Generating Feedback-Ladders for Logical Errors in Programming using Large Language Models

Hasnain Heickal
University of Massachusetts Amherst
Amherst, MA, USA
hheickal@cs.umass.edu

Andrew Lan
University of Massachusetts Amherst
Amherst, MA, USA
andrewlan@cs.umass.edu

ABSTRACT

In feedback generation for logical errors in programming assignments, large language model (LLM)-based methods have shown great promise. These methods ask the LLM to generate feedback given the problem statement and a student's (buggy) submission and have several issues. First, the generated feedback is often too direct in revealing the error in the submission and thus diminishes valuable learning opportunities for the students. Second, they do not consider the student's learning context, i.e., their previous submissions, current knowledge, etc. Third, they are not layered since existing methods use a single, shared prompt for all student submissions. In this paper, we explore using LLMs to generate a "feedback-ladder", i.e., multiple levels of feedback for the same problem-submission pair. We evaluate the quality of the generated feedback-ladder via a user study with students, educators, and researchers. We have observed diminishing effectiveness for higher-level feedback and higher-scoring submissions overall in the study. In practice, our method enables teachers to select an appropriate level of feedback to show to a student based on their personal learning context, or in a progressive manner to go more detailed if a higher-level feedback fails to correct the student's error.[1]

Keywords

Feedback, Large Language Models, Programming Assignments

1. INTRODUCTION

One of the primary ways humans learn is through feedback. In a class, where a teacher has to interact with numerous students, manually providing appropriate feedback for each student can be time-consuming. However, the effect of immediate, just-in-time [3] and personalized[4] feedback on student learning is hugely positive. The current practice of human feedback through teachers and teaching assistants can often be one-size-fits-all and not-on-time for students. Intelligent tutoring systems and online learning platforms can enable automatic feedback and scale up teachers' efforts, especially in the setting of learning programming. Due to the highly structured nature of programming languages, there exists a body of work on feedback generation for students in programming tasks that has some significant success.

Automated feedback methods in programming scenarios are categorized into *edit-based* feedback, which focuses on code edits to fix bugs, and *natural language*-based feedback, providing explanations, or suggestions conversationally. Edit-based approaches primarily target syntax errors or provide suggestions for fixing buggy code [11, 16]. Recent advancements in large language models (LLMs) allow for the generation of fluent utterances around feedback, leveraging generative capabilities and knowledge of code. Approaches for automated feedback generation also consider the type of student errors, distinguishing between *syntax* and *logical* errors. While syntax errors have been extensively studied, logical errors remain a challenge [19, 20, 13, 7].

Automated feedback for logical errors in programming education has also been studied. [6, 5] generates test cases and Socratic questions based on the student submission. [14] systematically compares the performance of ChatGPT and GPT-4 to that of human tutors for a variety of scenarios in programming education including feedback generation. Their method produces a concise, single-sentence hint for the bug in the program, which is evaluated using human annotators. Similarly, [15] investigates the role of LLMs in providing human tutor-style programming hints to students. It uses a similar prompting strategy to the one in [14] to generate hints augmented with the failing test case and the fixed program, and then validated by GPT3.5. [12] conducts an experimental study using GPT3.5 to generate real-time feedback for the students and measure the effect of the feedback. All of these methods lack personalization of feedback, only generate feedback of one type, and fall short in quality evaluation experiments for the generated feedback.

In this paper, we explore the capability of GPT-4 in generating "feedback-ladder", a multilevel feedback cascade for a single student-written buggy program. Different students need different types of help when solving problems [18, 9] and also at various stages of learning, students' help-seeking behavior changes [2]. From these findings, we expect that

[1]Full paper: `https://arxiv.org/abs/2405.00302`. The authors thank the NSF for partially supporting this work under grant DUE-2215193.

H. Heickal and A. Lan. Generating feedback-ladders for logical errors in programming using large language models. In B. Paaßen and C. D. Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 947–951, Atlanta, Georgia, USA, July 2024. International Educational Data Mining Society.

`https://doi.org/10.5281/zenodo.12730007`

generating feedback at *multiple levels*, forming a ladder, will be helpful to students. Feedback at a lower, coarse-grained level supports students conceptually and without giving away the solution – suitable for students with higher abilities. Feedback at a higher, fine-grained level supports students more specifically and directly points out their mistakes – suitable for students with lower abilities and prevents them from getting stuck. Therefore, teachers can use the feedback-ladder for personalized feedback by showing different levels to students with different needs, possibly by starting at a lower level and moving up if the student still struggles. Our key research question is: **Can we automatically generate relevant and effective feedback-ladders to address students' logical errors in introductory programming assignments?**

1.1 Contributions

In this paper, we detail a method for automatically generating feedback-ladders for logical errors in programming assignments using GPT-4. First, we propose an LLM-based method that automatically generates feedback-ladders to address logical errors in programming assignments. Second, we conduct a user study that consists of annotators who are students, instructors, or researchers in programming education to evaluate the quality of the generated feedback-ladder. We observe that each level has a similar effectiveness score across different questions, though higher levels have lower scores than the lower levels. We also observe that it is harder to generate effective feedback-ladder for higher-scoring submissions than lower-scoring submissions.

2. METHODOLOGY

2.1 Feedback-ladder

We define feedback-ladder as a set of varying levels of feedback for a student-submitted, incorrect (including partially correct) program instance, P_s. Each level in the ladder corresponds to how much information is contained in the hint shown to the student. With increasing levels of feedback in the ladder, the learning gain of the student diminishes [10]. The feedback levels are defined as:

Level 0 (Yes/No): The feedback only indicates whether P_s is correct or not. If a student can utilize this level of feedback, then they are going to learn the most.

Level 1 (Test Case): The feedback generates a test case C_{fail}, that when given to P_s will give the wrong answer. It consists of the input for C_{fail}, the expected correct answer according to the problem, and what P_s gives as the output. Utilizing this level teaches students debugging skills, which are hard to learn as beginners, and very conducive to their learning.

Level 2 (Hint): The feedback generates a high-level description of the logical error in the code that is responsible for the failure. This level of feedback should focus on the conceptual error that a student might have and must avoid any suggestion regarding editing the code. Feedback on this level has the best balance for a student to correct the mistake, as well as achieve learning gains.

Level 3 (Location): This level points out the lines at which the mistake occurred in the program. The feedback should refrain from explicitly mentioning the actual mistake or any edit suggestions. A student with sufficient programming knowledge should be able to correct the mistake with feedback on this level.

Level 4 (Edit): At this level, the feedback generates edits that turn the student-submitted buggy program, P_s, into a correct program. The feedback should modify the existing structure of P_s and produce something as close as possible while showing only the edits instead of the whole corrected program. The learning gain for this level is minimal, though it can be helpful for students who are absolute beginners.

Table 1: Prompt for feedback-ladder generation in GPT-4.

There can be different levels of feedback for a student who is trying to solve a programming assignment. Below we describe each level.
Level 0: Just the correct or incorrect verdict for the code.
Level 1: Giving a test case where the code fails. The test case contains just input, expected output and the code output. No explanations.
Level 2: A high-level explanation of why the code failed in the test case. No mention of how to modify the code.
Level 3: A high-level suggestion about the location in the code where you should make the changes.
Level 4: Suggestion in actual programming language how to change the code to get the correct solution. Just the statements where change is necessary are mentioned. The full solution code is never given.
For the given problem and code, generate feedback for each of these levels. When generating test cases make sure the generated test case falls inside the valid range.
Problem: Write a function in Java that implements the following logic: Given 2 ints, a and b, return their sum. However, sums in the range 10..19 inclusive, are forbidden, so in that case just return 20.
Code:

```
public int sortaSum(int a, int b){
    if (a + b <= 10 && a + b >= 20)
        return 20;
    else return a + b;
}
```

2.2 Generating Feedback-Ladder

We use the definitions of different levels in feedback-ladder in the prompts for GPT-4 to generate feedback in each level for a student-submitted buggy program. The prompt includes these definitions plus both the problem statement (Q) and the buggy program P_s itself. We use the openAI API to prompt GPT-4. Table 1 shows an example prompt to generate the feedback-ladder. Table 2 shows the generated feedback-ladder for the prompt in Table 1 by GPT-4.

3. EXPERIMENTS

3.1 Dataset

We use the dataset from the 2nd CSEDM Data Challenge, hereafter referred to as the CSEDM dataset [17]. It is a college-level, publicly-available dataset with students' actual code submissions. It contains 246 college students' 46,825 full submissions on each of the 50 programming questions over an entire semester with rich textual information on problem statements and student code submissions and other relevant metadata such as the programming concepts involved in each question and all error messages returned by the compiler. From the full dataset, we only take submissions that do not have syntax errors, since our work

Table 2: Corresponding response for the prompt in Table 1

Level 0: Incorrect
Level 1:
The program fails in the following case.
Input: a = 5, b = 6
Expected Output: 20
Your Output: 11
Level 2: The code fails because it wrongly handles the condition for the sum of a and b. The condition should be if the sum is greater than or equal to 10 and less than or equal to 19, then return 20.
Level 3: The code fails because it mistakenly checks if the sum of a and b is less than or equal to 10 and greater than or equal to 20. This condition will never be true as a number cannot be both less than or equal to 10 and greater than or equal to 20 at the same time.
Level 4: Change the if condition from `if (a + b <= 10 && a + b >= 20)` to `if (a + b >= 10 && a + b < 20)`.

focuses on generating feedback for programs that have no syntax errors but contain logical errors in solving the given problem. Therefore, we only curate such submissions from the dataset with less-than-perfect scores. To keep the scale of the user study manageable, we handpicked 5 problems. These 5 problems require students to use arrays, strings, loops, and conditions in their programs. Therefore, these problems test a comprehensive set of programming knowledge for the students. These problems are also complex enough to have room for nuanced mistakes from the students. For each of these five problems, we select three different student-submitted programs (submissions), one each from the following three categories:
Low-scoring submissions: Submissions scoring less than 20% in the test cases. These submissions have numerous errors and require significant edits to fix.
Mid-scoring submissions: Submissions scoring between 40% and 60% in the test cases. These submissions have fewer mistakes than the low-scoring submissions.
High-scoring submissions: Submissions scoring more than 80% in the test cases. These are almost correct and often require a few edits to get the full score. However, finding the errors in these submissions is difficult since they often miss only a few corner test cases.

3.2 Experiment Design

We conduct a user study to evaluate the quality of the generated feedback-ladder. The annotators for the user study are recruited from several universities, with different knowledge levels of programming: students, CS instructors, and AI researchers. There are a total of 10 annotators. The user study consists of several different phases, detailed below.

Eligibility determination phase: The purpose is to determine that the annotators have sufficient programming skills required by the study. First, the annotators are shown a simple program in Java. Second, they are given a set of inputs for the given program. Third, they are asked to determine the output of the program on the given set of inputs. Failing this disqualifies them from participating in the study.
Calibration phase: The purpose is to calibrate the evaluation objective among annotators to align with our expectations. First, the annotators are shown examples of different feedback-ladder. Each example contains a problem statement, a submitted program, and a generated feedback-ladder. Second, they are asked various questions about the shown examples. The questions are related to identifying the proper level of a feedback text, evaluating qualitative measures, etc. If the annotator got any of the answers wrong, we show which questions are wrong but do not reveal the answer. They try again and choose different answers for the same questions. This phase ends when the annotators answer all the questions correctly.
Evaluation phase: The purpose is to evaluate the quality of the generated feedback-ladder for 15 total programs, i.e., 5 chosen problems with 3 different submissions each. First, the annotator is shown a problem description. Then they are shown a low-scoring student-submitted program and the generated feedback-ladder for it. Second, they are asked to rate two different metrics of each of the feedback on a 5-point Likert scale, which we detail below. This repeats two more times for the mid and high-scoring submissions. The whole thing repeats for four more problems.
Evaluation Metrics: We ask study annotators to rate the feedback of each level on the feedback-ladder on *Relevance* and *Effectiveness*. The *relevance* of feedback for a particular level is based on how well the feedback matches its level definition. 5 corresponds to a perfect match and 1 corresponds to no match. The *effectiveness* of a generated feedback for a particular level is how effectively it can help a student, according to the annotator's judgment. 5 corresponds to highly effective feedback that should help the student fix their bug and 1 corresponds to feedback that does not help the student or may confuse them.

Table 3: Inter-rater agreement between annotators measured in Pearson correlation coefficient (PCC) values.

Annotator	A	B	C	D	E	F	G	H	I	J	Avg
A	1.00	0.65	0.11	0.04	0.21	0.59	0.65	0.18	0.48	0.18	0.41
B	0.65	1.00	0.20	0.13	0.07	0.53	0.48	0.14	0.40	0.27	0.39
C	0.11	0.20	1.00	-0.05	-0.09	0.03	0.12	0.37	0.00	0.48	0.22
D	0.04	0.13	-0.05	1.00	0.11	-0.10	0.14	-0.06	-0.06	0.09	0.12
E	0.21	0.07	-0.09	0.11	1.00	0.17	0.09	0.06	0.03	0.10	0.18
F	0.59	0.53	0.03	-0.10	0.17	1.00	0.39	0.23	0.43	0.19	0.35
G	0.65	0.48	0.12	0.14	0.09	0.39	1.00	0.01	0.37	0.28	0.36
H	0.18	0.14	0.37	-0.06	0.06	0.23	0.01	1.00	0.14	0.25	0.23
I	0.48	0.40	0.00	-0.06	0.03	0.43	0.37	0.14	1.00	0.05	0.29
J	0.18	0.27	0.48	0.09	0.10	0.19	0.28	0.25	0.05	1.00	0.29
Avg	0.41	0.39	0.22	0.12	0.18	0.35	0.36	0.23	0.29	0.29	**0.28**

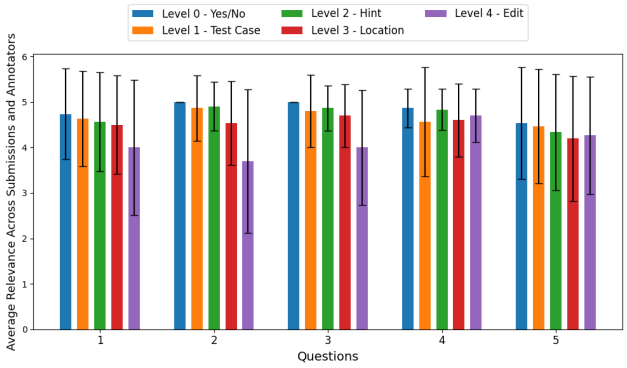
4. RESULTS AND DISCUSSION

4.1 Inter-Rater Agreement

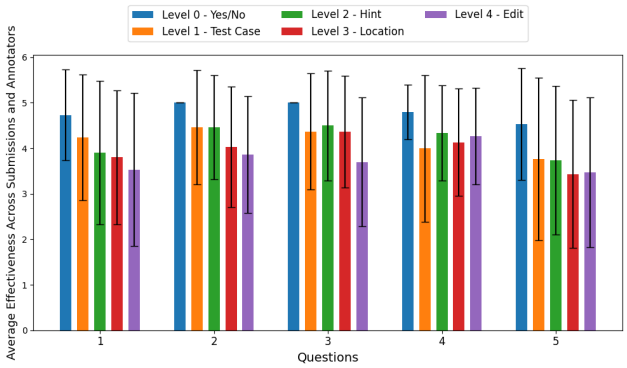
Table 3 measures the inter-rater agreement between pairs of annotators in our study using Pearson correlation coefficient (PCC) [1], with average PCC values reported at the end of the row and column for the corresponding annotators. The overall average of all the pairwise PCC values is 0.28, which indicates moderate agreement across annotators.

4.2 Trends Found in the User Study

Figure 1 shows the trend for how annotators' relevance and effectiveness ratings change across questions. There are five sets of bars, each for one question in our study. Each set contains the average score for each of the levels represented using a single bar, together with standard deviations. The scores are averaged over all scores from all the annotators and all three submissions for each question. Figure 2 shows the trend on the same evaluation metrics across different

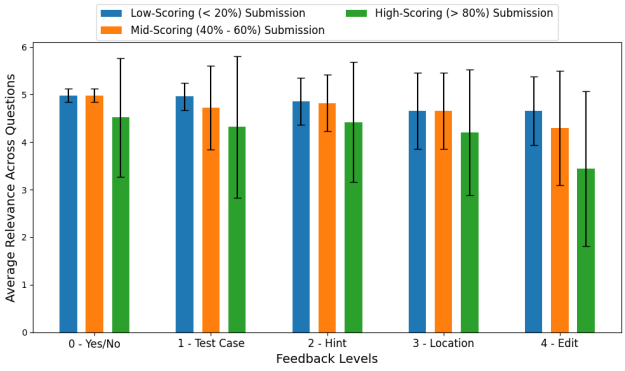


(a) Relevance

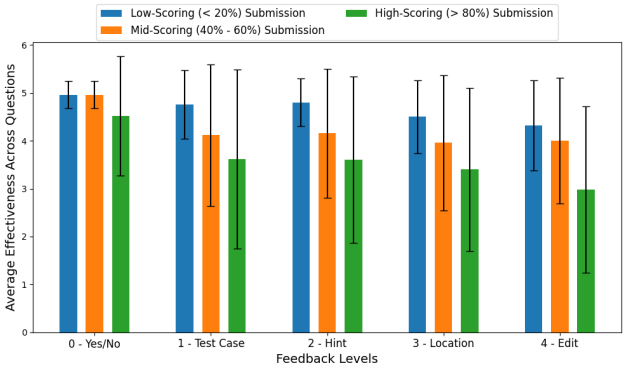


(b) Effectiveness

Figure 1: Relevance and Effectiveness score for each question, reported for each level - averaged over all three submissions and all annotators. Error margin is standard deviation.



(a) Relevance



(b) Effectiveness

Figure 2: Relevance and Effectiveness score for each level, reported for all three submissions - averaged over all problems and annotators. Error margin is standard deviation.

feedback levels. Two key observations from these results:

Quality of feedback for each level is consistent across questions. We see that the quality of feedback for lower levels is better than for higher levels in 4 out of 5 questions. In terms of relevance, our method can generate highly relevant lower-level feedback. However, for higher-level feedback, the relevance diminishes. Upon manual inspection, we observe that as the level increases, the feedback diverges from the level definition more and more. For example, level 4 feedback often has the whole program listed or contains a brand new program in the generated feedback-ladder, which violates the definition of level 4 feedback. On the other hand, the generated feedback is almost perfectly relevant at level 0 and level 1 since these levels require highly structured feedback. Overall, the method struggles to maintain relevance for levels 2, 3, and 4 a little bit more than levels 0 or 1. In terms of effectiveness, we see a similar trend as the one for relevance, except that the effectiveness at higher levels is much lower. In the annotators' opinion, listing whole programs or replacing them with a new program has a more negative impact on effectiveness compared to relevance. An exception can be found between levels 1 and 2, where even though level 1 has low effectiveness, level 2 has higher effectiveness for questions 2, 3, and 4. This means that LLM-generated hints have higher quality than generated test cases.

Quality of feedback is higher for low-scoring submissions than high-scoring submissions. Among submissions with different scores (low-scoring: $< 20\%$, mid-scoring: $40\% - 60\%$, high-scoring: $> 80\%$), we observe a clear trend that GPT-4 struggles to generate good feedback for higher-scoring submissions. This result is likely because finding a mistake in low-scoring submissions is much easier than doing so for higher-scoring submissions; high-scoring submissions are almost correct and often contain tiny mistakes that fail one or two corner test cases. Even human experts struggle to find these mistakes. We also find evidence of these situations upon manual inspection of the annotator ratings: The high-scoring submissions for Q1 and Q5 were so close to being correct that two of our annotators mistakenly believed they were correct. Therefore, they rated the generated feedback-ladder as irrelevant and ineffective at all levels.

5. CONCLUSION AND FUTURE WORKS

In our paper, GPT-4's ability to generate feedback-ladders for programming assignments is examined. While promising, our user study revealed limitations in providing highly effective feedback, especially at advanced levels. However, the method remains valuable for teachers to tailor feedback to student proficiency levels, reducing workload by addressing mistakes in low-scoring submissions. Future research could include large-scale classroom studies for real-time interventions, personalized feedback models, making feedback personalized to each student's knowledge [8] and dedicated LLM training for feedback generation to mitigate time and cost concerns associated with GPT-4.

6. REFERENCES

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

[2] Gao, Z., Erickson, B., Xu, Y., Lynch, C., Heckman, S., Barnes, T.: You asked, now what? modeling students' help-seeking and coding actions from request to resolution. Journal of Educational Data Mining **14**(3), 109–131 (2022)

[3] Johnston, K.: The effects of immediate correctness feedback on student learning, understanding, and achievement (2015), `https://api.semanticscholar.org/CorpusID:188952447`

[4] Kochmar, E., Vu, D.D., Belfer, R., Gupta, V., Serban, I., Pineau, J.: Automated personalized feedback improves learning gains in an intelligent tutoring system. Artificial Intelligence in Education **12164**, 140 – 146 (2020), `https://api.semanticscholar.org/CorpusID:218516674`

[5] Kumar, N.A., Lan, A.: Improving socratic question generation using data augmentation and preference optimization (2024)

[6] Kumar, N.A., Lan, A.: Using large language models for student-code guided test case generation in computer science education (2024)

[7] Leinonen, J., Hellas, A., Sarsa, S., Reeves, B., Denny, P., Prather, J., Becker, B.A.: Using large language models to enhance programming error messages. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. pp. 563–569 (2023)

[8] Liu, N., Wang, Z., Baraniuk, R., Lan, A.: Open-ended knowledge tracing for computer science education. In: EMNLP (2022)

[9] Marwan, S., Jay Williams, J., Price, T.: An evaluation of the impact of automated programming hints on performance and learning. In: Proceedings of the 2019 ACM Conference on International Computing Education Research. pp. 61–70 (2019)

[10] Miwa, K., Terai, H., Kanzaki, N., Nakaike, R.: Stoic behavior in hint seeking when learning using an intelligent tutoring system. In: Proceedings of the Annual Meeting of the Cognitive Science Society. vol. 35 (2013)

[11] Paaßen, B., Hammer, B., Price, T.W., Barnes, T., Gross, S., Pinkwart, N.: The continuous hint factory-providing hints in vast and sparsely populated edit distance spaces. arXiv preprint arXiv:1708.06564 (2017)

[12] Pankiewicz, M., Baker, R.S.: Large language models (gpt) for automating feedback on programming assignments. arXiv preprint arXiv:2307.00150 (2023)

[13] Phung, T., Cambronero, J., Gulwani, S., Kohn, T., Majumdar, R., Singla, A., Soares, G.: Generating high-precision feedback for programming syntax errors using large language models. arXiv preprint arXiv:2302.04662 (2023)

[14] Phung, T., Pădurean, V.A., Cambronero, J., Gulwani, S., Kohn, T., Majumdar, R., Singla, A., Soares, G.: Generative ai for programming education: Benchmarking chatgpt, gpt-4, and human tutors. International Journal of Management **21**(2), 100790 (2023)

[15] Phung, T., Pădurean, V.A., Singh, A., Brooks, C., Cambronero, J., Gulwani, S., Singla, A., Soares, G.: Automating human tutor-style programming feedback: Leveraging gpt-4 tutor model for hint generation and gpt-3.5 student model for hint validation. arXiv preprint arXiv:2310.03780 (2023)

[16] Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M., Guibas, L.: Learning program embeddings to propagate feedback on student code. In: International conference on machine Learning. pp. 1093–1102. PMLR (2015)

[17] Price, T., Shi, Y.: Codeworkout data spring 2019. 2nd CSEDM Data Challenge (2021), `https://sites.google.com/ncsu.edu/csedm-dc-2021/`

[18] Sheese, B., Liffiton, M., Savelka, J., Denny, P.: Patterns of student help-seeking when using a large language model-powered programming assistant. arXiv preprint arXiv:2310.16984 (2023)

[19] Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback. In: International Conference on Machine Learning. pp. 10799–10808. PMLR (2020)

[20] Yasunaga, M., Liang, P.: Break-it-fix-it: Unsupervised learning for program repair. In: International Conference on Machine Learning. pp. 11941–11952. PMLR (2021)