have the user-defined type declarations but only the annotations. We supplement this dataset with additional data extracted from the same set of projects that includes user-defined type declarations across the existing dataset splits in TypeBERT [**95**]. The purpose of a project level data split is twofold: (1) so files seen at test time are never seen at training time and (2) to accurately compare the contribution of training on user-definitions.

To extract user-defined type declarations and the locations of their use, we wrote a code parser to localize user declarations denoted with keywords `interface` and `class`. The parser finds use of a user-defined type corresponding to the declaration within project scope. The supplemental user-defined types dataset contains 362,759 (32%) declarations and 1,141,734 (68%) uses across 14,309 projects and 225,551 files. This supplemental dataset is only used in training and is not used to evaluate the model's performance. We have released the supplemental dataset in our GitHub repository.

To evaluate DiverseTyper, we follow standard evaluation procedure from previous works [**95**, **157**, **230**] and only use human-annotated types for evaluation. The intuition is that compiler inferred types are typically easy and saturate performance scores, where as, human annotations are more difficult and meaningful. Following standard practice we allow the model to train with both the "easy" compiler inferred types and the "hard" human-annotations. Also in line with other works, we exclude the wildcard type `any` in our evaluation. Finally, a key practice is to perform de-duplication on a dataset [**7**]. Code duplication between and within training and evaluation sets has historically existed in previous works, prior to Allamanis [**7**] demonstrating duplication leads to artificially elevated evaluation scores. To conclude this section, we provide the breakdown of

Table 4.4. Top Types In Datasets

Type-Inference Dataset			**User Definition Dataset**		
Types	Count	Data %	**User Defs**	Count	Data %
string	2,103,227	16.19	Node	7,583	.0584
void	1,324,632	10.20	State	6,843	.0527
number	1,213,432	9.34	Props	6,536	.0503
array	915,837	7.05	User	5,798	.0446
object	635,155	4.89	Context	5,367	.0413
Promise	549,219	4.23	Type	3,961	.0305
boolean	514,801	3.96	Player	3,386	.0261
Sum	**7,256,303**	**55.87**		**39,474**	**.3039**

top types in each dataset. Observe the type breakdown in Table 4.4. User-defined types occur infrequently while common-types account for >55% of the original type annotations. We believe the stark distributional difference between common-types and user-defined types necessitates the separate mechanism for predicting user-defined types. Again, this is because the vast majority of types are often project-specific, locally defined, and only occur only a handful of times. It is important to note that DiverseTyper contextualizes user-definitions, so different declarations with the same name, such as User or State, will be represented by a separate data point. This is particularly useful for the model because it can condition on small differences in the definition such as the presence of attributes.

TABLE 4.5. Accuracy Comparisons with DIVERSETYPER Across Binned Types

Model	Top 1 Acc %				Top 5 Acc %			
	Top 100	Other	User Def	Overall	Top 100	Other	User Def	Overall
LambdaNet [**230**]	66.9	N/R	53.4	64.2	86.2	N/R	77.7	84.5
OPTTyper [**157**]	76	N/R	N/R	N/R	N/R	N/R	N/R	N/R
TypeBERT [**95**]	**89.51**	**50.49**	41.40	71.12	**98.51**	**70.34**	55.02	81.88
DIVERSETYPER$_{\text{ARBITER}}$	83.51	46.92	**72.53**	**79.30**	92.94	60.53	**81.59**	**88.59**
DIVERSETYPER$_{\text{ARBITER}_{\text{NN}}}$	84.78	43.29	**71.56**	**79.71**	90.88	53.25	**77.18**	**85.62**

TABLE 4.6. DIVERSETYPER Ablations

Model	Top 1 Acc %				Top 5 Acc %			
	Top 100	Other	User Def	Overall	Top 100	Other	User Def	Overall
DT_{base}	82.28	24.69	23.41	59.77	95.76	41.48	36.49	73.10
$\text{DT}_{\text{base + UNK filler}}$	82.46	33.56	46.40	68.67	96.59	59.44	78.69	79.43
$\text{DT}_{\text{TypeBERT + U.D. network}}$	**89.69**	**50.49**	41.39	71.16	**98.57**	**70.34**	55.0	81.86
$\text{DT}_{\text{base}_{\text{e2e}} + \text{U.D. network}_{\text{e2e}} + \text{Arbiter}}$	**83.51**	46.92	**72.53**	**79.30**	92.94	60.53	**81.59**	**88.59**
$\text{DT}_{\text{base}_{\text{e2e}} + \text{U.D. network}_{\text{e2e}} + \text{Arbiter}_{\text{NN}}}$	**84.78**	43.29	**71.56**	**79.71**	90.88	53.25	77.18	**85.62**

base: DIVERSETYPER with only common-type classifier (same as TypeBERT) used for evaluation
base + `UNK` filler: fill `UNK` predictions with top 1 guess from user-defined type mechanism.
TypeBERT + User-Defined network: initialize DIVERSETYPER with TypeBERT's weights. These weights are not changed.
base$_{\text{e2e}}$ + User-Defined network$_{\text{e2e}}$ + Arbiter: Use basic (sort) arbiter to pick top 1 type between common-type and user-defined type mechanism.
base$_{\text{e2e}}$ + User-Defined network$_{\text{e2e}}$ + Arbiter$_{\text{NN}}$: Use neural network arbiter to pick top 1 type between common-type and user-defined type mechanism.
* e2e: learned jointly with end-to-end training
* NN: Neural network

4.7.2 Metrics

We use Top-1 Accuracy (exact match) and Top-5 Accuracy (correct prediction in the top 5 guesses) for subsets of types exactly in line with previous works. The categories are as follows:

Top 100: The most frequent types such as native types `int` and `string` and types not considered user-defined within the top 100 rank [**95**, **157**, **230**].

Other: Types that are common but occur outside of the top 100 and are not user-defined. Examples would be commonly used library types like `ArrayBuffer`, `Entity`, `FunctionComponent` just to name a few.

User Defined: Types that correspond to a class, enum, or type interface where the type is declared within the same project scope. Examples would be developer specified types that occur quite rarely if at all in other projects, e.g., `KindaShiftView`, `VRMSpringBone`, `Iterm2ColorName`.

Unknown: In previous works with fixed type vocabularies [**78**, **95**, **142**, **157**, **172**] type inference models would predict `UNK` if the type exceeded its classification capabilities. TypeBERT [**95**] accounted for `UNK` predictions by counting them against the performance of TypeBERT. This meant that ∼8% of predictions in the test set were automatically considered incorrect as a function of its model architecture. DiverseTyper has no type limitations, and *never predicts* `UNK` as it can always defer to the user-defined similarity vector.

4.7.3 Baselines

We compare DiverseTyper to three TypeScript baselines: LambdaNet [**230**], OptTyper [**157**], and TypeBERT [**95**].

LambdaNet a graph neural network (GNN) approach that links variables and logical constraints to approximate a type dependency graph. The architecture can predict common-types in the top 100 and user-defined types available in the type-space with a pointer network.

OptTyper performs probabilistic type inference across the top 100 most-frequent types. OptTyper combines a continuous interpretation of logical constraints derived by static type inference with the natural constraints learned from deep learning across large code bases.

TypeBERT uses BERT-style pre-training with large scale corpora in addition to a large fine-tuning dataset to train a type inference model. The implementation is similar to sequence tagging in NLP.

4.7.4 RQ1: Effectiveness of DiverseTyper

We evaluate the effectiveness of DiverseTyper over the type categories in Section 4.7.2. We also perform ablation analysis by varying different elements of its architecture to understand why DiverseTyper is effective.

Type Performance

As shown in Table 5.3, we report the top-1 accuracy and top-5 accuracy across type categories

defined in metrics. DiverseTyper has the strongest Top 1 overall scores at 79.71% accuracy overall a +8.59% absolute improvement over TypeBERT. DiverseTyper scores the highest Top 5 accuracy overall meaning that DiverseTyper is providing more relevant scores across all of its predictions. Both DiverseTyper models demonstrate Top-5 scores higher than TypeBERT and LambdaNet which is notable because LambdaNet uses static analysis and pointer mechanisms to predict user-defined types. This means that not only does DiverseTyper do a better job at recognizing user-defined types, but is additionally capable of referencing declarations from its kNN search even when the declarations are unavailable or missing in the source code.

We observe a trade-off in the top 40,000 types (Top 100 and Others). This might be due to complications arising from training of **Task 1** and **Task 2** together. A possible source of this, is that often developers override library types such as `Node`, `State`, `User`, `Context` etc. where the common-type classifier gets disrupted in favor of learning user-defined type declarations. The aforementioned types are also the most frequent user-defined types as seen back in Table 4.4.

Ablations

Deep learning models are difficult to understand and ablations provide insights to the model's learned representations. We vary the architecture in meaningful ways to gain insights to how different methods affect prediction capability. We organize our ablation results in Table 4.6.

First we define a base version of DiverseTyper, DT_{base}, where the model is trained jointly on the two tasks: **Task 1** and **Task 2**. This model is evaluated with **only** the common-type classifier. The purpose of this evaluation is to demonstrate how much accuracy was lost in the traditional classifier from the jointly learned objectives. We observe that others and user-defined types perform very poorly. This indicates that the type representation r_τ (defined in **Task 1** is not a meaningful type representation of user-defined types anymore. This intuition is confirmed with the performance of the same model using the user-defined kNN search. The model that employs the kNN search is denoted $\textbf{base}_{\textbf{e2e}}$ **+** $\textbf{U.D network}_{\textbf{e2e}}$ **+ Arbiter**. From a training perspective, the network is equivalent to DT_{base} and yet performs +50%. This indicates that the user-defined type representations are *moving* in favor of the similarity representation. This comparison between the two models with the same training but different inference mechanisms shows the effectiveness of our proposed learning approach.

In order of incremental improvement, we try a model where we use **base** and only sample from the user-defined types when the classifier predictions `UNK` viz. the model does not know the type. **base + `UNK` filler** improves performance but not considerably.

The next ablation is to initialized DiverseTyper with TypeBERT weights and train with only the user-defined supplemental dataset while holding the TypeBERT weights stationary. This model has the label **TypeBERT + U.D network**. We can see that the user-defined types were not learned by the model and this model has almost the same performance as TypeBERT. We anticipate that this akin performance is because the final type representations learned in TypeBERT drop relevant features about user-defined types in the process of partitioning the common-type space. Interestingly, the performance of this model marginally surpasses TypeBERT indicating the network learned relevant information pertaining to the Top 100 in the user-defined type dataset.

The final two ablations are our best models where we use the base model, plus the kNN search, and the arbiter for inference. The first model uses only the sorting arbiter and is labeled $\textbf{base}_{\textbf{e2e}}$ + $\textbf{U.D network}_{\textbf{e2e}}$ + **Arbiter**. In this ablation, DiverseTyper is capable of using the information learned during training to match declarations with annotations. This model performs best in user-defined type exact-match and top-5 overall. The second of the two best DiverseTyper models uses the same base model, kNN search, but with a neural network arbiter. This model is denoted $\textbf{base}_{\textbf{e2e}}$ + $\textbf{U.D network}_{\textbf{e2e}}$ + $\textbf{Arbiter}_{\textbf{NN}}$ and performs the best overall in exact matches. These models reinforce the hypothesis that the model is capable of taking advantage of user-defined type matching with declarations with a sizeable performance increase in the overall and user-defined types category.

DiverseTyper improves overall performance **8.59%** over TypeBERT, a **13.38%** percent increase of TypeBERT's improvement over LambdaNet.

4.7.5 RQ2: Prediction on User-Defined Types

In this section we evaluate DiverseTyper's capabilities on user-defined types. Observe the reported user-defined type accuracy in Table 5.3 and Table 4.6. In comparison with other approaches DiverseTyper performs 31.13% better than TypeBERT across user-defined types with almost the

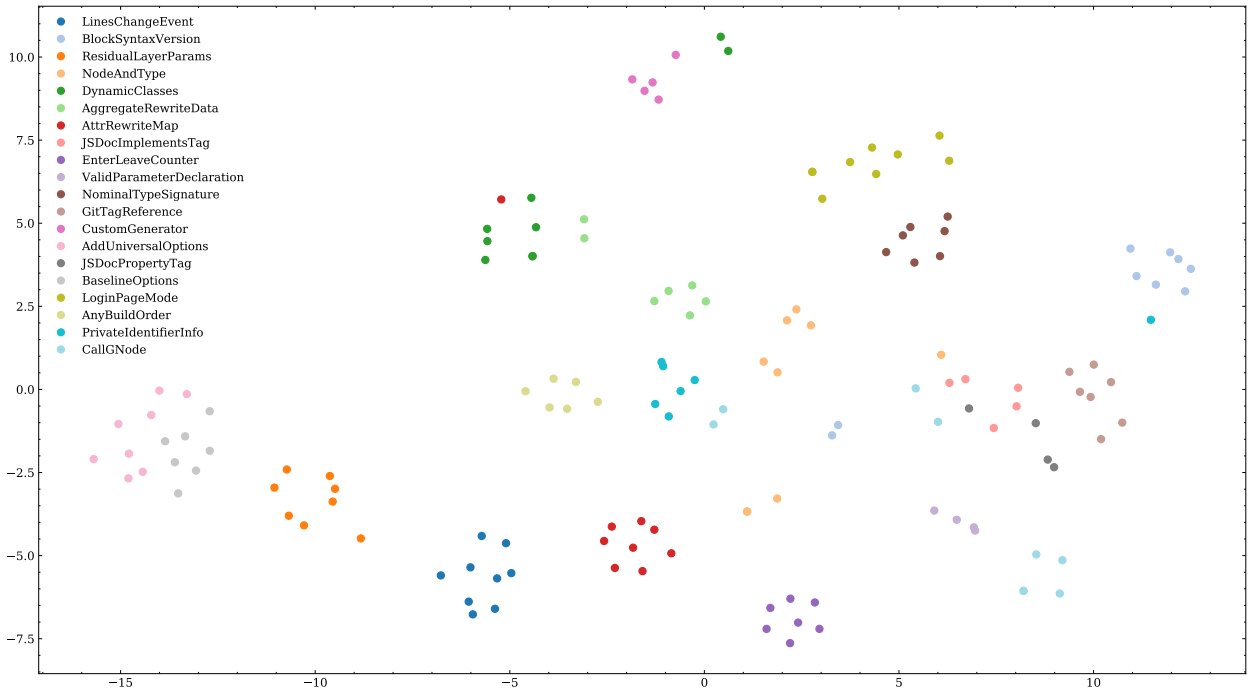


FIGURE 4.8. t-SNE plot of aligned user-defined types and the respective usage. When a developer defines a new type and requires type inference of this new type, the usage embedding will be clustered with the definition making type inference of new user-defined types possible with high accuracy.

same architecture. The performance of DIVERSETYPER informs us that the user-defined similarity embedding is substantially more effective for rare and user-defined types than a fixed vocabulary. The performance improvement can be attributed two characteristics of the user-defined similarity embeddings. First, **Task 2** introduces a representational "slack" by clustering similar representations rather than strict *partitioning* of dimensional space. Second, unlike other deep similarity learning approaches [**11**, **149**] that only group similar annotations, DIVERSETYPER places the `class` and `interface` declarations into the user-defined type-space. By learning the relationship between declaration and annotation, DIVERSETYPER makes the user-defined type set generalizable to novel `class` and `interface` declarations.

DIVERSETYPER improves user-defined type accuracy on LambdaNet by **19.13**% and **31.13**% over TypeBERT.

Table 4.7. DiverseTyper on Never Seen Types

Type Accuracy							
Top 1 Acc				Top 5 Acc			
Top 100	Other	User Def	Overall	Top 100	Other	User Def	Overall
0%	19.44%	80.56%	100%	0%	19.44%	80.56%	100%
N/A	1.723	54.82	44.50	N/A	3.62	58.48	47.82

9.04% of test set contains types never seen before. Top 100 accounts for 0% of never seen types. Percents under type categories are proportion of never seen types. For example, 80.56% of never seen types are user definitions.

4.7.6 RQ3: Performance on Never Seen Types

We test DiverseTyper with the hardest type annotations, i.e., types that have never been seen before. By evaluating the model on never seen types, we gauge how well DiverseTyper will do on brand-new types added by developers. Table 4.7 shows that DiverseTyper scores a commendable 44.50% top-1 accuracy on types it has never seen. DiverseTyper performs even better for user-defined types, at 54.82% top-1 accuracy. This result is more consequential when we see that user-defined types occupy about ∼81% of never seen types. This is a promising result for a deep learning based approach where performance is typically dependent on comprehensive examples from training data.

DiverseTyper's user-defined type mechanism successfully annotates never seen types ∼**55%** of the time.

4.8 Qualitative Evaluation

Our qualitative evaluation serves to elucidate the inner workings of DiverseTyper and its user-defined type similarity predictions. In this section we answer the following research question:

RQ4: Can we visually inspect DiverseTyper's performance over other methods?

RQ5: How does DiverseTyper cluster typical user-defined types?

4.8.1 RQ4: Inspecting Consequential Annotations

Figure 4.2 is a snippet from a popular Microsoft GitHub repository. We can see that the type `CodeSandboxLanguage` is defined with the `type` keyword indicating it is a user-defined type. It is defined within a TSX file (TypeScript's equivalence to JSX) and imported into the main `.ts` file. The user-defined type `CodeSandboxLanguage` is used in the definition of an object `createPackageJson`. DiverseTyper recognizes the intra-project type declaration and properly assigns it in the object. DiverseTyper recommends both lexical similar types such as `Language` and hints at functionality preservation with relevant type `RemoteDebugLanguage` for a sandbox environment. TypeBERT only recommends lexical similar types such as `Language`, `ILanguage`, and `LanguageCode`. LambdaNet considers the correct type `CodeSandboxLanguage` to be Out-Of-Vocabulary (OOV) because it did not populate in it's list of possible types.

The outcomes from the three models are not surprising given their advantages and disadvantages. LambdaNet is restricted to 100 types and the types it discovers within the project. LambdaNet has failed to discover `CodeSandboxLanguage` and thus declared it outside of its predictive capability. TypeBERT has seen contexts, especially in pretraining, where semantically similar notions of `CodeSandboxLanguage` occur with types `Language` and `LanguageCode`; thus the recommendation. Even if TypeBERT had an unbounded classification layer, which is not currently possible in machine learning, it is still less likely that TypeBERT would get a majority of user-defined types correct. This is because TypeBERT must accurately predict the exact type out of an unbounded list of types where as DiverseTyper only has to match the correct declaration to the context and use it correctly.

Figure 4.3 is a code snippet from another popular coding repository containing LeetCode webapp code. The user-defined type `LeetCodeSolutionProvider` is defined within the same file as the class usage. LambdaNet does not have `LeetCodeSolutionProvider` in its top 5 predictions, but shows some relevant predictions. We observe in other files within the same project, LambdaNet had relevant user-defined type predictions, but fails to place them as the top guess for reasons we do not know; likely a violation of type constraints according to LambdaNet since the type is in the same file. TypeBERT fails at most user-defined types as they exceed the vocabulary limit and are not defined in its architecture; this is expected because TypeBERT cannot predict infrequent and rare

types. DiverseTyper accurately predicts the right type where as TypeBERT and LambdaNet do not. We observe this similar outcome for other user-defined types in other files within the same project: `LeetCodePreviewProvider`, `LeetCodeExecutor`, `LeetCodeStatusBarController`, `LeetCodeTreeDataProvider`. Most impressively for DiverseTyper, these types are all defined and used **once**, demonstrating DiverseTyper's capability on rare and infrequent types.

DiverseTyper correctly associates *rare* and *infrequent* user-defined types within the same file and across the same project.

4.8.2 RQ5: Typical User-Defined Type Clusters

The neural type embeddings learned by DiverserTyper are information rich due to the extremely large corpus used for fine-tuning. The visualization of such type embeddings demonstrate important type relationships learned during this fine-tuning. Embedding visualizations reach many data exploratory domains [**61**, **111**, **150**]. Commonly, the embedding visualizations are crafted by transforming the high dimensional data into two dimensions while preserving the overall structure of the data. The t-SNE algorithm aims to perform dimensionality reduction to lower dimensions that humans can interpret (2D or 3D), while preserving the structure of the high-dimensional data, as the model interprets it. Figure 4.8 and Figure 4.9 are created with the t-SNE algorithm [**220**]. A visualization of specific embeddings, such as a category of types, can indicate performance across these embeddings. If the type clusters of the embeddings are indistinguishable, then the kNN search will likely not return the correct answer as a kNN search is a function of neighboring data points. An embedding space that is not clustered properly across types is undesirable as the model will fail to generalize properly.

Originally, the pre-trained embeddings are purely context driven, meaning that similarly occurring contexts amongst variables and function names will appear co-located in embeddings. DiverseTyper has altered this embedding space to shift context dissimilar sequences, such as type declarations, in a manner that is useful to typing. While this transformation is the goal, the model must to maintain relative structure of the embeddings in places that are not directly relevant to typing. Examples of this include general code syntax and the semantics the model derives from a

sequence. DiverseTyper must balance maintaining existing code semantics while aligning type declarations derived from those code semantics; especially to generalize on code snippets. We now direct the reader to Figure 4.8.

Figure 4.8 is a visualization of the most difficult types to classify, i.e, never seen user-defined types. To visualize the aforementioned clustering of infrequent user-defined types, we plot all user defined types in a t-distributed stochastic neighbor embedding (t-SNE) and select points based on whether the model has seen the type before. Listed in Figure 4.8, are 20 user-defined types that DiverseTyper has never seen in training and occur extremely infrequently. We can see that each type, represented by various colors, is grouped into small clusters of like-typed annotations. This shows that our approach for aligning user-defined types works correctly. In the same figure, we exam the relatedness of never before seen types with the purpose of maintaining semantic meaning. In Figure 4.8, left, `BaselineOptions` and `AddUniversalOptions` stand out as co-located and complementary in embedding structure. Upon further inspection of why this might be, it is clear that the types are related by instantiation, specifically, a subclass from the base class. The learned complex relationships, such as instantiation, are encouraging for future work because complex type relationships exist. Some of these complex type relationships include but not limited to union and inheritance types (outside the scope of this work), and are potentially tractable for this model architecture. On the right side of Figure 4.8, it can be observed that `JSDocImplementsTag` is close to `JSDocPropertyTag`. Again, with further inspection to how these tags are used in real projects, we find that both return `SymbolDisplayPart` related types. We conclude from the visualization that the model exhibits a basic understanding of how types are used *and* the complexities within the defined type behavior even when the types are incredibly sparse.

Naturally, we are curious about DiverseTyper's capability of clustering common-types with the user-defined type mechanism despite not being trained to do so. Again, we visualize the clustering of common-types with a t-SNE plot in Figure 4.9. From Figure 4.9, it can be concluded that common-types are clustered in a similar fashion to user-defined types, but with less defined margins or white space between groups. With **Task 2** only trained on user-defined types and common-types learned by a common-type classifier in **Task 1**, we expect DiverseTyper to completely deprioritize the learning of common-type similarity, yet surprisingly maintains proper structure. One might ask,

"Why does common type clustering matter when one can use the common type classifier?". The usefulness of common-type clustering is in the case of an arbiter misprediction. If the arbiter picks the user-defined type mechanism over the common-type classifier, the clustering of common-types should provide some redundancy. For example, if a common-type matches a real type annotation from the user-defined search (potentially an infrequent case of overriding a native type), the type prediction will still be correct.

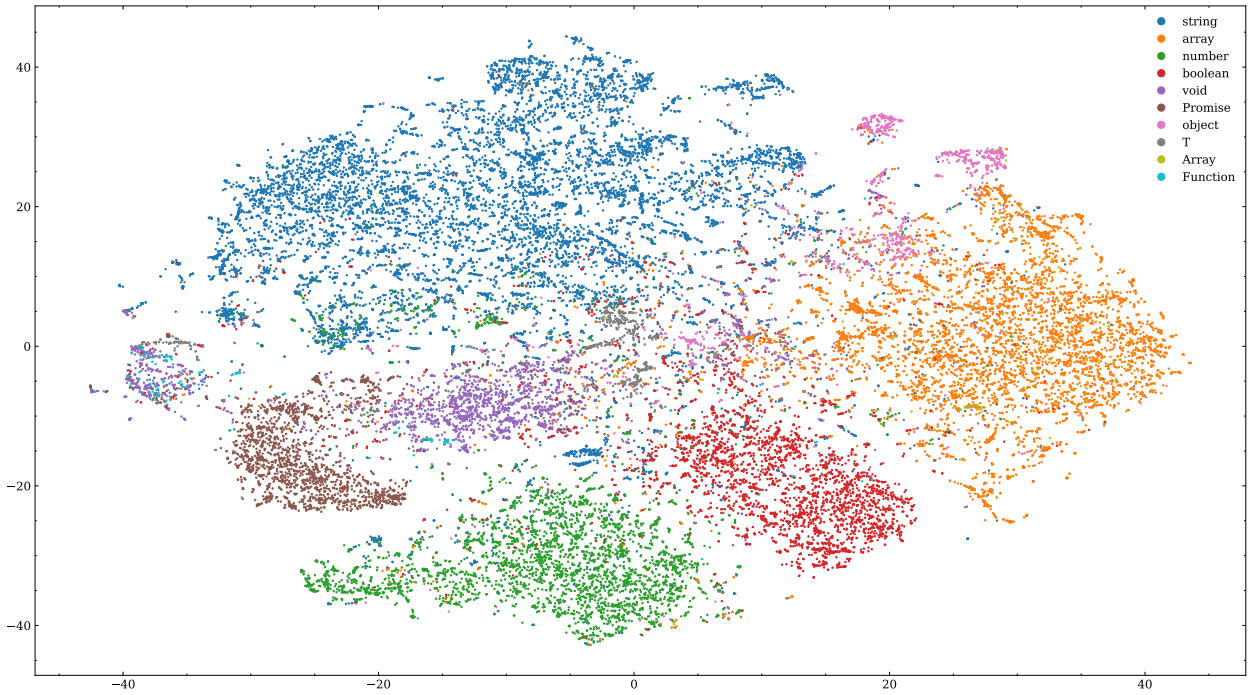


FIGURE 4.9. t-SNE plot of commonly used types. DIVERSETYPER's inherits a the strong performance of common-types across its classification layer. Additionally DIVERSETYPER demonstrates effective clustering in commonly occurring types. If a developer overrides a common-type, e.g., `string`, DIVERSETYPER has both a common type guess and user-defined type guess that the arbiter can choose from.

DIVERSETYPER groups rare user-defined types in a similar fashion to developers with regard to semantic and syntactic relatedness.

4.9 Related Work

Pre-Trained Foundational Models

Large scale pre-trained models [**54**, **134**, **168**, **175**, **242**] coined *foundational* models [**28**], are stacked transformers [**222**] with various autoencoding objective functions [**47**, **177**] on large unlabeled data. Their success in natural language processing (NLP) has warranted its application in other fields such as computer vision (CV) [**123**, **207**] and software engineering [**62**, **72**, **102**]. The extent of foundational "learning" is hotly debated [**23**] and examined in a software engineering context [**107**]. Albeit, the performance improvements from pre-training is undeniable as it has set benchmarks for many SOTA tasks across various domains.

Multi-Task Learning

Multi-task learning attempts to efficiently learn multiple objectives from a shared representation [**35**]. Multi-task learning is prevalent in machine learning fields of natural language processing [**48**], computer vision [**115**], and speech recognition [**89**], but is seldom used in software engineering [**129**, **196**]. Prior approaches use a naïve weighted sum of losses where the losses are uniformed or manually weighed. New approaches include dynamically weighing tasks from gradients [**132**] and uncertainty [**108**, **126**]. The dynamics of multi-task learning is still not very well understood but has been effective across several applications.

Type Inference

Dynamic type inference techniques [**13**, **184**] and type checkers [**1**, **2**, **24**, **179**] achieve soundness by enforcing type constraints. Dynamic type-checking provides the convenience of not requiring annotations, and/or having to fix compile-time errors; however, dynamic checking may miss coding errors un-executed parts of programs.

Machine learning can help programmers more conveniently make better use of static type-checking by suggesting type annotations. This works by learning natural type distributions across corpora of code [**180**]. Hellendoorn et al. [**78**] interpreted type annotation as a tagging task with DeepTyper. Pradel et al. [**172**] designed separate sequence models to infer function types in Python

and validate with a type checker. Wei et al. [**230**] used insights from [**10**] to train a GNN from type dependency graphs. Allamanis et al. [**11**] proposed a graph based approach to predict types with similarity learning and parametric type matching. Mir et al. [**149**] uses an approach akin to Allamanis with more data and improved results. Jesse et al. [**95**] uses pre-training to improve sequence tagging of types. This work extends the previous works by introducing a novel training approach and a data set for learning user-defined and rare types. Unlike DiverseTyper, existing approaches do not *contextualize* and provide new *associations* for novel developer defined types.

4.10 Conclusion

DiverseTyper presents a test of our hypothesis that user-defined type declarations and the corresponding type annotations can be aligned and used in type predictions. We demonstrated that deep learning models could learn to encode novel class and interface declarations, leverage the learned representations to guess rare and difficult user-defined types, and extend to never before seen types. Finally, we believe that our approach can be applied to other applications of machine-learning to software engineering, where developers can freely proliferate concepts, (*e.g.*, functions, interfaces, classes, generics, exceptions) and thus arbitrarily transcend any vocabulary limits pre-set by machine-learning models.

Acknowledgment

We would like to thank Vincent Hellendoorn for his thoughtful comments and helpful reviews. Kevin Jesse is supported from NSF CISE (SHF) Grant No. 1414172.

Chapter 5

ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference

5.1 Preface

Benchmarking type-inference models is complicated: most papers are evaluated on their own collections of source code from public repositories. A significant challenge in Chapter 3 and Chapter 4 was the lack availability of model weights with the corresponding model code; some models required specific virtual environments no longer available. Platforms like Huggingface make significant efforts to publish and standardize how models are ran on common hardware like Nvidia GPUs. DiverseTyper is a natural extension of TypeBERT and the evaluation dataset for both was the same, however, this is not often the case. To address this roadblock and lower the barrier of entry to published type inference models, we have published a dataset, with mining scripts, **and** popular models trained for type inference. These models are very easy to integrate into future tools and frameworks; one of which we discuss in Chapter 6. The dataset and evaluation is on Microsoft's code intelligence leaderboard[1] and GitHub[2]. This chapter is based on a MSR 2022 conference paper titled

[1] `https://microsoft.github.io/CodeXGLUE/`
[2] `https://github.com/microsoft/CodeXGLUE`

ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference completed with myself as lead author and Premkumar Devanbu in an advisory capacity.

5.2 Summary

In this chapter, we present ManyTypes4TypeScript, a very large corpus for training and evaluating machine-learning models for sequence-based type inference in TypeScript. The dataset includes over 9 million type annotations, across 13,953 projects and 539,571 files. The dataset is approximately 10x larger than analogous type inference datasets for Python, and is the largest available for TypeScript. We also provide API access to the dataset, which can be integrated into any tokenizer and used with any state-of-the-art sequence-based model. Finally, we provide analysis and performance results for state-of-the-art code-specific models, for baselining. ManyTypes4TypeScript is available on Huggingface, Zenodo, and CodeXGLUE.

5.3 Introduction

There is considerable interest recently in the application of machine learning (ML) models to a variety of software-related tasks and datasets. ML has largely focused on improving performance, using probabilistic models of source code that exploit code's regularity and patterns [**8**]. The type-inference problem is one such task where probabilistic code models work well. Probabilistic type guessers can infer types for developers, helping them avoid type errors, and lowering the annotation effort [**65**]. TypeScript and Python have been the primary languages targeted by researchers [**182**, **240**]. Recent ML-based methods [**11**, **78**, **95**, **149**, **157**, **164**, **172**, **230**] appear to work well, but are hard to compare, due to variability in evaluation practices.

The field of type inference varies quite a bit, in methods, data, and metrics. With the abundance of open source repositories, new methods often mine their own data or attempt to sample similar data from previous work [**11**, **95**, **172**, **230**]. Despite these works often using similar metrics, performance is confounded with scoring differences and sampling bias. Scoring differences arise when various subsets of types are evaluated and not others, for example, based on frequency (top-100), location (parameter, and function level), and annotation type (user-defined). Sampling bias occurs from type inference papers sampling different projects or files at various commits where code context

and the annotations themselves can differ. Though there have been some attempts at standardized comparisons for instance DeepTyper [**78**] and NL2Type [**142**], Typilus [**11**] and Type4Py [**149**], other recent publications showed quite a bit of variance in evaluation, *e.g.* some used Top 100 types [**157**]; some compare across different projects; others use the same projects, but at different time slices. We feel there is still a need for a comprehensive TypeScript dataset and metrics.

To help standardize training and evaluation for TypeScript type inference, we offer the ManyTypes4TypeScript dataset. This comprehensive dataset includes over 9 million type annotations, which is 10x more annotations than the next largest Python annotated dataset ManyTypes4Py [**148**]. The ManyTypes4TypeScript also comes with evaluation scripts, enabling models to be properly benchmarked against the test set. We make all of our collection scripts, unprocessed data (Zenodo[3]), processed API dataset (Huggingface[4]), usage examples, and evaluation script publicly accessible. The dataset was collected in mid January of 2022 for publicly available GitHub projects. Our contributions are as follows:

- A dataset containing a comprehensive set of code snippets and aligned type annotations across 13,953 TypeScript projects resulting in 9M type annotations.
- Standardized access across a range of state-of-the-art models on 🤗Huggingface.
- Standardized scoring with metrics and existing evaluation of state-of-the-art models.
- Additional word tokenized data for flexible model input, allowing choice of sub-tokenization methodologies. We include the mining scripts so the SE community can update the dataset as needed.

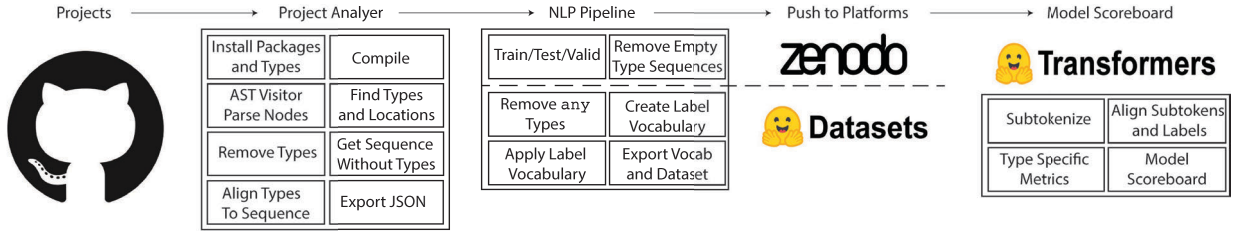
All of our code is publicly available. In the next section we discuss the collection process and parsing of projects.

5.4 Collection Process and Parsing

Figure 5.1 illustrates the collection process and parsing from project to machine learning dataset. First we use GraphQL[5] to gather a list of ∼29,500 public TypeScript projects on GitHub. The GraphQL query returns TypeScript projects by the number of GitHub stars to ensure the collection

[3]https://zenodo.org/record/6387001
[4]https://huggingface.co/datasets/kevinjesse/ManyTypes4TypeScript
[5]https://graphql.org

FIGURE 5.1. The collection and parsing process of ManyTypes4TypeScript



of quality projects. After mining the list of projects, a custom bash script attempts to install packages, types, and other requirements with Pnpm[6]. This is important for compiler inferred types as inferred types largely come from resolved package dependencies. Each file's AST (abstract syntax tree) is traversed, extracting both human annotations as well as compiler-inferred annotations. The traversal, gathers the tokens and labels types on the AST nodes. The types are removed and the tokens are pushed onto a queue. The types are aligned to the token sequence to create an aligned pair. This process is repeated recursively for each directory that contains a "`tsconfig.json`". The final output from our parser is a json for each project. We aggregate the project outputs and prepare the data for de-duplication.

De-duplication is essential, as shown by Allamanis [**7**], prior to training machine learning models; duplication can result in biased performance estimates. Lopes et al. [**136**] identified a large amount of near-duplicate code on GitHub; Allamanis [**7**] released a tool based on Jaccard similarity to help the community avoid this issue. We run the de-duplication tool[7] on the raw corpus to find & remove duplicates. Out of 1,128,744 original files, 204,358 duplicates (about 18%) were found and removed, leaving 924,386 files. After filtering files with annotations 539,571 files remained. The de-duplication is done *without* type annotations, to ensure that even differently annotated duplicates are safely removed; this is different from Mir et al. [**148**]. Mir et al. [**148**] performs lemmatization over variables for classic NLP techniques like TF-IDF. This limits input choices for model developers. With the adoption of subtokenization, subtokenizers *pretrained* on large code corpora are trained to tokenize complete token sequences. By leaving the sequences tokenized in contiguous words, it is up to the model designer to determine how to represent the input. Techiques include: words,

[6]https://pnpm.io
[7]https://github.com/Microsoft/near-duplicate-code-detector

TABLE 5.1. Statistics Across Data Splits

Split	Train	%	Test	%	Validation	%
Projects	11,413	81.8%	1,336	9.58%	1,204	8.62%
Files	486,477	90.16%	28,045	5.20%	25,049	4.64%
Examples	1,727,927	91.95%	81,627	4.34%	69,652	3.71%
Types	8,696,679	95.33%	224,415	2.46%	201,428	2.21%

The data set is split across projects.

TABLE 5.2. JSON schema in ManyTypes4TypeScript

JSON Field	Type	Description
`tokens`	list[string]	Sequence of tokens (word tokenization)
`labels`	list[string]	A list of corresponding types
`url`	string	Repository URL
`path`	string	Original file path that contains token sequence
`commit_hash`	string	Commit identifier in the original project
`file`	string	File name

identifier splitting [**208**], BPE [**193**], WordPiece [**116**], SentencePiece [**117**], lemmatization, etc. This is paramount as Shi et al. [**197**] recently showed that splitting identifiers when combined with BPE subtokens can improve performance.

The de-duplicated set of token sequences, type annotations, and type meta-information is split by projects ∼80%/10%/10% which provides a file split of ∼90%/5%/5% for train/test/validation respectively. More information on the data split can be found in Table 5.1. As shown in Figure 5.1, the JSONL unprocessed data splits are uploaded to Zenodo. Next we define a output vocabulary size of 50,000 and replace any type that exceeds rank 50,000 with an `UNK` token. In classification tasks with finite vocabulary, a special type token `UNK` represents a type guess that exceeds the classifiers prediction capabilities. This is a function of the model and can be changed for models using a larger or smaller classification layer. Additionally, the uninformative "`any`" type annotation is removed from the training and evaluation data. These are standard practices for classification tasks. The schema of files in the Huggingface dataset can be found in Table 5.2. Table 5.2 consists of tokens, labels, repository url, file path, commit hash and file name. This schema is fed into the dataloading script and can also be found on the Huggingface "Dataset card". Finally, the custom Huggingface dataloading script, named `ManyTypes4TypeScript.py`, can be used to generate and

push the dataset to the Huggingface *hub*. This script is available on the Zenodo dataset page so anyone can "fork" a customized ManyTypes4TypeScript dataset.

In the next section, we discuss the design choices of our API Huggingface dataset and how the design of the Datasets Hub [**122**] provides easy to use, optimally compressed access to over 12GB of type inference data.

5.5 Dataset Design and Useability

The ManyTypes4TypeScript dataset conforms to the Huggingface Datasets specification for several reasons. First, the compatible Huggingface transformers library incorporates state-of-the-art models including code specific models like CodeBERT [**62**], GraphCodeBERT [**72**], and CodeBERTa [**234**] which has been widely used across the field especially in CodeXGlue [**137**] for a wide set of tasks and model probing [**107**]. New advancements in transformers are often integrated into Huggingface, thus permitting new applications to existing tasks in addition to easily accessible models [**3**, **62**, **72**, **228**]. It is our goal to make the type inference task as widely applicable to new state of the art transformers with ManyTypes4TypeScript. In later sections we discuss our application of ManyTypes4TypeScript on three SOTA models.

Second, another reason for hosting ManyTypes4TypeScript on Huggingface are the efficiency and scale capabilities. The datasets are capable of being cached completely once downloaded and mapping operations i.e subtokenization and subtoken label alignment are also cached. The datasets are stored as compressed `.parquet` files with Git-LFS (large file storage) and work seamlessly with all available tokenizers and feature-extraction tools. Massive datasets can also be streamed. Model training and evaluation can be accelerated with the Huggingface accelerate[8] library which is particularly helpful for sequence tagging efficiency.

Finally, the tokenizer, dataset and any transformer model can be instantiated in the following five lines of code (LOC).

(1) The dataset is downloaded from Huggingface or instantiated from a local directory.

```
dataset = load_dataset('kevinjesse/ManyTypes4TypeScript')
```

[8]https://github.com/huggingface/accelerate

FIGURE 5.2. Frequency of annotation locations in ManyTypes4TypeScript.

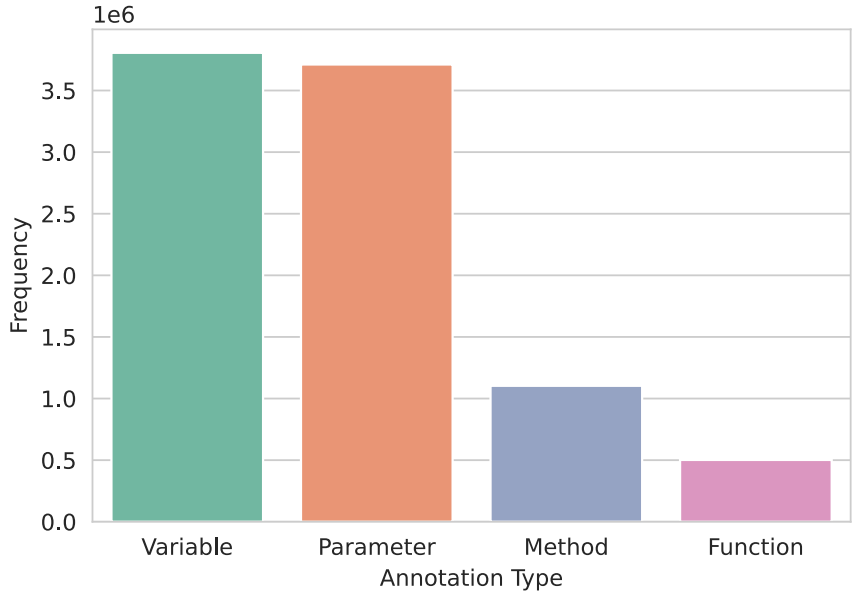
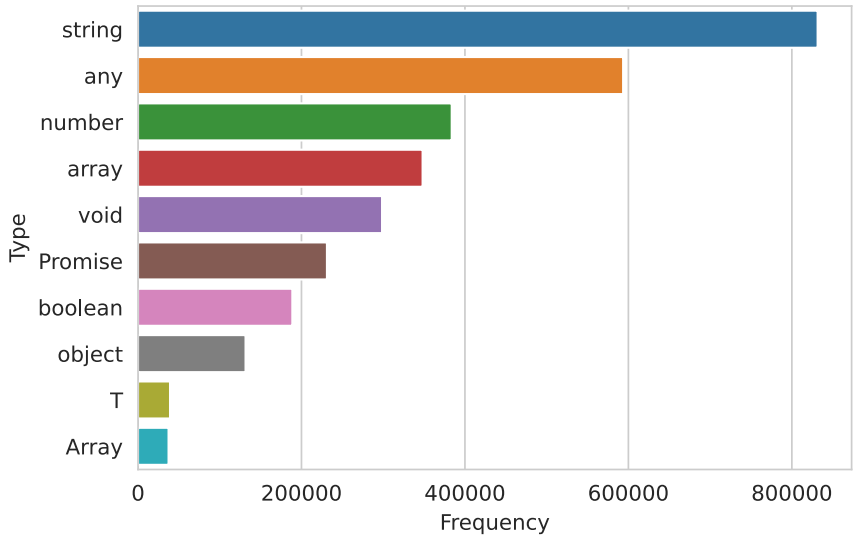


FIGURE 5.3. Top 10 most frequent types in ManyTypes4TypeScript.

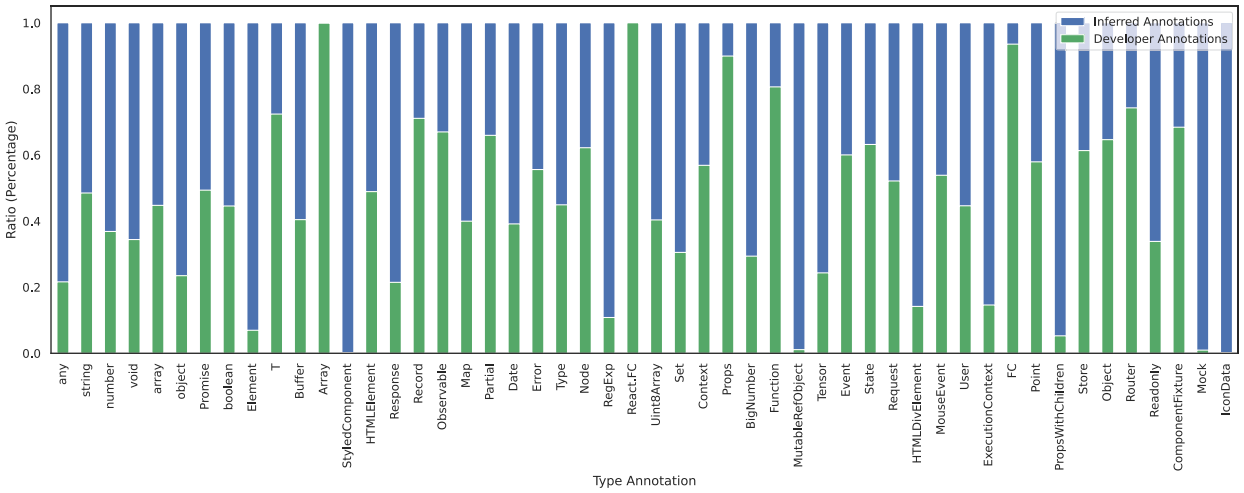


(2) Then the tokenizer is instantiated.

```
tokenizer = AutoTokenizer.from_pretrained('microsoft/graphcodebert-base')
```

(3) The dataset is tokenized into subtokens and the labels are aligned with our provided `align_labels` function to map labels to the first subtoken.

FIGURE 5.4. The ratio by percentage of developer vs. inferred annotations by type in the top 50 most frequent types.




```
tokenized_dataset = dataset.map(align_labels)
```

(4) The label list is extracted from the ManyTypes4TypeScript meta data.

```
label_list = tokenized_dataset["train"].features[f"labels"].feature.names
```

(5) The weights for GraphCodeBert [**72**] are instantiated with a projection layer fit to ManyTypes4TypeScript type vocabulary.

```
model = AutoModelForTokenClassification.from_pretrained('microsoft/
    graphcodebert-base', num_labels=len(label_list))
```

With the above steps, one can instantiate a model with the ManyTypes4TypeScript dataset; the model developer has end-to-end control of model input and output schemes. For example, the model developer can use the `GraphCodeBERT` contextual embeddings for a kNN (k-nearest neighbor) search rather than a classification layer; this would effectively expand the closed-vocabulary output.

The closed type output of the Huggingface API dataset is fixed to 50,000 type categories; but is amenable with the dataset scripts on Zenodo. The current type vocabulary on Huggingface covers approximately 94.08% of all type occurrences as most types are "common" types. The remaining types placed in the `UNK` category cover approximately 5.92% of the 9M types. These types are local and infrequent types, where the types occur less than 10 times corpus wide. Figure 5.2 represents

TABLE 5.3. Accuracy Comparisons On ManyTypes4TypeScript.

Model	Top 100				Overall			
	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy
CodeBERT [**62**]	84.58	85.98	85.27	87.94	59.34	59.80	59.57	61.72
GraphCodeBERT [**72**]	**84.67**	**86.41**	**85.53**	**88.08**	**60.06**	**61.08**	**60.57**	**62.51**
CodeBERTa [**234**]	81.31	82.72	82.01	85.94	56.57	56.85	56.71	59.81
PolyGot [**4**]	84.45	85.45	84.95	87.72	58.81	58.91	58.86	61.29
GraphPolyGot [**4**]	83.80	85.23	84.51	87.40	58.36	58.91	58.63	61.00
RoBERTa [**134**]	82.03	83.81	82.91	86.25	57.45	57.62	57.54	59.84
BERT [**55**]	80.04	81.50	80.76	84.97	54.18	54.02	54.10	57.52

Top 100 types are the most frequent 100 types. Overall is scored with all type locations. `UNK` is considered incorrect.

the frequency of type annotation locations where the majority are variable declarations and function parameters with 3.8 million and 3.7 million annotations respectively. Figure 5.3 represents the frequency of the top 10 most frequent types in the ManyTypes4TypeScript corpus. The majority of types are `string`, `any`, and `number`. With a large majority of human and compiler inferred types resolving to the uninformative "`any`" type, probabilistic type inference has the potential to increase type coverage; type coverage in the optional type setting reaches traditional static typing when all types are annotated or inferred. Finally in Figure 5.4, we examine the ratio of compiler inferred types to human annotations in ManyTypes4TypeScript. We examine that most types are mixed between compiler inferred and human annotations. Corpus wide, the ratio is approximately 57% inferred types to 43% human annotated types. Figure 5.4 shows that only 20% of "`any`" are labeled by humans and the vast majority are inferred by the compiler. The compiler resolves the type to be `any` when the compiler cannot determine the type from existing type constraints. Quantifying a model's ability to resolve the "`any`" type is a possible derivative work from our dataset as "`any`" type annotations are available in the Zenodo data. Lastly, in Figure 5.4, some types are all or nearly all human annotations. This is a unique opportunity for type inference models to assist compilers, alert developers to must have annotations, and resolve types accordingly.

In the next section, we discuss tracking models' performances with a public scoreboard and pushing models trained on the ManyTypes4TypeScript dataset to the Huggingface model hub.

5.6 Tracking Performance and Reproducibility

The ManyTypes4TypeScript dataset on Huggingface is integrated with "Papers With Code"[9] which tracks new papers with consistent metrics. The ManyTypes4TypeScript dataset on Huggingface keeps a list of all models trained or "fine-tuned" on ManyTypes4TypeScript. The models that are trained and evaluated on ManyTypes4TypeScript and pushed to the model hub are linked to the ManyTypes4TypeScript datacard *viz.* homepage. These models can be downloaded and verified in section 5.5. The ManyTypes4TypeScript is currently being integrated into the CodeXGLUE[10] [**137**] set of tasks. CodeXGLUE is a benchmark dataset and open challenge for code intelligence managed by Microsoft Research. With ManyTypes4TypeScript, there is a community driven approach to adding datasets, metrics, models, and documentation to institute a standardization across the type inference task for TypeScript. Next we discuss our supplied metrics.

5.7 Task Specific Metrics and Scores

In the dataset on Zenodo, standard sequence evaluation scripts `seqeval`[11] are available to evaluate the sequence predictions. We modify the ground truth and predictions such that scoring subsets of types can be done easily. We permit classic tagging scoring, considering UNK predictions as incorrect, and top-100 type scoring. The community can add various subsets to the existing metrics such as user-definition and location specific scoring. Our scoring metrics also permit per type evaluation. The dataset in CodeXGLUE will have detailed instructions and scripts to evaluate models, and these scripts will be used to track and verify the task leader-board.

Table 5.3 contains a list of state-of-the-art models scored with the aforementioned metrics. The performance of the models are similar in overall top 100 accuracy to Jesse et al. [**95**] which is completely pre-trained on JavaScript. The performance between the models is in line with previous comparisons [**4**, **107**]. The models provided by us serve as baselines for future contributions. We intend to increase the number of models evaluated across ManyTypes4TypeScript including but not limited to: C-BERT [**33**], CuBERT [**101**], CodeBERTa [**234**], PLBart [**3**], and CodeT5 [**228**].

[9] https://paperswithcode.com/dataset/manytypes4typescript
[10] `https://microsoft.github.io/CodeXGLUE`
[11] https://github.com/chakki-works/seqeval

Additionally, we plan to increase the granularity of the metrics so specific outcomes can be evaluated *viz.* user-defined types.

5.8 Conclusion

In this chapter, we present the ManyTypes4TypeScript dataset of over 9 million type annotations across 13,953 projects and 539,571 files. ManyTypes4TypeScript aims to facilitate the application of new advances in ML-based type inference, with easy to use APIs. ManyTypes4TypeScript standardizes evaluation with the provided test set, metrics, and baselines. By providing the tools used to extract ManyTypes4TypeScript and evaluate state-of-the-art models, we believe that the dataset itself can be a useful resource for the community to maintain and contribute to the type inference task for TypeScript.

Chapter 6

FlexType: A Plug-and-Play Framework for Type Inference Models

6.1 Preface

Chapter 3 through Chapter 5 develop a notion of *developer aligned* type inference models with the latter chapter democratizing access to type inference datasets and models. In this section, we introduce a framework that makes it easier to integrate any new type inference model into the IDE. Ideally, a new model trained and published on Huggingface would require a singular line of code, namely the models url, to be changed in the frameworks startup configuration. To this objective, we build a framework, FlexType, that can be used just that simply. The framework is open sourced as a Visual Studio plugin. This work was featured at Automated Software Engineering 2022 titled *FlexType: A Plug-and-Play Framework for Type Inference Models*. This work was led by myself and Sivani Voruganti with Premkumar Devanbu as an advisor.

6.2 Summary

Types in TypeScript play an important role in the correct usage of variables and APIs. Type errors such as variable or function misuse can be avoided with explicit type annotations. In this work, we introduce FlexType, an IDE extension that can be used on both JavaScript and TypeScript

to infer types in an interactive or automatic fashion. We perform experiments with FlexType in JavaScript to determine how many types FlexType could resolve if it were to be used to migrate top JavaScript projects to TypeScript. FlexType is able to annotate 56.69% of all types with high precision and confidence including native and imported types from modules. In addition to the automatic inference, we believe the interactive Visual Studio Code extension is inherently useful in both TypeScript and JavaScript especially when resolving types is taxing for the developer.

The source code is available at GitHub[1] and a video demonstration at `https://youtu.be/4dPVO5BWA8A`.

6.3 Introduction

Type inference for dynamically typed programming languages, like Python and TypeScript, can help developers improve code quality. By foregoing type annotations, developers coding in dynamically typed languages gain additional flexibility. This flexibility helps developers and designers avoid committing to particular design decisions regarding types. On the other hand, static typing helps detect bugs before execution, and supports both compilation performance and program understanding [**34**, **170**]. Developers have viewed the benefits of static typing as the most desired feature in languages like Python [**96**]. Leading technology companies have developed their own type systems for various languages; Microsoft's TypeScript, Facebook's Flow, and Google's Closure with TypeScript and Flow being syntactic supersets of JavaScript and Python respectively. TypeScript has exploded in popularity over the last few years jumping to the fourth most used language according to GitHub's Octoverse [**69**] in 2020 and 2021. While JavaScript remains the top language, it is a reasonable expectation for TypeScript to further increase in popularity since it can be applied to any JavaScript project with few modifications. TypeScript inherits JavaScript's long standing popularity and widespread adoption so tools built for TypeScript often benefit the JavaScript community as well.

Unlike JavaScript, TypeScript calls for a set of types (either explicitly annotated or inferred) that type the program consistently. Defining a set of types and annotating with said types is not a trivial task for developers; this is called the *type annotation tax*. Type declaration files (.d.ts) and

[1]https://github.com/vsiv16/typescriptsuggestions

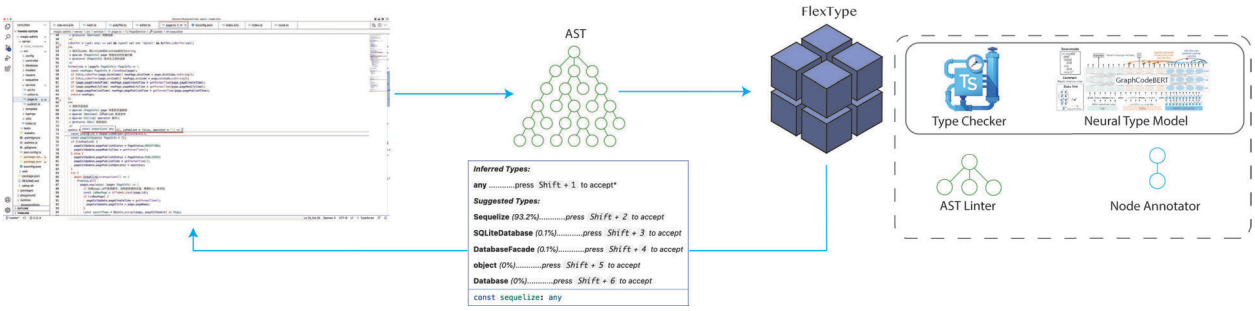


FIGURE 6.1. An overview workflow of the FLEXTYPE framework. To determine the type, the framework parses JavaScript or TypeScript ASTs and passes AST or token information to type checker and open source type inference neural model. The type is converted to a type node and added to the type attribute in the AST. Finally, FLEXTYPE converts the AST to a token sequence for the IDE.

repositories like DefinitelyTyped[2] help alleviate the typing cost by defining general, high quality types which are included automatically by the compiler. The convenience of importing existing types does not supplant the action of annotating the code elements. Moreover, the compiler cannot synthesize types where static constraints or dependencies are not satisfied in the type dependency graph. Frequently, existing tools like TypeScript's type checker are unable to infer types more specific than the generic "any" because it fails to find type hints from static type constraints or package dependencies. Type ambiguity often exists in dynamic typing, because the compiler has too few type constraints to resolve [**38**]. Type ambiguity is more prevalent in languages like JavaScript, than in explicitly typed languages like TypeScript, where developers have no explicit annotations and must rely on interpretation, documentation, and surrounding expressions to determine the likely types. Thus, developers could benefit from tools that recommend likely types and insert types with little to no effort.

For these reasons, the *type inference* task has been well studied, in the software engineering research community [**11**, **78**, **95**, **149**, **157**, **164**, **172**, **182**, **230**]. Most of these works in type inference are a result of the abundance of code and the success of deep learning for software engineering. The abundance of patterns in code warrants probabilistic models to exploit the regularity of code; in type inference it is the regularity of how types are used. The newest advances in machine learning [**55**, **121**, **125**, **134**, **177**] often come with downstream improvements to software engineering

[2]https://github.com/DefinitelyTyped/DefinitelyTyped

models [**3**, **10**, **62**, **72**, **102**, **228**], but in practice, these improvements have not been tangible to developers as most published models stop short of publishing IDE tools. We argue that the gap between model development and model deployment in integrated development environments is worthwhile, but challenging.

To address this gap, in this chapter, we present our tool FLEXTYPE, a plug-and-play framework for any new state-of-the-art type inference model in a VSCode environment for TypeScript *and* JavaScript. JetBrains found that 60% of JavaScript and TypeScript developers use Visual Studio or Visual Studio Code as their preferred IDE [**97**]. The core idea behind FLEXTYPE is the integration of such models in an interactive and automatic way that complements existing static type checking capabilities, even in dynamically typed languages like JavaScript. To evaluate our idea, we have implemented an extension for Visual Studio Code, a popular IDE from Microsoft, using one of the of several type inference models from ManyTypes4TypeScript [**94**]. Our contributions are as follows,

- An interactive, model-agnostic framework for type inference in Visual Studio Code.
- A tool that uses the AST to correctly insert type elements from sequence-based or graph-based models.
- A use-case experiment evaluating the effectiveness of FLEXTYPE in migrating JavaScript projects to TypeScript.

6.4 Related Work

The landscape of type completion tools ranges significantly in capability from static checking [**25**], neural type inference [**11**, **78**, **94**, **95**, **149**, **172**, **230**], and code completion-like type generation [**3**, **17**, **40**, **218**, **228**, **239**].

Static type checking from the TypeScript compiler occurs when the TypeScript compiler transpiles TypeScript to JavaScript. The TypeScript type checker can be accessed through a shipped version of TypeScript installed with the IDE. The IntelliSense feature in Visual Studio and Visual Studio Code can provide underlying types by relying on the internal type checker for TypeScript. The type checker is capable of performing type inference from the variable's value as long as the type constraints exist. For example, the variable `i` in `var i = 0` can be inferred as a number from the value in the assignment expression. Any high-level interpretation of `i`, such

as the use of `i` as an iterator, cannot be inferred by the type checker without a higher order type indicating such functionality. In JavaScript, the IntelliSense method signature information shows the uninformative "any" type for the method parameters because JavaScript is dynamic and does not enforce types [**145**]; this is not particularly helpful for a developer wishing to pass the correct type to the function.

Neural type inference and code completion aim to model attributes of source code probabilistically by exploiting the regularity of software [**8**, **81**] and an abundance of existing typed code on open source repositories. In contrast to static type inference, neural type models rely on large code corpora and can suggest richer, more contextualized type annotations overcoming the lack of existing type constraints realized when the compiler predicts "any"; in our experiments this occurs 63.46% of all typeable identifiers. Our goal here is to build a flexible way to integrate neural type inference models into an IDE, to make these models more accessible.

Some published neural type inference models Typilus [**11**], HiTyper [**164**], and LambdaNet [**230**] expose inference methods where the model can be called on a set of source code files and the appropriate annotations are logged in an output file; this is impractical for the typical developer and such models often require computing not found on a laptop. One model cites the need of a "high-end Nvidia GPU with at least 8GB of RAM" and "a CPU with 16 threads or higher" [**149**]. The requirements for running massive code generation models like Codex [**40**] (Copilot), Google's 137B parameter model [**17**], and PolyCoder [**239**] is further beyond any consumer PC, thus access to such models must be by remote API. Remote API access is viable for many developers, but communicating large token windows of proprietary software introduces valid security and privacy concerns [**40**, **159**, **209**]; a local type inference model is ideal. In contrast, FlexType uses local models that can run efficiently on a laptop CPU. The user can simply hover over the variable, parameter, function or method to get a drop down list of types including the compiler inferred type, if any, and see the type properly inserted.

In the following sections, we present our approach, implementation, and evaluation of FlexType.

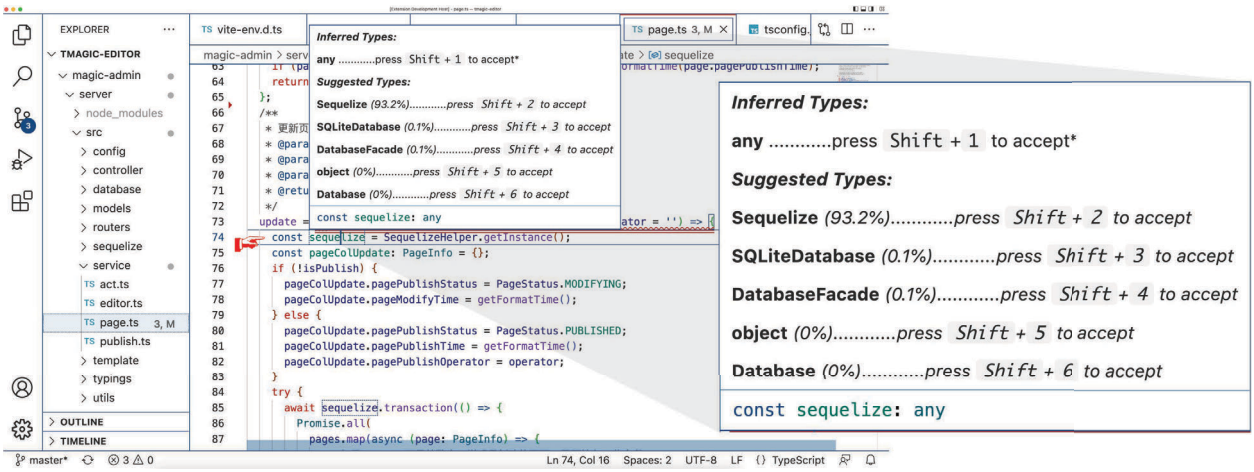


FIGURE 6.2. A snapshot of the FLEXTYPE VSCode extension.

6.5 Approach

Figure 6.1 shows how FLEXTYPE interactively works with the developer to recommend types. When the developer toggles the VSCode extension, FLEXTYPE activates the mouse hover action which pops up a list of types. By default, VSCode provides existing prototype information with type annotations that are written in the code such as `const sequelize: any` in Figure 6.2. FLEXTYPE presents an informative list to the developer integrating *compiler inferred types* (often useful for native types and user-defined types) with *the neural type suggestions*. The neural type suggestions can be quite useful to the developer, because the type recommendations derive from large corpora training, and elucidate types that local constraints often cannot resolve.

Figure 6.2 illustrates a key situation where the type assistant shines. With the current type constraints, the compiler cannot resolve what type `sequelize` is. The term "sequelize" in itself is a natural language hint, one that hints at it connecting to a SQL database often pronounced "see-kwl". While these natural language hints are not always readily available, the syntax and usage of function calls are, which deep learning models capture. The resulting list of contextually derived types, as seen in Figure 6.2, is helpful in understanding the likely functionality of such APIs. The developer has the liberty to choose which type annotations are useful with the model's perceived probabilities. This feature is available for TypeScript *and* JavaScript files as TypeScript data transpiles into JavaScript code and thus captures otherwise implicit type information in JavaScript. JavaScript

syntax does not permit types, so types are not "insertable" when interacting with JavaScript. We believe FlexType can help both TypeScript and JavaScript developers, as type information improves code readability, comprehension, and proper usage of code elements. In the following text, we discuss the details of the approach within the framework's pipeline.

FlexType starts with an incremental compilation[3] of the program, targeting just the current editor file for AST parsing. Then the framework digests the developers current word token index[4] and finds the character offset of the token which aligns best with the `pos` (position) field in the parsed AST. FlexType uses an AST linter to traverse the AST in *preorder*, filtering only valid typeable locations. For each leaf node (indicating a code token) the corresponding code token is appended to a list of tokens which will serve as the tokenized input to the machine learning model; tokenizing from the AST has the benefit of filtering out non-code related tokens such as comment blocks. In the traversal, the framework keeps a cache of the parent type because the parent node is where the type annotation is located, specifically, in a variable, parameter, function, or method declaration syntax node. Finally, when the identifier syntax node corresponding to the identifier of interest is visited, which is a child to the typed parent, this token is aligned to the cached AST type and to the current token index. The cached type node is fed to the type checker which returns the result of any the static type constraints, if any, for that identifier. Finally, the token sequence, inferred type, and token index is returned. If the developer's cursor location is not at a typeable variable, parameter, function, or method declaration, the AST linter is immediately returned with null values.

The token sequence and token index is passed through a localhost port to a WSGI Flask[5] server started as a background task when the extension is enabled. This server encapsulates the neural inference model. The token sequence is subtokenized using the neural model's tokenizer and the new subtoken index is calculated. The framework then determines an optimal context window around the identifier of interest; this is necessary for long files as a model's sequence-based input is limited. The type inference model in our demo, is a Huggingface type inference model based on

[3]An incremental compilation saves compute resources when previous changes are minimal across a set of files and project dependencies.
[4]The term *position* is usually synonymous for token indexes in sequences across NLP literature, but is confusing in the context of the AST, thus we only use it when referring to the AST.
[5]https://flask.palletsprojects.com/en/2.1.x/

the popular GraphCodeBert [**72**]. Here, we emphasize the "plug-and-play" dynamic where neural type inference models, such as our `from_pretrained('microsoft/graphcodebert-base')`, is amenable with alternative choices. With respect to future proofing our design, our GraphCodeBert [**72**] type inference model improves upon CodeBert, namely, where data flow awareness is principle to performance. For type inference, GraphCodeBert increases performance, likely due to the role data flow plays in types. Finally, these neural suggestions are serialized and returned to the VSCode portion of the framework where the types are displayed to the developer.

For a TypeScript (.ts) file, the framework presents the recommended types with keystrokes to embed the types as formal type annotations. For a JavaScript (.js) file, the framework shows the developer the type recommendations only. If the developer chooses a type, the framework performs a *postorder* AST traversal to return to the identifier's parent node, generate a type node from the type, and assign the type node to the parent's `type` field. The traversal is immediately returned, returning the root node of the sourcefile which is then used to synthesize the file with the type annotation in the correct location; this insertion technique is guaranteed by the compiler's printer to work for any valid type node. Since the type node's synthesis is independent of the actual type value, FlexType can always guarantee correct type placement. Finally, the VSCode editor is updated with the new type embedded sequence. In the next section, we discuss the high level implementation design.

6.6 Implementation

We implemented our approach as an extension to Microsoft's *Visual Studio Code*, Figure 6.2, which is the most adopted TypeScript/JavaScript IDE according to a JetBrains survey [**97**]. We implement the client in TypeScript where VSCode can pass actions such as hover, click and drag, and keyboard strokes to the client. Upon a hover over a type permissive location (variable, parameter, function, method), FlexType performs static and neural type inference and recommends types. The modularity of the static type checker and the neural type model permits the interchange of a variety of models with minimal changes. While sequence-based methods (RNN, Transformer, Pretrained Language Models) are very popular, there is an increasing demand for models that capture code structure (GNN [**10**], Hybrid [**80**]). In addition to the "plug-and-play" neural type

TABLE 6.1. Recall Percentage of Types Across Top 5 Projects

Repo	Stars	TC (%)	TC + NN (%)
goldbergyoni/nodebestpractices	77728	33.73	60.0
Dogfalo/materialize	38682	29.02	52.6
yangshun/front-end-interview-handbook	33963	19.83	45.69
quilljs/quill	32667	26.28	55.01
marktext/marktext	31921	33.1	56.63

FLEXTYPE recall uses only the static type checker (TC) and FLEXTYPE using both the type checker and neural type inference model (TC+NN).

architecture, FLEXTYPE re-synthesizes the snippet of code with the TypeScript Compiler API[6]. By altering the AST, rather than the code sequence itself, the framework is compatible with graph-based methods. Finally, for best results, we apply graph optimization and quantization to the neural type inference model, which results in blazing quick inference times under .4 seconds on a Intel 8th generation Coffee Lake and even faster on Apple M1. In the next section, we perform an experiment to simulate the impact of our tool for developers migrating from JavaScript to TypeScript and equivalently coding only in JavaScript.

6.7 Evaluation

FLEXTYPE uses both type checker and neural type inference models. To evaluate FLEXTYPE's effectiveness migrating JavaScript to TypeScript, we checkout over 150 most-starred Javascript repositories and let FLEXTYPE annotate them as best as it can. For brevity, we have only included 5 of these projects in Table 6.1 with the full results available at our GitHub.

The 150 JS projects have no human annotated types, so performance must be evaluated with an oracle. The neural type model *per se* can serve as an oracle if it's confidence threshold is set such that precision remains very high; only if a type prediction is above this threshold can it be labeled as correct. To calibrate, we measure the precision-recall curve of GraphCodeBERT on the ManyTypes4TypeScript [**94**] test set. This is a dataset of manually-annotated Typescript projects allowing direct performance evaluation. GraphCodeBERT achieves a precision of 89.10% and recall of 53.83% across 224,415 types with a 90% confidence threshold. Thus, we can use GraphCodeBERT's confidence threshold with a precision of 89.10% as a proxy to the number of

[6]https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API

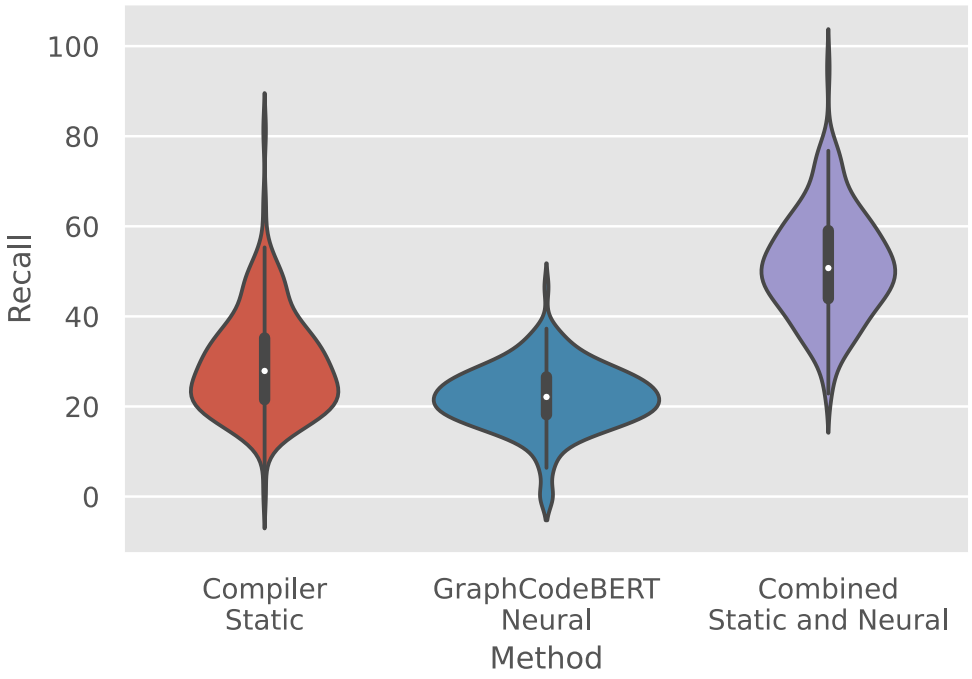


FIGURE 6.3. Static, neural, and combined recall of FlexType components per project.

types that can be resolved. In other words, 89.10% of JavaScript types with a confidence of 90% or greater is a reasonable metric for evaluation. While this method is effective, it is important for us to calculate how much we are potentially underestimating our model's performance.

The recall of 53.83% means that 46.17% of types fall below the confidence threshold. We can calculate the precision across the 46.17% set of types to determine how many types were missed. This precision is 31.51% and so the model's recall is underestimated at most by 15% (31.51% of 46.17%).

We emphasize that this performance is for the top-1 (the model's best guess), and ignores selecting the 2nd or 3rd best choice in the interactive dialog seen in Figure 6.2. In the interactive setting with 5 choices, the recall is naturally higher than in the top-1 setting. We use top-1 in our automatic evaluation of FLEXTYPE to estimate a lower bound of performance in a common use case, migrating JavaScript to TypeScript.

RQ1: What is the recall of types for FLEXTYPE across the top 150 starred JavaScript projects?

Across the set of 150 projects, 56.69% of types are resolved by FlexType. The recall of the compiler is 36.54% and the neural type inference model provides the additional 20.15% recall. On a per project evaluation, the mean project recall is 51.44% with the compiler providing 29.49% of the types and the neural type inference model providing an additional 21.95% recall. The per project recall distribution of each component can be seen in Figure 6.3.

This evaluation suggests that FlexType helps annotate a good fraction of type locations (56.69%) in JavaScript; this reduces the annotation burden in JavaScript to TypeScript migration. Moreover, we argue that developers will use the tool in an interactive fashion using the drop down menu in Figure 6.2. This should further increase the recall which represents the number of type constraints the developer could reasonably add with minimal effort.

6.8 Conclusion

As a development tool, FlexType can help increase the volume of type annotations. We also see an opportunity to use FlexType in an automated setting to improve type annotation coverage in existing and new projects. We hope the adoption of this framework can reduce the burden of adding type annotations in TypeScript and the reduce the misuse of variables and APIs in both TypeScript and JavaScript, thus improving software development and maintenance.

Chapter 7

Large Language Models and Simple, Stupid Bugs

7.1 Preface

So far we have presented our work on models that solve developer-facing problems like type inference. Code assistance has since captured the likes of OpenAI, Microsoft, and more. The meteoric rise of code completion tools like Copilot [**249**] is largely based on the success of large language modeling [**40**]. "Code assistants" are prevalent in 2023 and provide developers with entirely new capabilities. In this chapter, we examine the implications of these models, specifically, whether they generate hard-to-locate single statement bugs. This chapter serves as the groundwork for future work in Chapter 8. The paper is titled *Large Language Models and Simple, Stupid Bugs*, and is by myself, Toufique Ahmed, Premkumar Devanbu, and Emily Morgan; it will appear at Mining Software Repositories (MSR) 2023.

7.2 Summary

With the advent of powerful neural language models, AI-based systems to assist developers in coding tasks are becoming widely available; Copilot is one such system. Copilot uses Codex, a large language model (LLM), to complete code conditioned on a preceding "prompt". Codex, however,

is trained on public GitHub repositories, *viz.*, on code that may include bugs and vulnerabilities. Previous studies [**16**, **162**] show Codex reproduces vulnerabilities seen in training. In this study, we examine how prone Codex is to generate an interesting bug category, single statement bugs, commonly referred to as simple, stupid bugs or SStuBs in the MSR community. We find that Codex and similar LLMs do help avoid some SStuBs, but do produce *known, verbatim* SStuBs as much as 2x as likely than *known, verbatim* correct code. We explore the consequences of the Codex generated SStuBs and propose avoidance strategies that suggest the possibility of reducing the production of known, verbatim SStubs, and increase the possibility of producing known, verbatim fixes.

7.3 Introduction

The rise of language-model based AI coding tools promises to change programming practice. Developers can now use AI coding tools, which inherit their power from models trained on enormous corpora of open-source code. Copilot, a language-model based coding assistant [**249**], is available in many integrated development environments (IDEs). Copilot uses a model named Codex [**40**] to generate code completions. The full power of Codex is still being learned: it can already perform a diverse set of tasks including: code completion [**40**], automatic program repair (APR) [**173**], comment generation [**5**, **40**], program synthesis [**91**, **204**], and incident management [**6**].

Copilot is free to use, and is widely adopted. It is an attractive tool for developers at different skill levels; it helps provide starting points for developers [**219**] and can start functions from just input, output examples [**204**]. The capabilities of Codex and similar models [**63**, **153**, **217**, **239**] have raised many concerns, and have given rise to different research thrusts. Active avenues of Copilot research include how developers work with it: researchers report concerns on an over-reliance and unwarranted trust in Copilot-generated code [**190**, **219**], the quality of the completions [**152**, **243**], security implications [**16**, **160**, **166**, **189**], and copyright infringement [**106**]. These research topics are motivated by the free access and popularity of Copilot, particularly, the use of code it generates may give rise to broader ethical and functional concerns.

Codex has been found to work for program repair [**161**, **173**], problem solving [**17**, **106**], math [**58**, **211**], and translation from natural language to various target languages [**17**, **40**, **178**, **214**]

to name a few applications. With many use cases, Codex is a double-edged sword of utility and risk and ultimately we should find ways to minimize the risk and maximize the utility of Codex.

To that objective, this work examines Codex on the 2021 MSR mining challenge dataset ManySStubs4J [**105**]. The dataset consists of single statement 'simple, stupid bugs' (SStuBs) mined from Maven projects. Karampatis and Sutton found that SStuBs have a frequency of 1 in every 1,600 LOC and that static analyzers cannot detect them [**105**, **151**]. Mosolygó et al. [**151**] determined that SStuBs appear more in larger chunks of code authored by the same developer, perhaps due to loss of attention or misunderstanding of code functionality. We see this exact phenomena in Codex studies [**190**, **219**] where developers use Codex to generate large blocks of code and, *if* a bug is found, dive into a time-consuming rabbit hole to fix the code [**219**]. Worse, this study reports that developers often blindly trust generated code, or optimistically hope to fix problems later.

Surprisingly, so-called "simple, stupid" bugs can survive a long while; in the SStuBs dataset, fixes take around 240 days [**119**, **151**]. More worryingly, when Codex generates code, an 'agent' other than the active IDE user is actually 'coding', and thus the number of commits to fix the SStuB might be even longer (see Mosolygó et al. [**151**]). This disappointing possibility is supported by surveys [**190**, **219**], suggesting that developers don't always understand generated code, and struggle to fix any bugs therein.

The performance of Codex has been extensively benchmarked [**40**, **161**], checked for security vulnerabilities [**106**, **160**, **166**, **189**, **189**], and empirically evaluated [**16**, **152**, **190**, **219**]. However, Codex has not been evaluated against SStuBs, which are a special kind of bug [**88**]. To understand SStuBs related to AI-supported programming, we evaluate whether Codex and other code completion models produce SStuBs, or their fixes; also we look at the consequences of such bugs in code bases. Finally, we present a Codex experiment aimed at *automatically* communicating developer *intent*, using *generated* comments, to help avoid introducing simple, stupid bugs, and also producing commented code.

Our research questions are as follows, all primarily evaluated using the ManySStuBs4J dataset:

- **RQ1:** How often do Codex and similar language models (CodeGen [**153**] and PolyCoder [**239**]) produce simple, stupid bugs?

- **RQ2:** When Codex generates the same simple, stupid bug that a human does, how much time does the SStuB originally take to fix?
- **RQ3:** Does Codex produce buggy or correct code more confidently (*viz.* at higher probability)?
- **RQ4:** Does adding automatically generated comments to the prompt help Codex and akin language models avoid SStuBs? Do other types of prompt improvements help reduce SStuBs?
- **RQ5:** How do bug-derived code comments, when inserted in the prompt, affect code generated by Codex and other LLMs?

Our key findings are: (1) LLMs do help avoid some SStuBS in our dataset! (2) Codex and other LLMs do produce *known* SStuBs, and at a rather high rate (perhaps twice as often as they produce *known* correct, bug-fixing code); (3) When Codex generates a known SStuB, it's associated (historically) with longer fix-times; (4) Codex-generated completions appear equally 'natural' [**81**], regardless of whether they match buggy code or the related fix; if they match neither, they are less natural. (5) Automatically generated comments, when added to prompts, appear to reduce the known SStuB production rate for most models and improve the bug/patch ratio; (6) Even buggy comments help to reduce the bug/patch ratio in Codex, suggesting just attempting to comment code helps. The improvement in avoiding SStuBs with comments from neural comment generation model CodeTrans [**59**], suggest that using these models *with* Codex would be beneficial to avoid SStuBs.

Data from this study is available here[1].

7.4 Related Work

We discuss related work on language models and code quality.

7.4.1 Simple, Stupid Bugs

Simple, stupid bugs (SStuBs) are bugs that have single-statement fixes that match a small set of bug templates. They are called "simple" because they are usually fixed by small changes

[1] `https://doi.org/10.5281/zenodo.7676325`

and "stupid" because, *once located*, a developer can usually fix them quickly with minor changes. However, locating SStuBs can be time-consuming [**105**]. Karampatsis and Sutton [**105**] published a collection of SStuBs mined from a set of template bug types, *e.g.*, `CHANGE_IDENTIFIER` or `DIFFERENT_METHOD_SAME_ARGUMENTS`. Through their study of the dataset, Karampatsis found that SStuBs are prevalent in code bases, accounting for 33% of single statement bugs detected in 1000 Maven projects. They found that these bugs occur every 1,600 lines of code, and were not detected by static analysis. MSR once used ManySStuBs4J dataset as a mining challenge, to study these bugs, and manage their impact.

Mosolygó et al. [**151**] studied the history of SStuBs. They find that SStuBs are more frequent in code modified by the same developer, often when s/he writes large chunks of code. This is perhaps because such large coding tasks strain focus and attention. They found that only 40% of SStuBs were fixed by the same author in a median time of 4 days; when the SStuB is fixed by a different author, the SStuB took 136 days to find and fix! We hypothesize that if comments were present in a Codex prompt, we'd get better code completions, *and* the entire chunk would be easier to read & fix.

Zhu and Godfrey [**248**] studied how developers fix SStuBs. Similar to Mosolygó et al. [**151**] they found that developers fix their own bugs quicker, whereas bugs from other developers take significantly more time to fix. This suggests that Codex generated SStuBs *may* take longer to fix since they come from an artificial 'developer'. Codex-generated code-snippets that include SStuBs may require extensively debugging to be patched in a similar fashion. A Copilot study [**219**], found that users had trouble debugging generated code from Codex spending considerable time and effort to fix, for instance, a generated regular expression.

Madeiral and Durieux [**141**] discussed SStuBs in the context of code clones [**187**] and the changes that introduce them, *viz.* "change clones". They found that 29% of change clones introduced SStuBs by matching the 16 SStuB patterns. Since Codex is a language model that tends to repeat code it's seen, it could conceivably generate SStuBs in multiple locations, increasing the repair effort.

Peruma and Newman [**167**] examined SStuBs in unit test files. They found that SStuBs tend to occur in non-test files and that developers fix the bugs separately despite test and non-test files

being functionally related. Peruma and Newman also discovered that developers prioritize non-test files and the fixes in tests are associated with asserts.

Latendresse et al. [**119**] and Hua et al. [**88**] addressed the detection of SStuBs. Latendresse et al. [**119**] found that continuous integration (CI) tools cannot catch any SStuBs. Hua et al. [**88**] found that deep learning vulnerability detectors were suboptimal compared to traditional vulnerability detectors on SStuBs. Our results confirm that models as large as Codex, find SStuBs to be equally regular to the patches it generates.

Mashhadi et al. [**144**] applied CodeBERT [**62**] (fine-tuned for patching) to SStuBs and could fix 19% of de-duplicated Many4SStuBs4J dataset. Mashhadi et al. mentions an advantage of using CodeBERT: no special tokens are required like in SequenceR [**41**]; similarly, many APR techniques with Codex [**161**, **173**] rely on a prior of knowing a bug exists or even, more specifically, the bug location [**41**]. Adding comments to the prompt does not require making any of these assumptions.

Finally, PySStuBs [**100**] is a Python simple, stupid bug dataset. The more recent TSSB-3M [**185**] is also a Python SStuBs dataset mined at scale. Our study focuses on the established Java SStuBs patterns from the everpopular ManySStuBs4J dataset. We plan to expand our findings to other languages and SStuB patterns, resources permitting. Currently OpenAI restricts usage of Codex to 20 requests per minute. Inference on large models for ManySStuBs4J takes just over a day.

7.4.2 Examining Codex Completions

Vaithilingam et al. [**219**] studied the developer experience with Codex. The key findings were that most participants preferred Codex to Intellisense in Visual Studio IDE. Participants preferred to use Codex as a starting point in lieu of searching online. Unfortunately participants over-relied on Codex and then struggled when generated code was buggy. The authors reported three major issues: (1) participants often didn't understand and assess the correctness of generated code, (2) participants underestimated the repair-effort required when generated code was buggy, (3) the prompts used by participants were quite varied, sometimes resulting in undesired code completions.

Sarkar et al. [**190**] wrote an extensive review of programming with an AI assistant Codex. Sarkar et al. surveys previous work citing Codex's reliability, safety, and security implications. The

review covers studies in Codex usability, design, and user reports. Sandoval [**189**] and Perry [**166**] examine Codex security implications.

Yetistiren et al. [**243**] and Nguyen et al. [**152**] empirically studied Copilot's code suggestions. Yetistiren et al. [**243**] found Copilot mostly generated valid code and Copilot improved it's correctness with further input from the developer; sample examples, unit tests, docstrings, and prompts increased correctness further. Nguyen et al. [**152**] found Copilot correctness varies by programming language and does not differ in complexity (cognitive and cyclomatic) among programming languages.

Prenner et al. [**173**], Pearce et al. [**161**, **162**], Karmakar et al. [**106**], and Ahmed et al. [**5**, **6**] apply Codex in various settings: automatic program repair (APR) [**173**], security vulnerability prediction [**161**], HackerRank challenges [**106**], code summarization [**5**], and incident management [**6**]. In APR, Prenner et al. tried engineering prompts to find a way to push Codex to generate a non-buggy version of the code. Codex performed competitively to recent work but was sensitive in the prompt. Pearce et al. found Codex could repair 58% of real world security vulnerabilities. Karmaker et al. applied Codex on HackerRank problems with great success; some of the success was attributed to Codex already knowing the solution despite an incomplete prompt, in other words, memorizing the solution. Ahmed et al. trains Codex on few-shot project-specific code to achieve state-of-the-art code summarization. Ahmed et al. found success in using Codex to help engineers diagnose and mitigate production incidents. All of these works use prompting to illicit a desired response from Codex.

Prompting instructions with natural language or code is often implemented as a comment to the prompt passed to Codex. The prompts are implemented as comments to improve generated code, *e.g.*, natural language instructions in docstrings or input-output examples. Prompting relies on knowledge *a priori* that Codex should adopt into its generations. A prompt could be: an example completion, a problem description, input format, code that is vulnerable, a docstring documenting a bug location, input-output pairs, and snippets of bugs and corresponding patches (few-shot). In this work, we explore similar prompting techniques from Prenner [**173**], *and* focus on *prior-free* prompting through traditional commenting practices. We find that good commenting practices can guide Codex to more SStuB-free completions.

JBTabsImpl.java

```
1524        if (paintBorder.top > 0) {
1525            if (isHideTabs()) {
1526                if (isToDrawBorderIfTabsHidden()) {
1527                    g2d.setColor(borderColor);
1528                    g2d.fill(shaper.reset().doRect(boundsX, boundsY, boundsWidth,
1529                            1).getShape());
1530                }
1531            }
1532            else if (isStealthModeEffective()) {
1533                //g2d.setColor(borderColor);
1534                //g2d.fill(shaper.reset().doRect(boundsX, boundsY - 1,
1535                // boundsWidth, 1).getShape());
1536            }
1537            else  {
1538                Color tabFillColor = getActiveTabFillIn();
1539                if (tabFillColor == null) {
1540                    tabFillColor = shape.path.transformY1(shape.to, shape.from);
1541                }
1542
1543                g2d.setCcolor(tabFillColor);
1544                g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
1545                        boundsWidth, paintBorder.top).getShape());
1546
1547                //If the top of the border is a non-paint border, then the border is painted.
1548                if (
1549 Bug ->             paintBorder.top >= 1
1550                    paintBorder.top > 1   <- Fix
1551                ) {
1552                    g2d.setColor(borderColor);
1553                    final int topLine = topY + shape.path.deltaY(paintBorder.top - 1);
1554                    g2d.draw(shaper.reset().doRect(boundsX, topLine, boundsWidth - 1,
1555                            1).getShape());
1556                }
1557            }
1558        }
1559
1560        g2d.setColor(borderColor);
1561
1562        //bottom
1563        g2d.fill(shaper.reset().doRect(boundsX, Math.abs(shape.path.getMaxY() -
1564                shape.insets.bottom - paintBorder.bottom), boundsWidth,
1565                paintBorder.bottom).getShape());
1566
1567        //left
1568        g2d.fill(shaper.reset().doRect(boundsX, boundsY, paintBorder.left,
1569                boundsHeight).getShape());
1570
1571        //right
1572        g2d.fill(shaper.reset().doRect(shape.path.getMaxX() - shape.insets.right -
1573                paintBorder.right, boundsY, paintBorder.right,
1574                boundsHeight).getShape());
1575
1576    }
```

FIGURE 7.1. The orange highlighted code is the candidate single line completion that Codex can match to the automatic evaluation either the known bug or fix. Blue highlighted code is the prompt *a.k.a.* the text proceeding SStuB statement that Codex uses for completion. The purple highlighted code is the code after the SStuB.

7.5 Methodology

In this section, we describe the methodology for evaluating Codex [**40**], PolyCoder [**239**], and CodeGen [**153**] on SStuBs.

```
    if (WRITE_TYPE.isCache()) {
      try {
        int tLen = len;
        int tOff = off;
        while (tLen > 0) {
          if (mCurrentBlockLeftByte == 0) {
            getNextBlock();
          }
          if ("
```

Bug:

```
"mCurrentBlockLeftByte > tLen"
```

Fix:

```
"mCurrentBlockLeftByte >= tLen"
```

Model Davinci:

```
"tLen >= mCurrentBlockLeftByte ){
    mCurrentBlockOutputStream.write(b, tOff, mCurrentBlockLeftByte);
    tLen -= mCurrentBlockLeftByte;
    tOff += mCurrentBlockLeftByte;
    mCurrentBlockLeftByte = 0;"
```

bug | fix | incorrect | unsure

(a) This completion is incorrect.

```
  public static void checkMinimalPollingPeriod(TimeUnit pollUnit, int pollPeriod)
      throws IllegalArgumentException {
    int period = (int) MINIMAL_POLL_UNIT.convert(pollPeriod, pollUnit);
    Preconditions.checkArgument("
```

Bug:

```
"period < MINIMAL_POLL_PERIOD"
```

Fix:

```
"period >= MINIMAL_POLL_PERIOD"
```

Model Davinci:

```
"period >= 1, "Polling period %s %s is smaller than 1%s", pollPeriod, pollUnit, MINIMAL_POLL
}

/**
  * Checks if the poll period is smaller that the minimal poll period which is 1 second"
```

bug | fix | incorrect | unsure

```
",
        "Polling period %d %d is below than minimal polling period", pollPeriod, pollUnit);
  }
```

(b) This completion matches the fix. Earlier in the code, the constant `MINIMAL_POLL_PERIOD` is set to 1.

FIGURE 7.2. This annotation tool helps mark Codex completions that do not match any SStuB directly. This guarantees our evaluation is not missing reasonable alternatives to the SStuB that could be deemed a bug or fix.

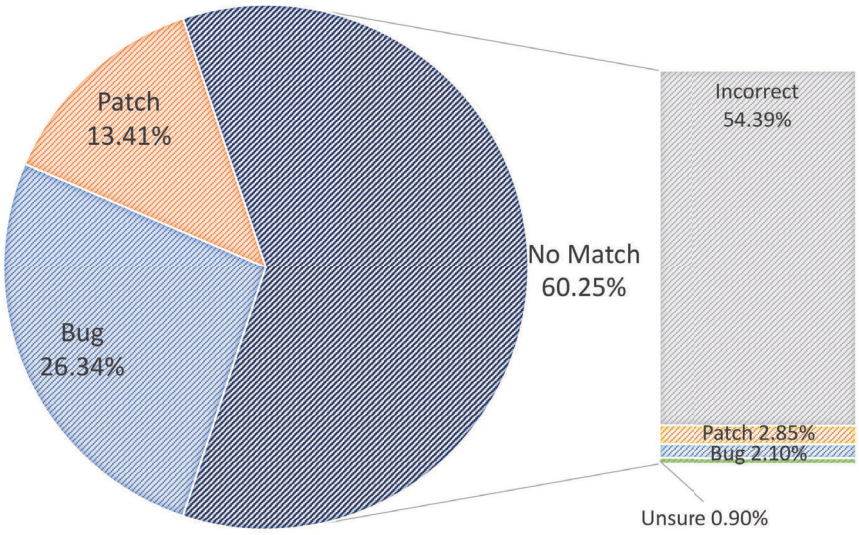


FIGURE 7.3. Match rate of Codex Davinci (left). Completions that do not match a patch or SStuB are validated by hand (right).

7.5.1 Experimental Setup

ManySStuBs4J

This dataset consists of a small and large dataset with 10,231 and 63,923 single-statement bug-fix changes (*a.k.a* SStuBs) mined from 100 and 1000 popular repositories respectively. These SStuBs must match one of 16 bug templates. The goal is to collect bugs that are difficult to locate, but easy to fix. It's natural to wonder if automated coding tools based on language models could introduce such single-statement bugs.

For our study, we use the ManySStuBs4J *large* dataset of 63,923 samples and use `git checkout` to obtain versions of the bug prior and after being fixed. We use `git blame` to determine when the bug was introduced and capture key statistics such as the number of commits to fix the SStuB. The bug locations within the files are indexed with fields `bugNodeStartChar` and `bugNodeLength`. Using these fields we can find the code before the bug, the bug, the fix, and the code after the bug/fix. Our experiment focuses on giving Codex a piece of code prior to the bug and seeing if Codex generates the correct code (fix) or incorrect code (bug). A large concern in this experiment is to make sure Codex has an equal opportunity to generate the known bug or patch. Therefore, we remove SStuBs that have other changes besides fixing the SStuB, which could otherwise *condition* or bias Codex to

make a decision; for example, a new variable only exists in the buggy version so Codex completes the bug. This leaves 34,595 bugs prior to deduplication. Then we drop duplicates for bugs that share the exact same prefix, bug, and fix. It is important to not inflate results by duplicate code examples [**7**]. The remaining 16,899 SStuBs are used for evaluating all models.

Codex, PolyCoder, CodeGen

Large language models like Codex, PolyCoder, and CodeGen are demonstrably useful in code completion tools like Copilot, and are available for experimentation. Other models exist, like AI21 Jurassic; however they are not free, and would be costly at our scales, so were excluded from our study. We also didn't have the computational resources to run very large models locally. For these reasons, we follow the methodology from Xu et al. [**239**] and use CodeGen, PolyCoder, and Codex.

Codex derives from GPT-3; its training data consists of natural language and source code from available sources like public GitHub repositories. Codex has two sizes one called *cushman-codex* and *davinci-codex* [**154**]. To query *cushman-codex* and *davinci-codex* models, we must make API requests using the OpenAI API (free, but rate-limited to 20 requests a minute). While the exact number of parameters is unknown, for both *cushman-codex* and *davinci-codex* models, prior work suggests sizes of 12B and 175B parameters [**40**, **173**, **178**] for *cushman-codex* and *davinci-codex* respectively. Codex is primarily trained on Python [**40**].

To determine if our results generalize to other large language models (LLMs) for code, we evaluate two additional families of models on SStuBs. These models are trained with different procedures and are readily available. CodeGen [**153**] is an auto-regressive transformer model, trained with the next-token prediction objective on a corpus of code and natural language from GitHub. CodeGen is trained on multiple languages, but Python is the primary language. PolyCoder [**239**] is based on GPT-2 architecture and is trained on 250GB of code across 12 programming languages with C, C++, and Java being the primary language. PolyCoder outperforms all other code LLMs in C including Codex. Codex, PolyCoder, and CodeGen represent a diverse set of models all with several model size versions. Testing our the SStuBs hypothesis on Codex, PolyCoder, and CodeGen highlights potential risks of inducing SStuBs while using LLMs. While we cannot say for certain, other LLMs trained on similar data could show similar behavior.

We use the aforementioned models to generate completions by prompting with the code before the SStuB. When models complete the prompt, we can analyze the completion, by matching the known bug, or fix, from the SStuBs dataset. To compare completions to the bug and patch ground truths, we use substring matching (ignoring whitespace and formatting). To verify the accuracy of the results, a survey was conducted where the authors determined if sampled completions (n=401) match the bug, fix, or no match in a manner the automatic evaluation could not capture. For each model family, the best performing model completions were subjected to finer scrutiny; it is important that semantic equivalents are properly counted, such as Codex replacing a constant for the equivalent literal value. The manual survey interface[2] is screen-shot in Figure 7.2. Figure 7.2a shows a completion that is logically incorrect. In Figure 7.2b, Codex actually replaces a constant with its literal value which matches the fix. 401 randomly selected SStuBs are evaluated across each of the three model families to guarantee a confidence level of $> 95\%$. The overwhelming majority of completions are semantically incorrect to the bug or patch, see Figure 7.3.

```
// Fix bugs in the below function.
...
g2d.setColor(tabFillColor);
g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
    boundsWidth, paintBorder.top).getShape());

if (
```

Code Listing 7.1. Prompting Codex with hint.

```
// Fix bugs in the below function

// Buggy Java
paintBorder.top >= 1

//Fixed Java
paintBorder.top > 1
```

[2]The annotation tool is a fork from localturk, a tool designed to emulate Amazon's Mechanical Turk. https://github.com/danvk/localturk

```
...
g2d.setColor(tabFillColor);
g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
    boundsWidth, paintBorder.top).getShape());

if (
```

Code Listing 7.2. Prompting Codex the bug and fix.

Prompting LLMs with Comments

Large language models were found to be surprisingly effective with good prompting [**130**]. "Prompt engineering" is the process of constructing a text prompt, either just a textual prefix and/or set of explicit instructions to induce generation of desired text. Prompt engineering has shown an effect on fixing programs, and generating solutions to coding questions [**106**, **161**, **226**]; see Code Listing 7.1 for an example. Effective prompts may include sample input-output pairs [**173**] or SQL queries [**178**, **214**]; Code Listing 7.2 is an example of bug-fixing comments according to the OpenAI API instructions [**154**]. This form of traditional hard-prompting [**226**], named hard because it uses hard-coded language, requires a *prior viz.* some known information about the input, *e.g.*, it is buggy. In Codex experience surveys [**219**], it appears that prompt engineering is not useful *in situ*, for actual coding. Still, prior work suggests prompt engineering can sometimes be useful [**169**, **173**, **226**].

We hypothesize including comments within a code prompt *might* pre-condition the model better, to produce more relevant and better overall code, in a couple ways: (1) generated code, if relevant, will be easier to understand; previous works show that comments help code comprehension [**213**, **219**, **235**]. (2) comments will help generated code be more maintainable [**75**, **127**]. This led us to use comments to augment the readability and maintainability of the code prior to the SStuB for Codex. Figure 7.4 illustrates the incremental addition of comments starting around the SStuB, and then to surrounding code. We *automatically* generate comments using CodeTrans comment generation model [**59**], trained on the DeepCom dataset [**87**]. The comment generation model [**59**] can use any number of statements

Original Code

```
else  {
    Color tabFillColor = getActiveTabFillIn();
    if (tabFillColor == null) {
        tabFillColor = shape.path.transformY1(shape.to, shape.from);
    }

    g2d.setColor(tabFillColor);
    g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
            boundsWidth, paintBorder.top).getShape());

    if (
```

One Comment

```
else  {
    Color tabFillColor = getActiveTabFillIn();
    if (tabFillColor == null) {
        tabFillColor = shape.path.transformY1(shape.to, shape.from);
    }

    g2d.setColor(tabFillColor);
    g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
            boundsWidth, paintBorder.top).getShape());

1   //If the top of the border is a non-paint border, then the border is painted.
    if (
```

Two Comments

```
else  {
    Color tabFillColor = getActiveTabFillIn();
    if (tabFillColor == null) {
        tabFillColor = shape.path.transformY1(shape.to, shape.from);
    }

2   //This assumes that the tabs are filled with a background color.
    g2d.setColor(tabFillColor);
    g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
            boundsWidth, paintBorder.top).getShape());

1   //If the top of the border is a non-paint border, then the border is painted.
    if (
```

FIGURE 7.4. Adding neural-generated comments, step by step, in the prompt preceeding the SStuB. The first added comment induces greatest improvement in generated code.

to condition its outputs on; we chose to use two statements for each comment, plus the buggy or fixed line, to keep the comment related to the SStuB.

Comments can be generated from either the buggy or fixed version of code. Comments generated from the fixed version of the code should represent the correct logical steps through the single statement bug. On the other hand, comments generated from the buggy version should represent

mostly correct steps with a minor single statement mistake. We test all models using *both versions* of generated comments; the fixed version representing a "non-buggy" comment and the buggy version a "buggy" comment; this facilitates an evaluation of how non-buggy vs. buggy comments influence Codex's ability to avoid/make a single-statement mistake. We suspect that commenting in general, regardless of minor mistakes in natural language descriptions, will still condition Codex to use more reliable, well commented data for generation. Figure 7.4 shows the incremental addition of comments with automatic comment generation tools. We find that the comments *are* beneficial for Codex models, irrespective of the developer's minor misunderstanding as conveyed in comments.

Prompt input, length, and SStuB completion

The code prior to the SStuB, if available, is the conditional input to the model or prompt. The prompt, or code prior to the SStuB, is identical for all 16,899 SStuBs which guarantees the model is not biased towards the bug or the fix. When commenting the SStuB, an automatically generated comment from CodeTrans is placed above the lines used to generate it, properly tabulated, such that the comment appears natural as a developer would place it; see Figure 7.1. The prompt includes the code prior to the bug and up to the maximum allowed amount for each model. The length of any comments reduces the available input we can pass to the LLMs due to the fixed-length token window. The token window for Davinci is 8000 and the other models, Cushman, PolyCoder and CodeGen, are 2048. The token window is ultimately reduced further by the code completion length as the model performs generation in an auto-regressive fashion.

LLM code completions can span several lines. Codex *can* use a stop token, such as the newline character, to terminate completion early. We found that LLMs, like Codex, often add arbitrary newlines and whitespace to completions; thus terminating completion on a newline might otherwise leave unmatched completions. Instead, we ask the LLMs to complete a length of 64 tokens, which is sufficient for almost all SStuBs; SStuBs have a mean length 29 tokens and a median length 25 tokens. After generating a sequence of length 64, the completion is compared to the SStuB ignoring whitespace. The generated sequence *must* match the SStuB completely to count as a bug or fix.

TABLE 7.1. SStuB production rate on off-the-shelf LLMs

(A) LLM completed SStuBs vs. correct code.

Model	Bugs	Patches	No Match	Bug/Patch	Match Rate (%)
PolyCoder 160M	3429	1635	11835	2.10	29.97
PolyCoder 0.4B	3672	1852	11375	1.98	32.69
PolyCoder 2.6B	3924	2096	10879	1.87	35.62
CodeGen 350	3709	1911	11279	1.94	33.26
CodeGen 2B	4102	2756	10041	1.49	40.58
CodeGen 6B	4168	2944	9787	1.42	42.09
CodeGen 16B	4299	3296	9304	1.30	44.94
Cushman 12B	3775	1833	11291	2.06	33.19
Davinci 175B	4452	2267	10180	1.96	39.76

(B) Manually examined model predictions when neither bug or patch *a.k.a.* "No Match" is detected.

Model	**PolyCoder 2B**		**CodeGen 16B**		**Davinci**	
	counts	%	counts	%	counts	%
Incorrect	361	90.02	357	89.03	362	90.27
Patch	19	4.74	28	6.98	19	4.74
Bug	15	3.74	10	2.49	14	3.49
Unsure	6	1.50	6	1.50	6	1.50

In the next section, we present the results from our findings: how often Codex and LLMs produce SStuBs, the number of commits to fix the generated SStuBs, and how annotating code with comments can improve performance on SStuBs.

7.6 Results

7.6.1 SStuB production in LLMs (RQ1)

Table 7.1 is the number of bugs, patches, and non-matching completions from studied LLMs. We use the bug/patch ratio, as a metric to universally compare models as the overall number of successful completions, either a SStuB or a patch, varies between models. Codex, PolyCoder, and CodeGen 350M all produce nearly 2x as many bugs as patches. *Davinci-codex* and *cushman-codex* perform surprisingly poorly given their size, and extensive training. It is plausible that the Codex models recapitulate bugs seen in training data [**106**], however, many of these SStuBs will have been addressed per their inclusion in the ManySStuBs4J corpus two years ago; *viz.* there is a fix.

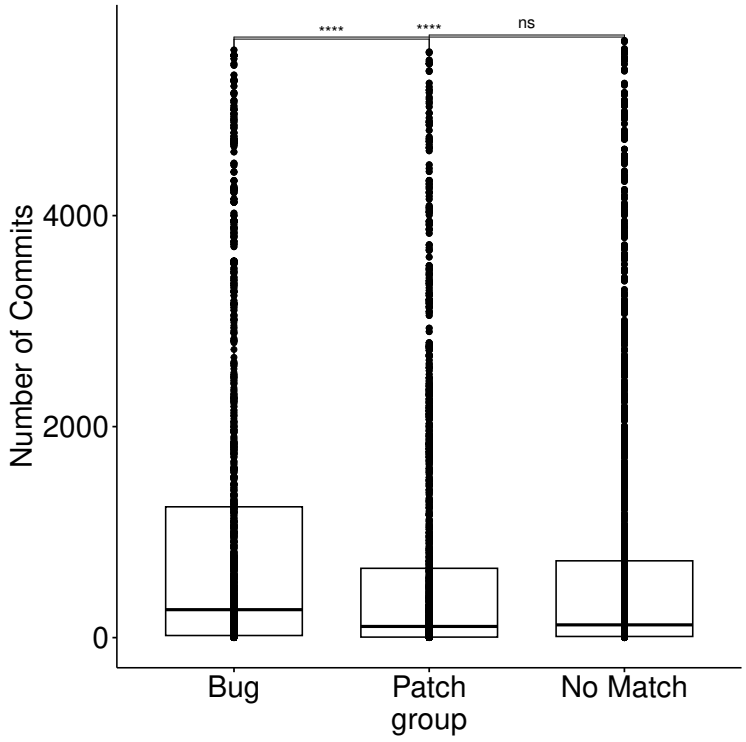


FIGURE 7.5. Developers take more time, measured in commits, to resolve SStuBs that Codex generates. All differences are pairwise statistically significant to $p \leq 0.0001$.

TABLE 7.2. Bug Rate and Patch Rate after adding a comment around the SSTUB.

Model Name	Model Size (Billions)	Bug Change	% Change	Patch Change	% Change	Bug/Patch Ratio Change	% Change	Match Rate Change
PolyCoder 160M	0.16	-427	-12.45	150	9.17	-0.42	-20.42	-1.64
PolyCoder 400M	0.4	-376	-10.24	250	13.50	-0.41	-21.94	-0.75
PolyCoder 2.6B	2.6	-407	-10.37	323	15.41	-0.42	-23.23	-0.50
CodeGen 350M	0.35	-427	-11.51	256	13.40	-0.43	-21.70	-1.01
CodeGen 2.7	2.7	-655	-15.97	129	4.68	-0.29	-18.73	-3.11
CodeGen 6B	6.1	-742	-17.80	-174	-5.91	-0.18	-11.59	-5.42
CodeGen 16B	16.1	-759	-17.66	-81	-2.46	-0.20	-10.24	-4.97
Codex Cushman	12.0	800	21.19	2329	127.06	-0.96	-44.90	18.52
Codex Davinci	175.0	877	19.70	2882	127.13	-0.93	-46.08	22.24

Although Codex produces a high rate of SStuBs, Codex is capable of avoiding 13.41% of SStuBs in the dataset.

RQ1: Codex and LLMs produce twice as many SStuBs as correct code. Codex manages to avoid 13.41% of SStuBs.

TABLE 7.3. LLM completed SStuBs vs correct code with a comment prior.

Model	Bugs	Patches	No Match	Bug\Patch	Match Rate (%)
PolyCoder 160M	3002	1785	12112	1.68	28.33
PolyCoder 0.4B	3296	2102	11501	1.57	31.94
PolyCoder 2.6B	3517	2419	10963	1.45	35.13
CodeGen 350M	3282	2167	11450	1.51	32.24
CodeGen 2B	3447	2885	10567	1.19	37.47
CodeGen 6B	3426	2770	10703	1.24	36.66
CodeGen 16B	3540	3215	10144	1.10	39.97
Cushman 12B	4575	4162	8162	1.10	51.70
Davinci 175B	5329	5149	6421	1.03	62.00

7.6.2 Number of commits to fix LLM produced SStuBs (RQ2)

Figure 7.5 shows the number of commits to fix of SStuBs where Codex generates the original human-created 'Bug', or a 'Patch', or something else ('No Match'). For each of these categories, we examine the version-control history to examine how long (count of commits from introduction to fix, using `git blame`) developers took to fix them. Unfortunately, the number of commits to fix when Codex (re)produces SStuBs (bugs) *is significantly longer* than in other cases. The median number of commits to fix for the bugs, patches, and no match is 265, 106, and 121 commits respectively. Significance of pairwise t-tests was sustained even after the conservative Bonferroni correction. This finding suggests that *when Codex generates SStuBs, these might inherently take human developers longer to fix*! If used widely in open-source code, Codex might spout SStuBs that live longer (in version history) and further pollute future Codex training data. We believe future, detailed investigation in the 4452 matching SStuBs might help improve Codex.

RQ2: The ManySStuBs4J data suggest that in cases where Codex wrongly generates simple, stupid bugs, these may take developers significantly longer to fix than in cases where Codex doesn't.

7.6.3 SStuB regularity (RQ3)

The significant number of commits to fix SStuBs vs. patches (RQ2) motivates a comparison of the "naturalness" of bugs, patches, and no match group of SStuBs. Figure 7.6 shows that there is little difference between the negative log-likelihood of bugs and patches. As expected, the 'no-matches' have a higher negative log-likelihood, since these completions were presumably not seen in the training set. The similar negative log-likelihood of SSTuBs and patches suggests that it may be challenging to fine-tune Codex to detect or avoid SStuBs, since Codex rates them both

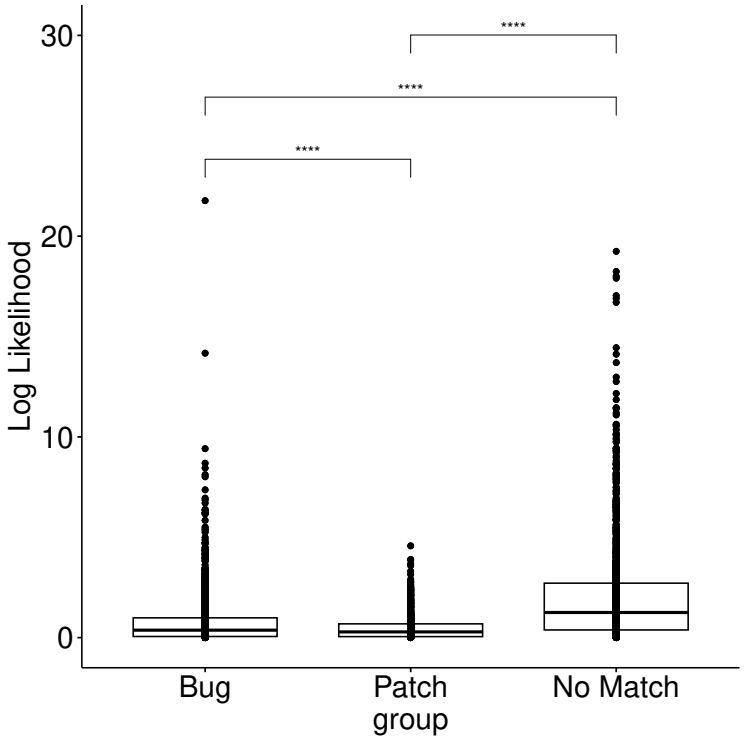


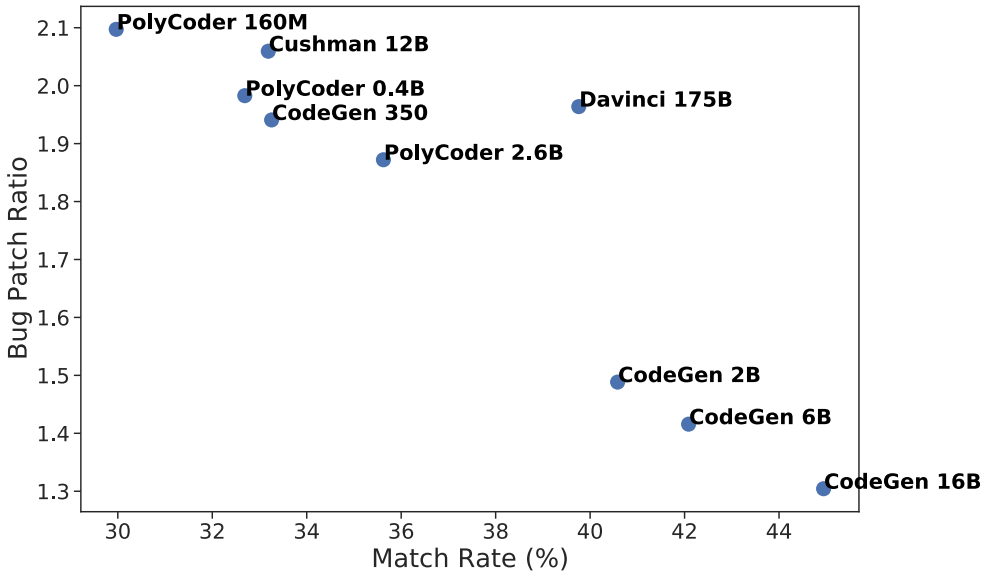
FIGURE 7.6. SStuB bugs and patches from Codex are equally more natural than other code it generates. All differences are pairwise statistically significant to $p \leq 0.0001$.

equally 'natural'; we leave this for future work. However we do study if proper prompt engineering (*e.g.,* with comments), might help matters.

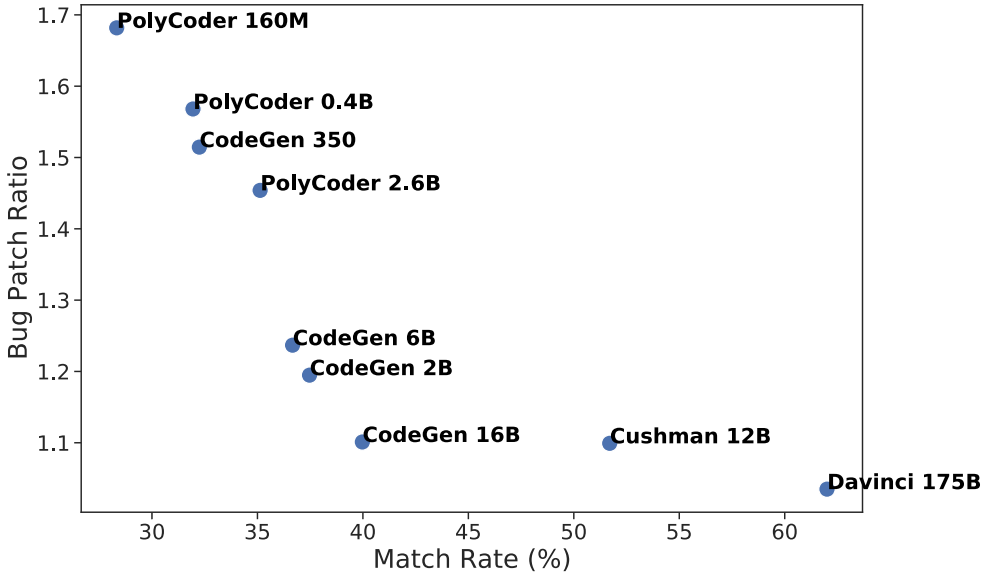
RQ3: Codex log-probabilities indicate that SStuBs (bugs and patches) are regular and natural, thus making detection difficult.

7.6.4 Avoiding SStuBs (RQ4)

We now turn to the question of whether adding natural language comments to the prompt suppresses SStuB generation by Codex. Table 7.3 shows the results of inserting a single comment into the prompt. Table 7.2 captures the rate change in bugs, patches, and bug/patch ratio after adding a comment prior to the SStuB. First, Codex and other LLMs PolyCoder and CodeGen behave differently, namely in the match rate; Codex match rate increases by 19-22% where as other LLMs do not change much. In PolyCoder and CodeGen, the number of bugs decreases from 10-18%



(a) Codex models (Cushman & Davinci), without comments, perform not well on bug/patch ratio and the match rate.



(b) Commenting code helps Codex achive best performance across SStuBs while improving the match rate;

FIGURE 7.7. Prompting with comments should be used to both avoid SStuBs

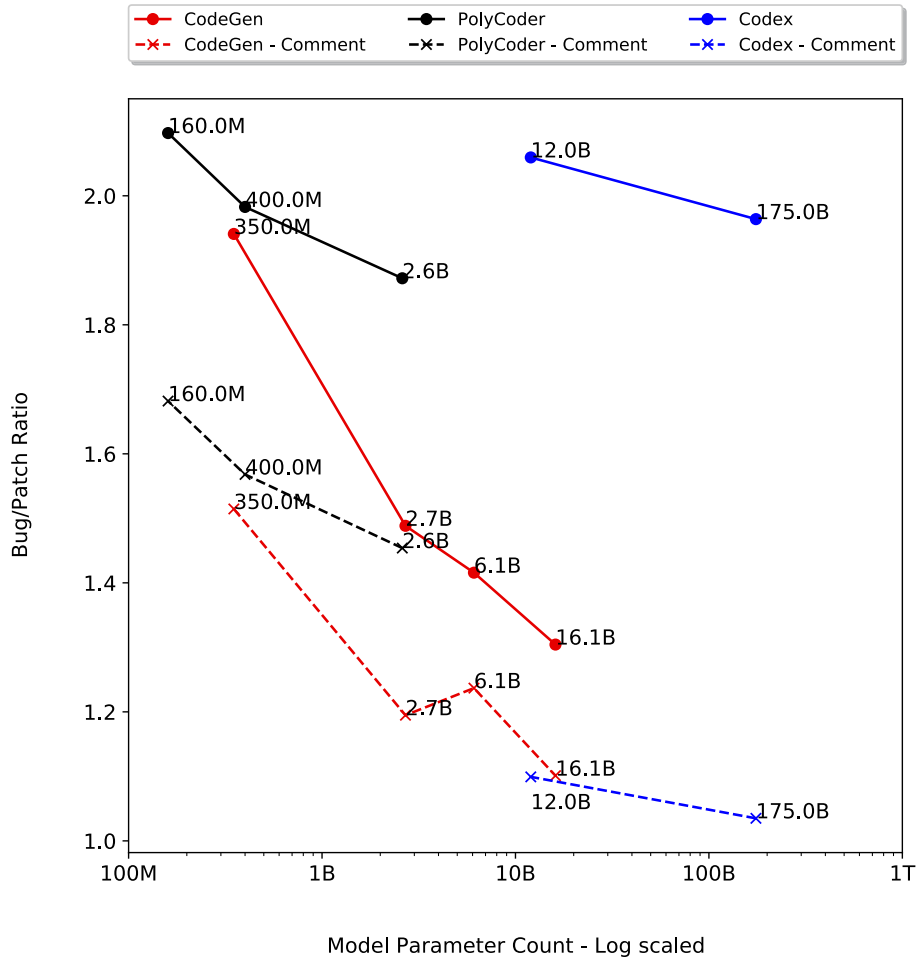


FIGURE 7.8. Bug/Patch Ratio vs. Parameter Count. Three model families at various sizes. The largest difference is the addition of 1 comment prior to the SStuB.

and the patch rate increases. The bug/patch ratio in all models improves! Codex Cushman and Davinci generate 20% more bugs, but then produce *127%* more patches. The bug/patch ratio is cut by almost half. This suggests that developers get better results by commenting code while using Codex: this amounts to about 3000 more patches (5149 vs. 2267).

Figure 7.7 are scatter plots showing the relationship between match rate and bug/patch ratio. Ideally, models will have a high match rate (less unknown cases) and a low bug/patch ratio. Per Figure 7.7a, Codex models Cushman and Davinci were not competitive to other off the shelf models. After adding comments to the SStuB prone code, Figure 7.7b, all models perform better and Codex performs *much* better than the next best model CodeGen 16B.

TABLE 7.4. Bug Rate and Patch Rate after adding erroneous comments around SSTUB.

Model Name	Model Size (Billions)	Bug Change	% Change	Patch Change	% Change	Bug/Patch Ratio Change	% Change	Match Rate Change
PolyCoder 160M	0.16	59	1.72	-237	-14.50	0.40	18.97	-1.05
PolyCoder 400M	0.4	171	4.66	-132	-7.13	0.25	12.69	0.23
PolyCoder 2.6B	2.6	234	5.96	-145	-6.92	0.26	13.84	0.53
CodeGen 350M	0.35	110	2.97	-178	-9.31	0.26	13.54	-0.40
CodeGen 2.7	2.7	-78	-1.90	-420	-15.24	0.23	15.74	-2.95
CodeGen 6B	6.1	-264	-6.33	-603	-20.48	0.25	17.79	-5.13
CodeGen 16B	16.1	-95	-2.21	-558	-16.93	0.23	17.72	-3.86
Codex Cushman	12.0	1511	40.03	1135	61.92	-0.28	-13.52	15.66
Codex Davinci	175.0	1620	36.39	1613	71.15	-0.40	-20.31	19.13

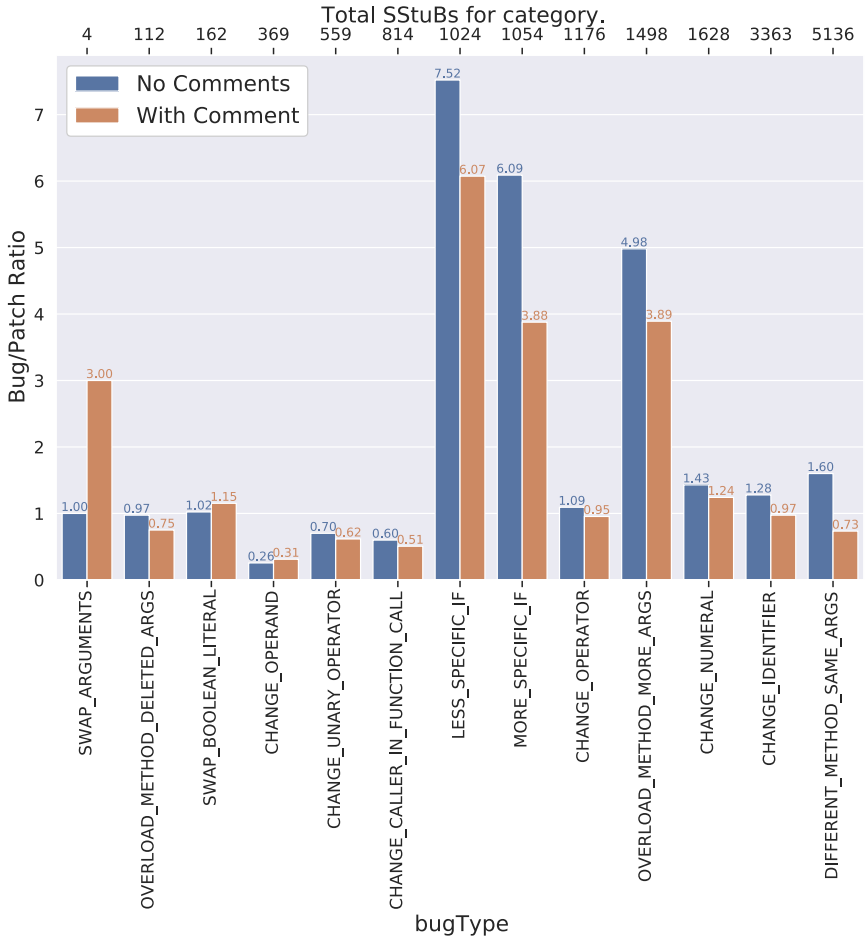


FIGURE 7.9. Comment effect on bug/patch ratio. Lower is better. Top axis is total SStuB count, little significance placed on bug categories with less than 100 samples.

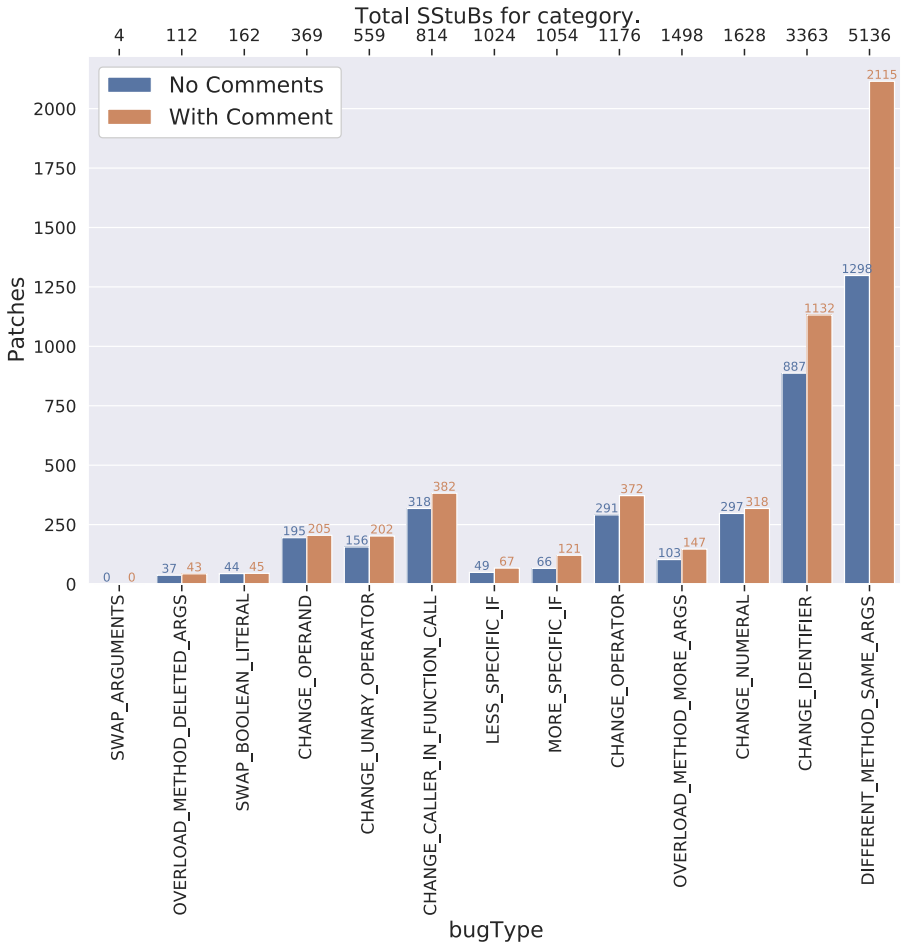


FIGURE 7.10. Comment effect on patches. Higher is better. Top axis is total SStuB count, little significance placed on bug categories with less than 100 samples.

TABLE 7.5. Codex (Davinci) performance across various prompting techniques.

Prompt	Bugs	Patches	No Match	Bug/Patch	Match Rate (%)
Hint	6228	4814	5857	1.29	65.34
Bug & Fix	6565	6055	4279	1.08	74.68
Comment	5329	5149	6421	1.03	62

Figure 7.8 shows the effect of adding comments to the bug/patch ratio with model parameter counts. Adding a comment in the prompt helps more than increasing parameter counts! A 160M parameter PolyCoder with a comment outperforms (by 14% improvement in bug/patch ratio) both the Codex Cushman 12B and Davinci 175B, without comments.

Finally, Figure 7.9 shows the bug/patch ratio and Figure 7.10 shows the number of patches by bug type (given in the ManySStuBs4J dataset) ranked left to right by the frequency in the dataset. The largest improvements in bug/patch ratio in a sufficient set of samples are `LESS_SPECIFIC_IF`, `MORE_SPECIFIC_IF`, `OVERLOAD_METHOD_MORE_ARGS`, and `DIFFERENT_METHOD_SAME_ARGS`. The bug types that gained the most patches are `CHANGE_IDENTIFIER` and `DIFFERENT_METHOD_SAME_ARGS`. `SWAP_ARGUMENTS` got worse, but only consists of 4 total examples, and it is hard to make any conclusions from this.

7.6.5 Commenting vs Traditional Prompting (RQ4 cont.)

Prompting LLMs both in NLP and SE applications often come in the form of instructions prior to the input. Previous empirical studies of Codex [**152**, **160**, **161**, **162**, **166**, **173**, **189**, **204**, **226**, **243**] prompt Codex with instructions, input-output pairs, or examples prior to the input that the model is conditioning the output on. We compare in a similar fashion to Prenner et al. [**173**] by providing Codex with hints (Code Listing 7.1), and SStuBs (Code Listing 7.2). We remind the reader that both prompting techniques found commonly in previous works, require knowing something about the snippet of code that will be generated; *e.g.*, a location in the code that has a SStuB (hint) and what the SStuB is and how to fix it (bug and fix).

While both approaches improve the bug/patch ratio and match rate, Table 7.5, neither does as well as adding natural language comments. Furthermore the number of bugs greatly increases in the traditional prompting techniques.

RQ4: Commenting code can lead to less generated SStuBs and more generated patches. Codex models improve the most from code comments.

7.6.6 What if we insert a 'buggy' comment? (RQ5)

Finally, we try inserting a *buggy* comment in the prompt, to mimic the developers description of the SStuB. The natural language comment for buggy code is created by conditioning CodeTrans on the SStuB (bug) rather than correct code (patch). Table 7.4 shows the difference in bugs, patches, bug/patch ratio, and match rate with a buggy comment before the SStuB. Surprisingly, Codex is robust and still improves bug/patch ratio over the "no comments" case. This suggests that the mere presence of relevant comments in the prompt sufficiently pushes the model to produce better code.

It's also interesting to note that the lower capacity models (and also the 16B parameter CodeGen) tend to be misled by the 'buggy' comments, whereas the larger capacity, well-trained Codex models are not.

RQ5: Misleading comments still condition Codex to produce less SStuBs. Commenting appears beneficial irrespective of the developer's understanding of the SStuB.

7.7 Discussion

7.7.1 Implications of Findings

Implications of Codex Producing SStuBs

The good news: Our study suggests that LLMs like Codex do help avoid a significant number of SStuBs in our dataset, out-of-the-box, and even more with non-buggy comments! But they do produce simple, stupid bugs. Even Codex produces up to 2x more SStuBs as patches, if used directly, nor does increasing model size (see Table I) necessarily help.

To better understand why Codex still produce SStuBs, one must further examine the training data (sadly, not available for many LLMs); we hope training data will become more available. Previous work [**16**, **160**, **204**] studying Codex-generated vulnerabilities blames Codex's language modeling roots, which push it to produce the most "likely completion (for a given prompt) based on the encountered samples during training". Also, SStuBs are capable of lasting for long periods of time [**151**] and are not detected by continuous integration [**119**] or static analysis [**88**, **105**, **151**] which explains why Codex recapitulates them (from it's training data). Simple, stupid bugs are likely regularly injected by devs; training Codex without SStuBs would be challenging, given training data is drawn from 54 million repositories [**40**]. The effect of Codex produced SStuBs is significant, and troubling. The number of commits to fix Codex produced SStuBs versus the avoided SStuBs is significant, taking more than twice as long to fix. Still we should bear in mind that Codex avoids 2,267 bugs on its own or 13.41% of the dataset, indicating an AI paired programmer is helpful in avoiding SStuBs too.

Avoiding SStuBs with LLM

Codex and other LLMs respond unpredictably to prompts, and developers often struggle to get LLMs to generate desired code [**219**]. Studies suggest that breaking coding tasks into manageable sub-problems helps [**214**, **219**]. NLP tasks work similarly; chain-of-thought [**231**] and reasoning step-by-step [**114**] improve problem-solving rate. Commenting is ideally a form of step-by-step reasoning, explaining high level steps, or clarifying confusing code. The generated comments we used appear to be high-level descriptions and not deep technical commentary of computed values and algorithmic mechanisms. Our work suggests minimal effort techniques, like automatically generated documentation, may help avoid SStuBs when using an AI programming assistant like Copilot. Not only can comments be automatically generated as documentation, but comments can be used directly as a prompt for Codex. To the best of our understanding, Codex might condition the generated code on a smaller search-space of non-buggy solutions, thus helping the developer avoid introducing SStuBs.

Lastly, comments can sometimes be used to check the implementation consistency given the desired functionality [**232**]. Future work could examine if SStuBs can be detected with the same tools given a set of generated comments.

In our experiments the placement of comments is uniform, and further work should be done to determine best possible comment placement in a density that is adequate and not excessive; automatic methods exist [**90**]. Excessive commenting is typically symptomatic of a lack of understanding of the code and a "code smell" [**20**].

Maintaining AI Generated Code

Language models for code like Codex, PolyCoder, CodeGen, and others [**3**, **62**, **72**, **102**, **228**] will become bigger, and better at code completion [**79**]. In a world where AI programming assistants learn from data at scale [**118**], it is hard to say how much of novel programming projects or code reuse [**7**] will guide such tools. Fundamentally, there is a need for improved readability and comprehensibility in AI-generated code [**219**]. Code comments can improve comprehensibility of inserted code, especially of more difficult statements. Code that is more readable and understandable is much more maintainable. Our work suggests that comments help avoid SStuBs, in addition to the traditional role of improving code readability. Prior work indicates that nearly half of SStuBs

are fixed by another developer and with greater effort [**248**]; note that any code (buggy or not) generated by a language model may be unfamiliar to a developer.

Lastly, the preliminary successes on SStuBs warrants further research in comment generation with AI programming assistants. Comment generation models like DeepCom [**87**] and CodeTrans [**59**] are fully automated and could function in a variety of roles for AI programming assistants. For example, comment generation models could serve as automatic code commenting for Codex completions, be used to check for implementation consistency and accuracy, and improve the quality of training data for Codex to name a few. Our approach of using comments with Codex should be reexamined under a variety of applications including program repair and defect prediction. This is an interesting future direction.

7.8 Threats to Validity

7.8.1 Internal Validity

ManySStuBs4J

We assume the samples in this dataset are mostly actual bugs. The authors report that changes related to refactoring, are removed, but some non-bug-fixing commits may remain.

Manual Inspection

We use automated matching to determine whether the models produce a known SStuB or patch. However, it is possible the automatic evaluation misses semantically equivalent but syntactically different bugs or patches. This could potentially hide the true number of bugs and patches. To reduce this threat to our results, we have three independent raters (the authors) inspect random samples from *davinci-codex* completions (for the cases with no-comments, and the cases with non-buggy comments) that matched neither bug nor fix (we call this unmatched subset "dark matter").

With fair agreement, Fleiss Kappa 0.40, the independent raters found the vast majority (over 80%) to be inappropriate code completions (neither bug nor fix — just wrong), and sparsely little bugs or patches for the no-comment case with Davinci Table 7.1a. With moderate agreement, Fleiss Kappa 0.6, the independent raters found a smaller majority (about 70%) of inappropriate code generations in the "dark matter sample" for the non-buggy comment case, Table 7.3. The

independent ratings all found that, even in the "dark matter" sample, adding comments in the prompt resulted in substantial increase in patches. While we acknowledge that Codex non-match completions pose a threat to our findings, our sampled examination of this "dark matter" in both settings (with and without comments) suggests that adding comments does help LLMs avoid the generation of SStuBs.

Data Leakage from LLM Training

We cannot independently verify that the ManySStuBs4J dataset is excluded in the training of the models since none of the models' training data is published. "Data leakage" is traditionally a concern when evaluating the performance of language models as data seen during training might artificially inflate results. In our case, data leakage will bias the model towards an outcome either the bug or the fix. We examined the latest fix date for the studied SStuBs and found 100% of the SStuBs were fixed by February 2019 *and* the earliest data collected for training is 2020 (*cushman*) and 2021 (*davinci*). If there is data leakage from ManySStuBs4J, we postulate the models would most likely see the *fixed* version of the code, but intriguingly, the models still produce 2x more SStuBs!

Reproductions of Generations

Depending on hyper-parameters, Codex models are nondeterministic in their text generation (for the same prompt). We sampled the top-1 completion for each ManySStuBs4J sample across all models (Codex, PolyCoder, and CodeGen), with and without comments.

7.8.2 External Validity

ManySStuBs4J

Generalizability is subject to the limits of ManySStuBs4J. The dataset consists of Java single statement bugs; our results may not generalize to other languages, or less simple bugs. PySStuBs [**100**] and TSSB-3M [**185**] are larger, and cite different SStuB patterns. The ManySStuBs4J dataset is the appropriate size given our constraints on available compute, and also API access to Codex.

We were limited by OpenAI's rate ceiling of 20 requests per minute; on local hardware, the largest model, CodeGen 16B takes over a day for a run with a single prompt on ManySStuBs4J.

Models at Scale

Language models are getting ever larger. Results may vary with the next generation of models.

7.9 Conclusion

Most importantly, we find that Codex and other large language models *significantly help avoid human-produced simple, stupid bugs*! In our best case, around 30% (5149) SStuBs were actually patched (avoided) by Codex Davinci. Still, we find that large language models might produce many more SStuBs than patches. *First*, Codex and PolyCoder produce nearly twice as many SStuBs as fixes. *Second*, and very worryingly, Codex generated SStuBs apparently took significantly longer to resolve and that the SStuBs appear to be as natural as the correct statements. Our results show that AI pair programming can introduce SStuBs, and the manner in which developers are known to use such tools is not conducive to avoiding SStuBs. Still, though the models were somewhat SStuB prone, even out-of-the-box LLMs could have avoided as many as 2,300 SStuBs had developers used code completion instead of writing them.

Since the simple, stupid bugs are quite obvious after detection, we explore the idea of guiding AI assistants by adding comments describing high-level functionality. The proposed strategy of communicating functional intent to Codex with comments improved the bug/patch ratio substantially. Finally, we explore minor misunderstandings in the intended functionality by using buggy comments and find that Codex may not require strict correctness in comments to avoid SStuBs. Our results suggest that good commenting practices, even in an automatic setting, can help other developers and Codex, especially in an era where AI generated code is regularly committed.

Overall, our findings are somewhat promising, LLMs may help avoid *at least some* simple, stupid bugs!

Chapter 8

Reflections, Future Work, and Concluding Thoughts

8.1 Reflection

In the last ten years, machine learning has permeated practically every aspect of modern life. The focus on AI in software engineering is more recent, but with increasing attention in mainstream culture. Forbes is one of many mainstream media outlets that sensationalize the capabilities of language models with articles titled *How ChatGPT and Natural Language Technology Might Affect Your Job If You Are A Computer Programmer* [**143**]. When I began the Ph.D., I never expected the field to be so hot, for better or for worse. My biggest fear is not the replacing of developers entirely; far from it. I am optimistic that the technologies we develop today are *helpful* in solving problems we could never tackle in the first place. Opportunities arise for models to guide developers into secure coding and better coding practices. Models that are trained with a better understanding of code, e.g. how static code translates to traces, can alert developers to potential edge cases or extremely improbable situations with catastrophic consequences. Take the Ariane 5 disaster, where the horizontal bias, represented as a 64 bit float, was typecasted to a 16 bit value causing the value to overflow; ultimately flipping the rocket 90 degrees, ripping it into pieces due to aerodynamic forces. Had machine learning been where we are today for software development, it is conceivable

that the engineers would have detected the critical runtime bug. This is not a far fetch assertion, as pretrained transformers and LLMs are widely integrated into the Software Development Life Cycle (SDLC): requirement analysis [**109**, **138**, **188**], design [**32**], construction [**26**, **210**], testing [**216**, **229**], analysis [**14**, **56**, **77**], and maintenance [**15**, **92**, **215**] to name a few.

8.1.1 Software Practices

AI4SE will birth many technologies that will solve decade old problems. Developers are traditionally plagued by monotonous tasks like sifting through log files to locate a bug when often the bug is difficult or impossible to locate without help; log files often contain redundant messages or content that is not particularly helpful. This can result in developers making a larger mess out of the code, e.g. adding an obscene amount of debug and error messages. One effective AI solution can possibly solve downstream code quality problems. Anecdotally, developers hard-code error values to better navigate log files and this technique is unfortunately common, but creates obfuscation for unfamiliar developers; I was a witness to this in recent memory at a large technology company! There is a moat of opportunity at improving general practices of developers. The best products will likely be localized in large tech company offerings via cloud development. However, individual tech innovators and startups have more power than ever to develop novel solutions on age-old software engineering challenges.

8.1.2 Software Processes

AI4SE solutions will improve the process of developing software too. When runtime failures are discovered it can be difficult to replicate the environment. AI solutions trained over probabilistic input and outputs can eliminate testing many benign values. Even the build and test aspect of code pipelines will be improved; QA testing will become more comprehensive and localizing problemsome commits and lines of code (LOCs) will be more efficient. Rather than checking out commits to test for runtime problems, imagine using a model to localize the behavior automatically. I am excited by easy to use, model driven software visualizations including visualizing: the distributions of variable values, the LOCs that need to be refactored, the potential attack vectors, and more. These models are amendable to practically any aspect of learning, given an objective function that aligns with the

problem; as seen in Chapter 4. Not only can machine learning benefit developers and the process of developing software, but also the users of products made with AI.

8.1.3 Software Users

Today, in an AI age, software users are pseudo-developers. The clicks, scrolls, and actions are reward signals in themselves. A UI developed with customer feedback through LLMs has the potential to create more engaging, useful software. This of course has several positive and negative implications. Software that is easier to use is inherently more useful, however, software features like infinite scrolling might drive up engagement with consequential societal implications. We live in an age of constant entertainment with the development of features like shorts and infinite scrolling. Research scientists at social media companies are responsible for understanding these consequences when developing such features or optimizing them to be more addictive. To generalize broadly, these models must be aligned with our values to be viable solutions.

8.1.4 Software Jobs

A recent paper from OpenAI [**60**] cites that GPTs (generative pretrained transformers) are GPTs (general purpose technologies). General purpose technologies are characterized by "widespread proliferation, continuous improvement, and the generation of complementary innovations". In AI4SE, GPTs are proliferating, improving, and will create many new software products. However, the current iteration of GPTs are not well aligned with developers; maybe RLHF will improve that eventually. For example, the transformer networks are constantly being modified to align to developer (e.g. secure code) and organizational (e.g. requirement understanding) values, and to improve code understanding (e.g. unnatural but valid coding practices) through techniques like prefix tuning [**76**], inductive biases integration [**72**], and better semantic understanding [**37**]. These adjustments require real engineers and developers for the foreseeable future to adjust off the shelf LLMs, or develop alternative solutions when LLMs perform poorly in forums such as closed-source software environments.

Revisiting controversial headlines like New York Times's *Tinkering With ChatGPT, Workers Wonder: Will This Take My Job?* fails to acknowledge the alignment problem currently plagued by LLMs; misalignment of software, developer, and societal values and the understandings therein.

This alignment problem exists in the domain LLMs are trained in, the values that they generate at inference, and propagate into the applications driving our society. Concluding on this thought, I think of many developers who have reached out concerned about the risk of the GPTs (ChatGPT, Copilot, etc). No, these models are not comprehensive problem solvers, nor do they understand the processes and values associated with software development. Software developers will be expected to work with generative models side-by-side, improve the workflow with them, and improve their capabilities over the next 10 years. This vision is guiding my research for the foreseeable future and hopefully downplays the apprehension developers are facing.

8.2 Worthwhile Research Pursuits

8.2.1 Increasing the Reach of LLMs

Off the shelf LLMs are general purpose generative tools that are widely trained on open-source data. Published models like CodeGen, CodeParrot, PolyCoder, even Codex do not contextualize their outputs on project specifics, e.g., existing function and variable nomenclature out of its current scope. Currently IDE tools like Copilot perform *some* dark magic to bias the model towards outputs that are somewhat contextualized, however, syntactically correct outputs are nowhere guaranteed. While approaches like prompting are effective, there is still a lack of awareness for the *rules* a project is constrained by; this can include language version, availability of different language constructs (type inference in Java), existing security and coding standards, and more. Finding ways to include, more broadly, developer and organizational constraints will greatly improve the usability of these models.

Along a similar thread, LLMs have an opportunity to suggest improvements to existing code: where and why code should be refactored, and even what code should go there. We often associate code completion as a "AI co-programmer" [**249**] but code completion is not truly a co-programmer. Human co-programmers recognize when functions should be split up, new classes created, and better use of language constructs such as polymorphism to improve the overall organization and performance of the code base. An unrealized gain of true "AI co-programming" is the prioritization of technical debt; models will recognize subpar code and will force developers to address it, especially, if future completions depend on significant refactorings. It is in the interest of the developer to

abide by the AI recommendations, such that future recommendations are not contributing to an increased technical debt. Finally, a dialogue with the developer on why completion performance is dependent on addressing existing technical debt is a great motivator.

8.2.2 Runtime Information

Many software engineering tools rely on pre-trained code embeddings. The code embeddings stem from on-going paradigm shift with the rise of language models defined by billions of parameters, encoded from self-supervision on open source data at scale. Large scale pre-trained embeddings have dramatically improved natural language processing task performance and many new models continue to be applied to code. Much of the adoption of these models do not account for the many differences between code and text. Some works account for this by adapting the structure of the model or training signal to account for code specific features like data/control flow; some models even encode the AST for the code completion. However most works ignore an obvious difference of code and text: run time information. Code is executable and LOC often repeat with different values. A new research direction based on runtime information understanding can condition static code embeddings with much more information! Finding ways to augment static embeddings with information related to the runtime environment, e.g. traces, will give developers power to predict how code will run, and the problems therein, prior to the introduction in the code base.

It is widely recognized that existing language models do not understand the specifics of how code executes [**17**]. Austin et al. states, *"even our largest models are generally unable to predict the output of a program given a particular input, whether few-shot or with fine-tuning"* and I have verified this more broadly on open-source models: CodeBERT, GraphCodeBERT, CodeBERTa, etc. Trace analysis is a promising direction although somewhat inefficient due to the sheer scale required. I am optimistic that larger language models will require less code coverage to better understand runtime information. Any pursuit on the integration of this untapped potential can benefit code understanding broadly across all applications using ML-based methods.

8.2.3 Prompt and Prefix Tuning

Prompt and prefix tuning are promising approaches. Prefix tuning can adjust .1% of model parameters [**120**, **124**] and prompt tuning only adjust .01% of model parameters [**120**]. These

approaches can distill relevant information for specific tasks. It would be interesting to see how much project specific information can be distilled by such methods to improve performance of off the shelf LLMs. Prompt and prefix tuning has already been shown to be effective for generating secure code [**76**]. I am intrigued by future applications of prompt and prefix tuning.

8.2.4 Hybrid ML and SE approaches

Software engineering is defined by two fundamental channels [**36**]. Engineers communicate with a natural channel which contributes program semantics. For example, developers can use the natural channel to convey code meaning, developer goals, code functionality, explanations, code requirements, and more. The formal channel is defined by language syntax and algorithmic execution of code. Often software solutions reside in either the algorithmic channel, e.g. static analysis and program verification, or the natural channel, e.g. code completion. Programs generated with code completion tools lack the rigor of formal correctness and often hallucinate with the inclusion of fictitious variables and functions [**17**, **40**]. A promising approach is a hybrid between the formal and natural channels. Take HiTyper [**164**], a hybrid type inference approach where the expressiveness of deep learning meets the formal checking of static type inference. This approach uses the formal correctness of type dependency graphs with the expressiveness of ML based type inference. Uniquely, this type inference model produces type consistent code. Hybrid approaches are more prevalent in code completion settings today [**40**, **41**, **139**].

8.2.5 Code Compression

Code is often repeated and a majority of machine learning based models learn from duplicated (sub)sequences of source code. The snippets of source code passed into sequence based models often have long dependencies that are not available to the model due to a limited context window. Context windows in LLMs are getting larger with sufficient hardware in data centers, however, smaller localized modeling is greatly limited by the amount of relevant information that can be included to the prompt. I believe there are properties unique to code that can be highly compressed; the removal of tokens like brackets and braces seem to have little impact in code intelligence task performance. Inspired by several works on transformer position encoding [**110**, **163**, **199**], I believe

that efficient representations will allow more relevant tokens in finite context windows for a wide variety of models.

8.2.6 Reinforcement learning with Human Feedback (RLHF)

The best "supervised signal" is real human interaction with LLMs. Feedback in the form of ranking [**156**], natural language [**39**], and dialogue [**153**, **237**] effectively capture real time feedback from real users; no need for complex (and ineffective) simulators. As users accept and reject completions from tools like GitHubs' Copilot, a ground truth of human feedback is accumulated. This dataset of feedback can be optimized in traditional settings or with reinforcement learning. The widespread paradigm shift to RLHF from traditional approaches, like increasing model parameters, is promising for institutions interested in value alignment. In our current conversation of making models more *developer aligned*, RLHF is arguably the most promising prospect.

8.3 Conclusion

This thesis presents several approaches that build better models and representations of code for developers. The gradual improvement of representations is achieved by task-specific design of the machine learning methods, the code representation it attempts to understand, and mining of relevant, readily available data. To model specific nuances of code, such as user-defined types, we found that the additional mining of necessary declarations in conjunction with learning associations between these declarations was sufficient. Augmenting the existing representations between class declaration statements and the downstream use of that declaration is an alignment problem between human understanding and machine's interpretation of code.

A popular theme in natural language processing is that the path to more comprehensive representations are models with more parameters [**200**] and larger corpora. This path is promising overall and runs parallel with improving the networks' ability to represent complete program semantics. Such networks [**9**, **80**, **163**] unfortunately still require subject matter experts to determine meaningful hyper-edges and translate the data accordingly [**86**]. Indeed, representing entire programs in a model at scale will result in incredibly useful embeddings and general purpose models, but

still require task specific alignments; for example the user-defined types problem. This should not discourage the pursuit of better general code representations.

A major takeaway from this thesis is that we (model architects) can find meaningful representations of code by combining formal and natural channels with powerful models. In Chapter 3, we found that existing modalities, namely partial token sequences, was sufficient to learning common types when supplemented with giga-scale self-supervision. The improvement in top-1 accuracy across the most frequent types was substantial (89.51% vs. 66.9%) indicating that existing graph neural networks (GNNs) were sub-par for frequently used types.

TypeBert demonstrated a trade-off between engineering sophisticated, graphical inductive biases, *vs.* general purpose representations that may only learn sequential patterns of code. This simple intuition of code is apparently sufficient to allow TypeBert to make relevant predictions across all type locations; albeit not guaranteeing formal correctness. The GNN approach often couldn't make simple type guesses with incomplete projects; but guarantees a higher degree of correctness. The approximate nature of partial contexts lacks formal correctness, but in real-world applications it may prove to be more practical. In following the theme of the thesis, we found that the representations TypeBert used were ill-conditioned for a fundamental characteristic of code; the neologism of user-defined types. With the rich contextual encoding of source code across giga-token corpora, we postulated from the outset that user-defined type declarations could be embedded and aligned. By (re)formulating a machine construct, self-supervised embeddings, in a fashion in which humans interpreted user-defined types, we were able to perform large scale alignment as almost all projects have user-defined type declarations.

In Chapter 4 we take the ideas presented in Chapter 3 further and explore a core typing problem. *Can we jointly learn traditional common types and the more expressive user-declared types with partial contexts?* We explore this goal by supplementing our existing type inference dataset with an user-declared type dataset and jointly train DiverseTyper with two losses: cross-entropy and triplet loss. We attempt to preserve the prediction accuracy that TypeBert has while gradually aligning the type declaration to occurrence of its use. We argue that our approach to user-defined type inference is not a *task* specific solution, but an adaptation to a particular dynamic of code; a *modeling* solution. This approach lead to a 72.85 percent increase in user-defined type inference

(30.16% absolute) and an absolute overall improvement of 8.59% increase over TypeBert. In terms of practical usefulness, DiverseTyper [**93**] was correct almost 55% of the time on just the novel types. Beyond statistics, DiverseTyper colored an important picture; translating program semantics as humans interpret them for machines leads to big performance gains. Qualitatively, DiverseTyper seemed to always get the type correct, mainly attributable to its comprehensive dataset and its ability to localize declarations. In summary, by removing the representational ambiguity between two seemingly distinct tokens to the machine learning model, we explicitly encode type usage conventions and create a more meaningful representation of code for type inference. The general principle of optimizing the way machine learners see code will be a focus of many future works.

Chapter 5 realizes a dataset, ManyTypes4TypeScript, that facilitates a goal of standardized evaluation for type inference [**94**]. The inclusion of the dataset and evaluation scripts on Microsoft's CodeXGLUE will encourage the development of new type inference models and model comparisons. I also introduced a suite of new models that use data and control flow for type inference.

Chapter 6 is the actualization of a framework that can be used for virtually any type inference model [**225**]. With the ManyTypes4TypeScript dataset in Chapter 5, creating new models is easier than ever. However, most models stay in GitHub repositories or Huggingface model cards. The goal of FlexType is to bring the model into a visual studio environment for rapid use by developers. I am pleased to contribute this tool to the open-source community.

In our final piece, *Large Language Models and Simple, Stupid Bugs*, Chapter 7, we study popular code completion tools on their performance on SStuBs (Simple, Stupid Bugs). We were alarmed that the Copilot model, Codex, generates ≈2x more bugs than correct code! I further studied the implications of LLM completed SStuBs and find they are equally natural to their correct counterpart, thus making SStuBs very difficult to spot; this confirms existing work in the field [**105**]. I also studied how long SStuBs last in projects and find that LLM generated bugs last longer than the SStuBs Codex avoided. In an age of AI "copilots", this is a toxic combination as more AI generated code will exist and be difficult to localize and fix. Furthermore, AI generated code is often ill understood by their users [**219**] and may introduce simple, stupid bugs at an increasing rate [**92**]. As future AI models are trained on open source code, this can ultimately lead to a negative feedback

loop. Finally, we present how just commenting code, either automatically or manually, can condition LLMs to avoid SStuBs.

8.3.1 Final Remarks

Holistically, we hope that this body of work demonstrates the value of finding *developer aligned* representations of code. The pursuit of finding meaningful representations of code that reflect developer, organizational, and societal values is ambiguous, however, in my humble opinion, a very enterprising ambition. Through the techniques discussed, it should be clear that there is still a plethora of software engineering data, much of it still untapped and unrealized, that is widely receptive to innovative learning methodologies.

Bibliography

[1] *Mypy*.

[2] *Pytype*.

[3] W. U. Ahmad et al., *Unified pre-training for program understanding and generation*, 2021.

[4] T. Ahmed and P. Devanbu, *Multilingual training for software engineering*, arXiv preprint arXiv:2112.02043, (2021).

[5] ———, *Few-shot training llms for project-specific code-summarization*, in 37th IEEE/ACM International Conference on Automated Software Engineering, ASE22, New York, NY, USA, 2022, Association for Computing Machinery.

[6] 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*, arXiv preprint arXiv:2301.03797, (2023).

[7] 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.

[8] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, *A survey of machine learning for big code and naturalness*, ACM Computing Surveys (CSUR), 51 (2018), pp. 1–37.

[9] M. Allamanis, M. Brockschmidt, and M. Khademi, *Learning to represent programs with graphs*, 2018.

[10] M. Allamanis et al., *Learning to represent programs with graphs*, in International Conference on Learning Representations, 2018.

[11] ———, *Typilus: neural type hints*, arXiv preprint arXiv:2004.10657, (2020).

[12] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, *code2vec: Learning distributed representations of code*, 2018.

[13] J.-h. An, A. Chaudhuri, J. S. Foster, and M. Hicks, *Dynamic inference of static types for ruby*, ACM SIGPLAN Notices, 46 (2011), pp. 459–472.

[14] C. Ancourt, F. Coelho, and F. Irigoin, *A modular static analysis approach to affine loop invariants detection*, Electronic Notes in Theoretical Computer Science, 267 (2010), pp. 3–16.

[15] M. Aniche, E. Maziero, R. Durelli, and V. Durelli, *The effectiveness of supervised machine learning algorithms in predicting software refactoring*, IEEE Transactions on Software Engineering, (2020).

[16] O. Asare, M. Nagappan, and N. Asokan, *Is github's copilot as bad as humans at introducing vulnerabilities in code?*, arXiv preprint arXiv:2204.04741, (2022).

[17] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., *Program synthesis with large language models*, arXiv preprint arXiv:2108.07732, (2021).

[18] D. Bahdanau, K. Cho, and Y. Bengio, *Neural machine translation by jointly learning to align and translate*, 2016.

[19] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, et al., *Training a helpful and harmless assistant with reinforcement learning from human feedback*, arXiv preprint arXiv:2204.05862, (2022).

[20] P. Becker, M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 1999.

[21] I. Beltagy, M. E. Peters, and A. Cohan, *Longformer: The long-document transformer*, arXiv preprint arXiv:2004.05150, (2020).

[22] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, *On the dangers of stochastic parrots: Can language models be too big?*, in Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency, 2021, pp. 610–623.

[23] 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.

[24] G. Bierman, M. Abadi, and M. Torgersen, *Understanding typescript*, in European Conference on Object-Oriented Programming, Springer, 2014.

[25] G. M. Bierman, M. Abadi, and M. Torgersen, *Understanding typescript*, in ECOOP, 2014.

[26] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, *Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools*, Queue, 20 (2022), pp. 35–57.

[27] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, *Enriching word vectors with subword information*, Transactions of the Association for Computational Linguistics, 5 (2017), pp. 135–146.

[28] 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).

[29] K. Bostrom and G. Durrett, *Byte pair encoding is suboptimal for language model pretraining*, arXiv preprint arXiv:2004.03720, (2020).

[30] G. Bracha, *Pluggable type systems*, (2004).

[31] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. 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. M. 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*, 2020.

[32] S. Bukhari and T. Waheed, *Model driven transformation between design models to system test models using uml: A survey*, in Proceedings of the 2010 National Software Engineering Conference, 2010, pp. 1–6.

[33] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang, et al., *Exploring software naturalness through neural language models*, arXiv preprint arXiv:2006.12641, (2020).

[34] L. Cardelli, *Type systems*, ACM Computing Surveys (CSUR), 28 (1996), pp. 263–264.

[35] R. Caruana, *A dozen tricks with multitask learning*, in Neural networks: tricks of the trade, Springer, 1998, pp. 165–191.

[36] C. Casalnuovo, E. T. Barr, S. K. Dash, P. Devanbu, and E. Morgan, *A theory of dual channel constraints*, in 2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), IEEE, 2020, pp. 25–28.

[37] 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.

[38] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi, *Fast and precise type checking for javascript*, Proceedings of the ACM on Programming Languages, 1 (2017), pp. 1–30.

[39] A. Chen, J. Scheurer, T. Korbak, J. A. Campos, J. S. Chan, S. R. Bowman, K. Cho, and E. Perez, *Improving code generation by training with natural language feedback*, arXiv preprint arXiv:2303.16749, (2023).

[40] 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).

[41] Z. Chen, S. 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, 47 (2019), pp. 1943–1959.

[42] R. Child, S. Gray, A. Radford, and I. Sutskever, *Generating long sequences with sparse transformers*, 2019.

[43] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. Colwell, and A. Weller, *Rethinking attention with performers*, 2021.

[44] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, *Deep reinforcement learning from human preferences*, Advances in neural information processing systems, 30 (2017).

[45] B. Chung et al., *Kafka: gradual typing for objects*, 2018.

[46] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, *What does bert look at? an analysis of bert's attention*, arXiv preprint arXiv:1906.04341, (2019).

[47] 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).

[48] R. Collobert and J. Weston, *A unified architecture for natural language processing: Deep neural networks with multitask learning*, in Proceedings of the 25th international conference on Machine learning, 2008, pp. 160–167.

[49] D. Crockford, *Jsmin: The javascript minifier*, 2003.

[50] A. M. Dai and Q. V. Le, *Semi-supervised sequence learning*, 2015.

[51] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming, et al., *Github copilot ai pair programmer: Asset or liability?*, arXiv preprint arXiv:2206.15331, (2022).

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

[53] W. De Mulder, S. Bethard, and M.-F. Moens, *A survey on the application of recurrent neural networks to statistical language modeling*, Computer Speech & Language, 30 (2015), pp. 61–98.

[54] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019.

[55] J. Devlin et al., *Bert: Pre-training of deep bidirectional transformers for language understanding*, arXiv preprint arXiv:1810.04805, (2018).

[56] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, *Detecting code smells using machine learning techniques: are we there yet?*, in 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner), IEEE, 2018, pp. 612–621.

[57] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, *Hoppity: Learning graph transformations to detect and fix bugs in programs*, in International Conference on Learning Representations (ICLR), 2020.

[58] I. Drori, S. Zhang, R. Shuttleworth, L. Tang, A. Lu, E. Ke, K. Liu, L. Chen, S. Tran, N. Cheng, et al., *A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level*, Proceedings of the National Academy of Sciences, 119 (2022), p. e2123433119.

[59] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, and B. Rost, *Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing*, arXiv preprint arXiv:2104.02443, (2021).

[60] T. Eloundou, S. Manning, P. Mishkin, and D. Rock, *Gpts are gpts: An early look at the labor market impact potential of large language models*, arXiv preprint arXiv:2303.10130, (2023).

[61] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, *Dermatologist-level classification of skin cancer with deep neural networks*, nature, 542 (2017), pp. 115–118.

[62] Z. Feng et al., *Codebert: A pre-trained model for programming and natural languages*, arXiv preprint arXiv:2002.08155, (2020).

[63] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, *Incoder: A generative model for code infilling and synthesis*, arXiv preprint arXiv:2204.05999, (2022).

[64] M. Gabel and Z. Su, *A study of the uniqueness of source code*, in Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, 2010, pp. 147–156.

[65] Z. Gao, C. Bird, and E. T. Barr, *To type or not to type: quantifying detectable bugs in javascript*, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 758–769.

[66] Z. Gao, C. Bird, and E. T. Barr, *To type or not to type: Quantifying detectable bugs in javascript*, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 758–769.

[67] Z. Gao, C. Bird, and E. T. Barr, *To type or not to type: Quantifying detectable bugs in javascript*, in Proceedings of the 39th International Conference on Software Engineering, ICSE '17, IEEE Press, 2017, p. 758769.

[68] J. Gilmer et al., *Neural message passing for quantum chemistry*, arXiv preprint arXiv:1704.01212, (2017).

[69] Github, *The 2021 state of the octoverse, 2021*, (2021).

[70] T. Gong, T. Lee, C. Stephenson, V. Renduchintala, S. Padhy, A. Ndirango, G. Keskin, and O. H. Elibol, *A comparison of loss weighting strategies for multi task learning in deep neural networks*, IEEE Access, 7 (2019), pp. 141627–141632.

[71] W. G. Griswold, *Program restructuring as an aid to software maintenance.*, (1992).

[72] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., *Graphcodebert: Pre-training code representations with data flow*, arXiv preprint arXiv:2009.08366, (2020).

[73] R. Hadsell, S. Chopra, and Y. LeCun, *Dimensionality reduction by learning an invariant mapping*, in 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), vol. 2, IEEE, 2006, pp. 1735–1742.

[74] S. Hanenberg, S. Kleinschmager, R. Robbes, E. Tanter, and A. Stefik, *An empirical study on the impact of static typing on software maintainability*, Empirical Softw. Engg., 19 (2014), p. 13351382.

[75] C. S. Hartzman and C. F. Austin, *Maintenance productivity: Observations based on an experience in a large system environment*, in Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1, 1993, pp. 138–170.

[76] J. He and M. Vechev, *Controlling large language models to generate secure and vulnerable code*, arXiv preprint arXiv:2302.05319, (2023).

[77] V. J. Hellendoorn, P. Devanbu, A. Polozov, et al., *Learning to infer run-time invariants from source code*, in NeurIPS 2020 Workshop on Computer-Assisted Programming.

[78] V. J. Hellendoorn et al., *Deep learning type inference*, in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018.

[79] V. J. Hellendoorn and A. A. Sawant, *The growing cost of deep learning for source code*, Communications of the ACM, 65 (2021), pp. 31–33.

[80] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, *Global relational models of source code*, in International conference on learning representations, 2019.

[81] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, *On the naturalness of software*, Communications of the ACM, 59 (2016), pp. 122–131.

[82] 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.

[83] J. Ho, N. Kalchbrenner, D. Weissenborn, and T. Salimans, *Axial attention in multidimensional transformers*, 2019.

[84] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, et al., *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*, 2001.

[85] J. Howard and S. Ruder, *Universal language model fine-tuning for text classification*, 2018.

[86] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec, *Strategies for pre-training graph neural networks*, arXiv preprint arXiv:1905.12265, (2019).

[87] 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.

[88] J. Hua and H. Wang, *On the effectiveness of deep vulnerability detectors to simple stupid bug detection*, in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 530–534.

[89] J.-T. Huang, J. Li, D. Yu, L. Deng, and Y. Gong, *Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers*, in 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, IEEE, 2013, pp. 7304–7308.

[90] Y. Huang, X. Hu, N. Jia, X. Chen, Y. Xiong, and Z. Zheng, *Learning code context information to predict comment locations*, IEEE Transactions on Reliability, 69 (2019), pp. 88–105.

[91] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, *Jigsaw: Large language models meet program synthesis*, in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1219–1231.

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

[93] K. Jesse, P. Devanbu, and A. A. Sawant, *Learning to predict user-defined types*, IEEE Transactions on Software Engineering, (2022).

[94] K. Jesse and P. T. Devanbu, *Manytypes4typescript: A comprehensive typescript dataset for sequence-based type inference*, (2022).

[95] 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.

[96] JetBrains and Contributors, *Python developer survey conducted by jetbrains and python software foundation, 2020*, (2020).

[97] ——, *Typescript developer survey conducted by jetbrains, 2020*, (2020).

[98] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, *Treebert: A tree-based pre-trained model for programming language*, arXiv preprint arXiv:2105.12485, (2021).

[99] D. Jurafsky and J. H. Martin, *Speech and language processing: An introduction to speech recognition, computational linguistics and natural language processing*, Upper Saddle River, NJ: Prentice Hall, (2008).

[100] A. V. Kamienski, L. Palechor, C.-P. Bezemer, and A. Hindle, *Pysstubs: Characterizing single-statement bugs in popular open-source python projects*, in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 520–524.

[101] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, *Pre-trained contextual embedding of source code*, (2019).

[102] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, *Learning and evaluating contextual embedding of source code*, 2020.

[103] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, *Big code!= big vocabulary: Open-vocabulary models for source code*, in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1073–1085.

[104] ——, *Big code!= big vocabulary: Open-vocabulary models for source code*, in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, New York, NY, USA, 2020, Association for Computing Machinery, p. 10731085.

[105] R.-M. Karampatsis and C. Sutton, *How often do single-statement bugs occur? the manysstubs4j dataset*, in Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 573–577.

[106] A. Karmakar, J. A. Prenner, M. D'Ambros, and R. Robbes, *Codex hacks hackerrank: Memorization issues and a framework for code synthesis evaluation*, arXiv preprint arXiv:2212.02684, (2022).

[107] A. Karmakar and R. Robbes, *What do pre-trained code models know about code?*, arXiv preprint arXiv:2108.11308, (2021).

[108] A. Kendall, Y. Gal, and R. Cipolla, *Multi-task learning using uncertainty to weigh losses for scene geometry and semantics*, 2018.

[109] D. Kici, G. Malik, M. Cevik, D. Parikh, and A. Basar, *A bert-based transfer learning approach to text classification on software requirements specifications.*, in Canadian Conference on AI, 2021.

[110] S. Kim, J. Zhao, Y. Tian, and S. Chandra, *Code prediction by feeding trees to transformers*, 2021.

[111] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, arXiv preprint arXiv:1609.02907, (2016).

[112] N. Kitaev, Ł. Kaiser, and A. Levskaya, *Reformer: The efficient transformer*, arXiv preprint arXiv:2001.04451, (2020).

[113] S. Kleinschmager, R. Robbes, et al., *Do static type systems improve the maintainability of software systems? an empirical study*, in 2012 20th IEEE International Conference on Program Comprehension (ICPC), 2012, pp. 153–162.

[114] 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).

[115] I. Kokkinos, *Ubernet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory*, in Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 6129–6138.

[116] T. Kudo, *Subword regularization: Improving neural network translation models with multiple subword candidates*, 2018.

[117] T. Kudo and J. Richardson, *Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing*, arXiv preprint arXiv:1808.06226, (2018).

[118] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, *Unsupervised translation of programming languages*, arXiv preprint arXiv:2006.03511, (2020).

[119] J. Latendresse, R. Abdalkareem, D. E. Costa, and E. Shihab, *How effective is continuous integration in indicating single-statement bugs?*, in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 500–504.

[120] B. Lester, R. Al-Rfou, and N. Constant, *The power of scale for parameter-efficient prompt tuning*, arXiv preprint arXiv:2104.08691, (2021).

[121] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, *Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension*, 2019.

[122] Q. Lhoest, A. V. del Moral, Y. Jernite, A. Thakur, P. von Platen, S. Patil, J. Chaumond, M. Drame, J. Plu, L. Tunstall, et al., *Datasets: A community library for natural language processing*, arXiv preprint arXiv:2109.02846, (2021).

[123] L. H. Li, M. Yatskar, D. Yin, C.-J. Hsieh, and K.-W. Chang, *Visualbert: A simple and performant baseline for vision and language*, arXiv preprint arXiv:1908.03557, (2019).

[124] X. L. Li and P. Liang, *Prefix-tuning: Optimizing continuous prompts for generation*, arXiv preprint arXiv:2101.00190, (2021).

[125] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, *Gated graph sequence neural networks*, arXiv preprint arXiv:1511.05493, (2015).

[126] L. Liebel and M. Körner, *Auxiliary tasks in multi-task learning*, arXiv preprint arXiv:1805.06334, (2018).

[127] B. P. Lientz, *Issues in software maintenance*, ACM Computing Surveys (CSUR), 15 (1983), pp. 271–278.

[128] T. Lin, Y. Wang, X. Liu, and X. Qiu, *A survey of transformers*, 2021.

[129] F. Liu, G. Li, Y. Zhao, and Z. Jin, *Multi-task learning based pre-trained language model for code completion*, in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 473–485.

[130] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, *Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing*, arXiv preprint arXiv:2107.13586, (2021).

[131] ———, *Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing*, ACM Computing Surveys, 55 (2023), pp. 1–35.

[132] S. Liu, E. Johns, and A. J. Davison, *End-to-end multi-task learning with attention*, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 1871–1880.

[133] Y. Liu, Z.-Y. Dou, and P. Liu, *Refsum: Refactoring neural summarization*, arXiv preprint arXiv:2104.07210, (2021).

[134] 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).

[135] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhotak, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, *In defense of soundiness: A manifesto*, Communications of the ACM, 58 (2015), pp. 44–46.

[136] 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.

[137] 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).

[138] X. Luo, Y. Xue, Z. Xing, and J. Sun, *Prcbert: Prompt learning for requirement classification using bert-based pretrained language models*, in 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–13.

[139] 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.

[140] J. MacGlashan, M. K. Ho, R. Loftin, B. Peng, G. Wang, D. L. Roberts, M. E. Taylor, and M. L. Littman, *Interactive learning from policy-dependent human feedback*, in International Conference on Machine Learning, PMLR, 2017, pp. 2285–2294.

[141] F. Madeiral and T. Durieux, *A large-scale study on human-cloned changes for automated program repair*, in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 510–514.

[142] R. S. Malik, J. Patra, and M. Pradel, *Nl2type: inferring javascript function types from natural language information*, in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 304–315.

[143] B. Marr, *How chatgpt and natural language technology might affect your job if you are a computer programmer*, Jan 2023.

[144] E. Mashhadi and H. Hemmati, *Applying codebert for automated program repair of java simple bugs*, in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 505–509.

[145] Microsoft and Contributors, *Intellisense, 2022*, (2022).

[146] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, *Recurrent neural network based language model.*, in Interspeech, vol. 2, Makuhari, 2010, pp. 1045–1048.

[147] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality*, 2013.

[148] A. M. Mir, E. Latoskinas, and G. Gousios, *Manytypes4py: A benchmark python dataset for machine learning-based type inference*, arXiv preprint arXiv:2104.04706, (2021).

[149] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, *Type4py: Deep similarity learning-based type inference for python*, arXiv preprint arXiv:2101.04470, (2021).

[150] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., *Human-level control through deep reinforcement learning*, nature, 518 (2015), pp. 529–533.

[151] B. Mosolygó, N. Vándor, G. Antal, and P. Hegedűs, *On the rise and fall of simple stupid bugs: a life-cycle analysis of sstubs*, in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 495–499.

[152] N. Nguyen and S. Nadi, *An empirical evaluation of github copilot's code suggestions*, in Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 1–5.

[153] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, *A conversational paradigm for program synthesis*, arXiv preprint arXiv:2203.13474, (2022).

[154] OpenAI.

[155] ———, *Gpt-4 technical report*, 2023.

[156] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al., *Training language models to follow instructions with human feedback*, arXiv preprint arXiv:2203.02155, (2022).

[157] I. V. Pandi et al., *Opttyper: Probabilistic type inference by optimising logical and natural constraints*, arXiv preprint arXiv:2004.00348, (2020).

[158] J. Park, *Javascript api misuse detection by using typescript*, in Proceedings of the companion publication of the 13th international conference on Modularity, 2014.

[159] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, *An empirical cybersecurity evaluation of github copilot's code contributions*, arXiv e-prints, (2021), pp. arXiv–2108.

[160] ———, *Asleep at the keyboard? assessing the security of github copilots code contributions*, in 2022 IEEE Symposium on Security and Privacy (SP), IEEE, 2022, pp. 754–768.

[161] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, *Can openai codex and other large language models help us fix security bugs?*, arXiv preprint arXiv:2112.02125, (2021).

[162] ———, *Examining zero-shot vulnerability repair with large language models*, in 2023 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, 2022, pp. 1–18.

[163] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin, *Integrating tree path in transformer for code representation*, Advances in Neural Information Processing Systems, 34 (2021).

[164] Y. Peng, Z. Li, C. Gao, B. Gao, D. Lo, and M. Lyu, *Hityper: A hybrid static type inference framework with neural prediction*, 2021.

[165] J. Pennington, R. Socher, and C. D. Manning, *Glove: Global vectors for word representation*, in Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 2014, pp. 1532–1543.

[166] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, *Do users write more insecure code with ai assistants?*, arXiv preprint arXiv:2211.03622, (2022).

[167] A. Peruma and C. D. Newman, *On the distribution of" simple stupid bugs" in unit test files: An exploratory study*, in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 525–529.

[168] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, *Deep contextualized word representations*, arXiv preprint arXiv:1802.05365, (2018).

[169] F. Petroni, P. Lewis, A. Piktus, T. Rocktäschel, Y. Wu, A. H. Miller, and S. Riedel, *How context affects language models' factual predictions*, arXiv preprint arXiv:2005.04611, (2020).

[170] B. C. Pierce, *Types and programming languages*, MIT press, 2002.

[171] M. Pradel et al., *Typewriter: Neural type prediction with search-based validation*, arXiv preprint arXiv:1912.03768, (2019).

[172] M. Pradel, G. Gousios, J. Liu, and S. Chandra, *Typewriter: Neural type prediction with search-based validation*, in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 209–220.

[173] J. A. Prenner and R. Robbes, *Automatic program repair with openai's codex: Evaluating quixbugs*, arXiv preprint arXiv:2111.03922, (2021).

[174] C. Quilty-Harper, *Ai chatgpt chatbot related prompt engineer jobs pay up to $335,000*, Mar 2023.

[175] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, *Improving language understanding by generative pre-training*, 2018.

[176] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, *Language models are unsupervised multitask learners*, OpenAI blog, 1 (2019), p. 9.

[177] 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).

[178] N. Rajkumar, R. Li, and D. Bahdanau, *Evaluating the text-to-sql capabilities of large language models*, arXiv preprint arXiv:2204.00498, (2022).

[179] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris, *Safe & efficient gradual typing for typescript*, in Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2015, pp. 167–180.

[180] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, *On the" naturalness" of buggy code*, in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 428–439.

[181] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, *A large scale study of programming languages and code quality in github*, in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, New York, NY, USA, 2014, Association for Computing Machinery, p. 155165.

[182] V. Raychev, M. Vechev, and A. Krause, *Predicting program properties from" big code"*, ACM SIGPLAN Notices, 50 (2015), pp. 111–124.

[183] V. Raychev, M. Vechev, and E. Yahav, *Code completion with statistical language models*, in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 419–428.

[184] B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster, *The ruby type checker*, in Proceedings of the 28th Annual ACM Symposium on Applied Computing, 2013, pp. 1565–1572.

[185] C. Richter and H. Wehrheim, *Tssb-3m: Mining single statement bugs at massive scale*, arXiv preprint arXiv:2201.12046, (2022).

[186] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, *The programmer's assistant: Conversational interaction with a large language model for software development*, arXiv preprint arXiv:2302.07080, (2023).

[187] C. K. Roy and J. R. Cordy, *A survey on software clone detection research*, Queens School of computing TR, 541 (2007), pp. 64–68.

[188] A. Sainani, P. R. Anish, V. Joshi, and S. Ghaisas, *Extracting and classifying requirements from software engineering contracts*, in 2020 IEEE 28th international requirements engineering conference (RE), IEEE, 2020, pp. 147–157.

[189] G. Sandoval, H. Pearce, T. Nys, R. Karri, B. Dolan-Gavitt, and S. Garg, *Security implications of large language model code assistants: A user study*, arXiv preprint arXiv:2208.09727, (2022).

[190] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, *What is it like to program with artificial intelligence?*, arXiv preprint arXiv:2208.06213, (2022).

[191] A. A. Sawant and P. Devanbu, *Naturally!: How breakthroughs in natural language processing can dramatically help developers*, IEEE Software, 38 (2021), pp. 118–123.

[192] F. Schroff, D. Kalenichenko, and J. Philbin, *Facenet: A unified embedding for face recognition and clustering*, 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), (2015).

[193] R. Sennrich et al., *Neural machine translation of rare words with subword units*, arXiv preprint arXiv:1508.07909, (2015).

[194] R. Sennrich, B. Haddow, and A. Birch, *Neural machine translation of rare words with subword units*, 2016.

[195] P. Shaw, J. Uszkoreit, and A. Vaswani, *Self-attention with relative position representations*, 2018.

[196] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, *Mtfuzz: fuzzing with a multi-task neural network*, in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 737–749.

[197] J. Shi, Z. Yang, J. He, B. Xu, and D. Lo, *Can identifier splitting improve open-vocabulary language model of code?*, arXiv preprint arXiv:2201.01988, (2022).

[198] T. Shin, Y. Razeghi, R. L. Logan IV, E. Wallace, and S. Singh, *Autoprompt: Eliciting knowledge from language models with automatically generated prompts*, arXiv preprint arXiv:2010.15980, (2020).

[199] V. Shiv and C. Quirk, *Novel positional encodings to enable tree-based transformers*, Advances in Neural Information Processing Systems, 32 (2019), pp. 12081–12091.

[200] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, *Megatron-lm: Training multi-billion parameter language models using model parallelism*, arXiv preprint arXiv:1909.08053, (2019).

[201] D. Shrivastava, H. Larochelle, and D. Tarlow, *Repository-level prompt generation for large language models of code*, arXiv preprint arXiv:2206.12839, (2022).

[202] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, *Learning loop invariants for program verification*, in Neural Information Processing Systems, 2018.

[203] J. Siek and W. Taha, *Gradual typing for objects*, in European Conference on Object-Oriented Programming, Springer, 2007, pp. 2–27.

[204] D. Sobania, M. Briesch, and F. Rothlauf, *Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming*, in Proceedings of the Genetic and Evolutionary Computation Conference, 2022, pp. 1019–1027.

[205] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano, *Learning to summarize with human feedback*, Advances in Neural Information Processing Systems, 33 (2020), pp. 3008–3021.

[206] A. Stuchlik and S. Hanenberg, *Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time*, in Proceedings of the 7th Symposium on Dynamic Languages, DLS '11, New York, NY, USA, 2011, Association for Computing Machinery, p. 97106.

[207] C. Sun, A. Myers, C. Vondrick, K. Murphy, and C. Schmid, *Videobert: A joint model for video and language representation learning*, in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 7464–7473.

[208] X. Sun, X. Liu, J. Hu, and J. Zhu, *Empirical studies on the nlp techniques for source code data preprocessing*, in Proceedings of the 2014 3rd international workshop on evidential assessment of software technologies, 2014, pp. 32–39.

[209] Z. Sun, X. Du, F. Song, M. Ni, and L. Li, *Coprotector: Protect open-source code against unauthorized training usage with data poisoning*, Proceedings of the ACM Web Conference 2022, (2022).

[210] A. Svyatkovskiy, S. Fakhoury, N. Ghorbani, T. Mytkowicz, E. Dinella, C. Bird, J. Jang, N. Sundaresan, and S. K. Lahiri, *Program merge conflict resolution via neural transformers*, in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 822–833.

[211] L. Tang, E. Ke, N. Singh, N. Verma, and I. Drori, *Solving probability and statistics problems by program synthesis*, arXiv preprint arXiv:2111.08267, (2021).

[212] Y. Tay, D. Bahri, L. Yang, D. Metzler, and D.-C. Juan, *Sparse sinkhorn attention*, 2020.

[213] T. Tenny, *Program readability: Procedures versus comments*, IEEE Transactions on Software Engineering, 14 (1988), p. 1271.

[214] I. Trummer, *Codexdb: Generating code for processing sql queries using gpt-3 codex*, arXiv preprint arXiv:2204.08941, (2022).

[215] N. Tsantalis, A. Ketkar, and D. Dig, *Refactoringminer 2.0*, IEEE Transactions on Software Engineering, (2020).

[216] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, *Generating accurate assert statements for unit test cases using pretrained transformers*, in Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test, 2022, pp. 54–64.

[217] L. Tunstall, L. von Werra, and T. Wolf, *Natural language processing with transformers*, " O'Reilly Media, Inc.", 2022.

[218] L. Tunstall, L. v. Werra, T. Wolf, and A. Géron, *Natural language processing with transformers: Building language applications with hugging face*, O'Reilly Media, 2022.

[219] P. Vaithilingam, T. Zhang, and E. L. Glassman, *Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models*, in CHI Conference on Human Factors in Computing Systems Extended Abstracts, 2022, pp. 1–7.

[220] L. Van der Maaten and G. Hinton, *Visualizing data using t-sne.*, Journal of machine learning research, 9 (2008).

[221] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, *Neural program repair by jointly learning to localize and repair*, 2019.

[222] 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.

[223] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, *Extracting and composing robust features with denoising autoencoders*, in Proceedings of the 25th international conference on Machine learning, 2008, pp. 1096–1103.

[224] M. M. Vitousek et al., *Design and evaluation of gradual typing for python*, in Proceedings of the 10th ACM Symposium on Dynamic languages, 2014, pp. 45–56.

[225] S. Voruganti, K. Jesse, and P. Devanbu, *Flextype: A plug-and-play framework for type inference models*, in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–5.

[226] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, *No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence*, in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 382–394.

[227] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, *Linformer: Self-attention with linear complexity*, arXiv preprint arXiv:2006.04768, (2020).

[228] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, *Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation*, arXiv preprint arXiv:2109.00859, (2021).

[229] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, *On learning meaningful assert statements for unit test cases*, in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1398–1409.

[230] J. Wei et al., *Lambdanet: Probabilistic type inference using graph neural networks*, arXiv preprint arXiv:2005.02161, (2020).

[231] 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).

[232] F. Wen, C. Nagy, G. Bavota, and M. Lanza, *A large-scale empirical study on code-comment inconsistencies*, in 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE, 2019, pp. 53–64.

[233] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, *Toward deep learning software repositories*, in 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 334–345.

[234] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al., *Huggingface's transformers: State-of-the-art natural language processing*, arXiv preprint arXiv:1910.03771, (2019).

[235] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, *The effect of modularization and comments on program comprehension*, in Proceedings of the 5th international conference on Software engineering, 1981, pp. 215–223.

[236] J. Wu, L. Ouyang, D. M. Ziegler, N. Stiennon, R. Lowe, J. Leike, and P. Christiano, *Recursively summarizing books with human feedback*, arXiv preprint arXiv:2109.10862, (2021).

[237] C. S. Xia and L. Zhang, *Conversational automated program repair*, arXiv preprint arXiv:2301.13246, (2023).

[238] Y. Xiong, Z. Zeng, R. Chakraborty, M. Tan, G. Fung, Y. Li, and V. Singh, *Nyströmformer: A nyström-based algorithm for approximating self-attention*, 2021.

[239] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, *A systematic evaluation of large language models of code*, arXiv preprint arXiv:2202.13169, (2022).

[240] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, *Python probabilistic type inference with natural language support*, in Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, 2016, pp. 607–618.

[241] Y. Yang, X. Xia, D. Lo, and J. Grundy, *A survey on deep learning for software engineering*, arXiv preprint arXiv:2011.14597, (2020).

[242] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, *Xlnet: Generalized autoregressive pretraining for language understanding*, in Advances in neural information processing systems, 2019, pp. 5753–5763.

[243] B. Yetistiren, I. Ozsoy, and E. Tuzun, *Assessing the quality of github copilots code generation*, in Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, 2022, pp. 62–71.

[244] S. Yovine, *Kronos: A verification tool for real-time systems*, International Journal on Software Tools for Technology Transfer, 1 (1997), pp. 123–133.

[245] L. A. Zadeh, *Soft computing and fuzzy logic*, in Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi a Zadeh, World Scientific, 1996, pp. 796–804.

[246] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, *Big bird: Transformers for longer sequences*, 2021.

[247] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, *A novel neural source code representation based on abstract syntax tree*, in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 783–794.

[248] W. Zhu and M. W. Godfrey, *Mea culpa: How developers fix their own simple bugs differently from other developers*, in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 515–519.

[249] A. Ziegler, *https://github.blog/2021-06-30-github-copilot-research-recitation/*.

[250] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, *Fine-tuning language models from human preferences*, arXiv preprint arXiv:1909.08593, (2019).