ECKT: Enhancing Code Knowledge Tracing via Large Language Models

Yang Yu **Yingbo Zhou** **Yaokang Zhu** **Yutong Ye**

Liangyu Chen **Mingsong Chen**

Shanghai Institute of Artificial Intelligence for Education, East China Normal University, Shanghai, China
MoE Engineering Research Center of SW/HW Co-Design Tech. and App., East China Normal University, Shanghai, China

Abstract

Code Knowledge Tracing (CKT) aims to model students' programming proficiency from their coding activities. Existing approaches mainly rely on answer records and lack problem descriptions and knowledge concepts, thus fail to capture the inherent information. To solve this problem, we propose **ECKT**, an **E**nhanced **C**ode **K**nowledge **T**racing framework using Large Language Models (LLMs), which simulates human cognitive process through chain-of-thought prompting and adapts quickly to new tasks with limited data using few-shot learning. Specifically, ECKT generates detailed problem descriptions and knowledge concepts from student code, enhancing the model's understanding of programming concepts and proficiency. Additionally, ECKT incorporates task difficulty information by correlating problems with difficulty levels based on student performance scores. This integration allows for a more accurate assessment of student proficiency across varying levels of difficulty. Also, ECKT can explicitly capture the essence of code and learn a better representation. Experimental results demonstrate that ECKT effectively improves the performance of knowledge tracing in programming education. This advancement not only supports personalized learning but also contributes to a deeper understanding of coding activities.

Keywords: artificial intelligence; education; skill acquisition and learning; programming; large language models

Introduction

With the popularization of computer science education, programming skills have become an integral part of the modern education system. In the realm of educational technology, Knowledge Tracing (KT) serves as a crucial approach for predicting student learning outcomes by analyzing their interactions with educational content (Abdelrahman, Wang, & Nunes, 2023). This approach has been impactful in programming education, where Code Knowledge Tracing (CKT) is developed to assess students' coding proficiency. It identifies students' strengths and weaknesses in programming. Code knowledge tracing provides a new perspective to dynamically track and predict students' learning states by analyzing their behavioral data in solving programming problems.

The existing methods can be broadly categorized into two groups: traditional KT models and advanced KT models that incorporate additional features. Traditional KT models, such as Deep Knowledge Tracing (DKT) (Piech et al., 2015), primarily focus on the correctness of student responses to problems. These models often overlook the content of the student's code, which can provide valuable insights into their understanding of programming concepts. Advanced KT models, such as Code-DKT (Shi, Chi, Barnes, & Price, 2022) and the approach proposed in (Zhu, Han, Yuan, & Lu, 2022), seek to address these limitations by incorporating code analysis into the KT framework. However, the field is confronted with challenges such as the absence of rich textual information from problems, the complexity of code representation, and limitations in the adaptability of models to diverse student abilities. The current CKT models, such as Code-DKT, have made significant strides but still face several issues. These models often fail to fully leverage the textual context of problems, which is crucial for understanding the student's problem-solving process. Additionally, the representation of code is a complex task, as traditional methods may not effectively capture the hierarchical structure and semantic meaning of code. Furthermore, the learning capacity of these models is constrained, which limits their ability to generalize across different programming tasks and student skill levels.

To address these issues, we propose an innovative framework called Enhanced Code Knowledge Tracing (ECKT). ECKT begins by leveraging Large Language Models (LLMs) to generate problem descriptions and knowledge concepts from student code responses through chain-of-thought (CoT) and few-shot learning. Then, we employ BERT to convert these knowledge concept texts into vector embeddings. These embeddings serve as additional features for the model, enhancing its ability to capture the semantic information of programming concepts. In addition, we introduce a method to rank problems by difficulty based on student performance, associating each problem ID with its corresponding difficulty level. This allows the model to learn the varying difficulty levels of different problems, offering a more detailed analysis of student performance. Finally, we utilize a stacked GRU architecture to improve the model's sequence learning capabilities, enabling it to better understand the progression of student skills over time.

Our work makes the following major contributions:

- Leveraging large language models, ECKT generates problem descriptions and knowledge concepts from student code through chain-of-thought and few-shot learning.
- The framework establishes a correlation between problems and their difficulty levels based on student scores, allowing the model to account for the varying difficulty levels of different problems.

In L. K. Samuelson, S. L. Frank, M. Toneva, A. Mackey, & E. Hazeltine (Eds.), *Proceedings of the 46th Annual Conference of the Cognitive Science Society.*

- A stacked GRU architecture is employed to improve the model's sequence learning capabilities, enabling it to better track the progression of student skills over time.

By integrating these enhancements, the proposed framework ECKT aims to provide a more comprehensive and accurate assessment of student programming proficiency. This framework not only addresses the existing challenges but also finds a new way for the integration of advanced AI techniques in educational analytics. Experimental results show ECKT outperforms traditional models, demonstraing its effectiveness in code knowledge tracing.

Related Work

Code Knowledge Tracing

In the field of educational data mining, code knowledge tracing has taken on a significant role, focusing on the assessment of students' understanding of programming concepts. Traditional knowledge tracing models, such as Bayesian Knowledge Tracing (BKT) (Bulut, Shin, Yildirim-Erbasli, Gorgun, & Pardos, 2023), Factor Analysis (FA) (Chi, Koedinger, Gordon, Jordan, & VanLehn, 2011), and Item Response Theory (IRT) (Su et al., 2021), have laid the groundwork for this field, employing probabilistic models to track the acquisition of knowledge over time. For instance, BKT estimates mastery through parameters like initial knowledge probability and acquisition rates, while FA incorporates instructional factors, and IRT relates student ability to problem difficulty.

Deep learning has been introduced innovative models such as DKT (Piech et al., 2015), enhancing the predictive capabilities of traditional knowledge tracing approaches. DKT employs RNNs to convert past performance into a time series, predicting the likelihood of solving subsequent problems. (Tong, Zhou, & Wang, 2020) Calculate the similarity of the text vector of the problem and use hierarchical clustering methods to extract the semantic features of the problem. Then, the extracted features and student responses are input into the RNN structure to predict the student's response results (Russin, Jo, O'Reilly, & Bengio, 2020). The Code-DKT model improves the predictive performance of the model for programming tasks by combining Deep Knowledge Tracing (DKT) (Piech et al., 2015) and code2vec models to extract features of the code. Code2Vec learns distributed representations of code for code analysis, derived from Word2Vec, which is a popular word embedding model. These embeddings capture the semantic relationships between code elements, providing a rich feature representation for downstream tasks (Compton, Frank, Patros, & Koay, 2020). ASTs serve as a fundamental representation of the syntactic structure of source code. They have been extensively used to capture the hierarchical feature of code (Wang, Li, Ma, Xia, & Jin, 2020). ASTs enable the extraction of syntactic features that can be leveraged to understand the structure and logic of code. AST analysis methods extract code features by parsing the structure of the code. These features can include control flow structure and data flow analysis, which helps to understand the structural information of the code.

Large Language Models

The advent of large language models (LLMs) and BERT has changed the way of natural language processing and understanding (Chuang, Hubbard, & Austerweil, 2020)(Liu et al., 2023). These models have demonstrated remarkable capabilities in capturing the nuances of language, which can be leveraged to enhance the performance of knowledge tracing systems in programming education (Kasneci et al., 2023). LLMs, such as GPT-3 and BERT, have been employed in various educational applications to generate coherent and contextually relevant text (Floridi & Chiriatti, 2020). In the context of programming, these models can be utilized to create detailed problem descriptions from student code snippets, providing a richer source of information for knowledge tracing models. Few-shot learning allows models to rapidly learn new tasks from a small number of examples, effectively utilizing their extensive pre-existing knowledge (Perez, Kiela, & Cho, 2021). This technique is particularly advantageous in domains where extensive data is not readily available. Chain of Thought (COT) prompting enhances the reasoning capabilities of large language models by encouraging them to mimic human-like thought processes (Truzzi & Cusack, 2020). This approach not only strengthens the models' ability to solve problems but also increases their interpretability, providing a clearer understanding of the reasoning steps (Wei et al., 2022). BERT's ability to understand the context of words within a sentence has been harnessed to transform knowledge concept texts into vector representations (Liang, Cao, Zheng, Ren, & Gao, 2021). These embeddings serve as a powerful feature set for knowledge tracing models, enhancing their ability to predict student performance by capturing the semantic relationships between programming concepts. Studies have shown that language models acquire a significant amount of world knowledge from the vast text corpora they train (Petroni et al., 2019). Some methods generate feature vectors of problem text using BERT to obtain the knowledge distribution and difficulty characteristics of the problem. Traditional knowledge tracing algorithms typically use one or several features to predict student behavior without considering the potential relationships between these features, which may limit and ignore important information within the features. MLFBK (Li, Jacobsen, Shi, Zhou, & Wang, 2023) is a BERT-based knowledge tracing method that utilizes multiple features and explores potential relationships between them to improve the performance of knowledge tracing models.

Methodology

Problem Definition

The Code Knowledge Tracing (CKT) task is designed to assess the extent to which students have mastered programming skills by analyzing their programming submissions. This task typically involves constructing models to predict how stu-

dents will perform in future attempts and to track their mastery of specific programming concepts (Portelance, Degen, & Frank, 2020). In programming education, CKT models can help teachers provide personalized feedback to guide student learning more effectively.

A student's interaction within a programming exercise typically includes information about the code submitted by the student, the ID of the exercise, and the correctness of the code. The code knowledge tracing model aims to predict students' ability to solve new programming tasks by analyzing their historical programming practices. The task is to predict students' subsequent performance based on their historical response information (i.e., correct or incorrect answers). The process is defined as follows.

$$\begin{aligned} D &= \{S_1, S_2, \ldots, S_N\}, \\ S_i &= \{x_{i1}, x_{i2}, \ldots, x_{it}, \ldots, x_{iM}\}, \\ x_{it} &= \{q_{it}, a_{it}, c_{it}\}, \\ P_{i_{t+1}} &= M(x_{it}), \end{aligned} \tag{1}$$

where D represents the collection of students, and S_i denotes the series of responses from an individual student. At a given timestamp t, q_{it} specifies the problem number, a_{it} reflects the correctness of the answer (with 0 indicating an incorrect response and 1 indicating a correct response), and c_{it} is the code submitted. $P_{i_{t+1}}$ represents the model's prediction of a student's accuracy on a subsequent attempt.

Model Overview

As shown in Figure 1, our method first employs a large language model to generate problem descriptions and knowledge concepts from student code submissions, leveraging chain-of-thought and few-shot learning. This process enhances the model's ability to simulate human reasoning and adapt to diverse coding tasks. Subsequently, BERT is used to embed these knowledge concepts into vectors, augmenting the model with a deeper understanding of the educational content. Then, we derive difficulty levels from scoring distributions. In parallel, we utilize attention mechanisms to obtain code embeddings from the Abstract Syntax Tree (AST), identifying key paths that represent the essence of the student's code. These embeddings, along with the knowledge and difficulty vectors, are fused and send to a stack of Gated Recurrent Units (SGRU) for the final prediction.

The Structure of ECKT

Code representation. To capture the deep structural and semantic information of the student code, We represent the code by utilizing the Abstract Syntax Tree (AST) (Shi et al., 2022). AST is a tree structure where each node represents a syntactic element in the code, such as a variable, a function, a loop, and so on. We first convert student code into an AST. Then, by traversing the AST, we can extract key paths in the code that reflect the structures and concepts used by the students while programming. For example, the path of a loop may reveal how well a student understands loop control.

Figure 2 illustrates a simplified AST of the add function, containing the argument list [a, b] and the function body. The function body consists of a return node that contains a binary operation node, the addition operation. The addition operation node has two child nodes: the left node and the right node, which are the operands a and b, respectively. To convert the AST structure into a form that can be handled by deep learning models, we use the code2vec model to extract the code path (Compton et al., 2020). Each leaf node is encoded as a vector, and the leaf-to-leaf path is also encoded as a vector. A path consists of three parts: the start node, the path, and the end node.

A piece of student code contains a series of paths $P = \{p_1, \ldots, p_k\}$. One of the paths is $p_i = \langle s_i, r_i, d_i \rangle$. s_i and d_i are nodes, which are embedded by the node embedding matrix W_{n}. r_i is the path connecting two nodes, which is embedded by the path embedding matrix W_{p}. The values of the two matrices are randomly initialized and updated during the training process. For knowledge tracing, the problem-answer vectors are important features, so the path p_i is fused with the problem-answer vectors and then embedded by the embedding matrix:

$$\begin{aligned} \mathbf{e}_{n_i} &= \left[W_n\left(\mathbf{s_i}, \mathbf{r_i}, \mathbf{d_i}\right), \mathbf{x_t}\right], \\ \mathbf{e}_{p_i} &= \left[W_p\left(\mathbf{s_i}, \mathbf{r_i}, \mathbf{d_i}\right), \mathbf{x_t}\right], \end{aligned} \tag{2}$$

where $\mathbf{s_i}$, $\mathbf{r_i}$, $\mathbf{d_i}$, and $\mathbf{x_t}$ are the embedded vectors for the source node, path, destination node, and problem-answer vectors, respectively.

In order to determine which AST paths are most important for the knowledge tracing model, we introduce an attention mechanism. The attention weight α_i is used to measure the contribution of each path to the model predictions.

An AST generated by student code contains a large number of paths, each of which has a different importance. The nodes and connections in each path also have different importance, so it is necessary to learn how to combine these parts. We introduce a fully connected layer W that gives different paths different importance. This gives more attention to key paths and less attention to common paths. Based on (Compton et al., 2020), the formula is as follows:

$$p_i = \tanh(W c_i), \tag{3}$$

$$\alpha_i = \frac{e^{(\mathbf{p}_i^T \cdot \mathbf{a})}}{\sum_{j=1}^{k} e^{(\mathbf{p}_j^T \cdot \mathbf{a})}}, \tag{4}$$

where W is the learnable weight matrix, tanh is the hyperbolic tangent function. α_i is the computed importance of the paths and the attention vector a is randomly initialized and learned while the network is being trained. The path importance scores are then multiplied by the path vectors to generate the summed importance of all paths. This is followed by a fully connected layer to get the final code representation, which is computed as follows:

$$\mathbf{v} = W_c \sum_{j=1}^{k} \alpha_i e_i. \tag{5}$$

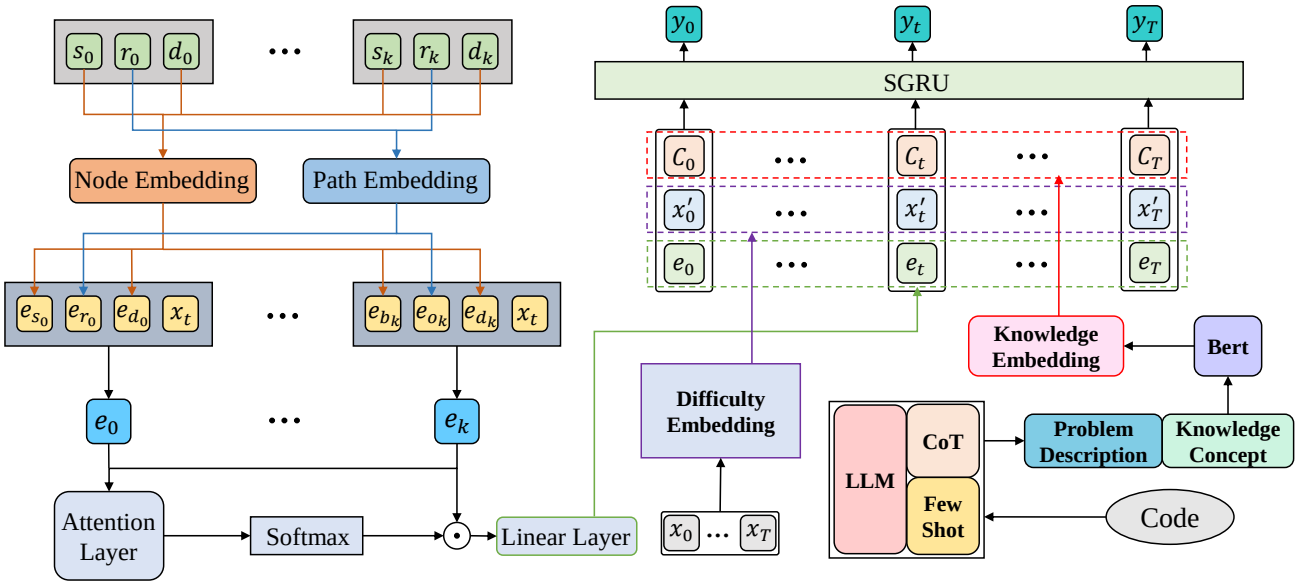

Figure 1: The framework of ECKT model.

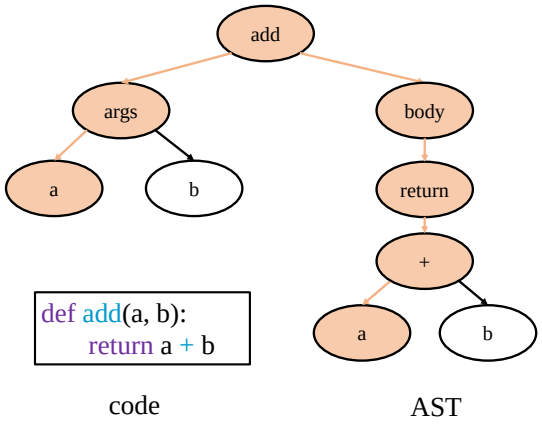

Figure 2: An example of an abstract syntax tree where orange-colored nodes and edges form a path.

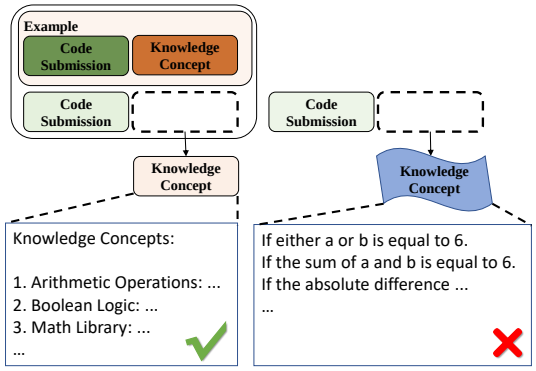

Figure 3: An example of prompt-based few-shot learning.

Chain-of-thought and few-shot learning for LLM-based content generation. We utilize large-scale language models to generate problem descriptions and knowledge concepts that correspond to code submissions. Our approach begins with the generation of problem descriptions which outline the task's high-level requirements and imply the necessary knowledge concepts. Subsequently, we apply chain-of-thought (CoT) reasoning, which involves breaking down the task into a series of logical steps, mirroring the human problem-solving process (Wei et al., 2022). This step-by-step analysis includes identifying inputs, processing steps, expected outputs, and error-handling mechanisms. For each step in the CoT, identify relevant programming elements and generate knowledge concepts that explain these elements. This process is iteratively refined, ensuring that the knowledge concepts are comprehensive and accurate. The chain-of-thoughts prompting allows our model to simulate the human cognitive process, explicitly reasoning through intermediate steps to reach a final result (Edmonds, 2019). This method is particularly effective in enhancing the model's comprehension of the transition from code to knowledge concepts, as it generates explanatory steps that clarify the underlying logic. On the other hand, prompt-based few-shot learning allows our model to swiftly adapt to new tasks with minimal data (Lake, Linzen, & Baroni, 2019). This approach leverages the model's existing knowledge to learn from a small set of examples. It utilizes an additional code submission along with its corresponding knowledge concepts as an example, as shown in Figure 3. This approach is advantageous in educational and programming contexts where large annotated datasets are scarce, allowing for efficient learning from limited resources.

BERT embedding with feature enhancement. BERT model can generate high-quality text vectors that capture

the contextual information and semantic structure of the text (Misra, Ettinger, & Rayz, 2020). We use the BERT model to embed these knowledge concept texts(Devlin, Chang, Lee, & Toutanova, 2019). The BERT model can capture the deep semantic information in the text and convert it into a fixed-length vector representation.

Problem difficulty correlation and embedding. To track students' knowledge states more accurately, we introduce the concept of problem difficulty. We first determine the difficulty of a problem based on the performance scores of code submissions, and then associate the problem ID with the problem difficulty. We define a difficulty estimation function $f_{\text{Dif}}(q)$, which estimates the difficulty of a problem based on the score of the students' answers. This function can be expressed as:

$$f_{\text{Dif}}(q) = \sigma\left(\frac{1}{\sum_{n=1}^{N}\sum_{t=1}^{T}} \sum_{n=1}^{N}\sum_{t=1}^{T} \text{score}(q_{nt})\right), \tag{6}$$

where t is the attempt that the student made to solve the problem, N is the number of students who attempted the problem, $\text{score}(q_{nt})$ is the score of student n on problem q, and σ is the sort function.

We correlate problem IDs with their estimated difficulties to embed difficulty information in the model. This is achieved by incorporating a score-based ranking of the problems into the model's input features.

Stacked gated recurrent unit. To improve the deep learning capability of the model, we adopt a stacked gated recurrent unit (GRU) (Bukhari et al., 2024). Stacked GRU enhances the model's ability to capture long-term dependencies by adding additional GRU layers to the original GRU(Aurnhammer & Frank, 2019).

Our ECKT model receives a sequence of student code submissions $S = \{(q_1, a_1, c_1, kv_1), \ldots, (q_T, a_T, c_T, kv_T)\}$ as inputs, where q_t is the problem ID, a_t is the correctness of the attempt, c_t is the code submission, and kv_t is the knowledge concept. The output $Y = \{y_1, y_2, \ldots, y_T\}$ of the model is a sequence of probabilities that the student will submit correctly in the next attempt. The loss function is defined as:

$$L = -\sum_{t}\left(a_t \log(\hat{y}_t) + (1 - a_t)\log(1 - \hat{y}_t)\right), \tag{7}$$

where a_t is the correctness of the tth student attempt and $\hat{y}$ is the model prediction.

Experiments

To validate the effectiveness of our approach, we designed a series of experiments to evaluate the performance of the model. These experiments are designed to compare our proposed model with existing knowledge tracing models. The primary goal is to analyze the impact of our approach on model performance. Specifically, we focus on how our model enhances the understanding of programming tasks and predicts student performance in problem-solving.

Table 1: Answer data content description.

Field name	Field Description
ProblemID	Problem Number
CourseID	ID of the course
SubjectID	ID of the student
EventType	Running state of program
CompileMessageType	Compile information of Program
Score	Score of the attempt

Experimental Settings

Dataset. The dataset used in this study originated from a programming course at a large university in the United States (Shi et al., 2022). These data record students' attempts at solving programming assignments, including the code they submitted, the correctness of each attempt, and the problem ID. The dataset consists of 50 programming problems tackled by 410 students, distributed across 5 assignments. Each student in the dataset made multiple attempts at each problem until it was successfully solved.

Baselines. We set up multiple experimental groups, each using a different model configuration. These configurations included different models and settings. We also compare model performance using different enhancements, including a baseline model using only problem IDs and correctness labels and an augmented model with the addition of knowledge concept embedding vectors of generated problem knowledge concepts. We first reproduce the BKT (Bulut et al., 2023), DKT (Piech et al., 2015), and Code-DKT (Shi et al., 2022) models as a baseline and introduce improvements based on them. To ensure the comparability of the experiments, we adopt the same dataset partitioning strategy as Code-DKT, i.e., 80% of the data is used for training, and the remaining is used for testing.

Evaluation metrics. We utilize Area Under the Curve (AUC) rather than accuracy (ACC) due to its robustness against class imbalances. For instance, in a dataset where 90% of submissions are incorrect, a model that always predicts incorrect would achieve high ACC, yet AUC would expose its ineffectiveness. We evaluate model performance using two metrics, i.e., all attempts and first attempt. All attempts measure the student's cumulative performance across all submissions. In contrast, first attempt assesses the student's initial problem-solving ability.

Performance Comparison

The experimental results show that our proposed model outperforms the baseline models on several evaluation metrics. In particular, the model's performance in predicting students' programming proficiency is enhanced by incorporating the generated knowledge concept embeddings and difficulty embeddings. This augmentation leads to improvements in the model prediction performance, particularly in assessing the students' problem-solving capabilities.

The comparative analysis across all assignments, as de-

Table 2: Performance Comparison on all assignments.

Model	DKT	Code-DKT	ECKT
A1	71.26%	74.32%	**76.53%**
A2	73.17%	76.54%	**77.09%**
A3	76.86%	80.23%	**80.47%**
A4	69.04%	72.71%	**74.14%**
A5	75.25%	79.22%	**79.53%**

Table 3: All and the first attempt performance of all models on assignment A1.

Model	Overall	First Attempt
ECKT	**76.53%**	**80.01**%
Code-DKT	74.32%	75.71%
DKT	71.26%	72.01%
BKT	63.50%	50.06%

tailed in Table 2, demonstrates the robustness of the ECKT framework. ECKT consistently outperforms both DKT and Code-DKT across all assignments, indicating its effectiveness in understanding the subtle differences in student programming proficiency. This improvement is particularly notable in assignments A1 and A4, where ECKT achieves the highest AUC scores, suggesting that the integration of our proposed enhancements contributes significantly to the model's predictive performance.

In Table 3, the performance for all attempts and first attempt on Assignment A1 highlights the importance of ECKT's ability to provide early and accurate predictions. ECKT's higher AUC scores for both overall and first attempt indicate a superior ability to predict student performance. This enhanced understanding of problem difficulty and knowledge concepts is crucial for timely intervention and personalized learning support.

Performance Analysis on Specific Problems

The detailed AUC performance on individual problems within Assignment A1, as presented in Table 4, shows ECKT's adaptability to various programming challenges.

Table 4: AUC performance of Code-DKT, DKT and ECKT on different problems on assignment A1.

Problems	Code-DKT		DKT		ECKT	
	Overall	First	Overall	First	Overall	First
234	**64.60%**	**71.38%**	63.75%	73.48%	59.95%	70.10%
13	78.45%	86.55%	63.59%	68.81%	72.37%	**87.59%**
232	**74.93%**	**78.99%**	72.49%	73.09%	72.01%	78.02%
233	64.79%	74.57%	67.18%	76.33%	**70.52%**	**83.26%**
5	75.38%	81.34%	74.28%	81.79%	**78.77%**	**86.10%**
235	70.65%	71.96%	75.03%	70.80%	**71.19%**	**78.49%**
236	74.25%	74.30%	**78.68%**	**77.06%**	70.64%	74.22%
1	68.62%	70.32%	66.67%	73.20%	**78.57%**	**73.63%**
3	71.00%	71.00%	64.02%	64.02%	**79.32%**	**81.76%**

Table 5: Ablation study of ECKT on assignment A1.

Model	Overall	First Attempt
ECKT	**76.53%**	**80.01%**
ECKT-dg	74.15%	77.16%
ECKT-df	75.04%	79.19%
ECKT-kv	75.15%	79.64%
ECKT-df-kv	75.31%	79.41%
ECKT-dg-kv	76.18%	**80.01%**
ECKT-dg-df	75.36%	78.93%

ECKT's superior performance indicates that the combination of LLM-generated problem texts, BERT embeddings, and difficulty-aware features allows ECKT to effectively trace knowledge in diverse coding contexts.

Ablation Study

In our ablation study, we denote the ECKT framework's variations as follows: dg signifies stacked gated recurrent units, df indicates difficulty embeddings, and kv represents problem description and knowledge concept embeddings derived from the large language model. The hyphen (-) in model names indicates the exclusive presence of the specified module. The ablation study on Assignment A1 evaluates the contributions of various components within the ECKT framework. The full ECKT model achieves the highest AUC at 76.53%. While ECKT-dg (SGRU only) yields a lower AUC of 74.15%, emphasizing the importance of LLM-generated content and BERT embeddings. ECKT-df (difficulty embedding only) achieves an AUC of 75.04%, highlighting the value of difficulty information. ECKT-kv (knowledge concept embeddings only) reaches an AUC of 75.15%, underscoring the significance of textual knowledge representation. ECKT-df-kv results in an AUC of 75.31%, demonstrating the added benefit of combining two components. ECKT-dg-df, with stacked GRU and difficulty embeddings, reaches an AUC of 75.36%, emphasizing the need for knowledge concept embeddings.

Conclusion

In this work, we propose Enhanced Code Knowledge Tracing (ECKT), which addresses the limitations of traditional CKT models by incorporating several contributions. ECKT leverages large language models to generate problem descriptions and knowledge concepts from student code, utilizing chain-of-thought (CoT) and few-shot learning. These contents are embedded using BERT, providing the model with a more comprehensive understanding. Furthermore, ECKT associates difficulty with problems, allowing for a more detailed assessment of student proficiency. The integration of a stacked GRU enhances the model's ability to capture the temporal dynamics of student interactions. Experimental results show that ECKT outperforms baseline models, offering a more effective method for programming education.

Acknowledgments

This work was supported by the Natural Science Foundation of China (62272170), and the Shanghai International Joint Lab of Trustworthy Intelligent Software (22510750100). Mingsong Chen (mschen@sei.ecnu.edu.cn) and Liangyu Chen (lychen@sei.ecnu.edu.cn) are the corresponding authors.

References

Abdelrahman, G., Wang, Q., & Nunes, B. (2023). Knowledge tracing: A survey. *ACM Computing Surveys*, *55*(11), 1–37.

Aurnhammer, C., & Frank, S. (2019). Comparing gated and simple recurrent neural network architectures as models of human sentence processing. In *Proceedings of the 41th annual meeting of the cognitive science society (CogSci)* (pp. 112–118).

Bukhari, S. M. S., Moosavi, S. K. R., Zafar, M. H., Mansoor, M., Mohyuddin, H., Ullah, S. S., ... Sanfilippo, F. (2024). Federated transfer learning with orchard-optimized convsgru: A novel approach to secure and accurate photovoltaic power forecasting. *Renewable Energy Focus*, *48*, 100520.

Bulut, O., Shin, J., Yildirim-Erbasli, S. N., Gorgun, G., & Pardos, Z. A. (2023). An introduction to bayesian knowledge tracing with pybkt. *Psych*, *5*(3), 770–786.

Chi, M., Koedinger, K. R., Gordon, G. J., Jordan, P. W., & VanLehn, K. (2011). Instructional factors analysis: A cognitive model for multiple instructional interventions. In *Proceedings of the 4th international conference on educational data mining(EDM)* (pp. 61–70). www.educationaldatamining.org.

Chuang, Y.-S., Hubbard, E., & Austerweil, J. L. (2020). The" fraction sense" emerges from a deep convolutional neural network. In *Proceedings of the 42th annual meeting of the cognitive science society* (pp. 1207–1213).

Compton, R., Frank, E., Patros, P., & Koay, A. (2020). Embedding java classes with code2vec: Improvements from variable obfuscation. In *Proceedings of the 17th international conference on mining software repositories (MSR)* (pp. 243–253). ACM.

Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the north american chapter of the association for computational linguistics: Human language technologies, NAACL-HLT* (pp. 4171–4186). Association for Computational Linguistics.

Edmonds, M. (2019). Decomposing human causal leanring: bottom-up associative learning and top-down schema reasoning. In *Proceedings of the annual meeting of the cognitive science society (CogSci)*.

Floridi, L., & Chiriatti, M. (2020). Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, *30*, 681–694.

Kasneci, E., Seßler, K., Küchemann, S., Bannert, M., Dementieva, D., Fischer, F., ... others (2023). Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and individual differences*, *103*, 102274.

Lake, B. M., Linzen, T., & Baroni, M. (2019). Human few-shot learning of compositional instructions. In *Proceedings of the annual meeting of the cognitive science society (CogSci)*.

Li, Z., Jacobsen, M., Shi, L., Zhou, Y., & Wang, J. (2023). Broader and deeper: A multi-features with latent relations BERT knowledge tracing model. In *Proceedings of the 18th european conference on technology enhanced learning (EC-TEL)* (Vol. 14200, pp. 183–197). Aveiro, Portugal: Springer.

Liang, Y., Cao, R., Zheng, J., Ren, J., & Gao, L. (2021). Learning to remove: Towards isotropic pre-trained bert embedding. In *Proceedings of the 30th international conference on artificial neural networks (ICANN)* (pp. 448–459). Springer.

Liu, Y., Han, T., Ma, S., Zhang, J., Yang, Y., Tian, J., ... others (2023). Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-Radiology*, 100017.

Misra, K., Ettinger, A., & Rayz, J. (2020). Exploring lexical relations in bert using semantic priming. In *Proceedings of the annual meeting of the cognitive science society (CogSci)* (p. 1939).

Perez, E., Kiela, D., & Cho, K. (2021). True few-shot learning with language models. In *Proceedings of the 35th annual conference on neural information processing systems (NeurIPS)* (pp. 11054–11070).

Petroni, F., Rocktäschel, T., Riedel, S., Lewis, P. S. H., Bakhtin, A., Wu, Y., & Miller, A. H. (2019). Language models as knowledge bases? In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)* (pp. 2463–2473). Association for Computational Linguistics.

Piech, C., Bassen, J., Huang, J., Ganguli, S., Sahami, M., Guibas, L. J., & Sohl-Dickstein, J. (2015). Deep knowledge tracing. In *Proceedings of the 29th annual conference on neural information processing systems (NeurIPS)* (pp. 505–513).

Portelance, E., Degen, J., & Frank, M. C. (2020). Predicting age of acquisition in early word learning using recurrent neural networks. In *Proceedings of the annual meeting of the cognitive science society (CogSci)* (pp. 1057–1063).

Russin, J. L., Jo, J., O'Reilly, R. C., & Bengio, Y. (2020). Systematicity in a recurrent neural network by factorizing syntax and semantics. In *Proceedings of the annual meeting of the cognitive science society (CogSci)* (pp. 109–115).

Shi, Y., Chi, M., Barnes, T., & Price, T. W. (2022). Code-dkt: A code-based knowledge tracing model for programming tasks. In *Proceedings of the 15th international conference on educational data mining (EDM)*. International Educa-

tional Data Mining Society.

Su, Y., Cheng, Z., Luo, P., Wu, J., Zhang, L., Liu, Q., & Wang, S. (2021). Time-and-concept enhanced deep multidimensional item response theory for interpretable knowledge tracing. *Knowledge-Based Systems*, *218*, 106819.

Tong, H., Zhou, Y., & Wang, Z. (2020). Exercise hierarchical feature enhanced knowledge tracing. In *Proceedings of the 21st international conference on artificial intelligence in education (AIED)* (Vol. 12164, pp. 324–328). Ifrane, Morocco: Springer.

Truzzi, A., & Cusack, R. (2020). Can visual object representations in the human brain be modelled by untrained convolutional neural networks with random weights? In *Proceedings of the annual meeting of the cognitive science society (CogSci)* (p. 2810).

Wang, W., Li, G., Ma, B., Xia, X., & Jin, Z. (2020). Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *Proceedings of the 27th international conference on software analysis, evolution and reengineering (SANER)* (pp. 261–271). IEEE.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., … Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th annual conference on neural information processing systems (NeurIPS)*.

Zhu, M., Han, S., Yuan, P., & Lu, X. (2022). Enhancing programming knowledge tracing by interacting programming skills and student code. In *Proceedings of the international learning analytics and knowledge conference (LAK)* (pp. 438–443). ACM.