A GENERATIVE MODEL FOR NEXT-STEP CODE PREDICTION TOWARD PROACTIVE SUPPORT

Daiki Matsumoto[1], Atsushi Shimada[2] and Yuta Taniguchi[3]
[1]*Graduate School of Information Science and Electrical Engineering, Kyushu University, 744, Motooka, Nishi-ku, Fukuoka 819-0395, Japan*
[2]*Faculty of Information Science and Electrical Engineering, Kyushu University, 744, Motooka, Nishi-ku, Fukuoka 819-0395, Japan*
[3] *Section of Educational Information Research Institute for Information Technology, Kyushu University, 744, Motooka, Nishi-ku, Fukuoka 819-0395, Japan*

ABSTRACT

Predicting learner actions and intentions is crucial for providing personalized real-time support and early intervention in programming education. This approach enables proactive, context-aware assistance that is difficult for human instructors to deliver by foreseeing signs of potential struggles and misconceptions, or by inferring a learner's understanding and coding intent through early prediction of their intended solution. Traditional frameworks such as Knowledge Tracing are limited to predicting student performance on subsequent tasks, focusing only on how learners will perform in the next task rather than modeling progress within the current task. Existing approaches to single-task prediction primarily aim to generate the final submitted code. Consequently, the development of models capable of predicting the step-by-step evolution of the code within a single task remains underexplored. This paper proposes a generative model that traces the historical evolution of a learner's code to predict the next code snapshot in a task. Specifically, we develop a deep learning model that encodes "learner context" by feeding a time-series of code snapshots into an LSTM. This context, combined with the current code, is used to decode the next code. In our evaluation using real-world data collected from programming exercise classes, our model achieves a BLEU score of 0.639. The results confirm that incorporating learner context is essential for improving prediction accuracy, yielding up to a 7.5% improvement over a baseline model. We also identify effective model architectures and fine-tuning techniques that contribute to performance gains.

KEYWORDS

Programming Process, Learner Modeling, Context-Aware Code Generation

1. INTRODUCTION

Recently, the importance of personalized support tailored to individual learners' understanding and progress has grown in the field of programming education (Rojas-López & García-Peñalvo, 2022). Teachers can grasp the current state of a student's understanding as well as their coding intention by reading their source code, hearing their description of immediate issues, or asking questions. However, such interactions require considerable time in real-world educational settings, making it difficult for instructors and teaching assistants to consistently provide timely and in-depth support to every learner, especially in large classes (Singh, 2022).

Recent advances in natural language processing have introduced new forms of automated support into programming education, complementing earlier approaches such as Intelligent Tutoring Systems (Anderson et al., 1985). For instance, chatbots based on Large Language Models (LLMs), such as like ChatGPT (Ouyang et al., 2022), are now used interactively by learners, and various support applications that leverage these models have been proposed in academic research (Birillo et al., 2024). However, even with the availability of such tools, achieving automation that replicates human-like assistance requires accurately capturing the learner's current knowledge and coding intentions, which in turn demands a contextual understanding of the learner's coding process. Although effectively leveraging such a context remains difficult for chatbots, incorporating learner context and predicting future actions could significantly enhance their responsiveness and adaptability.

Knowledge Tracing (KT) (Corbett and Anderson, 1994) is a primary approach for predicting future learner performance based on past behavior. However, traditional frameworks such as KT are limited to predicting

student performance across tasks, focusing on how learners will perform in the next task rather than modeling their progress within a single task. Other approaches for single-task prediction primarily aim to generate the final submitted code (Liu et al., 2022). However, these methods are not designed to predict the intermediate steps that occur as learners gradually construct their solutions. Consequently, the development of models capable of predicting the step-by-step evolution of code within a single task while capturing the unique characteristics and progression patterns of each learner remains an underexplored area.

Predicting this step-by-step evolution is crucial for achieving proactive, context-aware support. By anticipating a learner's next likely coding actions, early signs of confusion or misconceptions can be detected before they escalate, thus enabling timely intervention. Furthermore, by forecasting the learner's intended code at each step, support systems can gain deeper insights into the learner's comprehension and coding intentions. This facilitates a level of personalized assistance that is difficult for human instructors to provide consistently at scale.

Based on this perspective, we propose a novel task: predicting future code states from a sequence of intermediate source code snapshots observed as learners iteratively revise their programs. This requires modeling how learners incrementally develop their solutions over time. To address this, we propose a deep generative model that incorporates the learner context into its predictions. Each past code snapshot is converted into an embedding and fed sequentially into an LSTM (Hochreiter & Schmidhuber, 1997), which estimates a dynamic "learner context" representing the learner's evolving state. This context is subsequently combined with the embedding of the most recent code snapshot and passed to a GPT-2 decoder (Radford et al., 2019) to generate the next predicted code. We investigate the effects of pre-training and fine-tuning on the model components to enhance prediction accuracy. Furthermore, we compare multiple strategies for integrating learner contexts with code embeddings and evaluate their impact on predictive performance.

2. RELATED WORK

Our study aims to predict the learner's next source code based on their coding process. We classify related prior work into two main categories based on the granularity of the data they handle: (1) "cross-task source code prediction," which predicts a learner's source code for the next task by modeling their evolving knowledge across multiple tasks, and (2) "local source code prediction," which predicts the next segment of code from an incomplete state within a single task.

2.1 Cross-Task Source Code Prediction

In this section, we review prior cross-task studies that predict source code based on a learner's history across multiple tasks.

The Knowledge Tracing (KT) framework, which is traditionally used for problems with binary answers, has been extended to free-response programming tasks (Liu et al., 2022). They proposed a new framework that sequentially estimates the learner's knowledge state based on the source code submitted across multiple tasks. Specifically, their model encodes the submitted code, uses an LSTM to accumulate the knowledge state, and uses a decoder to generate the next source code submission. Similar to our study, their study focuses on source code prediction based on learning history.

In a related approach for acquiring a knowledge state, InfoGAN (Chen et al., 2016) was proposed for source code generation. The model learns interpretable latent representations corresponding to the learner's coding patterns in an unsupervised manner by maximizing the mutual information between the generated code and its latent variables. This study also performs source code predictions based on multiple tasks. However, it differs slightly from the model proposed by Liu et al. in that it represents a learner's characteristic coding tendencies in a latent space and generates codes based on that representation.

Similar to our study, these studies aim to generate source code by predicting the next output based on the accumulated learning history. However, these studies handle the submission history of the source code across multiple tasks, and their prediction targets are also at the task level. In contrast, our study targets the sequential writing process within a single task, meaning that the granularity of the source code history that we handle is different.

2.2 Local Source Code Prediction

Other study focuses on generating the next source code segment based on the current state of the code within a single task. One proposed framework generates the next single line of code from the current source code by combining a Large Language Model (LLM) with static analysis (Birillo et al., 2024). Through a three-stage process involving subgoal extraction, source code generation, and explanation generation, this method can present the next segment of the code or provide relevant hints based on the learner's current code.

Another approach focuses on the history of source code edits, learning "edit vectors" that capture the intent of a correction from pre- and post-edit code pairs and their test results (Heickal and Lan, 2025). This embedding representation is designed such that similar correction operations are represented by similar vectors, enabling source code generation that reflects a learner's corrective actions.

These studies have a structure similar to that of our study in that they directly generate the next source code based on information from the current code. However, both are designed to take a single source code snapshot as the input, and do not have an architecture that makes contextual predictions based on the entire source code history. In our study, by capturing the entire history using a time-series model, we aim to realize a more natural form of source code generation that reflects the learners' progress and stylistic tendencies.

3. PROPOSED METHOD

3.1 Problem Definition

The objective of this study is to sequentially predict the source code to be written in the next step, taking a learner's source code history up to a certain point within a single task and the task description as the input. In this context, the source code history is a sequence of source code snapshots recorded as in-progress work while a learner engages with a task. Based on the snapshot at each step, the model predicts the source code to be written in the next step.

3.2 Proposed Model

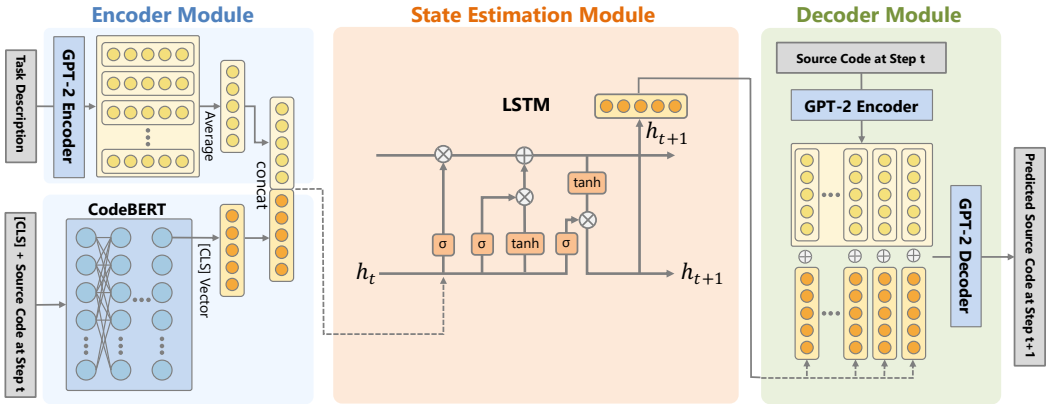

Figure 1. Overview of the proposed model architecture.

An overview of the proposed method is shown in Figure 1. The proposed method comprises of three main modules. First, the encoder module converts the learner's current source code and task description into a combined embedding. Next, the state-estimation module uses an LSTM to sequentially update a hidden state, which represents the learner's context based on their source code history. Finally, the decoder module integrates the learner context with the source code obtained from the previous step to generate a prediction for the next source code. This process is applied at each step to sequentially predict the learner's source code. These components are described in detail as follows.

In the encoder module, the learner's source code and task description are converted into embedding representations. To encode the task description, the Japanese text is translated into English and then input into a GPT-2 encoder to obtain an embedding for each token. The overall embedding for the entire task description is constructed by averaging these token-level embeddings. For the source code, it is converted into a token sequence using a CodeBERT (Feng et al., 2020) tokenizer, and a special [CLS] token is added to the beginning before the sequence is input into CodeBERT. The [CLS] token is commonly used in BERT-based models to capture the features of an entire sequence. In this study, we utilize the vector representation corresponding to this [CLS] token as the embedding representation for the source code. Finally, the task description embedding and source code embedding are combined and passed to the subsequent processing stage.

In the state estimation module, a time-series internal representation is updated using an LSTM based on the history of the previously written source code and the representation of the current source code. In the initial step, the LSTM's initial hidden state and cell state, along with the source code embedding output from the encoder module, are used as inputs to derive a new hidden state and cell state. In each subsequent step, the internal representation is sequentially updated by inputting the previous hidden and cell states and the corresponding source code embedding into the LSTM. In this study, we treat the hidden state obtained from the LSTM as the "learner context," which serves as the contextual information for the next source code prediction. The learner context is expected to reflect the learner's internal state at that point, including their coding patterns and syntactic progression.

In the decoder module, the source code that the learner will write next is generated based on the source code from the previous step and learner context. First, the source code from the preceding step is input into a GPT-2 encoder to obtain an embedding for each token. Next, the learner context, h, is integrated with the token embeddings, p_n. We investigate the following four integration methods, where p_n is the embedding for the n-th token, h is the learner context, and p'_n is the resulting token embedding after integration.

(A) Linear Addition: $p'_n = p_n + f(h)$
(B) Scalar Addition: $p'_n = p_n + \alpha \cdot h$
(C) Concatenation and Transformation (Concat. & Trans.): $p'_n = f(concat(p_n, h))$
(D) Final Token Addition (FTA): $p'_{last} = p_{last} + f(h)$

Method (A) adds the linearly transformed learner context to each token's embedding. Method (B) adds the learner context after scaling it by a learnable scalar constant, α. Method (C) concatenates the token embedding with the learner context and integrates them through a linear transformation. Method (D) adds the linearly transformed learner context only to the final token in the sequence. Finally, the integrated token embeddings, p', are input into the GPT-2 decoder to generate the next predicted code. Although we adopted GPT-2 due to the high computational and memory demands of processing our time-series data, our approach is not limited to this model. Any GPT-style generative model could be substituted, and we expect that using larger models, especially those pretrained on source code corpora, would further improve the quality of the generated code.

3.6 Pre-Training and Fine-Tuning Each Module

The encoder module uses CodeBERT, a pre-trained model, to represent the source code of the learner. Although CodeBERT is pre-trained on a large-scale source code dataset, in its original state, it has not been explicitly trained for the [CLS] token, which is added to represent an entire sequence. Because this study uses the output corresponding to the [CLS] token as the embedding for the entire source code, we introduced fine-tuning using SimCSE (Gao et al., 2021) to improve the utility of this representation. SimCSE uses contrastive learning to learn enhanced embeddings in order to better distinguish between data points. In this approach, an input sentence is treated as a positive example, whereas other data points are treated as negative examples. This process is expected to yield embeddings that can capture fine-grained differences between various source codes.

For the state estimation module, the LSTM was pre-trained on a regression task to predict the increase or decrease in the number of tokens in a source code sequence. Specifically, using the source code history as the input, a two-layer linear transformation was connected to the hidden state at each step, and the model was trained to predict the extent to which the token count of the source code of the next step would increase or decrease compared with the current source code. This pre-training process was designed to enable the LSTM to learn the trends of source code changes during the writing process. Consequently, it can generate a learner

context that reflects the direction and magnitude of these changes, such as how much code the learner might add or how much they might delete or modify, in the next step.

The decoder module utilizes a pre-trained language model GPT-2, which is capable of generating grammatical natural language text. However, in its original state, the capacity to generate code that conforms to the unique syntactic structures and descriptive styles of the source code is insufficient. Therefore, to supplement the expressive power necessary for source code generation, this study fine-tuned GPT-2 using source code. The model was fine-tuned using the Python source code from two datasets: The Stack (Kocetkov et al., 2023), a large-scale collection of open-source code, and MBPP (Austin et al., 2021), a small-scale dataset of basic programming problems. The process involved first performing auto-regressive learning on 100,000 samples from The Stack, followed by additional training using MBPP. This helps the decoder learn patterns specific to the source code, such as syntax and coding conventions, to enable more natural code generation.

4. EXPERIMENTAL RESULTS AND DISCUSSION

To verify the effectiveness of the proposed method described in Section 3, we conduct evaluation experiments using a newly created dataset. Specifically, we predict the source code for the next step based on the source code history up to a certain point and compare it with the code written by the learner. The following subsections describe the dataset, evaluation metrics, and experimental results.

4.1 Dataset

The dataset used in this experiment consists of source code recorded during introductory Python programming courses at Kyushu University in the 2023 and 2024 academic years. The target task for this study was from the sixth session of a total of thirteen exercise sessions, requiring learners to implement a function that finds the maximum value in a list. A total of 2,839 source code snapshots from 121 learners were collected. This study was approved by the university's research ethics committee. We explained the data collection and usage to the students in advance, obtained informed consent, and used data only from those who agreed to participate.

The data were recorded using WEVL (Taniguchi et al., 2022), an online programming exercise environment. WEVL's system is designed to automatically save the current source code whenever a learner stops typing for three or more seconds or when they execute or submit a task. In this study, each of these automatic saves is treated as a single "step."

4.2 Evaluation Metrics

In this study, we use three main metrics to evaluate the similarity and generation accuracy between the generated and ground-truth source codes.

Our primary evaluation metric is the BLEU score (Papineni et al., 2002). This metric is commonly used to measure the degree of similarity of source code by calculating the n-gram match rate between a generated output and a ground-truth reference. In this study, we calculate the BLEU score using the CodeBERT tokenizer[1] for a token-level comparison.

Additionally, we use edit distance to measure the difference between the generated source code and ground-truth source code more directly (Levenshtein et al., 1966). Edit distance is defined as the minimum number of insertions, deletions, or substitution operations required to transform one string or token sequence into another. For this study, we use two types of edit distances: a token-level distance calculated using the CodeBERT tokenizer, and a character-level distance based on individual characters.

Finally, we evaluate the generalization performance of the model by measuring the test loss, calculated as the cross-entropy error between the model's output probabilities and the ground truth on an unseen dataset.

[1] https://huggingface.co/microsoft/codebert-base

4.3 Experimental Results

Using the dataset described in Section 4.1, we trained our proposed model and conducted evaluation experiments. Cross-validation was used for training and evaluation. The dataset was divided into five folds and configured such that every source code sample was used as test data exactly once. The results are summarized in Table 1. Each result represents the average value across the five folds.

Table 1: Evaluation results for each setting.

No.	CodeBERT Fine-Tuning	Integration Method	GPT Fine-Tuning	LSTM Pre-Training	BLEU	Edit Distance (Token)	Edit Distance (Char)	Test Loss
1	N/A	Baseline		N/A	0.591	19.873	30.401	0.368
2		Linear Addition			0.608	19.785	29.503	0.365
3	✓	Linear Addition			0.620	19.136	29.050	0.371
4	✓	Scalar Addition			0.616	18.479	28.038	0.370
5	✓	FTA			0.608	21.060	30.494	0.366
6	✓	Concat. & Trans.			0.367	35.944	51.756	0.727
7	✓	Linear Addition	✓		0.625	18.877	28.491	**0.357**
8	✓	Linear Addition		✓	**0.639**	**17.244**	**25.580**	0.379
9	✓	Linear Addition	✓	✓	**0.639**	17.527	25.957	0.370

The baseline setting (No. 1 in the table) serves as the initial point of comparison. In this configuration, the model predicts the source code for the next step using only the immediately preceding source code as input without leveraging the hidden state from the state estimation module. That is, this setting makes predictions without using the learner's source code history, resembling a scenario in which an LLM-based chatbot is asked to predict the immediate next code without prior context. By contrast, all settings from No. 2 onward use the learner's source code history and make predictions that incorporate contextual information via the hidden state.

Next, we discuss the results of each experimental setting. First, a comparison between the baseline model (No. 1) and subsequent settings that use the learner context (No. 2-9) shows that the latter achieved consistently higher BLEU scores. Specifically, our best-performing model (No. 8) achieved a BLEU score of 0.639, representing a 7.5% improvement over the baseline model. This indicates that leveraging information from past source code history improves prediction accuracy compared to using only the immediately preceding code (No. 1). This suggests that the learner context from the state estimation module functions as effective contextual information for prediction.

Comparing Nos. 2 and 3, No. 3, which included fine-tuning CodeBERT in the encoder, demonstrated a higher BLEU score. This result suggests that contrastive learning from SimCSE yielded embeddings that capture syntactic differences in the source code, which contributed to an improvement in prediction accuracy.

Comparing the integration methods (Nos. 3-6), linear addition (No. 3) performed best on the BLEU score, likely because its architecture effectively incorporates the historical context while maintaining informational consistency. Scalar addition (No. 4) performed similarly; although this method is simpler, the state vector itself may have adapted during the learning process to function effectively. However, the final token addition (No. 5) was slightly less effective on both metrics because modifying only a single token had a limited impact on the overall representation. Finally, concatenation and transformation (No. 6) significantly degraded the accuracy of both BLEU and edit distance. Despite offering high flexibility, the model likely failed to utilize the learner context effectively due to the increased dimensionality and complexity of the representation space.

Additionally, comparing Nos. 3 and 7, No. 7, which involved fine-tuning the GPT-2 decoder, showed a slight improvement in the BLEU score. This result suggests that pre-training specialized for source code supplemented the decoder's prior knowledge of the output representation, thereby enabling more natural source code generation.

Finally, a comparison between No. 3 and No. 8 showed that No. 8, which applied pre-training to the LSTM in the state estimation module, achieved a higher prediction accuracy. This is likely because the LSTM, having been pre-trained on a task to predict token count fluctuations, was better able to capture learner's progress trends during the writing process, leading to improved prediction performance.

Focusing on token-based edit distance, the average was 19.873 for No. 1, which improved 17.244 for No. 8, the best-performing model. This represents a reduction of approximately 2.6 required correction operations per generation, indicating that the output of the model more closely matched the ground-truth code.

4.4 Generation Examples

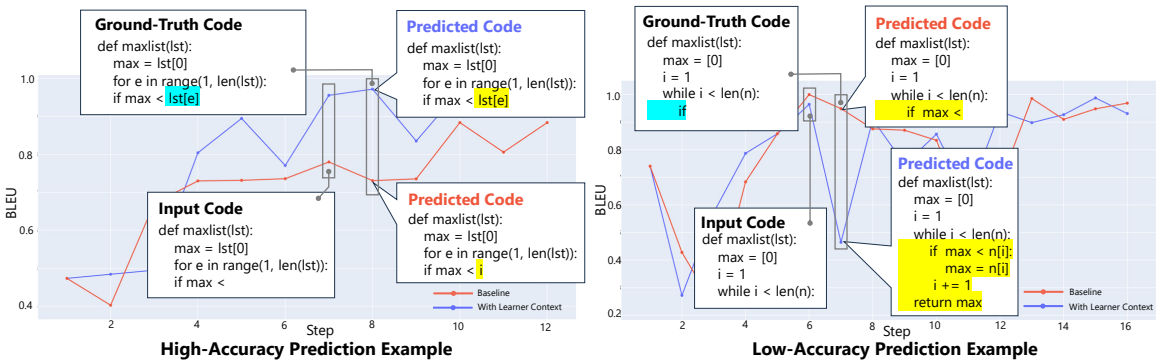

Figure 2. Generation example showing cases of high accuracy (left) and low accuracy (right).

In this section, we present specific examples of the model's prediction results to analyze their consistency with the ground-truth source code and examine the prediction trends. Figure 2 shows two cases: an example of a high-accuracy prediction (left panel) and an example of a low-accuracy prediction (right panel). In both panels, the line graph displays the BLEU score at each step; the red line represents the baseline setting (No. 1) and the blue line represents the setting that utilizes the learner context (No. 9). "Input Code" and "Ground-Truth Code" are what the learner actually wrote, while "Predicted Code" is the code generated by the model at the corresponding step. In the code blocks, the code highlighted in blue, including white spaces, represents the code added by the learner, while the yellow-highlighted code indicates the code added by our model.

In the high-accuracy example (left panel), given the input at step 7, the model with learner context (No. 9) accurately predicts the code at step 8, including the learner's progress and writing style. The baseline model (No. 1), on the other hand, produces an output that differs from the ground-truth source code.

In the second example (right panel), there is a discrepancy between the prediction and the ground-truth source code. In this case, for the input at step 6, the ground-truth source code required only a minor change, that is, the addition of a single if statement. Although the code generated by the baseline model (No. 1) is grammatically and semantically natural, it indicates more progress than the ground-truth source code, suggesting that the magnitude of the change was slightly overestimated. Furthermore, the model that utilized the learner context (No. 9) generated an output that almost entirely completed the source code, showing that it overestimated the learner's progress compared with their actual coding stage. These examples suggest that challenges remain in accurately capturing aspects such as a learner's intent and progress.

5. CONCLUSION AND FUTURE WORK

This study addressed the underexplored challenge of predicting the step-by-step evolution of code within a single task, where the paths of evolution often differ across individual learners. We developed a generative model based on the time-series of a learner's code snapshots to predict the code that will be written in the next step. In the proposed model, an LSTM captures the historical learner context, which is then fed into a GPT-2 decoder to generate the next-step code. We successfully identified specific techniques that improve performance, such as targeted pre-training, fine-tuning, and an effective context integration method. Evaluations on real-world classroom data showed that incorporating each learner's historical code evolution consistently improved prediction accuracy, achieving a BLEU score of 0.639 and outperforming a baseline that uses only the immediately preceding code by up to 7.5%.

However, our work is limited by the use of GPT-2, adopted due to computational constraints, and by reliance on a single dataset; future work should explore larger code-pretrained models and more diverse contexts. In addition, our model sometimes fails to predict the learner's next step, overestimating progress by

predicting too far ahead. This could become an obstacle when providing support based on predictions, as it may lead to feedback misaligned with the learner's actual situation. Future work should address this issue by exploring ways to capture subtle individual traits such as coding pace, coding styles, and problem-solving strategies, which may require incorporating source code histories from other tasks.

Overall, our approach enables finer-grained next-step predictions that better capture diverse individual trajectories. These results extend beyond traditional knowledge tracing and suggest a path toward models that can more closely mimic learners' behaviors in programming classes. Our findings provide a foundation for automated systems that deliver personalized, context-aware support in programming education.

ACKNOWLEDGEMENT

This work was supported by JST CREST Grant Number JPMJCR22D1, JSPS KAKENHI Grant Number JP24K15209 and JP22H00551.

REFERENCES

Anderson, J. R. et al., 1985. Intelligent tutoring systems. *Science*, Vol. 228, No. 4698, pp. 456-462.

Austin, J. et al., 2021. Program Synthesis with Large Language Models. *arXiv preprint,* arXiv:2108.07732.

Birillo, A. et al., 2024. One Step at a Time: Combining LLMs and Static Analysis to Generate Next-Step Hints for Programming Tasks. *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, New York, NY, USA, Article no. 9.

Chen, X. et al., 2016. InfoGAN: interpretable representation learning by information maximizing generative adversarial nets. *Proceedings of the 30th International Conference on Neural Information Processing Systems*, Barcelona, Spain, pp. 2180–2188.

Corbett, A. T. and Anderson, J. R., 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction*, Vol. 4, No. 4, pp. 253-278.

Feng, Z. et al., 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*, Online, pp. 1536–1547.

Gao, T. et al., 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Online and Punta Cana, Dominican Republic, pp. 6894–6910.

Heickal, H. and Lan, A., 2025. Learning Code-Edit Embedding to Model Student Debugging Behavior. *Proceedings of the 26th International Conference on Artificial Intelligence in Education*, Palermo, Italy, pp. 91-98.

Hochreiter, S. and Schmidhuber, J., 1997. Long Short-Term Memory. *Neural Computation*, Vol. 9, No. 8, pp. 1735–1780.

Kocetkov, D. et al., 2023. The Stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*.

Levenshtein, V. I. et al., 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, Vol. 10, No. 8, pp. 707–710.

Liu, N. et al., 2022. Open-ended Knowledge Tracing for Computer Science Education, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Abu Dhabi, United Arab Emirates, pp. 3849–3862.

Ouyang, L. et al., 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, New Orleans, USA, pp. 27730-27744.

Papineni, K. et al., 2002. BLEU: a method for automatic evaluation of machine translation. *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, Philadelphia, Pennsylvania, USA, pp. 311–318.

Radford, A. et al., 2019. Language models are unsupervised multitask learners. *OpenAI blog*, Vol. 1, No. 8, p. 9.

Rojas-López, A. and García-Peñalvo, F. J., 2022, Personalized education for a programming course in higher education. *Research Anthology on Computational Thinking, Programming, and Robotics in the Classroom*, IGI Global, Hershey, PA, USA, pp. 344–367.

Singh, S., 2022. Identifying Learning Challenges faced by Novice/Beginner Computer Programming Students: An Action Research Approach. *Proceedings of the 6th Software Engineering Education Workshop*, Online, pp. 63–71.

Taniguchi, Y. et al., 2022. Visualizing Source-Code Evolution for Understanding Class-Wide Programming Processes. *Sustainability,* Vol. 14, No. 13, Article no. 8084.