Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation

Tung Phung
mphung@mpi-sws.org
MPI-SWS
Saarbrücken, Germany

Victor-Alexandru Pădurean
vpadurea@mpi-sws.org
MPI-SWS
Saarbrücken, Germany

Anjali Singh
singhanj@umich.edu
University of Michigan
Ann Arbor, USA

Christopher Brooks*
brooksch@umich.edu
University of Michigan
Ann Arbor, USA

José Cambronero*
jcambronero@microsoft.com
Microsoft
Redmond, USA

Sumit Gulwani*
sumitg@microsoft.com
Microsoft
Redmond, USA

Adish Singla*
adishs@mpi-sws.org
MPI-SWS
Saarbrücken, Germany

Gustavo Soares*
gsoares@microsoft.com
Microsoft
Redmond, USA

ABSTRACT

Generative AI and large language models hold great promise in enhancing programming education by automatically generating individualized feedback for students. We investigate the role of generative AI models in providing human tutor-style programming hints to help students resolve errors in their buggy programs. Recent works have benchmarked state-of-the-art models for various feedback generation scenarios; however, their overall quality is still inferior to human tutors and not yet ready for real-world deployment. In this paper, we seek to push the limits of generative AI models toward providing high-quality programming hints and develop a novel technique, GPT4Hints-GPT3.5Val. As a first step, our technique leverages GPT-4 as a "tutor" model to generate hints – it boosts the generative quality by using symbolic information of failing test cases and fixes in prompts. As a next step, our technique leverages GPT-3.5, a weaker model, as a "student" model to further validate the hint quality – it performs an automatic quality validation by simulating the potential utility of providing this feedback. We show the efficacy of our technique via extensive evaluation using three real-world datasets of Python programs covering a variety of concepts ranging from basic algorithms to regular expressions and data analysis using *pandas* library.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; • **Computing methodologies** → **Artificial intelligence**.

*These authors are listed in alphabetical order.

KEYWORDS

1 INTRODUCTION

Generative AI and large language models (LLMs) have the potential to drastically improve the landscape of computing and programming education by powering next-generation educational technologies. This potential lies in the advanced capabilities of state-of-the-art models—like OpenAI's GPT-4 [23] and ChatGPT (based on GPT-3.5) [22]—to automatically generate high-quality personalized content and feedback for students [4, 27, 32]. A series of recent works have already shown us sparks of their capabilities for various programming education scenarios, including generating new programming assignments [24, 29], providing code explanations [15, 29], repairing buggy programs [26, 35], enhancing programming-error-messages [13, 26], and acting as pair programmer [9, 19].

In this paper, we investigate the role of LLMs in providing human tutor-style programming hints to help students resolve errors in their buggy programs. More concretely, given a programming task and a student's buggy program, we want to generate natural language hints to help the student resolve bug(s) and make progress, inspired by how a human tutor would give pedagogical feedback. With the current scale of enrollments in introductory programming courses [18], it has become infeasible for human tutors to promptly provide individualized feedback to students, thereby motivating

| Given a string S, check if it is palindrome or not. | Expected Time Complexity: $O(|S|)$. |
| | Constraints: $1 \leq |S| \leq 2 * 10^5$. |

(a) Description of the programming task

```
1  #User function Template for python3
2  class Solution:
3    def reverse(self, b, e, S):
4      while b < e:
5        S[b], S[e] = S[e], S[b]
6        e = e - 1
7        b = b + 1
8      return S
9
10   def isPalindrome(self, S):
11     S = list(S)
12     beg = 0
13     end = len(S) - 1
14     rev = self.reverse(beg,end,S)
15     if S == rev:
16       return 1
17     return 0
18
19 #{ Driver Code Starts
20 #Initial Template for Python 3
21 if __name__ == '__main__':
22   T=int(input())
23   for i in range(T):
24     S = input()
25     ob = Solution()
26     answer = ob.isPalindrome(S)
27     print(answer)
28 # } Driver Code Ends
```

(b) Student's buggy program

```
1  #User function Template for python3
2  class Solution:
3    def reverse(self, b, e, S):
4      while b < e:
5        S[b], S[e] = S[e], S[b]
6        e = e - 1
7        b = b + 1
8      return S
9
10   def isPalindrome(self, S):
11     S = list(S)
12     beg = 0
13     end = len(S) - 1
14     rev = self.reverse(beg,end,S.copy())
15     if S == rev:
16       return 1
17     return 0
18
19 #{ Driver Code Starts
20 #Initial Template for Python 3
21 if __name__ == '__main__':
22   T=int(input())
23   for i in range(T):
24     S = input()
25     ob = Solution()
26     answer = ob.isPalindrome(S)
27     print(answer)
28 # } Driver Code Ends
```

(c) Fixed program

For Input: hq

Your Code's output is:

1

It's Correct output is:

0

(d) Failing test case

The bug in the program is that the original string S is being modified in the reverse function. When you compare the reversed string with the original string, they are always the same because they are both pointing to the same modified list. The fix is to pass a copy of the original string to the reverse function so that the original string remains unmodified.

(e) Detailed explanation

Consider the effect of Python's pass-by-reference behavior on mutable objects like lists when you're reversing the string.

(f) Single-sentence hint

(g) Validation

Figure 1: Illustrative example showcasing GPT4Hints-GPT3.5Val for the Palindrome problem shown in (a) from the BasicAlgo dataset. (b) shows a real-world buggy program. (c) shows a fixed program generated by the technique in an intermediate step, and (d) shows a test case where the buggy program fails to produce the correct output. (e) shows a detailed explanation generated by the technique that is used later in the validation stage. (f) shows the generated feedback (a single-sentence hint). (g) highlights that the validation stage of the technique *successfully accepted* the generated feedback as high-quality and suitable for sharing with the student.

the need to develop automatic feedback generation techniques. To this end, we aim to leverage generative AI and LLMs for automating human tutor-style programming feedback to support students' learning and reduce human tutors' workload.

Recent works have studied state-of-the-art LLMs for generating various forms of programming feedback for students, including detailed explanations about bugs or single-sentence hints [13, 26, 27]. Despite promising initial results, the overall quality of feedback generated by LLMs is substantially inferior to that of human tutors and not yet ready for deployment in real-life classroom settings. For instance, a recent benchmark study in [27] evaluated GPT-4 in generating hints for buggy programs on introductory Python programming tasks and assessed its quality performance using expert annotations – GPT-4's performance in terms of hints quality is only about 60% in contrast to human tutors's performance of over 90%. This performance gap between GPT-4 vs. human tutors can be attributed to several factors, as discussed next. First, state-of-the-art models still struggle with symbolic reasoning and program execution abilities crucial for understanding the underlying bugs and possible student misconceptions [2, 4, 27, 32]. Second, these models also suffer from hallucination issues and the generated feedback text—even though seemingly plausible—may contain inaccurate information that could have detrimental effects on students' learning [2, 12, 14]. Third, these models still lack a calibration mechanism to decide whether the generated content is of high quality or not [26]; in particular, they are unable to do a human tutor-style reasoning from a student's perspective and judge if the generated feedback would likely help the student.

The motivation of the problem is to investigate any evidence of a link between vaccine efficacy and sex of the child. For this, you should compute the ratio of the number of children who contracted chickenpox but were vaccinated against it (at least one varicella dose) versus those who were vaccinated but did not contract chicken pox. Return results by sex.	Design your function so that it returns a dictionary in the following format: {"male": 0.2, "female": 0.4}

<div align="center">(a) Description of the programming task</div>

```python
def chickenpox_by_sex():
    import pandas as pd
    import numpy as np
    df = pd.read_csv('assets/data.csv', index_col
        =0)
    male = df[df['SEX'] == 1]
    vac_m = male[male['P_NUMVRC'] == 1]
    cpox_m = vac_m[vac_m['HAD_CPOX'] == 1]
    no_m = vac_m[vac_m['HAD_CPOX'] == 2]
    total_cpox_m = cpox_m['SEX'].count()
    total_nocpox_m = no_m['SEX'].count()
    m = total_cpox_m/total_nocpox_m

    ... code omitted to save space ...

    ans={"male":  m,
        "female": f}

    return ans
```

```python
def chickenpox_by_sex():
    import pandas as pd
    import numpy as np
    df = pd.read_csv('assets/data.csv', index_col
        =0)
    male = df[df['SEX'] == 1]
    vac_m = male[male['P_NUMVRC'] >= 1]
    cpox_m = vac_m[vac_m['HAD_CPOX'] == 1]
    no_m = vac_m[vac_m['HAD_CPOX'] == 2]
    total_cpox_m = cpox_m['SEX'].count()
    total_nocpox_m = no_m['SEX'].count()
    m = total_cpox_m/total_nocpox_m

    ... code omitted to save space ...

    ans={"male":  m,
        "female": f}

    return ans
```

Your Code's
output is:
{'male': 0.0092887664,
'female': 0.0076827368 }

It's Correct
output is:
{'male': 0.0097655722,
'female': 0.0076827368 }

<div align="center">(b) Student's buggy program　　　　　　　(c) Fixed program　　　　　　　(d) Failing test case</div>

The bug in the program is in the line where you are filtering the male children who were vaccinated. You are only considering those who received exactly one dose of the vaccine ('male['P_NUMVRC'] == 1'), but the problem statement asks for children who received at least one dose of the vaccine. The fix is to change the condition to 'male['P_NUMVRC'] >= 1'.	Check the conditions you are using to filter the vaccinated children, especially the number of doses they received.	✓

<div align="center">(e) Detailed explanation　　　　　　　(f) Single-sentence hint　　　　　　　(g) Validation</div>

Figure 2: Similar to Figure 1, this example showcases GPT4HINTS-GPT3.5VAL on a buggy program from the DATAANALYSIS dataset.

1.1 Our Approach and Contributions

In this paper, we seek to push the limits of generative AI and state-of-the-art LLMs toward providing high-quality programming hints. Given a base model, this would require improving the model's abilities at input-level by developing better prompting strategies [34], at output-level by developing mechanisms to validate the generated content [5, 16, 26], or at model-level itself by fine-tuning (when considering open-source models [33]). In our work, we consider OpenAI's GPT-4 [23] as the base model—the latest model presumably with over a trillion parameters—as it has shown to drastically improve existing models across various programming education scenarios [27].

We develop a novel technique, GPT4HINTS-GPT3.5VAL, to provide human tutor-style high-quality programming hints. Our technique leverages the GPT-4 model in the role of a "tutor" to generate hints and boosts the generative quality at the input level by prompting it with symbolic information of failing test cases and fixed programs. At the output level, it further validates the hint quality by leveraging the GPT-3.5 model as a "student" to simulate the potential utility of providing this feedback to human students. This validation step is designed to provide a quality assurance layer and decides whether the generated feedback should be provided to the

human student or not – thereby trading off *coverage* (how many students are given automatic feedback) and *precision* (quality of the given feedback). We show the efficacy of our technique by conducting an extensive evaluation using three real-world datasets of Python programs covering a variety of concepts ranging from writing basic algorithms to regular expressions and data analysis using *pandas* [17]. Figures 1 and 2 showcase GPT4HINTS-GPT3.5VAL on two different buggy programs.[1] More broadly, our work makes the following contributions in leveraging generative AI and LLMs for computing and programming education:

I. We showcase the utility of prompting the models with symbolic information, such as failing test cases and fixed programs, to enhance their reasoning abilities about the underlying bugs crucial for providing high-quality hints.

II. We showcase the utility of using LLMs in a flipped role as a "student" model to simulate the potential effect of feedback on real human students. Our results highlight that using a

---

[1]When presenting these illustrative examples in this paper, we slightly obfuscate the students' buggy programs to avoid showing exact real-world programs. We do so by altering variable names and formatting conventions while preserving the original bugs exactly the same, as considered in related works [26, 27]. Accordingly, if needed, we apply the same adjustments to the generated output to maintain consistency with these alterations.

weaker model (GPT-3.5, instead of GPT-4) provides better validation of programming hints from GPT-4. This flipped role opens up new opportunities in utilizing generative AI for in-context student modeling for automatic assessments, learning analytics, and simulations.

III. Our technique achieves a precision of around 95% (reaching the quality of human tutors in our evaluation) while maintaining a high coverage of over 70% across three real-world Python programming datasets.[2]

1.2 Related Work

Feedback generation for programming education. Prior to recent developments in generative AI and LLMs, the research on feedback generation for programming education had primarily focused on fixing buggy programs because of challenges in automatically generating natural language explanations [10, 31]. A parallel line of research explored crowdsourcing approaches to obtain explanations provided by other students/tutors [11]. Our work builds on recent developments in leveraging LLMs for generating programming feedback [13, 25–27], in particular, motivated by recent survey [27] that highlighted a substantial gap in GPT-4's performance in terms of hints quality w.r.t. human tutors. Another closely related work is [26] that proposed PyFiXV technique for generating high-precision feedback for syntax errors. PyFiXV has a run-time feedback validation mechanism by leveraging OpenAI's Codex-Edit model [21] at varying temperatures as a "student" model. Inspired by [26], we also leverage an LLM-based "student" model to perform validation. However, the validation mechanism used in PyFiXV is not directly applicable to our setting as it is designed only for syntax errors that substantially simplify the validation process; crucially, GPT4Hints-GPT3.5Val is designed to provide feedback for any types of errors a student might encounter, including errors related to the program's time complexity.

Enhancing a model's generative performance. A series of recent works have focused on enhancing the generative performance of a base model in a *black-box setting*, given the high monetary or computational costs involved in fine-tuning state-of-the-art models (in fact, the latest OpenAI's GPT-4 model doesn't have public APIs for fine-tuning). These works operate either at the input level by developing better prompting strategies [34] or at the output level by analyzing and correcting the generated content [5, 16, 26]. At the output level enhancements, *Self-Debugging* [5] and *Self-Refine* [16] are two recently proposed methods that enable an LLM to analyze and correct its output automatically. Another recent work in [20] introduced the concept of *Self-Repair* that showed substantial performance gains when allowing an LLM to repair its output by receiving feedback from a more powerful LLM or expert. The key intuition behind the validation mechanism in GPT4Hints-GPT3.5Val differs from these works and is more related to [26] discussed above—we utilize another LLM as a "student" model to simulate the potential effect of feedback on real human students.

Integration of generative AI in educational sites. There has also been increasing interest in integrating generative AI and LLMs in educational sites. For instance, Khanmigo [1] by Khan Academy and Q-Chat by Quizlet [28] are AI-powered systems based on OpenAI's GPT models. These recent developments also serve as our motivation to develop principled techniques that can generate high-quality feedback. Overall, we see our work as complementary to these systems and believe that the proposed techniques can be useful in further improving the performance of these systems.

2 PROBLEM SETUP

Programming task and student's buggy program as input. We start with a programming task $\mathcal{T}$ and a buggy program $\mathcal{P}_b$. A task $\mathcal{T}$, such as shown in Figures 1a and 2a, is represented by a textual description of the programming problem. Additionally, this description encompasses all requisite information essential for problem solving, such as expected algorithm complexity and any constraints on input, as applicable. In cases where the task necessitates interaction with an external file, $\mathcal{T}$ should also contain all pertinent information of that file crucial for solving the problem, such as the file's format or structure. $\mathcal{P}_b$, as illustrated in Figures 1b and 2b, is an unsuccessful attempt of the student to solve $\mathcal{T}$. This program fails to pass at least one of the test cases in the test suite for $\mathcal{T}$. In general, $\mathcal{P}_b$ may contain one or multiple errors, spanning various error types including syntax and semantic errors.

Tutor-style hint as output and quality assessment. Given $\mathcal{T}$ and $\mathcal{P}_b$, we aim to generate a human tutor-style natural language hint $\mathcal{H}$ as feedback to aid the student in understanding and resolving the programming error. We assess the quality of generated feedback along four quality attributes following the rubric used in [27]. All attributes are binary, with a value of 1 being better. *HCorrect* captures whether the generated hint provides correct information for resolving issues in the student's buggy program. *HInformative* captures whether the generated hint provides useful information to help the student resolve bug(s); this attribute is set to 0 by default when the hint is incorrect. *HConceal* captures that the information in the generated hint is not too detailed, so the student would also have to reason about implementing the fixes; this attribute is set to 0 by default when the hint is incorrect. *HComprehensible* captures whether the generated hint is easy to understand, presented in a readable format, and doesn't contain redundant information. In our evaluation, human experts (evaluators) assess the quality of generated hints along these four attributes. We measure the overall quality of the generated hint by *HOverall* that takes the value of 1 (good quality) if all the four quality attributes are satisfied and otherwise 0 (bad quality).

Performance metrics and objective. Next, we describe the overall performance metrics used to evaluate a feedback generation technique. For a given student's buggy program $\mathcal{P}_b$, we seek to design techniques that generate feedback and also decide whether the generated feedback is suitable for sharing with the student. Similar to [26], we measure the performance of a technique using two metrics: (i) *Coverage* measuring the percentage number of times the generated feedback is provided to the student; (ii) *Precision* measuring the percentage number of times the provided feedback is of good quality w.r.t. the *HOverall* quality introduced above. In our experiments, we will compute these metrics on a dataset comprising a set of students' buggy programs. Our goal is to design feedback generation techniques with high precision, which is imperative before deploying such techniques in classrooms. In particular, we aim to develop

---

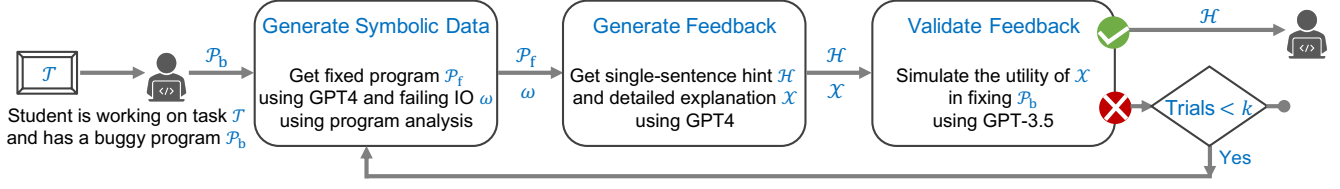[2]https://github.com/machine-teaching-group/lak2024_GPT4-Hints-GPT3.5Val

Figure 3: Illustration of different stages in GPT4Hints-GPT3.5Val's feedback generation process.

techniques that achieve a precision level of human tutors while maintaining an effective trade-off between precision and coverage.

3 OUR TECHNIQUE: GPT4Hints-GPT3.5Val

This section gives details about our proposed technique, namely GPT4Hints-GPT3.5Val, which leverages and improves upon generative AI models for feedback generation. Figure 3 shows an overview of our technique. In essence, GPT4Hints-GPT3.5Val employs GPT-4 as a simulated "tutor" model for generating feedback and GPT-3.5 as a simulated "student" model for feedback validation. In Section 3.1, we describe two types of symbolic information that are helpful for generating feedback and how to obtain them; in Section 3.2, we describe the process of feedback generation augmented with this symbolic information. Subsequently, in Section 3.3, we introduce a novel validation mechanism aiming to elevate the precision of the delivered feedback while maintaining a high level of coverage.

3.1 Stage-1: Generate Symbolic Data

Overview and intuition. As discussed in Section 1, there remains a notable performance gap between state-of-the-art generative AI models and human tutors regarding hint generation. One key factor contributing to this disparity is the inability to do symbolic reasoning and program execution. GPT-4 lacks the capability to execute the given code to retrieve an output, which could help it gain deeper understanding of the underlying bugs. To mitigate this gap, we employ external tools to execute programs and extract useful symbolic information. We then supply this information to GPT-4 for feedback generation. Our approach centers on leveraging two categories of symbolic data: failing test cases and fixed programs.

Input/output for a failing test case. To highlight the error in the buggy program $\mathcal{P}_b$, we provide GPT-4 with a test case for which $\mathcal{P}_b$ fails to produce the expected output. To acquire this test case, we run $\mathcal{P}_b$ on the existing test suite given for the corresponding task $\mathcal{T}$. The first test case in which $\mathcal{P}_b$ fails is selected. We denote the triplet comprising this input, the output generated by $\mathcal{P}_b$, and the expected output, as ω and include it in the prompt for feedback generation.

Fixed program. The fixed program, denoted as $\mathcal{P}_f$, is generated using GPT-4, employing a procedure adapted from the work in [26]. To be more specific, we initiate the process by requesting the model to produce 10 independent fixed programs. For this purpose, we include $\mathcal{T}$ and $\mathcal{P}_b$ in the prompt[3] to ask for 10 outputs (each output contains a fixed program) with the hyperparameter *temperature* set

to 0.5. Then, from this set of 10, we take the programs that pass the test suite for $\mathcal{T}$ and among them, identify $\mathcal{P}_f$ as the one with the smallest token-edit distance w.r.t. $\mathcal{P}_b$. To compute the token-edit distance between two programs, we first tokenize them using Pygments library [3] and then calculate the Levenshtein edit distance based on the tokenized strings. If $\mathcal{P}_f$ is found, we include it in the prompt for feedback generation. If, however, none of the generated programs is correct, we opt to exclude this symbolic information from the prompt.

3.2 Stage-2: Generate Feedback

Overview and intuition. In this stage, we aim to obtain a human tutor-style hint $\mathcal{H}$ as feedback to be given to the student, as previously mentioned in Section 2. In addition to our request for a hint $\mathcal{H}$ from GPT-4, we also ask for a detailed explanation, denoted as $\mathcal{X}$, for the bugs in $\mathcal{P}_b$. The reason to ask for this explanation draws inspiration from Chain-of-Thought [34], an established method renowned for enhancing the reasoning capabilities of LLMs. The essence of the Chain-of-Thought approach lies in encouraging LLMs to explain their thought process meticulously, step by step, prior to presenting the final output. Within the specific context of hint generation, we allow the model to elaborate its reasoning through $\mathcal{X}$ before coming up with the concise single-sentence hint $\mathcal{H}$, which is essentially an abstracted version of the explanation. Furthermore, $\mathcal{X}$ will also play a pivotal role in the subsequent feedback validation stage, which will be elaborated upon in Section 3.3.
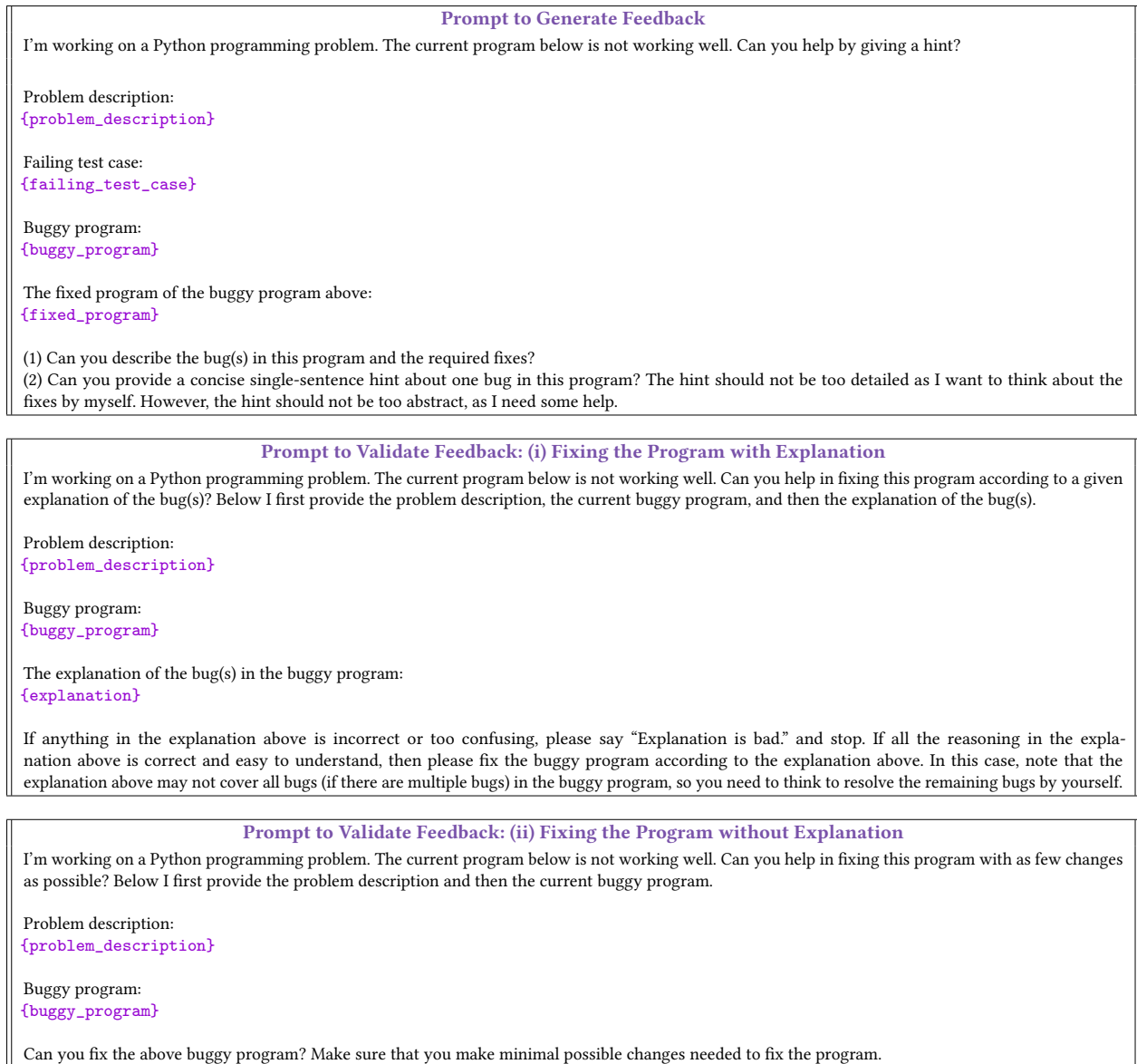
Prompt for feedback generation. In Figure 4 (first prompt), we provide our prompt for generating feedback. This prompt comprises the problem description for $\mathcal{T}$, the buggy program $\mathcal{P}_b$, the symbolic information as extracted from the previous stage, and a request for an explanation $\mathcal{X}$ along with a hint $\mathcal{H}$. To get a response from GPT-4, we use this prompt while configuring the hyperparameter *temperature* to 0, indicating our preference for the most probable answer. All other hyperparameters are kept at their default settings. Following this, $\mathcal{X}$ and $\mathcal{H}$ are then extracted automatically from the output.

3.3 Stage-3: Validate Feedback

Overview and intuition. This validation stage aims to enhance the precision of the feedback provided to the student. It is worth noting that despite the inclusion of augmented symbolic information in the prompt, the hint generated in Stage-2 may not always align with the desired quality criteria outlined in Section 2. To mitigate this issue, we introduce a validation mechanism that adds a run-time quality assurance layer and decides whether the generated feedback is suitable for sharing with the student. The key idea behind this validation mechanism is to leverage an additional AI model as a

---

[3]The prompt used here has the same format as shown in Figure 4 (third prompt).

Prompt to Generate Feedback

I'm working on a Python programming problem. The current program below is not working well. Can you help by giving a hint?

Problem description:
{problem_description}

Failing test case:
{failing_test_case}

Buggy program:
{buggy_program}

The fixed program of the buggy program above:
{fixed_program}

(1) Can you describe the bug(s) in this program and the required fixes?
(2) Can you provide a concise single-sentence hint about one bug in this program? The hint should not be too detailed as I want to think about the fixes by myself. However, the hint should not be too abstract, as I need some help.

Prompt to Validate Feedback: (i) Fixing the Program with Explanation

I'm working on a Python programming problem. The current program below is not working well. Can you help in fixing this program according to a given explanation of the bug(s)? Below I first provide the problem description, the current buggy program, and then the explanation of the bug(s).

Problem description:
{problem_description}

Buggy program:
{buggy_program}

The explanation of the bug(s) in the buggy program:
{explanation}

If anything in the explanation above is incorrect or too confusing, please say "Explanation is bad." and stop. If all the reasoning in the explanation above is correct and easy to understand, then please fix the buggy program according to the explanation above. In this case, note that the explanation above may not cover all bugs (if there are multiple bugs) in the buggy program, so you need to think to resolve the remaining bugs by yourself.

Prompt to Validate Feedback: (ii) Fixing the Program without Explanation

I'm working on a Python programming problem. The current program below is not working well. Can you help in fixing this program with as few changes as possible? Below I first provide the problem description and then the current buggy program.

Problem description:
{problem_description}

Buggy program:
{buggy_program}

Can you fix the above buggy program? Make sure that you make minimal possible changes needed to fix the program.

Figure 4: Prompts employed by GPT4HINTS-GPT3.5VAL for feedback generation (first) and feedback validation (second and third).

"student" model to simulate the potential utility of providing this feedback to human students. More concretely, we seek to evaluate the quality of feedback by assessing its impact on the simulated students' ability to fix the bugs. If the simulated students find it easier to fix $\mathcal{P}_b$ with the help of the feedback, then the feedback is deemed high-quality and can be subsequently provided to the real student. In terms of the "student" model, we use a weaker model GPT-3.5, instead of GPT-4. The key intuition is that a weaker model provides a better differential effect in quantifying the utility of feedback in fixing the buggy program; moreover, we use the "student" model at a high temperature to add further stochasticity

in the process of fixing the program.[4] Furthermore, we will use the detailed explanation $\mathcal{X}$ (instead of the single-sentence hint $\mathcal{H}$) to assess the utility of feedback for fixing the bugs. In our evaluation (Section 4.4 and Figure 7), we will demonstrate the effectiveness of these design choices.

Two prompts for validation. Figure 4 (second and third prompts) illustrates the two prompts used by the feedback validation mechanism. Both prompts essentially instruct the "student" model (GPT-3.5) to fix $\mathcal{P}_b$. The primary distinction lies in the fact that, in contrast to the third (standard) prompt, the second (augmented) prompt

---

[4]We refer the reader to recent results in [27, 30] to see the performance of different GPT-based models across various programming education scenarios.

additionally incorporates the explanation $\mathcal{X}$. More concretely, the third (standard) prompt is the same as the prompt used in Stage-1 when generating a fixed program; the second (augmented) prompt puts emphasis on the detailed explanation to serve as an instruction for the "student" model when fixing the program. For each prompt, we ask GPT-3.5 to generate a set of $n = 10$ independent outputs (the *temperature* is set to 0.5, similar to in Stage-1), effectively utilizing GPT-3.5 in the role of 10 simulated students. We shall denote the number of correct output programs resulting from the standard prompt as n_1, and the number of correct output programs resulting from the augmented prompt as n_2. The correctness of a program is determined by its ability to pass the whole test suite for the corresponding task $\mathcal{T}$. Next, we explain how we use these quantities for feedback validation.

Validation threshold rules. Our main idea for validation is that good feedback should help students find it easier to fix the buggy program than without it. Thus, the primary rule for feedback validation is to have $\frac{n_2}{n} \geq \frac{n_1}{n}$. Nonetheless, in situations where n_1 assumes particularly low values, e.g., $n_1 = 0$ or $n_1 = 1$, this condition becomes less stringent, and any feedback, regardless of its quality, may pass the validation. To address this, we incorporate an additional requirement to ensure that $\frac{n_2}{n}$ attains a sufficient level independently. This is achieved through the inclusion of the following condition: $\left(\frac{n_2}{n} \geq \alpha \right) \vee \left(\frac{n_2}{n} \geq \frac{n_1}{n} + \beta \right)$, where we instantiate α as 0.50 and β as 0.25. In other words, we require the ratio of correct output programs generated with the help of the explanation to either exceed a certain fixed threshold (i.e., $\frac{n_2}{n} \geq 0.5$) or be substantially higher than the ratio of correct output programs generated without the explanation (i.e., $\frac{n_2}{n} \geq \frac{n_1}{n} + 0.25$), or both. Consequently, our final validation mechanism approves a feedback instance only when the following condition holds true: $\left(\left(\frac{n_2}{n} \geq \frac{n_1}{n} \right) \wedge \left(\left(\frac{n_2}{n} \geq 0.50 \right) \vee \left(\frac{n_2}{n} \geq \frac{n_1}{n} + 0.25 \right) \right) \right)$, and rejects it otherwise. In our experiments (Section 4), we will also compare the performance of different variants of threshold rules.

Multiple trials. When the validation mechanism rejects a feedback instance, it is not provided to the human student. While this is expected to boost the precision metric, it could also lead to a significant drop in the coverage metric [26]. Given the stochasticity of the generation and validation processes, we introduce an additional layer to the overall process to boost the coverage while ensuring high precision. More concretely, if a feedback instance is rejected, we restart the process, including acquiring symbolic information, generating hints, and the subsequent validation. We maintain this iterative cycle until either a generated feedback instance is approved by the validation mechanism or a predefined maximum number of iterations, denoted as k, is attained (we set $k = 3$). After k trials, if none of the feedback instances pass validation, we terminate this outer loop and will not provide any feedback to the human student. When deploying our technique in real-world classroom settings, where no automatic feedback is being provided, a human tutor could step in and take over the work of providing feedback to the student.

4 EXPERIMENTAL EVALUATION

In this section, we evaluate our technique, GPT4Hints-GPT3.5Val, across three datasets spanning different domains of introductory Python programming. We assess GPT4Hints-GPT3.5Val in comparison to baselines such as GPT-4 and human tutors. Furthermore, we compare our validation with various alternative variants. In our experiments, we use OpenAI's GPT-4 (model=*gpt-4-0613*) as the "tutor" model and ChatGPT based on GPT-3.5 (model=*gpt-3.5-turbo-0613*) as the "student" model unless otherwise stated.

4.1 Datasets

To comprehensively assess the techniques' performance across diverse domains within introductory programming education, we use three datasets representing different types of learning objectives, as summarized in Figure 5. All datasets consist of students' Python buggy programs. Below, we provide a detailed description of each of these datasets.

The first dataset, BasicAlgo, was introduced in [27]. It covers five popular introductory Python problems, and for each problem, there are five corresponding buggy programs. The problems capture a diverse set of basic programming concepts and include the following: GCD (finding the greatest common divisor of two given numbers), Fibonacci (generating the list of Fibonacci numbers up to a given value), DivisorsDiv3 (counting the number of divisors that divide 3 of a given number), Palindrome (checking whether a given string is palindrome or not), and MergeStrs (merging two given strings alternatively). The buggy programs come from different users on the *geeksforgeeks.org* platform [8], and capture a variety of bug types and code lengths. Figures 1 and 10 show two examples of buggy programs with bugs related to misconception regarding the mutability of lists and a mistake regarding the ordering of the merging strings.

The second dataset, DataRegex, comes from an introductory data science programming course. This course is a part of an online Master's degree program in applied data science; students enrolling in the course are required to have basic Python programming and statistics knowledge. We examine the second exercise from the first assignment of the course, which requires students to use regular expressions to extract information from a text file. In particular, the text file contains people's names and their corresponding grades; the students need to fix a given buggy function so that it correctly reads the file, matches a regular expression, captures and returns a list of people who got a grade of B.[5] To solve the problem, students need knowledge of basic regular expression concepts such as wildcard characters, grouping, look around, and quantification. This dataset contains 24 buggy submissions, each from a unique student. For each student, if there are multiple buggy submissions, we take only the median submission w.r.t to submission times to include in the dataset. Some common types of bugs are mishandling of grouping (Figure 9), returning names of all people, and returning only people's last names. It is worth noting that there is only one test case in the test suite for this problem; this is in contrast to algorithmic problems, such as the ones in BasicAlgo, in which the test suites usually comprise a large number of input/output cases.

The third dataset, DataAnalysis, is from the second exercise of the second assignment in the same data science course. By that time, the students learnt to use data manipulation libraries such as *pandas*

---

[5] For GPT-4, instead of giving it the file, we describe the file format in the prompt; the description is provided as part of our implementation (see Footnote 2).

Properties	BASICALGO	DATAREGEX	DATAANALYSIS
Number of programming tasks	5	1	1
Number of buggy programs	25	24	30
Average lines of student code	10.7	2.2	12.1
Task's objective	Write an algorithm in Python	Fix a regular expression in Python	Perform data analysis in Python
Domain and concepts	Python syntax, basic algorithms	Regular expressions, information extraction	*pandas* library, data analysis

Figure 5: Overview of the datasets used in this work. See Section 4.1 for details.

to load, filter, and extract meaningful information from data-frames. For this problem, the students are given a *csv* format file that contains a data-frame, a 252-page data guide PDF,[6] a problem description, and a function signature. The students need to complete the given empty function to compute the ratios of vaccinated children who contracted chickenpox versus those who were vaccinated but did not contract chickenpox, separated by sex. To solve this problem, besides the basic Python syntax, the students also need to know how to select and use relevant libraries (such as *pandas*), understand and search for relevant information from the extensive data guide, and deal with missing data. To form this third dataset, we sample 30 buggy programs using the same procedure as used for second dataset. Some bugs in the dataset are: mis-filtering of data (Figure 2), misreading of the requirements and computing a wrong ratio, and forgetting to handle or wrongly handling of missing values.

4.2 Baselines and Variants of Our Technique

Baseline GPT-4 and human tutors. As our first baseline, we employ GPT-4 in a straightforward manner by presenting it with the task description and the buggy program in the prompt to generate feedback. The format of the prompt closely resembles that depicted in Figure 4 (first prompt), albeit without the inclusion of additional symbolic information. The second baseline employs human tutors with experience in Python programming and tutoring, which serves as the gold standard for our technique to match. In our experiments, two human tutors are employed to give hints independently. From here on, we refer to these baselines as GPT4HINTS-BASE and TUTORHINTS, respectively.

Variants of our technique without validation. As mentioned previously, we introduce two additional types of symbolic information into our prompt for feedback generation. These additions consist of a failing test case and a fixed program, given that a correct fixed program can be produced (see Section 3.1). Accordingly, we have formulated two variant techniques: (i) GPT4HINTS-IO involves enhancing GPT4HINTS-BASE by incorporating the failing test case into the prompt; (ii) GPT4HINTS-IOFIX integrates both of these types of symbolic information into the prompt. Note that neither of these techniques employ any validation, i.e., the generated feedback is always deemed suitable for sharing.

Variations of validation stage in our technique. Next, we will consider variants of GPT4HINTS-GPT3.5VAL in terms of the validation stage. First, we look at the role of multiple trials when a feedback instance fails validation. We compare our technique with a variant

where there is only a single trial (i.e., $k = 1$). Second, we examine the performance when GPT-4 is used as the simulated "student" model instead of GPT-3.5. Third, we investigate the case wherein the generated single-sentence hint, instead of the detailed explanation, is utilized in the validation process. Fourth and last, we vary the threshold rule used for validation. In this regard, there are three variations: $\left(\frac{n_2}{n} \geq \alpha\right)$, where n_1 is not considered in the rule; $\left(\left(\frac{n_2}{n} \geq \frac{n_1}{n}\right) \wedge \left(\frac{n_2}{n} \geq \alpha\right)\right)$ where β is not considered in the rule; $\left(\frac{n_2}{n} \geq \frac{n_1}{n}\right)$ where α and β are not considered in the rule.

4.3 Evaluation Procedure

As discussed in Section 2, we employ human experts (evaluators) to assess the quality of generated feedback. More concretely, two human evaluators independently rated the feedback generated by techniques along the quality attributes as introduced in Section 2.[7] Then, given the ratings from each evaluator, we compute precision and coverage (based on the overall feedback quality *HOverall*).[8] Finally, for each technique and dataset, we aggregate across evaluators and report averaged results as *mean (stderr)*. We obtained Cohen's kappa reliability value 0.65 indicating *substantial agreement* between evaluators [7]. Next, we elaborate on our experimental results.

4.4 Results

Comparison with baselines and human tutors. Figure 6 provides an overview of results, comparing our technique and baselines. It is evident that GPT4HINTS-BASE exhibits a substantial performance gap when compared to TUTORHINTS. This gap is partially mitigated with the incorporation of failing test cases and fixed programs in the prompt, as seen with GPT4HINTS-IO and GPT4HINTS-IOFIX, respectively.[9] Our final technique, GPT4HINTS-GPT3.5VAL, consistently achieves precision levels comparable to TUTORHINTS, around 95% across all datasets.[10] Importantly, the trade-off in coverage required to attain such high precision is effective, and our technique maintains a coverage rate exceeding 70% for all three datasets. In Figure 8, we provide fine-grained results across different attributes,

---

[6]The data guide is meant to exercise students on extracting relevant information. Typically, students would search the PDF using keywords such as 'chickenpox' to spot relevant columns needed. For GPT-4, we extract and provide in the prompt a short summary describing the relevant columns; the summary is provided as part of our implementation (see Footnote 2).

[7]Similar to [27], these two human evaluators are same as two human tutors employed in the TUTORHINTS technique. When evaluating TUTORHINTS technique, an evaluator does not assess their own feedback produced while acting as a tutor.

[8]In addition, we also asked the evaluators to rate on *ECorrect*, a binary attribute capturing the correctness of the detailed explanation X. Further analysis regarding this additional attribute will be discussed in Section 4.4 and Figure 8.

[9]If, for a buggy program, no correct fixed program is obtained (see Section 3.1), the prompt of GPT4HINTS-IOFIX is the same as GPT4HINTS-IO's. The rates at which we obtained at least one correct fix for BASICALGO, DATAREGEX, and DATAANALYSIS datasets are 92%, 100%, and 93%, respectively.

[10]When comparing GPT4HINTS-GPT3.5VAL with other techniques in Figure 6, the results are significantly different w.r.t. χ^2 tests [6] ($p \leq 0.0001$); here, we use contingency tables with two rows (techniques) and four columns (data points are mapped to four possible precision/coverage outcomes).

Technique	BasicAlgo		DataRegex		DataAnalysis	
	Precision	Coverage	Precision	Coverage	Precision	Coverage
GPT4Hints-Base	66.0 (2.0)	100.0	85.4 (2.1)	100.0	78.3 (5.0)	100.0
GPT4Hints-IO	72.0 (4.0)	100.0	85.4 (2.1)	100.0	85.0 (5.0)	100.0
GPT4Hints-IOFix	82.0 (2.0)	100.0	91.7 (4.2)	100.0	93.3 (3.3)	100.0
TutorHints	92.0 (4.0)	100.0	91.7 (4.2)	100.0	91.7 (8.3)	100.0
GPT4Hints-GPT3.5Val	94.7 (0.0)	76.0 (0.0)	97.6 (2.4)	87.5 (0.0)	95.5 (4.5)	73.3 (0.0)

Figure 6: Results for different techniques on three real-world Python programming datasets. For each technique and dataset, results are averaged across two evaluators and reported as mean (stderr) as per the evaluation procedure in Section 4.3. Our technique, GPT4Hints-GPT3.5Val, performs validation of the generated feedback to achieve a higher quality of the feedback in terms of precision level, thereby trading off precision and coverage. Our technique can achieve a precision of around 95% reaching the quality of human tutors while maintaining a high coverage of over 70% across three real-world datasets; see Section 4.4 for a detailed discussion of results.

Variants of Validation Stage in GPT4Hints-GPT3.5Val	BasicAlgo		DataRegex		DataAnalysis	
	Precision	Coverage	Precision	Coverage	Precision	Coverage
Default with trials $k = 3$, GPT-3.5 student model, use $\mathcal{X}$, and threshold rule $\left(\left(\frac{n_2}{n} \geq \frac{n_1}{n}\right) \wedge \left(\left(\frac{n_2}{n} \geq 0.50\right) \vee \left(\frac{n_2}{n} \geq \frac{n_1}{n} + 0.25\right)\right)\right)$	94.7 (0.0)	76.0	97.6 (2.4)	87.5	95.5 (4.5)	73.3
Single trial $k = 1$ instead of $k = 3$	91.7 (0.0)	48.0	96.4 (3.6)	58.3	94.4 (5.6)	60.0
GPT-4 student model instead of GPT-3.5 student model	84.8 (2.2)	92.0	93.5 (2.2)	95.8	93.1 (3.4)	96.7
Using single-sentence hint $\mathcal{H}$ instead of detailed explanation $\mathcal{X}$	89.3 (3.6)	56.0	93.5 (2.2)	95.8	95.0 (5.0)	66.7
Threshold rule without considering n_1, i.e., $\left(\frac{n_2}{n} \geq 0.50\right)$	86.8 (2.6)	76.0	91.7 (4.1)	100.0	95.5 (4.5)	73.3
Simplified threshold rule without β, i.e., $\left(\left(\frac{n_2}{n} \geq \frac{n_1}{n}\right) \wedge \left(\frac{n_2}{n} \geq 0.50\right)\right)$	94.1 (0.0)	68.0	97.6 (2.4)	87.5	95.5 (4.5)	73.3
Simplified threshold rule without α, β, i.e., $\left(\frac{n_2}{n} \geq \frac{n_1}{n}\right)$	95.2 (0.0)	84.0	97.6 (2.4)	87.5	92.3 (3.8)	86.7

Figure 7: Comparison of performance between GPT4Hints-GPT3.5Val and different variants w.r.t the validation stage. The first four variations (single trial, GPT-4 student model, using $\mathcal{H}$, and threshold without considering n_1) show how different design choices in our validation stage helps improve precision-coverage trade off. The last two variations with simplified threshold rules shows the robustness of the default threshold rule in terms of α and β. See Sections 3.3 and 4.4 for further details.
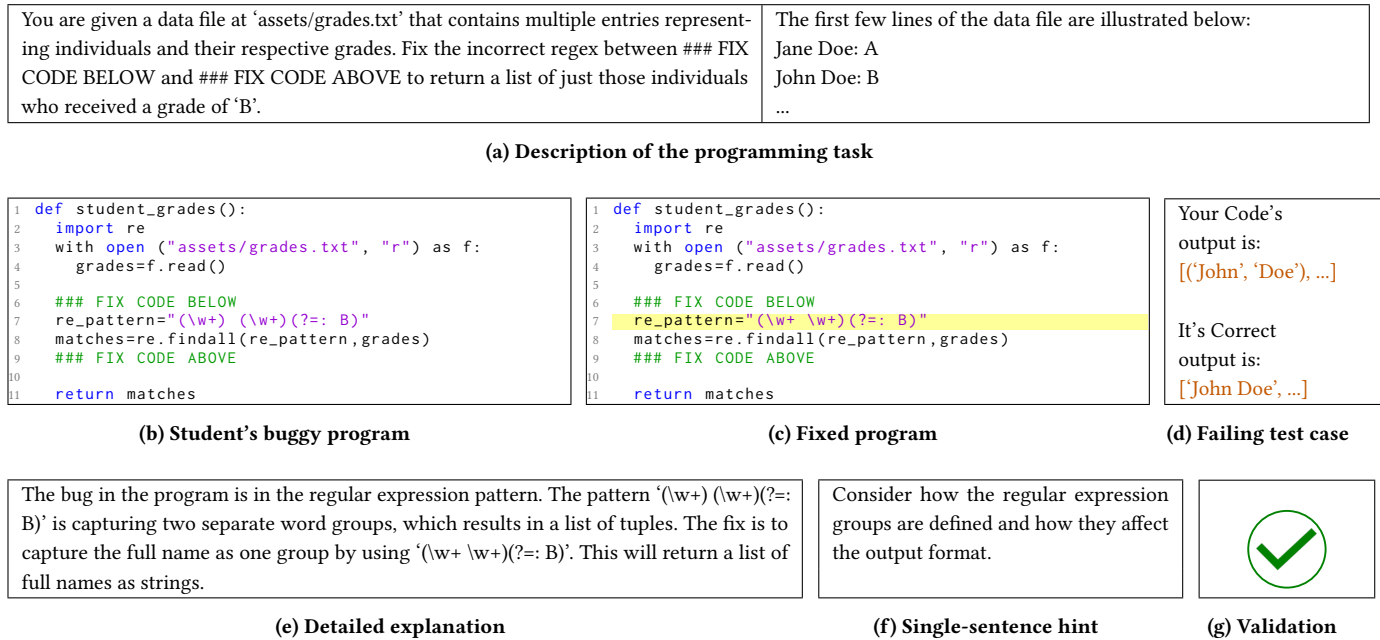
Method	Hint					Explanation	(Hint, Explanation)
	HOverall	HCorrect	HInformative	HConceal	HComprehensible	ECorrect	HOverall, ECorrect
GPT4Hints-Base	66.0	68.0	66.0	68.0	100.0	58.0	56.0
GPT4Hints-IO	72.0	78.0	74.0	76.0	98.0	66.0	62.0
GPT4Hints-IOFix	82.0	84.0	82.0	84.0	100.0	82.0	80.0
GPT4Hints-GPT3.5Val	94.7	94.7	94.7	94.7	100.0	91.1	92.1

Figure 8: Fine-grained results w.r.t. evaluation rubric that assesses the quality of generated feedback across different attributes as discussed in Sections 2 and 4.3. For our technique, these fine-grained results demonstrate a high correlation between generating a high-quality hint and a correct detailed explanation (used in the validation stage).

demonstrating a high correlation between generating a high-quality hint and a correct detailed explanation – this further justifies why the explanation can be used to validate the hint.

Comparison with variations of validation stage. Figure 7 shows the performance of different variants in comparison to our technique. Notably, with a single trial (i.e., $k = 1$), there is a substantial decrease in coverage across all datasets. This result underscores the marked effect of incorporating multiple trials in maintaining a high coverage level. Intriguingly, when we substitute GPT-3.5 with the more advanced model, GPT-4, as the simulated "student" model,

there is actually a reduction in precision. We observed that GPT-4 is worse than GPT-3.5 in terms of achieved precision as it tends to correctly fix the buggy program even if the explanation in the validation prompt is wrong. These results highlight that a weaker model (here, GPT-3.5 instead of GPT-4) could be better suited as a simulated "student" model. When using hints instead of explanations for validation, it yields inferior performance in general as the explanation contains more details about the bugs and fixes (thus having a better differential effect between using the standard and the augmented prompt). Regarding variants of the validation rule,

You are given a data file at 'assets/grades.txt' that contains multiple entries representing individuals and their respective grades. Fix the incorrect regex between ### FIX CODE BELOW and ### FIX CODE ABOVE to return a list of just those individuals who received a grade of 'B'.	The first few lines of the data file are illustrated below: Jane Doe: A John Doe: B ...

(a) Description of the programming task

```
1  def student_grades():
2    import re
3    with open ("assets/grades.txt", "r") as f:
4      grades=f.read()
5
6    ### FIX CODE BELOW
7    re_pattern="(\w+) (\w+)(?=: B)"
8    matches=re.findall(re_pattern,grades)
9    ### FIX CODE ABOVE
10
11   return matches
```

```
1  def student_grades():
2    import re
3    with open ("assets/grades.txt", "r") as f:
4      grades=f.read()
5
6    ### FIX CODE BELOW
7    re_pattern="(\w+ \w+)(?=: B)"
8    matches=re.findall(re_pattern,grades)
9    ### FIX CODE ABOVE
10
11   return matches
```

Your Code's output is:
['John', 'Doe'), ...]

It's Correct output is:
['John Doe', ...]

(b) Student's buggy program   **(c) Fixed program**   **(d) Failing test case**

The bug in the program is in the regular expression pattern. The pattern '(\w+) (\w+)(?=: B)' is capturing two separate word groups, which results in a list of tuples. The fix is to capture the full name as one group by using '(\w+ \w+)(?=: B)'. This will return a list of full names as strings.	Consider how the regular expression groups are defined and how they affect the output format.	✅

(e) Detailed explanation   **(f) Single-sentence hint**   **(g) Validation**

Figure 9: Similar to Figure 1, this example showcases GPT4Hints-GPT3.5Val on a buggy program from the DataRegex dataset.

the overall performance remains relatively stable when α and β are excluded from the rule, suggesting a robust performance irrespective of specific settings for these hyperparameters. However, a noticeable decline in performance is observed when the relative condition ($\frac{n_2}{n} \geq \frac{n_1}{n}$) is omitted, highlighting its importance in the validation process.

Qualitative analysis. We have included a few illustrative examples to showcase the effectiveness of our technique. Figures 1, 2, and 9 exemplify cases where GPT4Hints-GPT3.5Val generated high-quality feedback during Stage-2 and then successfully accepted during Stage-3. Conversely, for the scenario in Figure 10, GPT4Hints-GPT3.5Val's Stage-2 failed to produce high-quality feedback in all three trials, but Stage-3 successfully rejected all of those low-quality feedback instances. To be more specific, the values of n_1 and n_2 for the three trials in this case were $\{n_1 = 8, n_2 = 0\}$, $\{n_1 = 6, n_2 = 0\}$, and $\{n_1 = 5, n_2 = 0\}$, respectively. In contrast, in the example shown in Figure 1, GPT4Hints-GPT3.5Val's Stage-2 generated high-quality feedback during the first trial and Stage-3 subsequently accepted it with values $\{n_1 = 2, n_2 = 6\}$. We have provided additional illustrative examples as part of our implementation (see Footnote 2).

5 CONCLUDING DISCUSSIONS

We investigated the role of generative AI and large language models in providing human tutor-style programming hints to help students resolve errors in their buggy programs. In particular, we focused on improving the quality of generated feedback, which is crucial for deployment in real-life classroom settings. We developed a novel technique, GPT4Hints-GPT3.5Val, that leverages GPT-4 as a "tutor" model to generate hints and GPT-3.5 as a "student" model to validate the hint quality. This validation step provides a layer

of quality assurance by trading off coverage (how many students are given automatic feedback) and precision (quality of the given feedback). We performed an extensive evaluation to showcase the efficacy of our technique on three real-world Python programming datasets, reaching the precision-level of human tutors.

Our work has two important implications for the research community interested in leveraging generative AI and large language models for computing and programming education. First, our results show how we can effectively utilize these models as "tutor" by prompting them with symbolic data such as failing test cases. This symbolic data essentially provides in-context information to enhance the reasoning and execution abilities of these models where they typically struggle. Second, our results show how we can utilize these models in a flipped role as "student" to simulate the effect of feedback on a real human student. Interestingly, we also showed that a weaker model (GPT-3.5, instead of GPT-4) serves as a better "student" model for validating the effect of feedback generated by GPT-4. This flipped role opens up new opportunities in utilizing generative models as in-context student models for automatic assessments, learning analytics, and simulations.

Next, we discuss some limitations of our current work and ideas to tackle them in the future. First, our work involved OpenAI's GPT family of models; it would be useful to evaluate alternate generative models, in particular, open-source variants like Llama-2. Moreover, we utilized the GPT-3.5 model at a higher temperature to simulate the potential utility of providing feedback; it would be interesting to investigate how to employ different LLMs to better simulate diverse student behaviors. Second, our work didn't leverage historical data on a given problem when generating hints, e.g., hints provided by human tutors for previous students' buggy attempts on a problem. It would be important to develop techniques that can

Given two strings S1 and S2 as input, the task is to merge them alternatively, i.e., the first character of S1 then the first character of S2 and so on till the strings end.	Expected Time Complexity: $O(|S1| + |S2|)$. Constraints: $1 \leq |S1|, |S2| \leq 10^3$.

<div align="center">(a) Description of the programming task</div>

```python
#User function Template for python3
class Solution:
  def merge(self, S1, S2):
    l1=len(S1)
    l2=len(S2)
    if l1>l2:
      res=""
      dif=l1-l2
      x=S1[:l2]
      y=S2
      d=[(i,j) for i,j in zip(x,y)]
      for i in d:
        for j in i:
          res+=j
      return res+S1[-dif:]
    elif l1<l2:
      res=""
      dif=l2-l1
      x=S2[:l1]
      y=S1
      d=[(i,j)for i,j in zip(x,y)]
      for i in d:
        for j in i:
          res+=j
      return res+S2[-dif:]
    else:
      res=""
      x=S1[:l2]
      y=S2
      d=[(i,j)for i,j in zip(x,y)]
      for i in d:
        for j in i:
          res+=j
      return res

#{ Driver Code Starts
#Initial Template for Python 3
if __name__ == '__main__':
    t = int(input())
    for _ in range(t):
        S1,S2 = map(str,input().strip().split())
        ob = Solution()
        print(ob.merge(S1, S2))
# } Driver Code Ends
```

<div align="center">(b) Student's buggy program</div>

```python
#User function Template for python3
class Solution:
  def merge(self, S1, S2):
    l1=len(S1)
    l2=len(S2)
    if l1>l2:
      res=""
      dif=l1-l2
      x=S1[:l2]
      y=S2
      d=[(i,j) for i,j in zip(x,y)]
      for i in d:
        for j in i:
          res+=j
      return res+S1[l2:]
    elif l1<l2:
      res=""
      dif=l2-l1
      x=S1
      y=S2[l1:]
      d=[(i,j)for i,j in zip(x,y)]
      for i in d:
        for j in i:
          res+=j
      return res+S2[l1:]
    else:
      res=""
      x=S1
      y=S2
      d=[(i,j)for i,j in zip(x,y)]
      for i in d:
        for j in i:
          res+=j
      return res

#{ Driver Code Starts
#Initial Template for Python 3
if __name__ == '__main__':
    t = int(input())
    for _ in range(t):
        S1,S2 = map(str,input().strip().split())
        ob = Solution()
        print(ob.merge(S1, S2))
# } Driver Code Ends
```

<div align="center">(c) Fixed program</div>

For Input: Qh eyNFX

Your Code's output is:

eQyhNFX

It's Correct output is:

QehyNFX

<div align="center">(d) Failing test case</div>

The bug in the program is in the slicing of the strings when one string is longer than the other. In the original program, when S1 is longer than S2, the remaining characters of S1 are not correctly added to the result. The same issue occurs when S2 is longer than S1. The fix is to slice the longer string from the length of the shorter string to the end, instead of slicing from the end.

<div align="center">(e) Detailed explanation</div>

Consider how you are slicing the longer string when one string is longer than the other.

<div align="center">(f) Single-sentence hint</div>

<div align="center">(g) Validation</div>

Figure 10: Similar to Figure 1, this example showcases GPT4Hints-GPT3.5Val on a buggy program for the MergeStrs problem from the BasicAlgo dataset. For this example, the generated detailed explanation and single-sentence hint feedback are not correct (e.g., the explanation suggests fixing the program based on a different slicing strategy, which is not related to the bug in this program). The validation stage of the technique (that evaluates the potential utility of this detailed explanation, cf. Figure 3) *successfully rejected* the generated hint as low-quality and not suitable for sharing with the student. See Section 4.4 for further discussion of results.

leverage this data, e.g., by fine-tuning these open-source variants to generate better-quality hints. Third, our evaluation considered small datasets comprising a total of 79 buggy programs; it would be useful to scale up the studies by considering larger-scale datasets.

Fourth, we focused only on Python programming education; it would be interesting to conduct a similar study for other programming languages and other domains beyond programming. Fifth, our evaluation only considered expert-based annotations and didn't

involve students; it would be important to conduct studies with students to evaluate techniques from their perspectives.

ACKNOWLEDGMENTS

REFERENCES

[1] Khan Academy. 2023. Khanmigo. https://www.khanacademy.org/khan-labs.
[2] Yejin Bang et al. 2023. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. *CoRR* abs/2302.04023 (2023).
[3] Georg Brandl, Matthäus Chajdas, and Jean Abou-Samra. 2006. Pygments. https://pygments.org/.
[4] Sébastien Bubeck et al. 2023. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. *CoRR* abs/2303.12712 (2023).
[5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *CoRR* abs/2304.05128 (2023).
[6] William G Cochran. 1952. The χ^2 Test of Goodness of Fit. *The Annals of Mathematical Statistics* (1952).
[7] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
[8] geeksforgeeks.org. 2009. GeeksforGeeks: A Computer Science Portal for Geeks. https://www.geeksforgeeks.org/.
[9] GitHub. 2022. GitHub Copilot: Your AI Pair Programmer. https://github.com/features/copilot.
[10] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *PLDI*.
[11] Andrew Head, Elena L. Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Learning @ Scale*.
[12] Natalie Kiesler, Dominic Lohr, and Hieke Keuning. 2023. Exploring the Potential of Large Language Models to Generate Formative Programming Feedback. In *FIE*.
[13] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent N. Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using Large Language Models to Enhance Programming Error Messages. In *SIGCSE*.
[14] Tiffany Wenting Li, Silas Hsu, Max Fowler, Zhilin Zhang, Craig B. Zilles, and Karrie Karahalios. 2023. Am I Wrong, or Is the Autograder Wrong? Effects of AI Grading Mistakes on Learning. In *ICER*.
[15] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *SIGCSE*.
[16] Aman Madaan et al. 2023. Self-Refine: Iterative Refinement with Self-Feedback. *CoRR* abs/2303.17651 (2023).
[17] Wes McKinney et al. 2011. pandas: A Foundational Python Library for Data Analysis and Statistics. *Python for High Performance and Scientific Computing* 14, 9 (2011), 1–9.
[18] Samim Mirhosseini, Austin Z. Henley, and Chris Parnin. 2023. What is Your Biggest Pain Point? An Investigation of CS Instructor Obstacles, Workarounds, and Desires. In *SIGCSE*.
[19] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *CoRR* abs/2210.14306 (2022).
[20] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation. *CoRR* abs/2306.09896 (2023).
[21] OpenAI. 2022. Codex-Edit. https://beta.openai.com/playground?mode=edit&model=code-davinci-edit-001.
[22] OpenAI. 2023. ChatGPT. https://openai.com/blog/chatgpt.
[23] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).
[24] Victor-Alexandru Pădurean, Georgios Tzannetos, and Adish Singla. 2023. Neural Task Synthesis for Visual Programming. *CoRR* abs/2305.18342 (2023).
[25] Maciej Pankiewicz and Ryan Shaun Baker. 2023. Large Language Models (GPT) for Automating Feedback on Programming Assignments. *CoRR* abs/2307.00150 (2023).
[26] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. In *EDM*.
[27] Tung Phung, Victor-Alexandru Pădurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors. In *ICER V.2*.
[28] Quizlet. 2023. Q-Chat. https://quizlet.com/qchat-personal-ai-tutor.
[29] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *ICER*.
[30] Jaromír Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. 2023. Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses?. In *ITiCSE*.
[31] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI*.
[32] Adish Singla. 2023. Evaluating ChatGPT and GPT-4 for Visual Programming. In *ICER V.2*.
[33] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).
[34] Jason Wei et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.
[35] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing Bugs in Python Assignments Using Large Language Models. *CoRR* abs/2209.14876 (2022).