Using Program Repair as a Proxy for Language Models' Feedback Ability in Programming Education

Charles Koutcheme and **Nicola Dainese** and **Arto Hellas**
Aalto University, Espoo, Finland
`first.last@aalto.fi`

Abstract

One of the key challenges in programming education is being able to provide high-quality feedback to learners. Such feedback often includes explanations of the issues in students' programs coupled with suggestions on how to fix these issues. Large language models (LLMs) have recently emerged as valuable tools that can help in this effort. In this article, we explore the relationship between the program repair ability of LLMs and their proficiency in providing natural language explanations of coding mistakes. We outline a benchmarking study that evaluates leading LLMs (including open-source ones) on program repair and explanation tasks. Our experiments study the capabilities of LLMs both on a course level and on a programming concept level, allowing us to assess whether the programming concepts practised in exercises with faulty student programs relate to the performance of the models. Our results highlight that LLMs proficient in repairing student programs tend to provide more complete and accurate natural language explanations of code issues. Overall, these results enhance our understanding of the role and capabilities of LLMs in programming education. Using program repair as a proxy for explanation evaluation opens the door for cost-effective assessment methods.

1 Introduction

Large Language Models (LLMs) and applications leveraging them such as ChatGPT have been embraced by both the general public and academia. The adoption is also visible in the domain of computing and programming education, where researchers have highlighted a variety of learning tasks that LLMs can tackle (Denny et al., 2023; Prather et al., 2023), including their performance in providing help and feedback to students (Hellas et al., 2023).

Feedback is a crucial part of learning (Hattie and Timperley, 2007). While various forms of feedback exist in programming (Keuning et al., 2018), explaining code issues in natural language can be particularly useful. Providing students with natural language explanations of the mistakes in their code allows them to gain a better understanding of gaps in their knowledge.

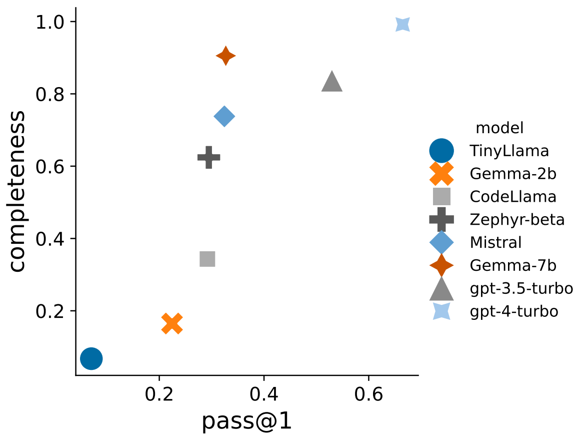

Figure 1: Summary benchmarking results. The quality of LLMs' Natural Language descriptions of issues in students' code (completeness) tends to increase with LLMs' ability to fix the student programs (pass@1).

With the increasing number of LLMs proficient at providing feedback (Koutcheme et al., 2023a) to some degree, selecting the best one before deploying it in classrooms (Liu et al., 2024) can be challenging. Human evaluation can take time, as it requires either manual assessment or annotated datasets. While research in the automated evaluation of LLM generation is on the rise (Zheng et al., 2023), also in educational areas (Fernandez et al., 2024), the developed methods often rely on other language models (e.g., utilizing powerful yet expensive LLMs such as GPT-4), which can induce computational or financial costs. A more cost-effective approach is needed.

Before the advent of LLMs, a stream of work in programming education has focused on educational program repair (Gulwani et al., 2018; Parihar et al., 2017; Yi et al., 2017), where the goal is to produce fixes for students' incorrect programs. Although repairs to student programs are not always directly provided to students, they serve as a fundamental step in generating different types of support, including next-step hints for Intelligent Tutoring Systems (McBroom et al., 2021). While direct evaluation of feedback with natural language explanations can be challenging, evaluating whether LLMs can fix programs is much more straightforward.

With this in mind, we hypothesize that the student program repair capability of an LLM may relate to its capability to provide natural language explanations of code issues. If this would hold, program repair capability – which is easier to assess – could serve as a proxy for evaluating feedback quality. Our intuition is supported by prior work that has found relationships between LLMs' abilities in related domains. For instance, LLMs that are proficient in solving specific problems are effective judges of the quality of explanations in those domains (Zheng et al., 2023). Similarly, there is some evidence that instruction-tuned LLMs trained on specific tasks can generalize to unseen parallel or close tasks (Wei et al., 2022a).

In this article, we investigate whether there effectively exists a relationship between the ability of LLMs to repair students' programs and their ability to explain code issues in natural language. If our hypothesis holds, researchers could more easily benchmark LLMs for other educational purposes, allowing educators to streamline the selection of LLMs. Our evaluation focuses on several leading and popular open-source language models, as well as proprietary models.

The main contributions of this article are (1) the benchmarking of several leading language models' abilities for program repair and (2) natural language explanation of code issues, as well as (3) the analysis and identification of the relationship between the two tasks.

2 Related Work

2.1 Program Repair and Feedback

Propagating feedback. Generating natural language explanations of the issues in student programs has been a long-standing challenge, with much work leveraging part of human annotations to bootstrap efforts (Piech et al., 2015; Malik et al., 2021; Koivisto and Hellas, 2022). In that area, early pretrained code language models have also shown useful (Wu et al., 2021) in making human annotations as data efficient as possible. However, coming up with such annotations remains a time-consuming endeavour.

Educational Program Repair. Trying to alleviate the need for manual annotation, feedback on programming assignments has often been generated with the aid of automated program repair tools (Hu et al., 2019a), attempting to repair syntax and/or semantic errors in students' programs. In this area, LLMs have also shown great promise. Much of this line of work has mainly used early versions of the OpenAI Codex model, thus obtaining both syntax fixes (Zhang et al., 2022; Ahmed et al., 2022; Leinonen et al., 2023) and semantic fixes for students' non-working solutions (Zhang et al., 2022). Such fixes can inform Intelligent Tutoring Systems, which could then provide next-step hints to students (Rivers and Koedinger, 2017). However, while automatically constructed next-step hints can tell the students *what* to do next (in templated natural language sentences), they are not always able to explain the reasons *why* the code does not work.

Natural Language Explanations. The rise of newer and more powerful LLMs (e.g., ChatGPT) has opened the possibility of directly generating high-quality code explanations (Sarsa et al., 2022). In addition to such progress, research in improving program repair remains useful. In particular, recent efforts suggest that generated repairs can be included in the prompt to allow language models to provide more accurate natural language explanations of a program's issues (Phung et al., 2023a). In parallel, prior work has also explored using program repair to validate the quality of LLM-generated feedback. In this space, the quality of LLM-generated repairs (i.e., whether the repairs pass all unit tests) would indicate whether the associated LLM-generated feedback would be given to students. The repairs could be generated by the LLM providing the feedback (Shubham Sahai, 2023), or by another, less powerful LLM acting as an artificial student (Phung et al., 2023b).

In contrast to efforts using program repair as a means for validating single generations, our work aims to assess whether the overall ability of a single language model to provide repair across a *larger set of programs* is indicative of the language model's overall ability to generate natural language explanations.

2.2 Evaluating Language Models

Benchmarking code language models. When new language models are released, their performance is often assessed through multiple code generation benchmarks such as HumanEval (Chen et al., 2021), APPS (Hendrycks et al., 2021), MBPP (Austin et al., 2021), or DS-1000 (Lai et al., 2022). In parallel, prior work has also evaluated LLMs' ability to fix buggy programs in benchmarks such as HumanEval+ (Muennighoff et al., 2023), CodeXGlue (Lu et al., 2021), or QuixBugs (Lin et al., 2017). However, while such benchmarks contain multiple tasks that could potentially inform us of LLMs' performance in educational contexts, it is important to note that students' submitted incorrect programs can contain issues/defects that go beyond mere simple bugs (e.g. implementation of the wrong algorithm). Hence, educational benchmarks are needed.

Benchmarking in education. In the educational context, much work has looked into the performance of proprietary models (Codex, and ChatGPT) on private datasets and educational datasets (Finnie-Ansley et al., 2022; Hellas et al., 2023) both for program synthesis (Finnie-Ansley et al., 2022; Savelka et al., 2023b) and feedback (Hellas et al., 2023).

Open-source language models. While there exist few efforts looking at the performance of open-language models for generating repairs (Koutcheme et al., 2023a; Koutcheme, 2023), or answering student programming questions (Hicke et al., 2023), only the work of (Koutcheme et al., 2024) look into the performance of open-source models for generating educational programming feedback. Still, none of these works studies the relationship between program repair abilities and the quality of LLM-generated natural language explanations.

3 Methodology

We (1) evaluate how LLMs perform in generating repairs to incorrect programs, (2) evaluate how LLMs perform in explaining the issues in programs, and (3) study the potential relationship between the ability to generate repairs and the ability to generate natural language explanations. To ensure a comprehensive assessment, our study encompasses zero-shot evaluations (Yogatama et al., 2019; Linzen, 2020) of proprietary and state-of-the-art open-source LLMs having less than 7 billion (7B) parameters. Our experiments leverage a publicly available dataset comprising real-life students' submissions to Python programming problems.

Next, we describe the programming dataset, outline our evaluation methodology, and list the language models included in this evaluation. We release the code used to perform our experiments as an additional contribution [1].

3.1 Dataset

We use a subset of the FalconCode (de Freitas et al., 2023) dataset, a large-scale dataset containing thousands of first-year students' solutions (over three semesters) to hundreds of Python programming assignments. It is the largest and most comprehensive publicly available dataset of student programs at the time of writing this manuscript. Beyond its substantial scale, this dataset distinguishes free-form assignments (i.e., not scoped to function writing), and exercise-level programming with concept annotations, enabling a broader evaluation of LLM feedback.

Dataset processing. Due to the financial and computational costs of running LLM evaluations, for our experiments, we curate a smaller subset of submissions. The dataset contains three semesters worth of submissions (fall 2021, spring 2021, and fall 2022). We start by selecting submissions from the last semester (fall 2022). Each exercise in the dataset can be categorized based on a type (practice, or exam) and a level of difficulty ("skill", "lab", or "project", i.e., easy, medium, hard). We omit exam exercises and focus on practice exercises (as these are the ones students require help with). Additionally, we exclude more complex "project" assignments, requiring extensive code writing across multiple files, and those requiring external files. Following Hu et al. (2019b), we select only the final incorrect submissions for each student for each assignment. Although this selection may not capture the full range of student difficulties, it aligns with the idea that a student's last attempt often reflects their final understanding. Finally, we remove submissions with identical abstract syntax tree structures after variable normalization (Koutcheme et al., 2023a,c). The final dataset contains 370 programs from 44 assignments.

[1] `https://github.com/KoutchemeCharles/bea2024`

3.2 Repairing Student Programs

Given a student's incorrect program in our test set, the first task is for an LLM to produce a repair to that incorrect program that passes all unit tests. Because of the wide range of issues found in students' programs, in contrast to classical program repair benchmarks (Lin et al., 2017; Muennighoff et al., 2023), in most educational scenarios, we do not assume the existence of a single unique ground truth repair to an incorrect program. However, while such unique ground truth does not exist, repairs that align with the original incorrect programs are often preferred. The general assumption is that closely aligned programs can generate (Phung et al., 2023a) or are associated with feedback (Koutcheme et al., 2023a) (e.g. natural language explanations or hints) that are more understandable to students, as this feedback would require a lower cognitive load to understand the issues in the program and the modifications that need to be operated to reach a solution (Shubham Sahai, 2023). Moreover, we aim to investigate whether the language model's ability to produce repairs that closely resemble the original incorrect programs correlates with its proficiency in generating complete and accurate natural language explanations of the issues in the programs. The constraints on functional correctness and closeness are reflected in our evaluation procedure, which we adapt from the work of Koutcheme et al. (2023a).

Evaluation procedure. To evaluate functional correctness, for each incorrect program in our test set, we generate a single repair using greedy decoding (Rozière et al., 2023). To measure the ability of the language model to generate close repairs, we compute the ROUGE-L (Lin, 2004) score between the incorrect program and the candidate repair extracted from the single greedy generation. While other distance measures exist and have been used to measure closeness between programs (e.g., BLEU (Papineni et al., 2002) and CodeBERT score (Zhou et al., 2023b)), the ROUGE-L score has been shown to correlate well with human judgement of high-quality repairs (Koutcheme et al., 2023b) while remaining fast to compute, as it does not rely on a language model.

We report the average repair success rate as the pass rate ('pass@1' (Chen et al., 2021)) and the average ROUGE-L score, abbreviated as 'rouge', over the programs in our test set.

3.3 Explaining Issues in Students Programs

The second task is for our language models to explain all the issues in a given student's incorrect program. For each incorrect program, we prompt our language model to explain the issues using the prompt shown in Figure 5 (Appendix A.1), a variant of the prompt used in (Hellas et al., 2023). Following prior work, we generate a single output using greedy decoding (Hellas et al., 2023; Savelka et al., 2023a; Leinonen et al., 2023).

Evaluation criteria. For each natural language explanation, we focus on two particular quantitative aspects of quality: (1) ensuring that the feedback is complete, i.e., it identifies and mentions all issues in the code, and (2) ensuring that it avoids hallucinations, i.e., it does not mention non-existent issues (Phung et al., 2023b; Hicke et al., 2023; Hellas et al., 2023). We highlight that our explanation task is a specific form of feedback that differs from hints. In the explanation task, the answer is meant to be given to students, while for hints (Roest et al., 2024), the feedback *helps* the students find the answer themselves. While prior work in hint generation has investigated other qualitative aspects, such as the "right level of detail"((Phung et al., 2023a; Scarlatos et al., 2024)), we believe these are less likely to be correlated with an LLM repair ability.

Automated Evaluation. Given the scale of our dataset and the multitude of language models to assess, conducting human evaluation would be impractical. Therefore, we rely on automated evaluation using language models (Zheng et al., 2023). Powerful language models like ChatGPT have exhibited near-human performance across various tasks, sparking interest in their application for evaluating other LLMs (Zhou et al., 2023a; Cui et al., 2023; Tunstall et al., 2023), including in educational contexts (McNichols et al., 2024; Hicke et al., 2023). Notably, GPT-4 has demonstrated good performance in evaluating programming feedback quality (Koutcheme et al., 2024). In our work, we ask GPT-4 to grade the quality of the natural language explanations for each incorrect program. We ask the model to provide a binary label of whether each criterion (completeness, and avoiding highlighting non-existent issues) holds for the feedback generated by each language model. Figure 6 (appendix A.1) shows our prompt. For each criterion, we report the average over the test set.

3.4 Models

We focus our evaluation on instruction-tuned and chat models. While pretrained language models can also be useful for multiple tasks, as prior studies using Codex (Phung et al., 2023a) have shown, instruction-tuned models alleviate the need for complex queries and allow easier interactions which benefit educators and researchers.

Closed-source models. We evaluate GPT-3.5 (`gpt-3.5-turbo`) and GPT-4-turbo (`gpt-4-1106-preview`) on our two tasks. Due to the financial costs of running GPT-4, we use the Turbo version for feedback generation, but we keep the standard GPT-4 for evaluating the quality of the natural language generations.

Open-source models. While prior work in programming feedback using LLMs has focused mainly on ChatGPT models (i.e., GPT-3.5 and GPT-4), we aim to cover the wider range of available options and include a selected number of instruction-tuned open-source/permissive models. We report the performance of the following family of models:

- TinyLLama (Zhang et al., 2024), a 1.1B parameter model following the Llama (Touvron et al., 2023) architecture.
- CodeLLAMA (Rozière et al., 2023), series of Llama (Touvron et al., 2023) models specialized for code. We report the performance of the 7B parameters model.
- Mistral 7B (Jiang et al., 2023), a 7B parameters language model released by the MistralAI team.
- Zephyr (Tunstall et al., 2023) are 7B parameters language models fine-tuned by HuggingFace using Direct Preference Optimization (Rafailov et al., 2023) on top of Mistral 7B model. We evaluate the performance of Zephyr 7B β.
- Gemma (Google, 2024), open source model released by Google DeepMind. We evaluate the performance of the 2B and 7B parameters models.

We chose these families of models because they are fully open-source and well-documented, they perform competitively on various code benchmarks (for models of their size), and they are widely adopted in the community. Additionally, within these families, we choose language models having 7 billion parameters or less, as such models can generally fit within one large GPU (without quantization). This choice is reflected by the potential need for educators to run models on custom hardware, who are unlikely to have the computational and financial resources to access more than a single GPU.

Table 1: Summary of the performance of the models in program repair and code issue explanation. For the metrics pass@1, rouge, and completeness, a higher score indicates better performance. Conversely, for the hallucination rate metric, a lower score is preferable. Legend: compl. (completeness), hall. rate (hallucination rate).

model	repair		explanation	
	pass@1	rouge	compl.	hall. rate (↓)
TinyLlama	0.070	0.062	0.068	0.335
Gemma-2b	0.224	0.175	0.165	0.400
CodeLlama	0.292	0.251	0.343	0.841
Zephyr-beta	0.295	0.236	0.624	0.716
Mistral	0.324	0.241	0.738	0.397
Gemma-7b	0.327	0.298	0.905	**0.005**
gpt-3.5-turbo	0.530	0.470	0.838	0.368
gpt-4-turbo	**0.665**	**0.536**	**0.992**	0.024

Technical details. We query ChatGPT models using OpenAI's Python API. We run the selected open-source language models using the HuggingFace Transformers library (Wolf et al., 2020), each model is run on a single NVIDIA A100 using our institution research cluster. We run all models using their recommended precision. The details of each model (the names) can be found in Table 3 (appendix A.2).

4 Results

First, we describe our general results, then, we outline an ablation analysis detailing the performance of the selected models over a set of programming concepts.

4.1 Main Results

Table 1 summarizes the performance of the LLMs in program repair and in explaining issues in code. We can make the following observations:

LLMs proficient in program repair generate repairs closer to the original incorrect program.

Figure 2 highlights the scaling relationship between the pass rate and the rouge score. We see that as language models become more and more proficient in generating repairs, these repairs become closer to students' original programs and thus more useful. One could expect that LLMs which produce more fixes could generate generic solutions (which are far away from the student code) (Koutcheme et al., 2023c) – however, this is not the case.

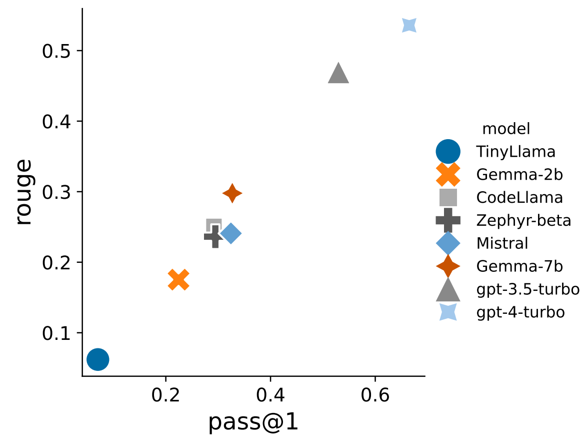

Figure 2: Relationship between pass rate and rouge score.

Hallucination conditionally decreases as a function of completeness. Figure 3 highlights the relationship between the ability of a model to identify all issues in a program (completeness), and the model's tendency to hallucinate (hallucination rate). If we omit language models with less than 2B parameters (i.e., TinyLlama and Gemma-2B), we observe that the hallucination rate decreases as completeness increases. This relationship seems to hold only for large enough language models. Our interpretation is supported by prior work that has shown that many emerging behaviours in language models appear when sufficiently large sizes are reached (Wei et al., 2022b) (e.g. their ability to solve new tasks via chain-of-thought prompting (Wei et al., 2023)).

The ability to explain moderately scales with the ability to repair. Figure 1 highlights the relationship between repair performance and explanation performance (in terms of completeness). Generally, a language model that is better at program repair tends to also produce more complete descriptions. In the set of our LLMs, only Gemma-7B and GPT-3.5 disrupt this relationship: although Gemma-7B has a lower pass rate than GPT-3.5 (only slightly

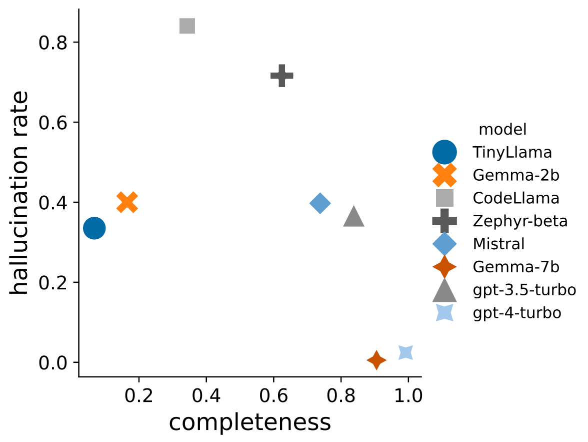

Figure 3: Relationship between completeness and hallucination rate.

better than Mistral), it produces very complete explanations (and with fewer hallucinations). Interestingly, the performance gap between models' ability to repair does not reflect the gap between their ability to explain in natural language. For instance, the difference between CodeLLama and Zephyr-7B in pass@1 (0.003) is almost 10 $\times$ smaller than the performance gap between the models' abilities to generate complete explanations (0.281).

Reparing student programs is harder than explaining issues in natural language. When looking at the maximum value that the pass@1 metric assumes (0.665), we see that it is smaller than the one of the completeness (0.992). We believe repairing programs is more challenging than providing explanations, as the latter requires understanding the issues while the former requires both comprehension and expertise on how to implement the fixes.

On base models and fine-tuning. We hypothesize that pass@1 and completeness are reflective of the capabilities of the underlying base model, while the hallucination rate seems to depend more on the fine-tuning procedure. Our intuition is justified by the following observations: (1) Mistral and Zephyr share the same base model (but only different fine-tuning) and have comparable pass@1 and completeness, but very different hallucination rates. OpenAI and Google invest significant efforts into curating datasets for fine-tuning to avoid hallucinations. On the other hand, small language models (TinyLLama and Gemma-2b) are probably too inaccurate (i.e., not powerful enough) to even hallucinate.

4.2 Concept Level Performance Analysis

The FalconCode dataset contains information about 20 programming concepts or "skills" (e.g., function definition, assignment, conditionals). The authors of the dataset manually annotated each exercise with information on whether each of these skills is practised (or needs to be mastered) in each exercise. We refer the reader to the original paper for details about the concepts (de Freitas et al., 2023).

In the same way that some students exhibit varying struggles with understanding and practising specific programming concepts (Liu et al., 2023), we suspect that language models might face a similar challenge. By examining the performance of language models on a per-concept basis, we aim to provide insights into their strengths and weaknesses in addressing specific programming challenges, thus informing educators and developers on their suitable application scenarios.

We thus conduct an ablation study looking at the per-concept performance of our language models for repair and natural language explanation generation.

Methodology. For each of the 20 concepts, we obtain the list of exercises practising the concept and subsequently retrieve the incorrect programs in our test set submitted to these exercises. For each concept, we then report and compare the performance of the language models for program repair and natural language generation (using the same evaluation metrics) based on the retrieved subset of incorrect programs.

It is important to note that because all exercises practice multiple concepts, knowing which *individual* concept is responsible for the language model failing to fix (or explain) the issues in a program is impossible. As such, the following results will give us an overview of the *likelihood* that an LLM would struggle to support students if an exercise *involves* such a concept. Table 4 (Appendix A.3) shows the number of exercises and programs that practice each specific concept. We limit our analysis to concepts practised in more than 3 exercises.

Results. Due to space limitations, we focus our analysis on the concepts with which language models struggle the most. Table 2 shows these concepts for all performance metrics, which are derived from Table 6 in Appendix B.2 showing the detailed scores of all models. We can make the following observations:

Table 2: Programming concepts performance summary. We show the programming concept for which each language model struggles the most. Legend: IS (input string), IC (input casting), C (conditionals), FC (function call), FD (function definition), L (list), LU (loop until), L2D (list 2D), hall. rate (hallucination rate).

	pass@1	rouge	completeness	hall. rate
TinyLlama	IC	IC	LU	IS
Gemma-2b	LU	IS	LU	L2D
CodeLlama	IC	IC	L2D	L
Zephyr-beta	IS	IS	FD	C
Mistral	IS	IS	FD	L
Gemma-7b	IS	IS	FC	LU
gpt-3.5-turbo	IC	IC	LU	FC
gpt-4-turbo	IS	IS	LU	L2D

When looking into the worst-practised concepts for repairing student programs, almost all of them are related to input manipulation (input string, or input casting), similar to what has been observed in LLMs capability to provide suggestions to programming help requests (Hellas et al., 2023). Moreover, LLMs that perform poorly at fixing a given concept are also likely to perform poorly at generating close solutions for these concepts.

When looking at the worst concepts for natural language explanations, these concern a wider range (looping, data structure, functions, basic operations). For completeness, there is not much variation in the performance in explaining issues for different concepts, but rather the overall performance is correlated with the pass@1 of the corresponding model. For hallucination rate, each model has its own "base performance", which doesn't correlate with pass@1 and it's roughly constant across concepts, with the exceptions of Zephyr and gpt-3.5-turbo, which respectively over- and underperform on function-related concepts concerning other concepts. There is no clear association between the concepts where LLMs are accurate and those where they hallucinate. Both small language models (less than 7B parameters) and proprietary models struggle most to be accurate with the 'looping until' concept, while language models of 7B parameters struggle more with function-related assignments.

It is important to note that "struggling" here is relative to the model's performance with other concepts. GPT-4 "struggling" more on completeness with looping is still accurate 90% of the time.

5 Discussion

Repair as a proxy for feedback. Our results suggest that language models' relative ability to fix students' programs (which is easy to evaluate) tells us how these language models will compare in finding all issues in students' code while avoiding hallucination (for big enough language models). Based on our discovery, one can devise more efficient LLM selection pipelines. For instance, a simple strategy consists of filtering out language models for which repair performance does not reach a certain threshold, a threshold set based on a few evaluations of LLM natural language generation performance. As an illustrative scenario, only evaluating the Mistral model on our dataset allows us to reasonably assume that language models performing worse than 0.32 in pass rate (pass@1) are unlikely to generate complete explanations for more than 73.8 % of programs. Using this pass rate value can thus act as a selection lower bound. As LLMs are becoming more widely adopted in education (Prather et al., 2023; Denny et al., 2024), and as the number of available models is increasing, these insights can help in the adoption process as institutions evaluating LLMs for their context can potentially reduce the number of LLMs to consider or limit the number of tasks conducted during the evaluation.

Open-source language models strike back. Another important finding emerging from our results is that while high-performance program repair must rely on proprietary models, recent 7B parameters models such as Gemma-7B can generate high-quality feedback competitive with SOTA models (Koutcheme et al., 2024). This has positive implications for educators interested primarily in giving students feedback rather than repairing solutions, as such feedback can also be generated via privacy-preserving open-source models.

However, it's important to acknowledge that running such models requires custom computational resources. In the literature, 7B parameter models are sometimes termed "small" due to their relative size compared to many large language models (e.g. Falcon-180B model (Almazrouei et al., 2023)). Yet, a 7B parameter is not small in terms of computational resources as it requires a large GPU to fit entirely into memory (without quantization). There is currently a trend in developing small language models (less than 3B parameters) such as TinyLlama and Gemma which can run on more modest hardware (e.g., consumer laptop GPU, or accelerated hardware). However, the performance of such LLMs, as our results suggest is still lagging behind their 7B parameters counterparts.

Identifying specific knowledge gaps. Unfortunately, our results do not yet allow us to identify which programming concepts LLMs will struggle to explain in natural language from their program repair performance. While individual repair performance depends on the concept being practised, a language model's performance in explaining issues does not (i.e., the performance is constant across all concepts). We hypothesize that the per-concept performance gap is only revealed for the harder task of fixing students' programs. Uncovering LLM knowledge gaps with automated measures might require us to rely on harder automatically evaluable tasks (e.g. QLCs (Lehtinen et al., 2024)).

Interplay of programming feedback types. Our primary research objective is to deepen our understanding of LLMs' feedback capabilities in educational contexts. Specifically, we seek to explore the relationship between different forms of feedback and program repair. While we treated feedback (identifying and explaining issues in programs) and program repair as distinct tasks in this study, we acknowledge their inherent interdependence. Previous research suggests that high-quality repairs can induce high-quality feedback when provided in context (Phung et al., 2023b,a). However, generating high-quality repairs is inherently challenging, as our results suggest, requiring the language model to comprehend what is wrong in a program and how to address the issues. In contrast, we believe explanations of issues in students' programs could serve as reasoning steps (Wei et al., 2023), enhancing the subsequent generation of repairs (Chen et al., 2023). These refined repairs, in turn, could facilitate the generation of high-quality next-step hints (Roest et al., 2024). Research investigating the interplay between different types of feedback is thus pivotal in unlocking the full potential of language models to support programming education. By studying the performance of generating repairs without conditioning on feedback, nor generating feedback based on repairs, our work establishes a foundational understanding that will allow the research community to assess the extent to which various prompting techniques enhance feedback performance.

6 Conclusions

In this article, we have uncovered an intriguing relationship between LLM performance in program repair and the capability to explain issues in code. Our evaluations encompassed both open-source and proprietary models, examining their generic performance as well as concept-specific proficiency.

While selecting and deploying a specific language model may not be challenging, identifying the most suitable one for a particular purpose can be complex, particularly when considering financial, hardware, or other limitations. At a time when there are calls to rethink how programming is taught (Denny et al., 2024), the insights gleaned from our work can provide valuable guidance for educators in choosing LLMs that align with their instructional contexts.

Future work. Our future work will involve two specific directions. First, we'll continue our investigation of the relationships between various types of programming feedback and program repair. all these efforts remain an attempt to streamline the selection process of language models based on automated evaluation measures.

Besides studying LLM performance, our second objective is to leverage our computational resources to improve these LLMs' ability to provide feedback. In particular, small language models' poor explaining performance suggests that these models will benefit from alignment procedures designed specifically to improve feedback abilities (Scarlatos et al., 2024).

Limitations

Our work is not free of limitations. We evaluated the LLMs on a subset of solutions from a single dataset (from one institution with one programming language). Moreover, our evaluation of natural language explanations relied on GPT-4, which, although a state-of-the-art language model, is not a perfect evaluator. Human evaluation is necessary to strengthen our results. Furthermore, refinement would benefit the evaluation prompt (e.g., allowing GPT-4 to reason (Wei et al., 2023) before providing its final answers). Additionally, the results of our evaluation also depend on the specific prompts used to interact with each language model. Similarly, our benchmarking experiment was not exhaustive – although we included many popular state-of-the-art open-source and proprietary models, many more exist. Including more models would be necessary to strengthen the claim of the relationship between repair and natural language explanations. Beyond this, the concept analysis is only indicative, as many assignments feature multiple concepts. Finally, we only considered single-turn zero-shot repair, which does not take advantage of LLMs' ability to reason with few-shot examples (Brown et al., 2020), or LLMs' ability to correct their own mistakes (Chen et al., 2023; Xia and Zhang, 2023).

Ethics Statement

The work in the present article has been conducted following national and institutional ethics guidelines. We recognize the increasing importance of ethical considerations in artificial intelligence research, particularly concerning data usage and potential societal impacts.

The dataset employed in this research is openly available to researchers. Our overarching goal is to contribute to the development and evaluation of open-source language models for providing feedback in programming education. By focusing on open-source models, we aim to promote transparency, accessibility, and accountability in AI research and development, thereby addressing concerns regarding the privacy implications of using proprietary language models.

We further acknowledge the broader ethical implications of our work, including issues related to fairness and accessibility of LLM feedback, how LLMs might favour certain styles of interaction, and how LLMs might contribute to inequalities in the quality of provided education worldwide.

References

Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. 2022. Synfix: Automatically fixing syntax errors using compiler diagnostics.

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The falcon series of open language models.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and

Charles Sutton. 2021. Program synthesis with large language models.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating language models trained on code.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug.

Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Wei Zhu, Yuan Ni, Guotong Xie, Zhiyuan Liu, and Maosong Sun. 2023. Ultrafeedback: Boosting language models with high-quality feedback.

Adrian de Freitas, Joel Coffman, Michelle de Freitas, Justin Wilson, and Troy Weingart. 2023. Falconcode: A multiyear dataset of python code samples from an introductory computer science course. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 938–944, New York, NY, USA. Association for Computing Machinery.

Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. 2023. Computing education in the era of generative ai. *arXiv preprint arXiv:2306.02608*.

Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing education in the era of generative ai. *Commun. ACM*, 67(2):56–67.

Nigel Fernandez, Alexander Scarlatos, and Andrew Lan. 2024. Syllabusqa: A course logistics question answering dataset.

James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, ACE '22, page 10–19, New York, NY, USA. Association for Computing Machinery.

Google. 2024. Gemma: Open models based on gemini research and technology. Technical report, Google DeepMind. Https://storage.googleapis.com/deepmind-media/gemma/gemma-report.pdf.

Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. ArXiv:1603.03165 [cs].

John Hattie and Helen Timperley. 2007. The power of feedback. *Review of educational research*, 77(1):81–112.

Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the responses of large language models to beginner programmers' help requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 93–105, New York, NY, USA. Association for Computing Machinery.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps.

Yann Hicke, Anmol Agarwal, Qianou Ma, and Paul Denny. 2023. Ai-ta: Towards an intelligent question-answer teaching assistant using open-source llms.

Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019a. Refactoring based program repair applied to programming assignments. In *2019 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, pages 388–398. IEEE/ACM.

Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019b. Refactoring based program repair applied to programming assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 388–398. IEEE/ACM.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b.

Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43.

Teemu Koivisto and Arto Hellas. 2022. Evaluating codeclusters for effectively providing feedback on code submissions. In *2022 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE.

Charles Koutcheme. 2023. Training Language Models for Programming Feedback Using Automated Repair Tools. In *Artificial Intelligence in Education*, pages 830–835, Cham. Springer Nature Switzerland.

Charles Koutcheme, Nicola Dainese, Sami Sarsa, Arto Hellas, Juho Leinonen, and Paul Denny. 2024. Open source language models can provide feedback: Evaluating llms' ability to help students using gpt-4-as-a-judge. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education, Volume 1*, ITICSE '24.

Charles Koutcheme, Nicola Dainese, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. 2023a. Benchmarking educational program repair. In *NeurIPS'23 Workshop on Generative AI for Education (GAIED)*. NeurIPS.

Charles Koutcheme, Sami Sarsa, Juho Leinonen, Lassi Haaranen, and Arto Hellas. 2023b. Evaluating distance measures for program repair. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 495–507, New York, NY, USA. Association for Computing Machinery.

Charles Koutcheme, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. 2023c. Automated Program Repair Using Generative Models for Code Infilling. In *Artificial Intelligence in Education*, pages 798–803, Cham. Springer Nature Switzerland.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation.

Teemu Lehtinen, Charles Koutcheme, and Arto Hellas. 2024. Let's ask ai about their programs: Exploring chatgpt's answers to program comprehension questions. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24.

Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using language models to enhance programming error messages. In *Proceedings of the 2023 ACM SIGCSE Technical Symposium on Computer Science Education*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you!

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2017, page 55–56, New York, NY, USA. Association for Computing Machinery.

Tal Linzen. 2020. How can we accelerate progress towards human-like linguistic generalization? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5210–5217, Online. Association for Computational Linguistics.

Qi Liu, Shuanghong Shen, Zhenya Huang, Enhong Chen, and Yonghe Zheng. 2023. A survey of knowledge tracing.

Rongxin Liu, Carter Zenke, Charlie Liu, Andrew Holmes, Patrick Thornton, and David J. Malan. 2024. Teaching cs50 with ai: Leveraging generative artificial intelligence in computer science education. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 750–756, New York, NY, USA. Association for Computing Machinery.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation.

Ali Malik, Mike Wu, Vrinda Vasavada, Jinpeng Song, Madison Coots, John Mitchell, Noah Goodman, and Chris Piech. 2021. Generative Grading: Near Human-level Accuracy for Automated Feedback on Richly Structured Problems. In *Proceedings of the 14th Educational Data Mining conference*.

Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2021. A survey of automated programming hint generation: The hints framework. *ACM Computing Surveys (CSUR)*, 54(8):1–27.

Hunter McNichols, Wanyong Feng, Jaewook Lee, Alexander Scarlatos, Digory Smith, Simon Woodhead, and Andrew Lan. 2024. Automated distractor and feedback generation for math multiple-choice questions via in-context learning.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM conference on innovation and technology in computer science education*, pages 92–97.

Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023a. Generating high-precision feedback for programming syntax errors using language models.

Tung Phung, Victor-Alexandru Pădurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. 2023b. Automating human tutor-style programming feedback: Leveraging gpt-4 tutor model for hint generation and gpt-3.5 student model for hint validation.

Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code.

James Prather, Paul Denny, Juho Leinonen, Brett A Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, et al. 2023. The robots are here: Navigating the generative ai revolution in computing education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education*, pages 108–159.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model.

Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64.

Lianne Roest, Hieke Keuning, and Johan Jeuring. 2024. Next-step hint generation for introductory programming using large language models. In *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, page 144–153, New York, NY, USA. Association for Computing Machinery.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. 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*, pages 27–43.

Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023a. Thrilled by your progress! large language models (gpt-4) no longer struggle to pass assessments in higher education programming courses. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 78–92, New York, NY, USA. Association for Computing Machinery.

Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. 2023b. Can generative pretrained transformers (gpt) pass assessments in higher education programming courses? *arXiv preprint*.

Alexander Scarlatos, Digory Smith, Simon Woodhead, and Andrew Lan. 2024. Improving the validity of automatically generated feedback via reinforcement learning.

Ben Leong Shubham Sahai, Umair Z. Ahmed. 2023. Improving the coverage of gpt for automated feedback on high school programming assignments. In *NeurIPS'23 Workshop on Generative AI for Education (GAIED)*. NeurIPS.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.

Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Kashif Rasul, Younes Belkada, Shengyi Huang, Leandro von Werra, Clémentine Fourrier, Nathan Habib, Nathan Sarrazin, Omar Sanseviero, Alexander M. Rush, and Thomas Wolf. 2023. Zephyr: Direct distillation of lm alignment.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022a. Finetuned language models are zero-shot learners.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022b. Emergent abilities of large language models.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Huggingface's transformers: State-of-the-art natural language processing.

Mike Wu, Noah D. Goodman, Chris Piech, and Chelsea Finn. 2021. Prototransformer: A meta-learning approach to providing student feedback. *CoRR*, abs/2107.14035.

Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair.

Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 740–751.

Dani Yogatama, Cyprien de Masson d'Autume, Jerome Connor, Tomas Kocisky, Mike Chrzanowski, Lingpeng Kong, Angeliki Lazaridou, Wang Ling, Lei Yu, Chris Dyer, and Phil Blunsom. 2019. Learning and evaluating general linguistic intelligence.

Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing bugs in python assignments using language models.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. Tinyllama: An open-source small language model.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023a. Lima: Less is more for alignment.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023b. CodeBERTScore: Evaluating code generation with pretrained models of code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13921–13937, Singapore. Association for Computational Linguistics.

A Experiment details

A.1 Prompts used

Figure 4 (resp. Figure 5) shows our prompts to obtain repairs (resp. feedback) from the language models. Figure 6 shows the prompt used to grade the feedback generated by the language models using GPT-4 as our automatic evaluator (we adapt the prompt from (Koutcheme et al., 2024)). The reported value for "completeness" corresponds to the proportion of "yes" responses across our test dataset to the first criterion, while the reported value for the hallucination rate corresponds to the proportion of "no" responses to the second criterion. We note that regarding the issues present in the students' incorrect program, we assumed them to be identified by GPT-4 during evaluation (without a separate prompt). We acknowledge the limitations of this prompting strategy (i.e., no space for reasoning) which we'll refine in future work.

A.2 Official model names

Table 3 translates each model name into their Hugginface id [2].

[2]https://huggingface.co/models

Repair generation

You are a computer science professor teaching introductory programming using Python. ①

Bellow is a problem description and an incorrect program submitted by a student. Repair the student program with as few changes as possible such that the corrected program fulfils the requirements of the problem description. The corrected Python code must be between "'python and "'." ②

$\left.\begin{array}{l}\text{**Problem:**}\\ \text{<handout>}\\ \\ \text{**Incorect code:**}\\ \text{<submitted\_code>}\end{array}\right\}$ ③

Figure 4: Our template for prompting the LLMs to provide feedback. (1) A system prompt specifying the behaviour of the model. (2) A description of the grading task. (3) Information necessary to grade the feedback.

Feedback generation

You are a computer science professor teaching introductory programming using Python. ①

Below is a problem statement and an incorrect program submitted by a student. List and explain all the issues in the student program that prevent it from solving the associated problem and fulfilling all the requirements in the problem description. ②

$\left.\begin{array}{l}\text{**Problem:**}\\ \text{<handout>}\\ \\ \text{**Incorect code:**}\\ \text{<submitted\_code>}\end{array}\right\}$ ③

Figure 5: Our template for prompting the LLMs to provide feedback. (1) A system prompt specifying the behaviour of the model. (2) A description of the grading task. (3) Information necessary to grade the feedback.

Table 3: Official model names for HuggingFace models.

name	HuggingFace/OpenAI id
TinyLlama	TinyLlama/TinyLlama-1.1B-Chat-v1.0
CodeLlama	codellama/CodeLlama-7b-hf
Llama	meta-llama/Llama-2-7b-chat-hf
Mistral	mistralai/Mistral-7B-v0.1
Zephyr	HuggingFaceH4/zephyr-7b-beta
Gemma	google/gemma-7b-it

Judging

You are a computer science professor teaching introductory programming using Python. ①

Below is a problem description, and an incorrect program written by a student. You are also provided with the feedback generated by a language model. Your task is to evaluate the quality of the feedback (by saying yes or no) to ensure it adheres to the multiple criteria outlined below. For each criterion, provide your answer in a separate line with the format '(CRITERIA_NUMBER): Yes/No'. Do not provide comments, but be attentive to the problem description requirements. ②

$\left.\begin{array}{l}\text{\#\# Problem description:}\\ \text{<handout>}\\ \\ \text{\#\# Student Code:}\\ \text{<submitted\_code>}\\ \\ \text{\#\# Feedback:}\\ \text{<feedback>}\\ \\ \text{\#\# Criteria:}\\ \\ \text{(1) Identifies and mentions all actual issues}\\ \text{(2) Does not mention any non-existent issue}\end{array}\right\}$ ③

Figure 6: Judging prompt template. We provide (1) a system prompt specifying GPT-4's behaviour, (2) a description of the grading task, and (3) contextual information.

A.3 Concept analysis

Table 4 shows the number of exercises which practice each concept. Additionally, figure 7 shows an upset plot of the number of incorrect programs for which each combination of programming concepts is practised.

B Results details

B.1 Additional performance scores

Some work in program synthesis has evaluated the ability of language models to generate programs using another method to estimate pass@1. This method, originally proposed in the work of Chen et al. (Chen et al., 2021), is based on generating multiple samples, and is particularly adapted to non-instruction tuned models. We report the results of the program repair performance evaluation based on this multi-sample strategy.

Multi-sample performance evaluation. For each incorrect program, we generate $n = 20$ samples using top_p nucleus sampling and a temperature of 0.2 (Chen et al., 2021; Li et al., 2023). We evaluate functional correctness using the pass@1

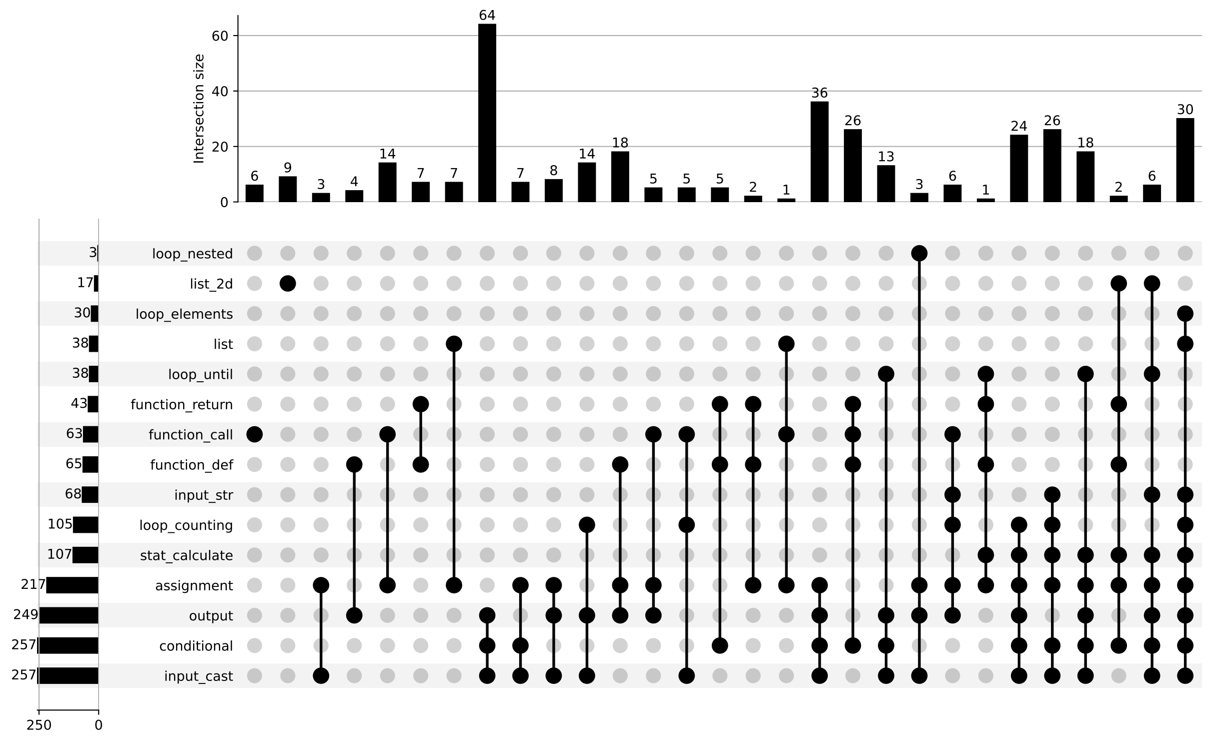

Figure 7: Programming concepts upset plot.

Table 4: Number of exercises and incorrect programs practised for each concept.

concept	# exercises	# programs
input string	4	18
input casting	27	257
output	28	249
assignment	26	217
conditional	22	257
function calling	8	63
function definition	9	65
function return	6	43
loop counting	9	105
loop until	5	38
loop elements	1	30
loop nested	1	3
stat calculation	10	38
list	3	38
list 2D	3	17

estimator, which tells us the probability that a language model will fix an incorrect program in a single attempt (Muennighoff et al., 2023).

To evaluate the ability of a language model to generate a solution close to the student program, we average the ROUGE-L score between each of the $k(k \leq n)$ candidate repairs that pass all unit tests and the incorrect program.

Results. Table 5 shows the performance results with the adapted pass@1 and rouge scores for a subset of the models (those with more than 7B parameters).

Table 5: We show the pass@1, rouge, completeness, and hallucination rate (hall. rate).

model	pass@1	rouge	completeness	hall. rate
Gemma-7b	0.267	0.353	0.905	0.005
Zephyr-beta	0.276	0.336	0.624	0.716
Mistral	0.304	0.365	0.738	0.397
gpt-3.5-turbo	0.529	0.561	0.838	0.368
gpt-4-turbo	0.634	0.559	0.992	0.024

In general, we notice an absolute drop in performance from the greedy decoding. Beyond this absolute difference, the main change is that the ranking of the model changed. Gemma-7B is now the least performant of the 7B parameters models.

The performance of the 7B parameters model are dependent on these.

B.2 Programming concepts performance

Table 6 shows the detailed per concept performance results for all models.

Table 6: Per concept performance results. Legend: IS (input string), IC (input casting), O (output), A (assignment), C (conditionals), FC (function call), FD (function definition), FR (function read), LC (loop counting), LU (loop until), SC (stat calculate), L (list), L2D (list 2D).

(a) Pass@1

	IS	IC	O	A	C	FC	FD	FR	LC	LU	SC	L	L2D
TinyLlama	0.04	0.03	0.03	0.07	0.05	0.21	0.09	0.14	0.03	0.03	0.04	0.03	0.06
Gemma-2b	0.06	0.13	0.13	0.12	0.19	0.44	0.60	0.77	0.13	0.05	0.10	0.11	0.47
CodeLlama	0.22	0.18	0.24	0.24	0.26	0.54	0.52	0.56	0.19	0.29	0.21	0.24	0.59
Zephyr-beta	0.10	0.17	0.17	0.24	0.25	0.60	0.58	0.86	0.23	0.26	0.26	0.26	0.41
Mistral	0.13	0.23	0.22	0.27	0.28	0.56	0.49	0.67	0.19	0.24	0.24	0.34	0.65
Gemma-7b	0.16	0.22	0.25	0.29	0.25	0.52	0.52	0.53	0.21	0.47	0.21	0.26	0.47
gpt-3.5-turbo	0.44	0.41	0.50	0.52	0.46	0.84	0.86	0.91	0.49	0.50	0.55	0.68	0.76
gpt-4-turbo	0.21	0.58	0.63	0.64	0.63	0.86	0.92	1.00	0.39	0.50	0.48	0.42	0.76
average	0.17	0.24	0.27	0.30	0.30	0.57	0.57	0.68	0.23	0.29	0.26	0.29	0.52

(b) Completeness

	IS	IC	O	A	C	FC	FD	FR	LC	LU	SC	L	L2D
TinyLlama	0.04	0.07	0.07	0.04	0.08	0.03	0.08	0.07	0.04	0.00	0.05	0.08	0.18
Gemma-2b	0.15	0.15	0.14	0.17	0.15	0.21	0.20	0.26	0.16	0.05	0.17	0.18	0.06
CodeLlama	0.31	0.33	0.32	0.37	0.35	0.43	0.35	0.35	0.33	0.45	0.42	0.26	0.24
Zephyr-beta	0.54	0.64	0.65	0.59	0.63	0.60	0.51	0.51	0.55	0.71	0.60	0.76	0.59
Mistral	0.81	0.74	0.71	0.77	0.75	0.79	0.69	0.79	0.73	0.76	0.79	0.79	0.82
Gemma-7b	0.94	0.94	0.92	0.91	0.95	0.76	0.86	0.91	0.90	1.00	0.94	0.95	1.00
gpt-3.5-turbo	0.93	0.82	0.82	0.87	0.84	0.84	0.86	0.81	0.83	0.76	0.89	0.97	0.94
gpt-4-turbo	1.00	0.99	0.99	1.00	0.99	0.98	0.98	0.98	0.99	0.97	1.00	1.00	1.00
average	0.59	0.58	0.58	0.59	0.59	0.58	0.57	0.58	0.57	0.59	0.61	0.62	0.60

(c) ROUGE

	IS	IC	O	A	C	FC	FD	FR	LC	LU	SC	L	L2D
TinyLlama	0.04	0.03	0.03	0.07	0.05	0.17	0.08	0.13	0.03	0.03	0.04	0.03	0.06
Gemma-2b	0.05	0.11	0.11	0.11	0.15	0.32	0.45	0.57	0.12	0.05	0.10	0.09	0.31
CodeLlama	0.20	0.15	0.20	0.22	0.21	0.46	0.45	0.47	0.17	0.25	0.18	0.21	0.51
Zephyr-beta	0.07	0.13	0.13	0.20	0.19	0.50	0.48	0.71	0.18	0.21	0.21	0.22	0.32
Mistral	0.08	0.15	0.16	0.20	0.20	0.44	0.38	0.52	0.14	0.15	0.16	0.26	0.48
Gemma-7b	0.16	0.20	0.22	0.28	0.23	0.49	0.47	0.49	0.20	0.43	0.21	0.25	0.44
gpt-3.5-turbo	0.41	0.37	0.44	0.47	0.41	0.74	0.76	0.80	0.45	0.45	0.51	0.63	0.72
gpt-4-turbo	0.17	0.46	0.51	0.52	0.50	0.70	0.72	0.77	0.31	0.40	0.38	0.35	0.61
average	0.15	0.20	0.22	0.26	0.24	0.48	0.47	0.56	0.20	0.25	0.22	0.26	0.43

(d) hallucination rate

	IS	IC	O	A	C	FC	FD	FR	LC	LU	SC	L	L2D
TinyLlama	0.47	0.30	0.25	0.35	0.31	0.41	0.42	0.40	0.43	0.21	0.40	0.39	0.18
Gemma-2b	0.12	0.37	0.42	0.37	0.35	0.32	0.48	0.40	0.24	0.55	0.28	0.13	0.65
CodeLlama	0.87	0.86	0.87	0.82	0.85	0.75	0.82	0.81	0.82	0.89	0.80	0.97	0.88
Zephyr-beta	0.88	0.79	0.78	0.80	0.78	0.46	0.60	0.49	0.86	0.61	0.89	0.87	0.65
Mistral	0.43	0.42	0.42	0.39	0.41	0.32	0.32	0.35	0.41	0.32	0.37	0.45	0.41
Gemma-7b	0.00	0.01	0.01	0.01	0.01	0.00	0.00	0.00	0.01	0.03	0.02	0.00	0.00
gpt-3.5-turbo	0.28	0.31	0.35	0.36	0.35	0.54	0.49	0.51	0.34	0.21	0.29	0.24	0.24
gpt-4-turbo	0.00	0.02	0.02	0.02	0.02	0.00	0.03	0.05	0.00	0.05	0.01	0.05	0.06
average	0.38	0.38	0.39	0.39	0.38	0.35	0.40	0.38	0.39	0.36	0.38	0.39	0.38