Automated Identification of Logical Errors in Programs: Advancing Scalable Analysis of Student Misconceptions

Muntasir Hoq
NC State University
mhoq@ncsu.edu

Ananya Rao
NC State University
arrao3@ncsu.edu

Reisha Jaishankar
NC State University
rjaisha@ncsu.edu

Krish Piryani
NC State University
kopiryan@ncsu.edu

Nithya Janapati
NC State University
nsjanapa@ncsu.edu

Jessica Vandenberg
NC State University
jvanden2@ncsu.edu

Bradford Mott
NC State University
bwmott@ncsu.edu

Narges Norouzi
UC Berkeley
norouzi@berkeley.edu

James Lester
NC State University
lester@ncsu.edu

Bita Akram
NC State University
bakram@ncsu.edu

ABSTRACT

In Computer Science (CS) education, understanding factors contributing to students' programming difficulties is crucial for effective learning support. By identifying specific issues students face, educators can provide targeted assistance to help them overcome obstacles and improve learning outcomes. While identifying sources of struggle, such as misconceptions, in real-time can be challenging in current educational practices, analyzing logical errors in students' code can offer valuable insights. This paper presents a scalable framework for automatically detecting logical errors in students' programming solutions. Our framework is based on an explainable Abstract Syntax Tree (AST) embedding model, the Subtree-based Attention Neural Network (SANN), that identifies the structural components of programs containing logical errors. We conducted a series of experiments to evaluate its effectiveness, and the results suggest that our framework can accurately capture students' logical errors and, more importantly, provide us with deeper insights into their learning processes, offering a valuable tool for enhancing programming education.

Keywords

computer science education, student modeling, logical error identification, programming misconceptions

1. INTRODUCTION

To effectively support students in mastering introductory programming, timely identification and resolution of factors contributing to students' programming challenges, especially difficulties with fundamental concepts, are imperative [48]. However, the increased number of CS majors has created significant challenges in supporting students with limited teaching resources [31]. As such, instructors' real-time monitoring of students' learning progress is often infeasible [47, 56]. Addressing this challenge requires an automated approach to efficiently identifying students' programming struggles, enabling personalized feedback delivery through either teachers or automated systems. However, automated identification of students' programming struggles is complex due to the latent nature of learning difficulties, which cannot be directly assessed. Nevertheless, these struggles can manifest as logical errors in student code [1, 19]. Computing education research has identified several potential underlying causes for logical errors, including strategic or algorithmic errors, misinterpretation of the problem description, carelessness, and foundational misconceptions [19, 55] with misconceptions noted as the most common source of logical errors [19]. Thus, analyzing logical error patterns over time can predict students' learning outcomes. More importantly, this analysis can shed light on potential misconceptions held by students, enabling scalable approaches for remediating through adaptive scaffolding [19].

Previous research has explored semi-automated methods for identifying errors and misconceptions using association rules in constrained domains [18, 23, 10]; however, these approaches have proven inadequate for complex, general-purpose tasks, such as programming [62]. More recent efforts have applied Deep Learning (DL) models for automatic logical error identification. However, they require manual data labeling via expert-designed rubrics and test cases and focus only on conceptual errors associated with specific rubric items [62]. These approaches may not detect logical errors spanning multiple rubric items or be applicable in scenarios where rubrics or test cases are not readily available. Recent studies [21, 42] have explored using Large Language Models (LLMs) to identify logical errors and misconceptions in stu-

Muntasir Hoq, Ananya Rao, Reisha Jaishankar, Krish Piryani, Nithya Janapati, Jessica Vandenberg, Bradford Mott, Narges Norouzi, James Lester, and Bita Akram. Automated Identification of Logical Errors in Programs: Advancing Scalable Analysis of Student Misconceptions. In Caitlin Mills, Giora Alexandron, Davide Taibi, Giosuè Lo Bosco, and Luc Paquette (eds.) Proceedings of the 18th International Conference on Educational Data Mining, Palermo, Italy, July, 2025, pp. 90–103. International Educational Data Mining Society (2025).

https://doi.org/10.5281/zenodo.15870203

dent programs, but require expert supervision [35, 67]. Regardless, DL-based approaches show promise in automating logical error detection, providing timely feedback to teachers [16] and students [57], even in large classrooms.

This paper proposes an automated framework for identifying students' logical errors within their programs using a DL framework. In this work, we utilized a publicly available dataset collected from an introductory programming course at a university in the United States [17]. Although the programs in this dataset were automatically labeled as correct or incorrect by executing the programs using test cases, we do not utilize these test cases or any rubric to pinpoint logical errors in incorrect student code. Instead, our framework utilizes a modified version of the state-of-the-art explainable DL model, Subtree-based Attention Neural Network (SANN) [26], which captures students' programming patterns by embedding program vectors based solely on program correctness information. We modify the attention layer of SANN, which highlights important substructures within program snippets, aiding in classifying code as correct or incorrect. Evaluation results indicate that these highlighted patterns closely correlate with logical errors in student code, with our method effectively identifying logical errors in students' incorrect programs. We further engage expert annotators to categorize identified logical errors as one of the two essential categories identified in the literature: *algorithmic error* (strategic) and *fundamental conceptual error* (misconceptions) [19, 55], and a third category, *compilable syntactic error* that we encountered during our analysis (details in Section 2.1), to better understand the relationship between logical errors and underlying students' struggles. Literature has also referred to *misinterpretation* and *carelessness* as other possible underlying causes for logical errors. However, distinguishing between issues arising from misinterpretation or carelessness rather than conceptual misunderstanding or strategic errors requires internal knowledge about a student's state of mind [1]. Therefore, we excluded *misinterpretation* and *carelessness* as possible underlying causes for logical errors in this paper.

Misconceptions have been identified as the most common source of logical errors [19]. Simultaneously, research has shown that incorporating misconception information can significantly enhance student modeling results [45]. Thus, we hypothesize that enriching code vectors with logical error information can shed light on students' learning progress. To verify this hypothesis, we embed the logical error information in program vectors and utilize them in Long Short-Term Memory-based Deep Knowledge Tracing (LSTM-DKT) models to predict students' short-term (next problem attempt results) and long-term (final exam grades) performance. Our results demonstrate improved predictive student modeling and potential for advancing the modeling of student progress. This framework can also assist in dynamically finding traces of logical errors and potential misconceptions in students' programming, supporting the investigation of their relationship with students' learning outcomes.

The main contributions of this work are as follows:

- Developing a novel, explainable deep learning framework that identifies logical errors in student code without relying on external rubrics or test cases.
- Validating the framework through expert annotation, categorizing them into syntactic, strategic, and conceptual errors to rigorously assess its effectiveness.
- Demonstrating improved predictive student modeling by incorporating logical error information into code embeddings.

We further discuss the framework's potential to enhance students' learning by enabling scalable, dynamic, and adaptive scaffolding for students while learning to program.

2. RELATED WORK

In this section, we discuss related work on logical errors in student code within computer science education, methods for identifying students' logical errors, and predictive student modeling for predicting student learning.

2.1 Logical Errors in CS Education

Various categorizations of programming errors have been proposed in the literature [40, 1], with common classifications categorizing errors into syntactic, semantic (logical), and runtime errors [32]. Logical errors are prevalent in students' programs and take the most time to identify and resolve due to a lack of direct external feedback, such as ones provided by compilers for syntactic errors [20, 3, 4]. Nevertheless, identifying logical errors can provide invaluable insights into students' struggles with learning to program, including their misconceptions about programming concepts [19].

Prior research in CS education has identified various sources for logical errors, including *conceptual*, *strategic*, *misinterpretation*, and *carelessness* [1, 19].

- ***Conceptual errors***: A conceptual error is a logical error involving a misunderstanding of how a programming construct works, reflecting a fundamental flaw in programming knowledge. An example of a conceptual error is when a student assumes that a variable retains its previous value after being reassigned.
- ***Strategic errors***: A strategic error occurs when a student's underlying algorithm or problem-solving approach is flawed, even if their understanding of programming constructs is sound. For example, failing to correctly handle the base case in recursion.
- ***Misinterpretation errors***: A misinterpretation error occurs when a student interprets a programming task or question differently than intended, leading to discrepancies between the expected and actual program output.
- ***Carelessness errors***: A carelessness error is an unintentional mistake resulting from oversight or inattentiveness rather than a deeper misunderstanding. Common examples include missing punctuation (e.g., semicolons) or typographical errors (e.g., using = instead of ==) that do not stem from conceptual issues.

In this paper, we focus on two categories of logical errors from the literature: *conceptual* and *strategic* errors, since the

other categories (*misinterpretation* and *carelessness*) cannot be readily inferred from student code due to their inherently latent nature and do not necessarily indicate a lack of programming knowledge [1]. Although we did not include identifying programming language-specific syntactic errors from uncompilable student submissions in this work that can be easily detected at compile time [19], we encountered a third category identified in a small percentage of student programs during our analysis, which we classify as *compilable syntactic* errors. For example, using `&` instead of `&&` in a conditional statement, which some compilers (e.g., Java) take as a bitwise operation and compile the code. This type of error can be language-specific and even stem from either *carelessness* or *conceptual* errors.

2.2 Logical Error Detection

Misconceptions in CS education refer to incorrect interpretations of particular concepts, leading to logical errors in programming [65, 19]. Despite the prevalence of logical errors among novice learners, these misunderstandings often evade detection by experts and instructors [64], a phenomenon known as the "expert blind spot" [51]. Identifying and comprehending these logical errors increases instructors' awareness and provides essential material for modeling students' learning over time to improve learning outcomes [44, 43]. Although previous research has explored automated methods for detecting student logical errors [23, 19, 62], many of these approaches require expert labeling of training data, presupposing familiarity with the specific logical errors being targeted.

In CS education, identifying logical errors and misconceptions traditionally relies on labor-intensive manual analysis of students' programs [52] or interviews [37]. The manual construction of "bug libraries" can also pose challenges for experts due to its time-consuming nature [6, 15, 5]. Nonetheless, such manual analyses, particularly those conducted on a series of program snapshots, have been shown to offer deeper insights into students' misconceptions. For instance, Albrecht and Grabowski [1] manually inspected students' programming data and classified errors into six categories: *syntactic*, *strategic*, *conceptual*, *carelessness*, *misinterpretation*, and *domain* knowledge.

Other research has explored data-driven approaches to uncovering logical errors. For example, Guzman et al. [23] used association rule mining to identify misconceptions from closed-ended multiple-choice questions. However, this method relies on experts to label rules and does not target open-ended programming problems. Ettles et al. [19] grouped common logical errors from students' incorrect programs using expert-designed test cases. A recent study [62] employed DL models for automatic misconception identification. Nonetheless, their methodology necessitated manual data labeling using an expert-designed rubric. Additionally, their approach targeted the detection of conceptual logic errors on a per-rubric-item basis within student code, lacking effectiveness in addressing logical errors that may extend across multiple rubric items or arise in scenarios where rubric items and test cases are absent. In some recent studies, LLMs were used to identify logical errors and misconceptions in students' programs [21, 42]. However, using LLMs without expert supervision and explainability might suffer from hallucination, precision, and stakeholder safety [35, 67].

2.3 Student Modeling

Student modeling is crucial in understanding and tracking students' mastery of skills for specific tasks within the learning paradigm [38]. Additionally, predicting students' performance and success is pivotal in adaptive learning environments designed to provide adaptive scaffolding [24]. Prior studies have focused on various objectives of performance prediction: short-term performance, such as next problem success prediction by knowledge tracing [60], and long-term performance, such as final exam grade prediction from students' programming submissions [68]. Research indicates that incorporating misconceptions can enhance student models [45], particularly given that misconceptions are a common source of logical errors [19]. Thus, code vectors enriched with such logical error information may offer deeper insights into students' learning trajectories and outcomes.

Knowledge tracing (KT) models student knowledge dynamically as students engage with problems, enabling predictions about future performance in students' problem-solving attempts. In KT, problems are typically associated with required skills, and the relationship between problems and skills can either be learned from data or derived directly from problem IDs [60]. KT models can predict students' success in future attempts by keeping track of the mastery of the required skills. Bayesian Knowledge Tracing (BKT) [14] is one of the most widely used KT models, and it has undergone several enhancements over the years [7, 69]. A significant advance was introduced by Piech et al. [54], who proposed Deep Knowledge Tracing (DKT). This model utilizes recurrent neural networks (RNNs) to predict a student's knowledge of each skill after each problem attempt. Despite these advancements, existing KT models have limitations when applied to programming data, particularly when incorporating code-related information. Recent iterations of DKT have attempted to address this gap by integrating code information [60, 43, 36, 59, 66]. However, these approaches often require experts to manually label errors/concepts, which can be resource-intensive. Furthermore, they often lack the interpretability to explain the failure prediction by mapping it back to the logical errors of students.

To track students' long-term learning progression and learning outcomes, researchers have predicted a range of outcomes, such as predicting student performance in final exams [24], detecting failing students [34], and identifying early dropouts [22]. A comprehensive review conducted by Shahiri et al. [58] demonstrated the reliance of many studies on grading-based features such as cumulative grade point averages (CGPA) and intermediate assessment scores (e.g., quizzes, midterms) for predicting course performance. In another study by Jamjoom et al., [34], decision tree and SVM were utilized to categorize enrolled students into passing and failing categories based on features such as quiz and midterm exam scores, with the aim of early intervention for at-risk students. A recent study [25] utilized student programming submission data to predict students' final exam grades in an explainable manner, aiming to explain the relationship between students' submission data and final exam grades. However, none of these studies incorporated direct programming code information into the prediction task

to integrate students' programming knowledge information for predicting students' performance. In recent work, DL frameworks have become increasingly prominent. In some instances, AST-based and control flow graph-based embedding models were employed to predict students' final exam grades from programming assignment data [68, 46].

Despite various efforts to detect logical errors in student code, most prior approaches rely heavily on expert-generated test cases and rubrics or lack explainability and face reliability concerns. In contrast, our work introduces a framework for automated, explainable, and scalable identification of logical errors from students' programs without the need for external rubrics or test cases. We further show that these program snippets embedded in code vectors, many of which represent misconceptions in students' programming knowledge, can be used in different predictive modeling tasks. We propose utilizing these program snippets directly to inform the provision of adaptive scaffolding based on vector similarity measurements as a viable solution to address potential misconceptions in students' programming knowledge and skills.

3. DATASET

We use a publicly available dataset[1] that was sourced from the CodeWorkout platform[2] [17] used in an introductory programming course at a public university in the United States. This dataset consists of anonymized student programming solutions in Java for 50 programming problems from five assignments used in a CS1 course in the Spring 2019 semester, without any demographic information. The submissions were evaluated on a $[0, 1]$ scale based on the number of passed test cases. We group the data into binary classes of correct and incorrect (if any of the test cases fails) solutions for model training. The dataset also consists of students' final exam grades normalized on a 0 to 1 scale. Throughout the semester, $57,670$ student code submissions were received from 368 students for 50 programming problems, comprising $18,787$ correct submissions and $38,883$ incorrect submissions. In this study, we leverage the Abstract Syntax Trees (ASTs) of programs to capture programming patterns. Consequently, consistent with prior research [61, 62], $9,906$ incorrect uncompilable solutions that could not be parsed into ASTs using our Java parser (javalang[3]) were excluded from the dataset.

4. METHODOLOGY

This section describes our adaptations to SANN for effectively identifying students' logical errors and the evaluation process we utilized. We then describe our processes for predicting students' short-term (next problem attempt) and long-term (final exam grade) success.

4.1 Extracting Logical Errors with SANN

SANN [26] is known for its explainability and ability to encode programs into condensed vectors by extracting substructures from ASTs. These vectors are versatile and have been utilized in different prediction tasks in the literature (i.e., program correctness prediction, algorithm detection, and LLM-generated code detection) [26, 28]. To identify logical errors in students' programs, we utilize SANN's attention mechanism to identify the subtrees extracted from program ASTs that are most influential in the prediction. Subtrees are embedded using a two-way embedding approach in SANN, where each subtree and its nodes are separately embedded. SANN merges the embeddings from the two-way embedding approach into a single embedded vector. Subsequently, the embedded vectors from both approaches are concatenated and passed through a time-distributed, fully connected layer to generate subtree vectors. Following this, an Attention Neural Network condenses all subtree vectors into a single source code vector. The attention mechanism assigns scalar weights to each subtree vector, supporting the aggregation of all subtree vectors into a weighted average. These weights are determined through a normalized inner product between each subtree vector and a global attention vector, followed by a softmax function.

The original SANN model focuses primarily on only one subtree with the most attention. Typically, this subtree includes an incorrectly implemented part (an error) of the program while predicting it as incorrect. However, we want to identify all logical errors present rather than only one error per incorrect student submission. Figure 1 illustrates the SANN model architecture. The original version of SANN uses an optimization approach to identify the best strategy for chunking an AST into its substructures for a prediction task. We use a modified version of SANN, where all possible substructures are included in the embedding and training process [29]. To do this, we extract all subtrees recursively from an AST, as illustrated in Figure 2. This enables us to capture logical errors of different granularities. Logical errors can vary in size, and a single error might involve interconnected parts of code working together. Some errors might even form a hierarchical structure, with a larger error encompassing a smaller one.

In addition to modifying the SANN subtree extraction process, the attention mechanism has been enhanced to improve its ability to focus on relevant subtrees. In the original version, attention weights were calculated using a softmax function to understand the importance of different subtrees within the code. However, this approach could inadvertently assign disproportionate attention to a single logical error, thereby increasing the risk of failing to identify multiple logical errors that may simultaneously contribute to the incorrectness of the code. In the modified version, we replaced the softmax function with a sigmoid activation function to compute the attention weights. Unlike softmax, sigmoid activation allows each attention weight to be computed independently without normalizing across all subtrees. This enables the model to assign high attention weights to multiple relevant subtrees and mitigates the risk of assigning excessive attention to a single logical error when multiple logical errors exist in incorrect code. However, based on our initial experimental results, this modification can result in higher false positive logical error predictions. To further refine the focus of the attention mechanism, we introduced an entropy regularization term to the model's loss function to selectively assign attention to fewer subtrees, reducing false positive predictions [53].

[1] https://pslcdatashop.web.cmu.edu/Files?datasetId=3458
[2] https://codeworkout.cs.vt.edu/
[3] https://github.com/c2nes/javalang

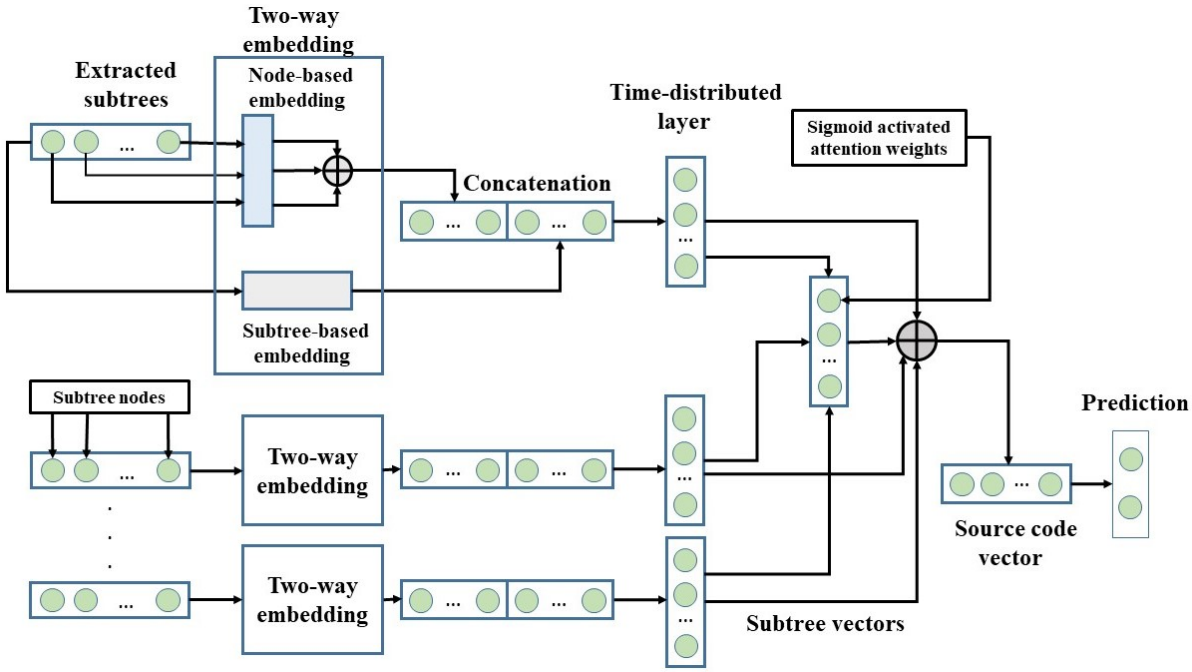

Figure 1: Model architecture of SANN.

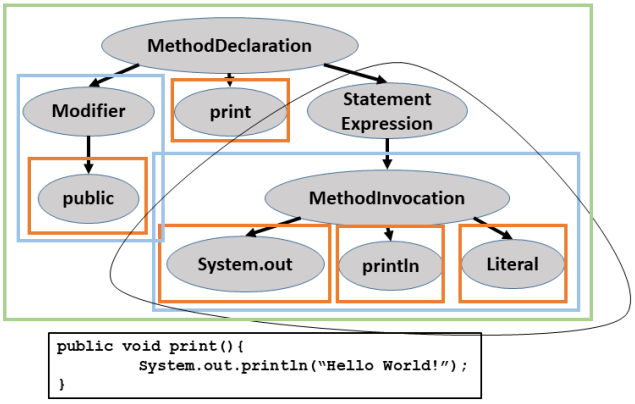

Figure 2: Subtree extraction from an AST.

For each subtree vector sv_i, the attention weight a_i is computed as:

$$a_i = \sigma(sv_i^T \cdot av)$$

where σ is the sigmoid activation function, and av is a global attention vector learned by the model. The entropy regularization term is computed as:

$$H(a) = -\lambda \sum_{i=1}^{n} a_i \log(a_i + \epsilon)$$

This entropy regularization loss is added to the overall loss function, along with the prediction loss. Here, λ is a regularization weight, ϵ is a small constant added for numerical stability, and the negative sign ensures entropy is a positive value. By penalizing high entropy, this term encourages the model to avoid distributing attention uniformly across all subtrees, effectively guiding it to focus selectively on a smaller subset of subtrees, thereby improving its ability to identify the logical errors in incorrect code predictions.

This combined approach allows our modified attention mechanism to balance the focus on multiple logical errors, if present, while avoiding assigning unnecessary attention to subtrees that do not contribute meaningfully to the prediction of an incorrect code. We hypothesize that the most important subtrees in an incorrect student code will likely encapsulate the logical errors therein. Leveraging our modified attention mechanism of the SANN model and extracting these important subtrees, we attempt to identify logical errors within an incorrect submission solely based on the program's correctness without requiring additional information, such as rubrics or test cases utilized in prior studies.

4.2 Evaluation of Logical Error Identification

To evaluate the effectiveness of our framework in detecting logical errors, two experts labeled all the incorrect submissions from one problem called `caughtSpeeding` (1,574 submissions: 617 correct, 957 incorrect) with their logical errors. Figure 3 depicts an example of an incorrect student solution to the `caughtSpeeding` problem with three errors highlighted in orange boxes and their corresponding corrected versions in white boxes. Later, we extended the evaluation to 4 other problems: `redTicket`, `countCode`, `sum13`, and `canBalance` and selected a subset of incorrect submissions (200 incorrect solutions each) to demonstrate the framework's robustness.

For the first problem, each expert independently labeled 10% of the dataset, identifying all existing logical errors in incorrect student submissions. Notably, if the student code con-

Write a function in Java that implements the following logic: You are driving a little too fast, and a police officer stops you. Write code to compute the result, encoded as an int value: 0=no ticket, 1=small ticket, or 2=big ticket. If speed is 60 or less, the result is 0. If speed is between 61 and 80 inclusive, the result is 1. If speed is 81 or more, the result is 2. Unless it is your birthday--on that day, your speed can be 5 higher in all cases.

```
public int caughtSpeeding(int speed, boolean isBirthday)
{
    if (isBirthday = true)        → isBirthday == true
    {
        speed = speed - 5;
    }
    if (speed < 60)               → speed <= 60
    {
        return 0;
    }
    else if (speed >= 61 & speed <= 80)   → &&
    {
        return 1;
    }
    else
    {
        return 2;
    }
}
```

Figure 3: Example incorrect solution for `caughtSpeeding`.

tained multiple logical errors and one expert labeled it with fewer errors, it was deemed an incorrect labeling. The labeling process was iterative, with experts calculating Cohen's Kappa scores (κ) after each round. When the agreement was below 0.8, they resolved disagreements and continued labeling new data [41]. The experts achieved $\kappa = 0.78$ in the first round and $\kappa = 0.94$ in the second round. After reaching sufficient agreement, they split the remaining 80% of the data, each expert labeling 40%. They also labeled each logical error with one of the three potential sources mentioned in Section 2.1. This includes compilable syntactic, strategic, and conceptual errors [9, 1, 19]. The kappa scores of the two rounds for the three categories were Syntactic: 0.84 and 0.96, Strategic: 0.72 and 0.86, and Conceptual: 0.78 and 0.92. The same labeling procedure was followed for the other problems. Figure 4 shows the distribution of different logical error categories from each problem. Subsequently, the experts then evaluated the effectiveness of the proposed framework in identifying logical errors by comparing them to their own identified logical errors.

Unlike prior work [42] that relied on accuracy for evaluation, we found it insufficient for measuring the model's ability to detect all relevant errors while avoiding false positives where the model incorrectly assigns higher attention to a subtree as a logical error. Thus, to evaluate the performance of our framework, we employed both recall and precision. Recall measures the proportion of errors detected by the model compared to the total number of errors identified by expert evaluators. This metric highlights the model's accuracy in capturing the errors that experts considered relevant. On the other hand, precision reflects the proportion of important subtrees identified by the model that truly contain errors, preventing an excess of false positives that could be misleading. Focusing on these metrics provides a more nuanced understanding of the model's ability to detect meaningful patterns while ensuring that the substructures flagged by the model are genuinely associated with logical errors.

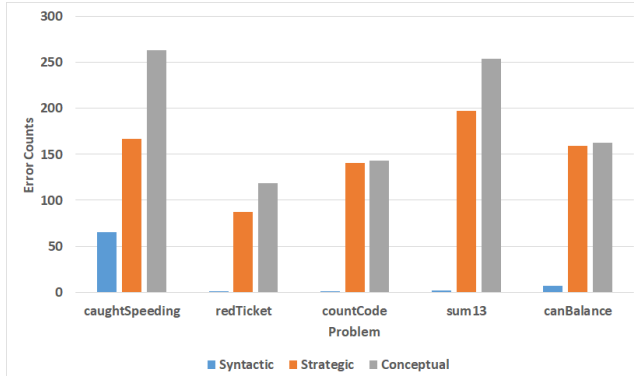

Figure 4: Distribution of logical error types across problems.

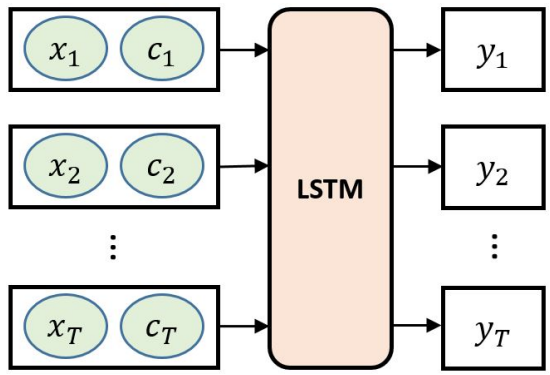

Figure 5: DKT with code vectors.

4.3 Deep Knowledge Tracing

Deep knowledge tracing (DKT) predicts students' short-term performance (next attempt's success), capturing their mastery of skills [54]. It represents a sequence of student attempts, $S = \{x_1, x_2, \ldots, x_T\}$, where each attempt x_t corresponds to a student's interaction at time t within a sequence of T total attempts. Each attempt involves a problem-correctness pair $\{q_t, a_t\}$, where q_t denotes the problem identifier and a_t indicates whether the attempt was correct (1) or incorrect (0). Given a total of M problems, each attempt x_t is one-hot encoded into a binary vector of size $2M$. The element corresponding to $x_{q_t+M(1-a_t)}$ is set to 1, while all other entries remain 0. For example, in the case of $M = 2$ problems, if a student correctly solves problem 1 (i.e., $q_t = 1$ and $a_t = 1$), the corresponding vector becomes $x = \{1, 0, 0, 0\}$, where $x_{1+2(1-1)} = 1$. Conversely, if the student fails on problem 1 (i.e., $q_t = 1$ and $a_t = 0$), the vector is represented as $x = \{0, 0, 1, 0\}$, where $x_{1+2(1-0)} = 1$.

We extract code vectors from the modified SANN model, which implicitly encodes logical error information. The explicit logical errors can be retrieved when necessary. In this study, we concatenate code vectors $(c_1, c_2, \ldots, c_T)$ with the original DKT [54] input structure to feed into an LSTM model, as shown in Figure 5. The output y_t is a vector of

length equal to the total number of problems, where each entry represents the predicted probability that the student will correctly answer a specific problem. The model's prediction for the correctness of the next attempt, a_{t+1}, can be inferred from the entry in y_t corresponding to the next problem q_{t+1}.

4.4 Student Performance Prediction

We employ the vector representations derived from students' programming submissions to all 50 problems ($P_1, P_2, \ldots, P_{50}$) to forecast their final exam grades in the course. This predictive task offers insights into the embedding capability of our approach, enabling the capture of student programming learning information for student modeling purposes. To conduct this analysis, we adopt a methodology similar to a previous study [68], which utilized code embeddings from the `code2vec` model to train an LSTM that predicts student performance based on their programming submission sequence. Figure 6 shows the architecture of the prediction process.

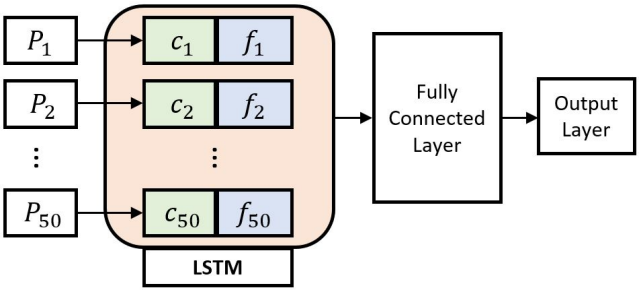

Figure 6: Student performance prediction architecture.

In our dataset, a student can have multiple submissions for each problem. We use all the code vectors for these submissions to predict students' final exam grades and encode their learning progression. We concatenate all student submissions for a single problem into a vector (c_i). Since there was a variable number of submissions for each student-problem pair, we truncated or padded all of the vectors for each submission to create a sequence of length 30. This length was chosen because, for the training dataset, 99% of student-assignment pairs had 30 or fewer submissions. The code vectors were pre-padded with vectors of zeros for problem-student pairs where students made fewer than 30 submissions. For problem-student pairs where students exceeded 30 submissions, the last 30 were kept, assuming the latest submissions were more useful.

We also incorporate additional features (f_i) for each problem, such as the number of correct and incorrect submissions, the total number of submissions, and the problem ID for each student. These features are obtained by aggregating a student's attempts over all 50 assignments. We then augment students' programs with information related to their problem-solving behavior to predict their final exam grades. We use an LSTM network to predict the final exam grades based on the time series feature set containing student submissions for the 50 problems and their problem-solving behavior. After the LSTM layer, we have a fully connected layer with *tanh* activation function to incorporate the information from the LSTM layer and improve the model's expressiveness. Finally, we have a dense layer to predict the final exam grade of a student based on the programming submissions.

Table 1: Performance comparison for program correctness prediction.

Model	Accuracy	Precision	Recall	F1-score
SVM	0.74	0.75	0.63	0.63
KNN	0.75	0.75	0.66	0.70
XGBoost	0.78	0.76	0.77	0.76
code2Vec	0.81	0.84	0.83	0.83
ASTNN	0.83	0.82	0.78	0.80
SANN	**0.87**	**0.86**	**0.88**	**0.87**

5. EXPERIMENTS AND RESULTS

In this section, we present the experts' evaluation of our framework to assess its effectiveness for identifying logical errors. We further demonstrate our framework's relative effectiveness in picking the logical errors from different categories (i.e., compilable syntactic, strategic, and conceptual). Lastly, we highlight the utility of the code vectors enriched with logical error information in capturing students' learning progressions by presenting the experimental results of student models that predict students' short-term and long-term successes.

5.1 Logical Error Identification Effectiveness

Before extracting logical errors from students' incorrect programs, we show the effectiveness of our modified SANN model in predicting program correctness compared to baseline models. We trained the modified SANN model based on the correctness of the student submissions for the 50 problems (correct: 18,787, incorrect: 28,977), framing it as a binary classification task. The dataset was split 80:10:10 for training, validation, and testing in a stratified way due to the imbalanced nature of the dataset. Key training parameters included setting the embedding size of the code vectors to 128, training over 100 epochs with an early stopping of 20 to avoid overfitting, and utilizing the *Adamax* optimizer (learning rate = 0.001) based on the validation data. The entropy regularization weight was carefully tuned to a minimal value (3.5×10^{-5}), ensuring that it did not dominate the primary prediction loss and consequently degrade overall predictive performance. To ensure our modified SANN model can successfully analyze students' code, we conducted comparative evaluations against several existing models from the literature. These models include code2vec [2] and ASTNN [70], both AST-based code representational models and a few traditional ML models. Table 1 shows that our modified SANN model outperformed the other models with the highest accuracy, precision, recall, and F1-score with values of 0.87, 0.86, 0.88, and 0.87, respectively. This demonstrates that our modified SANN model can effectively analyze students' programs.

To evaluate the effectiveness of our model on logical error identification, we calculated recall for the `caughtSpeeding` problem through expert evaluation. Our framework identified 96% of the logical errors from all the code (with an attention threshold of 20%). Next, we calculated the ability of our framework to identify multiple errors from an incorrect

submission. In this evaluation, we marked the identification of a program's logical errors as incorrect if our framework missed at least one of the logical errors present in the code. This means that if there were multiple errors, for example, three, and our framework identified only two of them, we marked it as an incorrect identification. From the expert labels, the maximum number of logical errors in an incorrect program was 10, and the minimum was 1. The average number of logical errors was 2.6. The recall of identifying all logical errors from incorrect solutions using this process was 86%. We also examined the logical errors from different categories labeled by the experts. Our model achieved recalls of 100% in identifying compilable syntactic errors, 95% in identifying strategic errors, and 94% in identifying conceptual errors from all the incorrect submissions. Using expert evaluation, we also calculated the precision of the model, which indicates the proportion of important subtrees identified by the model that contain errors. For the `caughtSpeeding` problem, the precision of the model was calculated to be 90%, highlighting its ability to pinpoint the erroneous substructures within the code effectively. This precision metric further suggests the model's effectiveness in identifying relevant code segments contributing to logical errors.

To further evaluate the robustness of our framework, we expanded the evaluation of our framework to four additional problems: `redTicket`, `countCode`, `sum13`, and `canBalance`, which cover topics of loops, strings, and arrays. These problems were selected to capture a range of logical errors within the constraints of the dataset and of varied complexity. Due to the time and resource limitations associated with expert evaluation, we focused on 200 incorrect student submissions per problem. As shown in Table 2, the framework achieved high recall across the problems, ranging from 83% to 97%. The precision of the model for the four new problems was calculated in the range of 82% to 92%. These recall and precision values indicate that most errors are being accurately identified. However, the model also highlights some subtrees of incorrect submissions that do not contain errors. Preliminary investigation suggests that these highlighted segments sometimes relate to code optimization or efficiency improvements. Additionally, some subtrees pertain to essential programming constructs that are necessary for correctly solving the given programming task. These observations point to future research exploring the connection between the highlighted code segments and students' knowledge state, offering deeper insights into the progression of students' coding abilities.

Baseline Comparison

Several tools have been designed to detect errors in student code, but most either do not provide explainable outputs, such as those based on LLMs, or rely on test cases or predefined rubrics to determine programming errors. We focus our comparisons on explainable approaches that do not require any test cases or rubrics. We developed a naive baseline model that predicts the average number of errors per problem for comparison. Using Root Mean Squared Error (RMSE) as the evaluation metric, the naive baseline achieved 1.65, while our framework outperformed it with an RMSE of 0.45. To compare the performance of our approach further, we considered another explainable model, `code2vec` [2, 62], which splits ASTs into different code paths using an attention network. A `code2vec` model was trained, and the code paths were extracted for the incorrect solutions to the `caughtSpeeding` problem. Due to the labor-intensive nature of expert labeling, we utilized GPT-4o-mini, inspired by recent literature using LLMs as evaluators [49, 12]. To ensure a consistent baseline comparison, GPT-4o-mini was used to evaluate the outputs of our model as well. We employed few-shot prompting, providing the LLM with examples of human-labeled data to facilitate in-context learning [11]. Based on the evaluation, `code2vec` achieved a recall of 43% and a precision of 37%. Our framework outperformed `code2vec` with a recall of 97% and a precision of 85%.

5.2 Tracing Student Knowledge

We used code vectors extracted from the modified SANN model to improve the original DKT model [54] for knowledge tracing. We hypothesize that the extracted code vectors' implicit embedding of logical error information would contribute to tracing student knowledge and be effective in predicting future submission success. We implemented the original DKT model with an LSTM network and incorporated code vectors with the DKT inputs. We used student submissions for the 50 problems for the experiment. The train-validation-test split was 80:10:100. The split was done at the student level to avoid overlap between the different splits. We used Keras-tuner[4] to tune the model's hyperparameters and set the number of neurons of the LSTM model to 200, a dropout of 0.2, and a learning rate of 0.005.

We calculated the AUC score to measure the performance of our modified DKT model with code vectors in predicting students' future success in upcoming attempts to solve a problem. We compared the performance of the original DKT model against our approach by training on all of the problems. Our approach showed a higher AUC score of 72.45% than the original DKT model (65.87%). Table 3 also shows assignment-wise AUC scores for the models, including Code-DKT [60], which was trained for single assignments. The experimental results suggest that our approach shows significant potential in improving deep knowledge tracing, with about 10% higher AUC score than the original DKT. For perspective, our improvement over the original DKT (10%) can be compared to Code-DKT's 3-4% improvement on CodeWorkout [60] or SAINT+'s 2.76% improvement on EdNet [13].

5.3 Predicting Student Performance

We employed code embeddings from our model to predict students' final exam grades based on their programming submissions across the 50 problems in the course curriculum. Predicting a student's final exam grade necessitates comprehensive information on their programming submissions. We fed sequential code vectors to train an LSTM-based student performance prediction model to forecast final exam grades. This approach enables us to evaluate the effectiveness of the generated code vectors in encapsulating sequential insights regarding students' programming proficiency and learning trajectory. We included several baseline models to gauge our method's performance. These included a no-skill model (predicting the average of the final exam grades), Linear Regression, Lasso Regression, SVM, and Gaussian Process

[4]https://keras.io/keras_tuner/

Table 2: Recall (%) and precision (%) of the performance on different problems.

Problem	All error (Recall)	Syntactic (Recall)	Strategic (Recall)	Conceptual (Recall)	Precision
`caughtSpeeding`	96	100	95	94	90
`redTicket`	90	100	85	96	88
`countCode`	97	100	85	96	82
`sum13`	83	100	93	76	90
`canBalance`	92	100	98	85	92

Table 3: Performance (AUC%) comparison for knowledge tracing. Significant improvements of SANN-DKT over DKT are marked with *($p < 0.05$), **($p < 0.01$), and ***($p < 0.001$).

Model	A1	A2	A3	A4	A5	All assignments
DKT	71.24	73.09	76.84	69.16	75.14	65.87
Code-DKT	74.31	76.56	80.4	72.75	79.14	-
SANN-DKT	80***	79.08**	79.8*	81.2***	82.75**	72.45**

Table 4: Students' final exam grade prediction.

Model	RMSE (std)	R^2 (std)
no-skill	0.25 (0.02)	-0.01 (0.02)
Linear	0.38 (0.3)	-0.27 (0.30)
Lasso	0.19 (0.03)	0.19 (0.20)
SVM	0.18 (0.03)	0.35 (0.10)
GPR	0.21 (0.02)	0.07 (0.08)
LSTM-code2vec	0.23 (0.02)	0.09 (0.16)
Proposed model	0.17 (0.03)	0.50 (0.10)

Regression (GPR). For the non-sequential models, we flattened the code features for all 50 problems for an individual student to accommodate the inputs. Additionally, we incorporated a baseline model (LSTM-code2vec) from prior literature [68]. The hyper-parameters for the models were set as follows: for Lasso Regression, α was chosen from $\{1e{-}10, 1e{-}8, 1e{-}6, 1e{-}4, 1e{-}2, 0.1, 1, 2, 5\}$; for SVM, the regularization parameter C was selected as 1 from the set $\{0.001, 0.1, 1, 10\}$ and the kernel was set to "*rbf*" from {"*linear*", "*poly*", "*rbf*"}; for GPR, the kernel was set to "*rbf*", with a length scale of 10 selected from $\{0.5, 1, 10\}$ and noise level of 5 from $\{0.5, 1, 5\}$. Additionally, we tuned our model based on the cross-validation results. We set the number of neurons in the LSTM model to 512 and the number of neurons in the fully connected layer to 128 based on the model's performance on the validation data.

To evaluate model performance, we measured RMSE to assess the deviation between predicted and actual final exam grades. RMSE is particularly suitable as it penalizes larger errors more heavily, providing insights into model accuracy [68]. Additionally, we reported the coefficient of determination (R^2), which indicates how well the independent variables explain the variation in final exam grades. Together, these metrics offer a comprehensive view of model performance. The results of student performance prediction are presented in Table 4. Our LSTM model outperformed other models with the lowest RMSE of 0.17 and the highest R^2 of 0.50, demonstrating its superior predictive capability for student final exam grades. These findings suggest the potential of our modified SANN-based code representations in encoding not just the syntactic and semantic elements of programming but also the effectiveness in capturing the latent states of students' learning, which can offer insights into students' learning progression during the semester.

6. DISCUSSION

Logical errors provide insight into students' challenges when learning and applying programming skills, including revealing potential underlying misconceptions. This insight can guide automated adaptive support to identify and address students' struggles with learning, broadening access to effective instructional support for introductory programming students. Novice programmers often struggle to identify logical errors, in part due to limited debugging experience and potential misconceptions in their programming knowledge [20, 50]. Our approach for automated identification of logical errors from students' programming submissions uses only program correctness information. This lays the foundation for building a generalized, scalable model that can reliably analyze students' programs without significant human input, such as rubrics or designed test cases. An important characteristic of our approach is its ability to explain its decision-making process, contributing to the trustworthiness of its decisions.

Misconceptions are the most common source of logical errors [19]. Additionally, incorporating misconception information can significantly improve student modeling [45]. We demonstrated the relationship between embedded programming code vectors implicitly encompassing students' competencies and logical errors, and students' learning by integrating students' consecutive embedded problem-solving attempts in an LSTM for short-term (next attempt) and long-term (final exam grade) success. Predicting final exam scores and predicting the next attempt success from behavioral data has been challenging in the learning analytics field [39, 68, 60]. We hypothesize that this is because most data used in such analysis are metadata (e.g., number of submissions, time on task, detailed rubrics), dynamic assessment of students' programming, such as test-case results [19], or code context without adequate competency and misconception information. Therefore, there is not enough resemblance between the data used for predicting students' performance and tracking students' learning progression. On the other hand, vectors that are directly trained on students' pro-

grams and in accordance with their correctness stand a better chance of capturing similar information about students' understanding of programming concepts and skills as tested through programming questions on the final exam [45].

In this paper, we focused on identifying logical errors and demonstrating how incorporating logical error information enhances student modeling, specifically by improving predictive accuracy in knowledge tracing. This approach introduces new possibilities for personalized learning and intervention strategies. By leveraging information directly from students' programming submissions, these predictive models can provide educators with valuable insights into students' academic progress and performance. This information can be used to identify students at risk of academic failure and enable educators to provide targeted support and interventions to help these students succeed [60].

While this work advances the automated identification of logical errors in students' programming submissions, it is important to acknowledge several limitations of this study. First, the dataset used in this study consisted of only Java programs, which were comparatively short in length (on average, $10 - 20$ statements). However, this is typical of CS1 programming skill development and practice tasks. Second, we discarded uncompilable programs due to the need for AST generation, consistent with the literature [61, 62], resulting in the loss of a portion of our data. Third, we did not consider logical errors stemming from misinterpretation and carelessness due to the inherent difficulty in identifying these latent constructs from behavioral data. Tracing students' learning progress across multiple solutions can shed light on the underlying reasons behind their behavior. For example, if a model traces students' problem-solving behavior across multiple problems and recognizes a high probability of a student's proficiency with regard to a programming construct, a later mistake related to this construct can represent carelessness or misinterpretation. Finally, during our evaluation, we identified a small subset (less than 1%) of incomplete submissions, lacking one or more critical components of the required functionality. This case can stem from students' carelessness or misinterpretation, or even from the trait of writing a portion of the code, submitting it, and checking whether it is correct or not. These submissions still contained some correct segments and did not necessarily exhibit logical errors in the existing code. Previous research on logical error detection has often overlooked such cases. Interestingly, our framework highlighted substructures indicative of missing constructs, suggesting the potential to extend our study. Specifically, we could integrate additional contextual information regarding core functionalities of the programming tasks rather than relying solely on dynamic test cases. This extension would allow us to detect missing components more effectively and provide personalized hints and feedback, thereby addressing a broader range of student difficulties.

7. DESIGN IMPLICATIONS

By encoding program snippets into latent vectors, the framework we have introduced can efficiently compare student submissions, identifying similar errors while ignoring superficial differences. This can enable instructors to understand the common error patterns in a large number of student submissions [62]. This can also enable instructors to provide targeted feedback at scale by designing feedback for a representative sample of solutions with regard to their encompassing logical errors propagated for similar solutions [63].

Beyond immediate feedback, the framework can support personalized learning by recommending targeted practice problems and instructional materials based on students' errors. By linking incorrect submissions to a repository of similar problems, students could receive recommendations on exercises to reinforce concepts where they have struggled the most. Worked examples, particularly beneficial for novice programmers, can be suggested to highlight correct approaches and common pitfalls, further scaffolding student learning [30, 8, 27]. Additionally, tracking students' problem-solving patterns allows for intelligent student modeling, enabling dynamic adjustments to learning pathways and problem difficulty [33]. This personalized, data-driven approach can promote deeper engagement, reduce unproductive struggles, and ensure continuous learning progression, making programming education more effective, reliable, and scalable.

8. CONCLUSION AND FUTURE WORK

Timely identification and intervention in students' programming struggles are crucial for enhancing engagement, learning, and, ultimately, retention. However, the persistent lack of resources to provide students with individualized instructional support poses a significant challenge in providing students with the crucial help they need while learning how to program. Automated and explainable data-driven approaches can help fill this gap by providing struggling students with adaptive support and assistance through reliable frameworks that address their individual needs. Current methods either rely heavily on manual effort to annotate the data or use complex large language models to provide support that lacks reliability and explainability. For effective integration in the classroom, we need to have code analysis methodologies that are reliable and explainable and require minimal input from human experts while still providing actionable insights for instructors and students.

This paper presented a framework for automated, explainable, and scalable identification of logical errors, leveraging their implicit presence in program embeddings to predict students' performance in programming courses. We accurately identified logical errors by training a modified SANN model on student programming submissions. Furthermore, we achieved promising results in predicting students' short-term (next problem attempts) success with 10% improvement over the original DKT and long-term (final exam grades) success in the course with 5% improvement over the best-performing baseline. Our findings demonstrate the potential of our framework in identifying student errors from their programming submissions to capture intricate details on their programming learning progression. This capability offers promise for seamless integration into adaptive instructional technologies, enabling automated and personalized scaffolding, including feedback and recommendations to support student learning experiences and outcomes.

This work suggests several promising directions for future work. First, it will be instructive to investigate how to au-

tomatically map student logical errors to underlying misconceptions by analyzing program elements and understanding the deficiencies in students' incorrect submissions. Second, exploring a misconception tracing approach that leverages multiple occurrences of related logical errors across student programs can enhance our understanding of students' learning challenges and inform targeted interventions to effectively address misconceptions. Third, integrating the framework into adaptive learning environments to provide students with personalized feedback and opportunities for mastery learning can reveal how to most effectively improve student learning experiences and outcomes in computer science education.

9. 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.

10. REFERENCES

[1] E. Albrecht and J. Grabowski. Sometimes it's just sloppiness-studying students' programming errors and misconceptions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 340–345, New York, NY, USA, 2020. Association for Computing Machinery.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[3] A. Altadmri and N. C. 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, New York, NY, USA, 2015. Association for Computing Machinery.

[4] N. Alzahrani and F. Vahid. Common logic errors for programming learners: A three-decade literature survey. In *Proceedings of the 2021 ASEE Annual Conference*, pages 2–18. American Society for Engineering Education, 2021.

[5] P. Ardimento, M. L. Bernardi, and M. Cimitile. Software analytics to support students in object-oriented programming tasks: An empirical study. *IEEE Access*, 8:132171–132187, 2020.

[6] P. Baffes. Learning to model students: Using theory refinement to detect misconceptions. Technical report, Artificial Intelligence, University of Texas at Austin, 1994.

[7] R. S. d. Baker, A. T. Corbett, and V. Aleven. More accurate student modeling through contextual estimation of slip and guess probabilities in bayesian knowledge tracing. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems*, pages 406–415. Springer, 2008.

[8] J. Barria-Pineda, K. Akhuseyinoglu, and P. Brusilovsky. Adaptive navigational support and explainable recommendations in a personalized programming practice system. In *Proceedings of the 34th ACM Conference on Hypertext and Social Media*, pages 1–9, New York, NY, USA, 2023. Association for Computing Machinery.

[9] P. Bayman and R. E. Mayer. Using conceptual models to teach basic computer programming. *Journal of Educational Psychology*, 80(3):291–298, 1988.

[10] P. Bian, B. Liang, W. Shi, J. Huang, and Y. Cai. Nar-miner: Discovering negative association rules from code for bug detection. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 411–422, New York, NY, USA, 2018. Association for Computing Machinery.

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, volume 33 of *NIPS '20*, pages 1877–1901, 2020.

[12] C.-H. Chiang and H.-y. Lee. Can large language models be an alternative to human evaluations? *arXiv preprint arXiv:2305.01937*, 2023.

[13] Y. Choi, Y. Lee, D. Shin, J. Cho, S. Park, S. Lee, J. Baek, C. Bae, B. Kim, and J. Heo. Ednet: A large-scale hierarchical dataset in education. In *Proceedings of the International Conference on Artificial Intelligence in Education*, pages 69–73. Springer, 2020.

[14] A. T. Corbett and J. R. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-adapted Interaction*, 4:253–278, 1994.

[15] R. Davies, R. Nyland, J. Chapman, and G. Allen. Using transaction-level data to diagnose knowledge gaps and misconceptions. In *Proceedings of the 5th International Conference on Learning Analytics and Knowledge*, pages 113–117, New York, NY, USA, 2015. Association for Computing Machinery.

[16] D. Demszky, J. Liu, H. C. Hill, D. Jurafsky, and C. Piech. Can automated feedback improve teachers' uptake of student ideas? evidence from a randomized controlled trial in a large-scale online course. *Educational Evaluation and Policy Analysis*, 46(3):483–505, 2024.

[17] S. H. Edwards and K. P. Murali. Codeworkout: Short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 188–193, New York, NY, USA, 2017. Association for Computing Machinery.

[18] M. Elmadani, M. Mathews, and A. Mitrovic. Data-driven misconception discovery in constraint-based intelligent tutoring systems. In *Proceedings of the 20th International Conference on Computers in Education*, pages 1–8. University of Canterbury, CSSE, 2012.

[19] A. Ettles, A. Luxton-Reilly, and P. Denny. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 83–89, New York, NY, USA, 2018. Association for Computing Machinery.

[20] S. Fitzgerald, G. Lewandowski, R. McCauley,

L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, 2008.

[21] H. L. Fwa. Experience report: Identifying common misconceptions and errors of novice programmers with chatgpt. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, pages 233–241, New York, NY, USA, 2024. Association for Computing Machinery.

[22] D. Glandorf, H. R. Lee, G. A. Orona, M. Pumptow, R. Yu, and C. Fischer. Temporal and between-group variability in college dropout prediction. In *Proceedings of the 14th International Conference on Learning Analytics and Knowledge*, pages 486–497, New York, NY, USA, 2024. Association for Computing Machinery.

[23] E. Guzmán, R. Conejo, and J. Gálvez. A data-driven technique for misconception elicitation. In *Proceedings of the User Modeling, Adaptation, and Personalization: 18th International Conference, UMAP*, pages 243–254. Springer, 2010.

[24] M. Hoq, P. Brusilovsky, and 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*, pages 79–90, Bengaluru, India, 2023. International Educational Data Mining Society.

[25] M. Hoq, P. Brusilovsky, and B. Akram. Explaining explainability: Early performance prediction with student programming pattern profiling. *Journal of Educational Data Mining*, 16(2):115–148, 2024.

[26] M. Hoq, S. R. Chilla, M. Ahmadi Ranjbar, P. Brusilovsky, and 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*, pages 783–792, New York, NY, USA, 2023. Association for Computing Machinery.

[27] M. Hoq, A. Patil, K. Akhuseyinoglu, P. Brusilovsky, and 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*, pages 527–533, New York, NY, USA, 2025. Association for Computing Machinery.

[28] M. Hoq, Y. Shi, J. Leinonen, D. Babalola, C. Lynch, T. Price, and 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*, page 526–532, New York, NY, USA, 2024. Association for Computing Machinery.

[29] M. Hoq, J. Vandenberg, B. Mott, J. Lester, N. Norouzi, and B. Akram. Towards attention-based automatic misconception identification in introductory programming courses. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE) V. 2*, pages 1680–1681, New York, NY, USA, 2024. Association for Computing Machinery.

[30] R. Hosseini, K. Akhuseyinoglu, P. Brusilovsky, L. Malmi, K. Pollari-Malmi, C. Schunn, and T. Sirkiä. Improving engagement in program construction examples for learning python programming. *International Journal of Artificial Intelligence in Education*, 30(2):299–336, 2020.

[31] C. L. Hovey, L. Barker, and V. Nagy. Survey results on why cs faculty adopt new teaching practices. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 483–489, New York, NY, USA, 2019. Association for Computing Machinery.

[32] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[33] Y. Huang, P. Brusilovsky, J. Guerra, K. Koedinger, and C. Schunn. Supporting skill integration in an intelligent tutoring system for code tracing. *Journal of Computer Assisted Learning*, 39(2):477–500, 2023.

[34] M. Jamjoom, E. Alabdulkreem, M. Hadjouni, F. Karim, and M. Qarh. Early prediction for at-risk students in an introductory programming course based on student self-efficacy. *Informatica*, 45(6):1–9, 2021.

[35] Q. Jia, J. Cui, R. Xi, C. Liu, P. Rashid, R. Li, and E. Gehringer. On assessing the faithfulness of llm-generated feedback on student assignments. In *Proceedings of the 17th International Conference on Educational Data Mining*, pages 491–499. International Educational Data Mining Society, 2024.

[36] Z. Jia, W. Su, J. Liu, and W. Yue. Attentive q-matrix learning for knowledge tracing. *arXiv preprint arXiv:2304.08168*, 2023.

[37] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, pages 107–111, New York, NY, USA, 2010. Association for Computing Machinery.

[38] M. Karlovčec, M. Córdova-Sánchez, and Z. A. Pardos. Knowledge component suggestion for untagged content in an intelligent tutoring system. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 195–200, 2012.

[39] G. Kennedy, C. Coffrin, P. De Barba, and L. Corrin. Predicting success: how learners' prior knowledge, skills and activities predict mooc performance. In *Proceedings of the 5th International Conference on Learning Analytics and Knowledge*, pages 136–140, New York, NY, USA, 2015. Association for Computing Machinery.

[40] A. J. Ko and B. A. Myers. Development and evaluation of a model of programming errors. In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments*, pages 7–14. IEEE, IEEE, 2003.

[41] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[42] Y. Lee, S. Jeong, and J. Kim. Improving llm classification of logical errors by integrating error

relationship into prompts. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 91–103. Springer, 2024.

[43] Y. Liang, T. Peng, Y. Pu, and W. Wu. Help-dkt: an interpretable cognitive model of how students learn programming based on deep knowledge tracing. *Scientific Reports*, 12(1):4012–4023, 2022.

[44] M.-h. Liu. Blending a class video blog to optimize student learning outcomes in higher education. *The Internet and Higher Education*, 30:44–53, 2016.

[45] R. Liu, R. Patel, and K. R. Koedinger. Modeling common misconceptions in learning process data. In *Proceedings of the 6th International Conference on Learning Analytics and Knowledge*, pages 369–377, New York, NY, USA, 2016. Association for Computing Machinery.

[46] J. Marsden, S. Yoder, and B. Akram. Predicting Student Performance with Control-flow Graph Embeddings. In B. Akram, T. W. Price, Y. Shi, P. Brusilovsky, and S. I. Han Hsiao, editors, *6th Educational Data Mining in Computer Science Education (CSEDM) Workshop*, pages 32–40, Durham, UK, 2022. `https://doi.org/10.5281/zenodo.6983402`.

[47] A. C. Martin, K. M. Ying, F. J. Rodríguez, C. S. Kahn, and K. E. Boyer. Intelligent support for all? a literature review of the (in) equitable design & evaluation of adaptive pedagogical systems for cs education. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*, pages 996–1002, New York, NY, USA, 2022. Association for Computing Machinery.

[48] S. Marwan, B. Akram, T. Barnes, and T. W. Price. Adaptive immediate feedback for block-based programming: Design and evaluation. *IEEE Transactions on Learning Technologies*, 15(3):406–420, 2022.

[49] J. McClure, D. Smyslova, A. J. Hall, and S. Jiang. Deductive coding's role in ai vs. human performance. In B. Paassen and C. D. Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 809–813. International Educational Data Mining Society, 2024.

[50] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Proceedings of the Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, pages 125–180. Association for Computing Machinery, New York, NY, USA, 2001.

[51] M. J. Nathan and A. Petrosino. Expert blind spot among preservice teachers. *American Educational Research Journal*, 40(4):905–928, 2003.

[52] W. Paul and J. Vahrenhold. Hunting high and low: Instruments to detect misconceptions related to algorithms and data structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages 29–34, New York, NY, USA, 2013. Association for Computing Machinery.

[53] G. Pereyra, G. Tucker, J. Chorowski, Ł. Kaiser, and G. Hinton. Regularizing neural networks by penalizing confident output distributions. *arXiv preprint arXiv:1701.06548*, 2017.

[54] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein. Deep knowledge tracing. In *Proceedings of the 29th International Conference on Neural Information Processing Systems*, page 505–513, Cambridge, MA, USA, 2015. MIT Press.

[55] Y. Qian and J. Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–24, 2017.

[56] J. Raigoza. A study of students' progress through introductory computer science programming courses. In *Proceedings of the 2017 IEEE Frontiers in Education Conference (FIE)*, pages 1–7. IEEE, 2017.

[57] K. Rivers and K. R. Koedinger. Automatic generation of programming feedback: A data-driven approach. In *Proceedings of the First Workshop on AI-supported Education for Computer Science (AIEDCS)*, volume 50, pages 50–59, 2013.

[58] A. M. Shahiri, W. Husain, et al. A review on predicting student's performance using data mining techniques. *Procedia Computer Science*, 72:414–422, 2015.

[59] Y. Shi, T. Barnes, M. Chi, and T. Price. Evaluating multi-knowledge component interpretability of deep knowledge tracing models in programming. In B. Paassen and C. D. Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 288–295. International Educational Data Mining Society, 2024.

[60] Y. Shi, M. Chi, T. Barnes, and T. Price. Code-dkt: A code-based knowledge tracing model for programming tasks. In A. Mitrovic and N. Bosch, editors, *Proceedings of the 15th International Conference on Educational Data Mining*, pages 50–61. International Educational Data Mining Society, 2022.

[61] Y. Shi, Y. Mao, T. Barnes, M. Chi, and T. W. Price. More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code. In I.-H. Hsiao, S. Sahebi, F. Bouchet, and J. Vie, editors, *Proceedings of the 14th International Conference on Educational Data Mining*, pages 446–453. International Educational Data Mining Society, 2021.

[62] Y. Shi, K. Shah, W. Wang, S. Marwan, P. Penmetsa, and T. Price. Toward semi-automatic misconception discovery using code embeddings. In *Proceedings of the 11th International Conference on Learning Analytics and Knowledge*, pages 606–612, New York, NY, USA, 2021. Association for Computing Machinery.

[63] P. Silva, E. Costa, and J. R. de Araújo. An adaptive approach to provide feedback for students in programming problem solving. In A. Coy, Y. Hayashi, and M. Chang, editors, *Proceedings of the 15th International Conference on Intelligent Tutoring Systems, ITS*, pages 14–23. Springer, 2019.

[64] J. Sorva. Notional machines and introductory programming education. *ACM Transactions Computing Education*, 13(2):1–31, 2013.

[65] A. Swidan, F. Hermans, and M. Smit. Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 151–159, New York, NY, USA, 2018. Association for Computing Machinery.

[66] D. Wang, L. Zhang, Y. Zhao, Y. Zhang, S. Yan, and M. Hou. Deep knowledge tracking integrating programming exercise difficulty and forgetting factors. In D.-S. Huang, Z. Si, and W. Chen, editors, *Proceedings of the International Conference on Intelligent Computing*, pages 192–203. Springer, 2024.

[67] Z. Xu, S. Jain, and M. Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*, 2024.

[68] S. Yoder, M. Hoq, P. Brusilovsky, and B. Akram. Exploring sequential code embeddings for predicting student success in an introductory programming course. In B. Akram, T. W. Price, Y. Shi, P. Brusilovsky, and S. I. Han Hsiao, editors, *Proceedings of the 6th Educational Data Mining in Computer Science Education (CSEDM) Workshop*, pages 2–9, Durham, UK, 2022.

[69] M. V. Yudelson, K. R. Koedinger, and G. J. Gordon. Individualized bayesian knowledge tracing models. In H. C. Lane, K. Yacef, J. Mostow, and P. Pavlik, editors, *Proceedings of the 16th International Conference on Artificial Intelligence in Education*, pages 171–180. Springer, 2013.

[70] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.