Knowledge Tracing in Programming Education Integrating Students' Questions

Doyoun Kim[1], **Suin Kim**[2], **Yohan Jo**[1]
[1]Seoul National University, [2]Elice
{xxxdokki, yohan.jo}@snu.ac.kr, suin.kim@elicer.com

Abstract

Knowledge tracing (KT) in programming education presents unique challenges due to the complexity of coding tasks and the diverse methods students use to solve problems. Although students' questions often contain valuable signals about their understanding and misconceptions, traditional KT models often neglect to incorporate these questions as inputs to address these challenges. This paper introduces SQKT (Students' Question-based Knowledge Tracing), a knowledge tracing model that leverages students' questions and automatically extracted skill information to enhance the accuracy of predicting students' performance on subsequent problems in programming education. Our method creates semantically rich embeddings that capture not only the surface-level content of the questions but also the student's mastery level and conceptual understanding. Experimental results demonstrate SQKT's superior performance in predicting student completion across various Python programming courses of differing difficulty levels. In in-domain experiments, SQKT achieved a 33.1% absolute improvement in AUC compared to baseline models. The model also exhibited robust generalization capabilities in cross-domain settings, effectively addressing data scarcity issues in advanced programming courses. SQKT can be used to tailor educational content to individual learning needs and design adaptive learning systems in computer science education.[1]

1 Introduction

Recent advancements in educational technologies have enabled the collection of dynamic data as students interact with learning systems. Consequently, researchers have paid considerable attention to knowledge tracing (KT), which involves monitoring students' knowledge states and predicting their future performance (Corbett and Anderson, 1994). A valuable source of signals about students' understanding and misconceptions is the questions they ask (Sun et al., 2021). With the growing popularity of online learning platforms and learning management systems (e.g., Moodle and Canvas) that include Q&A forums, student questions and interactions with educators have become increasingly accessible. However, traditional KT models overlook this rich source of information. This gap is particularly significant in programming education, where KT is challenging because students' competencies need be assessed from unstructured and noisy source code. In such contexts, students' questions offer clearer insights into their understanding and confusion (King, 1994).

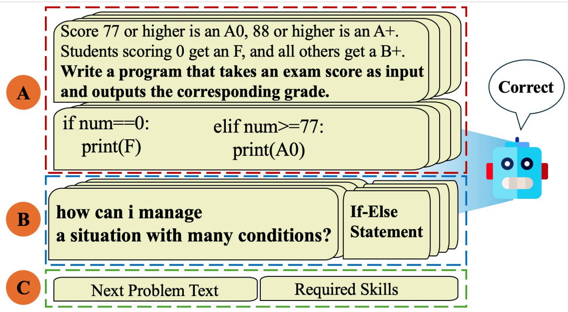

Figure 1: SQKT's process using an example from our dataset. A: All problem descriptions and code submissions from the student's history. B: The questions the student asked between submissions and the related skills extracted from these questions. C: The description of the next problem and the required skills inferred from the reference solution. The model uses the information from A and B and predicts the student's success or failure on the next problem.

In this paper, we present the first model that integrates rich signals from student questions to accurately predict students' performance on subsequent problems, as illustrated in Figure 1. As we will show, merely using a transformer to encode

[1]This paper is under review. We will release our source code upon the publication of the paper.

students' questions is suboptimal, because it might not fully represent the patterns of confusion and educational context that could be captured through the interaction between student and educator. Hence, our model enriches embeddings with two auxiliary signals: educator responses and skill information auto-extracted by GPT from students' questions. This approach creates a more comprehensive and detailed representation of the student's understanding, leading to improved prediction accuracy.

Experimental results show significant improvements over existing methods, with up to a 33.1% absolute improvement in AUC compared to baseline models in in-domain experiments. Our model's ability to generalize across diverse educational content, including unseen courses with limited data, highlights its robustness. Our analysis reveals that this performance boost stems from the questions and the automatically extracted skill information, which offer insights into conceptual understanding and reasoning processes that are difficult to capture from code submissions alone. The combination of student questions with dynamically extracted skill information enables more accurate and granular modeling of student knowledge states. Our approach is expected to contribute to more personalized and effective learning interventions in programming education.

The main contributions of this paper are:

- To the best of our knowledge, this is the first work to integrate students' questions into KT, enabling more accurate predictions of students' success or failure on subsequent problems.
- Our method for combining auto-extracted Python skills with student questions significantly improves model performance compared to relying solely on natural language questions.
- Our model's strong performance in cross-domain settings highlights its generalizability across different course materials and environments.

2 Related Works

Knowledge Tracing with Behavioral Data Knowledge tracing (KT) models students' knowledge over time to predict their future performance (Piech et al., 2015). Building upon the foundational approaches like Bayesian Knowledge Tracing (BKT) (Corbett and Anderson, 1994) and Deep Knowledge Tracing (DKT) (Piech et al., 2015), recent research has advanced KT by incorporating behavioral data, such as response times (Song et al., 2021), scaffolding interactions (Asselman et al., 2020), and attempt counts (Sun et al., 2022).

However, a significant gap remains in leveraging student-educator interactions. Questions arising from these interactions often reveal students' reasoning processes and areas of struggle in applying theoretical knowledge to coding (Sun et al., 2021). Yet, most existing models fail to capture the valuable insights embedded in students' questions. Our model addresses this challenge by directly integrating this rich data.

Knowledge Tracing in Programming Education Programming education poses unique challenges for KT due to the complexity of coding tasks and multiple correct solutions that can be derived using various skills. Traditional KT models often use the Q-matrix method to manually tag problems with the required skills (Yu et al., 2022), but this process is labor-intensive and often fails to capture the full range of skills students use. The diversity in problem-solving approaches complicates the tracing of specific skills mastered by a student, making it challenging to predict future performance accurately.

A key aspect of KT in programming education is the representation and modeling of knowledge components (KCs), such as "for loop", "recursion", or "object-oriented principles". Recent work has focused on analyzing code submissions to model these KCs and predict learning states. Shi et al. (2022) introduced Code-DKT, which uses attention mechanisms to extract domain-specific code features. Liu et al. (2022) developed an approach that considers the multi-skill nature of programming exercises by learning features from student code that reflect multiple skills.

However, these approaches still rely on manual tagging of KCs. Our approach advances this by using an automated skill-mapping system using GPT to extract KCs from student questions. This method allows more flexible use of KCs, enabling the model to identify and leverage skills without extensive manual tagging. This skill extraction method captures aspects of student knowledge that are not evident in code submissions alone, improving the prediction of student performance.

3 Methods

In this section, we introduce our Students' Question-based Knowledge Tracing (SQKT) model. Our primary goal is to predict a student's

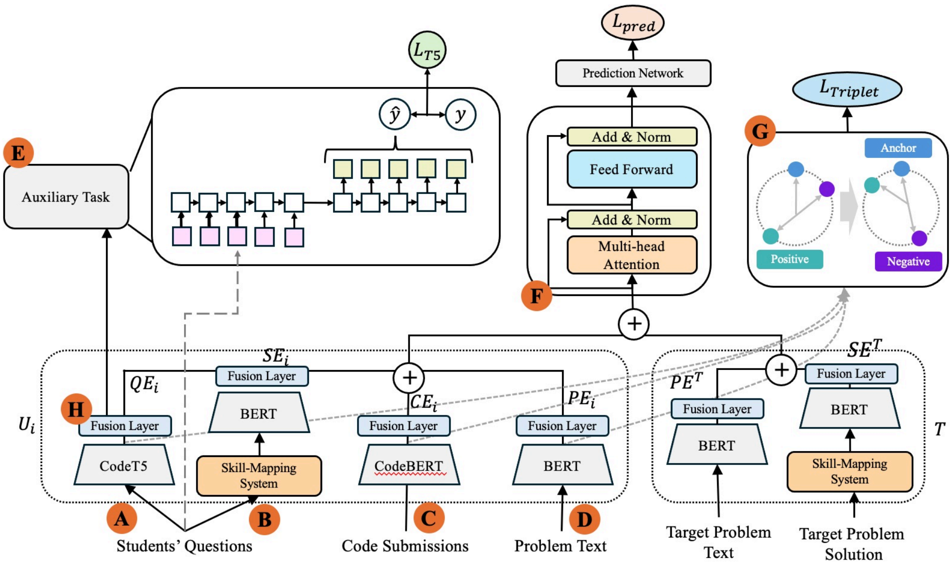

Figure 2: Comprehensive architecture of the SQKT. The model processes problem text, code submissions, and student questions through three embedding layers. Skill extraction is performed using a GPT-based skill-mapping system. All embeddings and extracted skills are combined through a fusion layer, which is then processed by transformer encoder layers to generate the final prediction output. The model is trained using multiple objective functions, including $L_{triplet}$ for aligning the diverse embeddings and L_{pred} for predicting students' performances on tasks. Additionally, the auxiliary objective function $L_{question}$ is included to enhance the model's robustness and generalization capabilities.

success on a problem by integrating information about the student's history of solving other problems. Our model takes a sequence of descriptions of problems the student has attempted in the past, associated code submissions, student questions, and skill information (each problem may have multiple submissions and questions), along with the description and required skills for the next problem. The model then predicts whether the student will correctly solve the next problem. The overall architecture is illustrated in Figure 2.

3.1 Multi-feature Inputs

SQKT integrates various input features, with a focus on students' questions and extracted skills. In this section, we first detail our main contribution: the integration of students' questions as input features, followed by an overview of the remaining input components.

Student Questions (Figure 2, A) Integrating student questions is motivated by valuable insights they provide into a student's mastery level, revealing areas of confusion and the depth of understanding of specific concepts (Sun et al., 2021). As illustrated by an example student-educator interaction in Figure 3, students' questions typically include two types of information: natural language ques-

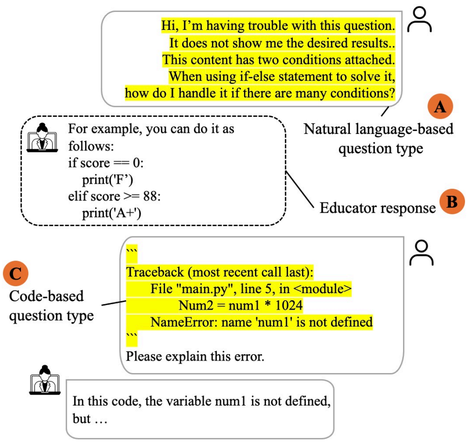

Figure 3: This figure illustrates different types of student questions and interactions. (A) Natural language-based questions (B) Educator responses (C) Code-based questions

tions that seek clarification on specific concepts or strategies (A), and code-based questions that address specific lines of code or errors (C). Educator responses further clarify misconceptions, provide additional context, and highlight key concepts (B).

To effectively leverage this information, we employ the CodeT5 model (Wang et al., 2021) for embedding student questions. CodeT5 was chosen for its ability to understand both natural language and code syntax, making it ideal for processing the mixed content of students' questions. If no student questions exist when SQKT expects a question embedding, a zero vector is used instead.

We further enhance this embedding process by fine-tuning CodeT5 with an auxiliary task of generating potential educator responses (Figure 2, E). This task helps the question embedding capture the gist of a student's question (mainly confusion and erroneous code) that is predictive of the educator's response. The following auxiliary loss function is integrated into our overall training objective:

$$L_{question} = -\Sigma_{(x,y)} \log P(y|x) \tag{1}$$

where (x, y) is a pair of student question and educator response. This enhances question embeddings and the model's overall prediction accuracy.

Skill Extraction (Figure 2, B) Identifying the skills students struggle with can improve our model's performance compared to relying solely on questions. Extracting skills from student questions is more straightforward and accurate than from submitted codes, as these questions often directly address the concepts students find challenging. By combining these extracted skills with those required for the target problem, the model can predict a student's performance more accurately. To achieve this, we developed a method to extract and leverage skill information from both student questions and target problems.

The first challenge was to define an effective set of Python skills. We identified a comprehensive set of 36 core Python concepts and 19 Python error types, drawing from Python's official documentation and books by Sweigart (2019) and Downey and Mayfield (2019), as shown in Table 5. Incorporating error types as skills was motivated by the pedagogical principle that errors reveal students' understanding and misconceptions, which are correlated with learning gaps (Altadmri and Brown, 2015; Becker et al., 2019; Hertz and Ford, 2013).

The next challenge was scaling skill extraction. Traditional approaches rely on experts to manually tag skills for each problem, which is labor-intensive and lacks scalability. To address this, we developed an automatic method using GPT-4o. Specifically, we provided GPT-4o with about 20 examples of student questions and a pre-defined list of skills. GPT-4o was then prompted to reference these examples and generate a Python script that could be used to map any student question to the relevant skills from our predefined skill list. The resulting skill extraction script, referred to as the *skill extractor*, uses specific rules to identify skills from both natural text and code. We found that a rule-based method is preferable to using GPT on the fly, due to high precision and consistency in skill identification. We reviewed the script and corrected inaccurate or unreliable rules based on some student questions manually labeled with skills.

To validate the skill extractor more systematically, we evaluated it on a random sample of 100 student questions from the "Python Basic" course (§4.1). These questions were annotated with ground-truth skills by a co-author of this study, and these annotations were further validated by a graduate student proficient in Python but not involved in this study, resulting in Cohen's kappa of 0.98. The skill extractor achieved a precision of 0.85, a recall of 0.88, and an F1-score of 0.86. These results indicate that the skill extractor produces reliable outputs that closely match human judgments. We considered this level of accuracy acceptable, as extracted skills substantially improve SQKT's predictive performance (as discussed in the experiment section).

We first use the skill extractor to extract skill information from student questions. Specifically, it processes student questions to identify the particular skills students are struggling with. These identified skills are concatenated as a single text and encoded into a *skill embedding* using the pre-trained BERT-base model (Devlin et al., 2018). In addition, the skills required to solve each problem are identified by applying the skill extractor to the reference solution code provided with each problem in our dataset. Taken together, this approach enables SQKT to align the extracted skills with specific skills required for the target problem, thereby enhancing its predictive accuracy.

Code Embedding (Figure 2, C) We use CodeBERT (Feng et al., 2020), a pre-trained transformer model designed for programming languages, to convert students' code submissions into vector representations. This captures both the syntactic and semantic properties of the code, providing insights into the student's coding abilities and problem-solving strategies.

Problem Embedding (Figure 2, D) Problem descriptions include the problem statement, input/output specifications, and constraints. They are processed through the pre-trained BERT-base model (Devlin et al., 2018) to generate an embedding that captures the contextual meanings of the problem statements. This information is crucial for understanding the task requirements and difficulty levels.

Fusion Layer (Figure 2, H) The fusion layer combines the above embeddings—questions, skills, problem descriptions, and code submissions—into a unified representation space. While each source provides unique insights, challenges remain in integrating these heterogeneous signals. The fusion layer addresses this by projecting each embedding type into a common 512-dimensional space based on the relationships among the embeddings.

Specifically, we employ triplet loss (Figure 2, G) to encourage embeddings from the same submission to be positioned closely together, while those from different submissions or representing distinct programming concepts are placed farther apart. The triplet loss is defined as:

$$L_{triplet} = max(0, d(a, p) - d(a, n) + margin), \quad (2)$$

where:

- a is the current problem's embedding derived from the student's code submission, serving as the anchor embedding.
- p is the embedding of the current problem's description or student questions, serving as positive samples.
- n is the embedding of a randomly selected problem's description or student questions, serving as negative samples.
- $d(x, y)$ is the Euclidean distance between two embeddings x and y.
- $margin$ is a hyperparameter enforcing a minimum distance between positives and negatives.

Consequently, the fusion layer enhances SQKT's ability to process heterogeneous yet semantically and contextually related signals more coherently.

3.2 Multi-Head Self-Attention Layers

All embeddings from the student's history and the next problem are encoded through a multi-head attention mechanism to predict the student's success or failure on the next problem (Figure 2, F). The target problem for prediction is represented by the following tensor:

$$T = [PE^T, SE^T] \in R^{2\times512}$$

where PE^T and SE^T are the problem and skill embeddings for the target problem.

For each problem i that the student attempted prior to the target problem, we construct a tensor U_i containing the input features associated with the ith problem:

$$U_i = [PE_i, CE_i, QE_i, SE_i] \in R^{K\times512}$$

where PE_i, CE_i, QE_i, and SE_i denote the problem, code, student question, and skill embeddings, respectively. Each CE_i, QE_i, and SE_i is a tensor with potentially multiple rows, consisting of embeddings accumulated from all code submissions and questions related to the ith problem. If the student asked no questions, QE_i is set to a zero vector. K increase as the student makes more submissions for the ith problem.

Taken together, the input to the multi-head self-attention layers consists of the target problem along with all preceding learning history $[U_1, U_2, \ldots, U_n, T]$.

This input sequence passes through six self-attention layers, each capturing complex interactions among different submissions and their components. After the final attention layer, max-pooling is applied to all output embeddings to derive a representation of the student's knowledge state. The resulting embedding is then fed to a classification head to predict the student's success or failure on the target problem. Binary cross-entropy is used as the loss function:

$$L_{pred} = -\Sigma(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})),$$

where y is the true label and $\hat{y}$ is the predicted label.

The final loss function is a weighted sum of the prediction loss, question loss (Eq. 1) and triplet loss (Eq. 2):

$$L_{total} = L_{pred} + L_{question} + \lambda L_{triplet}$$

where λ is a hyperparameter that adjusts the weight of the triplet loss.

4 Experiment Settings

To evaluate the performance and generalizability of our SQKT model, we conduct experiments aimed at answering the following research questions:

1. How does SQKT compare to existing knowledge tracing models in predicting student performance on programming problems?

Attribute	PB	FP	Algo.	PI
Unique problems	48	60	32	227
Submissions per problem	474	20,573	297	1,533
Students	160	8,141	77	1,092
Training	17,685	1,050,360	5,689	308,825
Validation	2,161	123,975	2,233	20,991
Test	2,926	60,071	1,587	18,251

Table 1: Statistics of the dataset. PB: Python Basic, FP: First Python, Algo: Algorithm, PI: Python Introduction.

2. To what extent does the integration of question data and skill information enhance the model's predictive accuracy?
3. How well does SQKT generalize across different courses with different difficulty levels?

4.1 Dataset

Our study uses data collected from a Korean online programming education platform between January 2022 and April 2024, with the consent of the copyright holders. These data cover four distinct Python programming courses, providing a diverse range of difficulty levels and topics. All data are in Korean and include Python code, covering code blocks and associated error messages. Statistics are summarized in Table 1 and description and examples are in Appendix B and D. Additionally, the data contain student-educator interactions including student questions and educator answers (Figure 3). The data for each course are split by students into training, validation, and test sets in an 8:1:1 ratio, with each student assigned to only one set to prevent the risk of information leakage.

4.2 Experimental Setup

We conduct a series of experiments to assess two critical aspects: the model's ability to predict student performance and generalize across different courses and difficulty levels. We perform both in-domain and cross-domain experiments.

In-domain We evaluate the model's performance when trained and tested on the same course. We experiment with three out of four courses, excluding one due to insufficient data for stable training.

Cross-Domain We selected courses to challenge the model's adaptability and generalization capabilities. In the first cross-domain setting, labeled 'content structure generalization', we train the model on the "Python Introduction" course (5,858 samples) and tested it on the "First Python" course (1,674 samples). This pair was chosen to evaluate the model's ability to transfer knowledge between courses with different content structures, including varying difficulty levels and vocabulary usage.

In the second cross-domain experiment, labeled 'data-scarce generalization', we train the model on combined data from all courses except "Algorithm" (9,390 samples) and tested it exclusively on the "Algorithm" course (300 samples). This choice was motivated by our initial observation that the model struggled on this course due to the small data size. Through this setting, we aimed to verify the model's ability to generalize to a specialized, data-scarce course by leveraging student questions.

Baseline Models To benchmark SQKT's performance and evaluate the impact of integrating student questions, we compare it with several baseline models. Our choice of baseline models is limited by the scarcity of prior KT models capable of processing code submissions without relying on predefined skill annotations. Moreover, to the best of our knowledge, no existing KT models incorporate student questions. To that end, we experiment with four baseline models: KTMFF (Xiao et al., 2023), OKT (Liu et al., 2022), and their variants. KTMFF and OKT are known for their strong performance in leveraging rich embeddings of code submissions and capturing the structural properties of code blocks. To verify the effectiveness our question embeddings and demonstrate their adaptability for enhancing different models, we introduce KTMFF+ and OKT+, variants of KTMFF and OKT that incorporate question embeddings as additional input.

Evaluation Metrics To evaluate each model's performance in predicting student success on programming problems, we use AUC, accuracy, and F1-score based on the model's predictions. Here, a student is considered successful on a problem if they achieve a score of 100 within a certain number of trials. This threshold is set to the average number of submissions across all students in each course. Any score below 100 or a submission count exceeding this threshold is considered a failure.

4.3 Training Setup

For each student, we predict the student's outcome for every problem they attempted, excluding the first problem since it has no preceding history. For

	Python Introduction			First Python			Python Basic		
	AUC	ACC	F1	AUC	ACC	F1	AUC	ACC	F1
KTMFF	70.2	64.5	56.5	69.4	61.8	60.3	78.0	73.5	80.0
KTMFF+	72.6	66.1	58.2	71.7	62.1	60.7	80.7	76.1	81.4
OKT	60.3	81.0	34.6	65.8	77.7	49.1	65.0	78.8	46.2
OKT+	66.7	83.3	49.8	66.7	83.3	49.8	78.4	82.1	70.2
SQKT	**93.4**	**89.2**	**88.4**	**90.3**	**87.1**	**84.9**	**93.3**	**88.4**	**89.8**

Table 2: Performance comparison of various models across three datasets. All values are in percentages.

any target problem, all the history U preceding that problem is used as the model input. Note that there is no risk of a student's history being exposed to testing, as students do not overlap between the training and test sets (see Table 7).

We conducted a grid search across a range of hyperparameters, including dropout rate, learning rate, batch size, and the weight for the triplet loss. The optimal hyperparameter values were chosen based on performance on the validation set. The best configuration obtained is as follows: a dropout rate of 0.1, a learning rate of 3e-5, a batch size of 16, and an auxiliary loss weight of 1.0.

The model is trained using the Adam optimizer on an NVIDIA A100 80GB PCIe GPU. The training times vary depending on the scenario: approximately 1 hour and 30 minutes for in-domain tasks and around 3 hours for cross-domain tasks.

5 Experiment Results

5.1 In-Domain Results

Across the three courses, SQKT consistently outperformed all baselines. SQKT achieved an AUC of 87.1–93.4, representing an absolute improvement of 12.6–20.8 compared to the best-performing baseline (KTMFF+). These results demonstrate that our SQKT model, which incorporates student questions, is highly effective in predicting students' future performance.

The improvement of KTMFF+ over KTMFF and OKT+ over OKT reinforces our research motivation that student questions provide valuable insights into student performance. It also suggests that our question embeddings can be integrated with general KT models to enhance their predictive accuracy. However, SQKT consistently outperformed these models, underscoring the efficacy of its architecture in leveraging student questions more effectively than the baselines.

Model	AUC (%)	ACC (%)	F1 (%)
SQKT	**93.4**	**89.2**	88.4
- Question (all-ones vector)	91.3	86.3	**89.9**
- Question (skill only)	90.9	86.2	88.7
- Skill (question only)	89.7	81.3	83.1
- Question and skill	85.4	80.7	82.7

Table 3: Ablation study on the "Python Introduction" course.

	Python Intro.			First Python			Python Basic		
	AUC	ACC	F1	AUC	ACC	F1	AUC	ACC	F1
SQKT	**92.3**	**86.3**	**87.3**	**93.0**	**87.1**	84.9	**93.3**	**88.4**	**89.8**
- Question	91.6	85.8	86.9	92.5	86.5	**86.9**	91.9	85.7	87.7
- Triplet	91.3	85.8	90.1	90.1	83.6	84.9	91.5	85.8	87.7

Table 4: Impact of response and triplet loss functions. All values are in percentages.

Ablation Study To evaluate the contribution of each component in SQKT, we conducted an ablation study. Basically, we explore removing question embeddings and skill embeddings both separately and together to assess their impact. Additionally, to examine the importance of the actual content of student questions, we replace the question embeddings with an all-ones vector to simply indicate the presence of a student question (potentially student confusion).

Table 3 presents the ablation results on the "Python Basic" course (the same pattern is observed in other courses). Using question indicators (row 2) reduces AUC and ACC, highlighting the importance of the actual content of student questions and its effective utilization. Relying solely on either skills (row 3) or questions (row 4) is suboptimal, demonstrating their synergistic contribution. Removing both questions and skills (row 5) significantly degrades the model's performance.

The results suggest that the superior performance of SQKT stems from the unique insights provided by student questions, such as their understanding of theoretical concepts and specific struggles, which are not always apparent in code submissions alone. Further, the additional step of explicitly identifying skills from their questions appears to further enhance the clarity of student performance.

Impact of Auxiliary Losses We analyzed the impact of the two auxiliary losses, i.e., question loss (Eq. 1) and triplet loss (Eq. 2). The results in Table 4 validate the importance of these additional objectives in improving performance across diverse

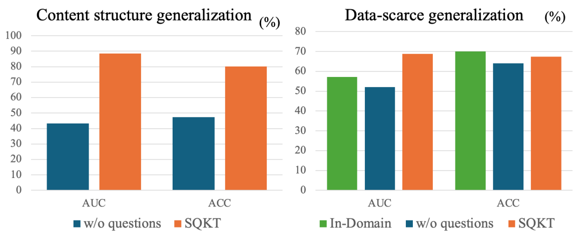

Figure 4: Cross-domain performance.

programming courses. The question loss, derived from the task of predicting educator responses to student questions, impacts performance across the three courses, with slight drops in performance when removed. This loss seems to enrich the embedding space by capturing important information in student questions better.

The triplet loss, designed to unify the embedding space for heterogeneous input features, has stronger impact, making a notable contribution especially for the "First Python" course. The triplet loss ensures effective integration of diverse data sources.

5.2 Cross-Domain Results

Figure 4 demonstrates the model's performance across two cross-domain settings, evaluating its ability to generalize to unseen courses.

In the setting of content structure generalization (Figure 4, left), we assessed SQKT's ability to transfer knowledge between courses with different levels and content structures. Our full model (orange) showed an absolute 45.3% improvement in AUC over without using question data (blue).

In the setting of data-scarce generalization (Figure 4, right), we trained SQKT on all courses except "Algorithm" and tested it on the course to evaluate the model's generalizability to higher difficulty levels and robustness in low-resource environments. Since the "Algorithm" course has small data, fine-tuning SQKT directly on the "Algorithms" data (green) shows an AUC score close to random. However, our full model (orange) showed a substantial improvement of 11.4% over the in-domain model (green) and 16.7% over the cross-domain model incorporating no student questions (blue).

Both experiments conclude that student questions convey generalizable insights into student performance across different courses and that leveraging them greatly enhances the model's ability to adapt to new courses with varying difficulty levels and limited data.

5.3 Error Analysis

To better understand our model, we conducted a detailed analysis focused on question-related mistakes. We randomly sampled 60 mispredictions from the test set, manually analyzed each data point by tagging one or more error types. Table 8 in Appendix presents a breakdown of these errors, their proportions, examples, and underlying reasons.

Our analysis shows that 'Complexity' is the most prevalent issue (55.6%), often due to code snippets containing mixed language syntax, which challenges the model's parsing capabilities. 'Confusion' is the second most common error type (40.7%), typically occurring when the error in the code is unrelated to the student's question, making it difficult for the model to establish the correct correlation. 'Ambiguity' (22.2%) and 'Incompleteness' (29.6%) also contribute significantly to model errors, emphasizing the need for clear, context-rich questions for accurate predictions. The analysis highlights key areas for improvement in the SQKT model. For example, incorporating more advanced natural language processing techniques to handle multi-lingual input could enhance the model's ability to interpret students' questions more accurately.

6 Conclusion

This paper introduces SQKT, a knowledge tracing model in programming education that addresses the unique challenges of predicting students' performance on subsequent problems in coding tasks. By integrating students' questions and auto-extracted skill information, SQKT provides a more comprehensive view of student knowledge than traditional KT models. We demonstrate the effectiveness of SQKT across various programming courses and difficulty levels, consistently outperforming baseline models in both in-domain and cross-domain settings. SQKT shows its ability to capture valuable information about students' programming competencies through their questions. We expect that our method can contribute to more personalized and effective learning interventions in programming education.

Limitations

This study has several limitations. First, we did not apply any preprocessing to the input questions prior to analysis. Although this approach more closely mirrors actual classroom conditions, inputs are often noisy. To address this, we implemented a skill extractor system designed to effectively extract information from such noisy inputs. Future research could explore whether introducing filtering or normalization steps might improve model performance.

Second, the skill extractor system employs a rule-based methodology rather than statistical machine learning techniques. This choice aims to ensure interpretability, offering a clear and explainable mapping between questions and skills. However, adopting machine learning methods could offer significant benefits, such as improved scalability and the ability to adapt to unseen patterns. Future studies could investigate hybrid approaches that combine the rule-based systems with machine learning models.

References

Amjad Altadmri and Neil CC Brown. 2015. 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.

Amal Asselman, Mohamed Khaldi, and Souhaib Aammou. 2020. Evaluating the impact of prior required scaffolding items on the improvement of student performance prediction. *Education and Information Technologies*, 25:3227–3249.

Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education*, pages 177–210.

Albert T Corbett and John R Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. In *User modeling and user-adapted interaction*, volume 4, pages 253–278. Springer.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Allen B Downey and Chris Mayfield, editors. 2019. *Think Java: How to think like a computer scientist*. O'Reilly Media.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Matthew Hertz and Sarah Michele Ford. 2013. Investigating factors of student learning in introductory courses. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 195–200.

Alison King. 1994. Guiding knowledge construction in the classroom: Effects of teaching children how to question and how to explain. *American educational research journal*, 31(2):338–368.

Naiming Liu, Zichao Wang, Richard Baraniuk, and Andrew Lan. 2022. Open-ended knowledge tracing for computer science education. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*.

Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. 2015. Deep knowledge tracing. *Advances in neural information processing systems*, 28.

Yang Shi, Min Chi, Tiffany Barnes, and Thomas Price. 2022. Code-dkt: A code-based knowledge tracing model for programming tasks. *arXiv preprint arXiv:2206.03545*.

Zhuoqing Song, Sirui Huang, and Ya Zhou. 2021. A deep knowledge tracking model integrating difficulty factors. In *The 2nd International Conference on Computing and Data Science*, pages 1–5.

Dan Sun, Fan Ouyang, Yan Li, and Caifeng Zhu. 2021. Comparing learners' knowledge, behaviors, and attitudes between two instructional modes of computer programming in secondary education. *International Journal of STEM Education*, 8:1–15.

Xia Sun, Xu Zhao, Bo Li, Yuan Ma, Richard Sutcliffe, and Jun Feng. 2022. Dynamic key-value memory networks with rich features for knowledge tracing. *IEEE Transactions on Cybernetics*.

Al Sweigart, editor. 2019. *Automate the boring stuff with Python: practical programming for total beginners*. No Starch Press.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Yongkang Xiao, Rong Xiao, Ning Huang, Yixin Hu, Huan Li, and Bo Sun. 2023. Knowledge tracing based on multi-feature fusion. *Neural Computing and Applications*, 35(2):1819–1833.

Liang Yu, Peng Tianhao, Pu Yanjun, and Wu Wenjun. 2022. Help-dkt: an interpretable cognitive model of how students learn programming based on deep knowledge tracing. In *Nature Scientific Reports*.

A Python Skill Set

Value	Variable Assign	Keywords	Operators	Operands
Type Convertor	input function	print function	Boolean Values	Boolean Expressions
Logical Operators	If-Else Statements	For Loops	While Loops	Break Statement
Continue Statement	Function Definitions	return Statement	Local, Global Scope	Strings
String Slicing	Indexing	Lists	Dictionaries	Import Statement
random	time	math	Opening files	Reading files
Writing files	Closing files	SyntaxError	NameError	TypeError
IndentationError	ValueError	AttributeError	IndexError	KeyError
TabError	UnicodeDecodeError	FileNotFoundError	ModuleNotFoundError	ZeroDivisionError
UnboundLocalError	ImportError	UnicodeEncodeError	LookupError	ConnectionError
RuntimeError				

Table 5: Categorized Python concepts and errors

B Description of dataset and train-validation-test split

Course Name	Description
Python Basic	Beginner-level course covering basic python concepts.
First Python	A beginner-friendly course focusing on fundamental Python programming.
Algorithm	An intermediate to advanced course on algorithms implemented in Python.
Python Introduction	A comprehensive course covering basic to intermediate python concepts.

Table 6: Description of dataset

Course Name	Type	Train	Validation	Test
Python Basic	# of students	128	16	16
	# of problems	2,665	362	412
First Python	# of students	6512	814	815
	# of problems	153,015	19,140	11,302
Algorithm	# of students	61	8	8
	# of problems	273	52	5
Python Introduction	# of students	873	109	110
	# of problemss	65,372	4,690	4,135

Table 7: Statistics for dataset splits

C Error Analysis

Error Type	Occurrence	Questions	Reason
Ambiguity	22.2%	Traceback (most recent call last): File "main.py", line 4, in ⟨module⟩ pr NameError: name 'pr' is not defined *"please show me the correct answer."*	The meaning of the question lacks context, allowing for various interpretations
Confusion	40.7%	Traceback (most recent call last): File "main.py", line 5, in ⟨module⟩ for key in len(int(data)) TypeError: int() argument must be a string, a bytes-like object or a number, not 'dict' *"If len(scores), does it include up to scores?* *If len(scores), shouldn't it be scores+1 to include the last score?"*	When the error message is unrelated to the student's question, making it difficult to easily determine their correlation, causing the model to be confused about what to focus on within the question
Incompleteness	29.6%	*"Please provide the answer"*	The question is too simple and lacks necessary information
Complexity	55.6%	File "main.py", line 5 init(self, "***sweet potato(Korean)***"): SyntaxError: invalid syntax *"please explain this error"*	The question contains a code snippet with a mix of Korean characters and English syntax. This combination of different character sets and languages introduces additional complexity

Table 8: Detailed error analysis of SQKT

D Dataset Example

D.1 Python Basic - 1

Problem	Data	Example
Alice the Rabbit's Math Homework	Problem Description	Write a program that takes a natural number as input and outputs the difference between the square of the sum and the sum of the squares for numbers from 1 to the given input.

	Problem Solution	`N = int(input())`
		`i_square = 0`
		`i_list = list(range(1, N + 1))`
		`for i in i_list:`
		`    i_square += i**2`
		`    i += 1`
		`sum_square = sum(i_list)**2`
		`print(sum_square - i_square)`

	Student's code submission	`summation = 0`
		`while num > 0:`
		`    summation = summation + 1`
		`    num = num - 1`
		`print(summation)`

	Student's question	Why does the summation variable not produce the correct summation when printed in the given code?
	Skill	While-loop, Print function, Operator
	Educator's response	Since the while loop increments summation by 1 in each iteration, if you input 10, the final value stored in summation will be 10.

Table 9: Example of Python Basic dataset - 1

D.2 Python Basic -2

Problem	Data	Example
Script Polishing	Problem Description	The variable 'sentence' contains a randomly generated even-length sentence read by the Mad Hatter. Prompt the user to input a special character to insert into the middle of the sentence. Then, insert the inputted special character into the middle of the string 'sentence' and save the result.

	Problem Solution	`sentence =`
		`sentence[: len(sentence) // 2] + input() + sentence[len(sentence) // 2`
		`:]`

	Student's Code Submission	`st_len1 = sentence[x:]`
		`st_len2 = sentence[:x]`
		`add_st = str(input())`
		`sentence = st_len1 + add_st + st_len2`

	Student's Question	Can't it be done using only parentheses? Is it better to use square brackets for distinction? Square brackets are used for index slicing. Also, how can I insert a string into the middle of another string?
	Skill	String, Operators, Indexing
	Educator's response	Parentheses do not function the same way.

Table 10: Example of Python Basic dataset - 2

D.3 Python Introduction - 1

Problem	Data	Example
Copycat Parrot	Problem Description	If you’ve entered the code on line 02, click [Run]. Do you see the cursor blinking in the output window? Type anything you want to say in this area, then press [Enter].
	Problem Solution	`var = input()` `print(’Parrot:’, var)`
	Student’s Code Submission	`var = raw_input(”Enter a value:”)` `print(’Parrot:’, var)`
	Student’s Question	var = input(”) print(’Parrot:’, var) What should I enter in the input(”) function?
	Skill	Variable Assign, Operands, input function, print function
	Educator’s response	Please enclose the string "Parrot" in quotation marks when entering it. For example, you can input it as follows: var = input(’Parrot’)

Table 11: Example of Python Introduction dataset - 1

D.4 Python Introduction - 2

Problem	Data	Example
Creating a Mysterious Data Dictionary	Problem Description	Enter your age as a number inside the parentheses. Enter your name as a string inside the parentheses. Enter a list containing your age and name inside the parentheses.

	Problem Solution	`print(20)`
		`print("Your Name")`
		`print([20, "Your Name"])`

	Student's Code Submission	`print(17)`
		`print("Your Name")`
		`print[17, James Bond]`

	Student's Question	`Traceback (most recent call last):`
		`File "main.py", line 8, in <module>`
		`print[17, James Bond]`
		`NameError: name 'James Bond' is not defined`

	Skill	Value, NameError]
	Educator's response	This error indicates that the variable `James Bond` has not been defined. Variables must be defined before they are used. To fix this, the line `print[17, James Bond]` needs to be corrected by defining `James Bond` as a string.

Table 12: Example of Python Introduction dataset - 2

D.5 First Python - 1

Problem	Data	Example
If Else Statements	Problem Description	Write the if condition on line 4 so that it evaluates to true when the entered password `password` matches the set password `answer`. Run the program and try entering the password `34566`.

	Problem Solution	`answer = 12345`
		`password = input("Enter the password: ")`
		`if password == answer:` print("Password OK!")
		`else: print("Password Not OK!")`

	Student's Code Submission	`answer = '12345'`
		`password = input('Enter the password: ')`
		`if answer==password: print('Password OK!')`
		`else: print('Password Not OK!')`

	Student's Quesion	If I input '12345' in the terminal, it shows incorrect. But if I input 12345, it shows correct. Why is that? Can't I set answer = 12345 instead? If answer = '12345', do I need to type 12345 in the terminal for it to match?
	Skill	input function, Variable
	Educator's response	The symbols " are used to represent a string. Since input values in the terminal are automatically processed as strings, there is no need to include " when typing input in the terminal.

Table 13: Example of First Python dataset - 1

D.6 First Python - 2

Problem	Data	Example
Prime Number Finder	Problem Description	Write a program to find prime numbers between `1` and `N`, where `N` is an input value. Run the program and enter `200` as the value of `N`.

	Problem Solution	`n = int(input("Enter the value of N: "))`
		`for a in range(2, n+1):`
		`  prime_yes = True`
		`  for i in range(2, a):`
		`    if a % i == 0: prime_yes = False break`
		`  if prime_yes:print(a, end=" ")`

	Student's Code Submission	`n = int(input("Enter the value of N: "))`
		`for a in range(2, n+1):`
		`  result = True`
		`  for i in range(2, a):`
		`    if a % i == 0: result = False`
		`    break`
		`  if result = True: print(a, end=" ")`

	Student's Question	In this part, doesn't 'if result:' mean the same as 'if result = True'? Why does it cause an error when I write 'if result = True'?
	Skill	If-Else Statements, Boolean Values
	Educator's response	You need to use ==.

Table 14: Example of First Python dataset - 2