Evaluating Multi-Knowledge Component Interpretability of Deep Knowledge Tracing Models in Programming

Yang Shi*, Tiffany Barnes, Min Chi, Thomas Price
NC State University
{yshi26,tmbarnes,mchi,twprice}@ncsu.edu

ABSTRACT

Knowledge tracing (KT) models have been a commonly used tool for tracking students' knowledge status. Recent advances in deep knowledge tracing (DKT) have demonstrated increased performance for knowledge tracing tasks in many datasets. However, interpreting students' states on single knowledge components (KCs) from DKT models could be challenging when tracking multiple KCs in one student submission attempt. In this paper, we evaluate the ability of DKT models to track students' knowledge using AUC scores. We further propose two possible solutions to improve multi-KC tracking performance: incorporating a layer that explicitly represents knowledge of each KC and incorporating code features into the DKT models. In experiments, we compare DKT to the proposed models and evaluate KC tracking performance in an introductory computer science course (CS1) dataset. Our results indicate that while all four models perform similarly on problem correctness predictions, incorporating KC layers may lead to limited improvement for KC tracking performance. Through a hand-labeled dataset with KC-specific correctness, our research shows that DKT has a limited performance when tracking multiple skills, especially when tracking incorrect submissions. We present potential ways, including designing a layer or incorporating student code information in the models, while the results show that only the layer yielded improvements.

Keywords

Student Modeling, Knowledge Component, KC, Knowledge Tracing, KT, DKT, Deep Knowledge Tracing, Computing Education, CSEd, CS1, interpertability

1. INTRODUCTION AND BACKGROUND

Many knowledge tracing models (e.g., BKT or DKT, [3, 11]) are built to model students' KCs (or skills; see [9]). However, as [19] note, these models are designed for problems broken down into steps that practice just one skill at a time, allowing each skill to be modeled somewhat independently. It is more challenging to model problems that practice multiple skills simultaneously, such as those found in many programming practice contexts. For example, in the problem shown in Figure 1, the student must simultaneously check if a variable lies between two constants (one KC) while also sequencing conditionals correctly (a separate KC). If a student gets this problem right, this serves as some evidence that they understand all of the relevant concepts; however, if a student gets this problem incorrect, it is unclear which of these concepts they need more practice on and which they may have mastered. For example, Figure 1 shows two hypothetical, incorrect student answers to this problem, with two very different errors, which should in theory lead to two very different updates to the student model; however, existing knowledge tracing approaches generally do not differentiate these responses, simply labeling them both as "incorrect." This causes possible issues when the goal is to track students' knowledge on each KC (also referred to as the student's "KC state" in this work).

```
if (age <= 20 || age >= 10) {
    Condition A;
} else {
    Condition B;
}
```

```
if (age <= 20 && age >= 10) {
    Condition B;
} else {
    Condition A;
}
```

Figure 1: Examples of two incorrect submissions with different KCs incorrectly demonstrated.

The need to model learning on problems where multiple skills are practiced simultaneously has led to a number of Bayesian models that extend BKT (e.g., [20, 8, 13]). For example, in [20], the authors used a linear regression method to model multiple skills at a time. In the work of [8], the authors proposed a method to assign "fair blame" to each KC on incorrect problem predictions, using weights derived from prior submissions from students to determine which KC(s) were at fault. While the work addresses our proposed issue, they achieve this in conjunctive BKT models, and it is not clear how to extend the work to modern DKT models or interpretations of their layers. Many recent KT models leverage deep learning methods to achieve better performance [6]. However, little work has explored how to track multiple skills at the same time. Khajah et al. commented that DKT's advantage of performance comes with a cost of interpretability, and specifically designed BKT models can result in similar accuracy with DKT [7]. Prior works suggest

*Yang Shi is also affiliated with Utah State University.

Y. Shi, M. Chi, T. Barnes, and T. Price. Evaluating multi-knowledge component interpretability of deep knowledge tracing models in programming. In B. Paaßen and C. D. Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 288–295, Atlanta, Georgia, USA, July 2024. International Educational Data Mining Society.

https://doi.org/10.5281/zenodo.12729818

the need for building deep KT models that can effectively model multiple skills at the same time, and, perhaps more importantly, finding ways to *evaluate* how well these models perform.

Building and evaluating deep knowledge tracing models to track multiple skills at the same time presents a few major challenges. First, the predictions of such models are usually next-submission correctness, and additional steps are needed to map these correctness predictions to KC values or skills. Unlike traditional Bayesian KT models, deep models are mostly used to predict problem *correctness* [11, 10, 15] (i.e., there is a 70% chance the student will get Problem 3 correct) instead of modeling knowledge of specific KCs (i.e., there is a 70% chance the student has mastered the skill of sequencing conditionals properly). While Q-matrices [1] provide information about which KCs are practiced in which problems, the models are not designed to incorporate this information. This may lead to a conceptual gap between problem correctness and KC labels in the training of models. Furthermore, mapping these correctness predictions to estimates of skill mastery becomes more complicated when problems practice multiple interdependent skills. The second challenge is that deep models often lack interpretability. Deep KT models have long been criticized for their lack of interpretability [14]. This is partly because the *structure* of deep models is much more complex than Bayesian models and not directly mapped to learning theory. However, in computing education, prior research observed much better modeling results from DKT than BKT models, showing the computational complexity of deep learning models helped the performance [15]. Finally, there's also an opportunity to solve these challenges by incorporating domain-specific information into the deep learning models, such as programming code [22], as it has been showing improvements on correctness prediction tasks (e.g., [15]).

In this paper, to address the first challenge, we use a method to infer KCs from DKT predictions and evaluate the performance of DKT models in tracking multiple KCs. Rather than evaluating the models only on their ability to predict the correctness of a problem attempt, we also evaluated them on their ability to predict whether a student demonstrated a particular KC correctly on that problem (regardless of whether the whole problem was correct). In order to do so, we had experts label student submissions and used adjusted AUC scores towards the goal of tracking the KCs on incorrect submissions from problems practicing multiple KCs. We further compare the correctness predictions to the KC predictions to evaluate whether DKT models can track multiple KCs at the same time, using the prediction-inferred KCs. The challenge of mapping correctness to skill mastery can be addressed by modifying the DKT models. We introduce a possible way to improve the interpretation of the DKT models by introducing a KC layer – a variant of DKT models named KCDKT that incorporates Q-matrix information as a layer into the DKT models. The layer is placed between the recurrent structure of the DKT models and the correctness prediction layer to learn KC values and use the KC values for correctness prediction. Because DKT is only trained on correctness labels, and we are measuring its ability to predict successful demonstration of individual KCs, the model may struggle. However, other features, such as the program code students write may have useful information for making these predictions. Therefore, we also explore the model performance if code features are used in the models, creating the code version of KCDKT (CodeKCDKT) and we evaluate the correctness and KC predictive performance of the models. BKT models are not compared in our experiments due to prior low performance in relevant research [15], and our exploration in this paper is focused on DKT models. Specifically, we answer the three research questions:

RQ1: How well does the original DKT model students' mastery of individual KCs when practicing on multiple-KC problems?

RQ2: How does adding an explicit layer to model KC mastery to DKT affect its performance on this task?

RQ3: How does adding programming code to DKT affect its performance?

2. METHOD

In this section, to facilitate the understanding of the method, we first introduce the measurements of KC mastery in this paper, then introduce the models to be evaluated: how code features are incorporated in DKT and how the KC layers are incorporated.

2.1 Measuring KC Mastery Modeling

While prior work has explored how to improve the *predictive performance* of models on multi-KC problems [8, 20], our goal is to more directly measure a model's ability to predict a student's mastery of individual KCs. While this mastery itself is unobservable, according to the KLI Framework [9], it should correspond closely with student performance on problems where those KCs are practiced. The fact that students may practice *multiple KCs* on each problem further complicates this challenge. The assumption is that when a student correctly submits an attempt at a problem, this is evidence of mastery of *all* KCs practiced in that problem. However, we cannot make these assumptions when a student gets a problem *incorrect*. The student may have demonstrated mastery of one KC (e.g. creating a correct conditional structure), but failed to demonstrate another (e.g. constructing the boolean expression in the if-statement). Alternatively, a student may have demonstrated mastery of both of these skills, but "slipped" by creating a syntax error and therefore got the problem wrong. Therefore, the only way to accurately label the demonstration of KC mastery on incorrect attempts is to do so manually. This is described in section 3.4. Once the labels are obtained, we can evaluate the model the same way KT models are normally evaluated; however, instead of using *correctness* labels, we use these expert-authored *KC-demonstration labels*. Now, instead of having one prediction instance per student per problem (counting only first-attempts, as is customary [3]), we have k, where k is the number of relevant KCs for that problem. We can otherwise evaluate the model's predictive performance using traditional metrics (e.g., AUC). Note that this evaluation approach gives extra weight to problems with more KCs (which produces more prediction instances), so it is also possible to weight the results to give each problem the same weight in the evaluation (e.g. a 2-KC problem

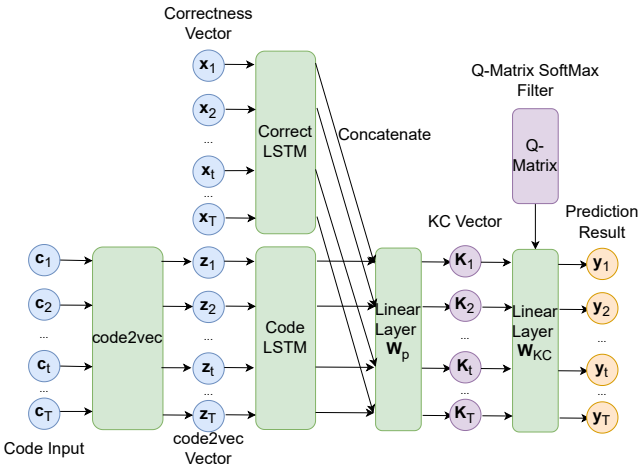

Figure 2: CodeDKT implementation with KC layer incorporated.

has 2 KC predictions, which are each given half the weight).

This evaluation of KC correctness is a key contribution of this work. It is different and more informative than evaluating the KC states of students on overall problem correctness for two main reasons. 1) While problem correctness prediction could provide useful information for teachers and students, for example, as feedback, the prediction on the KC level will be more informative. Correctness predictions can alert students when they start an attempt that is predicted to be incorrect. However, the prediction alone does not provide any information on *why* students may make a mistake. KC predictions will help students to locate the KCs that need more practice. 2) Good performance on predicting problem correctness does not necessarily mean that a model has a meaningful internal representation of KC mastery as latent variables. Deep learning models are famous for being black boxes [18], and they could learn different features (many of which many not be related to KCs). For example, it may capture students' usage of brackets (i.e., correct syntax) as a key feature for the prediction (as shown in the discussion of [16]). While this may be associated with students' coding behavior and ultimately may predict code correctness, it is not a KC we want to model for an assignment that focuses on conditionals.

2.2 Inferring KCs from DKT and CodeDKT Predictions

Assume that the prediction results of DKT [11]/CodeDKT [15] models for a student is a sequence of vectors $\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_T$, where $\mathbf{y}_t$ represents the predicted correctness of next submission $(t+1)$ for the student, we infer KC values from the $\mathbf{y}_t$ vectors (see Figure 2). When single KCs are practiced in each submission or practice step, the correctness predictions could directly represent the corresponding KCs through a Q-Matrix. However, in multi-KC scenarios, the predictive correctness of problems is not directly associated with the mastery of a single KC. Therefore, in our work, we infer the model's estimate of a student's mastery of a given KC, k, by averaging the model's predictions on all problems that practice the KC k, according to a Q-matrix. While these problems may also practice other KCs, our assumption is that the average across all such problems is a rough proxy for mastery of k. This assumption comes with certain limitations, as problems have convoluted and associated skills, making some problems harder and some easier.

2.3 KC Layer Design in DKT and CodeDKT

The prior approach has a number of limitations, as discussed above. Ideally, we should not have to infer a model's estimate of KC mastery from other outputs; we should be able to read it directly from this model. This would not only make the inference more direct, but it may also improve the model's ability to model these skills by incorporating the process in the model training process. We further introduce a layer specifically designed to represent KC states in the DKT models. Taking the CodeDKT model as an example, shown in Figure 2, we incorporate the KC layer $\mathbf{W}_{KC}$ before the final prediction of problem correctness. The input of the model has two parts, students' code submissions $\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_T$ and the corresponding correctness vectors $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_t$ of these submissions. Code features are separately processed by a code LSTM layer and concatenated with the correctness LSTM layer output. Then the merged vectors are processed by the KC layer which incorporates the Q-matrix. The incorporation is based on a hypothesis: KC states indicate students' probability of achieving relevant problems. We create a layer in the model that condenses its predictions down to I values where I is the number of KCs. The model then uses these KC knowledge estimates to predict student performance on all problems, where each problem prediction is a weighted sum of the relevant KCs for the problem, according to the Q-matrix. Assume we have KC vectors $\mathbf{K}_t$ at submission t, then the probability of students getting relevant problems correctly practiced is processed in the KC layer as

$$\mathbf{y}_t = \sigma(\mathbf{K}_t \mathbf{W}_{KC}), \tag{1}$$

where $\sigma(\cdot)$ represents the sigmoid function. We consider $\mathbf{W}_{KC} \in \mathbb{R}^{I \times N}$ as the weight of each of the KCs specific to problems, where I represents the number of KCs and N represents the number of problems. To learn KC states, the KC layer $\mathbf{W}_{KC}$ is designed to employ two properties: (1) on irrelevant problems, the weight is set to 0, and (2), the weight is normalized to a range of 0 to 1. I introduce the Q-matrix $\mathbf{Q} \in \{0,1\}^{I \times N}$ in this process for the filter purpose and use SoftMax to normalize among relevant KCs. Specifically, the KC layer weights are calculated as

$$\mathbf{W}_{KC} = normalize(\sigma(\mathbf{W}'_{KC}) \odot \mathbf{Q}), \tag{2}$$

where $\odot$ is an operator for element-wise multiplication, and $\mathbf{W}'$ denotes the original KC layer weight before applying the filtered SoftMax process. The goal is to use this layer to represent the relationship between KC states and predicted problem correctness. For example, if a problem has two relevant KCs, students' corresponding KC states should be used to calculate the probability of the submissions' correctness. The training of the model still follows DKT, where we use the next problem correctness ($\mathbf{y}$ sequences) to train the model but evaluate the interpreted KCs on the $\mathbf{K}$ vectors.

The way we chose to incorporate a Q-matrix into the KCDKT model is based on the defining properties of KCs. When stu-

dents practice KCs in problems, their knowledge of relevant KCs should indicate their probability of successfully achieving the problem. This relationship can be represented with a Q-matrix. In this project, the KC layer serves to achieve two goals. First, it allows the weights to be filtered by the Q-matrix. In the model forward pass, only relevant KCs are multiplied with a weight learned in the KC layer, while all irrelevant KCs are multiplied with zero. In the backward pass, only the gradients related to the relevant KCs are updated, forcing the designed dimensions of the recurrent layer output to be updated using information of relevant KCs. Our goal is to model KCs (skills), rather than misconceptions (which we leave for future work). Therefore, to make weights more interpretable, we constrained them to be positive. To do so, we filtered the weight of the KC layer to be positive, and the sum of the weights for a problem was normalized as one. The design of the KC layer serves the goal of incorporating the KC properties into the DKT model when tracking KCs.

3. EXPERIMENTS

3.1 Dataset Processing and Experiment Setup

In this work, we employ a public dataset from the 2nd CSEDM data challenge[1], which is collected from the Spring and Fall 2019 semesters at Virginia Tech in an introductory programming class. Students are required to practice their Java programming in the CodeWorkout online platform [4], and they receive test case feedback from the platform. During the analysis, no demographic information is made available for the researchers, nor does it bring ethical concerns. There are 410 students practicing in the dataset, and every submission averages less than 20 lines of code. Students practiced in five assignments, each comprising ten problems for students to practice certain topics. For example, we only use data from the first assignment, and the problems practice mainly concepts related to `if` conditions. In all submissions of the first assignment, 26.14% pass all test cases. We used the Q-matrix and KC labels introduced in Sections 3.4. In the raw data processing process, we followed the process described in recent work (e.g., [16]) to filter out potential cheating students in the dataset. Some students started trying multiple times, struggling with earlier and easier problems, but succeeded in one submission for every harder problem after a certain point. We filtered such students to avoid possible noises in the dataset with a threshold. Students who struggled first and achieved quickly in the more difficult problems later are filtered out and not analyzed in this work. After the filtering process, 351 students were analyzed in our experiments. For running the deep learning models, we used Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz and GeForce RTX 2080 Ti as the CPU and GPU for computing, with 64G RAM. The rough amount of time training a model is less than 10 minutes for one run.[2]

3.2 Hyperparameter Tuning

We fine-tuned two hyperparameters in the DKT and CodeDKT models, the epochs ep of running and the learning rate lr, and selected the best hyperparameters with grid search in 4-fold cross-validation. The grid search space is defined that: for the DKT and CodeDKT models, we search the space of $lr = \{0.0004, 0.0005, 0.0006, 0.0007\}$ and $ep = \{40, 60, 80, 100, 120\}$; for the KCDKT and CodeKCDKT models, the search space of hyperparameters are $lr = \{0.004, 0.006, 0.008, 0.01\}$ and $ep = \{80, 100, 120, 140, 160\}$. We explored the space of hyperparameters before the search and ensured that optimal correctness prediction performance are achieved through models trained in this space in cross-validation. The final hyperparameter we arrived at is $lr = 0.0007$, $ep = 80$ for DKT, $lr = 0.0004$, $ep = 40$ for CodeDKT, $lr = 0.01$, $ep = 100$ for DKT, $lr = 0.01$, $ep = 120$ for DKT. In testing, we ran each model 5 times to calculate and report the average result for correctness prediction and KC interpretation in the next section.

3.3 Evaluation Metrics

We evaluate models' predictive performance in AUC scores [2] and show the results in the first column of Table 1. The metric we use to evaluate the models' ability to track multiple KCs. The most direct evaluation would be to evaluate the AUC scores of KC values, as we have the KC labels for the testing dataset. We report the overall AUC scores for all KCs in the second column of the table. Besides the overall AUC scores, we further adjusted the weight of each of the KCs in the calculation of AUC scores, since for submissions with multiple KCs practiced, every practiced KCs are counted once for the AUC score. We proportionally weight the submissions of KCs such that every submission is equally considered in calculating AUC scores. The results of this adjustment are in the third column of Table 1. Finally, also calculated AUC for a subset of predictions, which fit two criteria we are specifically interested in: 1) Our ultimate goal is to help students when they need help, mainly when they make an incorrect submission. Therefore, the first criterion is to select incorrect submissions to evaluate the models' ability to track KCs. 2) The goal of multi-KC tracking also focuses on submissions practicing multiple KCs. This is because the correctness prediction represents KC directly for the single KC problems. We further filtered incorrect submissions practicing multiple KCs to evaluate AUC scores for the models. The results are in the last column of Table 1.

3.4 Q-matrix Definition and KC Labeling

The Q-matrix is defined by two experienced authors who have teaching experience in introductory programming classes for more than two semesters. We first familiarized ourselves with the dataset by checking the problem requirements and random incorrect submissions. Then, we summarized phrases to describe possible KCs practiced in the ten problems and determined which KCs were meaningful for students. For example, students need to know how to write `return` statements to generate the results for functions, as this is not a key practiced concept for the ten problems. We go through the ten problems and label the Q-matrix with a standard that no KCs are practiced in less than three problems or more than eight problems since KCs practiced in only two problems may be too specific, and KCs practiced in many problems may be too general. Too specific KCs would be irrelevant for this specific assignment (or the set of problems), and too general KCs are likely already practiced well in previous problems and now are frequently used. The definition of KCs are:

`Between`: Construct a boolean expression to determine if a

[1] https://sites.google.com/ncsu.edu/csedm-dc-2021/

[2] Code and data can be found in https://github.com/YangAzure/KC-Attribution-Tracking.

Table 1: Correctness and KC predictive performance of DKT models with/without incorporating KC layers. Results are presented in AUC score (standard deviation) +- standard error. Explanations of each column are detailed in Section 3.3.

Model	Correctness AUC	Relevant KC AUC	Adjusted Relevant KC AUC	Filtered multi-KC "Incorrect" AUC
CodeKCDKT	0.6853 (0.0121) +-0.0060	0.6478 (0.0095) +-0.0047	0.6625 (0.0132) +-0.0066	0.5148 (0.0215) +-0.0107
CodeDKT	0.7054 (0.0088) +-0.0044	0.6823 (0.0179) +-0.0089	0.6960 (0.0137) +-0.0068	0.5050 (0.0351) +-0.0175
KCDKT	0.6922 (0.0055) +-0.0027	0.7489 (0.0190) +-0.0095	0.7563 (0.0180) +-0.0090	0.5348 (0.0250) +-0.0125
DKT	0.7144 (0.0020) +-0.0010	0.7152 (0.0078) +-0.0039	0.7235 (0.0080) +-0.0040	0.5183 (0.0060) +-0.0030

variable is between (two) constants.

`N Way` Sequential Conditions: Order if-statements to reflect mutually exclusive outcomes based on a problem prompt.

`2xN`: Interacting Conditions: Nest if-statements (or create complex expressions in non-nested if-statements) to reflect outcomes at the intersection of 2 decisions.

After defining the Q-matrix, we continued to label the testing dataset of KCs. We sampled 48 students and labeled the errors to the KCs we defined in the Q-matrix. Two authors independently labeled ten students' submissions first and found that our labeling has been consistent with Cohen's Kappa value of 0.765, reaching excellent agreement [5]. Then, one author labeled all the rest students in the testing dataset.

4. RESULTS

4.1 Correctness Prediction

Interpretability doesn't matter if the models aren't predictive, so we need first to verify that they are all comparable to past data [15]. The results of correctness predictions are shown in the first column of Table 1. The results show that DKT and CodeDKT have slightly higher AUC scores on predictive performance compared to the KCDKT and CodeKCDKT models, while the correctness predictive performance of KCDKT and CodeKCDKT models with the incorporation of KC layers are only about 1 percentage point lower on AUC scores than the respective DKT models, showing that the four models perform similarly in predicting the next submission correctness on first submissions. The result shown here is lower than reported in [15], not only because we use a different implementation of the models but also because we filtered out all non-first submissions in the training data to keep consistent with the prior KT models, and because we only created KC labels for first attempts; this allowed us to use the same test dataset in Table 1. We also filtered out submissions from potential cheating students to reduce noise, as done in [16] as well.

4.2 KC Interpretation

The KC interpretation results in Table 1 show that the DKT model has a relatively good AUC score on interpreting KCs compared to CodeDKT and CodeKCDKT. Each metric in the table is explained in Section 3.3. The best performance comes from KCDKT, showing that the incorporation of KC layer to the DKT model may help it to track KCs better. However, the incorporation of code information does not seem to improve the KC prediction performance of the models across the two models using code features. With both the KC layer and code features leveraged in the model, CodeKCDKT achieves the lowest KC interpretation and correctness prediction performance, showing that incorporating both information won't help either correctness or KC predictions. The last column of this table shows that when only evaluating incorrect submissions practicing multiple KCs, all models experience low AUC scores. However, KCDKT is still the highest among the models. It shows that DKT models cannot distinguish the difference among KCs when multiple KCs are practiced in one submission, presenting a significant limitation of DKT models when tracking multiple KCs. One observation is that incorporating the KC layer still improves the performance by a little (1.55 percentage point on AUC score). However, none of the scores are enough to be considered to work. We do see that at least the improvement of the KC layer holds for the KCDKT compared with the DKT model, though it does not work well when the KC layer is incorporated into CodeDKT.

5. DISCUSSION

5.1 Research Questions

RQ1: KC from Correctness Predictions Inferring KC values from correctness predictions may work, but it may not. On the one hand, one might think that DKT shouldn't be able to predict KCs at all, since it's not given this information. We trained it on one set of labels and evaluated it on a different set. On the other hand, one might reasonably expect that correctness should be a byproduct of KCs, so predicting correctness might make for good KC predictions. What we find is that both are true in their own way. DKT is good at predicting KCs in some situations, and terrible in others. The AUC score of correctness prediction (0.714) is close to the adjusted relevant KC (0.723), meaning that the model is able to perform KC prediction as well as correctness prediction overall, however, we also see contradictive results. Two main differences exist between adjusted relevant KC AUC and correctness prediction AUC values. 1) The model predictions averages across problems (dictated by the Qmatrix) instead of single-problem predictions. 2) In the test data, some 0s were changed to 1s where the student got some or all of the relevant KCs correct. While the result could be due to either, these results at least suggest that, these changes in the test dataset overall did not harm DKT's performance. When comparing the KC AUC on incorrect submissions practicing multiple KCs (about a quarter of the total amount of testing data), the performance of DKT model, along with all other compared models, is much worse and close to 0.50. It means that when making KC predictions, DKT models do no better than random guessing when the submission is incorrect, and it practices multiple KCs. The results show that all models are not able to distinguish KCs practiced for multi-KC problems, despite the fact that they perform well in other scenarios. The DKT mod-

els work significantly worse on tracking multiple KCs when students make mistakes and unfortunately, it is the scenario when we need accurate KC tracing most to facilitate educational applications such as problem recommendation [21] and automated formative feedback [12]. This project directly evaluates the DKT models' ability to predict multiple KCs at the same time, and reveals the low performance of tracking multiple KCs. The results show that a model being good at predicting problem correctness *might* correlate with predicting when a student will get a problem *right* but not with predicting *why* a student got a problem *wrong*.

RQ2: The Incorporation of a KC Layer While none of our models had access to our KC demonstration labels, we do see some evidence (with caveats) that it may be possible to improve a model's ability to predict these labels by incorporating the Q-matrix into the model design. From Table 1, the KC interpretation performance of KCDKT is highest among all models. The result suggest that incorporating KC layers may serve as a method to improve the KC tracking performance on multiple skills. One further observation is that the layer does not lower the correct prediction performance much by itself. Compared with other DKT models, it has about 1 percentage point less AUC score on problem correctness prediction but brings at least 3.47 percentage point improvement on relevant KC AUC scores compared with other models. The tradeoff between correctness prediction and KC prediction pays when the goal is to track KCs instead of predicting student correctness. However, this result comes with an important caveat. When only comparing the performance on incorrect multi-KC submissions, even though KCDKT performs better than DKT, it still performs quite low (0.535 AUC) to the point where it clearly would not be helpful for predicting which KC is incorrect – one of the major goals of this evaluation. Additionally, when incorporating the KC layer into the CodeDKT model, the model KC prediction drops over 4 percentage points, showing that the improvement may not apply when more complicated features such as code are present. This may be because the KC variants condense the model's predictions down into just 3 activations (one per KC), before using them for problem performance prediction. For a more complex model like CodeDKT (with at least twice as many parameters), this condensing might be particularly harmful. This result could also simply be an indicator that the improvements are not robust, and may not work on all models and datasets.

RQ3: Incorporating Structural Code Features In order for a model to explain *why* a student got a problem wrong, and which KC(s) were responsible, it is reasonable to expect that having some representation of a student's code might help, as prior work suggests it can for performance prediction [15]. However, our experimental results show that incorporating code features may not improve (or may have a negative impact on) both correctness and KC predictions. This contradicts the results from [15], and there are several possible reasons behind this. One is that the implementation and the data processing procedures are different. In the model implemented in this project, we only included students' first attempts on every problem and filtered possible cheating students. The processing procedure follows the KT problem settings [3, 9], and the removal of possible student cheating is due to a closer look at the dataset. A lack of data in the training process may cause CodeDKT to have a low performance. Compared with [15], the training dataset in this work is significantly smaller, and the model may not be able to learn the features present in students' repeated submissions. Furthermore, when leveraging code features in KCDKT, the performance is worse compared with other models. The low performance of CodeDKT and CodeKCDKT may also be attributed to the overcomplicated structure of models, as the dataset for training is low. While our results show negative impacts for code-fused models in multi-KC tracking, future work may still explore the possibility of leveraging code features when more data is available.

5.2 Educational and Research Implications

The research implication of this work is twofold. This work contributes to a method for using expert labels of success at demonstrating particular KCs to evaluate a model's ability to predict those values, which hasn't been done in prior work. While it could still be further improved with future work, it shows how KT models perform on multi-KC tracking scenarios. By introducing the evaluation procedure, our research shows the important gap of the current DKT model, which lacks the ability to track multiple skills on students' incorrect submissions. Furthermore, the design of the KC layer incorporates the Q-matrix into the DKT model, and the performance of such incorporation shows improvements. The result indicates the potential of methods similar to these in future research. Finally, the results of incorporating code features negatively affect the predictions. It calls for more investigations in this discipline as the results contradict prior research [17], which a lack of training data could cause. For teaching, this project is a stepping stone to enabling intelligent tutoring systems for educational applications such as automated feedback [23] and problem recommendation. It shows that the learning systems are not tutoring systems, as the data collected from a learning system [4] has a long way to go before enabling intelligent applications such as formative feedback and automated problem recommendation. Enabling the tracking of multiple KCs will benefit students' learning through these applications, but it is not yet ready due to relatively low performance on KC detection for computing education.

5.3 Limitations

The work has limitations that could be further addressed in future research. One limitation in this work could come from the Q-matrix. The Q-matrix we use in this project is relatively simple, as it only contains three KCs, and some (4/10) problems only practice one KC that we track. With a Q-matrix created focusing on all possible KCs, the performance of the models could be improved, however, it is difficult to create one covering all. Furthermore, the setting of KCs also has gaps in the KC properties in theory. Practically, given a set of problems, we only focus on certain KCs when students practice. However, students may still experience difficulty with KCs out of the scope, especially in open-ended practices such as open programming. This leads to the incompleteness of the Q-matrix we model and incorporate into the model, and further improvement may be achieved with a more complete Q-matrix. Nevertheless, creating a high-quality Q-matrix is time-consuming

and difficult, even for experts in domains such as computer science. A second caveat of the work is that the model design is still relatively simple in incorporating educational theory but may be overly complicated in the programming features. For example, the training data may not support the learning of so many parameters in the model for code feature extraction; however, a better-designed KC layer may improve the model more. It is a research task to balance these two considerations in model design better. A third limitation of the work is that we only present the results for one dataset, and they could apply to another dataset, but they may not. Further research should focus on exploring the model performance on other datasets. However, multi-KC datasets are scarce in the computing education context. Finally, one observation of the data is that there may be cheating students in the set, which can cost the models their performance. We have used simple methods to filter potential cheating students, but it relies on future data collection work to collect high-quality data that filters cheating students out. In addition, there is room to investigate further the code features that caused the negative impact on performance. We will include a comparison in future work to discuss the code features. Finally, Bayesian Knowledge Tracing (BKT) models are not compared in this work. While prior work (e.g., [15]) cites a much lower performance of BKT models in programming knowledge tracing, further work could be focused on comparing performance between BKT and DKT models to evaluate multi-KC knowledge tracing performance on more models. In conclusion, this work is an exploration of DKT models on their ability to track multiple KCs at the same time. While the results show limited ability of DKT models, future work should be further investigated for more datasets and other methods on the task.

Acknowledgement

This material is based upon work supported by NSF under Grant No. #2013502 and #2112635, and the Goodnight Educational Foundation.

6. REFERENCES

[1] T. Barnes. The q-matrix method: Mining student response data for knowledge. In *AAAI 2005 Educational Data Mining Workshop*, pages 1–8, Washington, DC, 2005. Association for the Advancement of Artificial Intelligence (AAAI).

[2] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

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

[4] 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, New York, NY, USA, 2017. Association for Computing Machinery.

[5] J. L. Fleiss, B. Levin, and M. C. Paik. *Statistical methods for rates and proportions*. john wiley & sons, Toronto, Ontario, 2013.

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

[7] M. Khajah, R. V. Lindsey, and M. C. Mozer. How deep is knowledge tracing?. *International Educational Data Mining Society*, 2016.

[8] K. Koedinger, P. Pavlik Jr, J. Stamper, T. Nixon, and S. Ritter. Fair blame assignment in student modeling. In *Proceedings of the 4th International Conference on Educational Data Mining*, pages 91–100, 2011.

[9] K. R. Koedinger, A. T. Corbett, and C. Perfetti. The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning. *Cognitive science*, 36(5):757–798, 2012.

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

[11] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein. Deep knowledge tracing. *Advances in Neural Information Processing Systems*, 28:1–13, 2015.

[12] T. W. Price, R. Zhi, and T. Barnes. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *International conference on artificial intelligence in education*, pages 311–322, New York, NY, 2017. Springer, Springer Publishing.

[13] N. Salomons, E. Akdere, and B. Scassellati. Bkt-pomdp: Fast action selection for user skill modelling over tasks with multiple skills. In *IJCAI*, pages 4243–4249, 2021.

[14] R. Scruggs, R. Baker, and B. McLaren. Extending deep knowledge tracing: Inferring interpretable knowledge and predicting post system performance. In *Proceedings of the 28th International Conference on Computers in Education*, 2020.

[15] Y. Shi, M. Chi, T. Barnes, and T. Price. Code-dkt: A code-based knowledge tracing model for programming tasks. In *In Proceedings of the 15th International Conference on Educational Data Mining (EDM) 2022*, pages 50–61, Massachusetts, 2022. International Conference on Educational Data Mining.

[16] Y. Shi, R. Schmucker, M. Chi, T. Barnes, and T. Price. Kc-finder: Automated knowledge component discovery for programming problems. In *Proceedings of the 16th International Conference on Educational Data Mining (EDM) 2023*, Massachusetts, 2023. International Conference on Educational Data Mining.

[17] 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*, LAK21, page 606–612, New York, NY, USA, 2021. Association for Computing Machinery.

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

[19] X. Xiong, S. Zhao, E. G. Van Inwegen, and J. E. Beck. Going deeper with deep knowledge tracing.

International Educational Data Mining Society, 2016.

[20] Y. Xu and J. Mostow. Using logistic regression to trace multiple sub-skills in a dynamic bayes net. In *EDM*, pages 241–246. Citeseer, 2011.

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

[22] J. Zhang, J. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*, 2022.

[23] J. Zhang, D. Li, J. C. Kolesar, H. Shi, and R. Piskac. Automated feedback generation for competition-level code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.