A Static Evaluation of Code Completion by Large Language Models

Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, Bing Xiang
AWS AI Labs
{dhantian, kuvrun, tiayuche, zijwan, robkwiat, xiaopel
mkraman, rabaisha, parmib, sudipta, drot, bxiang}@amazon.com

Abstract

Large language models trained on code have shown great potential to increase productivity of software developers. Several execution-based benchmarks have been proposed to evaluate functional correctness of model-generated code on simple programming problems. Nevertheless, it is expensive to perform the same evaluation on complex real-world projects considering the execution cost. On the contrary, static analysis tools such as linters, which can detect errors without running the program, haven't been well explored for evaluating code generation models. In this work, we propose a static evaluation framework to quantify static errors in Python code completions, by leveraging Abstract Syntax Trees. Compared with execution-based evaluation, our method is not only more efficient, but also applicable to code in the wild. For experiments, we collect code context from open source repos to generate one million function bodies using public models. Our static analysis reveals that Undefined Name and Unused Variable are the most common errors among others made by language models. Through extensive studies, we also show the impact of sampling temperature, model size, and context on static errors in code completions.

1 Introduction

Automatic code completion by large language models trained on numerous code repositories has demonstrated great potential in accelerating software development. Code assistant services powered by these models provide developers with code suggestions following the current context in real-time. However, it has been shown that about 70% of the suggestions are discarded by users in a recent study (Ziegler et al., 2022). Even worse, misleading recommendations can lead to failure in completing programming tasks (Vaithilingam et al., 2022). Therefore, it is important to understand the weakness of current code generation models through comprehensive evaluation and analysis.

```
import torch

class Model(torch.nn.Module):

    def __init__(self):
        """Define layers and parameters."""
        super(Model, self).__init__()

        hidden_dim=200
        self.linear1 = torch.nn.Linear(100, 200)
        self.activation = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(200, 10)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):                          Context
        """Define forward computation."""
        x = self.linear1(x)
        x = self.activation(x)                     Completion
        x = self.linear2(x)
        x = F.softmax(x)
        return x
```

Figure 1: A function completion example, with an Unused Variable error (gray) in context, and an Undefined Name error (red) in completion.

Recently, execution-based evaluation has become increasingly popular, where model-generated code is executed with unit tests to check functional correctness. Several benchmarks have been proposed along this direction, such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), MBXP (Athiwaratkun et al., 2022), CodeContests (Li et al., 2022), and DS-1000 (Lai et al., 2022). Although these benchmarks are highly reliable and accurate, they only focus on well-defined algorithmic and data science problems, which do not reflect the need in general software development. Running execution-based evaluation with real-world codebases is, however, prohibitively expensive because each project requires a different setup and the computation cost is potentially unbounded.

In contrast to the execution-based approach, *static program analysis* (or *static analysis*) can analyze programs without executing them. Although static analysis is usually unable to determine functional correctness, it covers a large collection of

static error types, such as undefined names or unused variables that are illustrated in Figure 1. More importantly, the analysis can be very fast and does not require any project specific environment setup, which allows us to evaluate model completions for complex real-world code at large scale. Static analysis tools such as linters have been widely used, for example in code editors, to examine human-written code, but their value in evaluating code generation models has not been well explored yet.

In this work, we propose a static evaluation framework for Python language. Code snippets are first parsed into Abstract Syntax Trees (ASTs) and then analyzed by Pyflakes[1], a popular static analysis tool for Python. To simulate real-world use cases of auto completion, we collect code from public Github repositories to build a function completion dataset of 100K problems. In each problem, we randomly mask out a function body in a Python file and ask the model to complete it given the preceding context up until the function header. We then evaluate public models by sampling 10 completions for each problem, resulting in one million generations for each model and sampling temperature, which will be examined by our static evaluation pipeline.

During AST parsing, we find most of the errors arise from incomplete generations that hit the max length limit. Otherwise, models of all sizes perform quite well in producing parsable codes. Moving forward, Pyflakes analysis reveals that Undefined Name and Unused Variable are the most prominent static errors in model-generated code. We also observe higher temperatures consistently lead to more errors. Scaling up the model, while able to reduce errors of many types, do not show a clear benefit for preventing undefined names. Through a more fine-grained classification, we find larger models generate fewer undefined variables but more undefined methods, which add up to a mixed result. Finally, we demonstrate that errors in context can lead to errors of the same type in generation, which is likely a consequence of large language models' in context learning capability.

In summary, our main contributions include the following. (1) We propose a static evaluation framework for code completion. (2) Our evaluation on public models reveals common static errors and how they are impacted by various factors such as temperature, model size, and context.

[1] https://github.com/PyCQA/pyflakes

2 Background

Code Generation with Transformers Over recent years, it has become increasingly popular to train Transformer-based language models on source code (Feng et al., 2020; Ahmad et al., 2021; Wang et al., 2021; Lu et al., 2021; Guo et al., 2022) to support software engineering tasks (Iyer et al., 2018; Tufano et al., 2019). In particular, several decoder-only transformer models have been developed to facilitate code generation, such as Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2022), Incoder (Fried et al., 2022), and AlphaCode (Li et al., 2022). These pretrained causal language models can be used to predict the continuation of input code without any finetuning.

Abstract Syntax Tree An Abstract Syntax Tree (a.k.a., AST) is used to represent a source code in a concise tree form. By discarding unnecessary details of the underlying code and its corresponding parsed tree, AST only presents the main structural content of the source code following the language grammar (Aho et al., 2007).

Static Analysis Static analysis is a common way to detect software bugs without executing the program (Ayewah et al., 2008; Chess and McGraw, 2004; Chess and West, 2007; Zheng et al., 2006). Static analyzers tend to detect bugs by analyzing the static code text, its AST, documentation, etc. The users usually need to specify the error patterns and static analyzers use different AST, graph, and path analysis to find those patterns in the code. There are a plethora of static analysis tools and they can detect a wide range of errors depending on the specified patterns (Emanuelsson and Nilsson, 2008). For example, Linter is a popular tool that checks for coding style errors and thus, tries to enforce a coding standard (Van Oort et al., 2021).

3 The Function Completion Dataset

We introduce the *function completion* task, which is one of the most important use cases of auto completion services. Given an input code snippet that ends with a function signature plus an optional docstring, the model is asked to generate the function body. Previous works on code completion (Lu et al., 2021; Svyatkovskiy et al., 2019) have mainly focused on single-line completion. However, a single line is often too short to reveal models' capability in writing syntactically correct code. We believe function, as the fundamental building block in most programming languages, better serves this purpose.

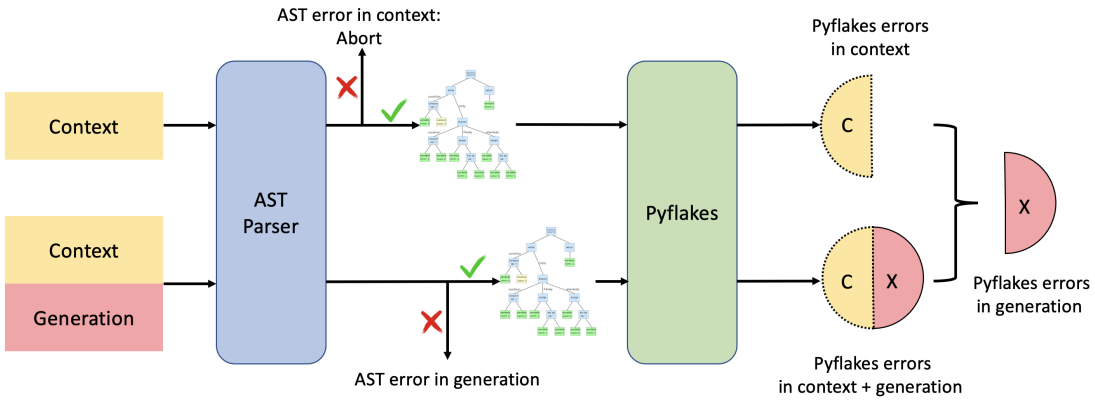

Figure 2: Evaluation pipeline. **Left**: We parse [context] and [context + generation] into ASTs. If [context] is not parsable, we stop without reporting any error on generation. If [context] is parsable, but [context + generation] is not, we report the AST error in generation. **Right**: If both are parsable, we run Pyflakes on the trees, which reports errors in [context] and errors in [context + generation]. Taking the difference gives us errors in generation.

Software developers use code generation models as black-box services on a diverse set of coding projects. To better simulate the real-world scenario, we build an evaluation set by sampling from public Github repositories. Specifically we collected permissively licensed Python code in repositories that were created between April, 2022 and August, 2022. The selection criterion precludes any chronological overlap between our evaluation data and the training data of models to be tested in this work.[2]

The collected Python codes are reformatted as function completion problems. We first use tree-sitter[3] to parse the whole file to identify all the functions. Then a function that contains a docstring is randomly selected. The code from the beginning of the file up until the end of the docstring is used as the context, and the function body is considered as the groundtruth. The rest of the file is discarded. At test time, we prompt the model with the context part as input, and let the model generate the function body. We choose only functions with docstrings so that context is well-defined and model can generate meaningful code completions. We further select test samples whose context length is between 64 and 768 tokens, and groundtruth length is shorter than 256 tokens, to match our model generation setting. Our final evaluation set consists of 100K function completion problems.

4 Static Error Analysis

We propose an evaluation pipeline to detect errors in function completions generated by models, illustrated in Figure 2. Suppose the model generates a completion x given the input context c. We cannot directly analyze x which is partial code without context. Meanwhile, c may also contain errors especially in real-world cases. Therefore, we perform our analysis in two passes. We first check c for any errors in the input that need to be excluded, and then do another pass on the full code (c, x), the concatenation of the context and model completion. Any error that is identified in (c, x) but not in c must arise from x, or in other words, be generated by the model. More specifically, we conduct the following two steps of analysis for Python code.

4.1 AST parsing

In the first step, we parse both c and (c, x) into abstract syntax trees using Python's native *ast* module. If the code is parsable, an AST will be returned. Otherwise, a syntax error is captured. Based on the parsing outcomes, we take the following actions:

1. If c is not parsable, we are unable to conclude any error in generation. Empirically this rarely happens, as we will show in the next section.
2. If c is parsable but (c, x) is not, then we can confirm the reported syntax error is caused by model generation. However, notice that only one error will be returned even if there are multiple, due to the nature of AST parsing.
3. If both c and (c, x) are parsable, there's no AST error in model generation. The ASTs will be used for static analysis in the next step.

4.2 Static analysis with Pyflakes

If both c and (c, x) can be parsed into ASTs, we perform static analysis using Pyflakes. Pyflakes is a static analysis tool that checks a Python source file for errors by examining the AST. One advantage

[2]CodeGen models were trained on data up until Oct, 2021.
[3]https://tree-sitter.github.io/tree-sitter/

is that the analysis does not rely on dependencies of the source file, which is important given the diversity of packages used in real-world code. We run Pyflakes on c and (c, x) to identify errors in context and in full code. Errors that are detected in (c, x) but not in c are considered as introduced by model completion.

5 Experiments

With the proposed pipeline we conduct error analysis for CodeGen models (Nijkamp et al., 2022) on the test set described in Section 3, and present the analysis results.

5.1 Experiment Setup

We evaluate CodeGen-mono models of all sizes, ranging from 350M to 16B. We generate function completions using nucleus sampling with top-p 0.95. Sampling temperature is varied between 0.2 and 0.8 for the 2B model, and fixed to 0.4 for the rest models. We sample 10 generations for each problem, which results in one million code completions for each model and temperature. The maximum generation length is 256 tokens. Generated code completions are then passed to our static evaluation pipeline built with Python 3.8 and Pyflakes 3.0.1. Evaluating one million generations takes only a few hours on a single CPU thread, and can be fully parallelized for acceleration.

5.2 Validation of Model Output

While we mainly focus on static errors in this study, it is also important to validate that the models do generate relevant code. A counter-example would be to generate a single line of "return" for every function signature, which is syntactically correct but not meaningful at all. Towards this end, we calculate the edit similarity between model generation and groundtruth, and compare against Pass@1 from HumanEval (Chen et al., 2021) which is a popular execution-based benchmark to evaluate code generation models. Specifically, for both datasets we generate 10 samples per problem, and report the averaged edit similarity or pass rate over all generations. As shown in Table 1, models of all sizes and temperatures are able to achieve reasonable edit similarity on the function completion dataset, which means the generations are semantically relevant. Moreover, edit similarity and HumanEval Pass@1 both improve as the model scales up, highlighting that model scale is crucial for accurate code generation. Finally, the strong positive correlation between the last two columns shows that edit similarity on the function completion dataset can be used as an alternative metric for model comparison.

Model	Temp	Edit Similarity	HumanEval Pass@1
CodeGen-16B	0.4	72.07	31.83
CodeGen-6B		68.76	26.46
CodeGen-2B		64.83	23.72
CodeGen-350M		56.47	12.62
CodeGen-2B	0.2	65.10	25.06
	0.4	64.83	23.72
	0.6	64.09	21.28
	0.8	62.62	17.56

Table 1: Edit similarity on function completion dataset and Pass@1 on HumanEval, of CodeGen models across different sizes and temperatures. (1) Edit similarity and HumanEval Pass@1 are positively correlated across different settings, which justifies edit similarity can be used as an alternative metric for model evaluation. (2) As expected, larger models have better edit similarity (a proxy to accuracy) on function completion task.

5.3 AST Results

We run AST parsing and find there are only 0.42% cases with unparsable context that need to be discarded. For the rest, we report percentage of generations with AST errors in Table 2. A full list of error types is included in Appendix A. For each type, we also show a code example in Appendix B.

While there are about 7-8% of unparsable generations, most of the parsing errors happen at the end of file (EOF), which means the generated code is incomplete due to the 256 max token limit. Extending generation length may help reduce EOF errors, but will require more computation and increase the perceived latency of the auto-completion service.

On the other hand, non-EOF errors only account for a tiny fraction, usually around 0.1-0.2%, which indicates CodeGen models can generally follow the abstract syntax grammar to produce parsable codes, regardless of model size and temperature.

Finding 1. *Codes generated by models, unless incomplete, are mostly parsable into ASTs, regardless of model size or temperature.*

We also show the top-3 non-EOF error types ranked by frequency, which are **Invalid syntax**, **Print Missing Parentheses**, and **Keyword Argument Repeated**. Notably, the first two categories are often related to Python's interpreter version. To illustrate, Python2-style print like *print "abc"*

Model	Temp	Total	EOF	Non EOF	Invalid Syntax	"print" Missing Parentheses	Keyword Argument Repeated
CodeGen-16B	0.4	7.330%	7.236%	0.094%	0.042%	0.041%	0.004%
CodeGen-6B		7.446%	7.253%	0.193%	0.081%	0.094%	0.006%
CodeGen-2B		7.272%	7.177%	0.095%	0.052%	0.018%	0.008%
CodeGen-350M		8.703%	8.593%	0.110%	0.041%	0.016%	0.028%
CodeGen-2B	0.2	8.067%	7.982%	0.085%	0.045%	0.018%	0.008%
	0.4	7.272%	7.177%	0.095%	0.052%	0.018%	0.008%
	0.6	6.823%	6.713%	0.110%	0.060%	0.020%	0.008%
	0.8	7.496%	7.337%	0.159%	0.085%	0.029%	0.014%

Table 2: Percentages of AST errors across different model sizes and temperatures. We show (1) total AST errors; (2) errors at the end of file (EOF); (3) errors not at EOF; (4) top 3 non-EOF errors. Models generally perform well at AST level except for EOF errors caused by max generation length limit.

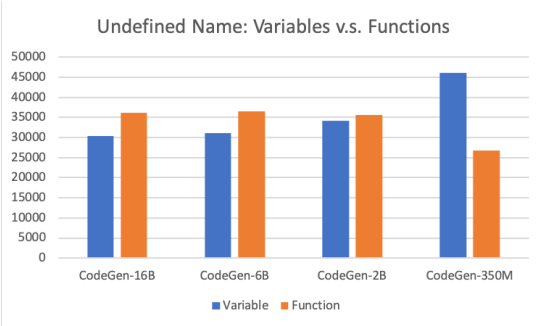

Figure 3: Number of undefined variables versus undefined functions. Larger models generate more undefined functions but fewer undefined variables.

will lead to Print Missing Parentheses in Python3. Another example is that using *async* as a variable name will cause Invalid Syntax because *async* has become a reserved word since Python3.7. Models learn to make such errors from their training data which consists of code written for different Python versions. In many cases, it is difficult for a model to infer the intended interpreter version directly from the limited context. An interesting future direction is to guide models to generate version-compatible code given the target environment.

Finding 2. *Interpreter version mismatch is one of the major reasons for non-EOF AST errors.*

5.4 Pyflakes Results

We present frequencies of top 6 linter errors from Pyflakes in Table 3, with code examples in Appendix B. While Pyflakes also finds other problems in code, most of them are very sparse and thus less important, which we leave to Appendix A. Notice that one code snippet may contain multiple errors. We count each type only once in every test sample.

Among all errors, **Undefined Name** and **Unused Variable** are the most common ones, where the model either calls a variable that is not defined, or defines a variable but never uses it. Closely related are **Unused Import**, **Redefined While Unused** and **Undefined Local**, which can be considered as special cases of the first two. Models also sometimes unnecessarily use f-strings by not giving any placeholder. It is worth pointing out that not all Pyflakes errors will impact execution. In fact among the six types, only Undefined Name and Undefined Local may cause runtime problems. However, all these errors can harm readability and maintenance which are critical for software development. Hence, it is important to address them to improve the quality of auto code completion.

Across sampling temperatures, we observe in every column that more errors are generated under higher temperatures, which is expected because generations in such cases are less confident.

Finding 3. *Higher temperature always leads to more errors of every type.*

The impact of model size on error rate is less consistent though. For Unused Variable, Unused Import, and Undefined Local, error rate does decrease as the model scales up. However, the other three categories do not manifest such correlation. We investigate the underlying reason for this mixed result particularly in the case of Undefined Name. Notice that if an undefined name is a *function call*, it can potentially be defined afterwards outside the current function completion scope. While not guaranteed, the model might be able to fix this error by itself if we allow generating longer code instead of only one function. In contrast, using a *variable* without first defining it is usually a mistake. Even in some rare cases where the variable definition is made up correctly after the usage, such ordering is often less preferred in terms of coding

Model	Temp	Undefined Name	Unused Variable	FString Missing Placeholders	Unused Import	Redefined While Unused	Undefined Local
CodeGen-16B	0.4	4.323%	1.729%	0.135%	0.107%	0.131%	0.047%
CodeGen-6B		4.374%	1.775%	0.089%	0.149%	0.126%	0.055%
CodeGen-2B		4.364%	1.810%	0.147%	0.150%	0.146%	0.065%
CodeGen-350M		4.472%	2.032%	0.151%	0.173%	0.155%	0.095%
CodeGen-2B	0.2	4.206%	1.751%	0.125%	0.139%	0.139%	0.067%
	0.4	4.364%	1.810%	0.147%	0.150%	0.146%	0.065%
	0.6	4.711%	2.000%	0.188%	0.170%	0.159%	0.076%
	0.8	5.377%	2.490%	0.240%	0.247%	0.184%	0.086%

Table 3: Percentages of Pyflakes errors across different model sizes and temperatures. Higher temperatures always lead to more errors in every category. On the other hand, larger models do not necessarily generate fewer errors.

style. In Figure 3, we break down the undefined names into *variables* and *functions*. We find that larger models yield fewer undefined variables, but more undefined functions, which demonstrates that the correlation between error count and model size varies for different errors types.

Finding 4. *While larger models are more accurate code generators (Nijkamp et al., 2022), scaling up model size does not lead to reduction in error counts for all error categories.*

5.5 Correlation with Errors in Context

We further study the correlation between errors in context and in generation. Denote by c the input context, x the model generation, e the error type. We write $e \in c$ to mean c contains an error of type e. For every e,[4] we calculate $\mathrm{P}(e \in x | e \in c)$, the generation error rate when context contains the same type of error(s). We also report the relative ratio $\frac{\mathrm{P}(e \in x | e \in c)}{\mathrm{P}(e \in x | e \notin c)}$ to measure the impact of context. From Table 4, if the model observes errors in context, it is more likely to produce the same type of errors in generation, and the error rate can be amplified by 7∼200 times depending on the type. This is possibly an undesired consequence of the *in-context learning* capability of large language models.

We also calculate $\mathrm{P}(e \in c | e \in x)$ to show how many of the generation errors co-occur with context errors. As indicated by the last column of Table 4, even though context errors can significantly amplify generations errors, the co-occurrences of two do not account for a large fraction. This implies problematic context is not the only factor for problematic generation, and it is often the case for models to produce errors even with correct context.

[4]We omit Unused Import from Table 3 because it is valid to have unused imports in the context that is yet to be completed.

Error type	$\mathrm{P}(e \in x \mid e \in c)$	$\frac{\mathrm{P}(e \in x \mid e \in c)}{\mathrm{P}(e \in x \mid e \notin c)}$	$\mathrm{P}(e \in c \mid e \in x)$
Undefined Name	26.33%	7.80	25.99%
Unused Variable	14.13%	8.45	8.56%
FString Missing Placeholders	20.63%	215.50	35.08%
Redefined While Unused	2.44%	21.16	22.30%
Undefined Local	7.00%	108.68	1.08%

Table 4: Correlation between errors in context and in generation for the 2B model. First two columns indicate errors in context can amplify errors in generation; the last column shows not all generations errors can be attributed to context. Other models have similar results.

Finding 5. *Errors in context generally lead to more errors in generation.*

6 Discussion

We present a static evaluation framework for code completions generated by large language models. By utilizing the proposed framework, we conduct error analysis of CodeGen models on a large scale real-world Python evaluation set. Our experiment reveals common static errors made by pretrained models, as well as their frequency trend across model sizes and sampling temperatures. By pointing out weaknesses of existing models, we hope our study also sheds light on future directions towards more accurate code generation.

There are a few limitations of this study. First, we focus on left-to-right code generation without considering right-side and cross-file context, which can be used to determine broader categories of errors with improved precision. Second, each static analysis tool has its own limitations. Thus, the presented analysis is limited by Pyflakes's accuracy and coverage to detect certain code issues.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2022. Multi-lingual evaluation of code generation models. *CoRR*, abs/2210.14868.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software*, 25(5):22–29.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Brian Chess and Gary McGraw. 2004. Static analysis for security. *IEEE security & privacy*, 2(6):76–79.

Brian Chess and Jacob West. 2007. *Secure programming with static analysis*. Pearson Education.

Pär Emanuelsson and Ulf Nilsson. 2008. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2022. DS-1000: A natural and reliable benchmark for data science code generation. *CoRR*, abs/2211.11501.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*.

Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 2727–2735. ACM.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4).

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA. Association for Computing Machinery.

Bart Van Oort, Luís Cruz, Maurício Aniche, and Arie Van Deursen. 2021. The prevalence of code smells in machine learning projects. In *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*, pages 1–8. IEEE.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE transactions on software engineering*, 32(4):240–253.

Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. 2022. Productivity assessment of neural code completion.

A Full Error Categories

In addition to those discussed in Section 5, we list all error categories that can be detected in model generated code in our experiments, with a minimal frequency of 0.001% by any of the models (i.e. 10 observations out of the total 1 million generations).

AST errors (EOF errors indicated by asterisk):

1. *unexpected EOF while parsing
2. *EOL while scanning string literal
3. *invalid syntax at EOF
4. *EOF while scanning triple-quoted string literal
5. invalid syntax not at EOF
6. missing parentheses in call to "print"
7. keyword argument repeated
8. leading zeros in decimal integer literals are not permitted; use an o prefix for octal integers
9. unmatched ")"
10. cannot assign to function call
11. positional argument follows keyword argument
12. expression cannot contain assignment

Pyflakes issues:

1. undefined name
2. unused variable
3. f-string missing placeholder
4. unused import
5. redefined while unused
6. indentation error
7. import shadowed by loop var
8. raise not implemented
9. invalid print syntax
10. is literal
11. string dot format extra positional argument
12. multi value repeated key literal
13. percent format positional count mismatch
14. tab error
15. string dot format extra named arguments
16. import star not permitted
17. percent format unsupported format character
18. assert tuple
19. percent format extra named arguments

B Examples for Top Error Types

Below we list one code example for each of the error categories shown in Table 2 and 3. Following the definition of function completion task, in every example, context is from the beginning until the end of the docstring of the last function, and model completion is the body of the last function.

```
"""Secondary Structure dataset."""

import numpy as np
from megatron import print_rank_0
from .data import ProteinPredictionAbstractDataset
from .data import build_tokens_paddings_from_text

class SecondaryStructureDataset(ProteinPredictionAbstractDataset):
    def __init__(self,
                 name: str,
                 datapaths,
                 tokenizer,
                 max_seq_length: int):
        super().__init__('secondary_structure', name, datapaths, tokenizer, max_seq_length)


    def build_samples(self, ids, paddings, label, unique_id, seq_len):
        """Convert to numpy and return a sample consumed by the batch producer."""

        # Seperate inputs and labels in lists
        input_tokens = [self.tokenizer.tokenize(seq) for seq in ids]
        input_tokens = [token for seq in input_tokens for token in seq]
        labels = [self.tokenizer.tokenize(seq) for seq in label]
        labels = [label for seq in labels for label in seq]

        # Add special tokens
        input_tokens, labels = self.add_special_tokens(input_tokens, labels)

        # Truncate and pad
        input_tokens, labels, paddings = self.truncate_and_pad(input_tokens, labels,
                                                                self
```

Listing 1: unexpected EOF while parsing (line 31)

```
from pathlib import Path
from wasabi import msg
from .remote_storage import RemoteStorage
from .remote_storage import get_content_hash, get_command_hash
from ..util import load_project_config
from ..util import project_cli, Arg, logger


@project_cli.command("push")
def project_push_cli(
    # fmt: off
    remote: str = Arg("default", help="Name or path of remote storage"),
    project_dir: Path = Arg(Path.cwd(), help="Location of project directory. Defaults to current working directory.", exists=True, file_okay=False),
    # fmt: on
):
    """Persist outputs to a remote storage. You can alias remotes in your project.yml by mapping them to storage paths. A storage can be anything that the smart-open library can upload to, e.g. AWS, Google Cloud Storage, SSH, local directories etc.

    DOCS: https://xxx
    """
    for nlp in load_project_config(project_dir, {"directories": [ANIMAL_TRAIN_DIR]}:
    remote_storage = RemoteStorage.get(remote)
    for command in ["train"]:
        logger.debug(f"Uploading {command} to remote storage '{remote_storage}'")
        path = Path(project_dir) / ANIMAL_TRAIN_DIR / command
        upload_project(remote_storage, path)
```

Listing 2: invalid syntax (line 23)

```
def prod(F, E):
    """Check that the factorization of P-1 is correct. F is the list of
       factors of P-1, E lists the number of occurrences of each factor."""
    M = prod_of_prime_factors(F, E)
    if not all(i == 1 for i in M):
        print "Error in prod"
        print F, E
        return
    P = product(F)
    P_1 = 1
    for i in range(len(F)):
        P_1 *= F[i]**E[i]
    if P != P_1:
        print "Error in prod"
        print F, E
        print P
        print P_1
        return
```

Listing 3: Missing parentheses in call to "print" (line 6)

```
import unittest
from datetime import datetime, timezone

from dateutil.relativedelta import relativedelta

from bot.utils import time


class TimeTests(unittest.TestCase):
    """Test helper functions in bot.utils.time."""

    def test_humanize_delta_handle_unknown_units(self):
        """humanize_delta should be able to handle unknown units, and will not abort."""
        self.assertEqual(
            time.humanize_delta(datetime.utcnow(), datetime.utcnow() - relativedelta(months=1, months=2)),
            "1 month and 2 months"
        )
```

Listing 4: keyword argument repeated (line 15)

```
"""
This program will continually ask our user to give a number
and will calculate the factorial result of the number and print it on the console.

The program ends when the user enter the EXIT number.
"""

EXIT = -100


def main():
  """
  This program will calculate the factorial result according to the number an user
  inputs.
  """
  print('<<< Welcome to the Factorial Calculator! >>>')
  num = int(input('Enter a number: '))
  print('The factorial of {} is {}.'.format(num, factorial(num)))
  if num == EXIT:
    print('\n<<< Thank you for using the Factorial Calculator. >>>')
  else:
    main()
```

Listing 5: undefined name "factorial" (line 18)

```python
def check(full_path, encoding):
    assert type(full_path) == str, f'\'full_path\' is of {type(full_path)}. Only type \'str\' is acceptable.'
    assert full_path != "", "\'full_path\' is empty."
    assert type(encoding) == str, f'\'full_path\' is of {type(encoding)}. Only type \'str\' is acceptable.'
    assert encoding != "", "\'encoding\' is empty."

def file_read(full_path: str, encoding = "utf8"):
    '''
    Author: xxx

        Reads file at "full_path" and returns its data in a list.
    '''

    check(full_path, encoding)
    encoding_check = encoding
    full_path = full_path.strip()
    f = open(full_path, "r", encoding = encoding)
    lines = f.readlines()
    f.close()
    lines = [line.replace("\n", "") for line in lines]
    return lines
```

Listing 6: local variable "encoding_check" is assigned to but never used (line 15)

```python
import os
import json

from convinse.library.utils import store_json_with_mkdir, get_logger


class HeterogeneousAnswering:
    def __init__(self, config):
        """Initialize HA module."""
        self.config = config
        self.logger = get_logger(__name__, config)

    def train(self, sources=["kb", "text", "table", "info"]):
        """ Method used in case no training required for HA phase. """
        self.logger.info(f"No need to train.")
        pass
```

Listing 7: f-string is missing placeholders (line 15)

```python
import os
import urllib.parse
import sqlite3

SQL = """
SELECT p.ZAUTHOR, p.ZTITLE, e.ZTITLE, e.ZASSETURL, e.ZPUBDATE
from ZMTEPISODE e
join ZMTPODCAST p
    on e.ZPODCASTUUID = p.ZUUID
where ZASSETURL NOTNULL;
"""


def check_imports():
    ''' Prompts for password to install dependencies, if needed '''
    import os, importlib, importlib.util
    import urllib.parse

    # Check for dependency installs
    # Can be done more simply, but this way I can avoid importing anything from zmodel,
    # which is nice since I can see what's going on.
    for k, v in DEPS.items():
        try:
            importlib.import_module(k)
        except ImportError as e:
            importlib.util.find_spec(k)
            if importlib.util.find_spec(k) is None:
                os.system(f'pip install {v}')
```

Listing 8: "urllib.parse" imported but unused (line 17)

```python
import kfp.deprecated as kfp
from kfp.deprecated import components, dsl, compiler

def get_run_info(run_id: str):
    """Example of getting run info for current pipeline run."""
    import kfp.dsl as dsl
    client = kfp.Client()
    run = client.run_details(run_id)
    print(f"Run details:\n{run}")
    print(f"Pipeline details:\n{run.pipeline_runtime}")
```

Listing 9: redefinition of unused "dsl" from line 2 (line 6)

```python
"""Check for nonlocal and used-before-assignment"""
# pylint: disable=missing-docstring, unused-variable, no-init, too-few-public-methods

__revision__ = 0

def test_ok():
    """ uses nonlocal """
    cnt = 1
    def wrap():
        nonlocal cnt
        cnt = cnt + 1
    wrap()

def test_fail():
    """ doesn't use nonlocal """
    cnt = 1
    def wrap():
        cnt = cnt + 1 # [used-before-assignment]
    wrap()
```

Listing 10: local variable "cnt" defined in enclosing scope on line 16 referenced before assignment (line 18)