Yang, J. C. et al. (Eds.) (2018). Proceedings of the 26th International Conference on Computers in Education. Philippines: Asia-Pacific Society for Computers in Education

Knowledge Tracing Within Single Programming Exercise Using Process Data

Bo JIANG[a*] , Yun YE[a], Haifeng ZHANG[b]
[a] *College of Education Science and Technology, Zhejiang University of Technology, China*
[b] *School of Computer Science, Carnegie Mellon University, USA*
*bjiang@zjut.edu.cn

Abstract: Knowledge tracing is a core technology in many intelligent learning systems. In this paper, we propose a novel knowledge tracing method that predicts learner's knowledge state within a single programming exercise. Given a programming task, a student's intermediate solution is represented by an abstract syntax tree and evaluated by computing its tree edit distance to the best solution. With the measure of solution quality, the learning trajectory of each student can be encoded as a real-valued sequence. Using the mean value of the sequence as a primary feature, we developed a logistic regression model to predict students' knowledge state. We compared our method with three popular models on a large-scale dataset collected from a classic block-based programming task. The experimental results suggest that the proposed method that captures features derived from student's problem-solving processes can significantly improve the prediction performance.

Keywords: Bayesian knowledge tracing, deep knowledge tracing, additive factor model, block-based programming

1. Introduction

Programming exercise for students is more than just coding. Indeed, it is a cognitive process that requires rather sophisticated computational problem-solving skills regarding concepts, practices and perspective (Mitchel *et al*., 2009). The assessment of such computational skills has important pedagogical value in computational thinking education. Most research on measuring computational skills in programming tasks have relied on the final code students completed checking the use of programming constructs such as loops, conditionals and logic. This approach is incomplete as it ignores the learning paths that can be substantially different among individuals and thus better reflect students' mastery of knowledge and skills than the finished products (Grover *et al*., 2017).

Knowledge tracing (KT) is a task that estimates students' proficiency of the required knowledge components using data collected from their problem-solving processes. Predicting students' knowledge states allows educators to recommend suitable learning resources in students' needs. The most popular knowledge tracing method in literature is the Bayesian Knowledge Tracing (BKT). It models a student's knowledge by a latent variable in a hidden Markov chain and updates its state by observing the correctness of each attempt in which he or she applied the knowledge and skills in answering the question (Corbett and Anderson, 1995). Another important KT method is called Additive Factor Model (AFM), which estimates the probability of a student being correct on the first attempt using a logistic model (Cen, *et al*., 2007). Most recently, Piech *et al., (2015*) proposed a method namely Deep Knowledge Tracing (DKT) that utilized long short-term memory (LSTM) model to predict students' knowledge state. All these KT techniques require students to complete multiple exercises and use information like their performance on knowledge component or the number of attempts to build prediction models. Inspired by this fundamental observation, in this paper, we seek to develop a more comprehensive and efficient solution to estimate knowledge state using deeper process data within a single programming exercise.

To the best of our knowledge, the earliest work in our research setting is a DKT-based method proposed by Wang et al. (2017). Their method uses a recursive neural network to vectorize the abstract syntax tree (AST) representation of student programs, and feeds them into a LSTM

network. Although that approach seems promising, it is also complicated, unintuitive and computationally expensive. Different from the way they used AST, we propose a simpler and yet more efficient KT method based on the standard logistic regression model. Particularly, instead of using AST directly, we introduced a metric called approaching index (AI) to quantify the closeness of students' current program to the best solution based on the idea of tree edit distance (TED). With the AI metric, each student's learning path, i.e., a series of programming solutions is represented by an AI sequence. We approximated a student's overall performance by the mean of all the values in an AI sequence, namely AIScore. Finally, we used three features, i.e., correctness of student's program, number of attempts and *AIScore* to construct a logistic regression model for prediction. We call our proposed method Process Knowledge Tracing (PKT), due to its strong connection to the learning process data.

2. Knowledge Tracing

One of the earliest knowledge tracing method is Bayesian Knowledge Tracing (BKT) that proposed by (Corbett and Anderson, 1995). BKT assumes that at any given opportunity to demonstrate a skill, e.g., student solves a programing problem, the knowledge state of a leaner is a binary variable, i.e., mastered or not, and the observed performance is a correct or incorrect response. The probability that the student has mastered the skill can be updated based on our observation on student's performance in each practice opportunity. In classical BKT, only the first attempt for each opportunity is taken into account, and it is assumed that each item corresponds to only a single skill (or knowledge component). Technically, the BKT model can be characterized by four parameters as follows, where the first two are learning parameters, while the last two are performance parameters.

- $P(L_n)$: Probability the skill is mastered after n opportunities of practices.
- $P(T)$: Probability of student's knowledge of a skill transitioning from *not known* to *known* state after a practice opportunity
- $P(G)$: Probability the student will guess correctly if the skill is not mastered.
- $P(S)$: Probability the student will make a mistake if the skill is mastered.

More precisely, BKT uses the following equations to infer student's latent knowledge based on his or her performance.

$$P(L_n|Correct_n) = \frac{P(L_{n-1}) * (1 - P(S))}{P(L_{n-1}) * (1 - P(S)) + (1 - P(L_{n-1})) * P(G)}$$

$$P(L_n|Incorrect_n) = \frac{P(L_{n-1}) * P(S)}{P(L_{n-1}) * P(S) + (1 - P(L_{n-1})) * (1 - P(G))}$$

$$P(L_n) = P(L_n|Observe_n) + (1 - P(L_n|Observe_n)) * P(T)$$

Most importantly, from above equations, the probability of a student will practice correctly in an upcoming practice is computed as

$$P(L_{n+1}) = P(L_n)(1 - P(S)) + (1 - P(L_n)) * P(G)$$

Knowledge tracing models applying logistic regression often define that the probability of a correct response to a task is a mathematical function of student and skill parameters. These models assume that the binary task response (correct/incorrect) follows a Bernoulli distribution. A notable model in this kind is the Addictive Factors Model (AFM) (Cen, *et al.*, 2007), which is a logistic model that predicts the likelihood of the student being correct on the first try on a task. More precisely, AFM computes

$$P(y_{ij} = 1|\sigma_{\theta_i}^2, \beta, \gamma) = \frac{1}{1 + e^{[-(\theta_i + \sum_{k=1}^{K} q_{jk}(\beta_k + \gamma_k T_{ik}))]}}$$

Where

- y_{ij} is the response of student i on task j.
- $\theta_i N(0, \sigma_{\theta_i}^2)$ is a random effect referring to the proficiency of student i.
- β_k is the coefficient for the learning rate of knowledge component k.
- T_{ik} is the number of practice opportunities student i has had on knowledge component k.
- $q_{jk} = 1$ if task j uses skill k, $q_{jk} = 0$ otherwise.

- K is the total number of knowledge components in the Q-matrix.
- $q_{jk} = 1$ if task j uses skill k, $q_{jk} = 0$ otherwise.
- K is the total number of knowledge component in the Q-matrix.

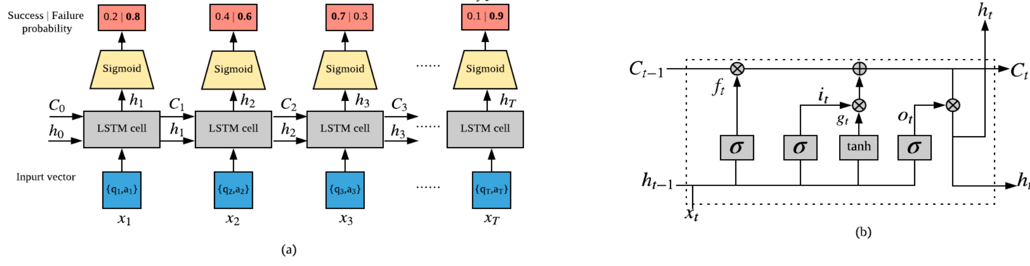

Figure 1. (a) A long short-term memory network for knowledge tracing. (b) The structure of LSTM cell

Most recently, (Piech *et al., 2015*) proposed Deep Knowledge Tracing (DKT) that uses the recurrent neural network to predict student's responses. The "deep" in its name refers to the recurrent structure of the neural network and the "depth" of information over time (Xiong *et al*., 2016). In DKT, the input vectors are representations of whether the student answered a particular question correctly or not at the previous time step, and the output vectors are representations of the probability that a student will get the question correctly at the following time step. Figure 1 (a) demonstrates an LSTM-based DTK model and its cell structure. Specifically, the LSTM update operations are given by

$$i_t = \sigma(W_{ix}x_t + W_{xh}h_{t-1})$$
$$f_t = \sigma\left(W_{fx}x_t + W_{fh}h_{t-1}\right)$$
$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1})$$
$$g_t = tanh\left(W_{gx}x_t + W_{gh}h_{t-1}\right)$$
$$C_t = f_t \cdot C_{t-1} + i_t \cdot g_t$$
$$h_t = o_t \cdot C_t$$

Where the input x_t in each time point is a one-hot encoding of the student's response tuple $x_t = q_t, a_t$, which represents whether the student answered a particular question q_t correctly ($a_t = 1$) or not ($a_t = 0$) at the time step t; h_{t-1} is the LSTM output from previous time step; σ and $tanh$ represent *sigmoid* and *tanh* non-linear transfer functions; W are the model weight parameters that can be learned when training the model. As shown in Figure 1 (b), y_t can be inferred at each time step by adding a sigmoid layer from the LSTM output h_t. As we can see from the above review, all the three popular knowledge tracing methods require students' multiple historical response data to predict future performance. However, rich historical response sequence is not existed, especially in open-ended learning environments.

3. Process Knowledge Tracing

A typical setting for our research problem is students are given a programing task (without loss of generality, we may assume there is only one best solution) that is allowed to be complete in multiple attempts with some programming language. In each attempt, students can compile programs and modify their programs according to the compiler's feedbacks, i.e., whether their last attempts were correct or not. Suppose we have access to their submitted programs, we can store each of them as an abstract syntax tree (AST), a finite, labeled and ordered tree, whose internal nodes are labeled by operators, and leaf nodes represent the operands of the node operators (Piech *et al., 2012*). The problem solving process in the programming task can be considered as a series of edit actions made to the source code. Note we can also think it of as a sequence of edits on an initial AST that transform it into other ASTs. Generally, three types of edit actions, i.e., insertion, deletion and replacement of tree nodes can change the state of an AST. If a student's current solution is incorrect, he/she will make one or more edits on his/her source code to solve the problem correctly. Moreover, if a student's current program is incorrect, there could be more than one pathways by which he/she can

reach the correct solution (or the goal state of AST). This means that there can be k different edit sequences $S = S_1, S_2, \ldots, S_k$ that can transform AST from T_1 to T_2. We used *tree edit distance* (TED) of AST, the shortest edit path from current solution to final solution, to evaluate every intermediate solution given by each student. We call it *approaching index* (AI) and give its formal definition as follow.

[**Definition 1**] *The approaching index of i-th student's j-th submission AI_{ij} is defined as the TED between the T_{ij} and the perfect solution T_p.*

$$AI_{ij} = d(T_{ij}, T_p) = min\,\gamma(S_{ij\to p})$$

Where $\gamma(S_{ij\to p})$ denotes the TED of all the possible edit sequences that can transform i-th student's current submission T_{ij} to the perfect solution T_p.

The minimization problem in Definition 1 is often treated as a dynamic programming problem, which can be solved efficiently by the well-known *Zhang&Shasha algorithm* (Zhang & Shasha, 1989). In this work, we applied *Zhang&Shasha* algorithm to compute the AI for each submission.

After the AI of each intermediate solution was computed, we obtained an AI sequence, which represents how the solution quality changed over time. Based on the AI sequence, we further constructed an overall measurement of the quality for each student. The metric is called *AIScore*, which is defined formally as

$$AIScore_i = \frac{\sum_{j=1}^{Q_i} AI_{ij}}{Q_i}$$

where Q_i denotes the number of attempts the i-th student made, i.e., the trajectory length. Consistent with prior research, we used *AIScore*, number of attempts, and the correctness of the current solution to predict student's performance on the next task that involves the same knowledge component as the previous one. Similar to AFM, a traditional logistic regression model is chosen in our current study.

4. Dataset

The dataset used in this work to compute AIs was generated when students solved one of the classic maze problems using a block-based programming language in the *Hour of Code*[1]. Instead of coding text, students drag and drop visual blocks to compose their programs. There are twenty classic maze problems that are aimed at teaching beginners fundamental programming constructs. In this paper, we focused on No.18 maze problem (*HOC18*), because it is a relatively complicated task that involves multiple programming constructs, such as *repeat-until*, *if-else*, and their nesting statements. The actual HOC18 problem and its perfect solution are shown in Figure 2.

This HOC18 dataset contains 79,553 unique code submissions and 83,955 trajectories, made by 263,569 students. All code submissions are stored as AST files, which can be downloaded from *Hour of Code*[2]. When preprocessing this dataset, we found 22,942 missing code submissions (19,257 invalid trajectories). Therefore, the valid number of code submission and trajectories are 49,533 and 64,698, respectively. There are 187,616 students who completed task No.18 and also attempted task No.19, both of which have the same knowledge component. Due to the missing code, we focused only on 164,221 students with valid code submissions and trajectories. Among these students, 150,875 of them solved the next problem HOC19 task successfully with a success rate about 91.87%. The HOC19 task is a little more complicated maze problem that has the same knowledge components with HOC18. When students completed HOC18 the system would jump to HOC19 task automatically. Observing students' performance in HOC19 task can help us figure out whether students learned knowledge from HOC18 task, especially for the students failed in HOC18.

5. Experiment results

[1]https://studio.code.org/courses
[2]https://code.org/research

In this section, we will compare the proposed KT method with the three most popular KT methods in the literature, i.e., BKT, AFM and DKT. The running environment is a MacBook laptop (OS 10.12.3) with Intel Core i5 2.7GHz CPU and 8GB memory. We did all data prepossessing and cleaning in Python 2.7, and used the Python package *zss*[3] (that implements Zhang&Shasha algorithm) to compute TEDs. The Python version of AFM implemented by (MacLellan *et al.,* 2015) and DKT by (Khajah *et al*., 2016) were used in our experiment. To make a fair comparison, all the four methods were run in a 10 fold cross-validation manner on the HOC18 dataset. Three widely used metrics, i.e., *accuracy*, *AUC* and *F1*, were used for performance evaluation.

Figure 2: The HOC 18 programming task (left) and it's best solution (right)

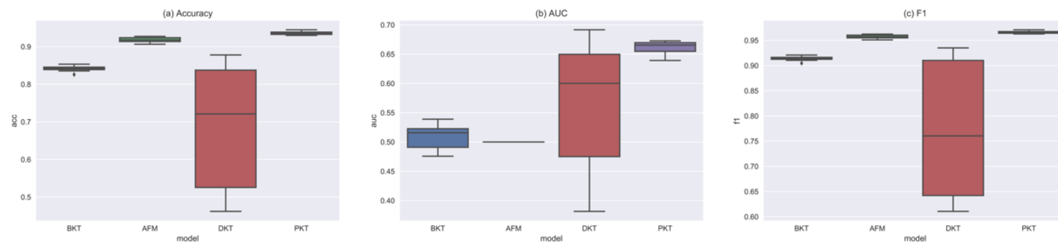

Figure 3: Boxplot of the prediction performance of the four methods

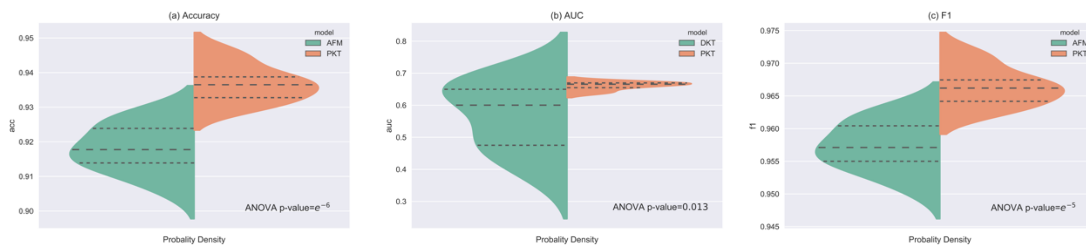

Figure 4: comparison between PKT and the best method among BKT, AFM and DKT on each indicator.

Figure 3 shows the box plot of the performance of the four models. First, we observe that our proposed PKT method achieves the highest mean value and the smallest deviation on all the three metrics. We also notice that AFM performs well on Accuracy and F1 index. Second, we see that the two logistic models (AFM and PKD) outperform the two sequence-based models (BKT and DKT) on Accuracy and F1 index. This result reveals potential limitation of the sequence-based approach: its predictive power can be extremely restricted when the lengths of input sequences are short. In fact, the presence of short sequences is common for many basic programming tasks that are designed to educate novices (e.g., kids) computational thinking skills. Third, the three simple models, BKT, AFM and PKT were found to have superior Accuracy and F1 score to DKT. This may be because DKT overfited our smaller amount data with its sheer number (about 16,400) of parameters. Moreover, the three simpler methods have just a few three parameters, saving significant amount of time in the model training. This finding also reminds us that if we do not have enough data, more sophisticated approach like DKT maybe not a good choice. Figure 4 compares PKT with the best method among the other three methods on each metrics. As depicted in the plot, PKT provides the higher average value and smaller deviation on all three performance measures. The ANOVA test

[3]https://github.com/timtadh/zhang-shasha

results, in the lower right corner of each figure, also shows that PKT performs significantly better than the remaining methods on all three metrics.

6. Conclusion

In this paper, we proposed a new knowledge tracing method that makes use of the data hidden in the intermediate solutions given by students in solving a single programming problem to estimate their knowledge mastery states. Compared with existing knowledge tracing methods, deeper process data was used to construct a logistic model. We compared our PKT method with three popular methods, e.g., BKT, AFM and DKT, on a large-scale dataset collected in a classic block-based programming task. The experimental results not only demonstrated the advantage of PKT over other methods, but also confirmed that the approaching index as we proposed is an effective and significant feature, which is unfortunately undiscovered in prior work on knowledge tracing.

Extending this model encompasses several potential directions to pursue. A technical challenge we encountered in developing current approach is how to compute the tree edit distances on AST both efficiently and accurately. In the future, we could explore using more efficient tree edit distance algorithms to compute the approaching index. Another interesting direction relates to the quest of how to extend the proposed model to open-end programming tasks (there can be more than one best solution). Lastly, we may also explore the possibility to directly feed the AI sequences into a LSTM model to do knowledge tracing.

Acknowledgements

This work is partly supported by the National Natural Science Foundation of China under Grant No. 61503340, Scientific Research Fund of Zhejiang University of Technology under No. Z20160133.

References

Cen, H., Koedinger, K. R., & Junker, B. (2007). Is Over Practice Necessary?-Improving Learning Efficiency with the Cognitive Tutor through Educational Data Mining. *Frontiers in artificial intelligence and applications*, *158*, 511.

Corbett, A. T.; Anderson, J. R. (1995). "Knowledge tracing: Modeling the acquisition of procedural knowledge". *User Modeling and User-Adapted Interaction*. **4** (4): 253–278.

Grover, S., Basu, S., Bienkowski, M., Eagle, M., Diana, N., Stamper, J. (2017). A Framework for Using Hypothesis-Driven Approaches to Support Data-Driven Learning Analytics in Measuring Computational Thinking in Block-Based Programming Environments A Framework for Using Hypothesis-Driven Approaches to Support Data-Driven Learning Ana. *ACM Transactions on Computing Education*, *17*(3), 1–25.

Khajah, M., Lindsey, R. V., & Mozer, M. C. (2016). How deep is knowledge tracing? In *Proceedings of the 9th International Conference on Educational Data Mining*.

MacLellan, C. J., Liu, R., & Koedinger, K. R. (2015). Accounting for Slipping and Other False Negatives in Logistic Models of Student Learning. In *Proceedings of the 8th International Conference on Educational Data Mining*.

Mitchel, R., John, M., Andres, M. H., & Natalie, R. (2009). Scratch: Programming for All. *Communications of the ACM*, *52*(11).

Piech, C., Bassen, J., Huang, J., Ganguli, S., Sahami, M., Guibas, L. J., & Sohl-Dickstein, J. (2015). Deep knowledge tracing. In *Advances in Neural Information Processing Systems* (pp. 505–513).

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 153–160).Wang, L., Sy, A., Liu, L., & Piech, C. (2017). Deep Knowledge Tracing On Programming Exercises. In *Proceedings of the 4th ACM Conference on Learning@ Scale* (pp. 201–204).

Xiong, X., Zhao, S., Van Inwegen, E., & Beck, J. (2016). Going Deeper with Deep Knowledge Tracing. In *Proceeding of 2016 International Conference on Educational Data Mining* (pp. 545–550).

Zhang, K., & Shasha, D. (1989). Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal on Computing*, *18*(6), 1245–1262.