Identifying Gaps In Students' Explanations of Code Using LLMs

Rabin Banjade[0000−0001−8048−0558], Priti Oli[0009−0007−8720−1487], Mahmudul Islam Sajib[0009−0000−4733−2166], and Vasile Rus[0009−0002−4739−0440]

University of Memphis, USA
{rbnjade1,poli,msajib,vrus}@memphis.edu

Abstract. This study investigates methods based on Large Language Models (LLMs) to identify gaps or missing parts in learners' self-explanations. This work is part of our broader effort to automate the evaluation of students' freely generated responses, which in this work are learners' self-explanations of code examples during code comprehension activities. We experimented with two methods and four distinct LLMs in two distinct settings. One method prompts LLMs to identify gaps in learners' self-explanations, whereas the other method relies on LLMs performing a sentence-level semantic similarity task to identify gaps. We evaluated these methods in two settings: (i) simulated data generated using LLMs and (ii) actual student data. Results revealed the semantic similarity method significantly improves task performance over the zero-shot prompting for gap identification (the holistic method), i.e., over the standard method of prompting LLMs to directly address the gap identification task.

Keywords: Self Explanation · Large Language Models · Automated Assessment · Scaffolding · Code Comprehension

1 Introduction

This paper explores several methods based on large language models to detect gaps or missing parts in learners' self-explanations of targeted instructional content, e.g., explanations of code examples. Self-explanation, i.e., elucidating learning material to oneself through speech or writing[8], has a positive impact on learning [1]. Students who self-explain while engaging in various instructional activities, such as reading code examples or solving problems, tend to learn more and develop a more profound understanding of the subject matter. However, as Renkl and colleagues [12] point out, the effectiveness of self-explanation might not be effective in instances where learners either neglect pertinent explanations or engage passively with the content. Building upon these insights, Intelligent Tutoring Systems (ITS) that rely on instructional strategies such as scaffolded self-explanation have been developed to enhance learning outcomes as demonstrated by various studies [6].

Scaffolding students' self-explanation relies on accurately assessing students' explanations in terms of correctness or completeness. This entails addressing two

crucial tasks: identifying incorrect student responses and identifying incomplete responses, the latter being the focus of our study. For instance, in the case of incorrect responses, the system may correct potential misconceptions articulated by the student, while for incomplete responses, it may offer appropriate hints to encourage learners to think about the missing components—such as steps in a problem solution. In this study, we explore using LLMs to identify missing parts in student self-explanations of code examples.

LLMs have emerged as state-of-the-art systems that can generate coherent content in response to input prompts [2], including computer code and accompanying explanations, which is relevant to our work here. Indeed, this generative feature can be very useful for our target task of assessing student responses because, for instance, in current state-of-the-art ITSs, this task relies on expert-authored benchmark explanations that are considered correct and complete (unless they are well-known misconceptions). The benchmark explanations play a crucial role in automated methods that rely on semantic similarity approaches to evaluate the correctness and completeness of student responses [15, 3]. In such approaches, if a student's answer is semantically equivalent to a corresponding benchmark response, the student response is deemed as having the same correctness and completeness value as the expert-generated benchmark response. It should be noted that semantic similarity has been the dominant approach to assessing self-explanations.

Using experts to author the benchmark responses is tedious and expensive. Therefore, such approaches do not scale well across topics and domains. LLMs have been shown to help significantly with code explanation generation for typical code examples used in intro-to-programming courses [14, 7, 10]. Furthermore, there is evidence that LLMs can competitively address the semantic similarity task between two texts [4, 9].

Therefore, we present in this paper the result of our investigation on the role of LLMs in addressing two major challenges in the automated assessment of students' self-explanation: (i) the generation of benchmark/reference explanations and (ii) auto-assessment of student-generated free responses during instructional tasks. For the latter task, auto-assessment, we explore two approaches: (1) a direct approach in which we prompt LLMs to identify gaps in self-explanations and (2) a semantic similarity approach in which we prompt LLMs to assess the similarity of a student self-explanation with respect to a benchmark/reference explanation.

As research questions, this paper addresses the following key questions.

RQ1: Can LLMs detect gaps in code explanations, indicating incomplete explanations?

RQ2: Can LLMs-generated explanations be used with semantic-similarity approaches to identify gaps in students' code explanations?

RQ3: Can LLMs accurately determine the semantic similarity of self-explanations at the sentence level?

2 Methodology

To meet our research objectives, we developed a comprehensive methodology. First, we generate code explanations to be used as benchmark explanations. Second, we used two approaches to identify missing details in student explanations. The two approaches are (1) a holistic approach in which LLMs are prompted to identify the missing parts given a student explanation and the corresponding code example and (2) a point-wise, semantic similarity approach at sentence level in which student explanations are broken down into units of analysis (sentences) and then LLMs are prompted to identify which such sentences match corresponding sentences in the benchmark explanations (a sentence in the benchmark/reference explanation is called an expectation). The methodology involves a selection of relevant models and prompts and the evaluation of LLMs in identifying missing expectations in student explanations.

Code Explanation Generation Our first step involved generating code explanations using four different models and different prompts to evaluate different types of code explanations. We aimed to understand the diversity and quality of explanations produced by these models. We employed four different LLMs to generate benchmark code explanations: GPT-4-0613 [11], GPT-3.5 Turbo-0613, Mixtral-8x7b-Instruct-v0.1 [16], and llama-2-70b-chat [17]. These LLMs have achieved state-of-the-art results in various tasks while differing in training data and algorithms, although the model size and training data for OpenAI models are not disclosed. We queried all the models using the unified interface provided by *LiteLLM*[1], ChatGPT-4 and ChatGPT-3.5 were queried through OpenAI API, whereas Mixtral and LLama-2 were prompted via the *Together API*[2]. For all these models, we used a temperature parameter value of 0 for consistency and reproducible results. We generated explanations for 10 Java code examples, sourced from the DeepCode codeset [13], which were diverse in complexity and concept coverage. These examples were previously used in a human subject experiment where self-explanations were collected from students.

P1: Provide a line-by-line explanation of the given java code {code}
P2: Explain the code to the student in a way that they understand necessary concepts. Focus on conceptual understanding of the code.{code}
P3: Summarize the given java code {code}.
P4: In this code example, {code}, we will focus on the concept of loops. We do that with the help of a program whose goal is to find the smallest divisor of a positive number. Your task is to read the code shown and understand what it does. Once you are done reading the code, type your explanation of what the code does. Try to identify the major blocks of code and their goals and how those goals are implemented. Please go on and do your best

[1] https://litellm.ai/
[2] https://docs.together.ai/docs/quickstart

to explain your understanding of the code and its output in as much detail as you can.

The prompts (P1,P2, P3,P4) were chosen to understand how LLMs generate explanations and which explanation to use as a benchmark explanation for our task of identifying gaps in student explanations. Prompt **P4** is identical to the prompt given to students when prompted to self-explain the code whereas prompts P1, P2, and P3 were used to generate different types of code explanations based on previous studies [10]. We evaluated the quality of explanations generated from each prompt to be used in our main task.

2.1 Data

We evaluated our methods using two data settings: student data and simulated data. From 60 students at a large US public university, we collected self-explanations prompted by **P4**. Sampling was based on word length to ensure detailed explanations, yielding 50 diverse explanations annotated with missing expectations relative to DeepCode codeset benchmarks [13]. For simulated data, we derived a dataset from LLM-generated explanations by randomly removing one expectation/sentence per explanation and repeated three times to create three samples each with one missing expectation. This simulated gaps in student explanations, with removed expectations serving as ground truth for evaluating LLMs' detection of missing components. Initial observations indicated that sentences from the generated explanations aligned with expectations, representing comprehension of specific code parts/concepts.

2.2 Prompting To Identify Gaps In Explanations of Code

In order to prompt LLMs to identify gaps or missing expectations in student or simulated explanations of code, we went through a prompt selection process that involved several hits and trials. We used two different settings to prompt for identifying missing parts in code explanations: (1) the holistic approach in which we prompt to identify what is missing given the student explanation and the corresponding code, and (2) the pointwise, sentence-level semantic similarity-based approach. For the holistic approach, we designed two different types of prompts with providing reference or benchmark explanations (see below prompt P6) and without (P7 - see below). For consistent evaluation between student data and simulated missing data, we prompted to generate a single sentence for each prompt.

P6: Given the following code:{code} and the following reference explanation: {reference explanation}, your task is to identify what is missing in the following student explanation:{student explanation} of the code. Generate the missing part as a single sentence.
P7: Given the following code:{code}, your task is to identify what is miss-

ing in the following student explanation:{student explanation} of the code. Generate the missing part as a single sentence.

As the holistic prompting didn't yield very promising results, we have introduced a novel approach to guide LLMs in identifying and generating the absent expectation: the pointwise, sentence-level semantic similarity approach. Rather than directly requesting the missing expectation, our method evaluates pairwise resemblances between the explanations produced by LLMs and those produced by students. When dealing with the simulated data, the pairwise similarity consistently yields completely accurate outcomes, given that we compare the same pair of sentences except for the missing one. As a result, we only report results here for the pairwise similarity technique when applied to the student explanations. The LLM-generated explanations function as expert/benchmark explanations. To identify the missing expectation, in this case, we pinpoint the sentence within the expert explanation (LLMs' explanation) that exhibits the lowest pairwise semantic similarity with any sentence in the simulated student explanation. This particular sentence is considered to result in missing expectations.

For implementing the pairwise similarity approach, we adopt the subsequent prompt format:

P8: Provide a semantic similarity score on a scale of 0 to 1, 0 being least similar and 1 being most similar, for the following two sentences {reference sentence} and {student sentence}.

We chose a scale of 0 to 1 for similarity prompting as Gatto et al. [5] showed that prompting for a similarity value between 0 to 1 has a better performance compared to using a 1-5 scale. This has been confirmed by our experiments as well.

Evaluation We evaluated code explanation generation using two metrics: *Correctness* and *Completeness*. Correctness assessed the accuracy of explanations, while Completeness measured coverage of code concepts. For missing expectation generation, we focused solely on Correctness, which indicates the proportion of accurate responses in identifying missing parts.

3 Results and Discussion

We compared the completeness and correctness metric of code explanations generated from four different LLMs using four prompting strategies, as mentioned in the methodology section. We obtained 100% correct explanations for all the prompts from all the models. However, we obtained 88%, 94%, 71%, and 80% completeness scores for prompts P1, P2, P3, and P4, respectively, averaged across all the models(n=40) for each prompt. This evaluation aimed to understand which prompt generated the most accurate explanations (correct and complete), which we can use to generate reference/benchmark explanations

instead of using experts. Prompt(**P2**) leads to the best explanations. The other prompts led to explanations that showed specific characteristics that made them inconsiderable as reference explanations. For instance, prompting for line-by-line explanations**(P1)** generates explanations focusing on code's syntactic elements, whereas code summarization **(P3)** prompts miss out on important details necessary for understanding and learning. Prompt **P4** generates explanations mostly on the block level and misses important syntactic knowledge components. In sum, explanations generated by **P2** are balanced regarding functional and syntactic level explanations. Therefore, we obtained reference explanations using the prompt(**P2**). Explanations from all the models for **P2** were similar in completeness and correctness, so we randomly sampled explanations from different models as benchmark explanations. One observation was that LLAMA2 explanations consisted of conversation-style filler sentences such as," Sure, here is an explanation of the given code," which we removed. For each explanation, we sampled a reference explanation for the same code such that the two differed.

3.1 Prompting For Missing Expectations

We prompted LLMs in zero-shot settings to identify missing parts (expected unit answers where the unit is a sentence) in student explanations. As mentioned earlier in the *Methodology* section, we evaluated the performance of LLMs on this task using two prompt settings, (1) with and (2) without the use of reference explanations, for two sources of explanations, (1) simulated data and (2) student data.

Table 1 shows our results for prompting for missing expectations under different settings using simulated and student data. As seen in the table, prompting to identify (and generate) missing expectations with or without reference explanations does not improve performance in terms of correctness. It also shows that additional context from reference explanation does not provide an additional boost in both simulated data and student data. Comparing among models, Mixtral and GPT-4 provide better performance in terms of correctness. Also, when the same prompts were given to student data, the performance of LLMs decreased, which can be attributed to the diverse nature of student explanations, which often adopted conversational styles that deviated from conventional grammar and writing standards in English. One of the observations worth noting is that generated missing expectations were not specific. For example, even though the goal was to understand the program that solved a specific task, most of the generated missing expectations were about complexity analysis; this was mostly prevalent in code examples such as binary search and insertion sort. One of the major challenges, as we observed, is to guide LLMs to focus on missing details compared to reference explanations, which would set the scope of learning. One could probably achieve this with various advanced prompting techniques, which can be further explored; we resorted to a prevalent approach, i.e., semantic similarity-based assessment of student answers. Towards this direction, As discussed in the methodology section, we prompted for pairwise similarity between sentences as shown in table 1 indicated by **P8**. Our evaluation is based

on a single missing expectation; our results indicate that we can vastly increase the correctness of our generated explanations. Our results also indicate that language models can perform well for semantic similarity tasks, as indicated by other works [5].

Table 1. Correctness comparison (n=50) for results obtained by prompting LLMs for missing expectation generation in simulated and student data.

	Simulated Data		**Student Data**		
Model	**P6**	**P7**	**P6**	**P7**	**P8**
GPT-3.5	47%	44%	37%	48%	82%
GPT-4	77%	66%	40%	41%	94%
LLAMA2	66%	60%	34%	42%	84%
MIXTRAL	77%	77%	52%	38%	90%

4 Conclusion and Future Work

In summary, we studied if we can identify missing gaps in student explanations using four state-of-the-art LLMs. Our experiments under different settings using explanations generated from LLMs as reference explanations showed that prompting for similarity can yield better results for finding missing expectations than zero-shot prompting of LLMs. This indicates that LLMs can determine semantic similarity in sentence level for student explanation in code comprehension tasks. One of the limitations of our work is maintaining temporal validity due to LLMs' evolving landscape. LLMs offer new research opportunities in generating programming exercises, unit tests, code explanations, and providing automated feedback on student code submissions, but effectively guiding novice programmers remains a challenge. Our efforts to utilize LLMs for scaffolding students' code understanding represent progress in this direction. We plan to explore various prompting techniques and leverage open-source models like LLAMA2 to enhance transparency and scalability in education.

ACKNOWLEDGMENTS

This work has been supported by the following grants awarded to Dr. Vasile Rus: the Learner Data Institute (NSF award 1934745); CSEdPad (NSF award 1822816); iCODE (IES award R305A220385). The opinions, findings, and results are solely those of the authors and do not reflect those of NSF or IES.

Acknowledgements

This work has been supported by the following awards: NSF #1934745, #1918751, #1822816, and IES award R305A220385. The opinions, findings, and results are solely those of the authors and do not reflect those of NSF or IES.

References

1. Aleven, V.A., Koedinger, K.R.: An effective metacognitive strategy: learning by doing and explaining with a computer-based cognitive tutor. Cognitive Science **2**(26), 147–179 (2002)
2. y Arcas, B.A.: Do large language models understand us? Daedalus **151**(2), 183–197 (2022)
3. Bexte, M., Horbach, A., Zesch, T.: Similarity-based content scoring - a more classroom-suitable alternative to instance-based scoring? In: Rogers, A., Boyd-Graber, J., Okazaki, N. (eds.) Findings of the Association for Computational Linguistics: ACL 2023. pp. 1892–1903. Association for Computational Linguistics, Toronto, Canada (Jul 2023). https://doi.org/10.18653/v1/2023.findings-acl.119, https://aclanthology.org/2023.findings-acl.119
4. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)
5. Gatto, J., Sharif, O., Seegmiller, P., Bohlman, P., Preum, S.M.: Text encoders lack knowledge: Leveraging generative llms for domain-specific semantic textual similarity. arXiv preprint arXiv:2309.06541 (2023)
6. Graesser, A.C., McNamara, D.S., VanLehn, K.: Scaffolding deep comprehension strategies through point&query, autotutor, and istart. Educational psychologist **40**(4), 225–234 (2005)
7. MacNeil, S., Tran, A., Mogil, D., Bernstein, S., Ross, E., Huang, Z.: Generating diverse code explanations using the gpt-3 large language model. In: Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2. pp. 37–39 (2022)
8. McNamara, D.S., Magliano, J.P.: Self-explanation and metacognition: The dynamics of reading. In: Handbook of metacognition in education, pp. 60–81. Routledge (2009)
9. Oli, P., Banjade, R., Chapagain, J., Rus, V.: Automated assessment of students' code comprehension using llms. arXiv preprint arXiv:2401.05399 (2023)
10. Oli, P., Banjade, R., Chapagain, J., Rus, V.: The behavior of large language models when prompted to generate code explanations. In: Proceedings of the workshop on Generative AI for Education (GAIED) at the Thirty-seventh Conference on Neural Information Processing Systems (NeurIPS 2023). arXiv (2023)
11. OpenAI, R.: Gpt-4 technical report. arxiv 2303.08774. View in Article **2**, 13 (2023)
12. Renkl, A.: Learning mathematics from worked-out examples: Analyzing and fostering self-explanations. European Journal of Psychology of Education **14**(4), 477–488 (1999)
13. Rus, V., Brusilovsky, P., Tamang, L.J., Akhuseyinoglu, K., Fleming, S.: Deepcode: An annotated set of instructional code examples to foster deep code comprehension and learning. In: International Conference on Intelligent Tutoring Systems. pp. 36–50. Springer (2022)
14. Sarsa, S., Denny, P., Hellas, A., Leinonen, J.: Automatic generation of programming exercises and code explanations using large language models. In: Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1. pp. 27–43 (2022)
15. Sung, C., Dhamecha, T., Saha, S., Ma, T., Reddy, V., Arora, R.: Pre-training bert on domain resources for short answer grading. In: Proceedings of the 2019

Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). pp. 6071–6075 (2019)
16. team, M.A.: Mixtral of experts (2022), https://mistral.ai/news/mixtral-of-experts/
17. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al.: Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023)