Leveraging static analysis for evaluating code-generation models

Sai Anuroop Kesanapalli, Indrani Panchangam, Abhishek Anand, Vishesh Mittal, Kayvan Shah
[kesanapa, panchang, anandabh, visheshr, kpshah] @usc.edu
Group 37
University of Southern California

Abstract

Code generation has recently become a popular application of Large Language Models (LLMs). ChatGPT (Brown et al., 2020), GitHub CoPilot, Code Llama (Rozière et al., 2023), Bard are some popular tools that aim to enhance developer productivity and shorten development time. Often, the code generated by these tools remains buggy, rendering them counterproductive. Existing methods that address this issue predominantly focus on runtime analysis, which is resource-intensive, while limited research has been conducted in the realm of static analysis, particularly for a narrow selection of programming languages. We aim to impart knowledge of static errors to the baseline code generation model, which potentially helps in improving code generation. Toward this goal, we propose two methods. First, a feedback pipeline that integrates errors from static code analysis as feedback to improve the baseline model. Second, we fine-tune the model to enhance its ability to directly generate code with fewer errors. Our findings demonstrate the effectiveness of both strategies regarding reducing frequencies of observed static errors.

1 Introduction

Large Language Models (LLMs) like ChatGPT (Brown et al., 2020), GitHub Copilot, and Code Llama (Rozière et al., 2023) have reshaped the coding paradigm with real-time code generation. However, reliability concerns persist. Studies (Ziegler et al., 2022) indicate that 70% of LLM-generated code is discarded by developers, suggesting potential issues.

(Le et al., 2023) propose a framework for refined evaluation of neural code generations. Notably, this work utilizes a static analyzer in their workflow akin to ours, however, we note that there is no feedback as proposed in Section 2. HumanEval dataset (Chen et al., 2021) is a pioneering work in this code-completion benchmarking domain where code is evaluated using unit test cases. However, as (Ding et al., 2023) mention, their framework, being runtime-based evaluation in nature, is expensive to be implemented at scale. We further note here that this work is restricted to Python code completion, whereas we also expand the static analysis framework to C++. (Odena et al., 2021) propose two benchmarks, Mostly Basic Programming Problems (MBPP) and MathQA-Python. The former is a collection of Python programs requiring numeric manipulations, whereas the latter is a Pythonized version of a popular text-to-code generation dataset.

As a common denominator, existing methods predominantly favor runtime analysis. However, the critical need for recognizing and understanding static errors in code generated by the model, which is pivotal for refining models, is often neglected. This oversight hampers the iterative improvement of models for more effective and accurate code generation. The novelty of our work thus lies in the methodology of providing feedback to the model laced with static analysis and further fine-tuning it.

Section 2 outlines the methodology, Section 3 details experiments, Section 4 presents results and analysis, Section 5 offers conclusive insights, Section 6 explores future research avenues, and Section 7 acknowledges individual contributions.

2 Methodology

Our methodology aims to improve the effectiveness of the code generation model through a) the integration of a feedback mechanism (Section 2.1) that utilizes errors observed (e) in the static analysis to minimize these in the code generated post feedback, b) fine-tuning a pre-trained model using Direct Preference Optimization (DPO) (Rafailov et al., 2023) for this task with the objective of optimizing the model for generating code with less static errors (Section 2.2).

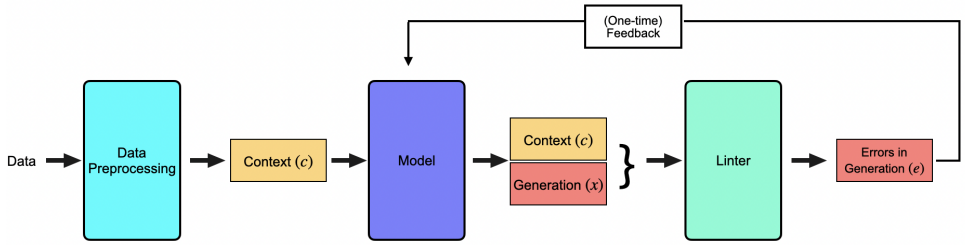

Figure 1: Feedback Pipeline

The code-base is made available on GitHub[1].

2.1 Feedback Pipeline

We incorporate automated feedback using linters (static code analyzers) to enhance error detection and enable model improvement. Figure 1 provides a visualization of this method, a multi-stage feedback pipeline for code generation. The context (c) generated from the pre-processing stage (Section 3.2) as part of a prompt, along with text from the dataset, is provided to the model for generating code (x) (Section 3.3). Linters run on (c, x) to detect errors (e). Errors identified are then fed back in natural language as feedback in prompts to the model for improving code generation. This method systematically identifies and minimizes errors in the code generation process through automated feedback without human intervention and any need for code execution. The importance of this process lies in its ability to offer precise insights into areas where the code generation model may exhibit shortcomings and enable targeted corrections with precise type and location of errors. Although complete automation of the feedback pipeline was considered a possibility during the early stages of this project, given the time constraints, we have automated the post-processing stage as a proof of concept.

2.2 Fine-Tuning

This approach aims to enhance the baseline model's initial accuracy in making correct predictions without relying on subsequent feedback corrections. This is achieved through fine-tuning the baseline model for code generation, specifically employing the DPO method. We utilize prompt construction with the same process as the first stage of our feedback pipeline (Section 2.1). For the preferred response in fine-tuning, we utilize code given as a gold label in the dataset, and for the rejected response, the code generated by the model with simple prompting containing static errors is used as the rejected response. This method helps us optimize the model directly to generate quality code with fewer static errors.

2.3 Static Analysis

We utilize static analysis in both methods to evaluate and improve code generation. A linter is utilized as a static analysis tool that examines a source file for errors by analyzing the Abstract Syntax Tree (AST). An inherent advantage lies in its independence from program execution and environment setup for a particular language. This benefit proves crucial in real-world code scenarios where such tools can be run with minimal effort for large code bases without human intervention in environment setup and unit test development. In both methods, linters are used for automated evaluation of code quality generated by the model, while in the first method, we also utilize errors they report as feedback in the pipeline.

3 Experiments

3.1 Datasets and Language Models

We have utilized XLCoST (Zhu et al., 2022) for the code completion task, which is a parallel dataset that encompasses solutions for problems related to data structures and algorithms in six languages: C++, Java, Python, PHP, C, and C#. Our experiment focuses specifically on C++ and Python program-level samples. Our baseline model is CodeLlama-7b-Instruct-hf. This model is configured with a temperature value of 0.1, an output token maximum length of 512, and a probability threshold of 0.9 for code generation. The model is used with full precision in the feedback pipeline, while for fine-tuning, we utilize 4 bit-QLoRA (Quantized Low Rank Adaptation) for loading the model and training it on T4 GPU (16GB).

[1] https://github.com/ksanu1998/static_analysis_codegen_llms

3.2 Data Preprocessing

Each sample in the XLCoST contains two fields – text and code. The context (c) is constructed by capturing all content until the initial function signature, excluding `main`, within the code field. Concatenating this contextual data with the text field produces a prompt that is fed into the model.

3.3 Implementation

We have used GPU-P100 on Kaggle and T4 on Google Colaboratory for our experiments. Although XLCoST has $11.2K$ data samples corresponding to our task, our evaluation of the feedback pipeline and the fine-tuned model was restricted to only 300 samples because of the extended processing time. For fine-tuning we use 240 samples from the dataset that excluded the evaluation samples used in evaluation.

3.3.1 Feedback Pipeline

The prompt constructed during pre-processing is used for generating code. Subsequently, the responses undergo post-processing to address certain considerations, such as extracting code and discarding any textual descriptions provided by the model. Following post-processing, the data is subjected to linting through cppcheck for C++ and flake8 for Python. Errors reported by linters are used as feedback to the model. Post feedback and regeneration, linters are again used to evaluate the responses.

3.3.2 Fine-Tuning

We randomly select 1000 samples from the dataset, and their respective prompt is used for code generation. Generated code and code from the dataset are analyzed through linters, and 240 samples are selected for fine-tuning the model with an aggregated count of more than 100 static errors in the model-generated code compared to the original code. These samples are used for DPO fine-tuning. A fine-tuned model and pre-trained model in the 4-bit quantized form are used to generate responses for our evaluation set of 300 samples, and the responses are analyzed using linters.

4 Results and Analysis

In this section, we first present the results of our experiments as a series of tables[2].

[2]Preliminary results and fine-grained statistics have been redacted from this report for brevity, and can be referred to in our presentation linked here.

4.1 Results: Feedback Pipeline

Table 1 summarizes the coarse-grained statistics of Python samples generated using the model with feedback, as described in Section 2.1. Coarse-grained statistics present the overall difference in the frequencies of errors before and after feedback. Fine-grained statistics describe the prevalence of the same error types[3] for a given sample despite the feedback, in terms of their frequencies, summed across all the samples (hence the name, fine-grained). Table 2 presents the results for C++ samples. We can immediately infer from these tables that feedback laced with static analysis helps improve the code generated by the baseline model, thus demonstrating the model's efficacy with feedback.

Freq. before / after Feedback	Difference	Error Code
290 / 0	290 ↓	W292
48 / 40	8 ↓	F401
11 / 4	7 ↓	F821
10 / 6	4 ↓	E999
3 / 0	3 ↓	E231
17 / 16	1 ↓	E501
4 / 4	0	E741
2 / 2	0	F403
1 / 1	0	F523
2 / 2	0	F821
21 / 22	1 ↑	E225

Table 1: Coarse-grained statistics | Feedback Pipeline | Python

Freq. before / after Feedback	Difference	Defect ID
65 / 14	51 ↓	`constParameter`
159 / 114	45 ↓	`unusedFunction`
46 / 16	30 ↓	`syntaxError`
21 / 7	14 ↓	`passedByValue`
5 / 4	1 ↓	`variableScope`
3 / 3	0	`negativeIndex`
1 / 1	0	`arrayIndexOutOfBounds`
4 / 6	2 ↑	`shadowVariable`
11 / 13	2 ↑	`unreadVariable`

Table 2: Coarse-grained statistics | Feedback Pipeline | C++

[3]PEP8 Error Codes

4.2 Results: Fine-Tuning

Fine-tuning, as described in detail in Section 2.2, was done using 240 Python samples, employing 4-bit quantization scheme and adapters to fit the model in memory. Table 3 lists results obtained using model fine-tuning on those 300 samples used earlier in feedback pipeline analysis.

Freq. before / after Fine-tuning	Difference	Error Code
212 / 95	117 ↓	E999
86 / 0	86 ↓	E303
41 / 20	21 ↓	F401
12 / 3	9 ↓	E501
3 / 0	3 ↓	E302
7 / 5	2 ↓	E225
2 / 0	2 ↓	F841
1 / 0	1 ↓	E305
2 / 1	1 ↓	F403
1 / 3	2 ↑	E741
0 / 2	2 ↑	E703
0 / 3	3 ↑	E711
0 / 3	3 ↑	F405
2 / 7	5 ↑	F821
0 / 6	6 ↑	E128

Table 3: Coarse-grained statistics | Fine-tuned Model | Python

4.3 Analysis

Results from the preliminary static analysis of C++ samples bolster our hypotheses mentioned during the presentation and proposal that non-EOF errors such as `syntaxError` may be more prevalent in C++ and that OOP-based errors such as `shadowFunction` may be observed. Errors like `undefined name` and `unused variables`, as highlighted in (Ding et al., 2023), were also identified in the reproduced results on Python samples. Our feedback pipeline seems to yield promising results as the static errors post feedback have reduced in the Python samples by $\sim \mathbf{76}\%$, based on the coarse-grained statistics we obtained (Table 1). For C++ samples, this figure is $\sim \mathbf{43}\%$ (Table 2). Results on Python samples suggest that the repetition errors constitute only $\sim \mathbf{20}\%$ of the errors before feedback, implying that repetitions do not often occur post-feedback. For C++ samples, this figure is $\sim \mathbf{45}\%$. Furthermore, these results indicate that the feedback works better for Python samples than C++ and is expected as C++ is a stricter language. Fine-tuning the baseline model also proved beneficial as the static errors post-fine-tuning were reduced in the Python samples by $\sim \mathbf{60}\%$, based on the coarse-grained statistics we obtained (Table 3). We leave the results of fine-tuning for C++ samples as future work.

5 Conclusion

We explored the idea of evaluating the code generated by models by using static analysis and then improving the models using the error reports thus obtained. The unavailability of publicly accessible code from the framework developed by (Ding et al., 2023) necessitated building the entire framework from scratch. Towards this goal, we first reproduced some results of (Ding et al., 2023) on Python, then extended a similar analysis to C++, incorporated the feedback loop, which demonstrated a noticeable improvement of the code generated by the model, and then fine-tuned the model aimed at enhancing its performance by learning from and mitigating common errors. Our work thus pioneers the concept of improvement of code generation models through static analysis feedback. We successfully implemented everything we had planned to, as stated in the project proposal.

6 Future Work

Extending our research involves broadening the analysis to include diverse programming languages. Given the limitations of individual linters in error identification, exploring this task with various linters could yield valuable insights. While the model was chosen for its state-of-the-art token-level code completion capabilities, further exploration may consider different line and block-level models. Expanding the token limit and re-examining evaluation statistics becomes especially critical when lengthy code segments are imperative for a comprehensive model performance evaluation. Continuous training strategies for language models are essential for adapting to evolving coding practices and emerging programming languages, ensuring their relevance and effectiveness over time. Different methods to incorporate static analysis into feedback can be explored, we have presented one such method. Furthermore, better performance analysis metrics can be devised.

Finally, we acknowledge the guidance received from the course staff of CSCI 544, Fall 2023, at the University of Southern California.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

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

Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023. A static evaluation of code completion by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 347–360, Toronto, Canada. Association for Computational Linguistics.

Kim Tuyen Le, Gabriel Rashidi, and Artur Andrzejak. 2023. A methodology for refined evaluation of neural code completion approaches. *Data Mining and Knowledge Discovery*, 37(1):167–204.

Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program synthesis with large language models. In *n/a*, page n/a, n/a. N/a.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence.

Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29.

7 Contributions

- AA & KS – Pre-processing stage of the feedback pipeline
- AA & IP – Code-completion stage of the feedback pipeline
- IP & SAK – Post-processing stage of the feedback pipeline
- SAK & VM – Evaluation stage of the feedback pipeline
- SAK & KS – Running feedback pipeline and analysing results
- All – Paper presentation, status report, and project presentation
- IP & AA – Running model fine-tuning and analysing results
- KS & VM – Automation of the post-processing stage of the feedback pipeline
- KS & AA – Code refactoring, re-organization, and documentation
- IP & SAK – Final report