Explainable AI in the Loop: An Instructor-Transformer Collaboration for Improving Explainability and Reliability of Feedback in Introductory Programming Classrooms

Muntasir Hoq[1], Bradford Mott[1], Seung Lee[1], Jessica Vandenberg[1], Narges Norouzi[2], James Lester[1] and Bita Akram[1]

[1]North Carolina State University, NC, USA
[2]University of California Berkeley, CA, USA

Abstract

Active learning is widely recognized as an effective approach to improving learning outcomes in introductory programming courses. However, limited instructional resources often restrict students' access to timely, personalized feedback, which is crucial for mastering foundational programming concepts. While advances in AI, particularly large language models (LLMs), offer scalable feedback solutions, their lack of explainability and reliability remains a challenge. This paper presents an AI-driven classroom assistant that leverages the strengths of LLMs in generating feedback while enhancing the explainability and reliability of the feedback. The AI-driven classroom assistant features an authoring tool that fosters instructor-LLM collaboration in designing programming problems, developing sample solutions, identifying patterns representing common misconceptions and logical errors, and generating feedback corresponding to each pattern. The assistant further features an AI engine that detects similar patterns in student code, matches them with instructor-identified patterns, and selects a set of relevant instructor-verified feedback. To enable this pattern-matching process, we pretrained an explainable embedding model on a large repository of Python code labeled by their correctness. The model is then fine-tuned on programming solutions for a new, unseen problem. Using an attention mechanism, key patterns from student submissions are identified and matched with instructor-selected programming patterns representing common errors and misconceptions, and the corresponding set of feedback is assigned to each programming solution. We conducted a human evaluation to assess the alignment of feedback selected by our system and that provided by instructors. The results demonstrate the system's strong potential in selecting appropriate feedback for student programming solutions.

Keywords

AI-driven classroom assistant, Explainable AI, Programming education, Adaptive feedback

1. Introduction

Active learning has been widely recognized as an effective approach for improving student engagement and learning outcomes in introductory programming courses [1]. A key component of active learning is adaptive and timely feedback, which helps students identify and correct misconceptions as they develop their programming skills [2]. However, the growing enrollment in computer science (CS) courses poses a significant challenge, as delivering individualized feedback at scale is resource-intensive, thereby constraining instructors' ability to effectively incorporate active learning activities [3].

Recent advancements in large language models (LLMs) have demonstrated their potential in analyzing student code and generating feedback [4, 5]. However, despite their strength, LLMs suffer from reliability and trustworthiness issues [6], making them unsuitable for unsupervised use in introductory programming courses [7]. Research has shown that misconceptions introduced early in a student's learning process can persist for a long time, further emphasizing the need for accurate and pedagogically sound feedback [8]. To address these challenges, we propose an AI-driven classroom assistant designed to support both instructors and students. This system facilitates activity design for instructors while ensuring students receive individualized, real-time feedback as they work through programming

CSEDM'25: 9th Educational Data Mining in Computer Science Education Workshop, June 20, 2025, Palermo, Sicily, Italy
mhoq@ncsu.edu (M. Hoq); bwmott@ncsu.edu (B. Mott); sylee@ncsu.edu (S. Lee); jvanden2@ncsu.edu (J. Vandenberg); norouzi@berkeley.edu (N. Norouzi); lester@ncsu.edu (J. Lester); bakram@ncsu.edu (B. Akram)

exercises. To ensure both reliability and explainability, we leverage a triangulated feedback mechanism that integrates instructors, LLMs, and an explainable AI model, leveraging the strengths of LLMs while reinforcing instructor expertise.

To enable effective instructor-LLM collaboration in designing instructional support, our AI-driven classroom assistant features an authoring tool that allows instructors to interact with the LLM in crafting programming problems, identifying common student errors, and generating targeted feedback. The system utilizes a modified version of the explainable Subtree-based Attention Neural Network (SANN) model [9, 10, 11, 12] to extract key structural components from student code. When incorrect submissions are detected, the model matches patterns in the code against instructor-defined misconceptions and assigns the most relevant feedback. The modified SANN model is pretrained on the FalconCode dataset, a large repository of Python programming submissions [13]. When a new programming problem is designed through the authoring tool, we generate a synthetic dataset of students' submissions based on instructor-authored problem/solution pairs, covering common student mistakes and misconceptions. We then fine-tune the pre-trained model on a synthetic dataset generated by LLMs to ensure that it selects relevant and accurate feedback for students' solutions generated in response to a problem unseen by the model. While our model enhances explainability, it remains susceptible to errors. To improve reliability, we incorporate LLMs as a secondary validation mechanism. When a code snippet is flagged as incorrect by our model, the LLM verifies this decision. If there is no consensus, feedback is withheld, and students are advised to consult teaching staff. If both models agree on an error, the system maps the erroneous code to instructor-identified patterns and selects the most relevant feedback.

To evaluate our approach, we conduct a case study using a problem from the FalconCode dataset that was not included in pretraining. We use the instructor authoring tool to generate potential solutions, common misconceptions, and relevant feedback. LLM-generated synthetic data for the new problem is then used to fine-tune our model before evaluating its performance on real student submissions from the Falcon dataset. Finally, we assess the effectiveness of our feedback-matching process through a human evaluation, comparing the feedback selected by our model to that chosen by expert instructors. Our results demonstrate the effectiveness of instructor-AI collaboration in providing reliable, trustworthy, timely, and scalable feedback to students in introductory programming courses. By combining explainable models, instructor expertise, and LLM capabilities, our approach enhances the quality of AI-assisted feedback, ensuring that students receive meaningful and pedagogically sound guidance while working on programming tasks.

2. Background

2.1. Active Learning in Programming Education

Active learning techniques have significantly influenced how introductory programming is taught, offering a participatory alternative to traditional lectures. Techniques such as pair programming, collaborative problem-solving, and real-time coding exercises help students better understand complex programming constructs [1, 14]. These approaches promote critical thinking and retention, particularly when students are actively engaged in constructing their own knowledge [15, 16].

Despite the growing adoption of active learning in STEM disciplines, the benefits in computing education often depend more on the instructional strategy than on the physical classroom environment [17, 18]. Tools such as classroom response systems and interactive IDEs have emerged to support engagement, but limitations remain in providing personalized and conceptual feedback to support deeper learning [19, 20, 21], which is difficult to scale as class sizes increase.

2.2. Automated Feedback Systems

Automated feedback systems have become integral to programming education, aiming to enhance student learning by providing timely and constructive responses to code submissions [22]. Research in this domain has predominantly concentrated on the automatic correction or repair of erroneous

programs, often neglecting the delivery of comprehensible natural language feedback [23, 24, 25, 26, 27]. Few systems have relied on rule-based automated feedback generation techniques that, while functional, can produce feedback that is challenging for students to interpret and apply effectively and most of the automated grading tools focus primarily on assessing the correctness of assignments in object-oriented languages, with limited emphasis on the clarity and educational value of the feedback provided [22].

The advances in Generative Artificial Intelligence (Gen-AI) and Large Language Models (LLMs) have introduced new possibilities for generating feedback in programming education. Recent studies have explored the deployment of LLM-based systems to produce feedback for student programs. These studies focused on the application of LLMs in providing feedback on syntax errors, bugs, or misconceptions in the programs [4, 5, 28, 7, 29]. Although LLMs can generate natural language feedback, recent studies have revealed that LLM-generated feedback lack explainability, are prone to incorrect suggestions and hallucinations, and suffer from reliability and precision issues of the feedback to ensure effective and adequate support to students [6, 7, 30].

3. INSIGHT Classroom Assistant System

To support scalable, reliable, and explainable feedback delivery, we developed INSIGHT, a web-based classroom assistant designed to facilitate programming instruction. While INSIGHT includes a broad set of features for real-time activity distribution, collaborative learning, and instructor monitoring, this paper focuses on its explainable feedback selection capabilities.

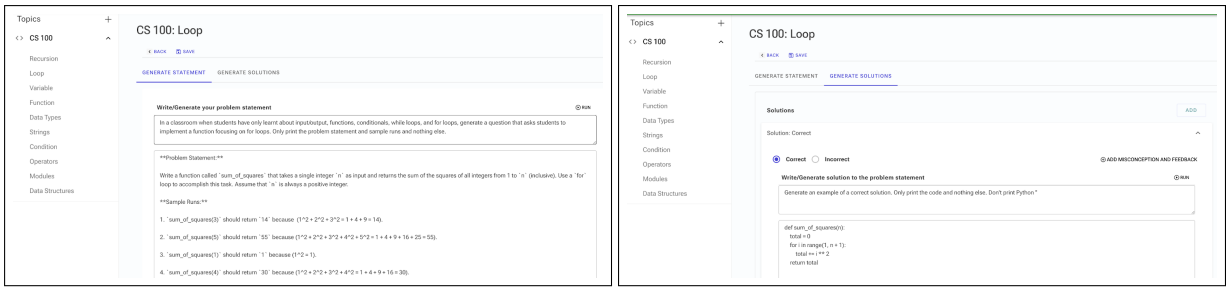
The system includes two primary interfaces: an Instructor app and a Student app. The Instructor app contains a dashboard and an authoring tool. The dashboard enables real-time distribution of exercises categorized by course topics, while the authoring tool allows instructors to create new exercises, provide correct and incorrect solution examples, and specify associated feedback. Instructors can optionally collaborate with an LLM to generate new problems and common student errors. The Student app, initially implemented as a Visual Studio Code extension, allows students to view assigned exercises, receive immediate feedback, and submit code within their development environment. All student-instructor interactions are synchronized to a cloud database for real-time analysis.

3.1. Instructor Authoring Tool

The Authoring tool enables instructors to create topic-tailored coding exercises that are specific to the course. All coding exercises are categorized by their course subject and topics in the tool, along with corresponding solutions and feedback [31]. The authoring tool allows instructors to add new coding exercises and provide sample solutions with common misconceptions about the exercises (Figure 1). The tool also allows instructors to leverage LLMs in generating coding exercises, solutions, and feedback. In this way, instructors can incorporate their pedagogical expertise with the extensive problem and solution spaces of LLMs to generate the most relevant exercises for students. All authored content is stored in and accessed from the cloud database, so that it can be shared with the dashboard.

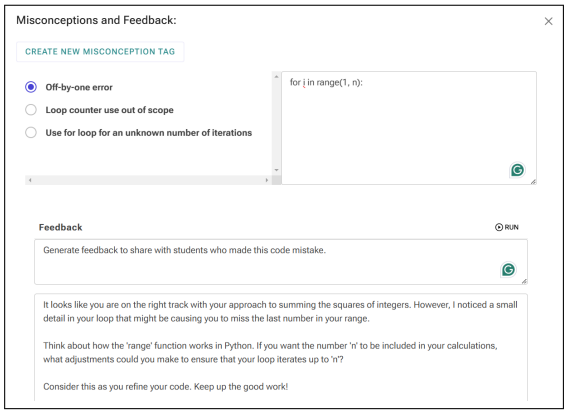
3.2. Student App

To allow students to receive the instructor-sent coding exercises and write their own solutions within the interface, we have developed the INSIGHT Student App. The student app enables students to access activities within the app, such as viewing coding exercises sent in the class, receiving immediate personalized feedback and hints to develop solution code, and submitting their solutions in real-time. The initial version of the INSIGHT student app was developed as a Visual Studio Code extension (Figure 2). By integrating with a development environment (IDE), the app provides seamless integration with the existing classroom workflow. We plan to support multiple IDEs as well as a standalone web-based interface.



(a) Problem design interface.

(b) Example solution creation interface.



(c) Misconception and feedback tagging interface.

Figure 1: The authoring tool of the INSIGHT classroom assistant system.

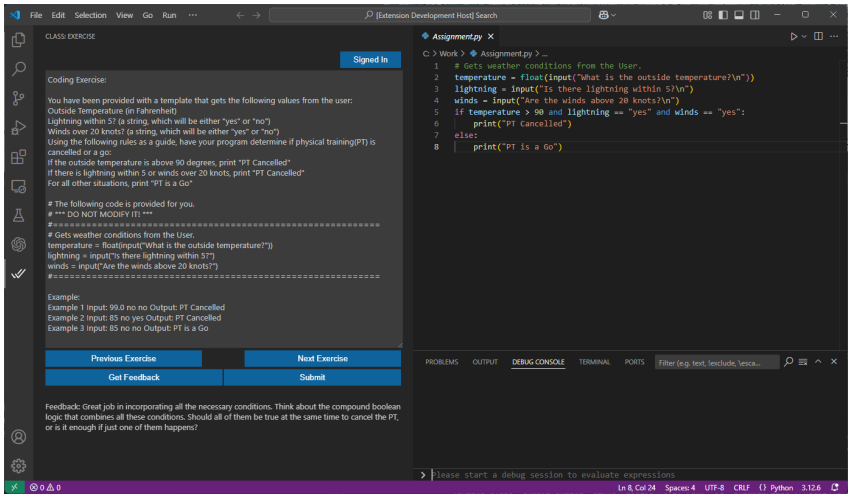


Figure 2: Student App as a VS Code Extension.

4. Dataset

For this work, we used the FalconCode dataset [13], a large and publicly available collection of 1.5 million Python programs submitted by over two thousand undergraduate students at the United States Air Force Academy over five semesters. The dataset encompasses code submissions for more than 800 Python programming assignments, along with metadata, including the problem prompts, test cases used for evaluation, and the specific skills required to solve each problem. The problems cover fundamental CS topics, such as conditionals, loops, files, functions, strings, lists, etc.

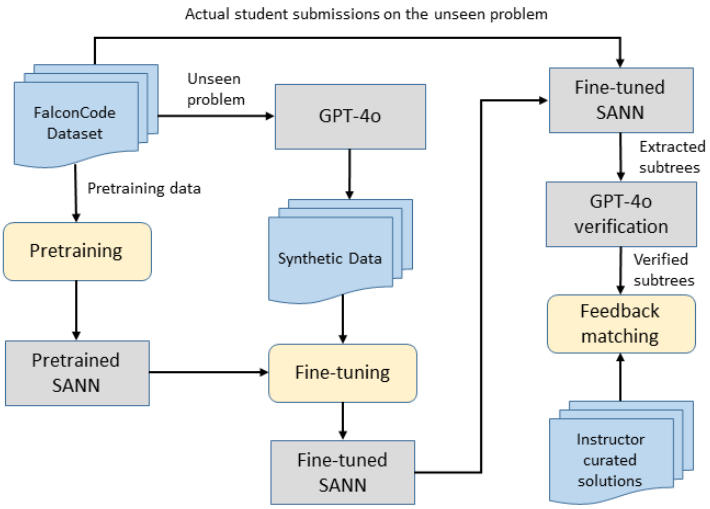


Figure 3: Feedback propagation framework.

5. Methodology

We propose a feedback propagation framework that combines an explainable AI model with generative AI to deliver accurate, reliable, and explainable feedback on student programming submissions. Our approach uses a pretrained and fine-tuned SANN model to identify the most influential subtrees in incorrect code, highlighting where feedback is needed. A secondary verification step using an LLM ensures that only genuinely erroneous code segments are flagged. These verified segments are then matched with instructor-defined examples to deliver precise, contextually relevant feedback to students. Figure 3 briefly illustrates the feedback propagation framework.

5.1. Explainable AI Model

Subtree-based Attention Neural Network (SANN) [9] is a model designed for explainable program representation learning, capable of encoding source code into compact vector forms by extracting meaningful substructures from Abstract Syntax Trees (ASTs). These vector representations have been effectively applied to various prediction tasks, such as code correctness predictions and algorithm detection [9, 32]. In this work, we leveraged SANN's attention mechanism to identify key subtrees that contribute most to the final classification task (whether a program is correct or not), enabling us to focus on the code parts where feedback is needed for an incorrect program.

SANN constructs vector representations of source code by first extracting subtrees from the AST and embedding them using a two-way embedding strategy (a combination of node-based and subtree-based embeddings) to generate subtree vectors that contain syntactic and semantic information of the program. These subtree vectors are then aggregated into a single vector representation of the entire program using an Attention Neural Network. The attention mechanism assigns scalar weights to each subtree vector, computed via a normalized inner product between the subtree vectors and a global attention vector, followed by a softmax function. This mechanism highlights the most influential subtrees, enhancing explainability by revealing which parts of a program contribute to the model's decision.

While the original SANN model employed a genetic search-based optimization technique to segment ASTs into non-overlapping subtrees of fixed sizes based on a specific task, we leveraged a modified version of SANN to consider all possible subtrees of varying sizes and incorporated a sigmoid attention activation function and an entropy regularization to effectively identify the subtrees with logical errors, as suggested by prior research [12]. This modification allows us to better capture and analyze erroneous patterns in student code by identifying the most significant subtrees influencing the model to predict the code as incorrect.

5.2. Model Pretraining

To simulate a real classroom scenario where students solve new problems without prior historical data for model training, we pretrained the SANN model using the FalconCode dataset. We filtered out problems that were ungraded or lacked correctness information, as the model was trained on a binary correctness classification task. To streamline the pretraining process and make it less resource-hungry, we randomly selected 50% of the available problems, resulting in a dataset of 234 problems and over 435,614 student submissions. The model was pretrained on a machine using a simplified configuration with 128GB RAM and an NVIDIA GeForce RTX 3060Ti GPU. The final pretrained model comprised approximately 100 million parameters, providing a robust initialization for downstream tasks.

5.3. Model Fine-tuning

To evaluate the model's ability to generalize to unseen problems, we selected a problem that was not included in the pretraining phase. This setup mirrors a real classroom scenario where students encounter new problems without prior historical data for model training. To fine-tune the pretrained SANN model on this new problem, we leveraged GPT-4o to generate synthetic student submissions. We generated 1,000 correct and 1,000 incorrect submissions using a structured prompt that instructed the model to mimic an introductory student programmer. The prompt also included instructions to cover a diverse solution space to ensure variations in correct implementations while incorporating common logical errors, conceptual misunderstandings, and edge case mishandling in incorrect solutions, along with some instructor-generated example solutions. After generating this synthetic dataset, we fine-tuned the pretrained model on the generated synthetic data to adapt to the new problem without using any historical submissions of real students, enabling it to identify errors effectively in student submissions.

5.4. Feedback Selection

In a real-time classroom setting, instructors can design new programming problems using our authoring tool, which integrates interactions with an LLM. Using the authoring tool, instructors can provide examples of both correct and incorrect solutions, highlighting erroneous segments in incorrect submissions along with corresponding feedback to guide students in similar problem-solving scenarios. Once a student submits a solution, our model, fine-tuned on synthetic data for the newly introduced problem, analyzes the submission to identify the most influential subtrees. To simulate this setting, we selected a problem from the FalconCode dataset in the fine-tuning phase that was not included in the pretraining data, as mentioned before. Using our authoring tool, a CS1 instructor created example solutions with corresponding feedback without referencing actual student data for the new problem. Using the tool's solution authoring interface, the instructor authored two correct solutions and a set of incorrect solutions representing common misconceptions. For each incorrect solution, the instructor annotated the associated erroneous code segments, specified the type of misconception, and provided corresponding feedback to guide students. While the instructor primarily crafted these examples manually, the tool also allowed optional interaction with an LLM to explore additional erroneous patterns and refine feedback suggestions.

The fine-tuned model then processed real student submissions on this problem, extracting the most significant subtrees. These extracted subtrees capture the errors in the incorrect submissions, providing insights into the feedback needed for students [12]. However, given the potential for false positives in the extracted subtrees, it is crucial to ensure that feedback is accurate and does not mislead students. To mitigate this risk, we incorporated a secondary verification layer using an LLM. In this work, GPT-4o was employed to assess whether an identified important subtree by the explainable model indeed contained an error. Only when both the explainable model and the LLM concurred on the presence of an error was the subtree considered for the feedback-matching stage. The fine-tuned model then analyzed instructor-defined incorrect solutions, and their corresponding subtrees were extracted along with associated feedback. To map the erroneous subtrees extracted from student submissions with

Table 1
Program correctness prediction performance on the unseen problem.

Model	Accuracy	Precision	Recall	F1-Score
Individual SANN (Synthetic Only)	0.83	0.81	0.76	0.78
Mixed Fine-tuned SANN (1% Pretraining Data + Synthetic)	0.84	0.83	0.77	0.80
Fine-tuned Pretrained SANN (Synthetic Only)	**0.89**	**0.89**	**0.84**	**0.86**

those in the instructor-provided examples, we computed the cosine similarity between their vector representations extracted from the explainable model. Feedback was assigned to the student's incorrect submission based on the most closely matched erroneous subtree, ensuring precise and contextually relevant guidance.

6. Evaluation

6.1. Pretraining the Modified SANN Model

To develop our generalizable model, we pretrained the modified SANN model on a large-scale program correctness prediction task. Specifically, we used $435,614$ student code submissions from 234 programming problems in the FalconCode dataset, consisting of $113,038$ correct and $322,576$ incorrect submissions. Given the inherent class imbalance, we performed a stratified 80:10:10 split for training, validation, and test sets to preserve the label distribution across all subsets.

Hyperparameters were tuned based on validation performance. We set the embedding dimension to 100, capped the number of AST nodes per subtree and the number of extracted subtrees per code to 100, and applied an early stopping criterion with a patience of 20 epochs within a maximum of 200 training epochs. To encourage sparse attention distributions and improve explainability and effective logical error identification, an entropy regularization term of 0.0003 was applied. Pretraining required approximately three hours to complete. The pretrained model achieved strong performance on the held-out test set, attaining an accuracy of 88%, precision of 88%, recall of 81%, and an F1-score of 83%. These results demonstrate the model's effectiveness in capturing syntactic and semantic patterns relevant to program correctness, providing a robust foundation for downstream tasks such as fine-tuning and feedback selection.

6.2. Program Correctness Prediction on Unseen Problem

Following the pretraining phase, we evaluated the ability of the pretrained SANN model to generalize to a previously unseen problem, simulating a real classroom setting in which an instructor introduces a new programming task without any historical student data. To enable this, we fine-tuned the pretrained model using $2,000$ synthetic code submissions generated by GPT-4o.

To assess the effectiveness of our fine-tuned model, we evaluated its performance on 548 actual student submissions from the unseen problem (correct: 168, incorrect: 380), which was excluded from the pretraining dataset. We compared our approach against two baselines: (i) an individual SANN model trained solely on the synthetic data, and (ii) a mixed fine-tuning strategy, in which the pretrained model was fine-tuned using the synthetic dataset augmented with a small portion (1%) of the pretraining data. This second baseline aligns with the concept of low-resource or few-shot domain adaptation, where limited in-domain examples are used alongside pretraining knowledge to improve generalization. As shown in Table 1, our fine-tuned model outperformed both baselines across all metrics: accuracy, precision, recall, and F1-score, demonstrating the advantage of using a pretrained model adapted specifically for unseen problems through synthetic, LLM-generated student data. These results suggest that our approach offers a viable path for building feedback systems in classroom settings where historical data is unavailable.

6.3. Feedback Selection Performance

To evaluate the effectiveness of our explainable feedback selection framework, we tested our pipeline on real student submissions for a programming problem excluded from the model's pretraining phase. The fine-tuned model was used to extract important subtrees from incorrect student submissions in the FalconCode dataset. We used an attention threshold of 15% to select the most influential subtrees, which we hypothesized to contain the erroneous logic responsible for incorrect program behavior. To ensure the reliability of these subtrees, we introduced a secondary verification step using GPT-4o. If the LLM disagreed with the model regarding the presence of an error in a subtree, that subtree was discarded to reduce the likelihood of false positives. For the remaining verified subtrees, we computed cosine similarity between their vector representations and those of instructor-annotated incorrect examples. If the maximum similarity score exceeded a threshold of 50%, the feedback associated with the closest instructor example was assigned to the student submission; otherwise, no feedback was returned.

We evaluated the quality of feedback matching based on a manual analysis conducted by one of the authors. The evaluation focused solely on the accuracy of feedback matching—i.e., whether the feedback corresponded appropriately to the actual student error, rather than the pedagogical quality of the feedback itself, as all feedback had been authored by the instructor. Before evaluation, we filtered out submissions with syntax or runtime errors, resulting in a final set of 169 incorrect student submissions. Quantitative analysis of the evaluation revealed that: 96 submissions (56.8%) received correctly matched feedback, 48 submissions (28.4%) were assigned incorrect feedback, and 25 submissions (14.8%) received no feedback due to low similarity or unmatched patterns.

Further analysis of the "no feedback" cases revealed that these submissions often deviated significantly from the problem specification—for example, by failing to use the provided boilerplate code for input handling, which made it difficult to align them with any instructor-authored examples. In practice, such cases would be flagged for instructor intervention. Among the "incorrect" matches, we observed that the student code structures bore little resemblance to any of the predefined error patterns provided by the instructor, resulting in incorrect matches. These errors highlight the need for richer example coverage during the authoring process or future incorporation of approximate structural matching techniques. Finally, an important limitation we observed within the "correct" feedback group was that, for submissions containing multiple errors, the matched feedback typically addressed only a single issue. While this granularity could support step-by-step debugging and reduce cognitive overload, it may also frustrate students who prefer comprehensive feedback. Addressing multi-error submissions and studying how sequential feedback impacts learning remains an important direction for future work [33].

7. Discussion

In this work, we addressed a critical challenge in CS education: delivering scalable, trustworthy, and pedagogically sound feedback to students in real-time classroom settings to facilitate active learning. As class sizes grow and the need for individualized support increases [34, 35], traditional feedback mechanisms become difficult to scale [3]. While LLMs have shown promise in code analysis and feedback generation, concerns around hallucination, explainability, and reliability hinder their direct deployment in educational environments [6]. Our work proposes a hybrid approach that integrates explainable deep learning models with LLMs and instructor-authored examples to ensure feedback remains accurate, contextually grounded, and pedagogically aligned.

Our approach combines a modified version of the explainable SANN model with LLM-based verification and instructor input. The SANN model is pretrained on a large-scale Python dataset consisting of student code submissions to learn meaningful program representations. When instructors design a new programming problem using the authoring tool of our INSIGHT classroom assistant system, we generate a synthetic dataset of student submissions with correct and incorrect variants, aided by LLMs. The model is then fine-tuned on this synthetic dataset to adapt to the new problem without access to historical student submissions. Once students begin submitting code, our system identifies important

subtrees using SANN's attention mechanism, verifies the presence of errors using GPT-4o, and matches verified erroneous patterns to instructor-authored feedback based on vector similarity.

Our evaluation demonstrated that this framework showed promising performance even when applied to an unseen programming problem. The fine-tuned model achieved high performance on actual student submissions for correctness prediction, outperforming models trained solely on synthetic data or using limited pretraining augmentation. In the feedback selection phase, more than half of the incorrect submissions received correct feedback, despite the challenges of aligning student logic with a sparse set of instructor-authored examples. These results highlight the promise of combining explainable program representations, LLM verification, and instructor domain knowledge to scale feedback generation in CS classrooms. Beyond performance metrics, this system presents an important opportunity for enhancing programming instruction. Instructors can use the authoring tool to scaffold common misconceptions, design problem-specific feedback, and collaborate with LLMs to co-generate solution spaces. The triangulated feedback mechanism ensures that only reliable, interpretable, and context-aware feedback reaches students, making it well-suited for real-time learning environments. This framework also empowers instructors by reducing the burden of individually assessing each student's work and enhancing transparency in automated decision-making, a key requirement in educational AI systems.

8. Limitations and Future Work

Our work has several limitations and important directions for future work. First, the effectiveness of the feedback matching mechanism is strongly dependent on the quality and coverage of instructor-authored incorrect solutions. Sparse or overly specific examples may limit the system's ability to match unseen student errors. In future iterations, we aim to automate the generation of diverse incorrect examples using LLMs under instructor supervision, thereby enriching the feedback space while retaining instructional control. Second, our current feedback-matching mechanism relies primarily on cosine similarity between subtree vectors. While effective in many cases, this approach can propagate incorrect feedback when structurally dissimilar errors yield incorrect matches due to scarce instructor examples. To mitigate this, we plan to incorporate LLM-based feedback validation as an additional layer of confirmation before any feedback is shown to the student, thereby reducing false positives in the feedback propagation pipeline. Third, our current framework struggles to handle submissions containing multiple errors. In these cases, the feedback typically addresses only one issue, which may not be sufficient for students aiming to resolve all problems at once. While sequential, step-by-step, and immediate feedback could support cognitive load management [33], we intend to explore multi-error detection and composite feedback generation in future work. This also includes studying whether iteratively addressing errors, one at a time, can enhance learning outcomes compared to presenting all issues simultaneously. Another promising future direction is incorporating LLMs in the correctness prediction stage itself. If both the fine-tuned model and the LLM agree on the correctness label of a submission, we can increase confidence in the decision and avoid propagating feedback for correct solutions, eliminating the need for instructors to manually author test cases for correctness validation. Additionally, this ensemble-style agreement between models could improve robustness and reduce reliance on any single component. Lastly, our evaluation focused on technical correctness rather than pedagogical effectiveness. While the system can match errors to instructor feedback, future work will involve classroom studies measuring the actual learning gains and usability outcomes associated with receiving AI-assisted feedback. This will ensure that the system is not only technically reliable but also pedagogically impactful.

9. Conclusion

This work presented a novel framework that combines explainable AI models, large language models, and instructor expertise to deliver scalable, explainable, and reliable feedback on student programming

submissions. By fine-tuning a pretrained model on synthetic data for unseen problems and integrating LLM-based validation, our system enables accurate feedback propagation without relying on historical student data. Our evaluation demonstrated the effectiveness of our approach in predicting program correctness and initiating feedback matching, even in classroom settings with newly designed exercises. While limitations remain, this work lays the groundwork for scalable, reliable, and trustworthy AI-assisted feedback systems in computing education.

Acknowledgments

This research was supported by the National Science Foundation (NSF) under Grants DUE-2236195 and DUE-2331965. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT and Grammarly to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] J. J. McConnell, Active learning and its use in computer science, in: Proceedings of the 1st Conference on Integrating Technology into Computer Science Education, 1996, pp. 52–54.

[2] P. Denny, S. MacNeil, J. Savelka, L. Porter, A. Luxton-Reilly, Desirable characteristics for ai teaching assistants in programming education, in: Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, 2024, pp. 408–414.

[3] X. Tang, S. Wong, M. Huynh, Z. He, Y. Yang, Y. Chen, Sphere: Scaling personalized feedback in programming classrooms with structured review of llm outputs, arXiv preprint arXiv:2410.16513 (2024).

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

[5] Q. Jia, J. Cui, H. Du, P. Rashid, R. Xi, R. Li, E. Gehringer, Llm-generated feedback in real classes and beyond: Perspectives from students and instructors, in: Proceedings of the 17th International Conference on Educational Data Mining, International Educational Data Mining Society, 2024, pp. 862–867.

[6] T. Phung, V.-A. Pădurean, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, G. Soares, Generative ai for programming education: Benchmarking chatgpt, gpt-4, and human tutors, in: Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 2, 2023, pp. 41–42.

[7] S. Jacobs, S. Jaschke, Evaluating the application of large language models to generate feedback in programming education, arXiv preprint arXiv:2403.09744 (2024).

[8] C. Chen, G. Sonnert, P. M. Sadler, D. Sasselov, C. Fredericks, The impact of student misconceptions on student persistence in a mooc, Journal of Research in Science Teaching 57 (2020) 879–910.

[9] M. Hoq, S. R. Chilla, M. Ahmadi Ranjbar, P. Brusilovsky, B. Akram, Sann: Programming code representation using attention neural network with optimized subtree extraction, in: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM), 2023, pp. 783–792.

[10] M. Hoq, P. Brusilovsky, B. Akram, Explaining explainability: Early performance prediction with student programming pattern profiling, Journal of Educational Data Mining 16 (2024) 115–148.

[11] M. Hoq, J. Vandenberg, B. Mott, J. Lester, N. Norouzi, B. Akram, Towards attention-based automatic misconception identification in introductory programming courses, in: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2, 2024, pp. 1680–1681.
[12] M. Hoq, A. Rao, R. Jaishankar, K. Piryani, N. Janapati, J. Vandenberg, B. Mott, N. Norouzi, J. Lester, B. Akram, Automated identification of logical errors in programs: Advancing scalable analysis of student misconceptions, in: Proceedings of the 18th International Conference on Educational Data Mining (EDM), International Educational Data Mining Society, Palermo, Italy, 2025, pp. –.
[13] A. de Freitas, J. Coffman, M. de Freitas, J. Wilson, T. Weingart, 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, 2023, pp. 938–944.
[14] D. Baldwin, Discovery learning in computer science, in: Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, 1996, pp. 222–226.
[15] J. E. Froyd, Evidence for the efficacy of student-active learning pedagogies, Project Kaleidoscope 66 (2007) 64–74.
[16] J. Pirker, M. Riffnaller-Schiefer, C. Gütl, Motivational active learning: engaging university students in computer science education, in: Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education, 2014, pp. 297–302.
[17] T. Greer, Q. Hao, M. Jing, B. Barnes, On the effects of active learning environments in computing education, in: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, 2019, pp. 267–272.
[18] Q. Hao, B. Barnes, M. Jing, Quantifying the effects of active learning environments: separating physical learning classrooms from pedagogical approaches, Learning Environments Research 24 (2021) 109–122.
[19] M. T. Chi, R. Wylie, The icap framework: Linking cognitive engagement to active learning outcomes, Educational Psychologist 49 (2014) 219–243.
[20] D. Lombardi, T. F. Shipley, A. Team, B. Team, C. Team, E. Team, G. Team, G. Team, P. Team, The curious construct of active learning, Psychological Science in the Public Interest 22 (2021) 8–43.
[21] M. Ebert, M. Ring, A presentation framework for programming in programing lectures, in: Proceedings of the 2016 IEEE Global Engineering Education Conference (EDUCON), IEEE, 2016, pp. 369–374.
[22] M. Messer, N. C. Brown, M. Kölling, M. Shi, Automated grading and feedback tools for programming education: A systematic review, ACM Transactions on Computing Education 24 (2024) 1–43.
[23] R. Singh, S. Gulwani, A. Solar-Lezama, Automated feedback generation for introductory programming assignments, in: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, 2013, pp. 15–26.
[24] S. Gulwani, I. Radiček, F. Zuleger, Automated clustering and program repair for introductory programming assignments, ACM SIGPLAN Notices 53 (2018) 465–480.
[25] S. Bhatia, P. Kohli, R. Singh, Neuro-symbolic program corrector for introductory programming assignments, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 60–70.
[26] R. Gupta, A. Kanade, S. Shevade, Deep reinforcement learning for syntactic error repair in student programs, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, 2019, pp. 930–937.
[27] J. Zhang, J. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, G. Verbruggen, Repairing bugs in python assignments using large language models, arXiv preprint arXiv:2209.14876 (2022).
[28] T. Phung, V.-A. Pădurean, A. Singh, C. Brooks, J. Cambronero, S. Gulwani, A. Singla, G. Soares, Automating human tutor-style programming feedback: Leveraging gpt-4 tutor model for hint generation and gpt-3.5 student model for hint validation, in: Proceedings of the 14th Learning Analytics and Knowledge Conference, 2024, pp. 12–23.
[29] Z. Xu, S. Jain, M. Kankanhalli, Hallucination is inevitable: An innate limitation of large language models, arXiv preprint arXiv:2401.11817 (2024).
[30] Q. Jia, J. Cui, R. Xi, C. Liu, P. Rashid, R. Li, E. Gehringer, On assessing the faithfulness of llm-

generated feedback on student assignments, in: Proceedings of the 17th International Conference on Educational Data Mining, International Educational Data Mining Society, 2024, pp. 491–499.

[31] M. Hoq, J. Vandenberg, S. Jiao, S. Lee, B. Mott, N. Norouzi, J. Lester, B. Akram, Facilitating instructors-llm collaboration for problem design in introductory programming classrooms, in: Proceedings of the CHI 2025 Workshop on Augmented Educators and AI: Shaping the Future of Human and AI Cooperation in Learning, 2025.

[32] M. Hoq, Y. Shi, J. Leinonen, D. Babalola, C. Lynch, T. Price, B. Akram, Detecting chatgpt-generated code submissions in a cs1 course using machine learning models, in: Proceedings of the 55th ACM Technical Symposium on Computer Science Education, Association for Computing Machinery, New York, NY, USA, 2024, p. 526–532.

[33] J. R. Anderson, F. G. Conrad, A. T. Corbett, Skill acquisition and the lisp tutor, Cognitive Science 13 (1989) 467–505.

[34] M. Hoq, P. Brusilovsky, B. Akram, Analysis of an explainable student performance prediction model in an introductory programming course, in: Proceedings of the 16th International Conference on Educational Data Mining, International Educational Data Mining Society, Bengaluru, India, 2023, pp. 79–90.

[35] M. Hoq, A. Patil, K. Akhuseyinoglu, P. Brusilovsky, B. Akram, An automated approach to recommending relevant worked examples for programming problems, in: Proceedings of the 56th ACM Technical Symposium on Computer Science Education (SIGCSE) V. 1, Association for Computing Machinery, New York, NY, USA, 2025, pp. 527–533.