Open-Ended Knowledge Tracing for Computer Science Education

Naiming Liu*, **Zichao Wang***, **Richard G. Baraniuk**
Rice University
nl35,zw16,richb@rice.edu

Andrew Lan
University of Massachusetts Amherst
andrewlan@cs.umass.edu

Abstract

In education applications, *knowledge tracing* refers to the problem of estimating students' time-varying concept/skill mastery level from their past responses to questions and predicting their future performance. One key limitation of most existing knowledge tracing methods is that they treat student responses to questions as *binary-valued*, i.e., whether they are correct or incorrect. Response correctness analysis/prediction ignores important information on student knowledge contained in the exact content of the responses, especially for open-ended questions. In this paper, we conduct the first exploration into *open-ended knowledge tracing* (OKT) by studying the new task of predicting students' exact open-ended responses to questions. Our work is grounded in the domain of computer science education with programming questions. We develop an initial solution to the OKT problem, a student knowledge-guided code generation approach, that combines program synthesis methods using language models with student knowledge tracing methods. We also conduct a series of quantitative and qualitative experiments on a real-world student code dataset to validate OKT and demonstrate its promise in educational applications.

1 Introduction

Knowledge tracing (KT) (Corbett and Anderson, 1994) refers to the problem of estimating student mastery of concepts/skills/knowledge components from their responses to questions and using these estimates to predict their future performance. KT methods play a key role in many of today's large-scale online learning platforms to automatically estimate the knowledge levels of a large number of students and provide each of them with personalized feedback and recommendation, leading to improved learning outcomes (Ritter et al., 2007). KT methods consist of two essential components. First, a *knowledge estimation* (KE) component, i.e.,

$$\mathbf{h}_{t+1} = \mathrm{KE}((\mathbf{p}_1, \mathbf{x}_1), \ldots, (\mathbf{p}_m, \mathbf{x}_t)), \quad (1)$$

estimates a student's current knowledge state $\mathbf{h}_{t+1}$ using questions ($\mathbf{p}$) and responses ($\mathbf{x}$) from previous (discrete) time steps for this student. Second, a *response prediction* (RP) component predicts the student's response to the next question (or future questions), i.e., $\mathbf{x}_{t+1} \sim \mathrm{RP}(\mathbf{h}_{t+1}, \mathbf{p}_{t+1})$. Section 4 contains a detailed overview of existing KT methods and how the question, responses, and knowledge state variables are represented.

One key limitation of almost all existing KT methods is that they only analyze and predict *binary-valued* student responses to questions, i.e., the *correctness* of the response. That is, the RP is typically a simple binary classifier. As a result, one can broadly apply KT methods to any question as long as student responses are graded. However, this approach loses important information regarding student mastery, since it does not make use of the *content* of questions and student responses, especially for open-ended questions. Past work has shown that students' open-ended responses to such questions contain useful information on their knowledge states, e.g., having a "buggy rule" (Brown and Burton, 1978), exhibiting misconceptions (Feldman et al., 2018; Feng et al., 2019; Smith III et al., 1994), or a general lack of knowledge (Anderson and Jeffries, 1985); this information is highly salient for instructors but cannot be captured by response correctness alone.

Generative language models such as GPT (Brown et al., 2020) provide an opportunity to fully exploit the rich information contained in open-ended student responses in various domains for the purposes of KT. In this paper, we focus on computer science education, where short programming questions require students to write code chunks that satisfy the question's requirements. The program synthesis capabilities of variants of pre-trained neu-

*The first two authors contributed equally.

ral language models such as CodeX (Chen et al., 2021) enable the generation of short chunks of code from natural language instructions, which we can leverage for open-ended response prediction. However, two key challenges make this task difficult: First, as students learn through practice, their knowledge on different programming concepts is *dynamic*; students can often learn and correct their errors given instructor-provided feedback or even error messages generated by the compiler. Therefore, *we need new KE models that can effectively trace time-varying student knowledge throughout their learning process.* Second, student-generated code is often *incorrect and exhibits various errors*; there may also exist multiple correct responses that capture different lines of thinking among students. This intricacy is not covered by program synthesis models, since their goal is to generate correct code and they are usually trained on code written by skilled programmers. Therefore, *we need new RP models that can generate student-written (possibly erroneous) code that reflects their (often imperfect) knowledge of programming concepts.*

1.1 Contributions

In this paper, we present the first attempt at analyzing and predicting exact, open-ended student responses, specifically for programming questions in computer science education. Our contributions can be summarized as follows:

- We define the **open-ended knowledge tracing (OKT)** framework, a novel KT framework for open-ended student responses, and a new KT task, exact student response prediction. We ground OKT in the domain of computer science education for student code submission analysis and prediction but emphasize that *OKT can be broadly applicable to a wide range of subjects that involve open-ended questions*.
- We develop an initial solution to the OKT task, a **knowledge-guided** code generation method. Our method combines KE components in existing binary-valued KT methods with code generation models, casting the OKT task as a *dynamic controllable generation* problem where the control, i.e., time-varying student knowledge states, are also learned.
- Through **extensive experiments on a real-world student code dataset**, we explore the effectiveness of OKT in reflecting variations of student code and especially errors in its knowledge state estimates. We explore the effectiveness of our solution in making reasonably accurate predictions of student-submitted code. We also discuss how these OKT capabilities can help computer science instructors and outline several new research directions.

[*]Find our code at https://github.com/lucy66666/OKT

2 OKT for Computer Science Education

We now define the OKT framework and detail specific model design choices in the domain of computer science education, where we focus on analyzing students' code submissions to programming questions. Figure 1 illustrates the three key components of OKT: knowledge representation (KR), KE, and response generation (RG), the last of which is the key difference between OKT and existing KT methods. Our key technical challenges are (i) how to represent programming questions and student code submissions (KR, Section 2.1) and use them to estimate student knowledge states (KE, Section 2.2); (ii) how to combine knowledge states with the question prompt to generate student code (RG, Section 2.3); and (iii) how to efficiently perform optimization to train the OKT model components (Section 2.4).

2.1 Knowledge Representation (KR)

The purpose of the KR component is to convert the prompt/statement of questions that students respond to and their corresponding code submissions to continuous representations. Our KR component is significantly different from existing binary-valued KT methods that ignore question/response content and one-hot encode them using question/concept IDs and response correctness.

Question Representation: We adopt the popular GPT-2 model[1] (Radford et al., 2019) for prompt representation: Given a question prompt $\mathbf{p}$, GPT-2 tokenizes it into a sequence of M word tokens, where each token has an embedding $\bar{\mathbf{p}}_m \in \mathbb{R}^K$. For GPT-2, the dimension of these embeddings is $K = 768$. This procedure produces a sequence of token embeddings $\{\bar{\mathbf{p}}_1, \bar{\mathbf{p}}_2, \ldots, \bar{\mathbf{p}}_M\}$. We then average the embeddings of each prompt token to

[1]One can use any language model; we choose GPT-2 since our RG component for student code is also built on GPT-2.

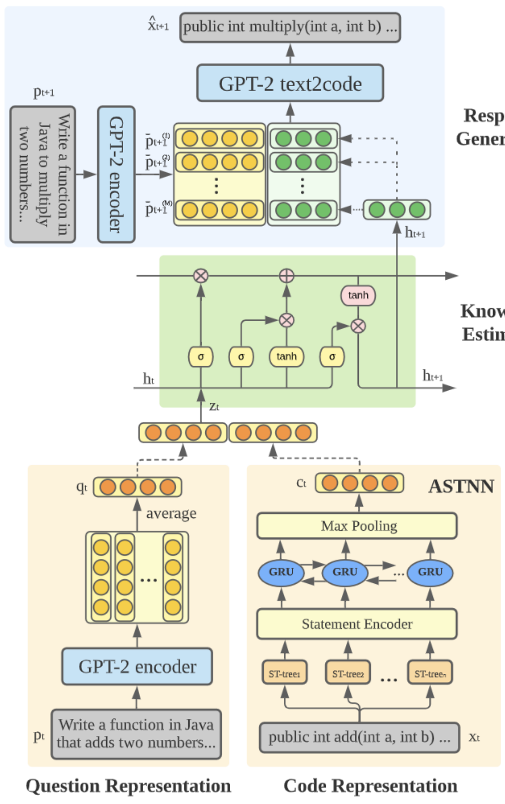


Figure 1: Open-ended knowledge tracing (OKT) block diagram. We update the student's current knowledge state $\mathbf{h}_{t+1}$ using the last question $\mathbf{p}_t$ and actual student code $\mathbf{x}_t$. We then combine it with the next question statement $\mathbf{p}_{t+1}$ to generate our prediction of the actual student code $\hat{\mathbf{x}}_{t+1}$.

get our prompt embedding $\mathbf{q} = \sum_{m=1}^{M} \frac{\bar{\mathbf{p}}_m}{M}$, where the average is computed element-wise on vectors.

Code Representation: In order to preserve both semantic and syntactic properties of programming code in embedding vectors, we utilize ASTNN (Zhang et al., 2019), a popular tool for code representation. We first parse student-submitted code into an abstract syntax tree (AST). We then split each full AST into a sequence of non-overlapping statement trees (ST-trees) through preorder traversal. Each ST-tree contains a statement node as the root and its corresponding AST nodes as children. We then pass the ST-trees through a recurrent statement encoder to obtain embedding vectors and use a bidirectional gated recurrent unit network (Bahdanau et al., 2014) to capture the naturalness of the statements and further enhance the capability of the recurrent layer. Eventually, we apply a max-pooling layer to capture the most important semantics for each dimension of the embedding. We denote this entire process as $\mathbf{c} = \text{ASTNN}(\mathbf{x})$ where $\mathbf{x}$ is student-submitted code and $\mathbf{c}$ is its code embedding vector, which we use as input to the KE component. We refer readers to (Zhang et al., 2019) for more details on ASTNN.

2.2 Knowledge Estimation (KE)

The purpose of the KE component is to turn a student's past question/code information into estimates of their current knowledge state. Following DKT (Piech et al., 2015a), a popular existing KT method, we use a long short-term memory (LSTM) model (Hochreiter and Schmidhuber, 1997) to update a student's current knowledge state, $\mathbf{h}_{t+1}$, given their previous response at the last time step. We use the output of the KR component, i.e., question prompt and code embeddings, as the input to the KE component as $\mathbf{h}_{t+1} = \text{LSTM}(\mathbf{h}_t, \mathbf{q}_t, \mathbf{c}_t)$ and use it as input to the RG component to generate predicted student code submissions. In principle, we can use any existing binary-valued KT method as OKT's KE component. We validate in our experiments (Section 3) that OKT is compatible with two other popular KT methods, DKVMN (Zhang et al., 2017) and AKT (Ghosh et al., 2020), that are based on external memory and attention networks.

2.3 Response Generation (RG)

The purpose of RG, OKT's core component, is to **predict open-ended responses**, i.e., generate predicted student code, which makes OKT significantly different from existing binary-valued KT methods with binary classifiers of response correctness. We fine-tune a base GPT-2 generative model into a text-to-code model P_Θ with parameter Θ on code data (see Section 2.5 for details). We choose language models over other code generation approaches since their text-to-code generation pipeline suits OKT well.

Our key technical challenge is how to use knowledge states as *control* in the code generation model to guide personalized code predictions for each student. Given the current question prompt, $\mathbf{p}_{t+1}$, and its sequence of M token embeddings $\{\bar{\mathbf{p}}_1, \bar{\mathbf{p}}_2, \ldots, \bar{\mathbf{p}}_M\}$, where $\bar{\mathbf{p}}_m \in \mathbb{R}^K$ (we drop the time step index t in prompt tokens for clarity), our approach for injecting student knowledge states into the code generation model is to replace raw token embeddings with *knowledge-guided* embeddings using an *alignment* function, i.e., $\mathbf{p}_m = f(\bar{\mathbf{p}}_m, \mathbf{h}_{t+1})$ for $m = 1, \ldots, M$. Therefore, the GPT-2 input embeddings are

$$\{\mathbf{p}_1, \ldots, \mathbf{p}_M\} = \{f(\bar{\mathbf{p}}_1, \mathbf{h}_{t+1}), \ldots, f(\bar{\mathbf{p}}_M, \mathbf{h}_{t+1})\}.$$

Intuitively, this (possibly learnable) alignment function aligns the space of knowledge states with the space of textual embeddings for the question prompt. Thus, knowledge states are responsible for predicting different code submitted to the same programming question by different students.

We explore four different alignment functions to combine knowledge states with question prompt token embeddings:

- Addition, i.e., $\mathbf{p}_m = \bar{\mathbf{p}}_m + \mathbf{h}_{t+1}$.
- Averaging, i.e., $\mathbf{p}_m = (\bar{\mathbf{p}}_m + \mathbf{h}_{t+1})/2$.
- Weighted addition, i.e., using a learnable weight for knowledge states, $\mathbf{p}_m = \bar{\mathbf{p}}_m + \alpha \cdot \mathbf{h}_{t+1}$.
- Linear combination, i.e., applying a learnable affine transformation to the knowledge states before adding it to token embeddings, $\mathbf{p}_m = \bar{\mathbf{p}}_m + \mathbf{A}\mathbf{h}_{t+1} + \mathbf{b}$.

The latter two functions are learnable with parameters $\alpha \in \mathbb{R}$, $\mathbf{A} \in \mathbb{R}^{D \times K}$, and $\mathbf{b} \in \mathbb{R}^K$. Therefore, the predicted student code (with N code tokens), $\mathbf{x} = \{x_1, x_2, \ldots, x_N\}$, is generated in an autoregressive manner by the RP component given the knowledge-guided question prompt token embeddings $\{\mathbf{p}_1, \ldots, \mathbf{p}_M\}$.

2.4 Optimization

During the training process, we jointly optimize the parameters of the KE and RG components of OKT; in essense, we are learning *both* a controllable generation model for student responses *and* the control itself, which is the student's time-varying knowledge state. We keep the knowledge representation encoders E_1 and E_2 fixed. The objective for one student code submission is given by

$$Loss = \textstyle\sum_{n=1}^{N} -\log P_\Theta\big(x_n \big| \{\mathbf{p}_1, \ldots, \mathbf{p}_M\}, \{x_{n'}\}_{n'=1}^{n-1}\big), \tag{2}$$

where Θ denotes the set of parameters in the RP component, including both the GPT-2 text-to-code model parameters and learnable parameters in the alignment function $f(\cdot)$. The final training objective is the sum of this loss over all code submissions made by all students.

We also design an efficient training setup for OKT. For existing neural network-based KT methods, at each training step, we use a batch of student (question, response) sequences to compute the correctness prediction loss across all time steps and all students in the batch. We cannot use this training method since OKT's loss for one student is the sum of code prediction losses over all time steps, whereas the loss at each time step is itself the sum of a sequence of cross entropy losses for code token predictions. As a result, if we use the training setup for existing KT methods, at each training step, we need to call the response generator for a total of $T \times B$ times where T is the number of time steps and B is the batch size, which will significantly slow down training. Instead of batching over students, we use a batch of (student, time step) pairs. Then, at each training step, we first apply the knowledge update component in OKT to compute the knowledge states for students in the batch, extract the knowledge states corresponding to the sampled time steps in the batch, and then feed them into the response generator. This setup enables efficient training for OKT.

2.5 Pre-training Models

Before training OKT, we pre-train its KE component using the binary-valued correctness prediction loss with question and code embeddings as input, following (Mao et al., 2021; Zhu et al., 2021). Since we cannot directly use CodeX (Chen et al., 2021) due to our need to adjust the input embeddings with student knowledge states, we pre-train a text-to-code pipeline by fine-tuning a standard GPT-2 model on the Funcom dataset (LeClair and McMillan, 2019), which contains 2.1 million Java code snippets and their textual descriptions.

3 Experiments

We now present a series of experiments to explore the capabilities of OKT. We first introduce the dataset, various quantitative metrics on which we evaluate various methods, and detail quantitative results. We then qualitatively illustrate that OKT (i) learns a meaningful latent student knowledge space and (ii) generates predicted student code that capture their coding patterns and error types.

Dataset: We use the dataset from the CSEDM Data Challenge, henceforth referred to as the **CSEDM dataset**.[2] To our knowledge, this is the only college-level, publicly-available dataset *with students' actual code submissions*; a concurrent work (Singla and Theodoropoulos, 2022) uses the

[2]Challenge: `https://sites.google.com/ncsu.edu/csedm-dc-2021/`. The dataset is called "CodeWorkout data Spring 2019" in Datashop (`pslcdatashop.web.cmu.edu`).

Hour of Code dataset, which has some similarities with this dataset but only has two questions. The CSEDM dataset contains 246 college students' 46,825 full submissions on each of the 50 programming questions over the course of an entire semester. The dataset contains rich textual information on question prompt and students code submissions as well as other relevant metadata such as the programming concepts involved in each question and all error messages returned by the compiler. See Section A in the Supplementary Material for detailed data statistics and preprocessing steps.

Evaluation Metrics: In the context of predicting students code submissions, we need a variety of different metrics to fully understand the effectiveness of OKT. We thus use two types of evaluation metrics. First, we need metrics that can measure OKT's ability to predict student code on the test set after training. For this purpose, we use two metrics, including **CodeBLEU** (Ren et al., 2020), a variant of the classic BLEU metric adapted to code that measures the similarity between predicted code and actual student code. The other metric is the average **test loss** across code tokens computed using OKT methods with the lowest validation loss. Second, we need metrics that can measure the diversity of predicted student code since we do not want OKT to simply memorize frequent student code in the training data. For this purpose, we use the **dist-N** metric (Li et al., 2016) that computes the ratio of unique N-grams in the predictions over all N-grams. We choose $N = 1$ in this work since uni-gram setting is more compatible with the limited coding vocabulary size. We note that predicting whether a student code submission passes test cases is another important task for OKT evaluation; however, since test cases are not included in the CSEDM dataset, we cannot conduct this evaluation and leave it for future work.

Methods for Comparison: Since exact student code prediction is a novel task, there are **no existing baselines** that we can compare against. We thus compare among variants of OKT to demonstrate that it is highly flexible and extensible. First, we test three different existing binary-valued KT methods, DKT, DKVMN, and AKT, as the KE component; one can apply any existing binary-valued KT method as the KE component that is suitable. As a strong baseline, we also test a version of OKT without KE and use the question prompt and code embeddings from the previous time step as additional input to the text-to-code RG component. Second, we compare different alignment functions between the knowledge and question prompt embedding spaces listed in Section 2.3. Third, we compare several training settings, including pre-training the KE and RG components and using a multi-task training objective by adding the binary-valued response correctness prediction loss to the code generation loss in Eq. 2, following (An et al., 2022).

setting	KT model	CodeBLEU ↑	Dist-1 ↑	Test Loss ↓
first submission	**DKT**	0.690	0.422	0.178
	AKT	0.581	0.401	0.193
	DKVMN	0.580	0.388	0.196
	None	0.518	0.426	0.215
all submissions	**DKT**	0.726	0.403	0.111
	AKT	0.632	0.396	0.125
	DKVMN	0.570	0.399	0.135
	None	0.471	0.385	0.151

Table 1: OKT results comparing different KT models as the KE component of OKT. AKT slightly outperforms DKVMN while DKT performs best under both settings.

Experimental Setup: Following typical settings in the KT literature, our goal is to predict the code a student submits to a question at the next time step t, $\mathbf{x}_{t+1}$, given their question prompts and code submissions in all previous time steps, i.e., $(\mathbf{p}_1, \mathbf{x}_1), \ldots, (\mathbf{p}_t, \mathbf{x}_t)$. We use two experimental settings in our experiments that capture different aspects of OKT: First, we analyze only the first submission to each question, ignoring later attempts. In this setting, knowledge states mostly capture a student's overall mastery of programming concepts. Second, we analyze all code submissions from each student, including multiple consecutive attempts at the same question. In this setting, knowledge states capture not only a student's programming concept mastery but also their debugging skills. We choose not to study only the final attempt since most students were able to submit correct code in the end. See Section B of the Supplementary Material for detailed experimental settings. Additionally, we perform another experiment on predicting student code submissions to new questions that are unseen during training; see Section 3.4 for details.

3.1 Quantitative Results

Table 1 shows the quantitative results evaluating OKT on the CSEDM dataset comparing DKT, AKT, and DKVMN as the KE component, averaged over all students and time steps. Overall, we observe that our initial OKT method performs reasonably well; as a reference, the CodeBLEU value for the examples in Table 3 are 0.8 and 0.65, respectively. Using existing binary-valued KT methods as the KE component significantly outperforms the baseline that relies on a standard text-to-code generation pipeline without this component, which suggests that KT is a key component in student-generated code prediction. Across the two experimental settings, analyzing first submissions leads to higher test loss and lower CodeBLEU score than analyzing all submissions, while performance on the Dist-1 metric does not vary much. These results can be explained by our observation that students rarely make substantial changes to their code across different submissions, often making minor tweaks; therefore, predicting a later code submission given the previous submissions becomes an easier task than predicting the first submission to a new question. Since these metrics are computed over all questions, we break down OKT's performance across questions in Section C of the Supplementary Material; performance varies significantly across questions (between 0.55 and 0.85 on CodeBLEU). This observation suggests that there is considerable room for improvement on the task of exact student code prediction since they have many nuanced variations, which we further illustrate in the qualitative experiments below.

We also see that using using DKT as the KE component of OKT significantly outperforms using AKT and DKVMN on all metrics in both experimental settings, while using AKT also outperforms DKVMN. These results suggest that DKT is more effective than AKT or DKVMN as the KE component of OKT, which also reported in (Zhu et al., 2021) for standard binary-valued KT on programming exercises, likely because DKT relies on a simple and robust LSTM model. In contrast, AKT and DKVMN have complicated model architectures and may require further parameter tuning and/or more training data in the context of OKT; typical binary-valued KT datasets are much larger in scale (up to ∼10M responses (Choi et al., 2020)).

Table 2 shows the quantitative results comparing different OKT designs and training settings with DKT as the KE component on first submissions. First, we see that aligning the knowledge state space with the prompt token embedding space with a learnable linear function is the most effective (with p-value of 0.01 for CodeBLEU), although other alignment functions are only slightly worse. Developing better alignment functions may further improve performance, which we leave for future

		CodeBLEU ↑	**Dist-1** ↑	**Test Loss** ↓
Alignment	add	0.681 ± 0.003	0.423 ± 0.004	0.179 ± 0.006
	average	0.680 ± 0.003	0.425 ± 0.003	0.179 ± 0.006
	weight	0.684 ± 0.008	0.422 ± 0.004	0.182 ± 0.007
	linear	0.696 ± 0.005	0.425 ± 0.004	0.178 ± 0.006
Pre-train LSTM	yes	0.681 ± 0.003	0.423 ± 0.004	0.179 ± 0.006
	no	0.678 ± 0.003	0.425 ± 0.002	0.180 ± 0.004
Pre-train GPT	yes	0.702 ± 0.004	0.423 ± 0.003	0.174 ± 0.003
	no	0.678 ± 0.005	0.415 ± 0.004	0.219 ± 0.006
Multi-task	yes	0.706 ± 0.002	0.423 ± 0.002	0.362 ± 0.008
	no	0.664 ± 0.019	0.426 ± 0.008	0.198 ± 0.009

Table 2: Linearly combining knowledge states and the prompt token embeddings, pre-training both KE and RG components, and using a multi-task loss lead to best OKT performance.

work. Second, we see that pre-training the KE and RG components result in limited improvement in OKT's performance. This result suggests that there are significant differences between (i) the nature of the binary-valued KT task and OKT's exact code prediction task and (ii) code written by professionals and by students who are still learning programming. Third, we see that a multi-task OKT training objective improves both code prediction performance and model robustness in our experiments. (with p-value of 0.018) This result suggests that multi-task learning with multiple objectives helps us learn better representations of the data, i.e., student knowledge state representations, in OKT.

3.2 Interpreting Learned Knowledge States

We now use a case study to show that the knowledge state space learned by OKT captures the variation in the content and structure of student code. Figure 2 visualizes the learned knowledge states, projected to a 2-D space via t-SNE (van der Maaten and Hinton, 2008), for the following question:

`Write a function in Java that implements the following logic: Your cell phone rings. Return true if you should answer it. Normally you answer, except in the morning you only answer if it is your mom calling. In all cases, if you are asleep, you do not answer.`

The right part of Figure 2 shows the knowledge states of all students when they respond to this question, where each dot represents the submission at a time step (a student may have multiple submissions at multiple time steps) and each color represents a student. We see that there are distinct clusters in these knowledge states that correspond to different student code. To further demonstrate this observation, we zoom in into two areas in the knowledge state space, shown in the two small plots on the left part of Figure 2 together with the corresponding actual student code submissions. We clearly see that the codes within each cluster share similar structural and syntactic properties and that codes from different clusters differ significantly. See Section D for a case study on how OKT's knowledge state space captures student code revisions across multiple submissions. These results suggest that the OKT-learned knowledge state space *aligns* with actual student code submissions.

In Figure 3, we compare the learned knowledge state space for OKT against that for existing KT methods. We see that binary-valued DKT learns knowledge states that belong to a few highly overlapping groups with little difference within each group. The KT method in (Mao et al., 2021) that uses code embeddings only as *input* to binary-valued KT learns a slightly more disentangled knowledge state space. In contrast, OKT's knowledge state space is highly informative with obvious clusters that correspond to actual student code. Overall, these results demonstrate that the knowledge state space learned by OKT captures important aspects of programming knowledge for each student. Therefore, OKT has potential in student and instructor-facing tasks such as hint generation and predicting when a student gets stuck and needs help. We can use OKT in a human-in-the-loop process for student modeling: First, OKT can identify clusters among student responses in an *unsupervised* way. Then, instructors and domain experts can *supervise* OKT by providing fine-grained concept or error labels on these clusters to further interpret the latent knowledge state space.

3.3 Knowledge-aware Prediction of Students' Code Submissions

We now use a case study to demonstrate OKT's ability to predict student-submitted code. Similar to most existing text-to-code models (Iyer et al., 2018; Lu et al., 2021), exact prediction of the actual student code is very difficult. However, OKT can still be effective in capturing coding styles and even predicting some error types with the help of the learned knowledge states. Table 3 shows the predicted code vs. actual student code for two questions. For the top example, we see that our generation model is able to predict the student's code structure, capturing their use of *for* loops (instead of another popular choice of *while* loops). In the bottom example, we see that while code prediction for this question is less accurate than for the

predicted code	**actual student code**

`public String zipZap(String str)`
`{`
`  for (int i = 0; i < str.length()- 2; i++)`
`    {`
`    if (str.charAt(i) == 'z' &&`
`        str.charAt(i + 2) == 'p')`
`    {`
`     str.replace("", str.substring(i + 1));`
`    }`
`  }`
` return str;`
`}`
	`public String zipZap(String str)`
	`{`
	`  for (int i = 0; i < str.length() - 2; i++)`
	`  {`
	`    if (str.charAt(i) == 'z' &&`
	`        str.charAt(i + 2) == 'p')`
	`    {`
	`      str.replace("0", str.substring(i + 1));`
	`      return str;`
	`    }`
	`  }`
	`  return str;`
	`}`

`public boolean evenlySpaced(int a, int b, int c)`
`{`
`    int diffone = b - a;`
`    int difftwo = c - b;`
`    if (diffone = difftwo) {`
`        return true;`
`    }`
`    else {`
`        return false;`
`    }`
`}`
	`public boolean evenlySpaced(int a, int b, int c)`
	`{`
	`    int diffone = b - a;`
	`    int difftwo = c - b;`
	`    boolean question = false;`
	`    if (diffone == difftwo) {`
	`        question = true;`
	`        return question;`
	`    }`
	`    else {`
	`        return question;`
	`    }`
	`}`

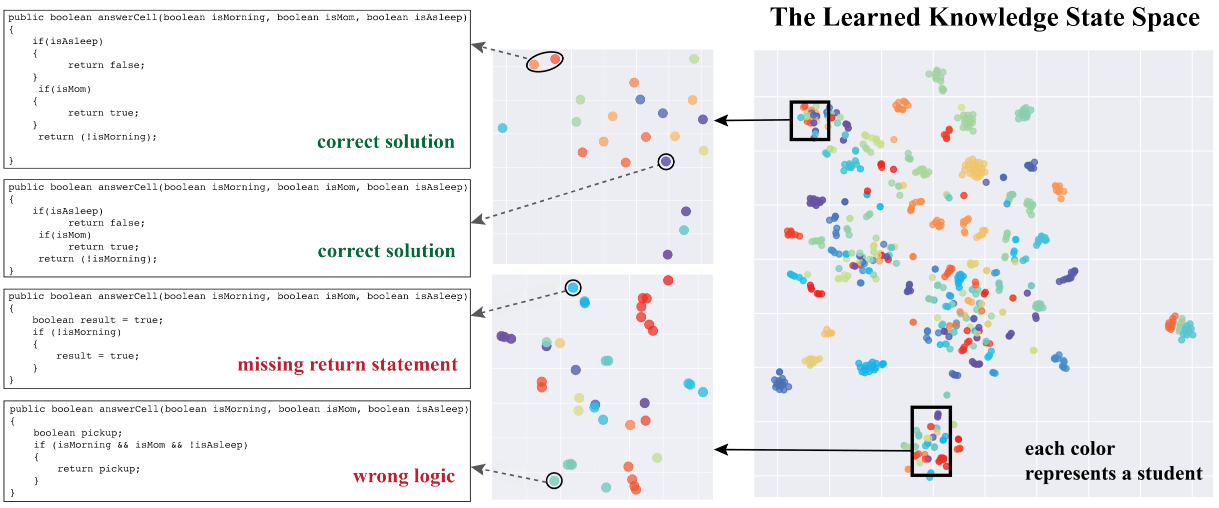
Table 3: OKT generated code vs. actual student code for two questions (differences highlighted in red boxes).

first question, OKT can still capture the main logic and most important parts of the student's actual code. These examples show that OKT can capture both code structure and knowledge gaps on programming concepts for individual students and even predict their possible errors; this capability has much more potential for student and instructor support than standard binary-valued KT methods.

3.4 Generalizing to Unseen Questions

One important limitation of binary-valued KT methods is that they cannot really generalize to new questions; if a question is not present during training, these methods can only predict a student's probability of responding to it correctly using its concept labels (which are often unavailable). On the contrary, OKT's KR and RG components utilize exact question and response content, enabling it to generalize to new questions and predict exact responses to these questions and specific errors. We conduct a preliminary experiment to demonstrate this advantage of OKT: we first remove one question from the dataset (say it occurs at time step t for a student) and then predict the response to this question using the estimated knowledge state $\mathbf{h}_t$. We explore two ways to estimate h_t: i) averaging the knowledge states from neighboring time steps, i.e., $\mathbf{h}_{t-1}$ and $\mathbf{h}_{t+1}$, and ii) using the knowledge state from the previous time step, i.e., $\mathbf{h}_{t-1}$. As a baseline, we also use randomly generated knowledge state vectors to predict the response.

Table 4 shows the average results over removing each question, using DKT on first submissions. We use a smaller amount of epochs for this experiment (10 compared to 25 epochs from Table 1), which explains some of the significant drop in CodeBLEU scores. Nevertheless, OKT still significantly outperforms the baseline approach with no KE compo-



```
public boolean answerCell(boolean isMorning, boolean isMom, boolean isAsleep)
{
    if(isAsleep)
    {
          return false;
    }
     if(isMom)
    {
          return true;
    }
     return (!isMorning);

}
```

correct solution

```
public boolean answerCell(boolean isMorning, boolean isMom, boolean isAsleep)
{
    if(isAsleep)
          return false;
     if(isMom)
          return true;
     return (!isMorning);
}
```

correct solution

```
public boolean answerCell(boolean isMorning, boolean isMom, boolean isAsleep)
{
    boolean result = true;
    if (!isMorning)
    {
        result = true;
    }
}
```

missing return statement

```
public boolean answerCell(boolean isMorning, boolean isMom, boolean isAsleep)
{
    boolean pickup;
    if (isMorning && isMom && !isAsleep)
    {
        return pickup;
    }
}
```

wrong logic

The Learned Knowledge State Space
each color represents a student

Figure 2: Visualization of latent student knowledge states (best viewed in color; each color corresponds to one student) and corresponding actual code. Knowledge states reflect the variation in student-generated code.

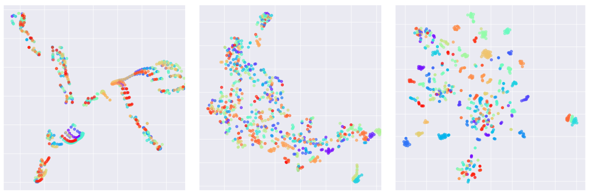


Figure 3: Comparison of the knowledge state spaces learned by DKT (left), DKT with code embeddings as input (Mao et al., 2021) (middle), and OKT (right). OKT learns a knowledge space with distinct clusters that capture variations in actual student code.

Method	CodeBLEU ↑	Dist-1 ↑
Previous	0.484	0.431
Average	0.504	0.419
Random	0.328	0.452

Table 4: OKT's generalization performance to new questions that are unseen during training, using knowledge states from the previous time step, neighboring time steps, and random values.

nent, with averaging knowledge states from neighboring time steps slightly outperforming using the previous time step. Figure 4 visualizes predicted code vs. actual student code embeddings for an unseen question with an average CodeBLEU value of 0.538 over all students. Blue dots correspond to actual student responses and green dots represent RG predicted responses in 2-D, while red dots correspond to pairs of predicted and actual code that are highly similar (76 out of 225). We clearly see that OKT is able to capture the majority of student code variations on this new question from their responses to other questions and left no parts of the code embedding space unaccounted for. OKT's capability of generalizing to new questions can potentially be used to provide feedback to teachers plan homeworks, by predicting typical errors in programming questions that students in their class may exhibit, before assigning them.

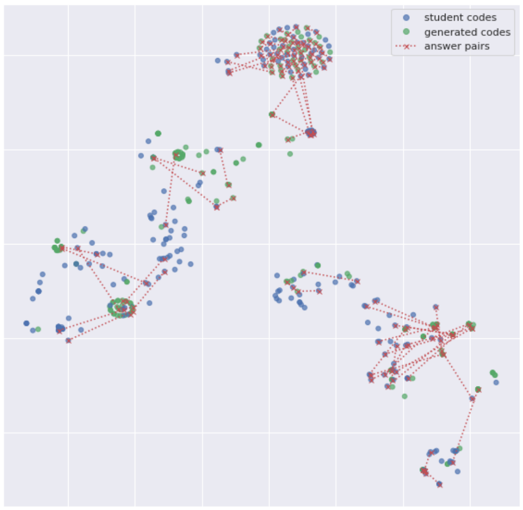


Figure 4: Visualization of actual student code (blue) compared to predicted code (green) for a new question unseen during training. Code pairs that are close in the code embedding space are connected (red).

4 Related Work

Knowledge Tracing: Existing methods for binary-valued KT can be broadly grouped by how they represent the student knowledge level variable, **h**, in Eq. 1. For example, classic Bayesian knowledge tracing methods (Khajah et al., 2014; Pardos and Heffernan, 2010; Yudelson et al., 2013) treat student knowledge as a binary-valued latent variable. The KE and RP components are noisy binary channels, resulting in excellent interpretability of the model parameters. Factor analysis-based methods (Cen et al., 2006; Choffin et al., 2019; Pavlik Jr et al., 2009) use features and latent ability parameters to model student knowledge. The RP component in these methods relies on item response theory models (van der Linden and Hambleton, 2013). More recently, deep learning-based KT methods (Ghosh et al., 2020; Pandey and Srivastava, 2020; Piech et al., 2015a; Shin et al., 2021; Zhang et al., 2017) treat student knowledge as hidden states in neural networks. The KE component often relies on variants of recurrent neural networks (Hochreiter and Schmidhuber, 1997), resulting in models that excel at future performance prediction but have limited interpretability.

Student responses, i.e., $\mathbf{x}$ in Eq. 1, are almost always treated as a binary-valued scalar indicating response correctness. Few methods characterize them as non-binary-valued such as option tracing (Ghosh et al., 2021), which analyzes the exact option students select on each multiple-choice question, and predict partial analysis (Wang and Heffernan, 2013). Questions, i.e., $\mathbf{q}$ in Eq. 1, are often one-hot encoded, either according to question IDs/concept tags, or in a few cases, represented with graph neural networks using question-concept dependencies (Yang et al., 2020). Few existing works use exact question content for $\mathbf{q}$. For example, (Liu et al., 2019; Wang et al., 2021) use pre-trained word embeddings such as word2vec (Mikolov et al., 2013) to encode questions in the RP component. Specifically for programming questions, (Wang et al., 2017; Mao et al., 2021; Zhu et al., 2021) use code representation techniques such as ASTNN (Zhang et al., 2019) and code2vec (Alon et al., 2019) to convert student code into vectors and use them as input to the KE component.

Program Synthesis and Computer Science Education: Program synthesis from natural language instructions (Desai et al., 2016) has attracted significant recent interest since pre-trained language models (Chen et al., 2021) or language model architectures (Li et al., 2022) have demonstrated their effectiveness on hard tasks such as solving coding challenge problems (Hendrycks et al., 2021a). These methods are pre-trained on large datasets containing publicly available code on the internet, which is primarily written by skilled programmers. There is a line of existing work on analyzing student-generated code, most noticeably using the Hour of Code dataset released by Code.org (Piech et al., 2015b; Singla and Theodoropoulos, 2022; Wang et al., 2017), for tasks such as error analysis and automated feedback generation that are meaningful in computer science education settings.

5 Conclusions and Future Work

In this paper, we have proposed a framework for open-ended knowledge tracing (OKT) to track student knowledge acquisition while predicting their full responses to open-ended questions. We have demonstrated how OKT can be applied to the computer science education domain, where we analyze students' code submissions to programming questions. We addressed the key technical challenge of integrating student knowledge representations into code generation methods, e.g., text-to-code models based on fine-tuning GPT-2. Our experiments on real-world computer science student data indicate that OKT has considerable promise for tracking and predicting student mastery and performance.

There are many avenues for future work. First, we can use code standardization techniques (Rivers and Koedinger, 2017) to further pre-process student code using semantic equivalence. Second, we can explore the applicability of OKT to other domains such as mathematics, where many pre-trained models for mathematical problem solving have been developed (Cobbe et al., 2021; Hendrycks et al., 2021b; Saxton et al., 2019) and explore whether students consistently exhibit certain errors (VanLehn, 1982). Third, we can develop knowledge tracing models that capture more specific aspects of knowledge, i.e., debugging skills, which is reflected in the *change* in student code across submissions to the same question after receiving automated feedback generated by the compiler or test cases. Fourth, we can further enhance the validity and interpretability of OKT by adding more human supervision, such as adding an additional loss on the test case scores of generated code. We can also use instructor- or expert-provided labels on student errors to make the latent knowledge state space more informative. Finally, we can further evaluate our framework on tasks relevant to instructor feedback, including compilation/runtime error category prediction and test case outcome prediction; see Section E in the Supplementary Material for a detailed discussion.

Acknowledgements

We thank the reviewers for their constructive feedback. This work is supported by NSF grants 1842378, 1937134, 1917713, 2118706, 2202506, ONR grant N0014-20-1-2534, AFOSR grant FA9550-18-1-0478, ONR grant N00014-18-1-2047, and a Vannevar Bush Faculty Fellowship.

Limitations

Being the first attempt at the task of predicting the exact content of open-ended student responses, OKT has several obvious limitations. First, the ability to predict variation in student responses depends on the fine-tuned language model's ability to generate correct responses given the question statement. Therefore, it is not clear whether OKT can generalize to domains where language models have not been shown to be highly accurate at generating correct open-ended responses. Second, OKT requires a large amount of student coding data, which may limit its applicability to learning platforms in their early stages that do not have a large number of student users. Third, the open-ended response generation process is sequential and can be time-consuming, which may limit OKT's ability to support instructors and students in real time in real-world computer science education scenarios.

Ethics Statement

Our work should be seen as exploratory rather than a finished tool that can readily be deployed in real-world computer science educational scenarios. Since OKT requires training on a large amount of student-generated code, there is a need to systematically study any potential negative biases towards underrepresented student populations. The effectiveness of exact open-ended response prediction in helping instructors adjust their instruction and benefit students remains to be seen, which requires principled evaluation using A/B testing.

References

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):1–29.

Suyeong An, Junghoon Kim, Minsam Kim, and Juneyoung Park. 2022. No task left behind: Multi-task learning of knowledge tracing and option tracing for better student assessment.

John R Anderson and Robin Jeffries. 1985. Novice lisp errors: Undetected losses of information from working memory. *Human–Computer Interact.*, 1(2):107–131.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

John Seely Brown and Richard R Burton. 1978. Diagnostic models for procedural bugs in basic mathematical skills. *Cogn. sci.*, 2(2):155–192.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Hao Cen, Kenneth Koedinger, and Brian Junker. 2006. Learning factors analysis–A general method for cognitive model evaluation and improvement. In *Proc. Int. Conf. Intell. Tutoring Syst.*, pages 164–175.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Benoît Choffin, Fabrice Popineau, Yolaine Bourda, and Jill-Jênn Vie. 2019. DAS3H: Modeling student learning and forgetting for pptimally scheduling distributed practice of skills. In *Proc. Int. Conf. Educ. Data Mining*, pages 29–38.

Youngduck Choi, Youngnam Lee, Dongmin Shin, Junghyun Cho, Seoyon Park, Seewoo Lee, Jineon Baek, Chan Bae, Byungsoo Kim, and Jaewe Heo. 2020. Ednet: A large-scale hierarchical dataset in education. In *Int. Conf. Artif. Intell. Educ.*, pages 69–73. Springer.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Albert Corbett and John Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Model. User-adapted Interact.*, 4(4):253–278.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proc. 38th Int. Conf. Softw. Eng.*, pages 345–356.

Molly Q Feldman, Ji Yong Cho, Monica Ong, Sumit Gulwani, Zoran Popović, and Erik Andersen. 2018. Automatic diagnosis of students' misconceptions in K-8 mathematics. In *Proc. CHI Conf. Human Factors Comput. Syst.*, pages 1–12.

Junchen Feng, Bo Zhang, Yuchen Li, and Qiushi Xu. 2019. Bayesian diagnosis tracing: Application of procedural misconceptions in knowledge tracing. In *Proc. Int. Conf. Artif. Intell. Educ.*, pages 84–88. Springer.

Aritra Ghosh, Neil Heffernan, and Andrew S Lan. 2020. Context-aware attentive knowledge tracing. In *Proc. ACM SIGKDD*, pages 2330–2339.

Aritra Ghosh, Jay Raspat, and Andrew Lan. 2021. Option tracing: Beyond correctness analysis in knowledge tracing. In *Int. Conf. Artif. Intell. Educ.*, pages 137–149. Springer.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021a. Measuring coding challenge competence with apps. In *Thirty-fifth Conf. Neural Inf. Process. Syst. Datasets and Benchmarks Track (Round 2)*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021b. Measuring mathematical problem solving with the math dataset. In *Proc. NeurIPS*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

MM Khajah, Y Huang, JP González-Brenes, MC Mozer, and P Brusilovsky. 2014. Integrating knowledge tracing and item response theory: A tale of two frameworks. In *Proc. Int. Workshop Personalization Approaches Learn. Environ.*, volume 1181, pages 7–15.

Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. *arXiv preprint arXiv:1904.02660*.

Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. 2016. A diversity-promoting objective function for neural conversation models. In *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Human Lang. Technol.*, pages 110–119.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Qi Liu, Zhenya Huang, Yu Yin, Enhong Chen, Hui Xiong, Yu Su, and Guoping Hu. 2019. Ekt: Exercise-aware knowledge tracing for student performance prediction. *IEEE Trans. Knowl. Data Eng.*, 33(1):100–115.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Ye Mao, Yang Shi, Samiha Marwan, Thomas W Price, Tiffany Barnes, and Min Chi. 2021. Knowing" when" and" where": Temporal-astnn for student learning progression in novice programming tasks. *Int. Educ. Data Mining Soc.*

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proc. NeurIPS*, pages 3111–3119.

Shalini Pandey and Jaideep Srivastava. 2020. Rkt: Relation-aware self-attention for knowledge tracing. *arXiv preprint arXiv:2008.12736*.

Zach A Pardos and Neil T Heffernan. 2010. Modeling individualization in a Bayesian networks implementation of knowledge tracing. In *Proc. Int. Conf. User Model. Adaptation Personalization*, pages 255–266.

Philip Pavlik Jr, Hao Cen, and Kenneth Koedinger. 2009. Performance factors analysis–A new alternative to knowledge tracing. In *Proc. Int. Conf. Artif. Intell. Educ.*

Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. 2015a. Deep knowledge tracing. In *Proc. NeurIPS*, pages 505–513.

Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015b. Autonomously generating hints by inferring problem solving policies. In *Proc. ACM conf. learn. Scale*, pages 195–204.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297*.

Steven Ritter, John R Anderson, Kenneth R Koedinger, and Albert Corbett. 2007. Cognitive tutor: Applied research in mathematics education. *Psychonomic Bulletin & Review*, 14(2):249–255.

Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *Int. J. Artif. Intell. Educ.*, 27(1):37–64.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. 2019. Analysing mathematical reasoning abilities of neural models. *arXiv preprint arXiv:1904.01557*.

Dongmin Shin, Yugeun Shim, Hangyeol Yu, Seewoo Lee, Byungsoo Kim, and Youngduck Choi. 2021. Saint+: Integrating temporal features for ednet correctness prediction. In *11th Int. Learn. Analytics Knowl. Conf.*, pages 490–496.

Adish Singla and Nikitas Theodoropoulos. 2022. From {Solution Synthesis} to {Student Attempt Synthesis} for block-based visual programming tasks. *arXiv preprint arXiv:2205.01265*.

John P Smith III, Andrea A DiSessa, and Jeremy Roschelle. 1994. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *j. learn. sci.*, 3(2):115–163.

W. J. van der Linden and R. K. Hambleton. 2013. *Handbook of Modern Item Response Theory*. Springer Science & Business Media.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *JMLR*, 9(86):2579–2605.

Kurt VanLehn. 1982. Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills. *J. Math. Behavior*.

Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. 2017. Learning to represent student knowledge on programming exercises using deep learning. *Int. Educ. Data Mining Soc.*

Tianqi Wang, Fenglong Ma, Yaqing Wang, Tang Tang, Longfei Zhang, and Jing Gao. 2021. Towards learning outcome prediction via modeling question explanations and student responses. In *Proc. SIAM Int. Conf. Data Mining*, pages 693–701. SIAM.

Yutao Wang and Neil Heffernan. 2013. Extending knowledge tracing to allow partial credit: Using continuous versus binary nodes. In *Int. conf. artif. intell. educ.*, pages 181–188. Springer.

Yang Yang, Jian Shen, Yanru Qu, Yunfei Liu, Kerong Wang, Yaoming Zhu, Weinan Zhang, and Yong Yu. 2020. Gikt: A graph-based interaction model for knowledge tracing. In *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*.

Michael V Yudelson, Kenneth R Koedinger, and Geoffrey J Gordon. 2013. Individualized bayesian knowledge tracing models. In *Int. Conf. artif. intell. educ.*, pages 171–180. Springer.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *IEEE/ACM 41st Int. Conf. Software Eng.*, pages 783–794. IEEE.

Jiani Zhang, Xingjian Shi, Irwin King, and Dit-Yan Yeung. 2017. Dynamic key-value memory networks for knowledge tracing. In *Proc. Int. Conf. World Wide Web*, pages 765–774.

Renyu Zhu, Dongxiang Zhang, Chengcheng Han, Ming Gao, Xuesong Lu, Weining Qian, and Aoying Zhou. 2021. Programming knowledge tracing: A comprehensive dataset and a new model. *arXiv preprint arXiv:2112.08273*.

A Dataset Statistics and Preprocessing Steps

Statistic	Raw	Processsed
#codes	46825	39796
#avg. lines of code per submission	17.52	17.64
#avg. submissions per student	190.34	161.77
#avg. submissions per problem	936.5	795.9

Table 5: Dataset statistics comparing the raw and our processed dataset, the latter of which is used throughout our experiments.

Since we choose to use the AST representation for code, we perform a preprocessing step to remove student solutions that cannot be converted to AST format. Overall, about 85% of all student solutions are AST-convertible, which means that this preprocessing step does not result in significant data loss. Table 5 describes the summary statistics of the original dataset and the resulting preprocessed dataset that we use for all our experiments. For KT, we follow standard procedure in the literature (Piech et al., 2015a; Ghosh et al., 2020; Zhang et al., 2017) by setting the maximum solution sequence for any student to 200. For students with more than 200 solutions, we split their solutions into separate sequences of length 200.

B Experimental Setup Details

In both settings, we split all students in the dataset into disjoint (train, validation, test) sets with a $80\% - 10\% - 10\%$ ratio and report all metrics on the test set. In the default setting, we add knowledge states into token embeddings for the question prompt as input to the RG component using a linear projection layer as detailed above. We pre-train both the KE and RG components of OKT on the training data with a correctness prediction objective (i.e., pre-train an existing KT method) and a prompt-to-code supervised generation objective, respectively. For the KE component, we follow the original DKT, DKVMN, and AKT methods with 768 hidden units in their models. When combining DKVMN or AKT with the answer generator, we use the context reader output from DKVMN and the hidden state from AKT, respectively, as input to the answer generator at each time step. We refer readers to (Zhang et al., 2017; Ghosh et al., 2020) for more details. For the RG component, we use a small GPT-2 with 12 transformer decoder layers (Radford et al., 2019). We use the RMSProp optimizer for the knowledge update component and the Adam optimizer for the RG component, both with a default learning rate of 0.00001. Also, we freeze the parameters of the question and code representation models and only train the KT model and the answer prediction model, Although the former two components can also be optimized.

We run all experiments using a single NVIDIA Quadro RTX 8000 GPU. The KT model pre-training usually takes less than 5 minutes per epoch of wall clock time. The OKT training with DKT as the KT model takes about 10 minutes and 30 minutes per epoch of wall clock time for the the two scenarios, namely, using only students' first submitted code and all submitted code that can be converted to AST format, respectively. OKT training with AKT as the KT model takes about the same time as DKT as the KT model while with DKVMN, training is about 1.5 times slower due to the more expensive memory computation (Zhang et al., 2017).

C Visualizing Quantitative Results

Following the results in Section 3.1 and Table 1, we additionally examine the model performance across questions and measure the correlation between its CodeBLEU score and some features (i.e. difficulty level, response diversity). Figure 5 shows that model performance has a positive correlation with student performance, i.e., the portion of correct responses, and a negative correlation with the number of student responses. In other words, an easy question with fewer submissions is more likely to achieve better prediction results. However, CodeBLEU is minimally correlated with the diversity in student responses. Also, the range of CodeBLEU performance across questions is relatively big, with the highest of 0.86 and lowest of 0.56.

D Visualizing Code Revisions

We also show how the learnt knowledge state space can be useful for tracing and understanding students' consecutive submissions to the same question. On the right-hand side of Figure 6, we show the knowledge state trajectories of two students responding to this question. The colors in these two figures represent knowledge states that correspond to wrong, partially correct and fully correct codes at different time steps. We see that both students start with a wrong solution. However, one student gradually proceeded to the correct solution after

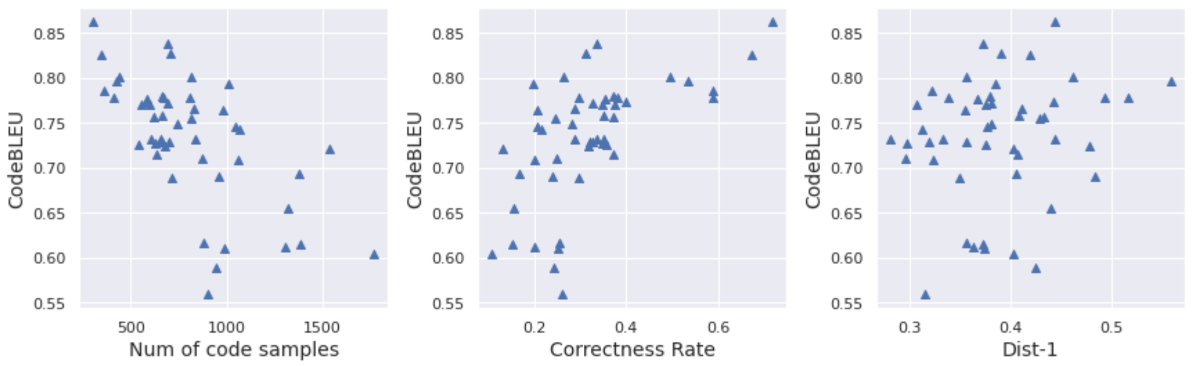
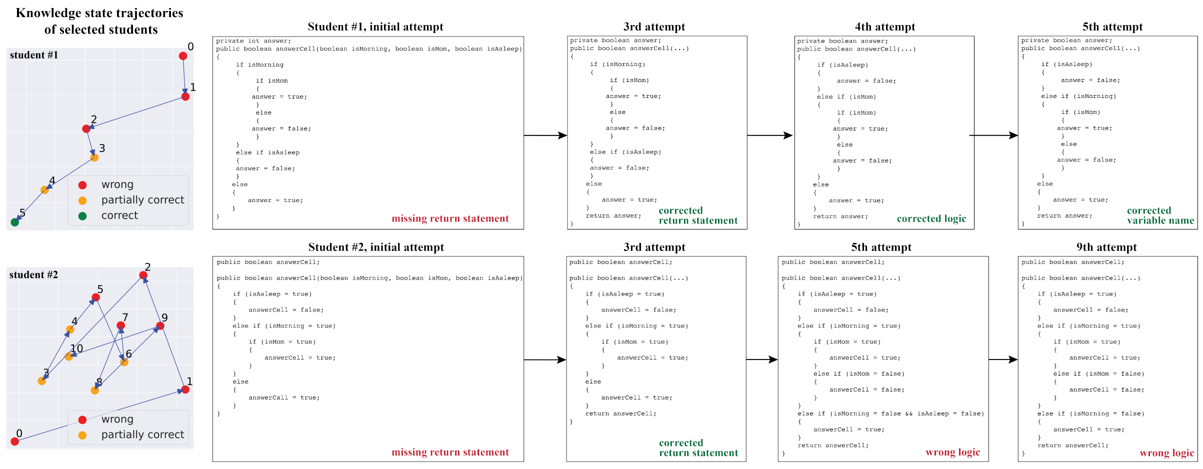


Figure 5: Visualization of CodeBLEU metric versus number of student responses (left), rate of correct submissions (middle) and Dist-1 metric (right) in each question. Each point represents one question.



Student #1, initial attempt

```
private int answer;
public boolean answerCell(boolean isMorning, boolean isMom, boolean isAsleep)
{
     if isMorning
     {
          if isMom
          {
         answer = true;
          }
          else
          {
         answer = false;
          }
     }
     else if isAsleep
     {
     answer = false;
     }
    else
    {
        answer = true;
    }
}
```

3rd attempt

```
private boolean answer;
public boolean answerCell(...)
{
     if (isMorning)
     {
          if (isMom)
          {
         answer = true;
          }
          else
          {
         answer = false;
          }
     }
     else if (isAsleep)
     {
     answer = false;
     }
    else
    {
        answer = true;
    }
    return answer;
}
```

4th attempt

```
private boolean answer;
public boolean answerCell(...)
{
     if (isAsleep)
     {
          answer = false;
     }
     else if (isMom)
     {
          if (isMom)
          {
         answer = true;
          }
          else
          {
         answer = false;
          }
     }
    else
    {
        answer = true;
    }
    return answer;
}
```

5th attempt

```
private boolean answer;
public boolean answerCell(...)
{
     if (isAsleep)
     {
          answer = false;
     }
     else if (isMorning)
     {
          if (isMom)
          {
         answer = true;
          }
          else
          {
         answer = false;
          }
     }
    else
    {
        answer = true;
    }
    return answer;
}
```

Student #2, initial attempt

```
public boolean answerCell;

public boolean answerCell(boolean isMorning, boolean isMom, boolean isAsleep)
{
    if (isAsleep = true)
    {
        answerCell = false;
    }
    else if (isMorning = true)
    {
        if (isMom = true)
        {
            answerCell = true;
        }
    }
    else
    {
        answerCall = true;
    }
}
```

3rd attempt

```
public boolean answerCell;

public boolean answerCell(...)
{
    if (isAsleep = true)
    {
        answerCell = false;
    }
    else if (isMorning = true)
    {
        if (isMom = true)
        {
            answerCell = true;
        }
    }
    else
    {
        answerCell = true;
    }
    return answerCell;
}
```

5th attempt

```
public boolean answerCell;

public boolean answerCell(...)
{
    if (isAsleep = true)
    {
        answerCell = false;
    }
    else if (isMorning = true)
    {
        if (isMom = true)
        {
            answerCell = true;
        }
        else if (isMom = false)
        {
            answerCell = false;
        }
    }
    else if (isMorning = false && isAsleep = false)
    {
        answerCell = true;
    }
    return answerCell;
}
```

9th attempt

```
public boolean answerCell;

public boolean answerCell(...)
{
    if (isAsleep = true)
    {
        answerCell = false;
    }
    else if (isMorning = true)
    {
        if (isMom = true)
        {
            answerCell = true;
        }
        else if (isMom = false)
        {
            answerCell = false;
        }
    }
    else if (isMorning = false)
    {
        answerCell = true;
    }
    return answerCell;
}
```

Figure 6: Four sample submissions of two students corresponding to the top right figures, respectively. One student gradually proceeded to a correct code while the other got stuck.

a few edits, whereas the other student got stuck after a few unsuccessful edits and eventually gave up on solving this prompt. The steady progress versus getting stuck is clearly visualized in these figures, where for the former student, the knowledge states gradually moves from the upper right corner in the knowledge state space to the lower left, whereas for the latter student, the knowledge states circle around and bounce back and forth in the space. We also show four selected submissions by each student during their response process, further illustrating how the first student made steady progress, i.e., adding the return statement (first two code submissions) and correcting logic (last two code submissions), and how the second student got stuck, i.e., making reasonable changes initially but then some repetitive edits.

E Real-World Use Cases and Implications

One crucial highlight of our work is that, through OKT, we can predict students' responses to open-ended questions *before actually assigning them.* On the contrary, existing student and teacher support tools can only be applied *after* observing their responses. Therefore, OKT can be used in practice in many ways by anticipating student errors and struggles ahead of time. For example, for teacher support, we can use OKT to provide diagnosis information to teachers via a dashboard. For any open-ended question that the teacher wants to assign to their class, we can predict the responses that each student will write given their current knowledge states and show teachers clusters that represent typical errors. This way, the teacher can anticipate student performance, switch to an easier (or more challenging) question if necessary, and prepare feedback for individual students ahead of time. For student support, if a student struggles, we can use OKT to find other students stuck in a similar place but ultimately succeeded in answering the question and provide incremental hints or feedback on their errors. These advantages over traditional KT methods will potentially enable OKT to become the next-generation workhorse for large-scale, intelligent educational systems.