Knowing *When* and *Where*: Temporal-ASTNN for Student Learning Progression in Novice Programming Tasks

Ye Mao
North Carolina State Univ.
Raleigh, NC, USA
ymao4@ncsu.edu

Yang Shi
North Carolina State Univ.
Raleigh, NC, USA
yshi26@ncsu.edu

Samiha Marwan
North Carolina State Univ.
Raleigh, NC, USA
amarwan@ncsu.edu

Thomas W. Price
North Carolina State Univ.
Raleigh, NC, USA
twprice@ncsu.edu

Tiffany Barnes
North Carolina State Univ.
Raleigh, NC, USA
tmbarnes@ncsu.edu

Min Chi
North Carolina State Univ.
Raleigh, NC, USA
mchi@ncsu.edu

ABSTRACT

As students learn how to program, both their programming code and their understanding of it evolves over time. In this work, we present a general data-driven approach, named *Temporal-ASTNN* for modeling student learning progression in open-ended programming domains. Temporal-ASTNN combines a novel neural network model based on abstract syntactic trees (AST), named ASTNN, and Long-Short Term Memory (LSTM) model. ASTNN handles the *linguistic* nature of student programming code, while LSTM handles the *temporal* nature of student learning progression. The effectiveness of ASTNN is first compared against other models including a state-of-the-art algorithm, Code2Vec across two programming domains: iSnap and Java on the task of program classification (*correct* or *incorrect*). Then the proposed temporal-ASTNN is compared against the original ASTNN and other temporal models on a challenging task of student success early prediction. Our results show that Temporal-ASTNN can achieve the best performance with only the first 4-minute temporal data and it continues to outperform all other models with longer trajectories.

Keywords

Student Modeling in Programming, LSTM, ASTNN

1. INTRODUCTION

Learning how to program is like learning how to write in a second language. As students learn to author code, both their programming code and their understanding of it evolves over time. Prior research has either focused *exclusively* on developing accurate *linguistic* models of their artifacts [30, 24, 1, 42], or developing *temporal* models of students comprehension of programming [11, 21, 23]. In this work, we propose a general data-driven approach named *Temporal-ASTNN*, which combines a state-of-the-art neural network model based on abstract syntax trees (AST) named ASTNN – addressing the *linguistic structure* of the students' artifacts – along with Long-Short Term Memory (LSTM), which handles their *learning progression*. In this way we effectively marry *both* aspects of the process in a single system.

Ye Mao, Yang Shi, Samiha Marwan, Thomas Price, Tiffany Barnes and Min Chi "Knowing When and Where: Temporal-ASTNN for Student Learning Progression in Novice Programming Tasks". 2021. In: Proceedings of The 14th International Conference on Educational Data Mining (EDM21). International Educational Data Mining Society, 172-182. https://educationaldatamining.org/edm2021/
EDM '21 June 29 - July 02 2021, Paris, France

Much as *language* is how people communicate, *programming languages* are how we communicate with machines, and various natural language processing (NLP) techniques can be applied to modeling programming languages [15]. Traditional approaches for code representation often treat code fragments as natural language texts and model them based on their tokens [7, 9]. Despite their simplicity, token-based methods omit the rich and explicit *structural* information [25] in student codes. Until recently, deep learning models have achieved state-of-the-art results on source code analysis, including code functionality classification [24], method name prediction [1], code clone detection [42] and so on. These successful models usually combine Abstract Syntax Tree (AST) representations with various neural networks to capture the structural information from the programming language. Their impressive performance shows that by addressing the *linguistic structural* nature of code, syntactic knowledge is indeed important to learn meaningful code representation.

On the other hand, modeling student learning progression in open-ended programming environments is also a type of student modeling. Generally speaking, student modeling has been widely applied to predict the student's future performance based on historical data. For well-defined learning environments, student models usually monitor students' learning progress (*correct* or *incorrect*) over time to infer their knowledge states, such as Bayesian Knowledge Tracing (BKT) [8] and Deep Knowledge Tracing (DKT) [29]. When it comes to open-ended programming environments, student modeling becomes much more challenging because 1) the correctness evaluation concerning each step taken by students will not be available, and 2) it is extremely hard to represent student states. As a result, prior research either has focused on utilizing other features such as hint usage, interface interactions to evaluate student learning outcomes [11], or creating meaningful states by transforming student click-like log files into fixed feature sets for various student

للاستشارات

www.manaraa.com

modeling tasks [21]. While such prior work is able to capture the *temporal* information from historical data, it ignores the *linguistic, structural* property of student code. As an accurate student model is a building block for any educational system that provides adaptivity and personalization, it is especially important to model student learning progression in open-ended programming tasks by addressing both *linguistic* and *temporal* characteristics in student code sequences.

In this work, we present a data-driven approach named *Temporal-ASTNN* to model student learning progression in open-ended programming domains. Temporal-ASTNN consists of two main modules: 1) ASTNN [42] for code representation learning, which can handle the *linguistic* structure of student code, and 2) LSTM [16] for temporal learning, which handles the *temporal* nature of student learning progression. In order to explore the effectiveness of our model, we focus on two types of student modeling tasks. One is *the task of program classification (correct or incorrect)*, in which the effectiveness of ASTNN is compared against other models including a state-of-the-art algorithm, Code2Vec [1] across two programming domains: an open-ended block-based programming environment named iSnap and a textual programming environment for the Java programming language. The other is the task of *student success early prediction* in which the effectiveness of temporal-ASTNN is compared against the original ASTNN and other models integrating with different feature embeddings on iSnap only because it has trajectories of student codes.

Our main contributions are: 1) To the best of our knowledge, Temporal-ASTNN is the first model to address both *linguistic* and *temporal* properties of student learning progression in programming tasks; 2) We explored the robustness and the effectiveness of our model on student success early prediction task and compared it with state-of-the-art temporal models; and 3) We evaluated the effectiveness of ASTNN against Code2Vec and various baseline models on student program classification tasks across two domains, while most prior research mainly focused on classic tasks of *professional* source code analysis instead of novice programming.

The remainder of this paper is structured as follows. Section 2 presents the methods. Section 3 and 4 describe the two types of programming tasks together with experimental settings and results. Section 5 presents the related work. Finally, we discuss and conclude our work in Section 6.

2. METHODS

Problem Definition: For the task of student program classification, our dataset can be represented $\langle \boldsymbol{X}, \boldsymbol{Y} \rangle = \{\langle \boldsymbol{x}^1, y^1 \rangle, \langle \boldsymbol{x}^2, y^2 \rangle, ..., \langle \boldsymbol{x}^N, y^N \rangle\}$ where N is the total number of codes in the dataset where $\boldsymbol{x}^i$ represents a code snippet of student i and binary y^i indicates whether the code is correct or not.

For the task of student success early prediction, our dataset can be represented as $\boldsymbol{X} = \{\boldsymbol{x}^1, \boldsymbol{x}^2, ..., \boldsymbol{x}^M\}$, where M is the number of students. For a given student k, $\boldsymbol{x}^k = \{\boldsymbol{x}^k_1, ..., \boldsymbol{x}^k_{T_k}\}$, where $\boldsymbol{x}^k_t$ represents student k's code at time step t in $\boldsymbol{x}^k$ and T_k is the total number of codes in the student k's learning trajectories which varies with different students. For each $\boldsymbol{x}^k$, we are provided with the outcome label y^k for the outcome of the sequence of codes. $y^k = 0$ indicates the student

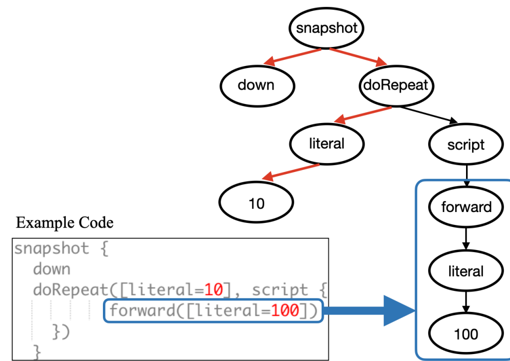


Figure 1: An examaple of iSnap code and the AST representing its syntactic structure. Red highlights a sample path, and blue highlights a sample ST-tree.

k succeeded, otherwise $y^k = 1$. The goal of student success early prediction is to predict the y^k using the student's codes from the beginning up to the certain minutes: $\boldsymbol{x}^k_1, \boldsymbol{x}^k_2, ..., \boldsymbol{x}^k_t$. For simplicity, we omit index k hereinafter when it does not cause ambiguity.

2.1 Temporal-ASTNN

Figure 2 shows the detailed structure of Temporal-ASTNN. Fundamentally, it contains a ASTNN which learns the embedding for student code and a LSTM layer which handles the temporal aspect. It is important to note that in Temporal-ASTNN, the two modules interact with each other to control how information flows.

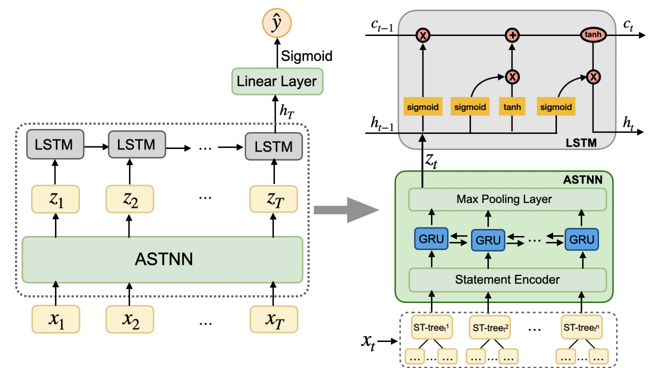


Figure 2: Temporal-ASTNN model structure: the output of ASTNN connects to the input of LSTM.

2.1.1 *ASTNN*

ASTNN is one of the state-of-the-art methods in source code analysis, and it's main idea is to learn a vector for the code through statement-level ASTs. Specifically, we split the large AST of a code fragment by the granularity of statement and extract a sequence of statement trees (ST-trees) via pre-order traversal. As shown in Figure 1 (highlighted in blue), we can get a ST-tree rooted at `forward`, whose child is `literal` and grandchild is `100`. In this way, we will get a sequence of ST-trees from the original AST, and feed them as the raw input of ASTNN. As shown in Figure 2, ST-trees

www.manaraa.com
للاستشارات

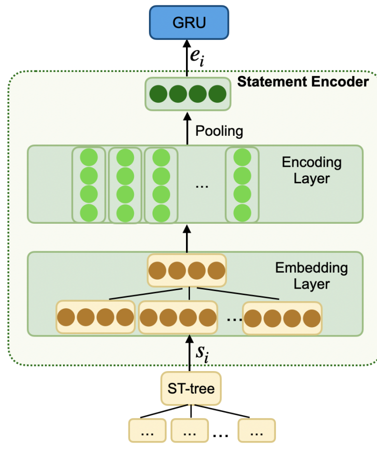


Figure 3: Statement Encoder of ASTNN, composing by an Embedding layer, an Encoding layer and a max-pooling layer.

will first pass Statement Encoder, then go through Bidirectional GRU (Bi-GRU) [2], finally pass max-pooling layer to get the vector for code representation.

Statement Encoder: Figure 3 shows the detailed structure of statement encoder. Assuming that there are J total nodes in a ST-tree s_i, for each node $n_i^j \in s_i$, $j \in [1, J]$, it will first go through the embedding layer to get initial embeddings $v_i^j = \mathbf{W_{embed}}^\top n_i^j$, where $\mathbf{W_{embed}} \in \mathbb{R}^{V \times d}$ is the pre-trained embedding matrix, V is the vocabulary size and d is the embedding dimension. Then the vector will be updated through a Recursive Neural Network [35] based encoder layer: $h_i^j = \sigma(\mathbf{W_{encode}}^\top v_i^j + \sum h_{\text{child}} + b_i^j)$. Here $\mathbf{W_{encode}} \in \mathbb{R}^{d \times k}$ is the encoding matrix and k is the encoding dimension. b is the biased term and σ is the activation function, in this work we followed the original paper to set σ as identify function. After recursive optimization of the vectors of all node in the ST-tree, we sample the final representation e_i via a max-pooing layer:

$$e_i = [max(h_i^{j_1}), max(h_i^{j_2}), ..., max(h_i^{j_k})], j \in [1, J] \quad (1)$$

Code Representation: For a set of ST-trees $(s_1, s_2, ..., s_L)$, where L is the number of ST-trees in the AST, our goal is to get a code vector z as final representation. After generating a sequence of vectors $(e_1, e_2, ..., e_L)$ from Statement Encoder, we will apply Bi-GRU to track the naturalness of statements sequence:

$$h_i = [\overrightarrow{GRU}(e_i), \overleftarrow{GRU}(e_i)], i \in [1, L] \quad (2)$$

The statement representation $h_i \in \mathbb{R}^{L \times 2m}$, where m is the embedding dim of Bi-GRU. Finally, similar to Statement Encoder, a max-pooling layer is used to sample the most important features on each of the embedding dimension. Thus we get $z \in \mathbb{R}^{2m}$, which is treated as the final vector representation of the original code fragment.

In original ASTNN, we can add another linear layer to directly fit z to the following prediction tasks. While in Temporal-ASTNN, z will be used as the input for LSTM memory cell.

2.1.2 LSTM

As shown in Figure 2, at each time step t, the output of ASTNN z_t will be used as the input for LSTM cell. Once ASTNN generates the code representation by learning the *linguistic* nature from code x_t:

$$z_t = \text{ASTNN}(x_t) \quad (3)$$

LSTM is trained utilizing input vector z_t to handle the temporal information. There are three major components: a forget gate, an input gate, and an output gate in a LSTM memory cell.

Forget Gate: In the first step, a function of the previous hidden state h_{t-1} and the new code input z_t passes through the forget gate, indicating what is probably irrelevant and can be taken out of the cell state. The forget component will calculate a weight f_t between 0 to 1 for each element in hidden state vector C_{t-1}. Here W_f and b_f are the weights and bias for the forget component.

$$f_t = \text{sigmoid}(\mathbf{W_f} \cdot [h_{t-1}, z_t] + b_f) \quad (4)$$

Input Gate: There are two steps involved in input component's calculation. In the first step, a *tanh* layer calculates a candidate vector $\tilde{C}_t$ that could be added to the current hidden state. In the second step, the input components calculate a weight vector i_t (ranging from 0 to 1) to determine to what extent $\tilde{C}_t$ should update the current memory state.

$$\begin{aligned} \tilde{C}_t &= \tanh(\mathbf{W_c} \cdot [h_{t-1}, z_t] + b_c) \\ i_t &= \text{sigmoid}(\mathbf{W_i} \cdot [h_{t-1}, z_t] + b_i) \end{aligned} \quad (5)$$

Output Gate: The output component is simply an activation function that filters elements in memory cell state C_t, where $C_t = C_{t-1} \cdot f_t + \tilde{C}_t \cdot i_t$. It calculates a weight vector to determine how much information is allowed to be revealed:

$$o_t = \text{sigmoid}(\mathbf{W_o} \cdot [h_{t-1}, z_t] + b_o) \quad (6)$$

Finally we get the output of time t: $h_t = o_t * \tanh(C_t)$. In this work, we used the last-step output from LSTM as the temporal representation of student code sequence.

2.1.3 Temporal-ASTNN: Truncated vs. Entire

As shown in Figure 2, by combining ASTNN and LSTM, the final Temporal-ASTNN can be described as:

$$\begin{aligned} z_1, ..., z_T &= \text{ASTNN}(x_1, ..., x_T) \\ h_T &= \text{LSTM}(z_1, ..., z_T) \\ \hat{y} &= \text{sigmoid}(\mathbf{W_l} h_T + b_l) \end{aligned} \quad (7)$$

where $\hat{y}$ is the output from Temporal-ASTNN, $\mathbf{W_l}$ is the weight matrix b_l is the bias term for the liner layer. The entire Temporal-ASTNN framework is learned by optimizing ASTNN and LSTM parameters spontaneously. They are optimized by minimizing the binary cross-entropy:

$$\mathcal{L}(\hat{y}, y; \Theta) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (8)$$

Prior research on applying ASTNN for source code analysis only used one snippet of code fragment to extract meaningful representation for following machine learning tasks. However, when combining ASTNN with LSTM on student

المنارة للاستشارات
www.manaraa.com

programming sequences such as iSnap for early prediction, we have the choices of either using the truncated training sequences or using the entire sequences. The advantage of using truncated sequences is that the training data would be more similar to the testing data and thus, the learned representations are more likely to emerge and be representative for the early success task. On the other hand, the advantage of using *entire* sequences is that the longer the sequences, the more meaningful AST patterns can be considered and discovered. Thus, we explored Temporal-ASTNN using both the *entire* and the *truncated* sequences for representation learning and referred as Temporal-ASTNN$_{\text{Trunc}}$ and Temporal-ASTNN$_{\text{Entire}}$ respectively.

2.2 Code2Vec

Code2Vec [1] leverages different features and model structures, and focuses on the dependency of distant components in code structures to achieve code classification tasks. As with ASTNN, Code2Vec is designed to address the *linguistic* structure of programming languages. Fundamentally, there are two main differences between these two models: 1) ASTNN takes a set of statement-level ASTs as inputs, while Code2Vec utilizes the syntactic paths of ASTs to learn the representation (an example path is shown in Figure 1 in red). And 2) After encoding the vector representations of ST-trees, ASTNN uses Bi-GRU to handle the sequence of vectors; while Code2Vec utilizes an attention mechanism to learn a weighted average of path vectors and thus to produce the final code representation. With the vector representing code, Code2Vec can also be used for various prediction tasks.

3. STUDENT PROGRAM CLASSIFICATION

In the task of student program classification, we aimed to predict the correctness (correct or incorrect) of student submitted code. The effectiveness of ASTNN is compared against Code2Vec and other token-based models across two programming domains: iSnap and Java.

3.1 Datasets

3.1.1 iSnap

iSnap is an extension to Snap*!* [13], a block-based programming environment, used in an introductory computing course for non-majors in a public university in the United States [32]. iSnap extends Snap*!* by providing students with data-driven hints derived from historical correct student solutions [31]. In addition, iSnap logs all students actions while programming (e.g. adding or deleting a block), as a *trace*, allowing us to detect the sequences of all student steps, as well as the *time* taken for each step. In this work, we focused on one homework exercise named Squiral, derived from the BJC curriculum [13]. In Squiral, students are asked to write a procedure that draws a square-like spiral. As shown in Figure 4, correct solutions require procedures, loops, and variables using at least 7 lines of code. We collected students' data for Squiral from Spring 2016, Fall 2016, Spring 2017, and Fall 2017. We excluded students who requested hints from iSnap to eliminate factors that might affect students' problem-solving progress, leaving a total of 65, 38, 29, and 39 student code traces from each semester, respectively.

The data collected from iSnap consists of a code trace for each student's attempt. This code trace represents a sequence of timestamped snapshots of student code. In prior research, an expert feature detector has been proposed to automatically detect 7 expert features of a student snapshot [43]. Those expert features are binary and indicate whether the corresponding feature presents or not. We ran the expert-feature detector to tag each snapshot in all 171 code traces, making a total of 31,064 tagged snapshots. With the temporal sequences, iSnap data is evaluated not only on this classification task, but also on the temporal early prediction task as described in Section 4.

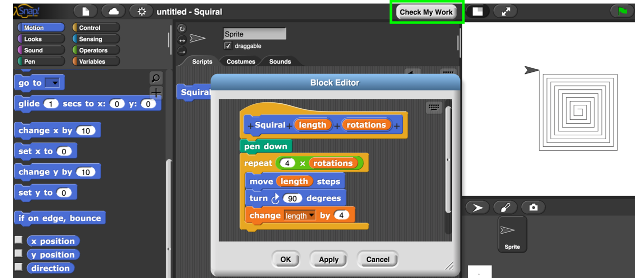


Figure 4: The iSnap interface, with the blocks palette on the left, the output stage on the right, the scripting area in the middle, and the hints button on top.

3.1.2 CodeWorkout

CodeWorkout[1] is an online and open system for programming in Java. It provides a web-based platform on which students from various backgrounds can practice programming and instructors can offer courses [10]. Different from iSnap, CodeWorkout *doesn't* log students' traces during programming but only their submissions. In this work, we focused on one programming exercise named isEverywhere, where the knowledge of loops and array will be mainly evaluated. In isEverywhere, students are asked to write a Java function to check if a value is "everywhere", that is in the given array if the value exists for every pair of adjacent elements. As shown in Figure 5, the system will show detailed feedback regarding the student's submission, indicating how it failed/succeed on the corresponding test cases.

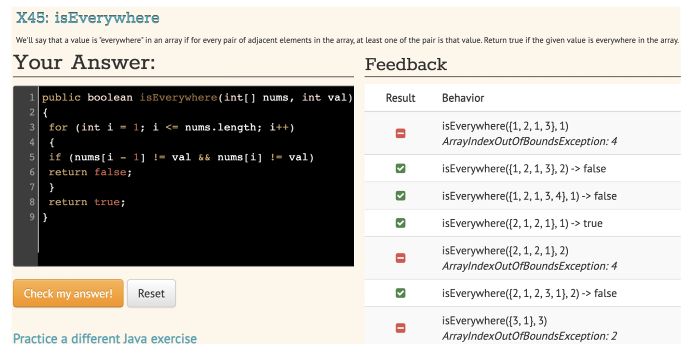


Figure 5: The CodeWorkout interface, with the problem description on the top, the coding area in the middle, and the feedback on the right.

The data collected from CodeWorkout is in Progsnap2 [33] format, and consists of two semesters: Spring 2019 and Fall

[1] https://codeworkout.cs.vt.edu/

المنارة للاستشارات
www.manaraa.com

2019. Similar to iSnap, we processed the data to eliminate factors that might affect students' problem-solving progress, and only kept the first compliable program from each student. In total, we have 448 and 307 student submissions from each semester, respectively. Please note that in CodeWorkout, only submissions from students are recorded and sequences of student edits are not available, thus it is only evaluated on the task of student program classification.

3.2 Task Description

For the task of student program classification, the ground truth labels are generated as follows: in iSnap, a student's submission is correct only if it satisfies all rubrics requirements, which are based on the expert-designed features and verified by humans; in CodeWorkout, a submission is correct if it passes all testing cases. Table 1 shows the number of correct and incorrect submissions across the semesters in each dataset. Note that here we include one submission per student to ensure that all data points are independent in both datasets. More specifically, for each student, we include the student's "own" submission before receiving any detailed feedback, which means the student's final submission for iSnap and the first submission for CodeWorkout.

Table 1: Data overview on Student Program Classification

Semester	iSnap	
	correct	incorrect
S16	24	41
F16	16	22
S17	12	17
F17	11	28
Total	63	108

Semester	CodeWorkout	
	correct	incorrect
S19	156	151
F19	223	265
Total	379	416

3.3 Experiments

3.3.1 Models Configuration

We conducted a series of experiments across both domains by comparing ASTNN against the state-of-the-art model *Code2Vec* and three *token-based* classic ML models.

Three Token-based ML Models: Three classic ML models, K-Nearest Neighbors (KNN), Logistic Regression (LG), and Support-Vector Machine (SVM) are explored. Following prior token-based approach, we applied TF-IDF to extract textual features [42, 34]. The input sentence for TF-IDF is the sequence of AST-tokens, which is generated by the pre-order traversal of original ASTs. For each of the three models, we explored different parameters to obtain the best results. For KNN, we had $k = 10$, for LG we used $L1$ regularization, and for SVM we used *linear* kernel. Those parameters are tuned from 10-fold cross-validation with grid search, and all three models are implemented through the sklearn library.

Two AST-based Deep Learning Models: Code2Vec takes a set of AST-based paths as input, where the number of paths may vary from different student submissions. Thus we manually padded the number of paths to 100 over all code submissions. During the training, we set the maximum training epochs as 200, with the *patience* of early stopping set to 100, tuned learning rate to 0.0002. Linear layer and embedding dimensions are kept default to 100. To ensure a highest efficiency of the model, we set the batch size as the full batch. For ASTNN, the inputs are a set of ST-trees, and we padded the statement sequences to the maximum length to accommodate the longest sequence before feeding to Bi-GRU. During the training, we leverage 32 as batch size, 0.001 as learning rate, and keep the max training epoch as 50. The encoding dim for the statement encoder is set to 128, and the number of hidden neurons for Bi-GRU is set to 100. We implemented both ASTNN and Code2Vec in Pytorch. Same as the classic models, 10-fold cross-validation was applied for hyperparameter tuning.

For the task of student program classification, we did not compare ASTNN and Code2Vec against any models that used expert-designed features for two reasons: one is that the expert-designed features are only available for iSnap but not CodeWorkout; and the other reason is that these expert-designed features are used to determine the ground truth label of the student's final submission in iSnap.

3.3.2 Evaluation Metrics

Our models were evaluated using Accuracy, Precision, Recall, F1 Score, and AUC (Area Under ROC curve). Accuracy represents the proportion of students whose labels were correctly identified. Precision is the proportion of students who were predicted to be *incorrect* by each model were actually in the *incorrect* group. Recall tells us what proportion of students, who will actually be *incorrect*, were correctly recognized by the model. F1 Score is the harmonic mean of Precision and Recall that sets their trade-off. AUC measures the ability of models to discriminate groups with different labels. Given the nature of the task, in the following, we consider Accuracy and AUC as the most important metrics because the former is most commonly accepted while AUC is believed to be generally more robust.

Finally, it is important to emphasize that all models were evaluated using semester-based temporal cross-validation for both domains in this task, which only applied data from previous semesters for training and is a much stricter approach than the standard cross-validation.

3.4 Results

Table 2 and 3 compare the performing of the five models in iSnap and CodeWorkout respectively. In iSnap, among the three token-based models, LG and SVM have very similar performance as both have an accuracy score of 0.6604; moreover the best AUC and Precision are from LG and the best Recall and F1 are from SVM. Both LG and SVM outperform KNN on all metrics. While in CodeWorkout, Table 3 shows that the best accuracy, AUC, and Precision are from SVM and the best Recall and F1 are from KNN. Between the two AST-based models, ASTNN outperforms Code2Vec in both domains. It suggests that across the two different student programming environments, ASTNN is more effective than Code2Vec on the task of student program classification.

The comparisons between AST-based models with token-based models show the former significantly out-perform the latter in both domains; the only exception is that SVM with token has the highest precision in Java (Table 3). Note that here the difference between the SVM and ASTNN on

للاستشارات
www.manaraa.com

Table 2: Student Program Classification Results in iSnap

Feature	Model	Accuracy	Precision	Recall	F1_score	AUC
Majority Baseline		0.6321	-	-	-	0.5
Tokens	**KNN**	0.6132	0.7321	0.6119	0.6667	0.6137
	LG	**0.6604**	**0.8298**	0.5821	0.6842	**0.6885**
	SVM	**0.6604**	0.7460	**0.7015**	**0.7231**	0.6456
ASTs	**Code2Vec**	0.6810	0.8038	0.6786	0.7239	0.7017
	ASTNN	**0.8113****	**0.8730****	**0.8209****	**0.8462****	**0.8079****

Note: best models in each group are in **bold**, and the overall best labeled with **

Table 3: Student Program Classification Results in CodeWorkout

Feature	Model	Accuracy	Precision	Recall	F1_score	AUC
Majority Baseline		0.5430	-	-	-	0.5
Tokens	**KNN**	0.8709	0.8915	**0.8679**	**0.8795**	0.8712
	LG	0.8299	0.8922	0.7811	0.8330	0.8345
	SVM	**0.8770**	**0.9437****	0.5093	0.6616	**0.8822**
ASTs	**Code2Vec**	0.9241	0.9299	0.9359	0.9329	0.9475
	ASTNN	**0.9529****	**0.9416**	**0.9736****	**0.9573****	**0.9509****

Note: best models in each group are in **bold**, and the overall best labeled with **

Precision is rather small while the former has a much worse accuracy, F1-score, and AUC than ASTNN.

To summarize, our results show that in both domains, ASTNN achieves the best performance. These results show that by capturing the meaningful *linguistic* structure in student code, ASTNN is indeed more robust on the task of student program classification. Given its effectiveness, we further explored the effectiveness of Temporal-ASTNN which combines ASTNN with powerful temporal model LSTM on the task of student success early prediction.

4. STUDENT SUCCESS EARLY PREDICTION

For student success early prediction task, Temporal-ASTNN is compared against the original ASTNN and other temporal models. As mentioned in Section 3.1.2, here we only explored the early prediction task in iSnap.

4.1 Task Description

In iSnap, we have a total of 171 students and 31,064 temporal snapshots. Following the definitions used in prior research [23], the *successful* students are those who completed the programming assignment within one hour *and* got full credit while the rest are counted as *unsuccessful*. We have 59 *successful* and 112 *unsuccessful* ones. The detailed statistics for iSnap dataset are shown in Table 4. Note that for the purpose of learning, unsuccessful students are of interest for this classification task.

To predict student early success, we are given the first *up to n minutes* of a student's sequence data and our goal is to predict whether the student will successfully complete the programming assignment at any given point in the remainder of the sequence. To conduct this task, we left-aligned all the students' trajectories by their starting times and our *observation window* (the part of data used to train and test different machine learning models) includes the sequences from the very beginning to the first n minutes. If a student's trajectory is less than n minutes, our observation window will include their entire sequence *except* the last one.

It is worth noting that student success early prediction is a much more challenging task compared to program classification: 1) besides the *linguistic* nature in student code, it also involves temporal information, and 2) the observation window is very early and thus student final submissions are not available for training or testing.

4.2 Experiments

4.2.1 Models Configuration

To further explore the power of ASTNN, we did extensive experiments and compared it with the start-of-the-art expert-designed features [43] and token-based features on the student success early prediction task. For each of the feature embedding (expert, token, AST), we explored two categories of models: the *last value*-based Logistic Regression (LG) models, and the *temporal* LSTM models. Note that LG is selected because, among the three classic ML methods explored on the task of student program classification in iSnap, LG has achieved the highest accuracy and AUC.

Last-Value Models: Motivated by prior work, we used a "Last Value" approach [4, 37, 23] to treat the last measurements within the given observation window as the input to train models. For early prediction settings, we truncated all the sequences in the training dataset in the same way as the testing dataset. For example, when our observation window is the first 4 minutes, we will only apply the last values in

المنارة للاستشارات
www.manaraa.com

Table 4: Detailed data statistics for iSnap, including total steps, total time spent in minutes, and the success labels distribution for each of the four semesters.

Semester	Total Steps				Total Time (minutes)				Success Labels	
	min	max	median	mean(std)	min	max	median	mean(std)	successful	unsuccessful
S16	10	1024	169	199 (175)	0.533	95.667	20.733	22.777 (17.149)	23	42
F16	28	884	121	167 (168)	3.283	119.083	16.325	22.379 (24.177)	15	23
S17	15	439	75	112 (94)	2.817	62.983	14.167	16.347 (11.872)	12	17
F17	10	2276	100	219 (376)	1.65	189.667	19.1	28.224 (33.869)	9	30

Table 5: iSnap Student Success Early Predictions at First-4-minute Only

Data	Feature	Model	Accuracy	Precision	Recall	F1_score	AUC
		Majority Baseline	0.6604	-	-	-	0.5
Last-Value	**Expert**	**Expert-LG**	0.6226	**0.8261****	0.5429	0.6552	**0.6603**
	Tokens	**Token-LG**	0.5566	0.7170	0.5429	0.6179	0.5631
	ASTs	**ASTNN**	**0.6698**	0.7612	**0.7286**	**0.7445**	0.6421
Temporal	**Expert**	**Expert-LSTM**	**0.7075**	0.7191	0.9143	0.8050	0.6099
	Tokens	**Token-LSTM**	0.6792	0.6915	**0.9286****	0.7927	0.5615
	ASTs	**Temporal-ASTNN$_{Trunc}$**	**0.7642****	0.7711	0.9143	**0.8366****	**0.6933****
		Temporal-ASTNN$_{Entire}$	0.7453	**0.7722**	0.8714	0.8188	0.6857

Note: best models in each group are in **bold**, and the overall best labeled with **

the sequence within the first-4-minute observation window and use them as inputs for each model. More specifically, we used the expert features of the last submission within the observation window to train and test expert-LG; similarly, the tokens from the last snapshot within the observation window to train and test token-LG; and the ASTs of the last submission within the observation window for both training and testing the original ASTNN.

Temporal Models: We applied LSTM to handle the temporal sequences of student code. Here we used the *temporal* sequences in the observation window for early predictions. Specifically for a given first-n-minute observation window: we used the sequences of expert features to train and test expert-LSTM; the sequences of token features to train and test token-LSTM. For Temporal-ASTNN, we explored Temporal-ASTNN$_{Trunc}$ and Temporal-ASTNN$_{Entire}$. Both models would first convert student code sequences in the observation window into sequences of AST vectors and then feed them into LSTM. They only differ on how their AST vectors are trained: the former uses truncated sequences while the latter uses entire sequences (see Section 2.1.3).

To summarize, we analyze two main model settings: *last-value* and *temporal*, together with three different feature embeddings: *expert*, *tokens*, and *ASTs*. Thus in total we explored the effectiveness of six models.

4.2.2 Evaluation Metrics

For student success early prediction, all the models are evaluated using Accuracy, Precision, Recall, F1 Score, and AUC. Similarly to the first task, we consider Accuracy and AUC as the most important metrics, and the more stringent semester-based temporal cross-validation was carried out.

4.3 Results

We present our results of student success early prediction by first comparing the effectiveness of all six models on first-4-minute early prediction and then by exploring their average performance across different observation windows up to the first-10-minute data.

4.3.1 Results at First-4-minute Only

Table 5 shows different performance measures of all the six models at first-4-minute. In the group of *Last-Value* models, ASTNN has the best accuracy, Recall and F1 scores while the best AUC and Precision are from Expert-LG, and both of them have better performance than Token-LG. Actually, in terms of accuracy, Expert-LG and Token-LG perform worse than the simple majority baseline. This is probably either because only relying on the first-4-minute is too early or because the last snapshot of the first-4-minute does not provide enough information for these models to make effective early predictions. The fact that across the five evaluation metrics, the best performance either comes from Expert feature or comes ASTNN suggests that ASTNN is comparable to expert-designed features because of its ability of handling the *linguistic* structure of student syntactic code.

In the *Temporal* group, Temporal-ASTNN based models are the best. More specifically, both Temporal-ASTNN$_{Trunc}$ and Temporal-ASTNN$_{Entire}$ outperform Expert-LSTM and Token-LSTM on accuracy, AUC, precision and F1 scores, except that the best recall is from Token-LSTM. Between the two Temporal-ASTNN models, Temporal-ASTNN$_{Trunc}$ is generally better than Temporal-ASTNN$_{Entire}$ as it achieves higher accuracy, Recall, F1-score, and AUC. This is probably because by using the truncated training data for representation learning, Temporal-ASTNN$_{Trunc}$ is more likely to capture the temporal information that are not only predictive of student success but also more likely to be observed in the testing with only the first-4-minute data.

When further comparing temporal models with last-value models, we can see that all *temporal* models achieve better accuracy than their corresponding *last-value* models. It is reasonable since *temporal* models are able to capture the temporal information related to student success from the

المنارة للاستشارات
www.manaraa.com

Table 6: iSnap Student Success Early Predictions in First-10-minute Overall

Data	Feature	Model	Accuracy	Precision	Recall	F1_score	AUC
	Majority Baseline		0.6604	-	-	-	0.5
Last-Value	Expert	Expert-LG	0.6566 (0.05)	**0.8209**** **(0.06)**	0.6229 (0.11)	0.7017 (0.06)	**0.6725** **(0.05)**
	Tokens	Token-LG	0.5528 (0.02)	0.7072 (0.03)	0.5571 (0.07)	0.6203 (0.04)	0.5508 (0.03)
	ASTs	ASTNN	**0.6642** **(0.01)**	0.7635 (0.03)	**0.7200** **(0.07)**	**0.7379** **(0.02)**	0.6378 (0.03)
Temporal	Expert	Expert-LSTM	0.7189 (0.03)	0.7305 (0.05)	0.9343 (0.04)	0.8145 (0.01)	0.6255 (0.07)
	Tokens	Token-LSTM	0.6887 (0.02)	0.6966 (0.03)	**0.9429**** **(0.04)**	0.8001 (0.01)	0.5687 (0.05)
	ASTs	Temporal-ASTNN$_{\text{Trunc}}$	0.7396 (0.02)	0.7597 (0.03)	0.8914 (0.03)	**0.8190**** **(0.01)**	0.6679 (0.04)
		Temporal-ASTNN$_{\text{Entire}}$	**0.7472**** **(0.02)**	**0.7932** **(0.04)**	0.8316 (0.04)	0.8110 (0.02)	**0.6943**** **(0.03)**

Note: best models in each group are in **bold**, and the overall best labeled with **

temporal sequences, but such information is not available to *last-value* models.

Generally speaking, Temporal-ASTNN achieves the best performance at the first-4-minute observation window, which indicates that by combining ASTNN with LSTM, the temporal-ASTNN is able to learn the *temporal* and *linguistic* knowledge from student code sequences.

4.3.2 Results in First-10-minute Overall

Figure 6 (a) and (b) report Accuracy and AUC performance respectively for four models predicting student success: three *temporal* models and the best *last-value* model, ASTNN. For each graph, x-axis is the observation window of early prediction, here we vary the observation window from the first 2 minutes up to 10 minutes; and y-axis is the Accuracy/AUC score. As shown in Table 1, students generally take 10 to 60 minutes to complete the task and thus we took a measurement every 2 minutes for the first 10 minutes to generate the early stage predictions for each model. Table 6 show the comparison of all six models for the student success early prediction in first-10-minute observation windows, we reported the mean value and corresponding standard deviation (in parenthesis) for each evaluation metric.

Table 6 shows a similar pattern as we observed earlier in Table 5. In the group of *Last-Value* models, ASTNN outperforms Expert-LG and Token-LG. Specifically, ASTNN continues to achieve the best accuracy, Recall and F1 scores in the first 10 minutes, and Expert-LG has the best AUC and Precision scores. In the group of *temporal* models, Temporal-ASTNN based models are still the best overall, with higher scores on accuracy, AUC, Precison and F1. Additionally, Temporal-ASTNN$_{\text{Entire}}$ is shown to be slightly better than Temporal-ASTNN$_{\text{Trunc}}$ as it achieves higher accuracy, AUC and Precision.

Both Figures 6 (a) and (b) show that Temporal-ASTNN$_{\text{Entire}}$ is the best model for student success early prediction as it stays on the top across all sizes of the observation window. As the length of observation window extends, all *temporal* models in general perform better, while the performance of *last-value* models fluctuates. This is because that training data includes more and more information and hereby the performance of *temporal* models improves over longer sequences. After 6 minutes, Expert-LSTM starts to perform as good as Temporal-ASTNN, which is not surprising. As the expert features are designed to detect student state for final grading, and student states will be more and more closer to their final submissions with the longer sequences. The fact that the best early predictions come from Temporal-ASTNN really suggests that addressing both *linguistic* and *temporal* nature of student code sequences brings us closer to the truth of student learning procession during programming, especially for the early stage (first 6 minutes).

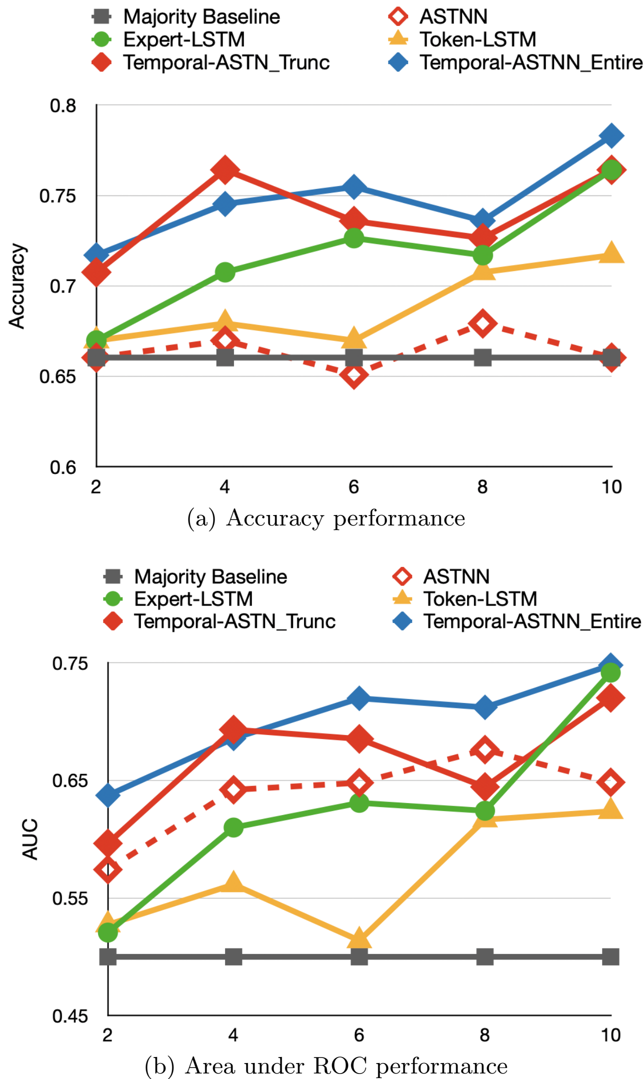
5. RELATED WORK

5.1 Linguistic-based Models for Programming

A wide range of work has applied NLP techniques for programming. Traditionally, some prior work directly uses the tokens of ASTs for source code tasks [38, 12], by treating programming languages as natural languages. Despite some similarities, programming languages and natural languages [25] differ in some important aspects. Programming is a complex activity, and thus programs contain rich and explicit structural information. Recently, deep learning models has shown the potential to grasp more information from AST in many tasks. For example, TBCNN [24] takes the whole AST of code as input and performs convolution computation over tree structures, and it outperforms token-based models in program functionalities classification and bubble-sort detection. In the educational domain, Piech et al. (2015) proposed NPM-RNN to simultaneously encode preconditions and postconditions into points where a program can be used as a linear mapping between these points [30]. Gupta et al. (2019) presented a tree-CNN based method, that can localize the bugs in a student program with respect to a failing test case, without running the program [14]. More recently, ASTNN and Code2Vec has shown great success.

Siting at the root of AST, ASTNN [42] was proposed to handling the long-term dependency problems when taking the large AST as input directly. AST is a form of representing abstract syntactic structure of the source code [5], and it

المنارة للاستشارات
www.manaraa.com



(a) Accuracy performance

(b) Area under ROC performance

Figure 6: Student Success Early Prediction on *iSnap*, last-value models are in dashed lines with empty symbols, temporal models are in solid lines with solid symbols, dark grey lines are from the majority baseline.

has been widely used in the domain of source code analysis. Similar to long texts in NLP, large ASTs can make deep learning models venerable to gradient vanishing problems. To address the issue, ASTNN splits the large AST of one code fragment into a set of small trees in statement-level and performs code vector embedding. It achieves state-of-the-art performance in both code functionalities classification and clone detection.

Code2Vec [1], on the other hand, utilizes AST-based paths and attention mechanism to learn code vector representation. Instead of a set of ST-trees, it takes a collection of leaf-to-leaf paths as input, and applies an attention layer to average those vectors. As a result, the attention weights can help to interpret the importance of paths. Code2Vec has shown to be very effective in predicting the names for program entities. Shi et al. (2021) also applied Code2Vec on a block-based programming dataset and used the learned embedding to cluster incorrect student submissions [34].

As far as we know, none of prior work has directly compared the effectiveness of ASTNN against Code2Vec. And in this work, we did extensive experiments across two programming domains: one is a block-based novice programming environment where the data size is relatively small; the other is a web programming platform in Java, in which more labeled data is available. Our results consistently suggest that ASTNN is able to capture more insights from student programs for correctness prediction.

5.2 Student Modeling for Programming

Student modeling has been widely and extensively explored by utilizing student temporal sequences. For example, BKT [8] and BKT-based models have been shown to be effective in predicting students' overall competence [26], predicting the students' next-step responses [41, 3, 27, 20], and the prediction of post-test scores [18, 22]. In recent years, deep learning models, especially Recurrent Neural Network (RNN) or RNN-based models such as LSTM have also been explored in student modeling [29, 36, 17, 39, 40, 19]. Some work showed that LSTM has superior performance over BKT-based models [22, 29] or Performance Factors Analysis [28]. However, it has also been shown that RNN and LSTM did not always have better performance when the simple, conventional models incorporated other parameters [17, 39].

In the programming domain, prior research has explored various temporal models for modeling student learning progression. For example, Wang et al. (2017) applied a recursive neural network similar to [30] as the embedding for student submission sequence, then feed them into a 3-layer LSTM to predict the student's future performance. Please note that the work is quite different from our proposed Temporal-ASTNN. In Temporal-ASTNN, all the components are optimized together during training, while they applied a global embedding to generate the input sequences for LSTM. On the other hand, Emerson et al. (2019) have utilized four categories of features: prior performance, hint usage, activity progress, and interface interaction to evaluate the accuracy of Logistic Regression models for multiple block-based programming activities [11]. In our earlier work, we have used the expert-designed features for a block-based programming problem to train various temporal models, then made early predictions on student learning outcomes [21, 23].

To our best knowledge, while most of the previous studies on analyzing student programming data treated student code as either *linguistic* or *temporal*, no prior work has combined the two characteristics of programming data for student learning progression. Thus our proposed Temporal-ASTNN is the first attempt to addressing *both* aspects in student code.

6. CONCLUSIONS

Tracing student learning progression at early stage is a crucial component of student modeling, since it allows tutoring systems to intervene by providing needed support, such as a hint, or by alerting an instructor. Both prediction tasks involved in this work are challenging, especially the early prediction task because: 1) the open-ended nature of programming environment hinders the prediction of student fi-

المنارة للاستشارات
www.manaraa.com

nal success, and 2) it is extremely hard to learn a meaningful representation from student code. In this work, we conducted a series of experiments to investigate the effectiveness of Temporal-ASTNN for student learning progression. We first evaluated ASTNN against Code2Vec on the task of classifying the correctness of student programs across two domains. Our results show that ASTNN consistently outperforms the other models including Code2Vec and other token-based baselines in both domains. And we can also find that AST-based models generally achieve better performance than token-based models, which is consistent with prior research [24, 42]. In the second task of student early prediction, we explored three different categories of features: expert, tokens, and ASTs. And further compared Temporal-ASTNN with other temporal models embedded with different feature set, as well as non-temporal baselines. Our findings can be concluded as follows: 1) temporal models usually outperforms non-temporal (last-value) models; 2) token-based models can only capture very limited information from student code; and 3) Temporal-ASTNN is the best out of all models in the early prediction task, it can achieve good performance with only the first-4-minute data.

Limitations: There are two main limitations in this work. First, we only explored the effectiveness of Temporal-ASTNN on one important student modeling task in one programming environment, and thus it is not clear whether the same results will hold for different tasks or in other programming domains. Second, time-aware LSTM [6] has shown to outperform LSTM on various early prediction tasks [23], while in this work we only compared our Temporal-ASTNN against normal LSTM without considering time-awareness. Nevertheless, one of the main goal in this work is to investigate the robustness of Temporal-ASTNN from both sequential and temporal embedding. Thus we have two different type of models (last-value vs. temporal) as well as another two different features (expert and tokens). Our experiments results have shown its superiority on both aspects, but still, we are not clear about the effects of time-awareness.

Future Work: An important direction for future work is to investigate the time-awareness on Temporal-ASTNN to determine how it contributes to the model in the same task. In addition, we are planning to employ Temporal-ASTNN to other temporal tasks or different domains to explore whether it continues to support improvement for programming environments. Also, this work will be applied to larger groups of students and longer programming tasks, along with integration of more informative features such as intervention and demographic features to develop more robust models.

7. ACKNOWLEDGMENTS

This research was supported by the NSF Grants: EXP: Data-Driven Support for Novice Programmers (1623470), Integrated Data-driven Technologies for Individualized Instruction in STEM Learning Environments(1726550), CAREER: Improving Adaptive Decision Making in Interactive Learning Environments (1651909), and Generalizing Data-Driven Technologies to Improve Individualized STEM Instruction by Intelligent Tutors (2013502).

8. REFERENCES

[1] 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.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] 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 *ITS*, pages 406–415, 2008.

[4] I. Batal, D. Fradkin, J. Harrison, F. Moerchen, and M. Hauskrecht. Mining recent temporal patterns for event detection in multivariate time series data. In *SIGKDD*, pages 280–288. ACM, 2012.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[6] I. M. Baytas, C. Xiao, X. Zhang, F. Wang, A. K. Jain, and J. Zhou. Patient subtyping via time-aware lstm networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 65–74, 2017.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

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

[9] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[10] S. H. Edwards and K. P. Murali. Codeworkout: short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 188–193, 2017.

[11] A. Emerson, F. J. Rodríguez, B. Mott, A. Smith, W. Min, K. E. Boyer, C. Smith, E. Wiebe, and J. Lester. Predicting early and often: Predictive student modeling for block-based programming environments. *International Educational Data Mining Society*, 2019.

[12] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[13] D. Garcia, B. Harvey, and T. Barnes. The Beauty and Joy of Computing. *ACM Inroads*, 6(4):71–79, 2015.

[14] R. Gupta, A. Kanade, and S. Shevade. Neural attribution for semantic bug-localization in student programs. *Network*, 1(P2):P2, 2019.

[15] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

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

www.manaraa.com

[17] M. Khajah, R. V. Lindsey, and M. C. Mozer. How deep is knowledge tracing? *arXiv preprint arXiv:1604.02416*, 2016.

[18] C. Lin and M. Chi. Intervention-bkt: incorporating instructional interventions into bayesian knowledge tracing. In *ITS*, pages 208–218. Springer, 2016.

[19] C. Lin and M. Chi. A comparisons of bkt, rnn and lstm for learning gain prediction. In *AIED*, pages 536–539. Springer, 2017.

[20] C. Lin, S. Shen, and M. Chi. Incorporating student response time and tutor instructional interventions into student modeling. In *UMAP*, pages 157–161. ACM, 2016.

[21] Y. Mao. One minute is enough: Early prediction of student success and event-level difficulty during novice programming tasks. In *In: Proceedings of the 12th International Conference on Educational Data Mining (EDM 2019)*, 2019.

[22] Y. Mao, C. Lin, and M. Chi. Deep learning vs. bayesian knowledge tracing: Student models for interventions. *JEDM*, 10(2):28–54, 2018.

[23] Y. Mao and S. Marwan. What time is it? student modeling needs to know. In *In proceedings of the 13th International Conference on Educational Data Mining*, 2020.

[24] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. Tbcnn: A tree-based convolutional neural network for programming language processing. *arXiv preprint arXiv:1409.5718*, 2014.

[25] J. F. Pane, B. A. Myers, et al. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2):237–264, 2001.

[26] Z. A. Pardos and N. T. Heffernan. Modeling individualization in a bayesian networks implementation of knowledge tracing. In *UMAP*, pages 255–266. Springer, 2010.

[27] Z. A. Pardos and N. T. Heffernan. Kt-idem: Introducing item difficulty to the knowledge tracing model. In *UMAP*, pages 243–254. Springer, 2011.

[28] P. I. Pavlik, H. Cen, and K. R. Koedinger. Performance factors analysis –a new alternative to knowledge tracing. In *AIED*, pages 531–538, 2009.

[29] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein. Deep knowledge tracing. In *NIPS*, pages 505–513, 2015.

[30] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *International conference on machine Learning*, pages 1093–1102. PMLR, 2015.

[31] T. Price, R. Zhi, and T. Barnes. Evaluation of a data-driven feedback algorithm for open-ended programming. *International Educational Data Mining Society*, 2017.

[32] T. W. Price, Y. Dong, and D. Lipovac. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, pages 483–488, 2017.

[33] 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.

[34] Y. Shi, K. Shah, W. Wang, S. Marwan, P. Penmetsa, and T. Price. Toward semi-automatic misconception discovery using code embeddings. In *The 11th International Conference on Learning Analytics & Knowledge (LAK 21)*, 2021.

[35] R. Socher, C. C.-Y. Lin, A. Y. Ng, and C. D. Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011.

[36] S. Tang, J. C. Peterson, and Z. A. Pardos. Deep neural networks and how they apply to sequential education data. In *L@S*, pages 321–324. ACM, 2016.

[37] L. Wang, A. Sy, L. Liu, and C. Piech. Learning to represent student knowledge on programming exercises using deep learning. *International Educational Data Mining Society*, 2017.

[38] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.

[39] K. H. Wilson, Y. Karklin, B. Han, and C. Ekanadham. Back to the basics: Bayesian extensions of irt outperform neural networks for proficiency estimation. *arXiv preprint arXiv:1604.02336*, 2016.

[40] X. Xiong, S. Zhao, E. Van Inwegen, and J. Beck. Going deeper with deep knowledge tracing. In *EDM*, pages 545–550, 2016.

[41] M. V. Yudelson, K. R. Koedinger, and G. J. Gordon. Individualized bayesian knowledge tracing models. In *AIED*, pages 171–180. Springer, 2013.

[42] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[43] R. Zhi, T. W. Price, N. Lytle, Y. Dong, and T. Barnes. Reducing the state space of programming problems through data-driven feature detection. In *EDM (Workshops)*, 2018.

المنارة للاستشارات
www.manaraa.com