How Robust are Neural Code Completion Models to Source Code Transformation?

Anonymous authors
Paper under double-blind review

Abstract

Neural language models hold great promise as tools for computer-aided programming, but questions remain over their reliability and the consequences of overreliance. In the domain of natural language, prior work has revealed these models can be sensitive to naturally-occurring variance and malfunction in unpredictable ways. A more methodical examination is necessary to understand their behavior on programming-related tasks. In this work, we develop a methodology for systematically evaluating neural code completion models using common source code transformations. We measure the distributional shift induced by applying those transformations to a dataset of handwritten code fragments on four pretrained models, which exhibit varying degrees of robustness under transformation. Preliminary results from those experiments and observations from a qualitative analysis suggest that while these models are promising, they should not be relied upon uncritically. Our analysis provides insights into the strengths and weaknesses of different models, and serves as a foundation for future work towards improving the accuracy and robustness of neural code completion.

1 Introduction

Neural language models (NLMs) play an increasingly synergistic role in software engineering, and are featured prominently in recent work on neural code completion (1). Code completion is the task of automatically providing a list of potential completions for missing tokens in source code. By training on a large corpus of source code, NLMs are capable of learning syntax (3; 19), stylistic elements like coding conventions (13) as well as more complex patterns like documentation (8). The same pretrained model can be applied to a variety of programming tasks, such as code summarization and bug detection, by strategically placing the holes and conditioning the model's predictions on an appropriate sample space.

However, there are risks associated with using NLMs (7). In the natural language domain, these models can be sensitive to stylistic variation, which is difficult to detect a priori and may lull users into a false sense of complacency. Likewise, applying NLMs in an unfamiliar coding context may result in predictions which are misaligned with developer intent, especially when the surrounding style, idioms, or APIs depart from the model's original training set. This phenomenon, called 'domain mismatch' or 'overfitting', is known to affect statistical learning writ large and the specific factors responsible for its occurrence in source code can be identified using domain-specific tests such as the ones we propose herein.

Inspired by recent work investigating the robustness of NLMs in both natural language (21) and source code (14), we develop a methodology for systematically evaluating neural code completion models using common source code transformations (SCTs). As a demonstration, we define a set of semantics-preserving transformations, and measure the distributional shift induced by applying them to a large collection of code fragments on four pretrained models. We compare the same models and transformations across three downstream tasks, i.e., code completion, document synthesis, and variable misuse detection. Our results provide insights into the strengths and weaknesses of those models, and serve as a foundation for future work towards improving the accuracy and robustness of neural code completion via retraining.

2 Related Work

Recent work in **neural language modeling** has shown impressive progress in long-range sequence prediction, starting with Vaswani et al.'s self-attention (20) mechanism, to the BERT (4) and RoBERTa (10) architectures, now widely available in neural code completion models such as CodeBERT (5) and GraphCodeBERT (6). Though impressive, these models have known limitations, such as their inability to represent long-term dependencies and their sensitivity to noise in the input space (18). Similar studies have been undertaken to characterize the robustness of pretrained language models of source code (7).

Our work is complementary to these studies in that we focus on neural language models and utilize structured program transformations as a kind of litmus test for detecting domain mismatch. We probe the models' understanding of source code using masked language modeling (MLM) (16), constrained decoding to generate contextually-relevant text, and a suite of semantically-preserving SCTs to measure the shift in prediction accuracy across three downstream tasks. Each of these tasks may be viewed as a special case of **sketch-based program synthesis** (17). Our approach also shares connections to automated software testing, self-supervised learning, and natural language inference as we describe below.

First conceived in the **software testing** literature, metamorphic testing (2) is a concept known in machine learning as *self-supervision*. When labels are scarce but invariant to certain groups of transformation or *metamorphic relations*, given a finite labeled dataset, one can generate an effectively limitless quantity of synthetic data by selecting and recombining those transformations in a systematic manner. For example, computer vision models should be invariant to shift, scale and rotation: given a small dataset of labeled images, we can apply these transformations to generate much larger training or validation set. Could similar kinds of transformations exist for code? One promising avenue is to consider term-rewriting.

Due to the precise distinction between syntax and semantics in programming languages, it is possible to generate semantically-admissible perturbations without requiring a learned similarity metric (e.g., word or sentence embedding). Using a simple **term rewriting** system with a context-sensitive grammar allows us to reason about plausible variations to source code. This idea is partly inspired by Josh Rule's work, "The Child as a Hacker" (15), whose thesis resonates with the authors' own experience learning to program. Coupled with a neural controller and some kind of interactive debugging mechanism, we believe this approach offers a compelling alternative to imitation learning (à la supervision on a gigantic corpus of text), and more faithfully resembles the way humans learn to write code.

Similar research has been undertaken (22; 11) to characterize the **formal grammars** which neural language models can recognize in theory. Our work builds on this literature from a practical standpoint: we investigate how neural code completion models respond to plausible cosmetic variation in real-world code fragments. Though rudimentary, our work can be seen as an early attempt to study how neural language models behave in realistic programming scenarios and lays the foundation for future work towards bridging the gap between natural and programming language understanding. By partitioning the Chomsky hierarchy into fine-grained semantic categories, we hope to precisely characterize the expressive power of neural code completion models via their understanding of programming language concepts.

Towards this end, we incorporate several ideas from **programming language theory** to probe the models' understanding of source code. For example, we leverage the concept of contextual equivalence (12) to justify the validity of our SCTs. We borrow the concept of capture-avoiding substitution from λ-calculus to prevent name collision under variable renaming. We adapt ideas from flow analysis to reorder statements which share no dataflow dependencies. Taken together, these ideas provide a principled way to reason about the correctness of our transformations and their impact on the underlying program's semantics. We believe that ideas from PL theory have been largely overlooked by the ML for code community and represent a significant opportunity for future research on code completion.

Last but not least, we tap into work on natural language inference and question-answering (9). This work contains a rich set of ideas for testing program comprehension in neural language models and may be useful for bug detection and automatic program repair.

3 Background

The same program can be written in many possible ways. Often, those variants contain superficial changes to the source code, but do not meaningfully alter the behavior of the underlying program. No matter the context, neural language models should not become suddenly confused when presented with a cosmetically altered program: a model trained on a language semantically invariant to certain rewrites should itself exhibit invariance under those same rewrites. However this property may not necessarily hold.

As documented by prior literature, neural language models can be prone to semantic drift that is not supported by the underlying language semantics. The question becomes, how do we estimate the robustness of those models to rewrites which are known *a priori* to be cosmetic in nature? Our work addresses this question by identifying four classes of cosmetic rewrites, applies them to a dataset of Java source code, and measures the relative drift of four SoTA pretrained language models across three separate code completion tasks.

Our work identifies three high-level categories of source code transformations:

1. **Syntactic**, which may produce a valid or invalid parse tree. For example: simple refactoring, typos, imbalanced parentheses, unparsable code.
2. **Semantic**, which may either preserve or alter the program's meaning. For example: functional code clones, dis-equal constant or expression rewriting.
3. **Cosmetic**, which is strictly superficial. For example: variable renaming, independent statement reordering, extra documentation, dead code, or logging.

In contrast with syntactic or semantic transformations, cosmetic transformations are semantically identical, syntactically valid and only superficially alter syntactic structure. We show that even in this highly restrictive space of transformations, source code has many degrees of freedom: two authors implementing the same function may select different variable names or other cosmetic features, such as whitespaces, diagnostic statements or comments. Yet our results suggest that even in this narrow space of transformations, SoTA neural code completion models can be surprisingly sensitive to noise in the cosmetic domain.

4 Method

Our goal is to measure the robustness of SoTA neural code completion models on natural code fragments exposed to various cosmetic transformations. Hence, we construct one SCT from each of the following four categories of cosmetic changes:

1. **Synonym renaming**: substitutes variable names with synonyms.
2. **Extraneous code**: sprinkles non-essential statements into code.
3. **Statement reordering**: reorders dataflow independent statements.
4. **Permute argument order**: scrambles user-defined method arguments.

Put simply, we use the following rewriting criteria to implement our SCTs:

1. The RenameTokens SCT substitutes each CamelCase subword in the most frequent user-defined token with a uniformly-sampled lemma from its WordNet hypernym ego graph up to three hops away, representing an alternately-chosen (e.g., variable or function) name of similar meaning.
2. The AddExtraLogging SCT adds intermittent print statements in linear chains of code, with a single argument synthesized by the code completion model for added variation. More generally, this can be any superfluous statement which does not change the runtime semantics.

3. The SwapIndepLines SCT swaps adjacent lines of equal scope and indentation which share no tokens in common. Although it may introduce semantic drift in some code fragments, this SCT ideally represents an alternate topsort on the DFG.
4. The PermuteArguments SCT shuffles the arguments of a user-defined function of dyadic or higher-arity, representing an alternate parameter order of a project-local function (i.e., defined outside the standard library).

For a more technical description of our rewrite semantics, please refer to Appendix A.

Three tasks are considered. Each task uses a custom mask generator, allowing us to simulate multiple downstream tasks with the same test harness, only modifying the mask locations and admissible completions. Each task is scored with a task-appropriate metric. For single-token completion, we compare the top-1 precision across the full vocabulary and all masks:

$$\text{Precision@1} = \frac{1}{|\textsc{Tokens}|} \sum_{\texttt{tok} \in \textsc{Tokens}} \textsc{Model}(\textsc{Code}\,\{\,\texttt{tok} \mapsto \texttt{<mask>}\,\}) = \textsc{Code}$$

For document synthesis, we use a variant of the ROUGE score measuring synonym overlap:

$$\text{ROUGE-synonym} = \Delta_{\text{Syn}}(\textsc{Model}^k(\textsc{Code}\,\{\,\texttt{//comment} \mapsto \texttt{//<mask>}\,\}),\ \texttt{//comment})$$

where $\textsc{Model}^k$ denotes autoregressively sampling k consecutive tokens, and Δ_{Syn} denotes the unigram synonym overlap between the original and synthetic comments.

For question answering, i.e., variable misuse, we use mean reciprocal rank (MRR) with a set of multiple-choice distractors sampled uniformly from the surrounding context:

$$\text{MRR} = \frac{1}{|\textsc{Tokens}|} \sum_{\texttt{tok} \in \textsc{Tokens}} \textsc{Rank}\big(\texttt{tok}, \textsc{Model}(\textsc{Code}\,\{\,\texttt{tok} \mapsto \texttt{<mask>}\,\}, \textsc{hints})\big)^{-1}$$

For code completion, we uniformly sample and mask N individual tokens in both the original and transformed code fragment for evaluation. We then collect the model's highest-scoring predictions for each mask location, and average the completion accuracy on the original and transformed code fragment. An example may be found in Fig. 2.

Similarly, for document synthesis, we mask a naturally-occurring comment and autoregressively synthesize a new one in its place, then compare the ROUGE-scores of the synthetic documents before and after transformation. An example of may be found in Fig. 3.

Our dataset was constructed by cloning a hundred of the highest-starring Java repositories hosted by GitHub organizations with over 100 forks and between 1 and 10 MB in size, sorted by issue activity. Selecting projects matching these criteria will retrieve a diverse collection of active repositories with enterprise-level code quality and reasonable stylistic diversity.

We executed a full factorial experiment consisting of four SCTs, four state-of-the-art pre-trained models (GraphCodeBERT, CodeBERT, CodeBERT-Small, RoBERTa-Java) and three downstream tasks (code completion and document synthesis). While the number of samples may vary per-model and per-bucket, we provide the same wall clock time (360 minutes) and hardware resources (nVIDIA Tesla V100) to each model. The number of code fragments each can evaluate in the allotted time varies depending on the architecture, but in each case, the significant figures have mostly converged.

Figure 1 captures the results of this process. Each table represents a 2D slice of the hypercube corresponding to a single task, with dual box-and-whisker plots for each model, before and after applying the transformation listed on the horizontal-axis. The raw data cube generated by our experimental pipeline can be found in the Appendix B.

5 Results and discussion

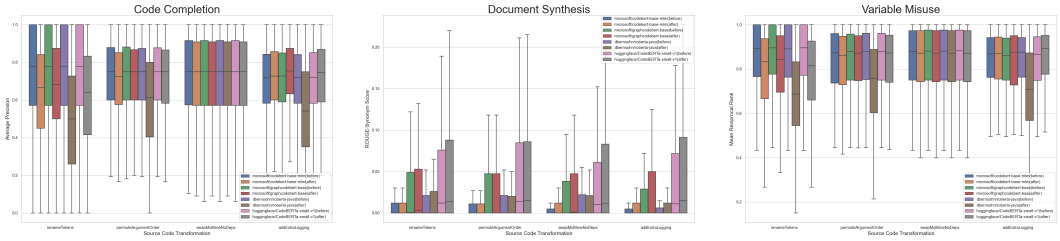

Figure 1: Semantic shift across all tasks (code completion, document synthesis and variable misuse), models (GraphCodeBERT, CodeBERT, CodeBERT-Small, RoBERTa-Java) and SCTs (Rename, AddExtraLogging, SwapIndepLines, PermuteArguments).

A careful examinination of the results in Fig. 1 allows us to draw the following conclusions:

- Models specifically intended to handle source code, (i.e., CodeBERT and GraphCodeBERT) are more robust than those designed for text alone (i.e., RoBERTa).
- RoBERTa would appear to be uniquely sensitive to PermuteArguments and AddExtraLogging, a pathology which none of its code cousins shared.
- RenameTokens appears to have a detrimental effect across the board.
- SwapIndepLines seems to have a negligible effect. This result aligns with the word order findings reported by Sinha et al. (16) in natural language.
- Our results lend credence to the relative model rankings reported by prior literature: RoBERTa $\ll$ CodeBERT $<$ GraphCodeBERT.
- Document synthesis has a fairly high variance. This may be due to poor alignment between the metric and the task. While comments are qualitatively passable, the Δ_{Sym} metric does poorly at discriminating between good comments and bad ones.

For samples from the code and document synthesizer, we refer the reader to Appendix D.

6 Future work

Our SCTs can be viewed as "possible worlds" in the tradition of modal logic: the original author plausibly could have written the same procedure in a slightly different form. Although we are unable to access all these worlds, we can posit the existence and likelihood of some, and given a dataset of code fragments, begin to probe a candidate model's predictions.

One intriguing avenue for future work would be to consider combinations of source code transformations. This would vastly expand the cardinality of the validation set, enabling us to access a much larger space of possible worlds, albeit potentially at the risk of lower semantic admissibility, as arbitrary combinations of SCTs can quickly produce invalid code. This presents an interesting engineering challenge and possible extension to this work.

Although we currently only use average precision, ROUGE-synonym and mean reciprocal rank, it would be useful to report Kantorovich-Rubinstein distance and other probability metrics. In addition to their utility for evaluating model robustness, these metrics can also be used to retrain those same models, a direction we hope to explore in future work.

Finally, one could imagine using the code completion model itself to generate code for testing the same model. We have implemented this functionality to a limited extent in the AddExtraLogging SCT, in which the model synthesizes a single token to log, and the InsertComment SCT, where the model inserts a short comment. While this approach could be a useful way to generate additional training data, it would would require careful monitoring and postprocessing to avoid introducing unintended feedback loops.

7 Conclusion

Neural language models hold much promise for improved code completion, however complacency can lead to increased reviewer burden or more serious technical debt if widely adopted. While trade secrecy may prevent third-party inspection of pretrained models, users would still like some assurance of their model's robustness to naturally-occurring variance. Our work helps to address this use case by treating the model as a black box: it does not require direct access to the model parameters or training data to evaluate its generalization ability.

Our contributions in this work are twofold: we demonstrate that SoTA neural language models for source code, despite their effectiveness on long-range sequence prediction tasks, are unpredictable in the presence of specifically-constructed cosmetic variation. We also describe a systematic approach and open source implementation of a newly-developed software toolkit which allows users to empirically probe a candidate model's robustness to various categories of syntactic and semantic source code transformations. Our results are fully reproducible and can be found at: `https://anonymous.4open.science/r/cstk-dl4c`.

We have shown that despite their impressive performance on certain tasks, neural code completion models are still susceptible to domain mismatch and can be sensitive to rewriting. This can be identified by designing specific rewriting scenarios, such as the ones we propose, that target the specific factors responsible for overfitting in source code. Our work can help developers better understand the strengths and weaknesses of different models and empirically compare the accuracy and robustness of neural code completion systems.

References

[1] Mark Chen et al. Evaluating large language models trained on code, 2021.

[2] TY Chen, J Feng, and TH Tse. Metamorphic testing of programs on partial differential. *Information and Computation*, 121(1):93–102, 1995.

[3] Nadezhda Chirkova and Sergey Troshin. Empirical study of transformers for source code. *arXiv preprint arXiv:2010.07987*, 2020.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[6] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, et al. GraphCodeBERT: Pre-training code representations with data flow, 2021.

[7] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[8] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[9] Chenxiao Liu and Xiaojun Wan. CodeQA: A question answering dataset for source code comprehension. *arXiv preprint arXiv:2109.08365*, 2021.

[10] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach, 2019.

[11] William Merrill, Yoav Goldberg, and Noah A Smith. On the power of saturated transformers: A view from circuit complexity. *arXiv preprint arXiv:2106.16213*, 2021.

[12] James Hiram Morris Jr. *Lambda-calculus models of programming languages.* PhD thesis, Massachusetts Institute of Technology, 1969.

[13] Terence Parr and Jurgen Vinju. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–151, 2016.

[14] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043*, 2020.

[15] Joshua Stewart Rule. *The child as hacker: building more human-like models of learning.* PhD thesis, Massachusetts Institute of Technology, 2020.

[16] Koustuv Sinha, Robin Jia, Dieuwke Hupkes, Joelle Pineau, Adina Williams, and Douwe Kiela. Masked language modeling and the distributional hypothesis: Order word matters pre-training for little. *arXiv preprint arXiv:2104.06644*, 2021.

[17] Armando Solar-Lezama. *Program synthesis by sketching.* University of California, Berkeley, 2008.

[18] Lichao Sun, Kazuma Hashimoto, Wenpeng Yin, Akari Asai, Jia Li, Philip Yu, and Caiming Xiong. Adv-BERT: BERT is not robust on misspellings! Generating nature adversarial samples on BERT. *arXiv preprint arXiv:2003.04985*, 2020.

[19] Sergey Troshin and Nadezhda Chirkova. Probing pretrained models of source code. *arXiv preprint arXiv:2202.08975*, 2022.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[21] Tianlu Wang, Xuezhi Wang, Yao Qin, Ben Packer, Kang Li, Jilin Chen, Alex Beutel, and Ed Chi. Cat-gen: Improving robustness in NLP models via controlled adversarial text generation. *arXiv preprint arXiv:2010.02338*, 2020.

[22] Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision RNNs for language recognition. *arXiv preprint arXiv:1805.04908*, 2018.

A Rewrite semantics

We define the following four transformations to illustrate the idea of semantic invariance.

$$\frac{\Gamma \vdash \texttt{function(args)} \qquad \texttt{function} \notin \textsc{StandardLibrary}}{\Gamma \vdash \textsc{Code}\,\{\texttt{function(args)} \mapsto \texttt{function(args)}\}}\ \textsc{PermuteArguments}$$

$$\frac{\Gamma \vdash \begin{array}{l}\texttt{␣line\_1;}\\ \texttt{␣line\_2;}\end{array} \qquad \textsc{Tokens}(\texttt{line\_1}) \cap \textsc{Tokens}(\texttt{line\_2}) = \varnothing}{\Gamma \vdash \begin{array}{l}\texttt{␣line\_2;}\\ \texttt{␣line\_1;}\end{array}}\ \textsc{SwapIndepLines}$$

$$\frac{\Gamma \vdash \texttt{var} \in \textsc{ScopedNames} \qquad \begin{array}{l}\texttt{␣line\_1;}\\ \texttt{␣line\_2;}\end{array}}{\Gamma \vdash \begin{array}{l}\texttt{␣line\_1;}\\ \texttt{␣println(var);}\\ \texttt{␣line\_3;}\end{array}}\ \textsc{AddExtraLogging}$$

$$\frac{\Gamma \vdash \texttt{var} \in \textsc{ScopedNames} \qquad \texttt{fresh} \in \textsc{Synonyms}(\texttt{var}) \setminus \textsc{ScopedNames}}{\Gamma \vdash \textsc{Code}\,\{\texttt{var} \mapsto \texttt{fresh}\}}\ \textsc{Rename}$$

For example, let $\textsc{Code} := \texttt{int old = 0; while (<mask> < 10) \{...\}}$ and assume we have a pretrained language model $\mathcal{M}_\theta : \texttt{String} \to \texttt{String}$ which gives the completion:

$$\mathcal{M}_\theta[\textsc{Code}] \vdash \textsc{Code}\,\{\texttt{<mask>} \mapsto \texttt{old}\} \tag{1}$$

If we then rename the variable `old` to `fresh` inside the fragment Code, we would expect our language model's completion to shift in kind, a property we call *name equivariance*. Importantly, we require fresh name generation to ensure capture-avoiding substitution:

$$\frac{\Gamma \vdash \mathcal{M}_\theta \qquad \texttt{code}\colon \textsc{Code} \qquad \{\texttt{old} \mapsto \texttt{fresh}\} : \textsc{Name} \to !\textsc{Name}}{\Gamma \vdash \mathcal{M}_\theta[\texttt{code}\,\{\texttt{old} \mapsto \texttt{fresh}\}] = \mathcal{M}_\theta[\texttt{code}]\{\texttt{old} \mapsto \texttt{fresh}\}}\ \textsc{NameEquivariance}$$

Consider another: if we reorder the arguments of a project-local function, we would expect the model to perform identically over all masked tokens, said arguments notwithstanding:

$$\frac{\Gamma \vdash \mathcal{M}_\theta \qquad \texttt{code}\colon \textsc{Code} \qquad \texttt{fn(args)}\colon \textsc{ProjectLocal}}{\Gamma \vdash \mathcal{M}_\theta[\texttt{code}] = \mathcal{M}_\theta[\texttt{code}\,\{\texttt{fn(...)} \mapsto \texttt{fn(🔀args)}\}]}\ \textsc{ArgOrderInvariance}$$

B Detailed Results

We ran three tasks, code completion, documentation synthesis and variable misuse.

Pretrained Language Model (μ, σ)		Rename	PermuteArgument	SwapIndepLines	AddExtraLogging
microsoft/codebert-base-mlm	Before	(0.7224, 0.2768)	(0.7194, 0.2191)	(0.7089, 0.2624)	(0.6996, 0.2077)
	After	(0.6100, 0.3085)	(0.6978, 0.2227)	(0.7058, 0.2607)	(0.7122, 0.2001)
microsoft/graphcodebert-base	Before	(0.7230, 0.2767)	(0.7227, 0.2173)	(0.7088, 0.2624)	(0.7050, 0.2047)
	After	(0.6410, 0.2998)	(0.7154, 0.2169)	(0.7058, 0.2607)	(0.7382, 0.1932)
dbernsohn/roberta-java	Before	(0.7219, 0.2773)	(0.7196, 0.2188)	(0.7087, 0.2624)	(0.6958, 0.2095)
	After	(0.4951, 0.3165)	(0.5995, 0.2632)	(0.7058, 0.2607)	(0.5479, 0.2577)
huggingface/CodeBERTa-small-v1	Before	(0.7226, 0.2771)	(0.7216, 0.2149)	(0.7088, 0.2624)	(0.7004, 0.2095)
	After	(0.5953, 0.3134)	(0.7096, 0.2249)	(0.7058, 0.2607)	(0.7113, 0.2119)

Table 1: Mean and standard deviation for single-token code completion accuracy.

Pretrained Language Model (μ, σ)		Rename	PermuteArgument	SwapIndepLines	AddExtraLogging
microsoft/codebert-base-mlm	Before	(0.0385, 0.2684)	(0.0427, 0.3500)	(0.0309, 0.2477)	(0.0545, 0.4637)
	After	(0.0430, 0.2996)	(0.0411, 0.3091)	(0.0419, 0.2959)	(0.0510, 0.3839)
microsoft/graphcodebert-base	Before	(0.0987, 0.3826)	(0.0983, 0.3214)	(0.0891, 0.4181)	(0.0850, 0.3000)
	After	(0.1026, 0.3590)	(0.0982, 0.3212)	(0.0989, 0.3549)	(0.0974, 0.2221)
dbernsohn/roberta-java	Before	(0.0827, 0.5086)	(0.0727, 0.4453)	(0.0793, 0.4769)	(0.0680, 0.5349)
	After	(0.0891, 0.5484)	(0.0781, 0.4792)	(0.0845, 0.5428)	(0.0733, 0.4994)
huggingface/CodeBERTa-small-v1	Before	(0.1101, 0.3932)	(0.1205, 0.4148)	(0.0997, 0.3912)	(0.1252, 0.5175)
	After	(0.1143, 0.3645)	(0.1170, 0.3668)	(0.1125, 0.3811)	(0.1230, 0.3891)

Table 2: Mean and standard deviation ROUGE-score for document synthesis.

Pretrained Language Model (μ, σ)		Rename	PermuteArgument	SwapIndepLines	AddExtraLogging
microsoft/codebert-base-mlm	Before	(0.8150, 0.2502)	(0.7692, 0.2947)	(0.7935, 0.2683)	(0.7568, 0.3086)
	After	(0.7508, 0.2663)	(0.7574, 0.2992)	(0.7861, 0.2720)	(0.7412, 0.3297)
microsoft/graphcodebert-base	Before	(0.8196, 0.2478)	(0.7758, 0.2873)	(0.7952, 0.2670)	(0.7529, 0.3071)
	After	(0.7638, 0.2638)	(0.7673, 0.2919)	(0.7861, 0.2720)	(0.7378, 0.3376)
dbernsohn/roberta-java	Before	(0.8149, 0.2486)	(0.7727, 0.2947)	(0.7952, 0.2670)	(0.7620, 0.3053)
	After	(0.6708, 0.2282)	(0.7148, 0.2399)	(0.7861, 0.2720)	(0.6797, 0.2580)
huggingface/CodeBERTa-small-v1	Before	(0.8182, 0.2480)	(0.7736, 0.2934)	(0.7954, 0.2670)	(0.7578, 0.3053)
	After	(0.7569, 0.2334)	(0.8023, 0.2318)	(0.7861, 0.2720)	(0.8137, 0.2413)

Table 3: Mean and standard deviation MRR for variable misuse detection.

C Experimental architecture

Though primarily an empirical study, this work also showcases a software framework for evaluating neural code completion models. It offers a number of advantages from an engineering standpoint: due to its functional implementation, it is efficient, parallelizable and highly modular, allowing others to easily reuse and extend our work with new benchmarks.

The design of this framework is to our knowledge, unique, and merits some discussion. The entire pipeline from data mining to preprocessing, evaluation and table generation is implemented as a pure functional program in the point-free style. Given a code completion model `cc: String→String`, a list of code fragments, `snps: List<String>`, a masking procedure, `msk: String→String`, an SCT, `sct: String→String`, and a single metric over code fragments, `mtr: (String, String)→Float`, we measure the average relative discrepancy before and after applying `sct` to `snps`:

```
fun evaluate(cc, snps, msk, sct, mtr) = Δ(
  zip(snps, snps | msk | cc) | mtr | average,
  zip(snps | sct, snps | sct | msk | cc) | mtr | average
)
```

where `|` maps a function over a sequence, and `zip` zips two sequences into a sequence of pairs. We assume `snps` and `msk` are fixed, and evaluate three neural code completion models across four different SCTs, and three separate tasks.

Using our framework, it is possible to view the marginals of a rank-n tensor, representing an n-dimensional hyper-distribution formed by the Cartesian product of all variables under investigation (e.g., SCT, task, model). During evaluation, we sample these independent variables uniformly using a quasirandom sequence to ensure entries are evenly populated. We then record the first and second moments of the dependent variable of interest using a sketch-based histogram. Results are continuously delivered to the user, who may preview 2D marginals of any pair and watch the error bounds grow tighter as additional samples are drawn. This feature is indispensable when running on preemptible infrastructure and can be trivially parallelized to increase the experiment's statistical power or explore larger swaths of the experimental design space.

D Examples

1.a) Original method

```
public void flush(int b) {
  buffer.write((byte) b);
  buffer.compact();
}
```

1.b) Synonymous variant

```
public void flush(int b) {
  cushion.write((byte) b);
  cushion.compact();
}
```

2.a) Multi-masked method

```
public void <MASK>(int b) {
  buffer.<MASK>((byte) b);
  <MASK>.compact();
}
```

2.b) Multi-masked variant

```
public void <MASK>(int b) {
  cushion.<MASK>((byte) b);
  <MASK>.compact();
}
```

3.a) Model predictions

```
public void output(int b) {
  buffer.write((byte) b);
  buffer.compact();
}
```

3.b) Model predictions

```
public void append(int b) {
  cushion.add((byte) b);
  cushion.compact();
}
```

Figure 2: Here, we apply the RenameTokens SCT, then mask various tokens in the surrounding context and report the model's predictions. In this example, the model correctly predicts $\frac{2}{3}$ masks in the original method and $\frac{1}{3}$ after renaming.

1.) Original method with ground truth document

```
public void testBuildSucceeds(String gradleVersion) {
  setup(gradleVersion);
  // Make sure the test build setup actually compiles
  BuildResult buildResult = getRunner().build();
  assertCompileOutcome(buildResult, SUCCESS);
}
```

2.) Synthetic document before applying SCT

```
public void testBuildSucceeds(String gradleVersion) {
  setup(gradleVersion);
  // build the tests with gradletuce compiler
  BuildResult buildResult = getRunner().build();
  assertCompileOutcome(buildResult, SUCCESS);
}
```

3.) Synthetic document after applying SCT

```
public void testBuildSucceeds(String gradleAdaptation) {
  setup(gradleAdaptation);
  // build the actual code for test suite generation
  BuildResult buildResult = getRunner().build();
  assertCompileOutcome(buildResult, SUCCESS);
}
```

Figure 3: Here, we apply the RENAME SCT, then mask the comment on line 3 and autoregressively sample tokens from the decoder to generate two synthetic comments, one before and one after applying the SCT. The stemmed synonym overlap is `build, compile, test,` before applying the SCT and `actual, build, test` afterwards.

Initially, we seeded the document completion using `//<MASK>` and applied a greedy autoregressive decoding strategy, recursively sampling the softmax top-1 token and subsequently discarding all malformed comments. This strategy turns out to have a very high rejection rate, due to its tendency to produce whitespace or unnatural language tokens (e.g., greedy decoding can lead to sequences like `// ///// //` or temporarily disabled code, e.g., `// System.out.println("debug")`). A simple fix is to select the highest-scoring prediction with natural language characters. By conditioning the decoder on at least one alphabetic character per token, one obtains more coherent documentation and rejects fewer samples from the resulting comment. It is possible to construct a more sophisticated natural language filter, however the authors did not explore this idea in great depth.

E TOKENIZATION

Let Σ be any alphabet (in practice, CodeBERT uses UTF-8 although it is possible to encode Chinese by switching to UTF-16). Let $\texttt{dict} \subset \Sigma^* \leftrightarrow \mathbb{Z}$ be a bijection between certain strings over Σ and the integers. Let bpe: $\Sigma^* \to \mathbb{Z}^*$ be an encoder that maps strings Σ^* to a list of integers $\mathbb{Z}^*$. `bpe` is defined as follows: $\texttt{bpe(s)} := \texttt{dict(s[1:p])} \oplus \texttt{bpe(s[p+1:|s|])}$ where $p = max\{i \in \texttt{(1,|p|]} | \texttt{s[1:i]} \in \texttt{dict}\}$, $\oplus$ denotes list concatenation and `s[a:b]` denotes the substring of `s` between indices a and b using 1-based indexing. `dict` has the following property: $\forall s \in \Sigma^*, \texttt{bpe(s)} = [i_1, i_2, \ldots, i_n]$ implies $\texttt{dict}^{-1}(i_1) \oplus \texttt{dict}^{-1}(i_2) \oplus \texttt{dict}^{-1}(\ldots) \oplus \texttt{dict}^{-1}(i_n) = \texttt{s}$, i.e., `bpe` is a lossless compression scheme. Furthermore, dict typically has the property that $\mathbb{E}[|\texttt{bpe(s)}|] \ll \mathbb{E}[|\texttt{s}|]$ over $\texttt{s} \in L \subset \Sigma^*$ where L is a language (e.g., NL or PL) in Σ^*, n.b., this is untrue if Σ^* due to the pigeonhole principle.