KC-Finder: Automated Knowledge Component Discovery for Programming Problems

Yang Shi[1], Robin Schmucker[2], Min Chi[1], Tiffany Barnes[1], Thomas Price[1]
[1]North Carolina State University
[2]Carnegie Mellon University
yshi26@ncsu.edu, rschmuck@andrew.cmu.edu, {mchi, tmbarnes, twprice}@ncsu.edu

ABSTRACT
Knowledge components (KCs) have many applications. In computing education, knowing the demonstration of specific KCs has been challenging. This paper introduces an entirely data-driven approach for (i) discovering KCs and (ii) demonstrating KCs, using students' actual code submissions. Our system is based on two expected properties of KCs: (i) generate learning curves following the power law of practice, and (ii) are predictive of response correctness. We train a neural architecture (named KC-Finder) that classifies the correctness of student code submissions and captures problem-KC relationships. Our evaluation on data from 351 students in an introductory Java course shows that the learned KCs can generate reasonable learning curves and predict code submission correctness. At the same time, some KCs can be interpreted to identify programming skills. We compare the learning curves described by our model to four baselines, showing that (i) identifying KCs with naive methods is a difficult task and (ii) our learning curves exhibit a substantially better curve fit. Our work represents a first step in solving the data-driven KC discovery problem in computing education.

Keywords
Computing Education, Knowledge Component, Interpretable Deep Learning, Neural Network, Code Analysis, Learning Representation

1. INTRODUCTION
Modeling new learning domains is a common task for researchers and educators [44, 23] which often involves identifying a set of domain-relevant skills[1], and determining when (e.g., in which problems) students practice each individual skill. Multiple data-driven approaches have been proposed to either improve existing learning domain models or to discover fully new models automatically using student log data. The benefit of such approaches is that they can alleviate the need for expert authoring (e.g., via cognitive task analysis [11]), and can reveal skills and problem relationships that may be counter-intuitive to the domain experts (e.g., due to blind spots [32]). Traditionally, these methods (e.g. [7, 9, 38, 24, 8, 15, 25, 34, 33]) output a Q-matrix, which maps the discovered skills to individual practice problems. With a defined Q-matrix, researchers and educators can leverage student modeling techniques such as knowledge tracing [13] to assess what exact skills the student knows and doesn't know and can provide personalized problems recommendations that target the student's specific knowledge gaps [7].

However, in domains such as programming where there exists heterogeneity in viable solution paths, simply knowing which skills are relevant to a given problem is often insufficient – we also want to know *when each of those individual skills is successfully demonstrated* in student practice, and when it is not. For example, if a student attempts a problem that requires the use of multiple skills (e.g. conditionals, iteration, logic, etc.), it is helpful to know which of these skills they have demonstrated successfully. This can aid us (i) better understand how students struggle and learn, and (ii) adapt teaching to individual student's needs by offering personalized help and instruction. In other words, rather than viewing success on a problem as a binary outcome (correct/incorrect), it would be helpful if a model could detect how successful a student's attempt is along the dimensions of each of the problem-relevant skills (e.g., loop correctness, iteration correctness, etc.). In domains like programming, many problems require students to apply multiple skills, and it is difficult to break problems down into single-skill substeps. Doing so requires making use not only of binary correctness information, as in prior work [9, 24, 38], but also information from students' actual code submissions. Note that this goal is distinct from that of predictive student modeling (i.e., knowledge tracing [13]), which in programming tasks predicts students' binary submission correctness (such has been done in [49]); instead, we are concerned with detecting more *fine-grained evidence* of knowledge being demonstrated during practice (i.e. successful or unsuccessful demonstration of multiple skills, rather than a whole problem).

As a first step towards this goal, this work explores how well a data-driven approach can discover *candidate* KCs (skills) that (i) can be detected automatically from student code

[1]Skills and knowledge components are used interchangeably in this paper. Knowledge components (KCs) are defined in [23] and introduced in Section 2.1.

submissions – allowing us to track when students successfully demonstrate each of them, and that (ii) conform to the learning theoretic properties of KCs suggested in prior literature [23]. For (ii) specifically, we attempt to discover candidate KCs that **fit idealized learning curves** [53] – meaning that students get better as they practice individual skills, quickly at first and then more slowly. The student error rate reduction over time is assumed to follow a power law (namely the power law of practice) [53]. KCs are also expected to be informative for predictions of students' success on the current problem [13]. Our goal in this work is to first understand how well the discovered candidate KCs meet these criteria, and then to explore how they can inform our understanding of student learning. We propose the KC-Finder algorithm which takes as input sequence data describing students' code submissions on a series of practice problems, and that outputs: (i) a set of candidate KCs; (ii) a Q-matrix mapping these KCs to individual practice problems where they are relevant; and (iii) a detector that can estimate, for a given student attempt on a problem, confidence values describing which of the relevant KCs were demonstrated correctly, allowing us to reason about why an incorrect attempt was made. We introduce a loss function inspired by learning curve analysis to train a deep learning model whose predictions conform to idealized learning curve [9]. We evaluate our approach by answering two research questions (RQs) using data from 351 undergraduate students in an introductory Java course:

- **RQ1:** To what degree do the discovered candidate KCs conform to the learning theoretical properties of KCs?
- **RQ2:** What kind of patterns have we discovered as KCs in students' code and how do they inform our understanding of student learning?

For **RQ1** we evaluate the candidate KCs by calculating a loss describing the fit to expected learning curves. For **RQ2**, we conduct a case study analysing concepts and skills tracked in student submissions. Our findings suggest that the KCs discovered by our data-driven approach induce learning curves conforming to the power law of practice. The discovered KCs are sometimes (but not always) meaningful and non-obvious to domain experts. However, we also found that the discovered KCs were no more predictive of student success than random, laying the groundwork to explore how to satisfy both learning theory and predictive performance.

2. RELATED WORK

2.1 Knowledge Components

We use "knowledge component" (KC) as a term to describe the skills students learn by practicing a set of programming problems in the computing education domain. The concept of KCs was introduced in the Knowledge-Learning-Instruction (KLI) framework by Koedinger et al. [23]. The KLI framework connects teaching and assessment via observable and unobservable events in the student learning process: instructional events, assessment events, and learning events. Learning events are defined as cognitive (or from a biological view, brain) changes occurring when students learn. While learning events cannot be directly observed or controlled, they are caused by instructional events such as explanations and lectures which are observable. Assessment events (exams, discussions, etc.) are used to probe the student's knowledge state which on its own is not directly observable. The framework defines the knowledge students learn through unobservable learning events as KCs which are a concept that builds the bridge between learning events and assessment events. In the general KLI framework, KCs can also refer to other terms (e.g. concept, principle, or fact). In this specific paper, we refer to KCs as skills, and by knowing a skill, we mean knowing certain concepts/principles/facts and how/when to use them. While there may be different kinds of skills (procedural, declarative, etc.), we do not distinguish these since this paper's focus is to find any skills relevant to programming tasks regardless of their nature. KCs can have different levels of granularity: for example, in programming, "knowing how to write iterations" is a skill, however, a more fine-granular KC can be "knowing how to use `for` correctly". Problem-KC relationships enable us to track students' knowledge mastery as they work through a set of problems [1] via knowledge tracing algorithms. Well-defined KCs and problem-KC relationships are essential for knowledge tracing algorithms such as Bayesian knowledge tracing (BKT) [13], AFM [9], PFA [37] and DKT [39] which estimate a student's mastery of skills based on the correctness of their responses to previous practice questions. The modeling of student knowledge states enables intelligent tutoring systems (ITSs) to adapt the workflow to individual students. For example, SE-COACH [12] uses KC-driven models to decide steps that need explanations, and Salden et al. [45] used KC-based student models to examine the process of studying worked examples and how knowledge is transfered when solving problems. In our work, KC-Finder automatically discovers candidate KCs from student code submissions for these systems to work in the CS education discipline.

2.2 KC Discovery & Data-Driven Refinement

Many student modeling tasks require the definition of KCs and problem-KC relationships. The task of identifying a set of suitable KCs and assigning them to individual practice problems is complex and requires substantial effort from domain experts and techniques such as cognitive task analysis (CTA) [11]. Even then the resulting KCs can suffer from biases and blind spot effect [32] inducing a need for additional refinement techniques (e.g., [7, 9, 16]). Further, the design of detectors that determine when a KC is demonstrated when a student attempts a certain practice problem is highly labor-intensive [24]. Data-driven techniques that leverage student log data have been proposed to refine existing expert Q-matrices (e.g., [7, 9, 24, 16, 34]) and to discover new KCs (e.g., [38, 8, 25, 34, 33]). These approaches demand less effort from human experts and can mitigate blind spot effects, but they may lead to less interpretable KCs.

One common method to evaluate KCs is learning curve (LC) analysis [9]. When evaluating KCs one hypothesis underlying LC analysis is that the collective error rate of a population of students in a KC decreases as they practice more. This trend is assumed to follow an exponential curve (e.g., the power law of practice [53]). Multiple methods have been proposed to improve the domain-specific Q-matrix using LC analysis. For example, learning factors analysis (LFA) [9] combines the additive factors model (AFM) with A* search to refine an expert Q-matrix, and relies on learning curve fit as optimization criterion. In this paper, our idea is similar to LFA. We also use learning curves to guide the optimization process, but differences exist. The biggest difference is that we do not require initially defined KCs and Q-matrix, and our main output is a set of discovered KCs. In addition to using learning curves in our loss function and evaluation metrics, inspired by performance predictions approaches (such as BKT [13] and DKT [39], and newer models that use learning curves in knowledge tracing [55]), we also add response correctness and actual student code submission information into the model optimization process to discover KCs suitable for performance predictions. This is similar to the work by Shi et al. [49] who incorporated code information into DKT. Shi et al. [49] also worked on a deep learning model, but the key difference is that we propose to discover KCs, while they focus solely on predicting student performance.

2.3 Student Modeling in CS Education

Student modeling in computer science education has its own challenges which set it apart from other domains such as math and science education. For example, open programming problems are hard to perform knowledge tracing on due to the inherent complexity of the individual problems and the heterogeneity of viable solutions. Some prior works aimed at finding KCs suitable for CS education. While the Lisp tutor [3] introduces KCs with the tutor's design, it does not allow students to write code freely, which greatly constrains the space of possible student code submission. It is difficult for teachers to perform CTA for open coding problems and to find suitable KCs. Gusukuma et al. [21] proposed a framework to identify misconceptions and find KCs accordingly, but the approach still requires substantial effort from domain experts, and the KCs they discovered have not yet been evaluated quantitatively. Rivers et al. [44] proposed using normalized nodes from abstract syntax tree (AST) representations of student code as KCs, and evaluated them with learning curve analysis. However, some KCs discovered by their model did result in learning curves not conforming to the power law of practice. One can hypothesize that this might be due to limitations of canonicalization algorithms (the process of converting student code into a standardized format). We look at this problem from a different point of view and hypothesize that the guidance of learning curves in the KC discovery process can help recover better fitting learning curves. We use a neural network structure subjected to a constraint inspired by ideal learning curves to identify KCs that warrant well-fitting representations. There is also related work with focus on applications of student models in CS education. Yudelson et al. [56] extracted student code features and used them for code recommendation; finding KCs can also help us attribute errors from student code, and thus may aid in subgoal detection tasks [31] and may enable better feedback [40] and hints [43] to students. Some recent works focused on student performance prediction [30, 28, 22, 58] by leveraging code submissions (though using experts or data-driven code features). Finding suitable KCs may help such models make more accurate predictions. Discovering KCs is still a key mission in CS education to improve many of these applications.

2.4 Deep Code Learning for CS Education

The advancement of deep learning and big data analysis algorithms has significant impacts in diverse domains including code analysis [2, 57, 10]. Many related techniques have found application in the CS education domain due to the increasing size of available datasets [26]. A frequently applied model is code2vec [2], which has been used to detect bugs and misconceptions from student code [50, 52, 51], and recent extensions also approach student performance prediction tasks [49, 29]. Other research used code2vec for general classifications of educational code in a block-based setting [19]. The recent Codex model (and the related CoPilot tool [10]) caught the attention of many CS educators. Codex is widely used for code auto-generation tasks, and has also achieved promising resulting when used to generate student code explanations [46]. While deep learning-based approaches often yield high predictive performance, they tend to be less interpretable then traditional modeling approaches (despite the effort from [17]) and it is difficulty to extract insights into the learning process that can be explained to students and teachers. Our approach leverages learning curve analysis to guide the model training process and aims to build a more trustworthy and explainable deep learning model for student modeling tasks in CS education.

3. METHOD

Our target is to discover KC candidates using constraints inspired by learning theory. Suitable KCs are expected to be informative in student performance predictions and should induce learning curves that follow the power law of practice. Overall, there are four assumptions about KCs made in our work to build the KC discovery model. They are introduced below along with the theoretical rationales behind them.

3.1 Assumptions

The assumptions underlying the model design are **A1:** The collective error rate of students on a given KC *decreases* with subsequent opportunities to practice that KC. This decreasing trend is assumed to follow the power law of practice [9, 53]. **A2:** The demonstration of KCs in a problem solution attempt should be *predictive* of the attempt's correctness. **A3:** KCs are *detectable* from a student's current code submission. **A4:** All problems have a *fixed set of KCs*, meaning that the related KCs for a problem are fixed, independent of the submissions from students. **A5:** All KCs have the same *initial error rate and learning rate.*

A1 states that observations from students practicing KCs should induce learning curves that conform to the power law of practice. It is natural to assume that when practicing KCs, as students practice more, they become more proficient in these KCs, and thus make fewer mistakes. The power law of practice postulates that the collective error rate from all students decreases as students practice more following a power function ($Y = aX^{-b}$). This assumption has been made by multiple prior student modeling approaches [9, 44].

A2 assumes that if a student knows all KCs relevant to a problem, it indicates they are likely to answer the problem correctly. When incorporating this assumption into a data-driven model, it implies that the demonstration of KCs relevant to a problem in a solution attempt should be predictive of attempt correctness. For example, if a problem requires the application of KCs A and B, a successful demonstration of KC A should suggest an increased likelihood of getting the problem right. This assumption has been used in many knowledge tracing models [13, 6, 35, 47] that make predictions on future submissions with a focus on domains with closed and structured questions. However, for open-ended, free-form computer science problems, the complexity and intertwinedness of individual KCs may cause more difficulty in performance prediction. We thus limit our assumption to the current submission correctness.

A3 postulates that KCs are observable and detectable from students' code submissions. We only have access to the code submissions, and in this work we only focus on KCs that can be extracted from code submissions. While some KCs may exist that are not directly observable through code submissions (for example, reading skills are also required when students try to solve programming tasks specified by text requirements), various KCs can be observed in code submissions. For example, Rivers et al. [44] extracted various KCs represented by AST nodes derived from code submissions.

A4 and **A5** are assumptions specifically made in the design process of our current model. They may not necessarily be true, but we use them to facilitate the creation and training of the KC-Finder model. Future work can focus on loosening these assumptions. A4 as a limitation, states that the number of practiced KCs is fixed for each problem. In many cases this is true, but whether students practice certain KCs can also be affected by the code they write and the solution path they choose. For example, in an open-ended programming problem, the instructor may expect students to solve the problem by using nested `if` conditions, but some students may choose more complex logic operations to avoid nested `if`. Under this assumption, we assume that the student *did not demonstrate* a correct practice of the required KC "nested `if`", while the student actually indeed *correctly practiced* the KC "complex logic operations" and "nested `if`" is **not required** by the problem. We use this assumption to allow the usage of the Q-matrix since if KCs are dependent on submissions, one Q-matrix cannot represent the KC-problem relationship since the relationship varies across different submissions. In A5, we have an assumption that requires all candidate KCs we discover to have the same starting error rate (a) and learning rate (b) in the power function for their learning curves. We set this assumption as a start for using properties of the power law curve, and save the automatic fitting of more specific learning curve parameters for later work. Our experiments show that we can still discover meaningful KCs under this assumption.

3.2 KC-Finder Model

Under the guidance of these theoretical properties, we define the KC-Finder model structure below. Figure 1 provides an overview of the model. We show a single student and their T code submissions $\{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_T\}$ for a course as example to illustrate the process of the model. The output of the model is specified in orange in the figure, where the current submission correctness is indicated as $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_T\}$.

In a submission $\mathcal{S}_t$, two types of information are available to the model for the correctness classification task: the problem ID p_t and the actual code submission c_t. While both cannot be immediately processed by a mathematical model, we separately represent the problem ID and code submission information as real-valued vectors. For the problem ID, we use one-hot vectors to represent the IDs as vectors, using a vector $\mathbf{x}_t$ to represent p_t, where the length of the vector equals the number of problems in the dataset, with all elements assigned as 0, except the element associated with the problem assigned as 1. Note that this setting is similar but different to typical knowledge tracing tasks [39, 49, 4] which also use one-hot representations for problem IDs. Knowledge tracing includes the correctness information in the task of the *next* submission performance prediction (which is denoted as $\mathbf{y}_{t+1}$). In contrast, we do not include the correctness information since our task is to classify the correctness of the *current* submission $\mathbf{y}_t$, and to *discover KCs* through the model learned from this task. Code submissions c_t are embedded through a code2vec model [2], which has been recently introduced to educational analytics in multiple tasks [51, 19]. The code2vec model can embed a code snippet c_t into a vector $\mathbf{z}_t$. This part of the model updates parameters in the training process, along with other layers in the model.

Neural networks have a common structure. Linear layers (also called fully-connected layers) are defined by weight matrices and apply linear transformations to vector inputs. The product of these multiplications is often followed by non-linear functions such as sigmoid or tanh to introduce non-linearity into the model. In the KC-Finder model, all linear layers ($\mathbf{W}_{KC}$, $\mathbf{W}_c$ and $\mathbf{W}_p$) have the same mathematical operations (with different weights), which first multiply with the input vectors and then apply the sigmoid function (denoted as $\phi(\cdot)$) to every element to compute the output. For example, when a code vector $\mathbf{z}_t$ passes through the linear layer $\mathbf{W}_c$, the equation for this process is $\mathbf{h}_t = \phi(\mathbf{W}_c \mathbf{z}_t)$.

The vector $\mathbf{h}_t$ is of dimension L, where L is the total number of KCs. We intend to interpret $\mathbf{h}_t$ as the error rate of students practicing KCs $\{l = 1, 2, ...L\}$, but there are some challenges. First, not all problems practice all KCs, and our model needs to learn problem-KC relationships. To this end, we leverage the one-hot problem embeddings $\mathbf{x}_t$ to infer weights and represent the KCs corresponding to the current problem. We use linear layer $\mathbf{W}_{KC}$ to learn a relationship between potential KCs and problems, and use a sigmoid function to scale the output of layer $\mathbf{m}$ to the $[0, 1]$ range. The layer-$\mathbf{m}$ weights then multiply with the values of $\mathbf{h}_t$ and generate a vector $\mathbf{k}_t$. The intuition behind this is that every problem can have a probability of practicing certain KCs, and we use $\mathbf{m}$ as this probability and multiply the $\mathbf{h}_t$ vector to represent the selected KC values. The output $\mathbf{k}_t$ is a masked representation of the knowledge status of a student. When using an ideal learning curve to force the distribution of the representations across students in a batch, it can readily be interpreted as the error rate of students practicing KCs as we expect it to follow the power law of practice. For a batch of students' submissions that practiced a KC, cumulatively they should also follow the power law of practice, and preserve their ability to predict the submission's correctness. These two properties lead to the design of the loss function which is used to train the KC-Finder model:

$$\mathcal{L} = \alpha(\frac{1}{N}\sum_{N} H(\mathbf{y}, \hat{\mathbf{y}})) + (1-\alpha)(\sum_{T,L} |\frac{1}{N}\sum_{N} k_{n,t,l} - \hat{k}_{t,l}|) + \gamma(||\mathbf{W}_{KC}||_1). \tag{1}$$

In Equation 1, the loss when the model has a batch of N student submissions comes from three sources. The first part

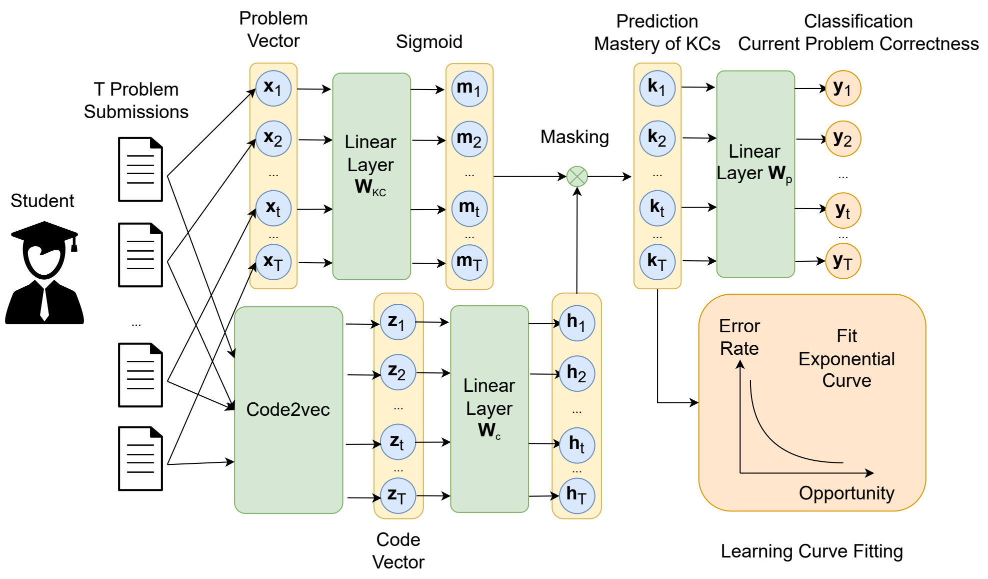


Figure 1: KC-Finder Model structure, where blue nodes are vectors, and green blocks represent neural network structures.

$H(\cdot)$ is the binary cross-entropy [14] of the classified results y and the ground truth correctness $\hat{y}$, leading the model to learn weights that produce a low submission correctness prediction error. The second part is the loss for the fitness of the learning curve to encourage predictions that conform to the power law of practice. Since the learning curve is calculated through a batch of students, we first average the error rate of practiced KCs for every student in the batch. For a certain knowledge component l, we calculate the assumed learning curve, where $\hat{k}_t = at^{-b}$. In the equation, a stands for the starting error rate for students when they have not practiced a skill, while b denotes the learning rate. Because we cannot estimate KC-specific a, b parameters a priori without employing additional information or assumptions, we assume all KCs have the same a, b parameters. This learning curve fitness loss is optimized so that the model produces $\mathbf{k}$ vectors that can be interpreted as the error rate of practiced KCs. We use an α hyperparameter to control the importance of the classification loss and the fitness loss. The last part of the loss is an L1-norm regularization term weighted by hyperparameter γ which ensures sparsity in the $\mathbf{W}_{KC}$ weights and thus creates a more sparse masks $\mathbf{m}$, allowing KCs to be removed from unrelated problems. While the output of this classification is a binary label describing whether a student succeeds in their submission, the key product of the work is the Q-matrix specified in $\mathbf{m}$ and the learning curves $\mathbf{k}$ on the corresponding knowledge components.

4. EXPERIMENT

4.1 Dataset

Our experiments use the publicly available CodeWorkout dataset[2]. The dataset is collected from an introductory Java course at Virginia Tech in Spring 2019. The dataset is stored in ProgSnap2 [42] format and is released to the public in the 2nd CSEDM data challenge[3]. No identifiable information (such as geographical information, GPA, etc) on individual students is released, and the dataset has been anonymized for ethical considerations. The dataset includes submissions from 410 students for 50 programming problems, which are grouped into 5 assignments according to the topics. For example, the first assignment mainly focuses on the `if` conditions, while a later assignment has more problems on `for` loops. The typical length of the student code submissions is 10 to 20 lines and 41.86 tokens, submitted to the CodeWorkout [18] platform, and tested by pre-defined test cases. The unknown tokens are specifically assigned as a unique identifier [unk] in the model. To avoid overfitting to the problems, user-defined variables and strings are also normalized to a fixed string in students' code. On average, 23.68% of all submissions (from all students) are correct, meaning they passed all test cases. Our model involves training, validation, and testing phases. We split the dataset according to students by a ratio of 3:1:1 for each of the three phases. We train the model with training data, use the validation set to tune hyperparameters, and test and evaluate the model on the testing set. The results are averaged through 5 times repeated sampling to ensure the result are reliable.

4.2 Data Preprocessing

Code submissions are complex, and we only use the first attempts of students on each problem for potential KC discovery. One reason is that it has been common for knowledge tracing tasks to only consider the first submissions for problems [48, 4], as students practice skills when they first see the problem and have not received any feedback on the specific problems. Another reason is that for code submissions, students tend to debug on their later submissions. This process involves more complicated behavior, which may not be fully explainable by conventional knowledge component modeling. Sometimes students even get intimidated by problems in case of repeated incorrect submissions, only to click the submit button multiple times and thus make invalid submissions that do not show what they know and don't know.

[2]https://pslcdatashop.web.cmu.edu/Files?datasetId=3458
[3]https://sites.google.com/ncsu.edu/csedm-dc-2021/home

Therefore, as it would be a non-trivial task to evaluate candidate KCs for multiple submission situations, we only use the first submissions from students for every problem. This is also different from other analyses such as "learning trajectory analysis" [54], as we only focus on the submissions when students practice skills for the first time.

Some students may also have exhibit cheating behavior in the dataset (since the system used to collect data don't have a detector for cheating, unfortunately). We also observed students submitting partial code as their first submission or potential cheating. For example, some students struggled with easier problems, but suddenly are able to make correct submissions on their first try after a certain problem, and those later problems are generally more complicated than the ones they failed many times before.

We added preliminary filters to keep only the first attempt submissions from students and avoid students with possible cheating behavior. To only keep the first submissions, we implemented a filter to detect the first submissions that are not too short (longer than 3 lines of code) and only kept these first submissions. We also implemented a filter to remove students with sudden changes in their submission patterns. Out of 410, we finally kept 351 students in the dataset for the KC discovery task.

4.3 Hyperparameter Tuning

We used validation sets for hyperparameter tuning. Specifically, we applied grid search to find hyperparameters yielding the highest classification AUC scores on validation sets for model evaluations. This process is repeated 5 times to reduce the risk of overfitting. We selected the learning rate for the model as 0.005 through space of $0.001, 0.003, 0.005$, and the training epochs are selected as 80 through the early-stopping process. We also tuned the α parameter (specified in Equation 1) and used a value of 0.97 (in a range of 0.03 through 0.97). At the same time, the lower weight on learning curve fitting loss does not have a significant effect on the potential KCs discovered (they still produce good learning curves even if set low). Still, a higher weight on classification loss is needed for the classification task. The γ parameter controls the speed of removing potential KCs from unrelated problems and is set to a low value of $3e-5$. The other model hyperparameters are kept the same as the default ones in the settings of the code2vec and DKT models [2, 39]. We did not tune on the different combinations of the a and b parameters of the expected learning curves of the KCs since it is hard to evaluate the quality of potential KCs quantitatively, as in real applications, it is also feasible to try different a and b parameters and check the quality of potential KCs found under various combinations. In our experiments, we used the values $a = 0.7, b = 0.6$. We also assumed that $L = 30$ potential KCs exist in the dataset, and while the model is able to reduce the number by learning a candidate KC not practiced in any problem, the number is an educated guess by the authors after checking through the problems. Future experiments can be introduced to evaluate more value combinations. We kept these values since there is no direct way to quantitatively evaluate the quality of programming skills or misconceptions discovered by the models. The model is trained with a computer equipped with an Nvidia GeForce RTX 2080 Ti GPU. A single run of the training takes less than 10 minutes, and inference of a single batch of students takes seconds of computation. The code is implemented in PyTorch [36] and publicly available[4].

4.4 Baseline KC Models

We compared the KC discovered by our model with four baseline methods. One can argue that topics of the problems can be extracted and assigned as the KCs in each question. As the first baseline in our experiment, one of the authors manually examined the problem requirements and code solutions, identified a set of 15 *Topic KCs*, and manually tagged each problem with a set of relevant KCs. The second baseline used an alternative way to extract KCs that examines the student code submissions and extracts the most frequently used code components (show up in more than 20% of all solutions) in correct submissions as KCs for certain problems. This is a simplified KC model compared to [44], as we do not have an automatic hint generator for Java compared to their work for Python programs. We implemented this method to extract the nodes from the AST representation of student code, and filtered nodes that show up in more than 20% of correct submissions as KCs required by the problem, resulting in 21 *Node KCs*. Textual and numerical leaf nodes of ASTs were removed from KCs since they vary among problems. Lastly, we considered two standard baselines from prior work [37] one which uses a single KC for all problems (i.e., general programming knowledge), and another that defines a separate KC for each problem (i.e., each problem is its own KC). We compared our discovered candidate KCs with the four baselines using the fitness errors of the induced learning curves.

5. RESULTS

5.1 Learning Curves

We first show three example learning curves generated from our model using the testing dataset shown in Figure 2. To evaluate the fitness of the learning curves of each of the potential KCs, we calculated an average absolute error $e = \frac{1}{T}\sum_T |k_t - \hat{k}_t|$ to compare the curves to the expected curves under the exponential curve $\hat{k}_t = at^{-b}$, where the a and b parameters are automatically fitted. Note that for each KC candidate, only problems practicing the KC are counted when calculating the error e. For example, the KC candidate #5 is practiced in almost all problems. The KCs shown in Figure 2 all have a relatively low error compared with the assumed exponential curve. KC candidate #5 has an error of 0.037, KC candidate #4 has an error of 0.026, and KC candidate #2 has an error of 0.032 All KCs candidates we extracted have a $e < 0.1$, and the mean error is 0.034, showing that the learning curves of the potential KCs are generally consistent to the learning curve, and follow the power law of practice. On the other hand, the baseline KCs do not create KCs that fit the expected exponential learning curve. We show four learning curves created from the baseline Topic KCs and the node KCs in Figure 3. The two learning curves on the left represent the error rate of the *submissions* when certain Topic KCs are practiced, and the right side shows learning curves when node KCs are practiced. We can clearly see that neither KC models on the

[4]Code Repository:
https://github.com/YangAzure/KC-finder

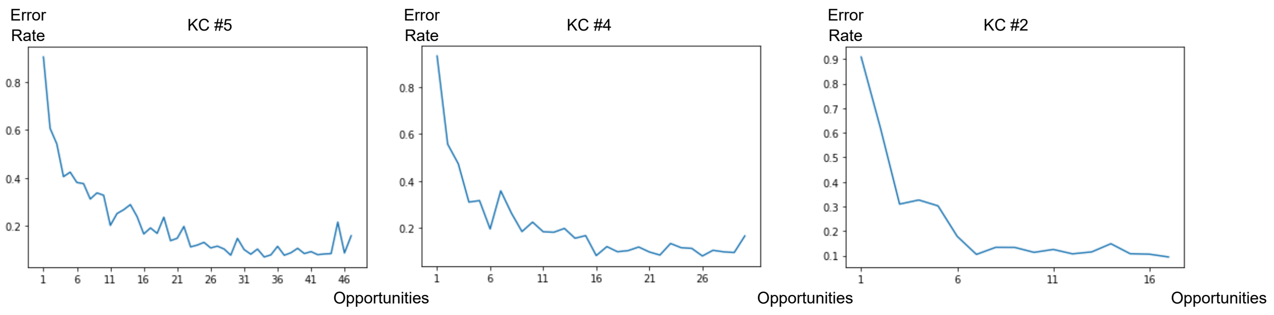


Figure 2: Learning curves of different KC candidates generated from students in test set.

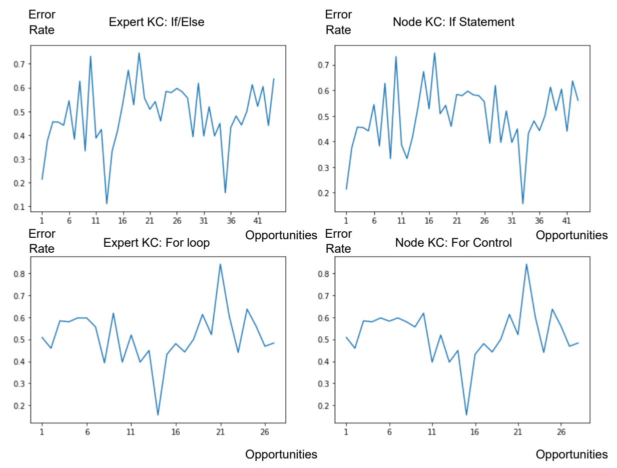


Figure 3: Learning curve examples of Topic KCs and Node KCs on concepts of `if` and `for` statements.

`if` or `for` statements show an exponential, or even decreasing trend in error rates. When calculating their fit to the assumed exponential learning curve for the model (with automatically fit parameters), we show in Table 1 that both KCs have very high fitness errors. Only a small subset of the KCs has a valid learning rate factor (b), presenting Topic KCs and node KCs cannot generate learning curves fitting the exponential curve with a fixed set of parameters. When looking at more learning curves generated from both KCs, they are similar to the examples in Figure 3 and do not have a decreasing trend over opportunities. The learning curves for Topic KCs and node KCs are similar to each other for a certain concept (for example the first two learning curves for `if` concept), which confirms that the expert extracted KCs are represented in code submissions as well. However, one limitation here is that the correctness may not directly represent the correct practice of certain KCs. The correctness metric represents students practicing all KCs in a certain problem correctly, but not on certain KCs. We use the correctness of the submissions to validate these two KC extraction methods as an approximation, which shows that our work can represent the learning curve for certain KCs without the need for a specifically designed partial evaluation of the submissions – only if we can explain the potential KCs discovered by the model.

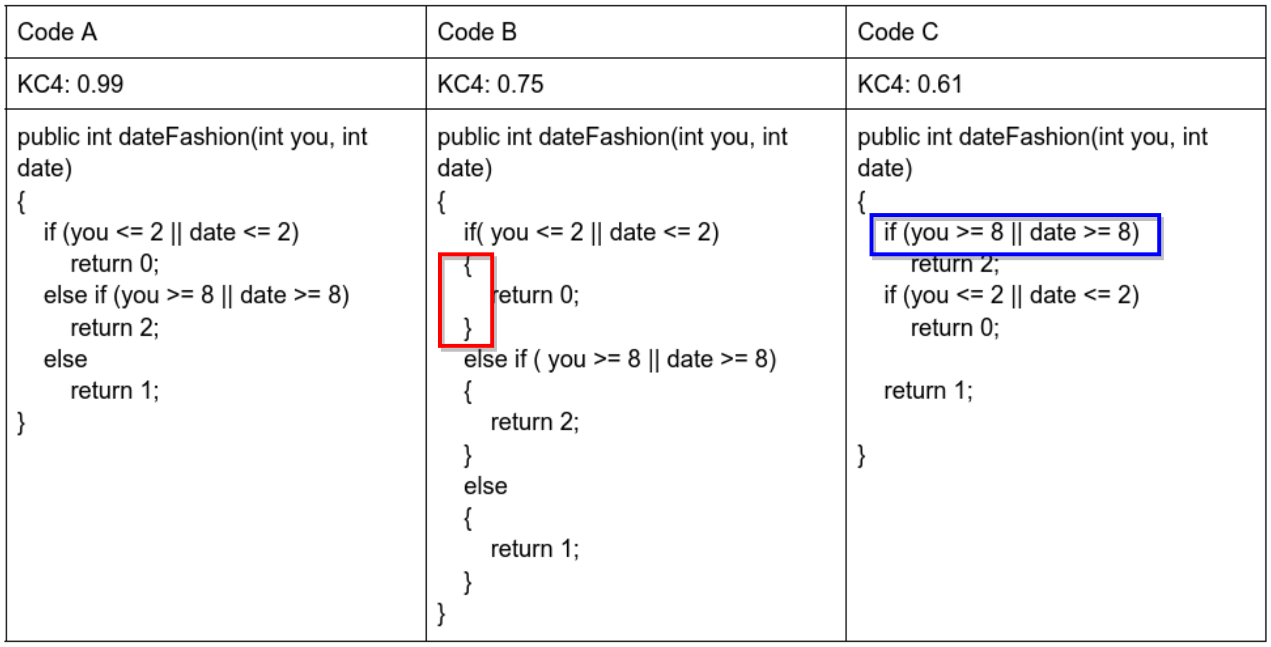
5.2 KC Interpretation

Our model discovers KC candidates represented as vectors that generate reasonable learning curves, and we show the interpretability of the KC candidates in this section. We manually examined the code and their corresponding KC values and found that we could find meaningful and interpretable KCs from these automatically discovered KC candidates. We manually inspected the discovered KCs across multiple problems, and show one example for the presentation purpose. In Figure 4, we show an example case of a KC candidate (KC #4) and explain what has been tracked in this KC in one problem. The problem requires students to use `if` conditions with logic operators, and one typical and non-obvious error from students is to use the order wrongly and return incorrect values that cause test cases to fail. It should be noted that the values do not indicate whether the KCs are practiced or not in certain problems. A low KC value means a failed demonstration of the candidate KC. Code A submission is correct, with a high KC value on KC candidate #4. Code C has a wrong order and has a low KC value on the same KC. While other reasons could cause the difference in KC values, Code C is incorrect due to the wrong order. This KC could possibly track bracket usage, as the only difference between Code A and Code B is bracket usage. Using brackets led to a lower KC value for Code B.

We also found some other concepts associated with KC candidate #4, for example, in one problem, all students who have an error in using = as comparison operator == got lower KC values. One KC may not represent a clearly defined concept by experts. The KC candidates are almost certainly amalgamations of different concepts, and no single behavior seems to explain the KC itself. Some of these concepts are consistent through different problems, as the example shows are all related to the if condition, and some of the concepts are problem-specific. As we do not have any specific design in the mode, KCs can be conceptual and can be distinct when they tend to improve together. Some candidate KCs are meaningful and important skills for the problem (e.g. the sequence of if conditions as shown in 4), which instructors might not have intuited; however

5.3 Code Classification

In our experiments, we report the classification results through 5 times running and calculate the average of the runs to get the classification results. KCs should be informative to predict the correctness of the code submission. To serve as a sanity check, before we evaluate the discovered potential



Code A	Code B	Code C
KC4: 0.99	KC4: 0.75	KC4: 0.61

```
public int dateFashion(int you, int
date)
{
   if (you <= 2 || date <= 2)
      return 0;
   else if (you >= 8 || date >= 8)
      return 2;
   else
      return 1;
}
```

```
public int dateFashion(int you, int
date)
{
   if( you <= 2 || date <= 2)
   {
      return 0;
   }
   else if ( you >= 8 || date >= 8)
   {
      return 2;
   }
   else
   {
      return 1;
   }
}
```

```
public int dateFashion(int you, int
date)
{
   if (you >= 8 || date >= 8)
      return 2;
   if (you <= 2 || date <= 2)
      return 0;

   return 1;

}
```

Figure 4: Comparison of code submissions and their corresponding scores of KC #4. Frames show the possible code difference that triggered the KC value difference.

Table 1: Fitness error of Topic KCs, Node KCs, and KCs discovered by our model. The Valid LCs column indicates the number of LCs with a positive learning rate parameter ($b > 0$). A negative learning rate indicates a degenerate KC (students get worse with practice).

KC	Error	Valid LCs
Topic KC	0.0663	1
Node KC	0.0785	5
Model KC	**0.0342**	**30**

Table 2: AIC, BIC of Topic KCs, node KCs, one KC, all KCs, and KCs discovered by this model.

KC Model	AIC	BIC
One KC	3223.92	3651.62
All KC	3179.95	4189.79
Topic KC	3220.99	3672.45
Node KC	3214.84	3678.18
Random KC	3076.08	3848.31
Model KC	3081.44	3853.67

KCs, we evaluated an average of running the model on different splits of training and validation sets five times and reached an average AUC score of 77.26%. Considering that no correctness information is given to the model, and only discovered KCs are used for making the classification, the potential KCs can be used to classify the code correctness, showing the necessity of using the correctness information in the loss function.

5.4 Q-Matrix Analysis

We present the Q-matrix we found from the testing dataset, ranked by the number of problems in Figure 5, and compare the corresponding AIC and BIC scores in Table 2 to evaluate how well the discovered and baseline KCs predict student performance/correctness, following prior methods [9]. These metrics are frequently used to evaluate the goodness of fit. More detailed equations for the metrics can be found in prior works (e.g. [9]). The Q-matrix shows that the KCs are relatively evenly distributed through all problems with good sparsity. In the comparison results, the model scores are similar to a random KC model. It would be unsurprising to have this result, as the model learned KC candidates that were amalgamations of different and overlapping micro-concepts; therefore, it makes sense that a Q-matrix involving these KCs would not be meaningful. While our KC model does not generate better AIC/BIC scores than random KC, the other baseline models (even manually defined KC models). This shows that it is difficult to create a predictive KC model with the dataset. Furthermore, it suggests that fitting a learning curve itself is sufficient to discover KCs for domain modeling. We did not manually inspect the baselines' KC quality since they are specifically designed to represent different levels of KCs. For example, one KC and all KC are naive baselines that any domain could use, while the remaining baselines are expert-defined, and thus already fit to expert understanding of KCs. One direction for future work would be to seed the model with an expert-authored Q-matrix and allow the model to discover KC candidates which match the pre-specified pattern. This KC-refinement task has been explored by various works (e.g. [24]). However, doing so with our approach would help to address the challenge of figuring out which relevant KC a student is struggling with when they get a problem wrong.

6. DISCUSSION

6.1 Expected Properties

We first answer RQ1 in this discussion section: *How closely do the candidate KCs detect match the expected properties of KCs?*

Our goal was to discover candidate KCs that matched two important properties of high-quality KCs. First, students should be more successful at demonstrating KCs as they practice them, following the power law of practice [9, 24]. Second, a student's success on a given problem should be predicted by how successfully they demonstrated relevant KCs for that problem [13, 47]. To address this, we trained a model to detect candidate KCs and then evaluated those KCs on a separate dataset. Our findings suggest that the discovered KCs largely meet these two goals, much more so than the four baselines we compare with (see Section 4.4).

First, the resulting learning curves appear high quality, fitting well to a power law curve. By contrast, none of our baselines produced viable learning curves, suggesting that naive approaches for defining KCs are ineffective with our dataset. Similarly, prior work [44] has found that learning curves in programming often fail to align with learning curves. This result suggests that our model could detect patterns in student code that become more frequent as students practice – quickly at first and then slower with time – as suggested by the power law of practice [9]. As we discuss below, some of these patterns likely correspond to skills that students develop over time, such as the usage of `if` conditions, or (the absence of) misconceptions that become rare over time, such as using an assignment operator (`=`) instead of comparison (`==`) inside of a conditional statement. However, some of these patterns may simply correspond to code constructs that are used more frequently as the semester progresses (e.g. variables, which are rare in early assignments), and thus naturally increase in frequency, without in reality having much to do with practice. Matching learning curves is not itself enough to say a KC is meaningful, but it does suggest that some of the discovered KC candidates may correspond to learned skills.

Second, we found that, for a given problem, the relevant KC candidates were, collectively, predictive of students' correctness on programming practice problems (AUC = 77.26%). These results suggest that whether a student successfully demonstrates a candidate KC discovered by our model gives insight into whether they will succeed at the current problem. In other words, the code patterns underlying these candidate KCs are also important code patterns for solving the programming problems in our dataset.

Overall, these results suggest that the candidate KCs we discovered do match the expected properties of idealized KCs in these two dimensions. Importantly, the results we presented were from a hold-out test dataset with unseen students, suggesting that KC candidates can generalize across different students within a course. However, while these criteria are necessary, they are not sufficient, and we will explore the limitations of the discovered KCs below.

6.2 Skill Tracking

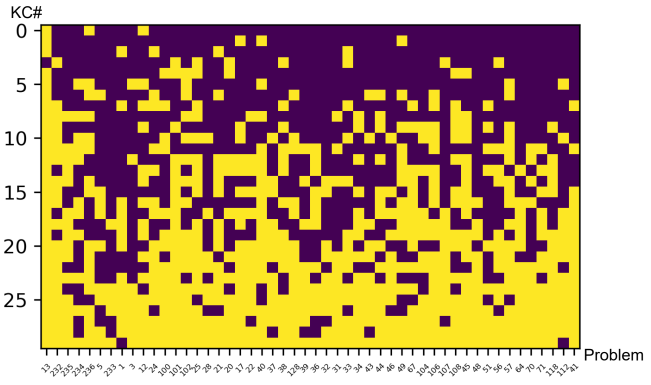


Figure 5: Q-Matrix representation of KCs and problems, where yellow cells represent a presence of the KCs in a problem, and dark cells represent an absence.

We answer RQ2 in this section: *What properties do the discovered KC candidates have? What kind of patterns have we discovered as KCs in students' code?*

First, we found that there are important differences between the discovered KC candidates and how experts would likely define KCs for a given domain. Rather than discrete concepts (conditionals), the KCs are amalgamations of different micro-concepts (e.g. correct ordering of the primary if-statements in a problem), where no single behavior seems to explain the KC itself, and different KCs overlap. Some of these micro-concepts show up across different problems (e.g. KC #4 detects the misuse of `=` in conditions (or `assignment in conditional`) misconception across various problems). Some of these are also problem-specific, e.g. the order of two if statements in the problem shown in 4. This suggests the need for further research on operationalizing the idea of a KC's ”consistency” – that a KC should mean the same thing when detected across students and problems. This is a non-trivial idea to encode in a model, which lacks any domain expertise. Ideally, such a definition should be defined based on student behavior (e.g. if a KC is consistent, students' performance for problems that use the KC will be correlated). In some ways, this idea is operationalized by approaches such as AFM [9]. However, since the actual domain information is not used, it would not be feasible to evaluate the performance of such methods. Although the Q-matrix learned by the model was not meaningful, we also found that the candidate KCs can inform our understanding of what skills students develop in a domain. For example, we found that KC #4 clearly detected a micro-concept focused on students' use of brackets in code. Importantly, these brackets did not change the function of the student's code, and it is unlikely an expert would have thought to include them as a discrete skill. However, our model identified this pattern as being predictive of student success and fitting a good learning curve. In retrospect, this makes sense as skill students develop: acquiring familiarity with syntax and style conventions (e.g. when brackets are and are not necessarily) can help students succeed on various problems, even if it does not directly affect their correctness. We also found that candidate KCs included misconceptions, such as the confusion of `=` and `==` in `if` conditions. More work is needed to develop methods for extracting the meaning of these data-driven KC candidates, and use this understanding to develop insight.

6.3 Design Choices

We made design choices according to our assumptions (See details in Section 3.1). Assumption **A3** specified that KCs should be detectable from code, and thus we used the code2vec model to process code into vectors $\mathbf{z}$ [2]. Code2vec has been used for educational data mining tasks recently [50, 49, 19], and the code embedding extracting module can be other models such as ASTNN [29, 57] as well. Assumption **A4** specified that problems should have a fixed set of KCs, and we thus 1-hot embedded the problem IDs into vectors $\mathbf{x}$, and use them to calculate a set of masks $\mathbf{m}$ to specify the KCs active for different problems. The masks select relative KCs for processed code vectors $\mathbf{h}$, and the selected values $\mathbf{k}$ participate in the loss calculation. According to assumption **A1**, one requirement is that if we let $\mathbf{k}$ correspond to KCs, they should follow the power law of practice. We designed the loss so that $\mathbf{k}$ values fit an exponential curve, generated by a fixed set of parameters, relaxed by assumption **A5**. Finally, since **A2** specifies the performance of KCs should be predictive of the code correctness, we use $\mathbf{k}$ values to make the predictions and have the model also train on the classification loss. When we assume the learning curves have the same parameters, the model may overlook KCs with very different starting error rates and learning rates. We thus used an L1-regularization to encourage the sparsity of $\mathbf{W}_{KC}$ and allow KCs to drop. While many variations and improvements can be made, this model is a proof of concept and serves as a prototype for the KC discovery task in the programming education domain.

6.4 Research and Educational Implications

This work introduced a fully data-driven KC-discovery algorithm designed for the CS education domain. It uses student log data describing the correctness of student responses and actual student code submissions to discover KCs and map them to individual programming problems. It connects pieces of *computer science education* with *learning theories* to discover a Q-matrix which conforms to the power law of practice [53]. It has been a different task from the KC refinement methods such as LFA, which uses learning curve analysis for KC-refinement, but it requires an initial Q-matrix [9, 24]. These student model improvement methods can reduce the load on experts when performing cognitive tasks analysis [11]. In contrast, our model directly reduces expert effort by providing a student model that produces KCs with learning curves fitting the power law of practice. Our model leverages deep learning structures, typically known as “black boxes”, however, we specifically designed the model such that the middle layer information can be interpreted as the KC ability estimates and thus made this model interpretable. The discovered KCs can be applied for performance prediction tasks by directly using the KC values or plugging the model structure to current knowledge tracing models for CS education [49]. While there are vector representations of student code submissions, they can also serve as language-agnostic representations to represent the mastery of KCs [27]. Our model can also be seen as a misconception attribution tool. When a student is predicted to have a high error probability on a certain KC, an automated hint or interference can be generated in an adaptive way to help the learning process [41]. Finally, the model can also be used to analyze the KCs covered by a set of problems using submission data from a semester, and thus to make more informed pedagogical decisions [20].

6.5 Limitations

Besides the assumptions we made to guide the model design, there are also other limitations present in this study. First, we did not incorporate the factor of using test cases. The run-time results of carefully designed test cases may contribute to the attribution of errors when students practice KCs. However, we do not have the full test case information for every problem in the dataset, and it is non-trivial to match KCs with specific test cases. Future work may integrate information about test case results into the existing method to enhance the KC discovery process. Second, we followed the tradition of knowledge tracing tasks and only used students' first submissions in model training and evaluation. One major limitation is that we cannot track the actual opportunities of practicing KCs in repeated submissions, especially if we don't assume that problems have fixed sets of KCs. We made this decision after the exploratory data analysis, during which we found lots of students made debugging submissions that are unnecessary. We found it can be complicated to explain this behavior, and are unsure if students actually intend to practice KCs when submitting a debugged code (for example, they may hit submission buttons multiple times, or they just wanted to exhaust possible choices to reach the correctness), as pointed out by Baker et al. [5]. It could also be interesting to investigate student behavior after their first submissions for programming problems. Finally, this model serves as a prototype and many variations could possibly generate better KCs. However, we do not have a metric to quantitatively evaluate the extent of KCs being reasonable and interpretable. While we do have a metric to examine the fitness of the learning curves and the classification of the code correctness, one limitation of the results is that the classification results are no better than random, and the discovered KCs are sometimes meaningful and non-obvious. The limited size of dataset may also cause a limited generalization on other datasets. We will explore ways to use expert knowledge to evaluate the quality of KC models, using this research to guide our future direction.

Acknowledgements

This material is based upon work supported by NSF under Grant No. #2013502 and #2112635.

7. REFERENCES

[1] V. Aleven and K. R. Koedinger. Knowledge component (kc) approaches to learner modeling. *Design Recommendations for Intelligent Tutoring Systems*, 1:165–182, 2013.

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

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

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

tracing models. In *In: Proceedings of the 14th International Conference on Educational Data Mining (EDM 2021)*. ERIC, 2021.

[5] R. S. Baker. Gaming the system: A retrospective look. *Philippine Computing Journal*, 6(2):9–13, 2011.

[6] R. S. Baker, A. T. Corbett, and V. Aleven. More accurate student modeling through contextual estimation of slip and guess probabilities in bayesian knowledge tracing. In *International conference on Intelligent Tutoring Systems*, pages 406–415. Springer, 2008.

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

[8] Y. Bergner, S. Droschler, G. Kortemeyer, S. Rayyan, D. Seaton, and D. E. Pritchard. Model-based collaborative filtering analysis of student response data: Machine-learning item response theory. *International Educational Data Mining Society*, 2012.

[9] H. Cen, K. Koedinger, and B. Junker. Learning factors analysis–a general method for cognitive model evaluation and improvement. In *International conference on intelligent tutoring systems*, pages 164–175. Springer, 2006.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[11] R. E. Clark, D. F. Feldon, J. J. van Merriënboer, K. A. Yates, and S. Early. Cognitive task analysis. In *Handbook of research on educational communications and technology*, pages 577–593. Routledge, 2008.

[12] C. Conati and K. VanLehn. A student model to assess self-explanation while learning from examples. In *UM99 User Modeling*, pages 303–305. Springer, 1999.

[13] 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.

[14] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

[15] M. C. Desmarais and R. Naceur. A matrix factorization method for mapping items to skills and for enhancing expert-based q-matrices. In H. C. Lane, K. Yacef, J. Mostow, and P. Pavlik, editors, *Artificial Intelligence in Education*, pages 441–450, Berlin, Germany, 2013. Springer.

[16] M. C. Desmarais and R. Naceur. A matrix factorization method for mapping items to skills and for enhancing expert-based q-matrices. In *International Conference on Artificial Intelligence in Education*, pages 441–450. Springer, 2013.

[17] M. Du, N. Liu, and X. Hu. Techniques for interpretable machine learning. *Communications of the ACM*, 63(1):68–77, 2019.

[18] S. H. Edwards and K. P. Murali. Codeworkout: short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM conference on innovation and technology in computer science education*, pages 188–193, 2017.

[19] B. Fein, I. Graßl, F. Beck, and G. Fraser. An evaluation of code2vec embeddings for scratch. In *In Proceedings of the 15th International Conference on Educational Data Mining (EDM) 2022*, 2022.

[20] S. Guerriero. Teachers' pedagogical knowledge and the teaching profession. *Teaching and Teacher Education*, 2(1):7, 2014.

[21] L. Gusukuma, A. C. Bart, D. Kafura, J. Ernst, and K. Cennamo. Instructional design+ knowledge components: A systematic method for refining instruction. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 338–343, 2018.

[22] M. Hoq, P. Brusilovsky, and B. Akram. Analysis of an explainable student performance prediction model in an introductory programming course. In *Proceedings of the 16th International Conference on Educational Data Mining (EDM) 2023*, 2023.

[23] 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.

[24] K. R. Koedinger, E. A. McLaughlin, and J. C. Stamper. Automated student model improvement. *International Educational Data Mining Society*, 2012.

[25] A. S. Lan, A. E. Waters, C. Studer, and R. G. Baraniuk. Sparse factor analysis for learning and content analytics. *Journal of Machine Learning Research (JMLR)*, 15(57):1959–2008, 2014.

[26] J. Leinonen. Open ide action log dataset from a cs1 mooc. In *Proceedings of the 6th Educational Data Mining in Computer Science Education (CSEDM) Workshop*, 2022.

[27] Y. Mao, F. Khoshnevisan, T. Price, T. Barnes, and M. Chi. Cross-lingual adversarial domain adaptation for novice programming. 2022.

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

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

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

[31] S. Marwan, Y. Shi, I. Menezes, M. Chi, T. Barnes, and T. W. Price. Just a few expert constraints can help: Humanizing data-driven subgoal detection for novice programming. *International Educational Data Mining Society*, 2021.

[32] M. J. Nathan, K. R. Koedinger, M. W. Alibali, et al. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the*

third international conference on cognitive science, volume 644648, 2001.

[33] B. Paaßen, M. Dywel, M. Fleckenstein, and N. Pinkwart. Sparse factor autoencoders for item response theory. In *Proceedings of the 15th International Conference on Educational Data Mining*, pages 17—-26, Durham, UK, 2022. EDM.

[34] Z. A. Pardos, A. Dadu, et al. dafm: Fusing psychometric and connectionist modeling for q-matrix refinement. *Journal of Educational Data Mining*, 10(2):1–27, 2018.

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

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[37] P. Pavlik Jr, H. Cen, and K. Koedinger. Performance factors analysis–a new alternative to knowledge tracing. In *Frontiers in Artificial Intelligence and Applications*, volume 200, pages 531–538, Amsterdam, Netherlands, 01 2009. IOS Press.

[38] P. I. Pavlik Jr, H. Cen, and K. R. Koedinger. Learning factors transfer analysis: Using learning curve analysis to automatically generate domain models. *Online Submission*, 2009.

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

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

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

[42] T. W. Price, D. Hovemeyer, K. Rivers, G. Gao, A. C. Bart, A. M. Kazerouni, B. A. Becker, A. Petersen, L. Gusukuma, S. H. Edwards, et al. Progsnap2: A flexible format for programming process data. In *ITiCSE'20*, pages 356–362, 2020.

[43] 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. Springer, 2017.

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

[45] R. J. Salden, V. A. Aleven, A. Renkl, and R. Schwonke. Worked examples and tutored problem solving: redundant or synergistic forms of support? *Topics in Cognitive Science*, 1(1):203–213, 2009.

[46] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, pages 27–43, 2022.

[47] R. Schmucker, J. Wang, S. Hu, T. Mitchell, et al. Assessing the knowledge state of online students-new data, new approaches, improved accuracy. *Journal of Educational Data Mining*, 14(1):1–45, 2022.

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

[49] 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*, 2022.

[50] Y. Shi, Y. Mao, T. Barnes, M. Chi, and T. W. Price. More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code. In *EDM'21*, 2021.

[51] Y. Shi and T. Price. An overview of code2vec in student modeling for programming education. *MMTC Communications-Frontiers*, 2022.

[52] 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.

[53] G. S. Snoddy. Learning and stability: a psychophysiological analysis of a case of motor learning with clinical applications. *Journal of Applied Psychology*, 10(1):1, 1926.

[54] P. Sztajn, J. Confrey, P. H. Wilson, and C. Edgington. Learning trajectory based instruction: Toward a theory of teaching. *Educational researcher*, 41(5):147–156, 2012.

[55] S. Yang, X. Liu, H. Su, M. Zhu, and X. Lu. Deep knowledge tracing with learning curves. In *2022 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 282–291. IEEE, 2022.

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

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

[58] Y. Zhang, J. D. Pinto, A. X. Fan, L. Paquette, et al. Using problem similarity-and orderbased weighting to model learner performance in introductory computer science problems. *Journal of Educational Data Mining*, 15(1):63–99, 2023.