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

1 Project Background

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 predictions. However, reliability concerns persist. Studies (Ziegler et al., 2022) indicate that 70% of LLM-generated code is discarded by developers, suggesting potential issues. While run-time benchmarks, such as HumanEval, are resource-intensive, static analysis, as discussed by (Ding et al., 2023), offers an efficient alternative. Our project performs static analysis on code samples derived from XLCoST (Zhu et al., 2022) dataset and completed using CodeLlama-7b-hf model, focusing on C++ and Python languages. By reintegrating these errors, we aim to enhance model performance without extensive re-training. This report details our progress (Section 2), challenges (Section 3), and mitigation plans (Section 4).

2 Project Advancements

2.1 Experiment Setup

A preliminary code review was conducted in accordance with Target 1.2.1 as outlined in the proposal. The model pipeline comprises four stages: **pre-processing**, **code-completion**, **post-processing**, and **evaluation**. In our current implementation[1], the initial two phases are written in Python notebooks, the third one is automated for C++ samples using a Python notebook and handled manually for Python samples. The final phase is scripted in `.sh` files, internally invoking the linters (cppcheck and flake8).

We utilized CodeLlama-7b-hf for code completion. The following steps are performed as part of the pipeline's **pre-processing** phase:

1. Selecting samples where the initial function is not `main()`.
2. The entire content up until the first function signature in the code field of XLCoST serves as context.
3. The context and the `text` field are concatenated to create a prompt, which is then fed into the model for **code-completion**. This process is done for 100 examples of Python and C++ respectively.

In the pipeline, **post-processing** phase is incorporated for the reasons outlined in Section 3.2. The refined output is subsequently directed to the static code analysis phase for thorough evaluation. The entire pipeline was written from scratch as the implementation of (Ding et al., 2023) was not made public.

We utilized a pre-trained model (CodeLlama-7b-hf) for our task. The temperature value is set to 0.1, and the maximum token length for output is 512. The probability threshold for the next token prediction is set at 0.9. While quantized versions of the models are more manageable due to their smaller size and faster generation time, for our current and future analyses to be fair across languages and models, we persist with full-size models for our experiments.

Our runs were executed on T4 GPUs in the cloud, accessible through Google Colaboratory.

2.2 Results and Observations

2.2.1 Static C++ code analysis using cppcheck

An evaluation was conducted on 100 generated C++ code samples. Table 1 lists the Defect IDs and their frequency as reported by cppcheck.
Notably, `syntaxError` emerged as the most common error type, indicating unclosed { braces. Moreover, observed errors, such as `negativeIndex`, may be attributed to the model's extensive training on Python code samples.

Furthermore, specific errors critical to

[1] https://github.com/ksanu1998/NLP_Group37

C++ and OOP, including `passedByValue` and `shadowFunction` respectively, were identified.

This preliminary analysis bolsters 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.

2.2.2 Static Python code analysis using flake8

Next, an evaluation was conducted on 100 generated Python code samples.

Frequency	Defect ID
54	`syntaxError`
2, 2, 2	`shadowFunction`, `unreadVariable`, `variableScope`
1, 1, 1, 1, 1	`arrayIndexOutOfBoundsCond`, `constParameter`, `missingInclude`, `negativeIndex`, `passedByValue`

Table 1: Static analysis of generated C++ code

Frequency	Error Code(s)	Description
1301, 189, 15, 7, 6	W291, W293, E225, E275, E712	white-space related
56, 32, 53, 26	E305, E302, W292, W391	blank/newline related
40	E501	line too long
61	W191	indentation contains tabs
17	F821	undefined name
12	F401	imported, unused
11	E999	SyntaxError – cannot generate AST
6	E712	`if` condition related
3	F841	assigned, not used
2	F405	Undefined fuction
2, 1	E741, F403	Miscellaneous

Table 2: Static analysis of generated Python code

flake8 incorporates Pyflakes along with a rich style-check based on PEP8 style-guide. From Table 2, it is evident that most of the reported errors are white-space, blank/newline, and line-length related. Notable ones like indentation errors that are particularly crucial in the context of Python, as well as other critical errors such as ambiguous variable names, irregular conditions in `if` statements, and glaringly bad syntax errors (E999) that make AST un-parsable, were observed.

Furthermore, errors like undefined name and unused variables, as highlighted in (Ding et al., 2023) were also identified. Unlike (Ding et al., 2023), we refrain from reporting frequencies in % as it does not reflect the absolute number of errors observed.

3 Risks and Challenges

1. Running CodeLlama-7b-hf on T4 GPUs through Google Colaboratory was highly time-consuming, with each instance taking about 100 seconds. Although XLCoST has $11.2K$ data samples that correspond to our task, our evaluation was restricted to only 100 samples because of the extended processing time.
2. Since CodeLlama-7b-hf is not instruction-tuned for chat, the response generated, at times, has duplicate/multiple definitions for some functions.
3. Accurate data cleaning in Python was deemed essential but proved tricky due to the risk of introducing subtle errors, especially in indentation. These mistakes could only result in an indentation error caught by the linter, preventing the actual model errors from being unearthed.

4 Plans to Mitigate

The strategies for mitigating each risk/challenge identified in Section 3 above are enlisted below:

1. We intend to expand our analysis over more data samples using AWS or CARC for Target 1.2.2.
2. This challenge was addressed through the introduction of the post-processing stage in the pipeline. In this phase, duplicate/multiple function definitions were eliminated, retaining only the first definition, and partial code completions resulting from reaching the maximum token length were preserved. Additionally, strategies such as adjusting the temperature or implementing a repetition penalty in code generation are being explored to minimize the issue further.
3. We ensured that the observed errors were genuine and not artifacts of our evaluation pipeline.

5 Individual Contributions

- AA & KS – Pre-processing stage of the pipeline
- AA & IP – Code-completion stage of the pipeline
- IP & SAK – Post-processing stage of the pipeline
- SAK & VM – Evaluation stage of the pipeline
- All – Status report

Roughly equal distribution of workload is planned for the remainder of the project.

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.

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.

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.