Imperfect Code Generation: Uncovering Weaknesses in Automatic Code Generation by Large Language Models

Xiaoli Lian, Shuaisong Wang, Jieping Ma, Xin Tan, Fang Liu, Lin Shi, Li Zhang
SKLSDE, Beihang University, China
{lianxiaoli,littletree,19373733,xintan,fangliu,shilin,lily}@buaa.edu.cn

Cuiyun Gao
Harbin Institute of Technology, China
gaocuiyun@hit.edu.cn

Abstract

The task of code generation has received significant attention in recent years, especially when the pre-trained large language models (LLMs) for code have consistently achieved state-of-the-art performance. However, there is currently a lack of a comprehensive weakness taxonomy in the field, uncovering weaknesses in automatic code generation by LLMs. This may lead the community to invest excessive efforts into well-known hotspots while neglecting many crucial yet unrecognized issues that deserve more attention. To bridge this gap, we conduct a systematic study on analyzing the weaknesses based on three state-of-the-art LLMs across three widely-used code generation datasets. Our study identifies eight types of weaknesses and assesses their prevalence across each LLM and dataset, aiming to inform and shape the trajectory of future research in the domain.

ACM Reference Format:
Xiaoli Lian, Shuaisong Wang, Jieping Ma, Xin Tan, Fang Liu, Lin Shi, Li Zhang and Cuiyun Gao. 2024. Imperfect Code Generation: Uncovering Weaknesses in Automatic Code Generation by Large Language Models. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3639478.3643081

1 Introduction

Automatic code generation has emerged as a pivotal technology in the field of software development, with the potential to reduce errors associated with manual coding and drastically improve productivity. In recent years, a range of sophisticated models have emerged, including sequence-based, tree-based, and pre-trained code generation models, evolving at an extraordinary pace. State-of-the-art (SOTA) code generation models, especially those derived from pre-trained large language models (LLMs), have shown promising results in producing source code across various programming languages and tasks.

Despite these achievements, there are instances where these models either fail or underperform. To this end, more intricate and life-like benchmarks have been introduced, and rigorous evaluations have been carried out. Although the efficacy of LLMs has undergone extensive assessment, there remains an absence of a comprehensive taxonomy of their weaknesses. This oversight could inadvertently lead the research community to focus predominantly on familiar challenges, leaving critical but less explored issues under-researched and lacking adequate scrutiny.

In this study, we aim to uncover the weaknesses in automatic code generation by conducting an in-depth analysis of the problematic codes produced by three SOTA LLMs across three widely-used datasets. Finally, we identified eight distinct types of weaknesses. From the perspective of benchmarks, we found that some problematic code generated are associated with inaccurate prompts (Type I) or overly complex (Type II). Besides, some weaknesses represent false negatives, a result indicative of biases toward single-answer solutions (Type III). Additionally, we observed model-related weaknesses where code generation failed to capture essential prompt semantics (Type IV), exhibited incorrect API usage (Type V), improperly applied domain knowledge (Type VI), produced more code than necessary (Type VII), or redundantly duplicated code snippets (Type VIII). We mapped the distribution of these weaknesses across the LLMs, aiming to highlight the extent to which different types of weaknesses predominate within each model for each benchmark.

To the best of our knowledge, we are the first to systematically analyze the weaknesses of code generation from a comprehensive perspective of benchmarks and the employed models. Our findings illuminate the spectrum of weakness distributions and call for an immediate increase in investment towards more nuanced research. This includes the crafting of benchmarks that offer API-diverse solutions, refined semantic representation of prompts, and strategies to mitigate 'gold plating' issues, leveraging either sophisticated LLMs or advanced code truncation techniques.

2 Research Subjects

Models: CodeGen2.5 (7B) [2] is a multi-turn program synthesis model. The new paradigm of human-in-the-loop makes that it is competitive with the SOTAs. **CodeGeeX2** (6B) [3] is a state-of-the-art, multilingual code-generation model. **GPT-4**, OpenAI's latest generative model, is regarded as one SOTA model in various domains, including code synthesis.
Datasets: We choose two types of datasets based on their evaluation methods: match-based evaluation (i.e., **CoNaLa**) and unit-test-based (i.e., **HumanEval+** and **DS-1000**). All of them are common used in code-generation research.
Metrics: We select one match-based (i.e., **Exact match (EM)**) and one (unit-test) execution-based metric (i.e.,**pass@1**), to evaluate the quality of generated source code.
Case Selection and Annotation. For valid annotation with a 95% confidence interval, we randomly selected the required sample size: 339 for CoNaLa, 115 for HumanEval+, and 277 for DS-1000. To develop a taxonomy of weaknesses, we curated a selection of code instances generated by the LLMs with an EM score of less than 1 for CoNaLa, or a pass@1 below 1 for HumanEval+ and DS-1000 datasets. Subsequently, we employed thematic analysis [1] to meticulously examine these problematic samples. Our team of annotators comprised eight CS majors, including one lecturer, one PhD candidate, and six postgraduate students.

As a preliminary step, the first three authors independently analyzed 33 random problematic cases from CoNaLa in a pilot study,

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04…$15.00
https://doi.org/10.1145/3639478.3643081

which led to the identification of an initial set of six weakness categories. These categories were derived by comparing the prompts, reference code, and responses generated by the three LLMs. A training session was then conducted to familiarize the annotators with the initial category before commencing the annotation tasks: four annotators focused on CoNaLa, two on HumanEval+, and four on DS-1000. Each annotator worked independently. Upon receiving the results, one of the leading authors joined to form a triad with two others for joint discussions to resolve any discrepancies. This collaborative review process resulted in the discovery of two additional types of weaknesses.

3 Weakness taxonomy

In this section, we introduce each of the weakness types, and list the weakness ratio of each PLM in each benchmark in Table 1.

I: Inaccurate prompts. If the prompt cannot describe the intent accurately, LLMs are almost impossible to generate the accurate source code. There are three sub-types. a) *I.i. Vagueness* implies the existence of multiple interpretations for a single prompt. As a result, different models may produce varied responses based on their unique understanding. b) *I.ii. Incompleteness.* Sometimes, the prompt does not encompass all essential details found in the reference code and required for clear intent. Generally overlooked elements include critical parameters (such as a specific string instance), the storage constraint of outcomes, as well as routine operations. c) *I.iii. Inconsistency.* We identified some cases in which prompts may display inconsistencies, or they may even conflict with the reference code provided. This can cause difficulties for LLMs that rely on clear and coherent prompts to accurately generate the intended source code solutions.

II: Too complex prompts. Intuitively, if prompts are excessively challenging to comprehend, it becomes increasingly difficult for LLMs to generate correspondingly high-quality source code. During our annotation process, an annotator deems a prompt as overly complex when: 1) repeated reading is necessary to grasp its meaning; or 2) it involves several intertwined steps; or 3) understanding the prompt requires specific domain knowledge; or 4) it has convoluted sentence structures (more subordinate clauses and prepositional phrase). If both annotators agree on the high complexity of a prompt, we label it as 'too complex'. This weakness is more serious in DS-1000.

III: Biases from single answer. This weakness is particularly evident in benchmarks that do not include test cases. For any given prompt, multiple valid implementations may exist. Yet, most current public benchmarks provide only one reference code snippet for each prompt. This implies that while the generated source code might differ from the provided solution, it could still be correct. Unfortunately, match-based evaluation methods do not accommodate such variability and are highly sensitive to any discrepancies. Consequently, biases stemming from reliance on single-answer references are introduced. In the case of CoNaLa, our analysis reveals that 43.66% of the answers generated by GPT-4, which were deemed problematic by CodeBLEU, are actually false negatives. Similarly, for CodeGeeX2 and CodeGen2.5, the proportions of falsely negative evaluations stand at 32.15% and 23.01%, respectively, as in Table 1.

IV. Missing pivotal semantics. Critical information within a prompt, such as specific constraints or formatting requirements, is sometimes absent in the code generated by PLMs, highlighting a lapse in capturing pivotal semantic details. For instance, CoNaLa-13567345 explicitly instructs to "calculate sum over all rows of a 2D numpy array." However, CodeGen2.5 and GPT-4 erroneously compute the sum across columns. And CodeGeeX2 aggregates the entire array's elements instead of summing each row individually.

According to Table 1, this type of omission is most prevalent among the three LLMs, denoted in bold (with the exception of GPT-4 on CoNaLa). A consistent trend emerges across all three models: the more complex the prompt, the greater the severity of this issue. In DS-1000, featuring the lengthiest prompts at an average of 137.01 tokens, the frequency of this weakness is highest. It is the second-highest in HumanEval+ with an average prompt length of 67.72 tokens, and it is least common in CoNaLa, which has the shortest prompts, averaging 10.06 tokens.

V. Wrong API usage. This weakness relates to the improper application of an existing API—using correct API names but with incorrect parameters—or invoking nonexistent APIs, both indicative of a type of *syntactic error*. It is important to note that scenarios where incorrect APIs are utilized correctly yet fail to address the task described in the prompt do not fall into this category.

GPT-4 exhibited the fewest related instances, while CodeGeeX2 displayed the most. Conversely, we found no occurrences of this weakness in the outputs from HumanEval+, which could be attributed to smaller sample sizes and the higher quality of its prompts.

VI. Lack of domain knowledge. The problems of generated code are in the wrong usage or missing some common knowledge in certain areas. For instance, when generating source code for the prompt of "loop through the IP address range "192.168.x.x""(CoNaLa-13368659), CodeGen2.5 wronly iterates the required two numeric segments in the range of (1025, 65535), and CodeGeeX2 in the range of (1,255).

VII. Gold plating. We use the term "gold plating" metaphorically to describe instances where an LLM produces code that is more elaborate or intricate than necessary to meet the specified prompt. We believe this issue is well-recognized, since numerous studies adopt truncation strategies for refining the generated output. However, these strategies are often too generalized to discern all vital information. This kind of weakness was observed from the generated code in CoNaLa and DS-1000 by all of the three models.

VIII. Code duplication. This weakness is identified from the generated code of CodeGen2.5 and CodeGeeX2. These models occasionally produce redundant code in response to certain prompts, where the repetition may or may not be pertinent to the given task.

Table 1: Weakness Distribution Across Each PLM (%).

Model	Dataset	Weakness Types							
		I	II	III	IV	V	VI	VII	VIII
GPT-4	CoNaLa	27.14	0.88	**43.66**	8.85	0.59	1.18	5.60	0.00
	HumanEval	0.00	0.00	0.00	**42.11**	0.00	2.63	0.00	0.00
	DS-1000	8.47	7.41	0.00	**55.56**	1.59	0.00	1.59	0.00
CodeGeeX2	CoNaLa	27.14	1.18	32.15	**39.82**	2.36	2.65	10.62	4.13
	HumanEval	0.00	0.00	0.00	**73.71**	0.00	3.66	0.00	1.22
	DS-1000	6.51	7.91	0.00	**77.21**	6.51	0.00	2.33	0.47
CodeGen2.5	CoNaLa	27.43	0.88	23.01	**54.28**	2.36	2.06	8.55	0.00
	HumanEval	0.00	0.00	0.00	**68.13**	0.00	2.20	0.00	0.00
	DS-1000	6.90	6.51	0.00	**75.48**	2.30	0.00	4.21	0.77

4 Acknowledgment

Funding for this work has been provided by the National Science Foundation of China Grant NO. 62102014. It is also partially supported by the State Key Laboratory of Software Development Environment No.SKLSDE-2023ZX-03.

References

[1] Virginia Braun and Victoria Clarke. 2022. Conceptual and design thinking for thematic analysis. *Qualitative Psychology* 9, 1 (2022), 3.
[2] Erik Nijkamp, Bo Pang, and etc. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[3] Qinkai Zheng, Xiao Xia, and etc. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X.