Toward diagnosis of semantic errors in Python programming platforms for beginners

Badmavasan Kirouchenassamy, Amel Yessad, Sébastien Jolivet, Vanda Luengo

To cite this version:

Badmavasan Kirouchenassamy, Amel Yessad, Sébastien Jolivet, Vanda Luengo. Toward diagnosis of semantic errors in Python programming platforms for beginners. Workshop RKDE - ECML PKDD Conference, Sep 2024, Vilnius, Lithuania. ⟨hal-05183363⟩

HAL Id: hal-05183363
https://hal.science/hal-05183363v1

Submitted on 23 Jul 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Toward diagnosis of semantic errors in Python programming platforms for beginners

Badmavasan Kirouchenassamy[1] [0009−0003−6502−154X],
Amel Yessad[1] [0000−0001−7575−6433],
Sébastien Jolivet[2] [0000−0003−3915−8465], and
Vanda Luengo[1] [0000−0003−3915−8465]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
{badamavsan.kirouchenassamy,amel.yessad,vanda.luengo}@lip6.fr
[2] IUFE & TECFA, Université de Genève, Suisse
sebastien.jolivet@unige.ch

Abstract. Reliable semantic error analysis of student codes can be used to improve both the learning and teaching experience, through a better understanding of students' codes. We propose a novel approach that integrates automatic semantic error detection in student codes with the construction of a solution space for analyzing students' programming trajectories.
This paper addresses the challenge of automatically annotating codes with predefined error tags and assisting the construction of a solution space, leveraging generative AI based on Large Language Models (LLMs) to streamline these processes and save time for experts. To achieve this, we carried out two studies.
The first study is focused on annotating student erroneous codes with error tags. Kappa analysis was employed to measure the agreement between human-annotated codes and AI-generated annotations, assessing the relevance and accuracy of the LLMs in error detection. The results revealed agreements upto a Cohen's kappa coefficient of 0.39.
The second study examined the construction of a solution space by analyzing the overlap between AI-generated and real student erroneous codes. This evaluation aimed to determine the capacity of the LLMs in generating erroneous codes used for constructing the solution space. The results showed that, on average, 68% of the analyzed students' erroneous codes were covered by the AI-generated codes.
The results of the first study are mitigated compared to the second ones but the combination of these two approaches could appear as positive to compensate for each other and improvements are ongoing investigation.

Keywords: Error Diagnosis · Code Analysis · Code Generation · Solution Space · Generative AI

1 Introduction

In the world of online learning platforms, especially those designed for beginner programmers, a key challenge is tailoring instructions to meet the needs of

individual students [1, 23, 8]. The ability to automatically provide meaningful and accurate understanding of student erroneous code is useful for adaptation. This could potentially foster the expansion of large-scale programming education initiatives, in particular, knowledge about students' errors and their error trajectories to the correct solution is useful for multiple reasons. For example, it can be used to improve the teaching experience by providing teachers with insights about the students' errors and misconceptions [18, 21]. It can also improve the learning experience by adapting instruction or feedback to each student [16]. Our long-term objectives are to provide teachers with insights into student errors and misconceptions and to enable tailored automatic feedback rather than just generic hints toward correct solutions [16, 3, 14].

In our research, we are interested in semantic errors in students' codes which are syntactically correct yet incorrect to the given programming task. Identifying such errors often takes time, making it difficult for teachers to provide feedback to students at scale [2, 25].

AI has been increasingly leveraged to detect errors in student-submitted code on online programming learning platforms. Researchers such as [8, 14, 3] have contributed to developing sophisticated systems that employ various AI techniques. Common approaches include machine learning algorithms that analyze code patterns, natural language processing (NLP) methods to understand code semantics, and heuristic-based models for identifying common syntax and logical errors. Techniques such as neural networks, decision trees, and clustering have been used to provide personalized feedback and identify recurring mistakes.

There are many issues with AI-approaches for understanding programming tasks which are open-ended tasks such as: the student codes are difficult and time-consuming to label with fine-grained errors and we need to understand student code even for the very first student, without using historical data of student's previous submissions. Instead of using experts' annotated historical data, we introduce a few-shot approach to automatically understand student-produced erroneous codes and student error trajectories. This approach is based on a framework that automatically assists the analysis of student codes and their trajectories towards reaching a correct code through two complementary methods: first, leveraging on an error annotation method to identify errors within students codes; and second assisting the construction of a solution space which is a graph of erroneous codes as well as correct solutions. The codes are linked by the repair distance. Positioning the student code in this space will allow us to gauge a gap between the student submission and one correct code from the solution space therefore by identifying the repair actions to reach this latter. These two methods are generally expert-time consuming.

To minimize the need for human intervention, our approach harnesses the capabilities of generative AI based on Large Language Models (LLM) to aid in the generation of erroneous solutions required for constructing the solution space. We tested several LLM, including ChatGPT and Mistral AI. Additionally, we tested these AI models to annotate student-produced erroneous codes with a predefined set of error tags. Since their emergence, large language models are

impacting the field of computing education with their code-generating capabilities [15, 5]
In this paper, we aim to answer two research questions:

- In what ways, and to what degree, do generative AI and human error annotation of student codes agree or disagree ? (*RQ1*)
- To what extent generative AIs can be used to generate student like code for assisting the automatic construction of the solution space ? (*RQ2*)

For **RQ1**, We assessed generative AI's ability to tag student code with errors by comparing AI-generated annotations with expert-annotated ones. For **RQ2**, we evaluated the overlap between AI-generated and student-produced erroneous codes to gauge the AI's capacity to generate student-like errors.

2 Related Work

There have been several research exploring means for error detection in student codes and the construction of a graph of students solutions.

Research in programming tutors has focused on analyzing student code to provide assistance, often by transforming erroneous and correct code into graph-based representations like Abstract Syntax Trees (ASTs) for comparison and applying edit transformations to convert student trees into correct ones [14, 7]. Singh et al. introduced an Error Model Language (EML) with correction rules, providing a structured approach to code correction [20]. More recent research has expanded to include a variety of techniques, ranging from conventional assembly methods [19] to dynamic neural programming embedding [24] and the use of Large Language Models (LLMs) [9].

In constructing solution spaces, approaches can be categorized into expert-driven and data-driven methods. Data-driven approaches, such as those by Piech et al.[13] and Rivers et al.[16], leverage student-produced erroneous code to generate solution spaces by applying transformations until the code matches expected correct solutions. This method aims to automate hint generation but faces challenges such as the extensive nature of solution spaces and the need for continual adjustment with each new student submission, especially given the uniqueness of each code. Additionally, new exercises often require few-shot techniques for effective adaptation [11]. An alternative approach involves creating a Markovian Decision Process (MDP) based on available data to establish transition probabilities between different states, as proposed by Barnes et al. [3]. While this method offers a degree of automation, it still requires human intervention to define the solution space.
Conversely, expert-driven methods, such as those discussed by Gross et al.[6] and Barnes et al.[22], involve constructing solution spaces with substantial input from experts. These methods, while precise, are time-consuming and heavily reliant on expert knowledge.
To address these challenges, our approach proposes a hybrid method that combines local error detection within student code with its projection into a pre-

constructed erroneous solution space. This integration aims to provide a comprehensive and precise perspective on errors, facilitating understanding even in zero-shot or few-shot contexts. By leveraging both local and global perspectives on error detection, our method could potentially be utilized for both inner loop (within-exercise tutor guidance) and outer loop (between-exercise tutor guidance) adaptation, enhancing the overall effectiveness of programming tutors [1].
The construction of solution spaces presents significant computational challenges. For instance, the approach described in [16] requires extensive computation time to calculate all possible power edits from a given student submission to the correct solutions. A common challenge inherent to all solution space methodologies is their reliance on historical student data to generate these spaces. Consequently, these approaches may face difficulties when applied to new exercises. The methods proposed by [10, 13] are particularly intricate as they attempt to infer the learner's thought process, which presents precision issues. Retracing a learner's logical thought process solely based on the submitted code proves to be a formidable task.
A prevalent challenge across all methods is that they aim to assist learners in overcoming errors by identifying these errors at an algorithmic execution level. We hypothesize that identifying and classifying student errors can lead to a more nuanced understanding of these errors. This improved understanding could facilitate the provision of more effective feedback and enhance error diagnosis, ultimately contributing to a better comprehension of learner behavior.

3 Context of the studies

We carried out two studies on the online programming platform [3], which hosts a diverse set of exercises organized in sections. We focused specifically on the "for loop" section because loops have been identified as among language features that are especially problematic for beginners, and could benefit from particular attention [17]. We gathered 55924 student codes from 5246 students across 6 exercises. Sixty codes were used for the first study on annotations and the entire dataset was used for the second study on AI erroneous code generation.
These data outline a curated set of six exercises designed to address distinct programming challenges. The selection process was meticulous, covering a spectrum of scenarios. The exercises encompass four console-display-oriented tasks (2,3 and 4) along with one factorial exercise necessitates mathematical proficiency, potentially impacting students' error patterns(5), one exercise in drawing figure (1) and finally one exercise on guiding a robot's movement from point A to point B (6).

4 Error annotation using Large-scale Language Models

For the first study aimed at addressing the research question 1 (RQ1), we enlisted a group of annotators. We selected programming exercises and student erroneous

[3] algopython.fr

codes from the platform Algopython, providing them as a corpus to annotate by both generative AI systems and human annotators. The objective was to investigate and compare the error annotations of the generative AI against that of human experts.

4.1 Context, LLMs and dataset

The study involved recruiting eight participants with diverse backgrounds, including high school math teachers, informatics teachers in high school, and university-level computer science instructors who all teach python. This diverse range of profiles ensures varied perspectives on python teaching and programming expertise. Recruitment was done through networks of computer science teachers and computer science laboratory members. The study was conducted online, providing a flexible and accessible environment for participants.
For this study, we aimed to utilize multiple large-scale language models (LLMs), opting for industry-grade models: GPT-4 from OpenAI and Mixtral8x7b from Mistral AI. We interfaced with these models via the OpenAI API and the Mistral AI API to facilitate communication with the large-scale language models.
Prior to the annotation process, we collaborated with a group of experts to predefined a set of errors applicable to all six programming exercises. These predefined errors by experts (see A.1) were then utilized to annotate student-produced erroneous code. To collect annotations, we developed a web application[4] to gather the following information:

- Expert annotations: Annotations of errors for student-produced erroneous codes and their comments on their annotation experience.
- Annotation Time: Annotation duration of each student-produced erroneous code.

In the annotation process using generative AI, to avoid a high variance of the models, we opted to generate multiple annotations for the same student-produced erroneous code, with the aim of stabilizing the generation process and ensuring consistency in the annotations. As input (See A.1), we provided the description of the exercise, all possible correct solutions, the student-produced erroneous code to annotate and finally the list of predefined error tags.
The human annotators conducted separate and independent annotations, while the two LLMs were prompted to generate annotations. The agreements between human annotators were evaluated by Fleiss's Kappa and the agreements between Generative AI models and experts were evaluated by Cohen's Kappa.

4.2 Results and Discussion

The Cohen's kappa calculated across all the student-produced erroneous codes did not demonstrate any significant agreement between zero-shot generative AI annotations and human experts (see **Fig.** 1). Consequently, we conducted a more

[4] http://annotation.mocah.lip6.fr/

detailed analysis of the annotations to explore potential reasons for this lack of agreement.

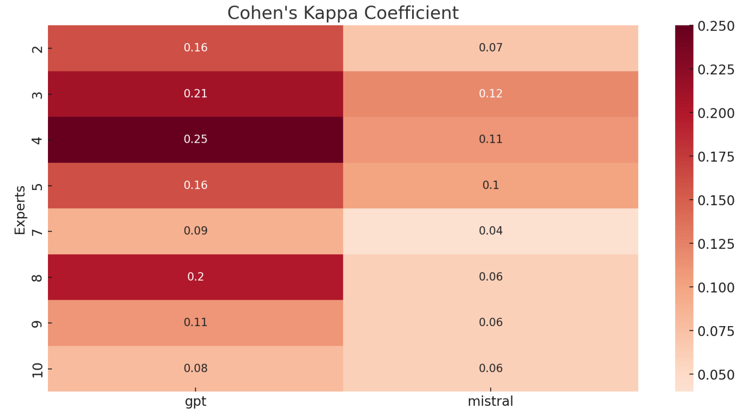

Fig. 1: Cohen's Kappa Coefficient across all exercises between experts and generative AI

For each exercise, we calculated the level of agreement among the experts. The results indicated that exercises 1, 2, and 6 exhibited good agreement among the experts. Exercise 4 also demonstrated good agreement among the experts, except for one annotator. Consequently, we removed the outlier, resulting in four exercises with good agreement among the experts. However, exercises 3 and 5 showed low agreement between experts. At first glance, exercises 3 and 5 appear to be slightly more challenging than the others. Exercise 3 involves nested loops, while exercise 5 requires prior understanding of mathematical concepts related to factorials, which may have influenced the difficulty level of the programming task. These complexities likely lead to the generation of more unique erroneous codes that are significantly distant from the expected solution. This highlights the lack of sufficient error tags to adequately address the complexity of these exercises, which is reflected in the low agreement among the annotators. This observation was also echoed by annotators through the platform.

Since the agreement between experts was acceptable only in these four exercises, we concentrated on calculating Cohen's kappa coefficient specifically for these exercises (see **Table** 1).

In the exercise 1, 2 and 6, despite good agreement between GPT and annotators, upon inspecting the generative AI annotations, we observed that Mistral annotated a "MISSING PRINT INSTRUCTION" error, whereas in the correct solution, there was no expected print statement. This discrepancy raises questions about whether it reflects a lack of understanding of the provided error tags

[4] * One Expert is not present in Exercise 1 as the expert did not annotate this given exercise.

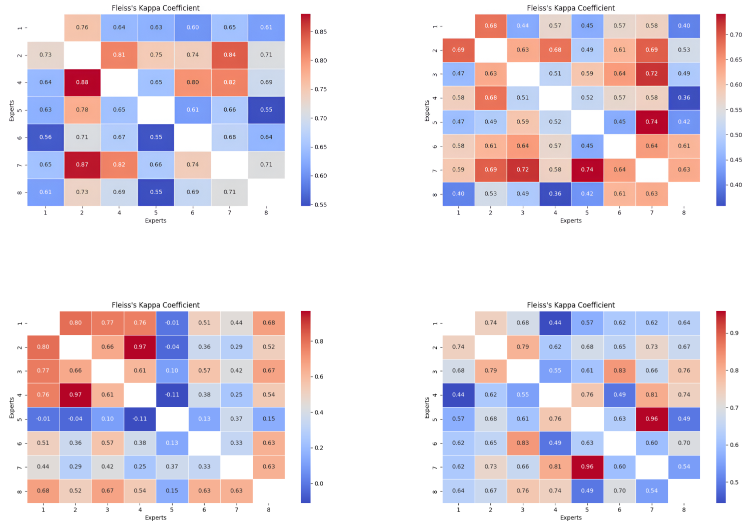

Fig. 2: Fleiss Kappa on exercises 1,2,4 and 6 (top left to right bottom)*

and their descriptions, or an understanding of the error present in the student code. It also demonstrates that between the two LLMs, the same prompt can yield different outputs. Out of all the four exercises, GPT-4 annotations outperform Mistral across three exercises, but only in the exercise 6 does Mistral outperforms GPT-4. This indicates that the understanding of the prompt as well as the interpretation of code differs between GPT-4 and Mistral AI. To explore these differences in detail, we investigated whether GPT and Mistral AI are in agreement. As a result of this evaluation, we obtain a Fleiss Kappa of 0.27 which is relatively low. Therefore, this indicates that the same prompt yields different results. However, it remains to be identified whether the difference stems from a lack of understanding of the student-produced erroneous code or from the prompt provided for the error tags.

As part of this study, we collected annotator comments about the annotation process, yielding several insights. Firstly, many annotators discussed the ambiguity present in the annotation process. They encountered situations where they were unsure if one error tag was sufficient or whether to include all relevant error tags, as some error tags are encompassed within others. For instance, when a student-produced erroneous code lacks instructions within the body of a for loop, annotators questioned whether the "FOR LOOP BODY MISMATCH" tag alone was adequate, or if "MISSING STATEMENTS" should also be included. Another insight that could aid in understanding the low agreement observed is that the annotation platform designed for this process allows annotators to annotate codes across multiple sessions. Several annotators mentioned that they couldn't recall the logic they followed to annotate previously during different

Table 1: Cohen's Kappa on exercises 1,2,4 and 6 (top left to right bottom)

Expert	Exercise 1		Exercise 2		Exercise 4		Exercise 6	
	GPT-4	Mistral	GPT-4	Mistral	GPT-4	Mistral	GPT-4	Mistral
1	**0.30**	0.19	0.21	-0.18	0.27	0.04	0.19	**0.31**
2	**0.33**	0.22	0.20	-0.14	0.27	0.06	0.21	**0.33**
3	-	-	**0.30**	-0.07	**0.31**	0.06	0.16	**0.32**
4	**0.33**	0.22	0.24	-0.11	0.29	0.07	0.16	0.19
5	0.18	0.02	0.18	-0.14	-0.06	-0.06	0.19	**0.30**
6	**0.38**	0.28	0.24	-0.06	**0.31**	0.01	0.18	0.28
7	0.23	0.14	0.22	-0.12	0.01	0.01	0.18	0.28
8	**0.39**	0.14	0.12	-0.14	0.17	-0.11	0.19	0.24

sessions, even when working on the same exercise. This variation in annotator behavior across sessions could potentially impact agreement between experts, as there is a lack of consistency within annotations from the same annotator.

4.3 Perspectives

To enhance the agreement between experts and generative AI models, we are studying three possible improvements:

- Provide examples of error tags: instead of relying solely on descriptions of error tags, it could be useful to provide examples of annotations for each tag. This approach will help verify whether the models truly understand the error tags by offering concrete instances of their application.
- Rework precise error tags: as previously highlighted, there is a need to refine and clarify the error tags to mitigate ambiguity. This involves revisiting and possibly restructuring the error tag taxonomy to ensure that annotators have clear guidelines for annotating student-produced erroneous code.
- Fine tune the LLMs on a few examples: training the models on student erroneous codes and their associated error tags can improve the prediction of error tags. By fine-tuning the models on such data, they can better comprehend and annotate errors in student code, ultimately leading to more accurate annotations.

5 Generating a solution space

We formalize a solution space as a weighted oriented graph consisting of student erroneous solutions and correct solutions interconnected with each other via edges. An edge is established between two nodes in the solution space if there are at least two successive submissions from a student corresponding to these nodes. The weight of an edge can be but not only the frequency of the edge which is calculated from students' historical data (see Fig 3). Depending on the context, different functions can assign different types of values to edges. For

example, the value of an edge can be a hint which allows the student to move from one node to another closer to the correct solution [16, 27] or an expert suggestion to encourage a student to make forward progress [13].

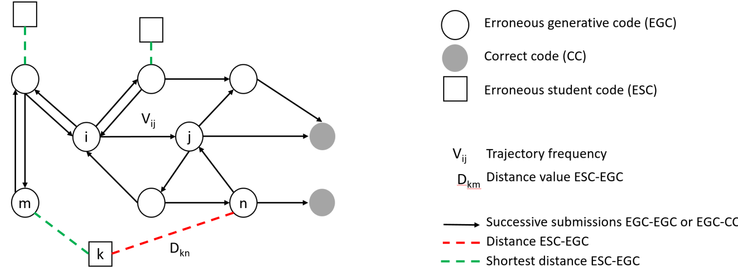

Fig. 3: Illustration of a solution space

Recent research [3, 8] has showed that the construction of a solution space is useful for multiple reasons. In our ongoing research, this space is useful to position a student erroneous partial code relatively to other erroneous solutions and to show possible trajectories to the corrects solutions. For each exercise, we have to construct a space of student solutions.

The manual construction of this space by experts is highly complex and time-consuming, while it is unbounded even for simple coding tasks and there is a huge variability in students' trajectories, and not all solutions including partial solutions can be predicted in advance [12, 16]. In cases where we have a sufficient number of student codes, we can use these to construct a solution space. But this latter method won't work in zero-shot or few-shot analysis with few student codes or in the case of a new exercise without any historical data.

The approach we propose is to use Generative AI to assist the generation of student-like erroneous solutions. We carried out a study to assess whether the Generative AI can generate student like erroneous solutions, including partial solutions.

5.1 Studied issues

In this second study, we aim to address the research question 2 (*RQ2*). In our endeavor, we identify two major issues associated with using generative AI to generate a solution space.

On the one hand, as the number of possible erroneous codes for one exercise can be very big and therefore the solution space unbounded we limit it to a reasonable-size space and propose a method to position each student solution in the space as follows: if the student solution is a node in the space then it is connected directly to this node. Otherwise, we connect the student solution to the closest node by using a variant of the Zhang Shasha distance [26].

On the other hand, Generative AI can suffer from hallucination, inferring erroneous codes that have nothing to do with codes written by students or fail to cover the huge variability in student codes. To assess these two aspects, we carried out a study whom the objective is to assess whether erroneous codes generated by generative AI closely resemble those produced by beginner-level programming students. We focused our investigation on codes originating from the online programming learning platform Algopython mentioned previously.

5.2 Distance between codes

To compare between a student code and a AI-generated codes, we first transformed them into abstract syntax trees (AST) and then calculating between ASTs a distance which is a derivative of the Zang Shahsa distance [26]. In our work, we focus on the structural and the algorithmic similarities between the two compared codes rather than on the exact correspondence between the constant values used in the statements. Thus, we have modified the original Zang Shahsa distance to consider all edit operations (node deletions, insertions, and replacements) except operations on constant values. For example, two codes are considered as identical even if different constant values are used in an assignment instruction or a different number of iterations in the case of the for loop. In Fig 4, the AI-generative code (right most) is the closest to the student code (left most) although the constants used are not identical but their structures are identical.

```
def isPositive(t):
   if t < 0:
       return True
   else:
       return False
```

```
def isPositive(t):
   if t == 0:
       return True
   else:
       return False
```

```
def isPositive(t):
   if t < 1:
       return True
   else:
       return False
```

Fig. 4: Erroneous student code (left most) and 2 generated erroneous codes (middle and right most)

5.3 Context and LLM

Throughout this second study, we define an erroneous code as one that is syntactically correct but contains semantic errors, deviating from the expected correct solution. For example, if a student-produced erroneous code contains manually repeated instructions instead of utilizing a for loop, it is considered syntactically correct but fails to meet the contextual requirements of the exercise. Such code differs entirely from the expected correct solution, which utilizes a for loop.

We employed an industry-grade code generation AI, ChatGPT (model gpt-4), renowned for its success in generating code [4]. we utilized the OpenAI API to directly access the GPT-4 model.

We gathered 55924 student codes from 5246 students across 6 exercises. These student-submitted codes served as the basis for assessing the overlap between AI-generated codes and actual student submissions.

5.4 Evaluation metrics

To quantitatively evaluate the overlap between student-produced erroneous codes and AI-generated erroneous codes, we defined two distinct metrics:

- **Coverage (COV):** this metric calculates the percentage of student-produced erroneous codes that are covered by AI-generated erroneous codes.
- **Precision of generation (PG):** this metric measures the number of AI-generated codes that match with student codes against the total number of AI-generated codes. A precision close to 100% indicates that generative AI is less prone to generating codes very distant from student codes, thereby reducing the likelihood of hallucinations.

5.5 Process, results and Insights

During the study, we iteratively queried the GPT-4 API with different prompts. The objective of the first queries was to assess the overlap between GPT-4 generated codes and student erroneous codes via the metrics COV and PG. We asked GPT-4 to generate 20 erroneous codes for each of the studied exercises. Following this first queries, the value of COV (respectively PG) was on average 22.02% (respectively on average 62.34%). The value of COV is relatively low considering our objectives . However, the high (PG) suggests that GPT-4 is not hallucinating and is effectively generating student-like erroneous code.

In response to the observed low values of COV, a more thorough analysis of student-produced erroneous codes revealed that several factors may have contributed to this low value. Primarily, discrepancies in prompts could result in inconsistencies within the generated codes. Additionally, the co-presence of multiple errors of diverse nature within student-produced erroneous codes while the first prompt didn't ask for generating code with multiples errors. Finally, the students' codes often manifested as partial, incomplete solutions, lacking the anticipated structural coherence and algorithmic consistency. To take into account these different characteristics in the generated codes, we designed another more complete prompt that takes into account all the mentioned factors.
Despite these improvements, the values of COV remained quite low whereas PG has increased(COV =26.46%, PG=98.3%) across all 6 exercises.

In an effort to improve the coverage (COV), we conducted two additional evaluations. Firstly, we gradually increased the number of AI-generated codes and monitored the evolution of COV and PG values (see **Fig.**5). Secondly, we fine-tuned the GPT-4 model using the dataset pertaining to exercise 2 and prompted generations on the fine-tuned model.
During the process of generating erroneous codes through multiple requests, as hypothesized, we observed a significant improvement in the similarity rate when

Table 2: Evaluation Metrics

Exercise Number	Percentage of Coverage	Precision of generation
1	29.96%	47.02%
2	67.21%	76.31%
3	48.16%	88%
4	41.78%	65.5%
5	11.57%	82.8%
6	53.99%	41.96%

increasing the number of AI-generated codes (see **Table.**2). However, an issue arose concerning the consistency of the output format across multiple generations. Despite specifying the desired output format in the prompt, inconsistencies persisted.

In some instances, we encountered variations in the output format, including the addition of comments and extraneous text. These inconsistencies hindered the standardization of the output, making it challenging to compare and analyze the generated codes effectively.

To address this issue and ensure consistency in the output format, we may need to refine our prompt and provide clearer instructions to the AI model. Additionally, further experimentation and adjustments to the generation process may be necessary to achieve more uniform results across multiple requests.

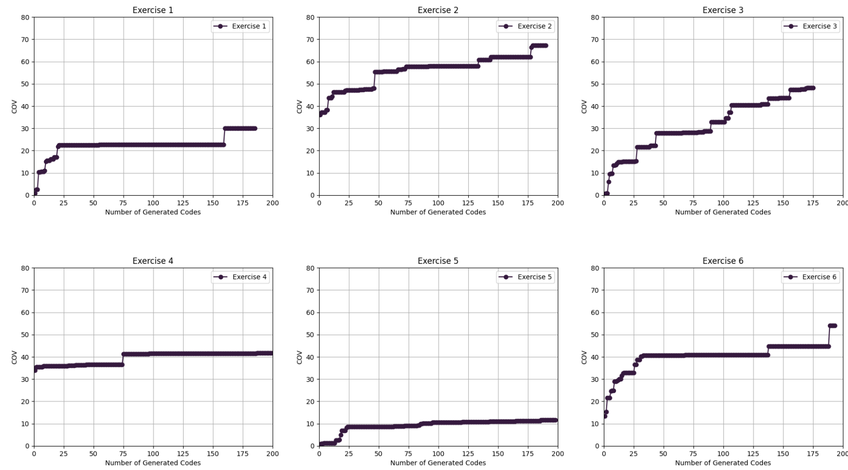

Fig. 5: Evolution of the COV values upon increasing number of AI-generated erroneous codes for exercises 1 to 6 (top left to right bottom)

Notably, Exercise 5, which pertains to factorial calculations, exhibited particularly low similarity compared to other exercises. Upon closer examination of student-produced erroneous codes, we discovered a prevalent misconception among students who mistakenly assumed that factorial was an integrated function in Python. While imprecise exercise descriptions may contribute to this confusion, many students also demonstrated a lack of knowledge regarding factorial calculations, resulting in incorrect formula applications.
Upon examining the results across six different exercises, two main conclusions can be drawn. Firstly, the **PG** is quite satisfactory, indicating that GPT is not entirely producing hallucinating. Secondly, the **COV** varies significantly across exercises. An interesting observation is that Exercise 6, which involves nested loops and is relatively more complex compared to Exercise 1 involving a single loop, exhibits better **COV** despite its complexity. It is noteworthy that the generated codes differ not only between different generations but also within the same generation. Throughout the experiment, approximately 200 codes were generated for each exercise to ensure an adequate representation within our solution space. However, it is plausible to induce creativity in generative AI by analyzing the generated graphs (**Fig.** 5). Increasing the number of generated codes tends to enhance the value of COV. Nevertheless, due to the inherent limitations of language model algorithms, particularly GPT-4, there may exist a saturation point in creativity that warrants further exploration. But in order to achieve this coverage with fewer generations, we fine-tuned the GPT-4 model specifically for our use case. In particular, we fine-tuned the model using few data from Exercise 2.
While our primary objective was to achieve a better **COV** between the generated codes and student-produced erroneous codes, we also aimed to standardize the output of the generated codes. Thus, through fine-tuning, we aimed to streamline the output format and ensure consistency in the generated codes. Fine-tuning the model with Exercise 2 data led to a notable increase in **COV** to **93.8%** and **PG** to **91.42%** between AI-generated and student-produced erroneous codes for that exercise. However, this improvement was accompanied by reduced the COV rates for other exercises, attributed to overfitting, where the model becomes too specialized in Exercise 2's patterns. Future studies aim to explore alternative approaches using open-source multi-models such as Mistral AI and Code Llama 3 to mitigate overfitting risks and enhance generalizability across all exercises.

6 Conclusion

In this paper, we present a few-shot framework to automatically understand student erroneous codes and student error trajectories. The ongoing research aims to automatically analyze student codes and their trajectories towards reaching a correct code through two complementary methods: annotating student codes with error tags and assisting the construction of a solution space. Two studies were carried out to explore the generative capabilities of large language models (LLM) in automating these two methods and saving expert time.

The study examining the ability of LLMs to annotate errors akin to human experts reveals that the agreement rate between LLM-generated annotations and expert annotations is slightly below the acceptance threshold. This indicates a need for fine-tuning the LLMs to better align with our specific use case. Furthermore, as previously noted, expert annotations should undergo multiple iterations to achieve high inter-expert agreement before comparing them with LLM outputs. Nonetheless, it can be concluded that LLMs do not exhibit hallucination behaviors, such as inventing new error tags or annotating non-existent errors. The primary limitation is that LLMs fail to annotate all errors identified by human experts.

The study on generating student-like code submissions using generative AI demonstrates success in both zero-shot and few-shot scenarios (with fine-tuning). The LLMs effectively produce code that closely resembles typical student submissions. However, achieving 100% resemblance is inherently challenging due to the presence of unique cases. Additionally, the platform's design for code submission and execution may introduce a bias, distinguishing between interactions with the machine and actual student submissions. This could account for the presence of multiple partial solutions.

As previously discussed, the objective of our novel approach is to develop a semantic error analysis of student codes that can operate effectively with minimal student data, specifically in zero-shot or few-shot scenarios. This first study shows that AI generative methods could be a reasonable solution in few-shot contexts through fine-tuning a pretrained foundation models. Our perspective is to study how this novel approach could be utilized for automatic code grading, feedback provision, and various applications in learning analytics.

Despite the potential commercial implications, alternatives such as Llama 3, Mistral AI, and Gemini could be considered. However, two major constraints exist: the need for annotated erroneous codes for fine-tuning, as open-source models do not achieve the precision of industrial models in zero-shot scenarios, and the computational resources required for model installation and fine-tuning, with models like Mistral AI necessitating significant GPU resources.

References

1. Vincent Aleven, Elizabeth A McLaughlin, R Amos Glenn, and Kenneth R Koedinger. Instruction based on adaptive learning technologies. *Handbook of research on learning and instruction*, 2:522–560, 2016.
2. Amjad Altadmri and Neil CC Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education*, pages 522–527, 2015.
3. Tiffany Barnes and John Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In *International conference on intelligent tutoring systems*, pages 373–382. Springer, 2008.
4. Arghavan Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel Desmarais, and Zhen Jiang. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734, 05 2023.

5. James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, pages 10–19, 2022.
6. Sebastian Gross, Bassam Mokbel, Barbara Hammer, and Niels Pinkwart. Feedback provision strategies in intelligent tutoring systems based on clustered solution spaces. 2012.
7. Bo Jiang, Wei Zhao, Nuan Zhang, and Feiyue Qiu. Programming trajectories analytics in block-based programming language learning. *Interactive Learning Environments*, 30(1):113–126, 2022.
8. Kenneth R Koedinger, Emma Brunskill, Ryan SJd Baker, Elizabeth A McLaughlin, and John Stamper. New potentials for data-driven intelligent tutoring system development and optimization. *AI Magazine*, 34(3):27–41, 2013.
9. Charles Koutcheme, Nicola Dainese, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. Benchmarking educational program repair. *Evaluation*, 3:4, 2023.
10. Ali Malik, Mike Wu, Vrinda Vasavada, Jinpeng Song, Madison Coots, John Mitchell, Noah Goodman, and Chris Piech. Generative grading: Near human-level accuracy for automated feedback on richly structured problems, 2021.
11. Allen Nie, Emma Brunskill, and Chris Piech. Play to grade: Testing coding games as classifying markov decision process. *Advances in Neural Information Processing Systems*, 34:1506–1518, 2021.
12. Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. The continuous hint factory-providing hints in vast and sparsely populated edit distance spaces. *arXiv preprint arXiv:1708.06564*, 2017.
13. Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the second (2015) acm conference on learning@ scale*, pages 195–204, 2015.
14. Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 153–160, 2012.
15. Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. Evaluating the performance of code generation models for solving parsons problems with small prompt variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, pages 299–305, 2023.
16. Kelly Rivers and Kenneth R Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27:37–64, 2017.
17. Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.
18. Philip M Sadler, Gerhard Sonnert, Harold P Coyle, Nancy Cook-Smith, and Jaimie L Miller. The influence of teachers' knowledge on student learning in middle school physical science classrooms. *American Educational Research Journal*, 50(5):1020–1049, 2013.
19. Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 313–316, New York, NY, USA, 2010. Association for Computing Machinery.

20. Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, 2013.
21. James C Spohrer and Elliot Soloway. Alternatives to construct-based programming misconceptions. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 183–191, 1986.
22. John Stamper, Tiffany Barnes, and Marvin Croy. Enhancing the automatic generation of hints with expert seeding. volume 21, pages 153–167, 01 2011.
23. Kurt VanLehn. Regulative loops, step loops and task loops. *International Journal of Artificial Intelligence in Education*, 26:107–112, 2016.
24. Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair, 2018.
25. Songwen Xu and Yam San Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4):360–384, 2003.
26. Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18:1245–1262, 12 1989.
27. Kurtis Zimmerman and Chandan R Rupakheti. An automated framework for recommending program elements to novices (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 283–288. IEEE, 2015.

A Appendix

A.1 Prompt Annotation

As an incorrect Python code classifier, your task is to analyze provided incorrect Python code snippet and corresponding correct code, along with a description of the exercise. Identify all relevant tags from the list of error tags given below. Do not use any other tag. Once tags are identified, return an array of tags without anything else. If no tags, then return NONE.
List of error tags (each tag has its explanation, but return only the tags and nothing else; example: ['tag1', 'tag2', ...]):

- **FOR LOOP MISSING**: A for loop is missing in the student's code.
- **FOR LOOP INCORRECT NUMBER OF ITERATION**: The number of iterations in a for loop is not correct.
- **FOR LOOP BODY MISMATCH**: The code inside the body of a for loop is incorrect.
- **INCORRECT ORDER OF STATEMENTS**: The order of statements in the student's code is incorrect compared to the correct code.
- **UNNECESSARY STATEMENTS**: There are one or more instructions in the student's code that are not in the correct code.
- **MISSING STATEMENTS**: There are missing instructions in the student's code compared to the correct code.

- **CONSTANT VALUE MISMATCH**: All instructions are present, but constant values such as numbers or strings are different from the correct code.
- **MISSING PRINT INSTRUCTION**: The print() function is missing in the student's code.
- **INVALID USE CASE OF FUNCTION**: The usage of predefined functions is incorrect.