Investigate Effectiveness of Code Features in Knowledge Tracing Task on Novice Programming Course.

Poorvaja Penmetsa, Yang Shi, Thomas Price
North Carolina State University
ppenmet@ncsu.edu, yshi26@ncsu.edu, twprice@ncsu.edu

ABSTRACT

Predicting student performance has been a major task in student modeling. Specifically, in open-ended domains such as computer science classes, the student submissions contain more information, however they also require more advanced analysis methods to extract this information. Traditional student modeling approaches use knowledge components (KCs) to predict a student's success on specific practiced skills. These approaches are useful and necessary in helping learning environments like Intelligent Tutoring Systems (ITS) personalize feedback, hints, and identify struggling students. However, when working with programming data, code features provide more information than skill tags representing KCs, and this information is not leveraged by traditional KC models. This work incorporates an implicit representation of KCs into a student model by including features extracted from students' code with data from an undergraduate introductory programming course. This representation is then evaluated by using deep learning predictive models and investigated to see how well they are able to leverage code features to model student knowledge and compare and contrast against other learning models. The study shows a modest, but consistent improvement in models that use time-sequential data with even the simplest code features, implying that these aspects may improve student modelling.

Keywords

Student Modeling, Code Features, Knowledge Tracing, Knowledge Components, LSTM, DKT

1. INTRODUCTION

Modeling student learning activities can improve their learning outcome, and automatically achieving this requires the system to model student's knowledge through tracking Knowledge Components (KCs) [2]. A knowledge component is an acquired concept that a learner uses to accomplish a task [8]. An exercise problem may consist of one or a combination of interrelated KCs depending on the domain, while multiple KCs are usually present in computer science problems involving programming [7]. Traditional KC models developed by domain experts used KCs to represent a student's knowledge state and are dependent on experts' knowledge [5].

Knowledge Tracing (KT) tracks KCs to model students' knowledge. It is a method of using data from their previous submissions on different problems (features), to predict on their performances on future problems (labels) [3]. Historically, student performance is defined as the academic performance of a student and KCs are represented with skill IDs [11, 6]. Our work builds data-driven KT models in computer science (CS) classes, and uses preliminary features from student code submissions to further tune and optimize standard KT models, and make predictions on student struggle.

Previous work has used data-driven methods to tune the parameters of a KT model, for example, Corbett and Anderson developed LISP tutors [6]. In their work, they used a model called Bayesian Knowledge Tracing (BKT) to monitor a student's changing knowledge state during programming assignments, and updated an estimate of the student's learned skills and modelled their knowledge state as students completed problems. Some recent works optimized the implementations of BKT, for example, pyBKT [4] uses expectation maximization to fit parameters.

This standard BKT model has been advanced by a lot of newer models such as Deep Knowledge Tracing (DKT) [12], Item Difficulty Effect Model (KT-IDEM) [11], etc., as they incorporate many different factors or use more complex models structure. Out of these improved models, we use DKT to serve as a model for our proof of concept, as it is widely served as a baseline model, and can also integrate student code to be used as features. Some recent works advanced DKT by adding more variants in the model [9, 10], while DKT is their base framework.

We compare DKT and BKT models in our work, along with classical deep learning models such as multi-layer perceptron (MLP) and Long-Short Term Memory (LSTM) to see if code features improve the ability of KT models, and if it also has a positive effect on static or non-time-sequential models. While skill IDs are hardly available for BKT without manual labeling, we use the problem or assignment IDs to serve as the KCs there [15, 12]. In DKT's case, the classical model can already use the IDs as inputs, automatically extracting knowledge components to be tracked [12].

While compared in our work, these standard KT models also have two important limitations when adding code features in: 1) **Need extra accommodation** standard KT models use correctness of KCs as features, while they do not accept code features with the original structure. There are some recent works adding textual information in KT models [15, 14], but they work with non-programming data. Some other works such as PKT [16] and Code DKT [17] use programming data, but they use an hour-long programming code database (Hour-of-Code) rather than a university course. 2) **Definition of Student Performance:** they usually define the binary score as the student's academic performance. However, students usually have a biased success rate in CS programming classes (in our case, $> 84\%$), as they are allowed to submit multiple times and receive feedback immediately. Predicting on whether students successfully finished the problem is less meaningful than directly detecting their struggles, as they may spend a lot of effort before reaching the goals.

The usage of code features, however, is not rare in general CS education domain or software engineering domain. Recent works have used sub-trees [19], Bag of Words [1], and automatically learned representations [13] for student progress modeling, grade prediction, or bug discovery tasks. Their models cannot be directly applied to sequential tasks such as KT. In our work, we use term frequency-inverse document frequency (TF-IDF) as the base features for this exploratory study.

We build preliminary KT models to predict whether novice university CS students struggle with the integration of code features. To verify whether code features can improve the KT models, we experiment with term frequency-inverse document frequency (TF-IDF), simply to show possible effects of the usage of code features in this task, and compare with other baseline models. To address the aforementioned limitations of classical KT models, we define our research question as: *Do code features improve the ability to model student struggle pattern in CS in time-sequential and non-time-sequential KT models?* Our results suggest that simple code features coupled with time sequential information have the potential to improve the KT model performance on student struggle prediction tasks.

2. METHODS

2.1 Data

The data used in this work is called CodeWorkout[1] and is from an undergraduate programming course at a large, public university in the southeast United States, collected in Spring 2019. The dataset includes code submissions from 5 assignments, 50 problems, and 413 students, recording students' attempts in implementing functions in Java. Each assignment contains 10 problems. A student could submit any number of attempts for a problem, each attempt's code is tested against given test cases and given a score from 0 to 1 based on how the code tested. Our statistics show that not all 413 students work on every problem and most students eventually get a full score (score = 1). Assignments are ordered by time stamp, and problems are grouped in each assignment. Different assignments covered different concepts, according to their problem descriptions, and the complexity of problems increases over time. The dataset also provides the descriptions of the problems. For example, early problems include simple concepts such as `if` conditions, while a later problem in the same assignment may ask about `nested-if` conditions.

Each assignment is a combination of multiple different concepts, with at least one new concept or advanced concept being introduced for each assignment. The entire course covers a wide range of introductory concepts including conditional statements, loops, strings, and arrays, which suggests that the knowledge tracing task across all problems is a non-trivial task.

The traditional goal of KT is to model knowledge to predict student performance. Historically, this has been interpreted as whether or not a student gets a problem correct. However, in many programming classes, including the one analyzed here, most students (at least 84% of students in this dataset) got a full score in the problems they attempt. So, it is more important to predict student struggle instead of success. In this programming setting, students can use multiple attempts and finally succeed. This allows us to use their number of attempts as a quantitative measure for their struggle. We define a struggling student as someone who uses more attempts than at least $T\%$ of the students. In order to find the threshold for T, we explored the data to find the division of two relatively separate groups of students corresponding to those who required a lot of effort to complete the problem, and those who didn't struggle. We found that the 75th percentile of attempts for each problem is a good division of attempts for most problems. A student is identified as struggling on the problem if they: 1) didn't pass all the test cases or 2) passed all test cases but took more than the 75th percentile of attempts for that particular problem. So a student who scored 1 on a problem might have also struggled with that problem. The prediction that a student is struggling is the positive class which is also the minority class.

2.2 Feature Representations

2.2.1 TF-IDF Features

We use simple code features in the form of Term Frequency-Inverse Document Frequency (TF-IDF) weights, a text analysis technique that represents documents based on its term frequency. The algorithm uses Term Frequency (TF) to find the frequency of a word in a document and normalizes these weights with Inverse Document Frequency (IDF), the frequency of the word in the corpus. The final weights reflect the importance of terms for a particular document.

This method is more commonly found in work involving processing text rather than code. However, the novice programming concepts present in this data can be partially represented by keywords since they are focused on familiarizing the students with syntax. More complex concepts, such as stacks and queues are not part of the course curriculum. The vocabulary used in our work consists of keywords in Java such as 'for', 'if', and 'public' that represent some novice concepts a beginner programming course might include. Other keywords such as 'throw' and 'extends' are also

[1]https://pslcdatashop.web.cmu.edu/Project?id=585

included to identify students who might have had a background in programming. These students may not struggle in the problems in this course compared with other students.

The models in this experiment predict at the problem level, so each problem done by a student is represented as one input. We define a corpus as the attempts across all students who worked on problem $p-1$ and use the TF-IDF vector of the last attempt a student made on this problem as code feature input. In this way, the frequencies of keywords for each attempt in a problem are calculated and the TF-IDF weights for the keywords in a student's best attempt are included in the code features.

Although other, more complex extraction methods such as code2vec, ASTNN, and pq-grams exist that extract deeper code and structural information, we use TF-IDF for this exploratory study. Our goal is not to use complicated vectorization approaches, but rather to examine whether a simple yet effective approach can represent information before moving on to more complex approaches. TF-IDF fits this goal because it not only simplifies model architecture, but it puts more weight on relatively rarer keywords and may help in identifying concepts in a problem.

2.2.2 *Feature Groups and Usage*

This experiment uses 4 sets of models: 1.) Standard BKT model [4], 2.) DKT models [12] with and without code features, 3.) MLP based models with and without code features, and 4.) LSTM-based models with and without code features. Each model throughout the experiment uses student struggle on past problems (student performance on problems 1 to $p-1$) along with other features to predict on the labels, or future student performance on problem p. We extracted the following features:

ID-Label Group: The Problem ID, Assignment ID, Uniform ID, and the binary label of whether the student struggles in problems 1 to $p-1$. Uniform ID groups employs the same IDs for all problems, with one ID that represents all KCs and only the label changes.

Attempt-Score Group: The number of attempts the student made (with any score) and the maximum score achieved on unit tests on any attempt for problems 1 to $p-1$.

Code Feature Group: The TF-IDF features of the last attempt at the problems 1 to $p-1$ (TF-IDF weights).

Because of the lack of IDs directly representing KCs in the data, we experiment with three different level of IDs (Problem ID, Assignment ID, Uniform ID). The BKT model uses the Uniform ID along with the labels of past struggle, assuming that one ID represents all the KCs in the dataset. The DKT set of models are compared using features across the entire ID-Label group.

The Attempt-Score group is used in the MLP and LSTM based models. Because of these features (Attempts and Score), these models are named Attempt-Score MLP (AS-MLP) and Attempt-Score LSTM (AS-LSTM). The Code Feature group is used for MLP, DKT, and LSTM models. We refer all models that don't use TF-IDF features as baseline models.

2.3 Models

2.3.1 *BKT*

We used the pyBKT implementation of the standard BKT model [4]. This model tracks student knowledge with the probability that a student has learned a skill. The output of one state is directly used as the input to the next state, making it a time-sequential model. For this reason, it requires inputs to be ordered sequentially. The purpose of this model in this experiment is to use a standard KT model on the current data. The input to pyBKT is two dimensional, including Uniform ID and the label for problem $p-1$.

2.3.2 *DKT*

We used the classical DKT implementation used in the Github repository[2], as the standard DKT model with Problem, Assignment, and Uniform ID as mentioned in the section above. We added the TF-IDF features to these models to create DKT with TF-IDF features, and compared those without TF-IDF features. Our DKT uses an embedding layer to initialize a random vector representing each input feature and updates this vector as the model trains. This is followed by an LSTM and a linear layer. The number of embeddings is one more than twice the number of questions (50) and the embedding dimension for D02 is 20, while D01 and D03 is 200. The batch size is 192 with a sequence size of 50, and learning rate of 0.001. The models with code features take in two inputs, the TF-IDF weights from code and the ID associated with the code. DKT with Problem ID and TF-IDF features (D04) and DKT with Uniform ID and TF-IDF features (D05) use a batch size of 192, a sequence size of 50, and a learning rate of 0.00001.

The general DKT model is designed to process time series data (i.e. multiple problem attempts over time) and uses LSTM networks which use a recurrent structure to serve the target. When predicting the label for student s and problem p, the input sequence includes the features for problems 1 to $p-1$ done by the student s. We used a fixed input sequence length of 50, the maximum number of problems a student can do, and a padding of 0's when needed. The padding makes sure that the input is always a fixed length. In addition to ID and/or TF-IDF features, the DKT models require student representation to measure the student performance, in this case, student struggle. We refer to this representation as the struggle feature where the model identifies the student's struggle on past problems. In both DKT and BKT sets of models, this struggle feature is binary because it represents whether or not a student struggled in the past problems.

In the DKT models without code features, models incorporate the struggle feature for problems 1 to $p-1$ to predict for problem p. There are three of these models with problem ID (D01), uniform ID (D02), or assignment ID (D03). Another two DKT models use code features. Besides the code features, one uses problem ID, and another one uses uniform ID. Both models, D04 and D05, combine TF-IDF vectors (length 63) with the embedded vector of ID and

[2]https://github.com/seewoo5/KT

struggle (length 20) resulting in the LSTM input dimension $63 + 20 = 83$.

2.3.3 Multilayer Perceptron Models

Attempt and Score MLP (AS-MLP) models with (M02) and without code features (M01) do not take in a sequential input and do not explicitly use the output of one state as input to the next, so they are not time-sequential. However, neural network structure allows MLP to incorporate code features. So, the purpose of these models is to compare the effect of performance features and code features in a static model on this struggle prediction task. Both AS-MLP models share the same architecture, with three linear layers. M01 uses a batch size of 260, and a learning rate of 0.001 and the AS-MLP model with code features (M02) uses the same batch size with a learning rate of 0.0001.

The inputs for the Attempt-Score-MLP models are less complex than that of any LSTM. Unlike the DKT input structure, the AS-MLP models don't use an empty sequence to predict for the 1st problem a student has attempted. In the AS-MLP model without code features (M01), the input size is two dimensional, including attempts and maximum score of problem $p-1$, while the other MLP model with code features has an input dimension of $63 + 2$ where 63 is the fixed length of TF-IDF code features.

2.3.4 LSTM Models

We created two Attempt-Score-LSTM (AS-LSTM) models: AS-LSTM with (L02) and without TF-IDF features (L01). These models incorporate performance features (attempt and score) compared to DKT's struggle feature. AS-LSTMs' input is similar to the DKT input, except that AS-LSTM doesn't predict on the 1st problem. In L01 both features represent the student performance on past problems and in L02, these performance features are appended to the TF-IDF features. So L01 also inputs a two dimensional array, while L02 inputs a vector sized $63+2$. Both AS-LSTM models share the same architecture of one LSTM layer followed by a linear layer. L01 uses a learning rate of 0.001 and a batch size of 260 over 100 epochs. L02 uses a learning rate of 0.000001 and batch size 260 over 250 epochs.

We used 8:2 split for data, and performed repeated resampling for AS-MLP, AS-LSTM models, while we use 5 fold cross validation for DKT and BKT models, reporting the average AUC/Recall/Precision and F1 scores in the result section.

3. RESULTS AND DISCUSSION

TF-IDF features and time-sequential information. Table 1 shows three pairs of models: D01 and D04, D02 and D05, M01 and M02, and L01 and L02 which are all identical except that the latter models (D04, D05, M02, and L02) include TF-IDF features. Before TF-IDF, the DKT models are biased towards label 1, while after adding TF-IDF features, the models are biased towards label 0. So we mainly use AUC, which is more symmetric to bias towards any class, to compare these models' performances in this section.

The results in Table 1 show how D04, D05, and L02 can distinguish between classes slightly better than D01, D02, or L01. L02 performs slightly better than D04 and D05 maybe due to its non-binary struggle feature. By looking at the confusion matrices of D04 and D01 in Table 2, it can be calculated that with TF-IDF, the DKT model is 6.2 times more likely to mark an at-risk student as needing help than a non-at-risk student by examining the ratio of $\frac{TP}{FP} : \frac{FN}{TN}$. Without TF-IDF, it's only almost twice as likely. Similarly, the confusion matrices of D02 and D05 show that D05 is 5.5 times more likely to mark an at-risk student as needing help while D02 is 3.9 times as likely. Meanwhile, L01 and L02 have a smaller difference with L01 at about 4.5 times likely to mark an at-risk student as needing help while L02 is about 4.7 times as likely.

While it has been an open question whether the addition of code features would do so, in our experiments, both L02 and D05 have a range of 3% to 6% improvement in their AUC. For example, Wang et. al. found that a large number of code features can lead models to overfit to the training data [18]. This has likely happened with D05 and L02, their LSTM input dimensions increased at least 4-fold from D02 and L01 respectively, making it possible that the models would overfit. However, it also shows that the AUC of D05 and L02 still increased, suggesting the models benefited from the additional complexity.

For AS-MLP models, M02 did not improve with TF-IDF features and performed similar to M01 as shown in Table 1. This suggest that time-sequential information is important for KT tasks and that TF-IDF features alone cannot improve model performance.

The current model performances are not ready for implementation in real education settings. This modest performance was predicted because examining the effect of TF-IDF features in modeling student knowledge is a very difficult task due to the open-ended nature of the domain and the wide range of concepts in the dataset. Although the performances of models with TF-IDF features were modest, there is still a 3% to 6% increase in AUC of these models compared to other baselines. This improvement marks potential for more complex code features.

3.1 Discussion

Difficulty of Knowledge Tracing for Programming: As hypothesized, this is a difficult KT task because of the wide range of concepts covered throughout the course. As the students progress through the different problems and assignments, new concepts are introduced which the models try to infer from either through an ID or student code features. Results across all models in Table 1 show that neither IDs nor code features are particularly successful in representing or inferring KCs. Of all the models in the experiment, BKT is the worst performing model with an AUC of 0.64 which suggests that this particular definition for student performance (labels) may not be the best fit with a standard BKT model.

Because there is a lack of tags to directly represent KCs, we use naive ways to represent them in the 3 DKT models. D01 treats each *problem* as a separate concept, D02 treats each *assignment* as a separate concept, and D03 treats all

Models	Precision (1)	Recall (1)	F1 Score (1)	Macro F1 score	AUC	ACC
BKT Uniform ID	0.57 ± 0.00	0.46 ± 0.01	0.51 ± 0.00	0.65 ± 0.00	0.64 ± 0.00	0.70 ± 0.00
MLP No TFIDF (M01)	0.35 ± 0.00	0.83 ± 0.01	0.50 ± 0.00	0.54 ± 0.00	0.70 ± 0.00	0.56 ± 0.00
MLP TFIDF (M02)	0.35 ± 0.01	0.80 ± 0.03	0.47 ± 0.00	0.55 ± 0.02	0.69 ± 0.02	0.56 ± 0.00
DKT Problem ID (D01)	0.37 ± 0.02	**0.89 ± 0.01**	0.52 ± 0.02	0.45 ± 0.05	0.71 ± 0.00	0.46 ± 0.04
DKT Uniform ID (D02)	0.42 ± 0.01	0.82 ± 0.01	**0.56 ± 0.00**	0.57 ± 0.02	0.71 ± 0.00	0.57 ±0.02
DKT Assignment ID (D03)	0.40 ± 0.01	0.88 ± 0.00	0.55 ± 0.00	0.51 ± 0.01	0.72 ± 0.00	0.51 ± 0.01
DKT TF-IDF & Problem ID (D04)	0.55 ± 0.01	0.33 ± 0.02	0.41 ± 0.01	0.63 ± 0.00	0.73 ± 0.00	**0.76 ± 0.00**
DKT TF-IDF & Uniform ID (D05)	**0.60 ± 0.02**	0.34 ± 0.02	0.44 ± 0.02	0.64 ± 0.00	0.75 ± 0.00	0.77 ± 0.00
LSTM No TF-IDF (L01)	0.50 ± 0.00	0.52 ± 0.07	0.50 ± 0.05	**0.69 ± 0.02**	0.74 ± 0.00	0.73 ± 0.03
LSTM TF-IDF (L02)	0.42± 0.03	0.74 ± 0.10	0.52 ± 0.00	0.62 ± 0.04	**0.76 ± 0.00**	0.64 ± 0.00

Table 1: Results of Models with or without Code features. Bold figures suggest best in the corresponding metric.

D04		D01	
TP = 301	FN = 575	TP = 398	FN = 454
FP = 238	TP = 2224	FP = 395	TP = 2008

Table 2: DKT Problem ID without (D01) and with (D04) TF-IDF features confusion matrices.

L02		L01	
TP = 541	FN = 311	TP = 398	FN = 454
FP = 618	TP = 1785	FP = 395	TP = 2008

Table 3: AS-LSTM without (L01) and with (L02) TF-IDF features confusion matrices.

concepts the same. As shown in Table 1, they all are biased towards label 1 (struggling student). Further examinations into D01 - D03 from Table 1 show that that they perform similarly, with D03 having the best AUC and D02 with the highest macro F1 score. Because both IDs have similar results, this suggests that neither is better at representing concepts.

Why were DKT and the other KT approaches unsuccessful?: The original DKT work that the D01 code is based on states that the algorithm can leverage skill or KC tags but doesn't need them to perform predictions [12]. Looking at the results in Table 1, it is clear that the D01 model which uses problem ID as features and incorporates the number of attempts in the labels, has mediocre results. The only difference between the D01 model and the original DKT model lies in the definition of student performance. However, there are other differences in the data used. The original DKT work used non-programming data where the problem ids represented the skills or KCs better. The original work also uses far more data from online courses than available from a formal education setting, so there is more data representing each KC. Where the smallest dataset in [12] has 4K students and 200K entries, the data used here has 400 students and 16K entries (after cleaning). The original work used each attempt as an entry and in this work, the last attempt is used. Moreover, students who worked on CodeWorkout did not all do problems in the same order for an assignment. This makes it very difficult to predict the feature importance of a problem based on past problems for all students.

D05		D02	
TP = 298	FN = 578	TP = 888	FN = 220
FP = 211	TP = 2251	FP = 1124	TP = 1106

Table 4: DKT Uniform ID without (D02) and with (D05) TF-IDF features confusion matrices.

Our results suggest that there is some improvement, but only for models that use time-sequential information. There is a 3% - 6% increase in the AUC for LSTM-based models that use TF-IDF features when compared with their counterparts that don't use them. Considering the difficulty of this task, this improvement suggests that more complex models may improve performance more.

When code features are added to static models such as AS-MLP (M02), there is no improvement in performance. Comparing M02 to the final time-sequential models, (D04, D05, L02), there is a 7% to 10% increase in AUC suggesting that in the presence of time-series data, the impact of code features is more. However, only one non-time-sequential model is implemented with code features, so it is not clear if these results are robust.

4. LIMITATIONS AND FUTURE WORK

One limitation in this work is that the vocabulary used for TF-IDF may not be generalizable to more advanced curricula. The second limitation is that we only use one type of non-time-sequential and one type of time-sequential models to compare code features. So the results may be specific to LSTM or MLP based models. In the future, we plan to work with expert-based features like PQ-grams and incorporate other models into our experiment.

In conclusion, in this work, we experimented with models with and without both simple code features and time-sequential information. The results show that even these simple code features can affect model performance and that time-sequential information is important when using these features. These experiments mark the potential that code features have in representing CS KCs over an entire course.

5. REFERENCES

[1] B. Akram et al. Assessment of students' computer science focal knowledge, skills, and abilities in game-based learning environments. 2019.

[2] V. Aleven. *Rule-Based Cognitive Modeling for Intelligent Tutoring Systems*, pages 33–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[3] J. Anderson, A. Corbett, K. Koedinger, and R. Pelletier. Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4:167–207, 04 1995.

[4] A. Badrinath, F. Wang, and Z. Pardos. pybkt: An accessible python library of bayesian knowledge tracing models, 2021.

[5] R. Clark, D. Feldon, J. J. G. Van Merrienboer, K. Yates, and S. Early. Cognitive task analysis. *Handbook of Research on Educational Communications and Technology*, pages 577–593, 01 2008.

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

[7] B. Haberman and O. Muller. Teaching abstraction to novices: Pattern-based and adt-based problem-solving processes. In *2008 38th Annual Frontiers in Education Conference*, pages F1C–7. IEEE, 2008.

[8] K. Koedinger, A. Corbett, and C. Perfetti. The knowledge-learning-instruction (kli) framework: Toward bridging the science-practice chasm to enhance robust student learning. *Cognitive science*, 36:757–98, 04 2012.

[9] S. Pandey and G. Karypis. A self-attentive model for knowledge tracing. *CoRR*, abs/1907.06837, 2019.

[10] S. Pandey and J. Srivastava. Rkt: Relation-aware self-attention for knowledge tracing. *Proceedings of the 29th ACM International Conference on Information Knowledge Management*, Oct 2020.

[11] Z. Pardos and N. Heffernan. Kt-idem: Introducing item difficulty to the knowledge tracing model. pages 243–254, 01 1970.

[12] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein. Deep knowledge tracing. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[13] 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.

[14] D. Shin, Y. Shim, H. Yu, S. Lee, B. Kim, and Y. Choi. SAINT+: integrating temporal features for ednet correctness prediction. *CoRR*, abs/2010.12042, 2020.

[15] Y. Su, Q. Liu, Q. Liu, Z. Huang, Y. Yin, E. Chen, C. H. Q. Ding, S. Wei, and G. Hu. Exercise-enhanced sequential modeling for student performance prediction. In *AAAI*, pages 2435–2443, 2018.

[16] L. Wang, A. Sy, L. Liu, and C. Piech. Deep knowledge tracing on programming exercises. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, page 201–204, New York, NY, USA, 2017. Association for Computing Machinery.

[17] L. Wang, A. Sy, L. Liu, and C. Piech. Deep knowledge tracing on programming exercises. pages 201–204. Association for Computing Machinery, 2017.

[18] W. Wang, Y. Rao, Y. Shi, A. Milliken, C. Martens, T. Barnes, and T. Price. Comparing feature engineering approaches to predict complex programming behaviors. 5 2020.

[19] R. Zhi, T. Price, N. Lytle, Y. Dong, and T. Barnes. Reducing the state space of programming problems through data-driven feature detection. 07 2018.