Decompositional Semantic Analysis for LLM-based Code Quality Evaluation*

Fangzhou Xu[1], Sai Zhang[1], Xiaowang Zhang[1,*] and Yahong Han[1]

[1]*College of Intelligence and Computing, Tianjin University, Tianjin, 300350, China*

Abstract

Code quality evaluation involves scoring generated code quality based on a reference code. Extensive research has demonstrated that current evaluations do not truly reflect code quality. We propose Decompositional Semantic Analysis for Code Quality Evaluation. We employ a decompositional approach to enable LLMs to analyze portions of code semantics independently each time, obtaining the code semantics through multiple interactions with LLMs. We designed a Semantic Storage unit to make independent analysis feasible, by retriving related semantic descriptions. Experimental results indicate that our approach surpasses existing state-of-the-art methods in correlation with code execution.

Keywords

Code evaluation, Large Language Models, Code Semantic

1. Introduction

Code quality evaluation involves scoring generated code quality based on a reference code for a specific problem statement. Existing methods [1] [2] rely on superficial code matching as an evaluation metric, which fails to capture code semantics accurately. Moreover, extensive research has demonstrated that existing methods do not truly reflect code quality [3].

With the development of large language models (LLMs) in recent years, studies [4] have proven the feasibility of using LLMs as evaluators for generative tasks. However, due to issues like hallucinations and uncertainty in LLMs [5], their correlation with code execution remains at a lower level [6], making the direct use of LLMs for code quality evaluation challenging. To address these issues, we propose **D**ecompositional **S**emantic **A**nalysis for LLM-based **C**ode **Q**uality **E**valuation (DSA-CQE). We employ a decompositional approach to enable LLMs to comprehend portions of code semantics independently each time, obtaining the code semantics through multiple interactions with LLMs. We designed a Semantic Storage unit to make independent analysis feasible, allowing LLMs to achieve more accurate semantics by breaking down complex problems. Finally, the generated code is scored based on a semantic comparison between the reference code and itself. Experimental results indicate that DSA-CQE surpasses existing state-of-the-art methods in terms of correlation with code execution.

Posters, Demos, and Industry Tracks at ISWC 2024, November 13–15, 2024, Baltimore, USA

*Corresponding author.

xu_fangzhou@tju.edu.cn (F. Xu); zhang_sai@tju.edu.cn (S. Zhang); xiaowangzhang@tju.edu.cn (X. Zhang); yahong@tju.edu.cn (Y. Han)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-3828/paper7.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

2. Approach

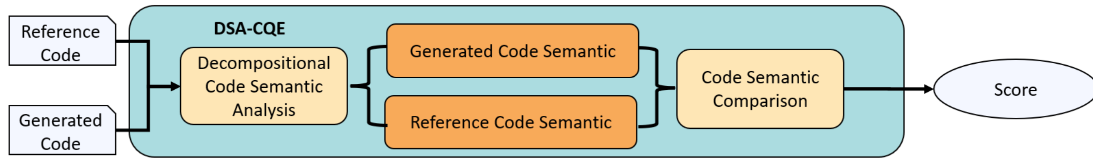

Figure 1: Framework of DSA-CQE.

Fig 1 illustrates the overall framework of DSA-CQE. DSA-CQE inputs the generated code and the reference code, the output is the score of the generated code. First, the semantic of both codes is obtained through a Decompositional Code Semantic Analysis unit. Subsequently, the code semantic comparison unit determines the differences in semantics. Finally, the generated code's score is derived by analyzing these semantic differences through an LLM. In Decompositional Code Semantic Analysis, we considered eight types of nodes of Abstract Syntax Tree (AST) [7] as our predefined nodes: "**For**", "**While**", "**Assign**", "**If**", "**ClassDef**", "**FunctionDef**", "**Switch**", and "**Call**". We perform a depth-first traversal of the code's AST, extracting the "subtrees" under these predefined nodes as sub-codes. This approach can decompose the originally complex code into simpler subcodes, allowing the LLM to perform semantic analysis[1] on each part separately, thereby reducing the hallucination phenomenon [5].

After decomposing the code into several sub-code, it is not feasible to analyze them individually, as most code segments are interrelated through references and dependencies. Analyzing them in isolation could lead to missing external references, such as variables and function definitions. We designed a Semantic Storage unit that stores textual descriptions of semantics during the analysis process, which may be required for subsequent code semantic analysis. As shown in Fig 2, a search is conducted within the Semantic Storage unit to retrieve relevant semantic descriptions. These descriptions are concatenated with the original sub-code and, together with a pre-designed prompt template, are input into the LLM to obtain the semantic description of the sub-code. For example, variables such as 'n', 'cap', and 'wei', which appeared previously in other sub-codes, can be easily misunderstood by the LLM without additional semantic information. Without context, the LLM might misinterpret n as any generic integer or cap as an abbreviation unrelated to the problem domain. However, after conducting semantic analysis on the earlier sub-codes, the semantics of these variables have already been stored in the Semantic Storage unit. We only need to retrieve these stored semantics and incorporate them into the prompt template to provide the LLM with the necessary semantic context for these external variables.

The semantics of the code stored in the Semantic Storage are not static. Each time a semantic description of a sub-code is obtained, the LLM is prompted to update the semantic descriptions of each external variable based on the new description. These updated semantic descriptions are then re-stored in the Semantic Storage unit for further analysis. As shown in Fig 2, the variable 'dp', initially described as "a dynamic programming array initialized to 0," is updated to "stores the maximum value for each possible weight" after semantic analysis.

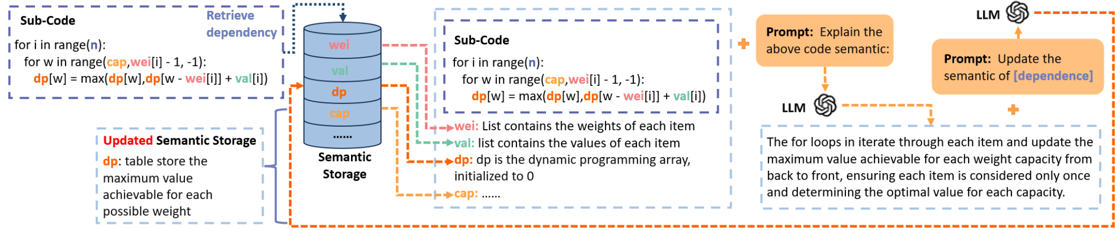

Figure 2: Demonstrates how the Semantic Storage unit eliminates external dependencies, as well as the process of updating its internal semantic descriptions

Table 1
Kendall-Tau (τ), Pearson (r_s) correlations. The best performance is **bold**.

Method	r_s	τ
CodeBleu	.295	.241
CodeBertScore	.430	.352
1-shot	.106	.105
Simplified DSA-CQE	.512	.470
DSA-CQE	**.594**	**.553**

3. Experiments

We conducted our experiments (following previous work [4]) on the HumanEval dataset [8] exclusively, as most of the code samples in the CoNaLa [9] subset of the dataset [3] used for evaluation are single-line codes lacking complex semantics. While the Card2Code Hearthstone [10] subset contains semantically more complex structures, such as “classes”, these “classes” follow a uniform structure with minimal variation. In practice, a significant portion of code demonstrates both complexity and semantic diversity. In contrast, the HumanEval dataset contains a rich and diverse range of code samples, making it the ideal choice for our experiments and evaluation. Cassano et al. [11] ran test cases on the HumanEval dataset and provided the functional correctness of each piece of code. We use the Pearson [12] and Kendall [13] correlation coefficient between the functional correctness scores and the scores given by different methods for comparison. To ensure fairness, we uniformly used GPT-3.5 Turbo [14] as the backbone model and set the LLM temperature to 0.2. We used state-of-the-art evaluation methods based on n-gram matching and deep learning, namely CodeBleu [1] and CodeBertScore [2], as baselines. The prompt for 1-shot utilized Zhou’s prompt template [4]. Simplified DSA-CQE is our framework, which replaces decomposition analysis with single-step analysis using LLMs[1].

The experimental results are shown in the table 1. As can be seen, DSA-CQE performed significantly better on the HumanEval dataset compared to traditional code evaluation methods, with a Pearson correlation coefficient of 0.594. The single-step prompt and Simplified DSA-CQE methods achieved Pearson correlation coefficients of 0.106 and 0.512, respectively. This indicates that DSA-CQE, through decompositional semantic analysis, enhances the LLM’s comprehension of code semantics and improves overall performance in code evaluation.

Our current experiment focuses solely on evaluating the quality of Python code. However, since the method relies on the Abstract Syntax Tree, adapting it to other programming languages

involves merely substituting the relevant parser. For instance, Java code can be parsed using JavaParser [15], while pycparser [16] can be used for C code.

4. Conclusion

In this poster, we propose Decompositional Semantic Analysis for LLM-based Code Quality Evaluation. We employ a decompositional approach to enable LLMs to analysis portions of code semantics independently each time, obtaining the code semantics through multiple interactions with LLMs. We designed a Semantic Storage unit to make independent analysis feasible, by retriving related semantic descriptions. The generated code is scored based on a semantic comparison between the reference code and itself. The experimental results show that DSA-CQE surpasses all existing methods in correlation with code execution.

References

[1] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, S. Ma, Codebleu: a method for automatic evaluation of code synthesis, arXiv preprint arXiv:2009.10297 (2020).

[2] S. Zhou, U. Alon, S. Agarwal, G. Neubig, Codebertscore: Evaluating code generation with pretrained models of code, arXiv preprint arXiv:2302.05527 (2023).

[3] M. Evtikhiev, E. Bogomolov, Y. Sokolov, T. Bryksin, Out of the bleu: how should we assess quality of the code generation models?, Journal of Systems and Software 203 (2023) 111741.

[4] T. Y. Zhuo, Large language models are state-of-the-art evaluators of code generation, arXiv preprint arXiv:2304.14317 (2023).

[5] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, P. Fung, Survey of hallucination in natural language generation, ACM Computing Surveys 55 (2023) 1–38.

[6] M. Zhong, Y. Liu, D. Yin, Y. Mao, Y. Jiao, P. Liu, C. Zhu, H. Ji, J. Han, Towards a unified multi-dimensional evaluator for text generation, arXiv preprint arXiv:2210.07197 (2022).

[7] I. Neamtiu, J. S. Foster, M. Hicks, Understanding source code evolution using abstract syntax tree matching, in: Proceedings of the 2005 international workshop on Mining software repositories, 2005, pp. 1–5.

[8] J. T. H. J. e. a. Chen, Mark, Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).

[9] P. Yin, B. Deng, E. Chen, B. Vasilescu, G. Neubig, Learning to mine aligned code and natural language pairs from stack overflow, in: Proceedings of the 15th international conference on mining software repositories, 2018, pp. 476–486.

[10] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, P. Blunsom, Latent predictor networks for code generation, arXiv preprint arXiv:1603.06744 (2016).

[11] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, A. Jangda, Multipl-e: A scalable and polyglot approach to benchmarking neural code generation, IEEE Transactions on Software Engineering 49 (2023) 3675–3691. doi:10.1109/TSE.2023.3267446.

[12] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, I. Cohen, Pearson correlation coefficient, Noise reduction in speech processing (2009) 1–4.

[13] M. G. Kendall, A new measure of rank correlation, Biometrika 30 (1938) 81–93.

[14] OpenAI., Openai gpt-3.5 turbo, https://platform.openai.com/docs/guides/text-generation/chat-completions-api, 2022.

[15] javaparser, https://github.com/javaparser/javaparser, n.d.

[16] pycparser, https://github.com/eliben/pycparser, n.d.