Code-DKT: A Code-based Knowledge Tracing Model for Programming Tasks

Yang Shi, Min Chi, Tiffany Barnes, Thomas W. Price
North Carolina State University
Raleigh, NC, USA
{yshi26, mchi, tmbarnes, twprice}@ncsu.edu

ABSTRACT
Knowledge tracing (KT) models are a popular approach for predicting students' future performance at practice problems using their prior attempts. Though many innovations have been made in KT, most models including the state-of-the-art Deep KT (DKT) mainly leverage each student's response either as correct or incorrect, ignoring *its content*. In this work, we propose **Code-based Deep Knowledge Tracing (Code-DKT)**, a model that uses an attention mechanism to automatically extract and select domain-specific code features to extend DKT. We compared the effectiveness of Code-DKT against Bayesian and Deep Knowledge Tracing (BKT and DKT) on a dataset from a class of 50 students attempting to solve 5 introductory programming assignments. Our results show that Code-DKT consistently outperforms DKT by $3.07 - 4.00\%$ AUC across the 5 assignments, a comparable improvement to other state-of-the-art domain-general KT models over DKT. Finally, we analyze problem-specific performance through a set of case studies for one assignment to demonstrate when and how code features improve Code-DKT's predictions.

Keywords
Knowledge Tracing, Deep Knowledge Tracing, CS Education, Code Analysis, Deep Learning

1. INTRODUCTION
Modeling student knowledge to predict performance on future problems, called Knowledge Tracing (KT), is a fundamental feature of intelligent tutoring systems [50]. KT models enable tutoring systems to support mastery learning [14], select appropriate next problems [1], provide help [46], and provide analytics to instructors [34], all of which can improve learning. KT models have increased in complexity from the early 4-parameter Bayesian Knowledge Tracing (BKT) to modern models that train deep neural networks with tens of thousands of parameters using the latest deep learning innovations (e.g. attention [54] and transformers [51]). This has led to improvement in KT model performance, especially for larger datasets, e.g. from ASSISTments [41, 17].

The *simplest version* of the KT problem uses only the sequence of: 1) which problems the student has attempted, and 2) whether or not each attempt was correct. While this makes KT models widely applicable across domains, this also omits a potential wealth of information about *how* the student attempted each problem. Increasingly, ITS being built to support complex problem solving tasks, like programming in Snap [35] and in games [22], logic proofs [30], science inquiry [25] and language learning [47]. In these domains, correctness may not provide enough information about student knowledge, varying significantly in the reasons both for incorrectness and correctness. In programming, for example, one incorrect attempt may have a minor syntax error while another includes a clear misconception. Similarly, two different *correct* answers could reveal dramatically different levels of concept mastery depending on their conciseness and the concepts used. Most KT models would treat all correct and all incorrect attempts identically. A domain-specific KT model, e.g. those for science by Rowe et al. [40], might greatly improve KT performance. Little work has investigated whether domain-general KT models can predict student success in programming, or how domain-specific features might improve performance.

In this paper, we explore when and how features extracted from students' submitted code can improve a KT model for programming. To do so, we introduce a novel code-based deep knowledge tracing (Code-DKT) model, which uses the code2vec model [4] to learn a meaningful representation of student code, and combines this with Deep Knowledge Tracing (DKT) [34] to track student progress. Specifically, student code submissions are represented with abstract syntax trees, and split into multiple code paths [4] (explained in Section 3). We assign the importance of different code paths by learning weights guided by the scores students received for the current and past submissions. We compared the performance of Code-DKT with baseline BKT and DKT models on a dataset of 50 introductory programming problems from 410 students, across 5 assignments. Our experiments show that the Code-DKT model is able to consistently improve DKT's performance by 3.08-4.00 percentage points in AUC. This improvement is comparable to that of other modern KT models over DKT (2-4%) [31, 44], suggesting that domain-specific features may be just as important as model structure. Finally, we investigate one assignment through 3

case studies to explore the mechanisms by which code features may improve the model, and when they are most useful. We also show that Code-DKT outperforms more naive code-feature models. Overall, this paper makes three contributions: 1) the Code-DKT model, which extends DKT for programming tasks; 2) evidence that Code-DKT outperforms both domain-general models and naive code-feature models; and 3) evidence of when and how Code-DKT's code features improve model performance.

2. RELATED WORK

In this section, we present related work on knowledge tracing, student modeling in computer science education, and deep learning models for code/programs.

2.1 Knowledge Tracing

Knowledge tracing (KT) models student knowledge as they solve problems to predict future performance. In KT, problems are labeled with needed skills (i.e. knowledge components, KC) [49], the skill or q-matrix can be learned from data [9], or the problem ID can be used instead. In Bayesian Knowledge Tracing (BKT), the most popular KT method [14], a simple Bayesian model is built to model student knowledge using parameters for guess (getting a problem right when a skill is not known), slip (getting it wrong when known), and transition from unlearned to learned after practicing. These parameters are learned from prior students' problem sequences, and then used to predict future performance. Researchers have improved BKT performance, for example, by calculating the bound or prior distribution of parameters [8], adding a priori estimates of student learning [32], or integrating speed factors [56].

A number of innovations have improved *domain general* KT, without using additional features from student's work (only the correctness of each problem attempt). With the development of machine learning technologies and increasingly available large datasets, models based on deep learning have been proven more effective, especially with enough available data [17]. Piech et al. introduced deep knowledge tracing (DKT), using recurrent neural networks (RNN) to predict a student's knowledge of each skill (or problem) after each problem attempt, and to learn the relationships among skills automatically [34]. As our work is based on this model, we will discuss the details of the model in Section 3. Some recent advances in deep learning for knowledge tracing focus on model structure, including SAKT and SAINT. Self-attentive knowledge tracing (SAKT) [31] added a self-attention mechanism [54] to DKT, while Separated Self-Attentive Neural Knowledge Tracing (SAINT) [12] later integrated a transformer (a type of deep neural network which has been successfully applied in text and image processing areas) into a knowledge tracing model [51]. Both of these models have outperformed DKT, especially on large datasets such as EdNet [13], e.g. by 2% AUC.

While these innovations have improved KT performance, often using complex networks and larger datasets, the datasets used generally only indicate whether a student's attempt was *correct*, but not the *content* of a student's answer or their *process* for achieving it, and the models therefore do not use this information. However, researchers have incorporated *other types* of information into deep models, such as course prerequisites or the relationships among problems. For instance, Chen et al. attached prerequisite information in the DKT modeling process for a more accurate prediction[10]. The prerequisite concepts were modeled as graph matrices (as done by Wang et al. [53]), serving as an additional input to knowledge tracing models, similar to skill or q-matrices that can also be learned from student data [9]. On the other hand, Ghosh et al. introduced attentive knowledge tracing (AKT) [18]. They introduced a decay parameter to explicitly reduce the impact of distant problems, and at the same time used a Rasch model [37] to incorporate problem contexts, then embedding the differences among the problems. Student information can also be used for knowledge tracing models. Educational priors such as a learning or forgetting curves can be integrated into deep knowledge tracing models [11]. The closest such models come to incorporating students' solution processes is including information about how *fast* students solved a problem. Yudelson et al. [56] added speed factors into BKT, and similar temporal information can also improve the performance of deep models (e.g. [44]).

None of the above work have used the student response information, besides submission correctness, in their models. This could be partially because of the simplicity of the problems. Most of them are true or false, multiple choice problems, or short answer problems. The availability of the exercise data is also limited, as some datasets only contains a sequence of binary correctness scores from students. Recent work (e.g. EKT, EERNN [45, 26]) used a joint embedding of *exercise* text and response correctness, combining the exercise text embedding together with student scores to represent student individualized submissions. This achieved better performance than the other models without using this information. However, these models only use *problem information*, but no information about the *students' answer* beyond binary correctness information. This suggests an opportunity to create improved, *domain-specific* KT models in areas such as programming, math, science or writing, where students' answers include complex written responses or structured problem-solving steps. Recent work has incorporated such problem-specific data in deep learning approaches used to adapt pedagogical policies for tutoring in logic [27], probability [58], or predict performance in programming [29] but generally have not been used to built KT models in these domains. In the domain of programming education, for example, students' code submissions contain rich information on the state of their current knowledge. As proposed in this paper, more structural information could be extracted from student code submission to infer students' learning status of certain concepts. We use these code features to make better knowledge tracing models.

2.2 Student Modeling in CS Education

Researchers in CS education have explored ways to model student source code for intelligent tutoring. In 2011, Jin et al. proposed that a linkage representation that reflected code structure could be used for programming hint generation [23]. In 2014, Yudelson et al. extracted code features from a MOOC on introductory Java programming to explore code recommendation methods [55]. Their work focused on using a combination of problem correctness and extracted code features to predict student success, and use this prediction to recommend an appropriate next problem to a stu-

dent. While they did not evaluate their model on a KT task per se, their approach of extracting atomic code features is somewhat similar to our TFIDF baseline (Section 4.3). Another work from Rivers et al. used code features for student learning curve analysis and attempted to directly extract meaningful knowledge components (and whether they were successfully applied) from student code [39]. In their work, student code submissions are represented as abstract syntax trees (ASTs), with the node types of ASTs (e.g. `for`, `if`) representing knowledge components (KCs). The error rate curves (referred to as "learning curves") were plotted over time, visualizing the mastery of different KCs. They showed that while code-based KCs produced well-fitting curves, others did not. While this suggests the possible validity of AST-based KC extraction, the work did not directly evaluate the utility of these KCs for knowledge tracing. Like our current work, Wang et al. showed that incorporating structural code features can improve DKT for a single problem from a large "hour of code" (HoC) dataset [52]. However, this HoC exercise has a very simple solution, so their results may not generalize. Additionally, their features were learned in an unsupervised way from ASTs, while our approach learns an embedding from the data.

Code features have also been used in tasks other than KT as well, such as common bug identification in student code. Traditionally, experts manually examined student code to identify common bugs in different student levels and programming languages, such as Java [48] or block based programs [20]. However, manual examination is expensive for large-scale and quantitative studies. More advanced work takes advantage of the growing size of datasets, and used data-driven methods to find bugs in student code submissions. For example, Choi et al. used simple machine learning methods to detect malicious code in code by using simple feature extraction methods such as counting neighboring tokens in code text (n-gram). With the recent advance of computational power and even bigger datasets, more deep learning methods have emerged. These methods focused on developing deep neural network methods to extract structural information for automatic student bug detection. For example, Gupta et al. used a matrix to represent the ASTs of student code to localize student code submissions [19] in a large dataset (270K samples). For smaller sized dataset, Shi et al. evaluated the bug detection performance with the help of semi-supervised learning [42], and have also shown that unsupervised learning is possible with the help of experts [43]. All these methods reported better performance than traditional data-driven models on their tasks, showing the feasibility of similar usage on KT tasks.

While we focus on using student code submissions to extract features for student programming KT tasks, other less complicated approaches exist. Original programming tutors such as ACT [15] and Lisp tutor [6] decompose computational problems into small steps and let students make choices. This facilitates the KT tasks, as in these datasets, student submissions are simple multiple choices. However, with the development of newer Intelligent Tutoring Systems (ITSs), more systems provide intelligent support to students' written code. This provides better practice for students, but also makes knowledge tracing in computer science a more challenging task. Our paper aims at extracting code features for KT tasks in these new datasets.

2.3 Deep Code Learning

Besides code feature extraction in the CS education domain, programming code has also been analyzed with data-driven models in software engineering research. For example, Allamanis et al. used neighboring tokens in source code (n-grams) to represent programming code, borrowing methods from natural language processing studies to predict method names in big code datasets [3]. Later work further explored extracting features from code structure, such as Raychev et al. who used decision trees to model programming code, making probabilistic predictions on the types of nodes in AST [38]. However, these simple structural approaches are often outperformed by newly developed deep learning models, especially when applied to big datasets.

Deep neural networks have been applied in the software engineering domain, and achieved better performance than traditional data-driven methods. For example, Allamanis et al. used convolutional neural networks (CNNs) to classify code functions [2]; Mou et al. reworked the CNNs to an AST version, using the parent-children direction information in tree representations. Both methods greatly improved method classification tasks on classical machine learning models. Another recent model, code2vec, outperformed these models. Alon et al. designed this model, which leverages nodes and traversal paths in the ASTs to represent programs [5]. In their work, the leaf nodes of the ASTs are selected to represent the semantic information about the code. In addition, as there is a path through the AST from every leaf node to any other leaf node, this path is extracted to represent the code's structural information. The traversal paths together with the corresponding leaf nodes serve as the basic units of a representation of code [4]. The code2vec model calculates the weight of each code path using an attention mechanism [54] to automatically classify function names. Code-DKT's code extraction component is based on the code2vec model, but adds score to the attention mechanism to assign weights to code paths [4] for predictions.

We chose code2vec to represent student code in DKT due to its recent successes for modeling code, and its attention mechanism. The attention mechanism learns weights for different features, allowing the model to directly use score information to select the most predictive code paths. Future work could investigate other code representations such as ASTNN, which has also been applied to make predictions from student code [28], or more recent advances such as CodeBERT [16].

3. METHOD

Problem Definition: Knowledge tracing (KT) tasks model a prediction problem: Given the history of a student's attempts at various KCs/problems, the model predicts if the student will succeed on their next attempt [1]. Specifically, we define each student attempt $\mathbf{x}_t$ at time t as (q_t, a_t, c_t), where q_t is the problem ID, a_t is the correctness, and c_t is the program code submitted for this attempt. Historically, KT algorithms have only utilized q_t and a_t, and in this work we extend the input sequence to include c_t. At each timestep T,

[1]We use problemIDs for KCs in this work

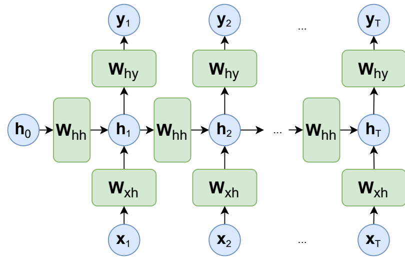

Figure 1: Recurrent neural network structure.

the model is given the T-length student attempt sequence $\mathcal{S}_T = \{(q_1, a_1, c_1), (q_2, a_2, c_2), ..., (q_T, a_T, c_T)\}$, and it predicts whether the student's next attempt $(T+1)$ on a given problem (q_{T+1}) will be correct (a_{T+1}). Note that students may attempt problems multiple times, and the model will make a prediction at each attempt.

Our proposed Deep Code Knowledge Tracing (Code-DKT) model integrates deep knowledge tracing (DKT) [34]) with the code2vec classification algorithm [5]. In this section we introduce the DKT model and how we enhance it with code feature extraction and selection.

3.1 Deep Knowledge Tracing

Deep knowledge tracing uses a recurrent neural network (RNN) structure to learn the probability that a student will make a correct attempt on a subsequent problem. In the original implementation of DKT, the authors also implemented a version of DKT using a long short term memory (LSTM) model [21], which is widely perceived as an advancement over RNNs. For simplicity, we explain DKT using an RNN model; we performed DKT using both RNNs and LSTMs. In the experiments, the LSTM version yielded higher performance[2] (see performance comparison in Section 5.1.4). We chose DKT as our baseline model, to compare with and to extend, as it is a commonly used baseline in other more recent KT papers [31, 44]. Further, its LSTM structure makes it straightforward to extend with code features and to directly evaluate those features' contributions. Some recent models have outperformed DKT, but only by about 2-4% AUC [31, 44], suggesting that DKT is still representative of modern deep KT models.

Model Input: For each student, DKT (RNN) takes as input a sequence $\mathcal{S} = \{\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_T\}$ of T attempt vectors $\mathbf{x}_t$. With M problems, each attempt consisting of problem-correctness pair $\{q_t, a_t\}$ at time t, is one-hot encoded into a binary vector $\mathbf{x}_t$ of size $2M$, where $x_{q_t+M(1-a_t)}$ is set to 1, and the other bits are set to 0. For example, with $M = 3$, for student success on problem 1, $q_t = 1, a_t = 1$, so $x_{1+3(1-1)} = 1$, so $\mathbf{x} = \{1, 0, 0, 0, 0, 0\}$, and failure on problem 1 $q_t = 1, a_t = 0$, so $x_{1+3(1-0)} = 1$, so x_4 is set to one, and $\mathbf{x}$ is $\{0, 0, 0, 1, 0, 0\}$.

Model Structure: The RNN version of DKT maps each input sequence $\mathcal{S}_T$ into an output sequence of predictions $\mathcal{Y} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_T\}$ with a set of hidden states $\mathbf{h}_1, \mathbf{h}_2, ...\mathbf{h}_T$. More specifically, as illustrated in Figure 1, this process is defined as:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}),$$

$$\mathbf{y}_t = \sigma(\mathbf{W}_{hy}\mathbf{h}_t).$$

In the equations, element-wise operators $\tanh(\cdot)$ and $\sigma(\cdot)$ are activation functions of the network, introducing non-linearity to the network. The parameters learned in the network are $\mathbf{W}_{xh}$ which transforms input $\mathbf{x}_t$ into the hidden space, $\mathbf{W}_{hh}$ which fuses the hidden state $\mathbf{h}_{t-1}$ from the prior input with the current hidden state $\mathbf{h}_t$, and $\mathbf{W}_{hy}$ which translates the hidden state h_t into an output. In both equations, the bias terms are omitted for simplicity, and the $\mathbf{h}_0$ is the initial hidden state, the zero-vector.

Model Output: The output sequence $\mathcal{Y}$ contains prediction vectors $\mathbf{y}_t$, sized M. Every element of the vector represents the probability of the student making a correct submission on corresponding problems in their next attempt. Note that while the model makes predictions for each problem at each timestep t, only the value for the next attempted problem q_{t+1} is used during training and evaluation.

3.2 Deep Code Knowledge Tracing

We extend DKT into Deep *Code* Knowledge Tracing (Code-DKT), by using the code2vec [5] representation of student code attempts, c_t, along with problem and correctness information.

Code Representation: Abstract syntax trees (ASTs) are used to represent the hierarchical structure of code, for example with a node for a function (`method`) with children representing the function's parameter (`input`) and body (`body`). AST leaf nodes often correspond to literal values or identifiers. Code-DKT extends the code2vec model for code classification, which encodes an AST using a set of *leaf-to-leaf paths* throughout the AST. For example, in Figure 2, a path from the leaf node `input` to the leaf node `"value"` (highlighted red in the example) consists of the nodes: [`input`, `method`, `body`, `String`, `"value"`]. Given an AST, code2vec extracts a set of leaf-to-leaf paths, as explained below.

Model Input: Since a deep learning model cannot operate directly on code paths, the Code-DKT must next convert this code-path representation of the AST into a binary vector. A student's code submission c_t at time t is represented as $\{p_0, p_1, ..., p_R\}$ where there are in total R randomly selected code paths in c_t. Every p_r has three components, namely the starting node of the code path $\mathbf{s}_r$, the textual representation of the full path $\mathbf{o}_r$, and the ending node $\mathbf{q}_r$, which are each one-hot encoded as binary vectors. For instance, for the example in Figure 2, $\mathbf{s}_r$ is `input`, $\mathbf{o}_r$ is a text string: `input|method|body|String|value`, and $\mathbf{q}_r$ is `value`.

Model Structure: Rather than using a *static* vector representation of students' code, Code-DKT *learns* an optimal embedding of student code. The detailed Code-DKT model structure is shown in Figure 3. This initial structure is drawn from code2vec. The nodes for each of R code paths in c_t (c_t has in total R paths), including starting and ending nodes $(\mathbf{s}_r, \mathbf{q}_r)$ and paths $\mathbf{o}_r$ for a single path r, are respectively embedded by the node embedding matrix $\mathbf{W}_{enode}$

[2] See the appendix of [34] for the LSTM DKT equations.

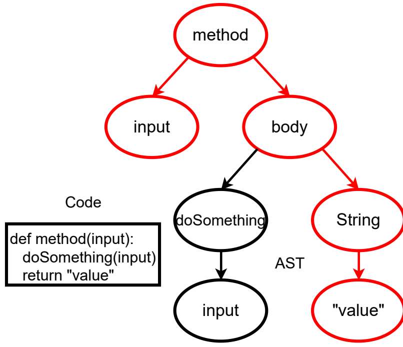

Figure 2: A simple AST where red nodes and edges represent a leaf-to-leaf path from `input` to `"value"`.

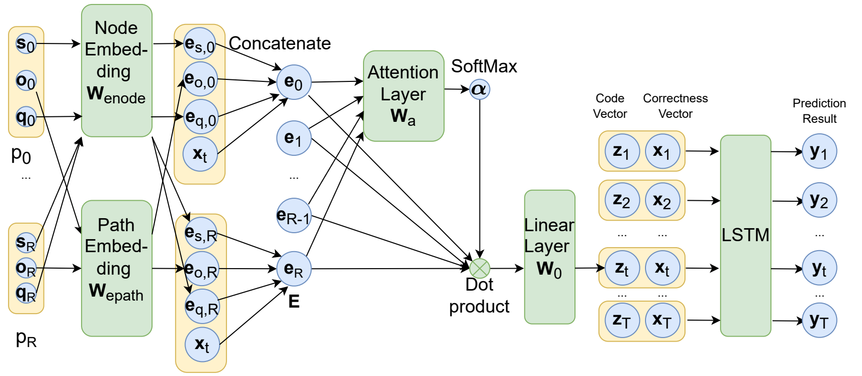

Figure 3: Code-DKT model structure.

and the path embedding matrix $\mathbf{W}_{epath}$. Both matrices are randomly initialized with a Gaussian distribution, but they are later updated during model training. The Code-DKT model structure then diverges somewhat from code2vec, to account for the specific needs of the KT problem. Specifically, the three embedded vectors representing c_t are concatenated with the problem-correctness vector x_t from DKT (introduced in Section 3.1). This serves as a numerical representation of (q_t, a_t, c_t). For a single code path p_r, this process is accomplished with embeddings for the start node ($\mathbf{e}_{s,r}$), path ($\mathbf{e}_{o,r}$), and end node ($\mathbf{e}_{q,r}$):

$$\mathbf{e}_{s,r} = \mathbf{W}_{enode}\mathbf{s}_r; \mathbf{e}_{o,r} = \mathbf{W}_{epath}\mathbf{o}_r; \mathbf{e}_{q,r} = \mathbf{W}_{enode}\mathbf{q}_r,$$

$$\mathbf{e}_r = [\mathbf{e}_{s,r}; \mathbf{e}_{o,r}; \mathbf{e}_{q,r}; \mathbf{x}_t].$$

Score-Attended Path Selection: Code-DKT now has an numerical representation of a single attempt: a set of R embedded vectors, $\mathbf{e}_r$, one for each code path in c_t. Note that the embedding, $\mathbf{e}_r$ not only includes the code information, but also the current correctness score information $\mathbf{x}_t$ at the submission t. However, not all parts of a student's code are relevant, and thus not all code paths $\mathbf{e}_r$ are important for predicting a student's future success. Therefore, the model uses an attention mechanism to identify how much weight to give to each of these paths. The embedding vectors $\mathbf{E} = \{\mathbf{e}_0, \mathbf{e}_1, ..., \mathbf{e}_R\}$ are multiplied by the attention matrix $\mathbf{W}_a$ to get R scalars $a_0, a_1, ..., a_R$, representing the importance (commonly known as the "attention") of each of the code paths. The importance a_r uses a SoftMax mechanism for normalization, having 1 as the sum. This process is formulated as:

$$\alpha = \text{SoftMax}(\mathbf{E}\mathbf{W}_a)$$

$$\text{SoftMax}(\mathbf{a}) = \frac{e^{a_i}}{\sum_{i=1}^{R} e^{a_i}}$$

where each elements α_r in $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_R\}$ are the calculated weights for the code path p_r. Finally, Code-DKT weights each code path $\mathbf{e_i}$ by its attention α_i, and sums them together, giving a weighted average: a single vector representing the important parts of the code. The weighted average vector is then multiplied by a matrix $\mathbf{W}_0$ to get the code vector $\mathbf{z}$, representing features extracted from code submissions, as in equation:

$$\mathbf{z} = \mathbf{W}_0(\sum_{i=1}^{R} \alpha_i \mathbf{e}_i).$$

In a sequence of T student attempts, Code-DKT produces T code vectors $\{\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_T\}$. The code vectors are concatenated with the correctness vectors $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_T\}$ as the input to the final LSTM (as in DKT), giving the predictions $\{\mathbf{y}_1, \mathbf{y}_2, ...\mathbf{y}_T\}$. Even though x_t was already used to produce $\mathbf{z}_t$, this final concatenation ensures the Code-DKT model has direct access to the student correctness score information.

4. EXPERIMENTS

We designed an experiment to evaluate 3 research questions about student modeling in the domain of programming:

RQ1 How effective are domain general KT approaches (DKT, BKT) on our programming dataset?
RQ2 How can features derived from students' code be used to improve KT models?
RQ3 When are these code features most useful, and how can they lead to improved predictions?

4.1 Dataset & Experiments Setup

Our study uses a dataset of an introductory Java programming class at a large, university in the US, collected in Spring 2019, stored in the ProgSnap2 format [36]. The dataset includes work from 410 students on 50 problems divided over 5 assignments. These were completed throughout the semester as homework, with each assignment focusing on a specific topic (e.g. conditionals, loops). For these problems, typical solutions ranged 10 to 20 lines of code. Students tended to make multiple submissions before succeeding finally, and 23.68% of the attempts were correct. Student code was automatically graded using test cases, and We treated a submission as correct (1) only when all test cases passed, and incorrect (0) otherwise.

For each assignment, students were then split into training and testing sets with a ratio of 4 : 1. One quarter of the training data were used for hyperparameter tuning and validation (see below). Then, we trained the model on the whole

Table 1: Performance Comparison on all assignments.

Model	A1	A2	A3	A4	A5
DKT	71.24%	73.09%	76.84%	69.16%	75.14%
Code-DKT	74.31%	76.56%	80.40%	72.75%	79.14%

Table 2: Overall and the first attempt performance of all models on assignment A1.

Models AUC (STD)	Overall	First Attempts
Code-DKT	**74.31%** (0.90%)	**75.74%** (0.69%)
DKT-TFIDF	69.94% (0.88%)	72.77% (0.79%)
DKT-Expert	69.52% (0.68%)	69.53% (0.72%)
DKT	71.24% (2.54%)	72.26% (3.69%)
BKT	63.78% (4.68%)	50.22% (2.86%)

training dataset, and tested on the holdout test dataset, repeating this process 10 times to account for model variation (e.g. due to random initialization). All deep learning models were implemented using the PyTorch[33] library, and our BKT implementation was pyBKT [7].

4.2 Hyperparameter Tuning & Optimization

For hyperparameter tuning, we split the training data into training and validation sets, and created a model with each possible set of hyperparameters (described below), and calculated AUC performance on the validation dataset. We repeated this process 100 times and chose the hyperparameter setting with the best average validation performance to use in testing/evaluation. Specifically, we selected the embedding size of code feature extraction as 300, from a range of (50, 100, 150, 300, and 350); learning rate was selected as 0.0005 from a range of (0.00005, 0.0005, 0.005, 0.01); the training epochs were set at 40 to save training time while keeping the best prediction results, selected from a range of (20, 40, 100). All other parameters were defaulted as the original settings of code2vec and DKT. We fixed the longest length of student attempts at 50 to filter extra long submission traces from students. In cases where more than 50 attempts were submitted, we used the last 50 submissions, assuming the latest submissions were more useful.

As the models were deep neural networks, we used binary cross entropy as a loss function to track the difference between the ground truth and predicted probabilities. The models used back propagation to update weight matrices (parameters), using the Adam optimizer [24], which is also a default for code2vec and DKT.[3]

4.3 Baselines

We compare the performance of Code-DKT to DKT, BKT, and two modified DKT methods: DKT-TFIDF adding data-driven features, and DKT-Expert adding expert features. Specifically, DKT-TFIDF uses TFIDF, a data-driven feature that counts the term frequency (TF) of tokens (variables, functions, and operations, etc.) in code text, and forms a frequency vector for every term. This frequency is multiplied by the inverse document frequency (IDF) to show how often terms show up in *unique* documents. As students use various variable names, we limited the top 50 best features (selected from a range of (30, 50, 100, 300) in hyperparameter tuning) in TFIDF to remove redundant features. For the DKT-Expert model, two authors examined the problems in the dataset, and determined 9 rule-based code features. These features include code component existence checks such as the usage of `else if` statements, the usage of `&&` operations, etc. These statements and operations represent students' usage of certain concepts such as writing alternative conditions, or using "and" logic to solve a problem.

To improve the TFIDF and Expert models to serve as more robust baseline models, we added one additional set of features (only to baseline models) to encode information about the skills practiced in each problem, as has been done in prior work [57]. Two authors examined the problem descriptions and solutions and agreed on 9 skills we expected students to learn. For example, one skill was solving problems with negative conditions in the instructions (using words such as "unless", "otherwise"), requiring students to negate these conditions in their code. We represented each problem as a binary vector of practiced skills, and we used this skill vector to represent problems, instead of the one-hot encoded problem ID (see Section 3.1 for model input encoding). Testing on the validation dataset showed slightly improved performance using these skill vectors.

Metric: Our primary performance metric is AUC, a standard evaluation metric for KT models [34, 44, 31], as it uses the predicted *probability* of success, rather than a binary correctness prediction, and is more appropriate than accuracy for imbalanced datasets like ours (23% positive).

5. RESULTS

5.1 Performance Comparison

5.1.1 Code-DKT vs DKT

Table 1 shows a comparison of DKT and Code-DKT across all 5 assignments (the average of the 10 test runs). Note that for each assignment, a new model is trained and tested separately, without using data from prior assignments. This was because assignments were spaced out with weeks between then, including additional learning content, so students' performance on prior assignments is less relevant. To address RQ1, we consider the overall performance of the baseline DKT model on our dataset, which has an AUC of 69-75% across assignments. This low score means it may be difficult to use model predictions to inform instruction or an automated intervention, as we discuss in Section 6. To address RQ2, we see that Code-DKT *consistently* outperforms DKT by 3-4% AUC on each assignment. This shows that our approach, which augments correctness features with additional information from student code, can improve DKT predictions. For perspective, this improvement is comparable to SAINT+'s improvement over DKT on EdNet (+2.76%) [13], or SAKT's improvement on various datasets (+3.8%) .

5.1.2 Code-DKT vs Naive Code Features and BKT

We now investigate a single assignment, A1, to illustrate Code-DKT's performance, and create a DKT-Expert baseline using assignment-specific, expert-authored code features. We selected assignment A1, as it came first (and was therefore not influenced by prior assignments) and its skills are

[3]Repository: https://github.com/YangAzure/Code-DKT

the least complex. Table 2 shows the performance of Code-DKT, DKT, as well as 3 new baselines: BKT, and 2 simple code-feature extensions of DKT: DKT-TFIDF and DKT-Expert (described in Section 4.3). Model performance is given for predicting all attempts (Overall) and for predicting only first attempts at each problem. The results show that neither the simple expert features nor the TFIDF data-driven features improve the overall performance of DKT. These simple features derived from student code instead negatively affect overall performance. This suggests that a more effective model structure is necessary for making use of code features, such as our Code-DKT model. We also see that BKT has an AUC score of only 63%, suggesting that deep models are more effective for our dataset.

5.1.3 When is Code-DKT Effective?

We used assignment A1 to investigate *when* Code-DKT was more effective than DKT, helping to answer RQ3.

Overall vs First Attempts: We investigated Code-DKT's performance at predicting a student's *first* attempt at each problem (Table 2, column 3). First attempts are important in a KT task because they represent points at which an ITS might make key interventions (e.g. offering a worked example if a student might fail at problem solving). Therefore, many KT evaluations differentiate a student's first attempt on a task (where a model must make predictions using only performance on *other* problems) from subsequent attempts. This distinction also helps us understand when the Code-DKT model is most effective. One might ask, is Code-DKT using student code submissions to learn a better representation of student knowledge (which *would* help it predict first attempts), or is it simply estimating how close a student is to solving the *current* problem (which would only help to predict *subsequent* attempts). Our results shows that Code-DKT actually performs *best* when predicting first attempts, and it also shows a similar improvement over DKT for first attempts (+3.48%), compared to all attempts (+2.93%) . This suggests that the content of a student's code is helpful for not only predicting how quickly they will solve the *current* problem, but also *future* problems.

Problem-specific Performance: Table 3 shows the decomposed AUC performance of Code-DKT and DKT on each problem. We observe that Code-DKT outperforms DKT overall on 6 of the 9 problems. The difference ranges from +15.54% AUC (problem 13) to -4.43% (problem 236), suggesting that the benefit of Code-DKT's code features depends somewhat on the programming problem. It also shows that code features *can* reduce model performance, but the potential for Code-DKT's improvement seems to be greater than the potential for harm.

To understand *when* Code-DKT's code features were useful, we investigated differences between the problems where it outperformed DKT and those where it did not. We found that many of the problems where there was improvement shared similar learning concepts and solution structure. For example, problems 3, 232 and 234 all used the "independent choice" programming pattern, which is often solved with nested if-statements. Similarly, problems 1, 3, 5 and 13 all included a pattern where one condition changes a value used in another condition. These common patterns seem to have

Table 3: Decomposed performance of Code-DKT and DKT AUC performance on different problems in assignment A1.

Problems	Code-DKT		DKT	
	Overall	First	Overall	First
234	**64.60%**	71.38%	63.75%	**73.48%**
13	**78.45%**	**86.55%**	63.59%	68.81%
232	**74.93%**	**78.99%**	72.49%	73.09%
233	64.79%	74.57%	**67.18%**	**76.33%**
5	**75.38%**	81.34%	74.28%	**81.79%**
235	70.65%	**71.96%**	**75.03%**	70.80%
236	74.25%	74.30%	**78.68%**	**77.06%**
1	**68.62%**	70.32%	66.67%	**73.20%**
3	**71.00%**	**71.00%**	64.02%	64.02%

helped the model make better predictions on problems that used them. However, 2 of the 3 of the problems where Code-DKT performed poorly involved a unique learning concept that did not appear in any other problems. For example, problem 236 requires students to check if any 2 of the 3 given variables are equal (which has no analog among other problems) and 233 requires the `Math.abs` function (which many students failed to use correctly). Together, these results suggest a hypothesis that Code-DKT's code features are most useful at predicting problems that share code structures with other problems, and less useful at predicting problems that emphasize novel code structures. This suggests Code-DKT may be successfully modeling students' knowledge of common code patterns.

5.1.4 Ablation Study

Our Code-DKT model design choices include: where to incorporate correctness information, how to update the embedding, and what underlying network to use (LSTM or RNN). Table 4 shows the results of an ablation study on assignment A1 to determine which of these choices improved the performance of our final DKT model (first row). The final Code-DKT model concatenates the *correctness* of a students' attempt with code features in two places (see Section 3.2): before the attention mechanism (the vector $\mathbf{e}_r$), and in the final trace fed into the LSTM ($\mathbf{z_i}$ concatenated with $\mathbf{x_i}$). The model in row 2 only includes correctness information in the first case, and row 3 includes it only in the second case. Both models lose performance, but not by much (0.5%), suggesting that correctness information helps both in attending to relevant code paths, and final predictions, but this information is somewhat redundant. We also investigated using an RNN (row 4) instead of an LSTM, but this was, as predicted, moderately less effective. Finally, recall that Code-DKT uses code2vec to embed students' code as a vector, and updates this embedding throughout model training. Row 5 shows a version where we pretrained this embedding on the training dataset, using code2vec to predict the correctness of students' code, and then fixed the embedding when training the LSTM. This model does much worse, suggesting that the relevant features for predicting the *correctness* of code are different from those for predicting *future performance*.

5.2 Case Studies

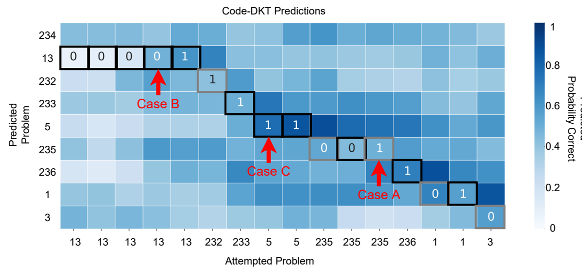

Figure 4: Code-DKT generated correctness predictions heatmap for a student.

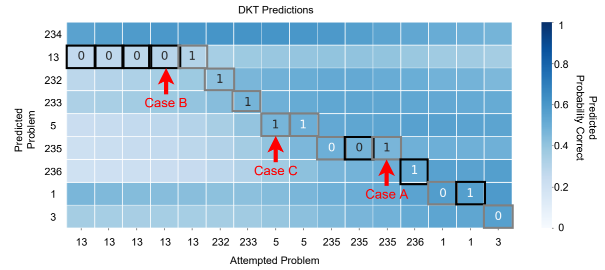

Figure 5: DKT generated correctness predictions heatmap for a student.

Table 4: Code-DKT ablation study on A1.

	Model	Overall AUC
1	Code-DKT (Final Model)	**74.31%**
2	Correctness: Attention Only	73.81%
3	Correctness: Trace Only	73.84%
4	Model: RNN	73.63%
5	Embedding: Static	68.74%

```
public int dateFashion(int you, int date)
{
    int value = 0;

    if (you >= 8 || date >= 8)
    {
        if (you <= 2 || date <=2)
        {
            value = 0;
        }
        else
        {
            return 2;
        }
    }
    else
    {
        value = 1;
    }
    return value;
}
```

c_t

```
public int dateFashion(int you, int date)
{
    int value = 0;

    if (you >= 8 || date >= 8)
    {
        if (you <= 2 || date <=2)
        {
            value = 0;
        }
        else
        {
            return 2;
        }
    }
    else
    {
        if (you <= 2 || date <=2)
        {
            value = 0;
        }
        else
        {
            value = 1;
        }
    }
    return value;
}
```

c_{t+1}

Figure 6: Code at times t and $t+1$ for Case A, where the code $c_1, ...c_t$ is used to predict correctness at $t+1$.

To further answer RQ3, we examined *how* code features may have improved Code-DKT through 3 case studies. We use prediction heatmaps from Code-DKT and DKT for one student, shown in Figures 4 and 5 for Code-DKT and DKT, respectively. The rectangular cells show which problem the student actually attempted (y-axis) at each time-step (x-axis), and the numbers in the cells represent the ground truth values of whether student's attempt was successful (1) or unsuccessful (0). Black frames indicate correct (i.e. accurate) model predictions, while grey ones indicate incorrect predictions. The color of the heatmap in each cell specifies the predicted probability of students making a correct submission on a given problem (y-axis) at the given time-step (x-axis), and darker means a higher probability of success. For example, in Figure 4, the student makes 4 unsuccessful attempts at problem 13, followed by a successful attempt, then succeeds at problems 232 and 233 in one attempt each.

The heatmaps for the student (Figures 4 and 5) show that Code-DKT is able to make better predictions on the traces than DKT, making 11 out of 16 successful predictions, while DKT is able to make 8 of them correct. Another observation is that Code-DKT heatmaps have much stronger predictions with values close to 1 or 0 compared with DKT, showing that with code features, the model is more confident.

Case A: Successful Prediction: In Case A, Code-DKT uses code features to make better predictions than DKT on the predictions of the student's final submission on Problem 235. As shown in Figures 4 and 5, while both Code-DKT and DKT can successfully predict the incorrect submission on the student's second submission of Problem 235 and fail to predict the correct submission on the third, Code-DKT gives a higher prediction than DKT. In Figure 6, the student's code submissions show the reason. The student's second submission is almost correct, demonstrating a correct (if inefficient) nested if-else structure, but they have omitted the nested condition in their else branch. Code-DKT is able to infer the quality of the student's code, since its prediction of success probability increased from 44.2% to 49.1% after the student's second (incorrect) attempt, while DKT's prediction decreased from 47.7% to 46.7%. Code-DKT's higher prediction may be because the if-else structure the student was missing was very similar to one they had already written, as shown in Figure 6. These code structures are easily captured by the path-based AST representation used by code2vec. Without code features, it is difficult for DKT to predict whether the student is going to succeed on t+1, since it only knows the student has failed twice, not how close they are to succeeding. Even with code features, there is still a great deal of uncertainty. No matter how close a student is to a correct answer, there is no guarantee they will achieve it on their next attempt. This may help to explain why Code-DKT does not more dramatically outperform DKT overall.

Case B: Unsuccessful Prediction: Case B shows that even when a student's code is nearly correct for a given problem, it doesn't guarantee that they will be successful on their next attempt. Sometimes Code-DKT is overconfident in these situations, and incorrectly predicts success, as in Case B. Figure 7 shows the last three attempts the student made on Problem 13: two incorrect followed by a final correct attempt. The only differences between the final attempt and the earlier two is shown in the red frames. The student's 4th attempt achieved the correct logic for Problem 13, the

Figure 7: Case B 4th, 5th and 6th code attempts.

Figure 8: Case C, using code from problems 232 and 233 to predict the same student's performance on problem 5

5th attempt adds an empty `return` statement, and the 6th and final attempt adds the appropriate return value. After seeing the almost-correct code at their 4th attempt, Code-DKT predicted that the student would succeed on the next attempt since the modifications they needed were minimal (just write "`return value;`"), but it took one extra attempt to get it right. An expert might make a similar conclusion, that the student was close enough to realize their mistake and submit a correct answer, and would have similarly been wrong. This highlights the uncertainty present in any KT task and the challenges of applying KT to student code.

Case C: Successful Prediction (First Attempt): Case C illustrates that Code-DKT can also use code from *previous problems* to improve its predictions of *first attempts* of new problems (as shown quantitatively in Table 2). For example, when the student successfully completes problems 232 and 233 in a single attempt, Code-DKT's prediction of the student's success on problem 5 increases from 42.0% to 50.0% to 72.5% respectively, leading it successfully predict success on the student's first attempt at problem 5. However, DKT's confidence only modestly increased from 41.8% to 47.3% to 47.0%, leading it to incorrectly predict failure. Both models know that these problems are related, and share some learning concepts (based on how other students' successes on the problems are related), but Code-DKT's analysis of the student code allowed it to infer more about the knowledge that was demonstrated in past problems.

We use these three consecutive code submissions to explain why this may be the case in Figure 8. For example, in Problem 232, the student directly uses Boolean variable in the if-condition (`if (vacation)`) rather than a superfluous comparison (`if (vacation == true)`) that many students use, demonstrating a higher level of understanding. This same direct usage of Boolean variables is seen in the `if` condition and `return` statement of Problem 5. The code submission on Problem 233 further suggests the student is able to combine logical operators with Boolean variables to `return` a Boolean expression. This occurs again in the `return` statement of the students' attempt at Problem 5, shown in the lower rectangular. While we cannot know for certain which code features Code-DKT used to make its success prediction for Problem 5, these repeated code structures are one possibility, given code2vec's ability to recognize repeated patterns in ASTs.

6. DISCUSSION

RQ1: How well do domain-general models perform? We used domain-general KT models (DKT) as the baseline models for our programming dataset. These models performed relatively poorly, averaging 73.09% AUC across assignments. While this is considerably better than chance, the performance may not be high enough to use in some student modeling contexts. For example, for assignment A1, the recall of DKT was 31.4% and the precision was 46.5%, so the model fails to identify two thirds of unsuccessful attempts, and over half of the time when the model predicts a failed attempt, the student actually succeeded. This suggests that KT is a difficult challenge on this dataset. By contrast, DKT has historically been effective on other datasets, which are both larger and in other domains, such as EdNet [13], Assistments [41] and KhanAcademy [34]. One possibility is that the more complex nature of programming problems, with myriad possible correct and incorrect solutions, makes KT prediction more challenging on this dataset, compared to those in other domains. If this is the case, several aspects of programming may contribute to the challenge of modeling student success. Programming problems often require many attempts to get correct (6.1 on average in our dataset), leading to class imbalance. In our dataset, the problem descriptions were complex, and their solutions involved complex conditional logic, and students had to write perfect Java syntax for the program to compile. These factors mean there are many ways for students to make small "slips", making the relationship between skill and success less direct.

Another possibility is that our dataset (410 students) was simply too small for complex deep models to find success, compared to the 1000s or even 100,000s of learners in other datasets where DKT has been evaluated. However, model complexity alone does not explain the difference, since the simpler BKT model did even worse than DKT, and our Code-DKT model, which had far more parameters, performed better. Additionally, DKT has historically performed well on some other small datasets (e.g. the "ASSIST-Chall" and "STATICS" datasets from [31] with 300-700 students). Regardless, many tutoring systems only have hundreds of students, and effective KT models must still be able to perform well on these small datasets. Thus, to the extent that

our datasets is representative of the domain, our results suggest the need for improved KT models for programming.

RQ2: How can code features improve KT models? Our results show that a simple extension of DKT with code features does not improve its performance. This result is somewhat surprising, given that relatively simple features (e.g. the presence of a `return` statement) should be at least somewhat related to how close a student is to a correct answer. It is possible such features may improve a model with different structure, but in our dataset, they were not helpful to DKT. This suggests the need for thoughtful approaches to incorporating domain-specific features into deep models. Our Code-DKT model was able to make reasonable improvements to DKT (+3.07% overall on A1). This is comparable to the improvement of SAINT over DKT on the EdNet dataset (they achieved +2.76% in AUC), or SAKT over DKT on various datasets (+3.8%) [31]. This suggests that domain-specific features can be just as important as model structure for effective KT. Code-DKT's improvement is also robust. It has a +3% to +4% improvement overall on all five assignments. Importantly, however this is still a relatively poor performance overall, suggesting the need for more work on leveraging domain-specific features for improved KT.

RQ3: When and how do code features work? We also explored when and how the code features improved model performance. We found that code features are most useful on problems that share similar learning concepts with other problems in the dataset, and less useful on problems with unique and difficult concepts (e.g. `Math.abs()`). This makes sense – if we make an analogy to the original BKT where each problem was labeled with KCs, if you had a unique KC, the model would have no way of predicting on that problem. In our case, the KCs are inferred by the model, but the same limitation exists. However, most problems in our dataset did share primary learning concepts (e.g. loops, conditionals) and benefit from code features, and this repeated practice is a common feature of many CS1 courses. We also found that code features are useful for predicting both first attempts and subsequent attempts. Our case studies reveal potential mechanisms for both of these effects. For repeated attempts, the model seems to use the relative correctness of a student's code to determine how close they are to a solution and therefore how likely they are to get it right on the next attempt. For first attempts, the model seems to identify code structures in prior attempts that indicate knowledge or competence with certain programming concepts, which it uses to make predictions on new problems. More work is needed to verify these hypotheses, and to understand how the model represents this knowledge.

Limitations: Our model and experiment have several limitations. 1) All models evaluated, including Code-DKT, have a relatively low performance, partially due to the difficulty of the problem and low data size (410 students), as discussed in Section 6. Still, they perform considerably better than chance, and such models could still be useful, e.g. in prioritizing help to struggling students. 2) Our dataset was from a single semester of a course. While our semester-long dataset of 50 problems is considerably more robust than some of the prior work on KT in programming (e.g. using 1-2 problems [52]), it is unclear how our results will generalize to other semesters, classes or programming languages. 3) We used only DKT as a baseline model to extend and to compare against, and it is possible code features may have different effects on other models. However, as explained in Section 3.1, DKT has a comparable performance to more modern deep models, and made sense as a starting point to explore the effect of code features.

7. CONCLUSION

The contributions of the paper are 1) the Code-DKT model, which extends DKT with embedded code feature extraction; 2) results showing that CodeDKT consistently improves over DKT in a programming dataset; and 3) comparisons and case studies highlighting when and why Code-DKT code features help. This paper compared our new Code-DKT model to domain-general BKT and DKT baselines, and two DKT models extended with simple code features, demonstrating improved performance for Code-DKT over these baselines. However, the best baseline model performance was about 73%, and Code-DKT was 74.3%, demonstrating considerable room for improvement on modeling for knowledge tracing in programming. The case studies in this paper illustrate specific situations where knowledge tracing can be particularly difficult in programming, and where there is potential for improving code KT, e.g. when common code structures are used across problems.

Acknowledgements: This material is based upon work supported by NSF under Grant No. #2013502.

8. REFERENCES

[1] F. Ai, Y. Chen, Y. Guo, Y. Zhao, Z. Wang, G. Fu, and G. Wang. Concept-aware deep knowledge tracing and exercise recommendation in an online learning system. 2019.

[2] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100. PMLR, 2016.

[3] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013.

[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 404–419, 2018.

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

[6] J. R. Anderson and B. J. Reiser. The lisp tutor. *Byte*, 10(4):159–175, 1985.

[7] A. Badrinath, F. Wang, and Z. Pardos. pybkt: An accessible python library of bayesian knowledge tracing models. In *In: Proceedings of the 14th International Conference on Educational Data Mining (EDM 2021)*. ERIC, 2021.

[8] R. S. Baker, A. T. Corbett, and V. Aleven. More accurate student modeling through contextual

estimation of slip and guess probabilities in bayesian knowledge tracing. In *International conference on Intelligent Tutoring Systems*, pages 406–415. Springer, 2008.

[9] T. Barnes. The q-matrix method: Mining student response data for knowledge. In *American Association for Artificial Intelligence 2005 Educational Data Mining Workshop*, pages 1–8. AAAI Press, Pittsburgh, PA, USA, 2005.

[10] P. Chen, Y. Lu, V. W. Zheng, and Y. Pian. Prerequisite-driven deep knowledge tracing. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 39–48. IEEE, 2018.

[11] Y. Chen, Q. Liu, Z. Huang, L. Wu, E. Chen, R. Wu, Y. Su, and G. Hu. Tracking knowledge proficiency of students with educational priors. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 989–998, 2017.

[12] Y. Choi, Y. Lee, J. Cho, J. Baek, B. Kim, Y. Cha, D. Shin, C. Bae, and J. Heo. Towards an appropriate query, key, and value computation for knowledge tracing. In *Proceedings of the Seventh ACM Conference on Learning @ Scale*, pages 341–344, 2020.

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

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

[15] A. T. Corbett and A. Bhatnagar. Student modeling in the act programming tutor: Adjusting a procedural learning model with declarative knowledge. In *User Modeling*, pages 243–254. Springer, 1997.

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[17] T. Gervet, K. Koedinger, J. Schneider, T. Mitchell, et al. When is deep learning the best approach to knowledge tracing? *Journal of Educational Data Mining*, 12(3):31–54, 2020.

[18] A. Ghosh, N. Heffernan, and A. S. Lan. Context-aware attentive knowledge tracing. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2330–2339, 2020.

[19] R. Gupta, A. Kanade, and S. Shevade. Neural attribution for semantic bug-localization in student programs. *Advances in Neural Information Processing Systems*, 32, 2019.

[20] F. Hermans and E. Aivaloglou. Do code smells hamper novice programming? a controlled experiment on scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[21] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[22] B. P. Iii, A. Hicks, and T. Barnes. Generating hints for programming problems using intermediate output. In *In: Proceedings of the 7th International Conference on Educational Data Mining (EDM 2014)*, 2014.

[23] W. Jin, L. Lehmann, M. Johnson, M. Eagle, B. Mostafavi, T. Barnes, and J. Stamper. Towards automatic hint generation for a data-driven novice programming tutor. In *Workshop on Knowledge Discovery in Educational Data, 17th ACM Conference on Knowledge Discovery and Data Mining*, 2011.

[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[25] K. Leelawong and G. Biswas. Designing learning by teaching agents: The betty's brain system. *International Journal of Artificial Intelligence in Education*, 18(3):181–208, 2008.

[26] Q. Liu, Z. Huang, Y. Yin, E. Chen, H. Xiong, Y. Su, and G. Hu. Ekt: Exercise-aware knowledge tracing for student performance prediction. *IEEE Transactions on Knowledge and Data Engineering*, 33(1):100–115, 2019.

[27] M. Maniktala, C. Cody, A. Isvik, N. Lytle, M. Chi, and T. Barnes. Extending the hint factory for the assistance dilemma: A novel, data-driven helpneed predictor for proactive problem-solving help. *Journal of Educational Data Mining*, 12(4):24–65, Dec 2020.

[28] Y. Mao, Y. Shi, S. Marwan, T. W. Price, T. Barnes, and M. Chi. Knowing" when" and" where": Temporal-astnn for student learning progression in novice programming tasks. In *In: Proceedings of the 14th International Conference on Educational Data Mining (EDM 2021)*, 2021.

[29] Y. Mao, R. Zhi, F. Khoshnevisan, T. W. Price, T. Barnes, and M. Chi. One minute is enough: Early prediction of student success and event-level difficulty during a novice programming task. In *In: Proceedings of the 12th International Conference on Educational Data Mining (EDM 2019)*, 2019.

[30] B. Mostafavi and T. Barnes. Evolution of an intelligent deductive logic tutor using data-driven elements. *International Journal of Artificial Intelligence in Education*, 27(1):5–36, 2017.

[31] S. Pandey and G. Karypis. A self-attentive model for knowledge tracing. In *In Proceedings of the 12th International Conference on Educational Data Mining (EDM) 2019*, 2019.

[32] Z. A. Pardos and N. T. Heffernan. Modeling individualization in a bayesian networks implementation of knowledge tracing. In *International Conference on User Modeling, Adaptation, and Personalization*, pages 255–266. Springer, 2010.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[34] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein. Deep knowledge tracing. *Advances in Neural Information Processing Systems*, 28, 2015.

[35] T. W. Price, Y. Dong, and D. Lipovac. isnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 483–488, 2017.

[36] T. W. Price, D. Hovemeyer, K. Rivers, G. Gao, A. C. Bart, A. M. Kazerouni, B. A. Becker, A. Petersen, L. Gusukuma, S. H. Edwards, et al. Progsnap2: A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 356–362, 2020.

[37] G. Rasch. *Probabilistic models for some intelligence and attainment tests.* ERIC, 1993.

[38] V. Raychev, P. Bielik, and M. Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 731–747, 2016.

[39] K. Rivers, E. Harpstead, and K. Koedinger. Learning curve analysis for programming: Which concepts do students struggle with? In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 143–151, 2016.

[40] E. Rowe, J. Asbell-Clarke, R. S. Baker, M. Eagle, A. G. Hicks, T. M. Barnes, R. A. Brown, and T. Edwards. Assessing implicit science learning in digital games. *Computers in Human Behavior*, 76:617–630, 2017.

[41] D. Selent, T. Patikorn, and N. Heffernan. Assistments dataset from multiple randomized controlled experiments. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*, pages 181–184, 2016.

[42] Y. Shi, Y. Mao, T. Barnes, M. Chi, and T. W. Price. More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code. In *In Proceedings of the 14th International Conference on Educational Data Mining (EDM) 2021*, 2021.

[43] Y. Shi, K. Shah, W. Wang, S. Marwan, P. Penmetsa, and T. Price. Toward semi-automatic misconception discovery using code embeddings. In *LAK21: 11th International Learning Analytics and Knowledge Conference*, pages 606–612, 2021.

[44] D. Shin, Y. Shim, H. Yu, S. Lee, B. Kim, and Y. Choi. Saint+: Integrating temporal features for ednet correctness prediction. In *LAK21: 11th International Learning Analytics and Knowledge Conference*, pages 490–496, 2021.

[45] Y. Su, Q. Liu, Q. Liu, Z. Huang, Y. Yin, E. Chen, C. Ding, S. Wei, and G. Hu. Exercise-enhanced sequential modeling for student performance prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[46] V. Swamy, A. Guo, S. Lau, W. Wu, M. Wu, Z. Pardos, and D. Culler. Deep knowledge tracing for free-form student code progression. In *International Conference on Artificial Intelligence in Education*, pages 348–352. Springer, 2018.

[47] M. L. Swartz and M. Yazdani. *Intelligent tutoring systems for foreign language learning: The bridge to international communication*, volume 80. Springer Science & Business Media, 2012.

[48] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pages 317–325. Citeseer, 2004.

[49] K. VanLehn. Student modeling. *Foundations of Intelligent Tutoring Systems*, 55:78, 1988.

[50] K. VanLehn. The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 16(3):227–265, 2006.

[51] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[52] L. Wang, A. Sy, L. Liu, and C. Piech. Learning to represent student knowledge on programming exercises using deep learning. In *In: Proceedings of the 10th International Conference on Educational Data Mining (EDM 2017)*, 2017.

[53] S. Wang, A. Ororbia, Z. Wu, K. Williams, C. Liang, B. Pursel, and C. L. Giles. Using prerequisites to extract concept maps from textbooks. In *Proceedings of the 25th ACM international conference on Information and Knowledge Management*, pages 317–326, 2016.

[54] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057. PMLR, 2015.

[55] M. Yudelson, R. Hosseini, A. Vihavainen, and P. Brusilovsky. Investigating automated student modeling in a java mooc. In *In Proceedings of the 7th International Conference on Educational Data Mining (EDM) 2014*, 2014.

[56] M. V. Yudelson, K. R. Koedinger, and G. J. Gordon. Individualized bayesian knowledge tracing models. In *International Conference on Artificial Intelligence in Education*, pages 171–180. Springer, 2013.

[57] L. Zhang, X. Xiong, S. Zhao, A. Botelho, and N. T. Heffernan. Incorporating rich features into deep knowledge tracing. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, pages 169–172, 2017.

[58] G. Zhou, H. Azizsoltani, M. S. Ausin, T. Barnes, and M. Chi. Hierarchical reinforcement learning for pedagogical policy induction. In *International conference on Artificial Intelligence in Education*, pages 544–556. Springer, 2019.