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

In this proposal document, we first discuss project domain and goals (Section 1). Next, we present some related work (Section 2), followed by a discussion on the datasets proposed to be used and their processing (Section 3), and comment about the challenges we expect to face in this work (Section 4).

1 Project Domain & Goals

1.1 Project Domain

Code generation has become a popular application of Large Language Models (LLMs) in recent times. ChatGPT (Brown et al., 2020), GitHub CoPilot , Code Llama (Rozière et al., 2023), Bard are such tools that aim to enhance developer productivity and shorten development time. Although the aim of these tools sounds noble, they are far from being perfect, as a result of which the code generated by them remains buggy. This often necessitates the developers to spend more time in analyzing and de-bugging the code generated by these tools, contrary to their spirit. As per (Ziegler et al., 2022), an alarming 70% of model generated code is discarded by the developers. It is crucial to discern the errors made by the model, as this knowledge is pivotal for refining them and improving their performance. This necessity underscores the importance of evolving metrics and evaluation methods. While execution-based benchmarks like HumanEval (Chen et al., 2021), and static analysis tools proposed by (Ding et al., 2023) exist, there is a room for exploration due to the limitations associated with these approaches. Among these, static code analysis stands out as a reliable choice. Unlike run-time analysis, it does not require code execution, ensuring rapid feedback without compromising resources. This efficiency, coupled with its comprehensive error detection capabilities, posits static analysis as a robust and cost-effective solution in our pursuit of enhancing code generation models. We thus adopt this approach for evaluating the code generated by LLMs, bringing it under the umbrella of Natural Language Processing (NLP). Our proposed work can also be categorized under quality assurance, owing to its evaluation nature.

1.2 Goals

Based on our analysis of (Ding et al., 2023) and other relevant works in this domain (Section 2), we have identified the following as our *targets* for this project.

1.2.1 Broadening horizons: Expanding the scope using a different language, linter, and model

We propose to carry forward the work done by (Ding et al., 2023) with a different language [C++], linter [Cppcheck], and model [Code Llama] (Rozière et al., 2023). This helps us explore the extensibility of the static evaluation framework considered and critically analyze some shortcomings of the same. Choice of C++ is guided purely by its use as an Object Oriented Programming language and syntactic complexity over Python, and its widespread use (Bissyandé et al., 2013) in lower system-level code and software applications. Cppcheck is chosen due to its ease of use, wide coverage of errors and incorporation of rule reflecting best coding practices.

1.2.2 Code generation improvement through feedback to model and fine-tuning

Next, we aim to explore the possibility of improving the code generation of these models through two approaches. First, we propose to feed the errors (obtained by the static evaluation framework proposed by (Ding et al., 2023) and re-developed by us in Target 1.2.1 above) back to the model, in the form of engineered prompts and re-evaluate the generated code. Successful implementation of this approach would showcase a practical application of the static evaluation framework, enabling

language models to learn *in-context* without requiring re-training. In the second approach, we propose to fine-tune the code generation model by re-training it using additional samples enriched with the knowledge obtained from static analysis. This, we believe, reinforces elimination of common errors. We will then evaluate the fine-tuned model performance on HumanEval (Chen et al., 2021) to compare its performance to the non fine-tuned models.

2 Related Work

(Le et al., 2023) propose a framework for refined evaluation of neural code completions. They are motivated by the observation that most of the existing works report accuracy of such models as cumulated metrics averaged over various types of code tokens, which they suggest is not the best way forward. To address this, they provide Code Token Type Taxonomy (CT3), an approach of categorizing tokens into multiple dimensions, and computing accuracies along each of these dimensions. Notably, this work utilizes a static analyzer in their workflow akin to our proposal, however, we observe that there is no feedback loop as we propose in Target 1.2.2. 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 in Target 1.2.1 we propose to 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 Python-ized version of a popular text-to-code generation dataset. As a common denominator, though the aforementioned works are carried-out in great detail, they are specific to Python and do not incorporate the feedback loop that we propose in Target 1.2.2.

3 Datasets

Given the relatively small size of the C++ program-level samples in the XLCoST (Zhu et al., 2022) dataset, ranging from 49 to 5.46k characters with a noticeable skew towards shorter lengths, they present an excellent fit for static code evaluation in Target 1.2.1. Each sample contains two fields - text and code. In order to use this sample for Target 1.2.1, we need to include the contents of text field as a docstring in the corresponding code field, for each sample.
For Target 1.2.2, our approach involves utilizing HumanEval (Chen et al., 2021). This dataset is appropriate for the purpose as it consists of 164 hand-written problems belonging to various genres such as language comprehension, reasoning, algorithms, and common mathematics, written in Python.

For both Targets 1.2.1 and 1.2.2, the code samples need to be randomly masked leaving out the function signature and docstring, as done in the (Ding et al., 2023).

4 Technical Challenges

The unavailability of publicly accessible code from the framework developed by (Ding et al., 2023) necessitates building the entire framework from scratch for Targets 1.2.1 and 1.2.2. Significant effort will be devoted to writing scripts that compile the model, run forward pass on evaluation set to get the generated code, build and patch AST to the linter, and generate statistics from both the AST and linter.

For the first approach in Target 1.2.2, hand-crafted prompts need to be fed into the model one sample at a time, requiring time. This process can be automated using tools such as LangChain, if feasible. For the second approach in Target 1.2.2, fine-tuning the model can prove to be a significant challenge in terms of compute resources, requiring access to CARC in this regard. All the challenges listed above go beyond the scope of the material discussed in the class, as it requires significant scripting, knowledge of AST and linter. Furthermore, prompt-engineering will be one of the valuable lessons to be learnt from this project while executing Target 1.2.2, in the scope of applied NLP. For Target 1.2.1, our objective is to produce tables similar to those in (Ding et al., 2023), as part of our evaluation framework. The efficacy of our approaches for Target 1.2.2 can be determined by the error rates of the feedback-enhanced and fine-tuned models.

References

Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. 2013. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*, pages 303–312. IEEE.

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.

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.

5 Division of Work

- SAK & IP - Dataset preparation (Target 1.2.1)
- AA & KS - Dataset preparation (Target 1.2.2)
- VM & SAK - Scripting (Target 1.2.1)
- IP & AA - Scripting (1st approach of Target 1.2.2)
- KS & VM - Scripting (2nd approach of Target 1.2.2)
- SAK & IP - Executing (1st approach of Target 1.2.2)
- AA & KS - Executing (2nd approach of Target 1.2.2)
- All - Status Report
- All - Final Report