Using Problem Similarity- and Order-based Weighting to Model Learner Performance in Introductory Computer Science Problems

Yingbin Zhang
South China Normal University
yingbinzhang25@hotmail.com

Juan D. Pinto
University of Illinois at Urbana-Champaign
jdpinto2@illinois.edu

Aysa Xuemo Fan
University of Illinois at Urbana-Champaign
xuemof2@illinois.edu

Luc Paquette
University of Illinois at Urbana-Champaign
lpaq@illinois.edu

The second CSEDM data challenge aimed at finding innovative methods to use students' programming traces to model their learning. The main challenge of this task is how to decide which past problems are relevant for predicting performance on a future problem. This paper proposes a set of weighting schemes to address this challenge. Specifically, students' behaviors and performance on past problems were weighted in predicting performance on future problems. The weight of a past problem was proportional to its similarity with the future problem. Problem similarity was quantified in terms of source code, problem prompts, and struggling patterns. In addition, we considered another weighting scheme where past problems were weighted by the order in which students attempted them. Prior studies have used problem similarity and order information in learner modeling, but the proposed weighting schemes are more flexible in capturing problem similarity on various problem properties and weighting various behaviors and performance information on past problems. We systematically investigate the utility of the weighting schemes on performance prediction through two analyses. The first analysis found that the weighting schemes based on source code similarity, struggling pattern similarity, and problem order improved the prediction performance, but the weighting scheme based on problem prompts did not. The second analysis found that the weighting scheme allows a simple and interpretable model, such as logistic regression, to have performance comparable to state-of-the-art deep-learning models. We discussed the implications of the weighting schemes for learner modeling and suggested directions for further improvement.

Keywords: learner modeling, programming trace, problem similarity, knowledge tracing, performance prediction

1. INTRODUCTION

The application of artificial intelligence in education (AIED) has the potential to address some long-existing challenges, such as "mentors for every learner" and "lifelong and lifewide learning" (Woolf et al., 2013). AIED can offer learners intelligent learning environments that are affectively sensitive and provide personalized support. The learner model is one of the key components of AIED (Luckin et al., 2016), which contains information about the learner, such as their knowledge and emotional states. The second CSEDM data challenge[1] presented a learner modeling task where participants used students' programming problem-solving data in the first half of an introductory computer science course to predict performance on problems in the second half (Section 2.1 provides more details). The challenge is how to decide which problems in the first half of the course are relevant for predicting performance on a problem in the second half. Treating all problems in the first half the same is not reasonable because some of them may not involve the knowledge and skills that are necessary for solving the problem in the second half.

The most common practice to address such a challenge is perhaps knowledge tracing (Corbett and Anderson, 1994), which has been shown to effectively enhance learning when integrated into learning systems (Aleven, 2010; Anderson et al., 1995). Knowledge tracing typically entails explicitly defining knowledge components (KC) and mapping KCs to problems. The mapping is used as a de facto way of determining which prior problems are relevant for predicting performance on a future problem. Nevertheless, the process of developing a high-quality mapping between KCs and problems is a time and effort-consuming process.

We address the second CSEDM challenge by proposing a set of weighting schemes based on problem similarity. Our approach is based on the assumption that, when predicting students' performance in a future problem, past problems that are similar to the future problem will be more predictive than past problems dissimilar to the future problem. In addition, we consider another weighting scheme in which past problems are weighted based on the order in which students attempted them. This weighting scheme aims at accounting for the decay impact of past problems: the earlier a problem was attempted, the less impact it had on the performance in a future problem (Gong et al., 2011). Utilizing information about problem similarity and order is common in learner modeling, but methods that do so are (1) typically applying the information to weight a limited number of features (e.g., response correctness), and (2) some of them are quite complex and require significant computing power, especially for deep-learning-based models. By contrast, our approach allows applying the information about problem similarity and order to weight various features in a simpler way, and it can be implemented in a model as simple as logistic regression.

This paper systematically investigates the impact of various weighting schemes on prediction performance through two analyses. The first analysis compares models with and without weighting to investigate to what extent weighting could improve the predictive performance and which weighting scheme performed better. The second analysis compares the weighting schemes with the state-of-the-art (SOTA) learner modeling methods to investigate

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

whether using problem similarity and order in a simple way could achieve the same predictive performance.

1.1. PROGRAMMING PROCESS ANALYSIS

Researchers have used programming traces to explore programming behaviors and proficiencies and predict future course performance. One well-known metric based on programming traces is the Error Quotient (Jadud, 2006a). The Error Quotient characterizes the extent to which a student struggles with syntax errors while solving a programming problem (Jadud, 2006a). Its computation has four steps: (1) generate pairs of consecutive compilation events on a problem; (2) assign a score to each pair based on an algorithm; (3) divide the score by 11, which is the maximum possible value (the maximum value is 9 in a more complex version; Jadud, 2006b); (4) compute the average of the normalized scores of all pairs. The algorithm in step 2 penalizes the student if they consecutively make the same errors. A pair of compilation events gets a score of 0 if at least one event does not end with any error, 8 if the two events have different types of errors, and 11 if both events have the same error type. Thus, a student with a high Error Quotient struggles with the problem more than a student with a low Error Quotient. Studies have found that Error Quotient predicts course performance (Jadud, 2006b; Tabanao et al., 2011).

Error Quotients focus on students' compilation behaviors and do not consider how students handle the semantic errors of a program (Carter et al., 2015). The Normalized Programing State Model (NPSM) addressed this limitation by modeling students' programming activities in terms of the change in both syntactic and semantic correctness (Carter et al., 2015). It divides students' program states into eleven categories based on syntactic and semantic correctness. For example, a program with unknown semantic correctness might be in a state that was edited to be syntactically correct by the student. This state might switch to a state of execution with the debugger on or off. The proportion of time in each state was used as a feature predicting course performance. The results showed that NPSM had better predictive power on assignment grades and final course grades than the Error Quotient. For a detailed overview of these and other related metrics, see Villamor (2020).

Based on the above metrics and other programming process studies, we engineered a set of features related to programming errors, behaviors, and debugging (see Table 3 in Section 2.2). The computation of our features was not the exact same as these metrics due to the properties of the dataset of the second CSEDM data challenge. For example, in the dataset, a submission was always executed with the debugger off, and the time that students worked on a submission or problem could not be calculated accurately.

1.2. KNOWLEDGE TRACING AND PERFORMANCE PREDICTION

Advances in the modeling of student knowledge have come about as the practice of knowledge tracing or inference, which has improved in recent decades, largely driven by the development of student models in intelligent tutoring systems (ITS; Shute and Zapata-Rivera, 2012). Knowledge tracing refers to the process of identifying what a learner knows at any one time. The most well-known knowledge tracing algorithm is Bayesian Knowledge Tracing (BKT), which uses a hidden Markov model that estimates student knowledge as a latent variable and updates these estimates based on continuous feedback in the form of student performance on future tasks (Corbett and Anderson, 1994). While also typically considered a form of knowledge tracing, Performance Factors Analysis (PFA; Pavlik Jr et al., 2009),

another popular approach, concerns itself solely with using prior performance to directly predict future performance rather than estimating latent student knowledge explicitly. One common trait of some knowledge tracing and performance prediction models is the necessity of explicitly defining the KCs—including skills, facts, or concepts—required to complete each step or problem correctly. KC labels are meant to capture how students develop different skills at different rates, and therefore success on a particular problem is dependent on having developed the requisite skill for that problem. Skills unrelated to the problem have no direct bearing on the probability of success. Ignoring this fact—that is, weighting all past problems the same, regardless of their dissimilarity to the future problem for which success is being predicted—means treating related and unrelated skills the same, causing the model to retain much irrelevant information. In other words, KC labels help us to weight past problems differentially when predicting performance on future problems.

However, developing high-quality KC labels can be an arduous task requiring subject domain expertise and careful iterative refinements. The KC labels have such profound implications for the accuracy of the knowledge tracing algorithm that specific techniques have been devised for evaluating the KC labels themselves (Cen et al., 2006; Stamper et al., 2011). As an alternative, modern techniques, such as deep learning knowledge tracing (DLKT; Sarsa et al., 2022), compensate for the lack of KC labels by automatically identifying the relationship between problems and weighting prior performance and behaviors based on this relationship rather than KC labels. Specifically, Deep Knowledge Tracing (DKT; Piech et al., 2015) and other recurrent-neural-network-based approaches—such as Dynamic Key-Value Memory Networks (DKVMN; Zhang et al., 2017) —model student performance as a sequence of successes or failures, carrying pertinent information from past problems into the prediction of future problems. Self-attention-based models such as Self-Attention Knowledge Tracing (SAKT; Pandey and Karypis, 2019) or Separated self-AttentIve Neural knowledge Tracing (SAINT; Choi et al., 2020) instead follow a feed-forward architecture that searches for similarities among problems in parallel, while still preserving order information through the use of positional embeddings. The primary advantage of all these DLKT techniques is that they allow for a very large hypothesis space wherein relationships between problems can be automatically identified. While KCs can still be explicitly defined in these models—along with additional features in models such as Exercise-aware Knowledge Tracing (EKT; Liu et al., 2021) or Neural Pedagogical Agent (NPA; Lee et al., 2019)—the inputs are commonly kept as simple pairs of *{problem ID, correctness}*. Bypassing manual KC mapping and feature engineering hands the reins to the network's backpropagation algorithm, which then takes care of identifying appropriate weights based on relationships in the input data. The high representational power of these models, however, comes at the cost of complexity and opaqueness.

Within the realm of performance prediction for computer science (CS) education, approaches have included temporal-pattern-based approaches such as Recent Temporal Patterns (RTP; Mao et al., 2019), as well as the Additive Factor Model (AFM; Yudelson et al., 2014), which is a member of the Item Response Theory (IRT) family. Some recent work has relied on DLKT for programming performance prediction (e.g., Shi et al., 2022; Wang et al., 2017), which tends to have high predictive performance but is difficult to interpret.

1.3. ITEM SIMILARITY IN EDUCATION

The similarity of educational items has many applications, such as automatic recommendation and student and domain modeling. The basic procedure of applying item similarity consists of

three steps (Pelánek, 2020): choosing data of items, computing the similarity matrix, and application. The first step is deciding the input data of items or which item property the similarity is about. We can measure item similarity in terms of item statement and metadata (e.g., the KC of an item), item solutions, and item performance (Pelánek, 2020). Taking programming problems as an example, the item statement is the prompt that explains the problem requirement. Item solutions can be the sample code provided by the instructor or the code submitted by students. Item performance can be the correctness of students' code, the number of students' attempts, and the time on the problem. The second step is to compute the pairwise item similarity matrix. There are a variety of measures for each type of input data (Cechák and Pelánek, 2021). For example, when the input data is solution code, researchers may first represent the code with a set of vectors via natural language processing techniques (e.g., bag-of-words models; Pelánek et al., 2018). Based on two problems' representation vectors, their similarity may be quantified by the Euclidean distance, cosine similarity, etc. The choice of input data has been found to be more important than the choice of similarity measures (Cechák and Pelánek, 2021; Pelánek et al., 2018). The final step is manipulating the similarity matrix for application. For example, with the similarity matrix, we can find the nearest neighbors of the item that a student just failed and recommend one neighbor item to the student (Pelánek, 2020).

Item similarity has been used in learner modeling but is typically referred to as item relations. For methods that entail explicit mapping between items and KCs, such as BKT and PFA, items' relations are measured in terms of metadata, i.e., their common KC. For methods that do not require an explicit mapping, such as DKT and SAKT, items' relations are typically captured based on item performance data (Pelánek, 2020). Recent developments in DLKT have started taking advantage of other types of data to capture item relations. For example, code-DKT extends DKT by using both code correctness and the representation of code content (Shi et al., 2022). That is, the item relations in code-DKT are measured by both item solution and performance data. Note that these models do not use item relations in a way aligning with the three-steps approach because they do not use an explicit similarity matrix, and item relations are computed implicitly during the training of model parameters. In Pelánek's (2020) words, these techniques capture item relations using a "model-based approach," while the three steps capture item relations using an "item similarity approach." Examples of other "model-based approaches" are item response theory (IRT) models (Embretson and Reise, 2000), canonical correlation analysis (Sahebi and Brusilovsky, 2018), and tensor factorization (Zhao et al., 2020).

1.4. THE RATIONALE BEHIND SIMILARITY WEIGHTING AND THE CURRENT STUDY

The current study proposes a way of using the "item similarity approach" (Pelánek, 2020) to capture item relations and then applying the relation information to learner modeling in the context where an explicit mapping between KCs and problems is lacking. As mentioned earlier, the KC mapping helps us to weight past problems differentially when predicting performance on future problems. Similarity weighting serves as an alternative to this mapping. Prior studies have used similarity weighting implicitly through the "model-based approach" (e.g., Piech et al., 2015; Pandey & Karypis, 2019; Zhao et al., 2020). The current study uses similarity weighting explicitly. Nevertheless, the basic rationale is the same: a past problem's weight to a future problem is proportional to their similarity. For a future problem where students' performance is to be predicted, students' behaviors and performance on a past problem that is similar to it are more indicative of its requisite knowledge than behaviors and

performance on a past problem dissimilar to it. Thus, behaviors and performance on past similar problems are more important in predicting performance on the future problem and should be assigned larger weights than dissimilar problems. Note that similarity is not limited to KCs. Two problems that do not involve the exact same KCs may still be similar because their KCs may be related to each other. For instance, iteration can be dependent on conditionals. Performance on a problem about conditionals may not rely on knowledge about iteration, but performance on a problem about iteration relies on knowledge about conditionals. Thus, the two problems are still similar, and performance on one problem is useful in predicting performance on another. Analogously, some complex KCs may only exist in problems later in a course because the instructor may not introduce these KCs until students master the prerequisite KCs. However, performance on problems about the prerequisite KCs is still useful in predicting performance on problems about the complex KCs.

The proposed weighting method differs in how it uses item similarity from the "model-based approach" for learner modeling. As mentioned before, the "model-based approach" uses item similarity to weight past problems implicitly while training the prediction model. For example, in DKT and SAKT (Pandey and Karypis, 2019; Piech et al., 2015), the various matrices (e.g., projection and embedding matrices) in the neural network function as a way to assign weights to past problems. The final weight of a past problem to a future problem is a non-linear transformation and combination of these matrices, which are iteratively optimized by algorithms like gradient descent. In methods based on matrix or tensor factorization (Sahebi et al., 2016; Zhao et al., 2020), the weight of a past problem to a future problem is determined by the Q matrix, which is a mapping between problems and latent concepts and optimized during model training. In IRT models (Embretson and Reise, 2000), the weight relies on item discrimination coefficients. Specifically, the performance on an item with high discrimination in a dimension impacts the ability estimates of this dimension more than the performance on an item with low dimension discrimination, and the ability estimates influence the probability of succeeding on items with high dimension discrimination more than items with low dimension discrimination. Item discrimination coefficients are optimized during parameter estimation. By contrast, in the current study, the weight of a past problem to a future problem is a linear transformation of their similarity (see Section 2.3.6), derived from one-time computation before training the prediction model. Thus, the proposed weighting method uses similarity weighting explicitly.

The explicit use of similarity weighting prevents optimizing the weights for a prediction task, but it allows experts to specify which problem properties the similarity is based on and what information or features to weight. For example, two problems' similarity in code content or correctness can be used to weight code correctness, content, errors, and the number of attempts on past problems. That is, a single weighting approach can be flexibly applied to all problem-specific features. Some DLKT models, such as the long-short-term-memory-based DKT, can also weight any features, but our approach can be more interpretable because the weighted features can be used by simpler machine learning (ML) models (e.g., logistic regression).

We propose three similarity weighting schemes, each based on one of Pelánek's (2020) three input data types (see Sections 2.3.2 to 2.3.4): problem statement, problem solutions, and problem performance. In addition to weighting problems by similarity, we also follow existing learner modeling techniques (e.g., PFA-decay model) in weighting problems by the order in which students attempted them to account for the decay impact of past problems. The rationale is that behaviors and performance on a recent problem are more indicative of their current knowledge state than behaviors and performance on an earlier problem (see Section 2.3.5 for

details). The four weighting schemes were applied when aggregating information over past problems to feature vectors (e.g., the proportion of solved problems; see Section 2.3.1). With these feature vectors as input, ML models working with tabular data (e.g., logistic regression) can access information about problem relations and attempt order, which are two types of information that these methods typically have difficulty using but which DLKT can exploit (Gervet et al., 2020). Thus, to some extent, the weighting schemes show a framework for using problem similarity and order weighting to engineer useful features and contribute to feature engineering in performance prediction.

The current study evaluates the proposed weighting schemes by comparing their prediction performance to the model without weighting and the models using weighting via the "model-based approach" in the task of the second CSEDM data challenge. In addition to the proposed weighting schemes and empirical evaluations, another contribution of this study is comparing the relative importance of various programming trace features in programming prediction.

2. METHOD

2.1. DATA

The data was collected from an introductory computer science (CS1) course and provided by the second CSEDM data challenge. The CS1 course requested students to complete five assignments that were released one by one. Each assignment contained ten problems, but problems within the same assignment did not necessarily involve the same programming knowledge. Problems within an assignment were released simultaneously, had the same deadline, and could be completed in any order.

The data challenge included two phases. Both phases asked participants to solve the same tasks but used different training data. In the within-semester phase, the training data included 373 students in the CS1 course in the Spring of 2019 and 367 students in that course in the Fall of 2019 (see Table 1), while the training data in the cross-semester phase only included 247 students of the Spring 2019 sample. The test data in both phases were a subset of 123 students who participated in the CS1 course in the Fall of 2019.

Table 1: Training and test data in cross-semester and within-semester phases.

Phase	Cross-semester	Within-semester
Training data	Spring 2019 sample one (247 students)	Spring 2019 samples one (247 students) and two (126 students) + Fall 2019 training sample (367 students)
Test data	Fall 2019 test sample (123 students)	

Each phase contained two tracks with different prediction tasks. This paper focuses on track 1, which asked participants to predict whether students would struggle with each of the 20 later problems in the CS1 course based on their submission traces on the 30 early problems. Struggling was defined as not successfully solving a problem or solving the problem with more submissions than 75% of their peers.

Table 2 displays the distribution of struggling in the 20 later problems in various samples. Note that each observation is a unique combination of a student and one of the 20 later problems. As such, the total number of observations is around 20 times the number of students in the dataset.

Table 2: The distribution of labels in various samples.

Student-problem observations	Struggling	Not struggling
Spring sample one	1,084 (25.80%)	3,117 (74.20%)
Spring sample two	388 (25.68%)	1,123 (74.32%)
Fall training	1,759 (25.05%)	5,262 (74.95%)
Fall test	633 (26.77%)	1,732 (73.23%)

The submission traces on problems contained information about syntax errors, test scores, and source code of each submission. The test score, ranging from 0 to 1, depended on whether the submission had syntax errors and how many tests it passed. If the submission had at least one syntax error or passed no test, the test score would be 0. If it had no syntax error and passed at least one test, the test score would be larger than 0. A score of 1 meant the submission passed all tests and was correct. Thus, if a submission without syntax error had a score smaller than 1, it meant that it failed at least one test and had semantic errors. The number of tests varied across problems.

2.2. FEATURE ENGINEERING

A total of 65 features were computed on the submission traces based on the script provided by the data challenge organizer (five features), from the winning entry in the first CSEDM data challenge (four features; Natti and Athrey, 2019), from the literature on programming process analysis (33 features), and from our feature engineering for the competition (23 features).[2] These features either characterize students or the 20 later problems (see Table 3). Students' characteristics include four categories: basic (e.g., the percentage of the 30 early problems that students struggled with), error-related features (e.g., the percentage of submissions on the early problems that contain syntax errors), debugging-related features (e.g., the average difference in test scores between two consecutive submissions), and others (e.g., the number of days with at least one submission). Table 3 indicates the sources of these features. For simplicity, some rows include multiple features, where the number of features is indicated in parentheses. For example, row 20 contains two features, the change in the percentage of problems that students struggled with from the first to the third assignments and such change in the percentage of problems that students solved. We argue that such changes may represent the growth in students' knowledge or skills, and the variation in the changes may represent differences in learning rates. The appendix presents the details of other features.

We also computed three student-problem interaction features to capture the difference between students' ability and problem difficulty. These features are inspired by psychometrics, specifically, the Rasch model (von Davier, 2016), where the probability that an examinee answers a problem correctly relies on the difference between their ability and the problem difficulty. For example, in the first student-problem interaction feature, *the percentage of problems that a student did not struggle with* may be viewed as the student's ability, while *the percentage of students that struggled with the problem* may be viewed as the

[2] The code for this paper is public in Github: https://github.com/yingbinz/JEDM-similarity-weighting

problem difficulty. While some models, such as ensemble trees, can implicitly capture feature interactions, we explicitly computed such interaction features to ease their interpretation.

In addition, we computed two sets of features aimed at capturing the linguistic information of students' code submissions. We first used the term frequency-inverse document frequency (TF-IDF) to compute linguistic features. This is inspired by natural language processing (NLP; Jurafsky and Martin, 2009), where information systems use TF-IDF features to retrieve documents on relevant topics. In our case, we treated each student's final submission on each problem as a natural language document and each token as a word. In this way, we obtained a 50-element embedding vector for each submission using the Scikit-learn Python library. However, the elements are zeros in most code submissions and have little variation. We selected elements that were not zero in at least 10% of submissions as features in our model. Fifteen elements met this criterion.

One major limitation of using TF-IDF and similar NLP approaches on code is that they explicitly capture linguistic information but ignore the structure of code. For this reason, we also applied code2vec to the source code of each submission and obtained a 384-element embedding vector for each submission. The details of this implementation are described in Section 2.3.2. We then performed dimensionality reduction using principal component analysis to reduce the 384-element embedding vector to a set of 16 features.

Among the 65 features in Table 3, 52 would be weighted if weighting schemes were used and were marked in gray. We did not weight features #1, #18, #19, and #21 because they were more about students' effort or problem-solving behaviors rather than performance. We did not weight features #4 and #20 because weighting was not applicable to them. For example, the median of weighted submissions might not be meaningful. Features #9 and #10 were not weighted because features highly related to them were weighted (e.g., #11 and #12). Problem features could not be weighted.

Table 3: The categories and sources of features.

	Basic features		**Source**
Student	1	# problems that a student attempted	Organizer
	2	% problems that a student solved eventually	
	3	% problems that a student solved on the first submission	
	4	Median and max submissions on early problems (2)	
	5	Mean submissions on early problems	Engineering
	6	% problems that a student struggled with	
	Error-related features		
	7	% problems on which a student made syntax errors	1st data challenge
	8	% problems on which a student made semantic errors	
	9	% submissions that contain syntax errors	Carter et al. (2015)
	10	% submissions that contain semantic errors	
	11	Average number of syntax errors on a problem	Becker et al. (2016)
	12	Average number of unique test scores on a problem	Engineering

	Debugging		
	13	Rate of fixing syntax errors	Jadud (2006a)
	14	Rate of making new syntax errors	
	15	Rate of improving test scores	Engineering
	16	Average difference in test scores between two consecutive submissions	
	17	Transition strength between code states (6)	Carter et al. (2015)
	Others		
	18	% problems that a student made a submission less than 15s	Pinto et al. (2021)
	19	Number of days with at least one submission	Yeckehzaare et al. (2022)
	20	Change in rows 2 and 6 from the first to the third assignments (2)	Engineering
	21	The number of lines added, deleted, and modified	Baumstark and Orsega (2016)
	22	TF-IDF vectors (15)	Engineering
	23	Principal components of code2vec vectors (16)	Shi et al. (2021)
Problem	24	% students that struggled with the problem	Engineering
	25	% students that made syntax errors on the problem	1st data challenge
	26	% students that made semantic errors on the problem	
Student-problem interaction	27	% problems that a student did not struggle with - % students that struggled with the problem	Rasch model (von Davier, 2016)
	28	% problems on which a student did not make syntax errors - % students that made syntax errors on the problem	
	29	% problems on which a student did not make semantic errors - % students that made semantic errors on the problem	

Note. #: The number of. %: The percentage of. Engineering: features are engineered. Features marked in gray would be weighted if weighting schemes were used.

2.3. WEIGHTING SCHEMES

The proposed weighting schemes are designed to adjust the contribution of students' interaction with each of the 30 early problems to the performance prediction on each of the 20 later problems. Three weighting schemes were based on similarity, and each corresponded to one of Pelánek's (2020) three input data types: item statement (problem prompt in this study), item solutions (source code in this study), and problem performance (struggling in this study). The fourth weighting scheme was based on problem order. In this section, we first provide information about how weights were applied when computing the value of each feature. Then, we present the specifics of each weighting scheme used in this study.

2.3.1. Application of weights

Here we present the way in which weights are applied when computing features. Figure 1 illustrates the hierarchical levels of features and where we used the weights. Features at the submission level (denoted as *S*, e.g., the number of syntax errors a submission contains) were aggregated to features at the early problem level (denoted as *EP,* e.g., the number of syntax errors that a student made on a certain early problem), which were further aggregated to features at the student-late problem level (denoted as *LP*, e.g., the average number of syntax errors that a student made on early problems). LP_m represents the value of a *LP* feature that is used in predicting performance on later problem *m*. Weights are used in the aggregation from *EP* features to *LP* features, as illustrated by equation (1):

$$LP_m = \frac{1}{N}\sum_{n=1}^{N} w_{nm} EP_n \tag{1}$$

N is the number of early problems that a student attempted. w_{nm} is the normalized weight of early problem *n* for late problem *m*, and $\sum_{n=1}^{N} w_{nm} = N$. EP_n is the feature value on early problem *n*. If the aggregation is summing, $\frac{1}{N}$ needs to be removed from equation (1). Not using weights is a special case of equation (1), where w_{nm} is always 1 regardless of *n* and *m*. Consequently, $LP_1 = LP_2 = \cdots = LP_{20}$.

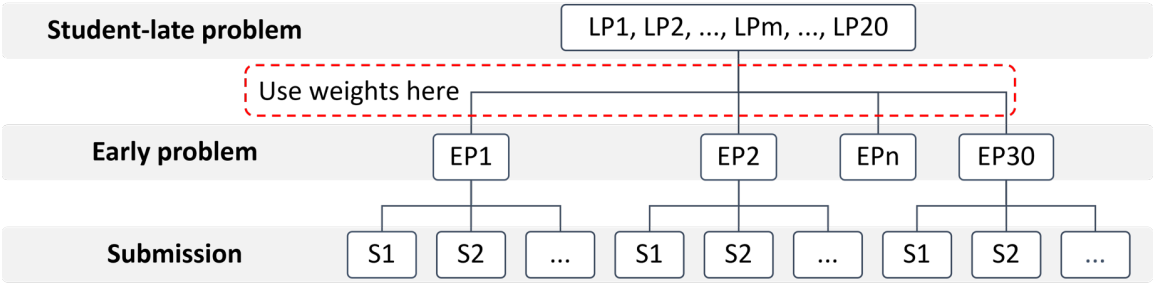

Figure 1: The hierarchical levels of features and where weights are used.

As an example, we will focus on the feature *proportion of problems that a student struggled with* to predict performance on late problem 43. Table 4 shows three students' responses to five early problems. Students A and B struggled with three problems, and their unweighted features were 0.6. However, B's weighted feature was almost twice A's weighted feature. This is because A struggled with problem 1, which had a low similarity with problem 43 and a small weight, and B struggled with problem 3, which had a high similarity with problem 43 and a large weight. Similarly, student C struggled with two problems, but C's weighted feature was larger than A's because C struggled with problems 3 and 13, both of which had large weights.

Alternatively, features at the submission level can be directly aggregated to the student-late problem level. In such cases, weights are used via equation (2).

$$\mathrm{LP}_m = \frac{1}{T}\sum_{n=1}^{N}\left(w_{nm}^{*}\sum_{t=1}^{T_n} S_{nt}\right) \tag{2}$$

Table 4: Three students' unweighted and weighted features for predicting performance on late problem 43.

Student	Early problem *n*	1	3	5	12	13	LP_{43}
	Similarity	0.02	0.51	0.31	0.22	0.32	
	Weight w_{n43}	0.07	1.86	1.14	0.79	1.14	
A	Original response EP_n	1	0	1	1	0	0.60
	Weighted	0.07	0.00	1.14	0.79	0.00	0.40
B	Original response EP_n	0	1	1	0	1	0.60
	Weighted	0.00	1.86	1.14	0.00	1.14	0.83
C	Original response EP_n	0	1	0	0	1	0.40
	Weighted	0.00	1.86	0.00	0.00	1.14	0.60

Note. Original response: 1 represents struggling, and 0 represents success.

T_n is the number of submissions on early problem n. T is the total submissions on all early problems, i.e., $\sum_{n=1}^{N} T_n$. S_{nt} is the feature of submission t on early problem n. w_{nm}^* is the normalized weight of submissions on early problem n for late problem m and $\sum_{n=1}^{N} T_n w_{nm}^* = T$. It is derived by normalizing w_{nm}:

$$w_{nm}^* = \frac{T}{\sum_{n=1}^{N} T_n w_{nm}} w_{nm} \tag{3}$$

With equation (3), submissions on the same early problem have the same weight for a late problem, but submissions on different early problems have different weights. Moreover, the ratio of the weight of a submission on early problem P to that on early problem Q is $\frac{w_{Pm}}{w_{Qm}}$ for late problem m, which is the same as the ratio of the weight of early problem P to early problem Q.

Equation (2) is suitable for features that require many submissions to generate reliable values. For instance, obtaining a reliable measurement of transition strength between code states may require 50 or more submissions (Bosch and Paquette, 2021), but in most cases, a student made far less than 50 submissions on a single problem. Thus, computing the transition strength between code states at the early-problem level would be improper, while using equation (2) to compute transition strength directly is more appropriate. All debugging-related features were weighted using equation (2), and the others were weighted using equation (1).

2.3.2. Weighting by source code similarity

In this weighting scheme, the weights were based on the similarity between two problems based on the source code of submissions on these problems. We used code2vec (Alon et al., 2019) to convert students' code to embeddings that retained some of the semantic properties of the code. We then used the similarities between the average embeddings as weights.

Code2vec emulates the success of distributed vectors of words (such as word2vec) in natural language processing (NLP) tasks but uses an approach that retains the semantic properties of code. It does this by extracting a series of leaf-to-leaf paths from an abstract syntax tree (AST) representation of the code and then using these paths as inputs in an artificial neural network (ANN) designed to predict method names in the code. Once the model is trained, it can be used to convert any code snippet into a vector embedding representation. Code2vec has previously been used in CS education research, including to

profile students based on their code (Azcona et al., 2019), to discover student misconceptions (Shi et al., 2021), and to automatically detect bugs in student code (Shi et al., 2021).

To calculate these weights, we first identified a single submission from each student-problem pair to convert to a code embedding. If a student successfully solved the problem, we used the code from their final correct submission. Otherwise, we used their final submission which had no compiler errors. The latter requirement is because we could not extract an AST from uncompilable code, making it also impossible to convert it to an embedding. We then ran the code snippets from each of these selected submissions through code2vec, converting the code to an AST, extracting paths from the AST, then using the path as input for the neural network trained to predict method names, and finally extracting the embedding used by the neural net in its prediction task. We used the exact neural network configuration described by Alon et al. (2019). The network was trained for eight epochs on 14 million examples of Java code snippets found on public GitHub repositories.

Once we had all code embeddings for our training data, we calculated the similarities between them in three different ways: inverse Euclidean distance, cosine similarity, and Pearson's correlation. We measured the inverse Euclidean distance between embeddings using $\frac{1}{\sqrt{\sum_{i=1}^{n}(LE_i - EE_i)^2}}$ where LE_i and EE_i are the i-th element in the late problem embedding and in the early problem embedding, respectively. We calculated the cosine similarity between embeddings using $\frac{LE \cdot EE}{\|LE\|\|EE\|}$ where $\|LE\|$ is the magnitude or Euclidean norm of the late embedding vector—that is, $\sqrt{\sum_{i=1}^{n} LE_i^2}$.

We reasoned that exploring multiple operationalizations of vector similarity would allow us to identify the better option, based on the nature of the embeddings themselves. For example, cosine similarity—which is the typical way of measuring similarities between word2vec vectors—only considers the angle between vectors and works best when the magnitude of a vector is not important. Euclidean distance, on the other hand, is quite intuitive as it measures actual distance, but these distances (and, for our purposes, weights) can become insignificant if the angle between vectors is very small. However, the current study did not find a difference between the three measures. This was in line with prior studies that found that the choice of similarity measures had relatively less impact than the choice of input data (Cechák and Pelánek, 2021; Pelánek et al., 2018). Thus, the result section only reports cosine similarity. The appendix presents the results of Euclidean distance and Pearson correlation.

2.3.3. Weighting by problem prompt similarity

An alternative way to identify similarities between problems is to compare the language in the problem prompts (Pelánek, 2020). To do this, we extracted a document embedding for each prompt using doc2vec (Le and Mikolov, 2014). Doc2vec builds on word2vec by adding an additional document vector that allows the model to create a document embedding (as opposed to a word embedding) that aims to capture the semantic concepts that make up the document. We made use of the doc2vec implementation found in the Gensim library for Python (Rehurek and Sojka, 2010).

The specific process we used to calculate weights began by tokenizing the problem prompts and removing typical stop words. We then trained a doc2vec model with tagged problem prompts over 120 epochs. We used this model to convert the text of each problem prompt into a 50-element document vector. Finally, we calculated the similarity between each late problem

prompt and each early problem prompt using the same three measures of difference as with the code embeddings—inverse Euclidean distance, cosine similarity, and Pearson's correlation. Again, there was no difference among the three measures in prediction performance. Thus, the result section only reports cosine similarity, and the appendix presents the results of the others.

2.3.4. Weighting by struggling similarity

The current weighting scheme used the correlation between the students' performance in two problems as the similarity metric. The rationale is if the knowledge and skills involved in two problems are related, a student performing well on one problem is likely to perform well on another. Thus, the correlation between students' performance in the two problems should be stronger than the correlation between two problems involving unrelated knowledge and skills. Performance in a problem can be measured using various indicators—for example, whether a student struggled with the problem, whether they made syntax errors in the problem, how many syntax errors they made in the problem, etc. Multiple performance indicators can be used to compute multiple similarity metrics, which could be used to weight different features. For instance, the similarity in struggling could be used to weight the proportion of problems a student struggled with, while the similarity in syntax errors could be used to weight the proportion of problems on which students made syntax errors. The current study only used the similarity in struggling, given that the goal of the task is predicting whether a student will struggle with a problem or not.

Struggling with a problem is a binary feature, so contingency table correlation is used. Specifically, we used the log-odds ratio because it ranges from -∞ to +∞ and approximates a normal distribution (Dagne et al., 2002), making it suitable for statistical analysis. Thus, the similarity between early problem *n* and late problem *m* is their log-odds ratio, which is $ln\frac{\left(a+\frac{1}{2}\right)/\left(b+\frac{1}{2}\right)}{\left(c+\frac{1}{2}\right)/\left(d+\frac{1}{2}\right)}$. *ln* is the natural logarithm, and Table 5 explains *a, b, c*, and *d*. For example, *a* is the number of students struggling with both problems. ½ is used to reduce the bias in the log-odds ratio estimate (Dagne et al., 2002). $\frac{\left(a+\frac{1}{2}\right)}{\left(b+\frac{1}{2}\right)}$ is the odds of struggling with late problem *m* in students struggling with early problem *n*, while $\frac{\left(c+\frac{1}{2}\right)}{\left(d+\frac{1}{2}\right)}$ is the odds of struggling with late problem *m* in students succeeding on earl problem *n*. A log-odds ratio of zero means that the odds of struggling with late problem *m* were independent of struggling with early problem *n* or not. A log-odds ratio larger than one means a positive correlation, while A log-odds ratio lower than one means a negative correlation.

Table 5: Contingency table of struggling in two problems.

Problem n \ Problem m	Struggling	Success
Struggling	*a*	*b*
Success	*c*	*d*

Among the 600 early-later problem pairs, 46 had negative log-odds ratios. A negative log-odds ratio indicates that students who struggled with the early problem were less likely to

struggle with the later problem. Forty-four negative log-odds ratios were small in terms of magnitude (> -0.4), indicating weak associations (Rosenthal, 1996). Such weak negative associations might occur due to randomness. Thus, if early problem *n* and late problem *m* had a negative log-odds ratio, we replaced the negative value with the minimum positive log-odds ratio of early problem *n*. This processing assumed that the interaction with any problem did not contribute to learning negatively, analogous to the assumption in Zhao et al. (2020) that no learning material was negatively related to a concept. The largest two negative log-odds ratios were both from the first early problem (ID 1), with later problems 51 (-0.41) and 64 (-0.55), respectively. This early problem had a negative log-odds ratio with 13 of the 20 later problems. Investigating the cause of negative log-odds ratios is beyond the scope of this paper, but it may be worth future studies.

2.3.5. Weighting by problem order

Unlike previous weighting schemes that address the lack of knowledge components, the current scheme aims to account for the decay impact of past problems (Gong et al., 2011). The decay impact assumes that behaviors and performance on a recent problem are more important than that on an earlier problem in predicting performance on a future problem. For example, compared with students' performance in the earlier part of the 30 early problems, e.g., the first five problems, their performance in the latter part of the 30 early problems, e.g., the last five problems, might more accurately represent their knowledge and skills when they were attempting the 20 later problems. Thus, we assigned lower weights to the earlier parts of the 30 early problems and higher weights to the latter parts. The reason behind the decay impact might be various. It might be that students' programming knowledge and skills grew during the period of the 30 early problems, which was one month, perhaps a significant amount of time for novice CS students. Meanwhile, students might forget the knowledge learned weeks ago. Also, compared to the earlier part of the 30 early problems, the latter part might involve content more similar to the 20 later problems.

For the early problems attempted by a student, we sorted these problems by the timestamp of the student's first submission on each problem. Then, the first problem was assigned a weight of 1, the second problem a weight of 2, and so on until the last problem, which was assigned a weight equal to the number of early problems attempted. Through this weight assignment, we hope to partially account for the learning in programming knowledge that occurs over time.

The weights can be assigned in different ways. For instance, the first problem may have a weight of 2, the second problem may have a weight of 3, and so on. In this weight assigning, if a student attempted all 30 early problems, the ratio of the first problem's weight to the last problem's is 2/31. By contrast, based on the weight assignment that we used, the ratio is 1/30, almost half of 2/31. Thus, our weight assignment implied a stronger decay in the impact of past problems. It is challenging to predetermine which weight assignment method works best, analogous to hyperparameters in ML. Similarly, the best weight assignment may be determined by cross-validation (CV). Indeed, we experimented with a few ways of assigning weights, where the ratio of the first problem's weight to the last problem's ranged from 1/60 to 1/3. The ratio of 1/30 worked best in terms of the average CV AUC in the training data, though the differences were small (~0.001 AUC).

2.3.6. Normalizing weights and combining different weighting schemes

Weights derived from the above computation are unnormalized, denoted by w'_{nm}. w'_{nm} cannot be used to weight features directly because it may cause unreasonable values in weighted features. For example, for the proportion of problems that a student struggled with, using w'_{nm} may result in a weighted proportion beyond 100%. Equation (4) normalizes w'_{nm}:

$$w_{nm} = \frac{N * w'_{nm}}{\sum_{n=1}^{N} w'_{nm}} \tag{4}$$

Different weighting schemes can be used together. When multiple weighting schemes are used, we take the mean of their normalized values as the final weight:

$$w_{nm} = \frac{1}{S}\sum_{s=1}^{S} w_{snm} \tag{5}$$

w_{snm} is the weight of early problem n for late problem m in weighting scheme s. S is the number of weighting schemes used. Note that taking the mean of different weighting schemes might not be a good choice because weighting schemes that were not useful might damage the performance of the average weighting. We return to this point in Section 4.2.

2.4. MODEL BUILDING

This study investigated the effectiveness of the weighting schemes in two analyses. The first analysis compared different weighting schemes to investigate to what extent the weighting could improve predictive performance and which weighting scheme performed better. The second analysis compared the weighting schemes with the SOTA learner modeling methods to investigate whether using problem similarity and order in a simple way could achieve the same predictive performance. Both analyses were conducted for cross- and within-semester prediction. Note that all weights were learned only from the training dataset. More specifically, the weights in cross-semester prediction were from Spring sample one as this is the training dataset in this phase. Similarly, the weights in within-semester prediction were from Spring samples one and two as well as the Fall training dataset. The evaluation metric was AUC because it was the criterion of the CSEDM data challenge.

2.4.1. Comparing different weighting schemes

We used three machine learning models: extreme gradient boosting (through the XGBoost Python library), random forest (through the Scikit-learn Python library), and lasso logistic regression (also through the Scikit-learn library). We used a 5-fold CV within the training dataset and grid search to tune the hyperparameters. CV was conducted at the student level, i.e., different folds contained different students. We first tuned the hyperparameters in the models with unweighted features, and then we applied the tuned hyperparameters to the models with weighted features. Table A-2 in the Appendix displays these hyperparameters.

We first compared the performance of models with and without weighting schemes with all features in Table 3. Because engineering all these features was effortful, we investigated the difference between models with and without weighting schemes when fewer features were used. We varied the number of features from five to all features. The features were selected based on their importance in the model with all features calculated on the training dataset. For example, when the number of features was five, the most important five features in a model

were used. Feature selection was implemented via the *SelectFromModel* function of the Scikit-learn Python library.

We reported the five most important features in the best models (random forest classifier for cross-semester prediction and lasso logistic regression for within-semester prediction) in the condition of all features. For the random forest classifier, the Gini importance was used, and for lasso logistic regression, the standardized coefficients were used. Because some features were highly correlated (e.g., Pearson correlations were around 0.90), we also ranked the importance of a feature based on the AUC of the one-feature model, which only used this feature. The AUC was evaluated via a 5-fold CV in the training dataset, and the average value was used as the importance measure.

2.4.2. Comparing weighting schemes with DLKT and IRT

We chose DLKT and IRT models as our baselines. We selected DLKT because they are SOTA and use problem similarity and order implicitly. We chose IRT models because they account for problem difficulty explicitly and inspired our student-problem interaction features. For our approach, we decided to use only two features that could be computed based on response correctness and problem ID sequences: the proportion of problems that a student struggled with and the proportion of students that struggled with the problem (#6 and #24 in Table 3). We chose the two features because they could be computed only using response correctness and problem ID sequences, which is the information used by the baselines (DLKT and IRT). Thus, with the two features, our approach used the same information for prediction as the baselines, and the comparison between our approach and the baselines might be fairer than using all features listed in Table 3. It is noteworthy that using many features may cause overfitting issues and damage the performance of a model (see Figure 3). Thus, to some extent, using only two features might give our approach an advantage.

The number of combinations of ML models plus weighting schemes was 24 (three ML models * eight weighting approaches; see Figure 2). Comparing all combinations to the baselines might inflate the performance of our approach. Thus, we selected the combination with the best average AUC across a 5-fold CV in the training dataset. The best combination was the lasso logistic regression with struggling similarity weighting in both cross- and within-semester prediction. We compared this combination with the baselines in the test dataset.

DLKT. We used three DLKT models as baselines. We used the LSTM implementation of DKT (Piech et al., 2015) since it was the first DLKT model published and serves as a good starting point for comparison. We also used two self-attention-based models, SAKT (Pandey & Karypis, 2019) and SAINT (Choi et al., 2020). The latter is the SOTA model on the official EdNet (Choi et al., 2020) leaderboard as of this writing, though its successor, SAINT+ (Shin et al., 2021), has reportedly performed marginally better. We wrote our LSTM-DKT implementation using Keras and TensorFlow and used publicly available open-source implementations of SAKT[3] and SAINT[4] written using PyTorch. All three DLKT models used

[3] https://github.com/hcnoh/knowledge-tracing-collection-pytorch/blob/main/models/sakt.py
[4] https://github.com/Nino-SEGALA/SAINT-pytorch

problem ID and correctness as inputs. We tuned the hyperparameters by training the model on 80% of the training data and validating it on the other 20%.

IRT models. We used the two-parameter logistic model (2PL), which accounts for problem difficulty and discrimination explicitly (Embretson and Reise, 2000). We did not use the Rasch model because we did not think that the programming problems had the same discriminative power. Moreover, in the multidimensional context, to some extent, it is through the discrimination coefficients that the 2PL model utilizes problem relations. Indeed, the Rasch model performed worse than the 2PL model in terms of both model fit and AUC in the training dataset. We implemented one- and two-dimensional 2PLs via the *mirt* package (Chalmers, 2012) in *R*. The problem structure of the two-dimensional model was determined based on the factor loading pattern generated by principal factor analysis with *promax* rotation. We regarded a problem related to a latent dimension when its loading on this dimension was larger than 0.2. A problem might be related to both dimensions. The problem-dimension relations were reported in Table A-2 of the Appendix.

3. RESULTS

3.1. COMPARING DIFFERENT WEIGHTING SCHEMES

Figure 2 depicts the test AUC of different weighting schemes when all features were used. Table A-2 in the appendix displays the exact values of train and test AUC. The value of the best test AUC per training dataset per ML model is displayed. Models with source code similarity weighting improved test AUC, compared to no feature weighting, in 5 out of 6 conditions (except for the condition of cross-semester prediction with random forest classifier). However, the magnitude of improvement is small, between 0.001 and 0.005. Problem prompt similarity weighting consistently reduced test AUC (0.003 to 0.009), compared to no weighting.

Problem order weighting improved the test AUC better than source code similarity weighting, except for the condition of cross-semester prediction with lasso logistic regressions. Compared with models without weighting, models with problem order weighting improved test AUC by 0.003 to 0.009. Struggling similarity weighting achieved better improvement, with a magnitude of 0.006 to 0.015. In both cross-semester and within-semester predictions, the model with the best AUC used struggling similarity weighting. For cross-semester prediction, random forest with struggling similarity weighting plus problem order weighting achieved the best test AUC (0.793). For within-semester prediction, lasso logistic regression with struggling similarity weighting achieved the best test AUC (0.797). In summary, source code similarity, struggling similarity, and problem order weighting improved model performance consistently but by a relatively small amount (≤ 0.015 AUC).

Problem order weighting aims to account for continued learning, while other weighting schemes aim to account for problem similarities. We investigated whether combining them could improve the prediction. The combination of problem order weighting and one of the problem similarity weighting schemes did improve the test AUC in some conditions, but the improvement was relatively small. For example, random forest with struggling similarity weighting plus problem order weighting achieved the best test AUC in cross-semester prediction, but only 0.0004 higher than random forest with only struggling similarity weighting. Combining all weighting schemes did not improve the test AUC. Table A-3 reports

the other combinations. Overall, combining weighting schemes did not improve the performance.

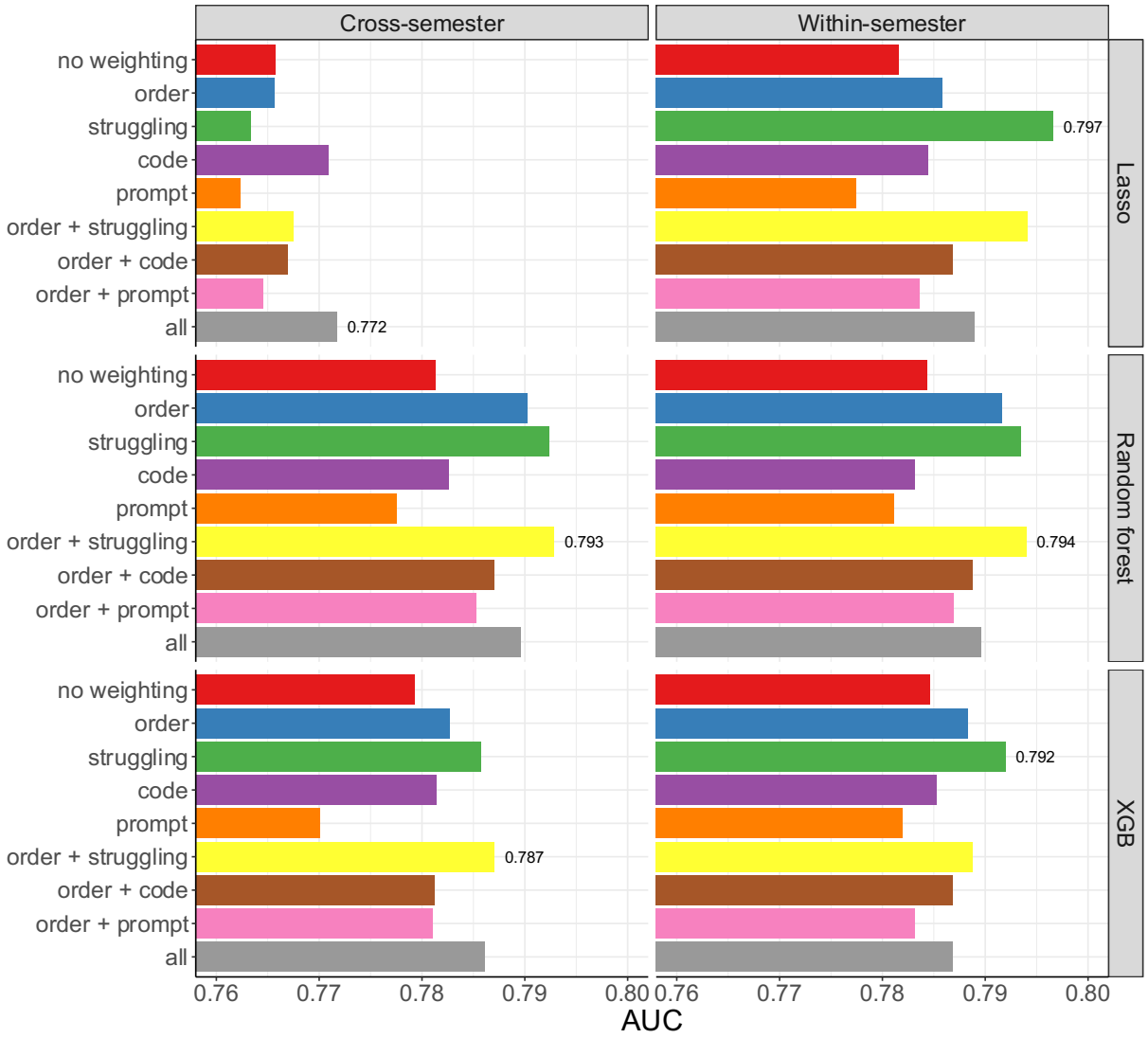

Figure 2. The test AUC of various weighting schemes.

Note. Cross-semester: training data contained spring sample one. Within-semester: training data contained spring samples one and two as well as the fall training sample. XGB: extreme gradient boosting.

3.1.1. The comparison under various numbers of features

Figure 3 displays the test AUC under various numbers of features (values smaller than 0.76 were not displayed for clarity). The relative performance of various weighting schemes in the condition of *not all features* was similar to that of *all features*. In most conditions, problem prompt similarity weighting showed lower test AUC than no weighting, and source code similarity weighting improved the test AUC slightly. Problem order weighting improved the

test AUC better than source code similarity weighting, and struggling weighting or its combination with problem order weighting improved the AUC the best. An exception occurred at the condition of lasso logistic regression for cross-semester prediction, where struggling weighting and its combination with problem order weighting performed much better than source code similarity weighting in the condition of five features. As the number of features increased, source code similarity weighting overtook these weighting schemes. In summary, source code similarity, struggling similarity, and problem order weighting improved model performance consistently regardless of the number of features, but the improvement decreased as the number of features increased.

A noticeable pattern was that the difference between models without weighting and with struggling (or struggling + order) weighting was the largest in the condition of five features, and it generally decreased as the number of features increased. Particularly, as the number of features increased from five to ten, the test AUC of the model without weighting increased greatly (0.009 ~ 0.042), but that of the model with the best weighting increased little (e.g., 0.000 ~ 0.005 for struggling weighting). Further analysis found that it was the change in the proportion of problems that a student struggled with from the first to the third assignment (for simplicity, we refer to it as the struggling change) that contributed to the improvement of the model without weighting. After including this feature, the test AUC increased 0.007 ~ 0.035 in the model without weighting but hardly changed in the model with weighting (e.g., 0.000 ~ 0.002 for struggling weighting).

Note that the struggling change was not weighted even when weighting was used. We compared its correlations with the other features between the conditions of struggling weighting and no weighting. From no weighting to struggling weighting, the average absolute correlation between the struggling change and features that would be weighted if using weighting increased from 0.16 to 0.21. Particularly, its correlation increased from 0.21 to 0.51 with the proportion of problems that a student struggled with, from 0.41 to 0.57 with the proportion of problems on which a student made semantic errors, and from 0.39 to 0.53 with the average number of unique test scores on a problem. The three features were some of the most important features (see Table 6) and included by the model when using only five features. Overall, this suggests that the struggling change contained useful information for prediction. However, when struggling weighting was used, the importance of this feature decreased because the weighting made its useful information contained by the other features.

The struggling weighting added information about problem struggling similarity to the other features. Was this information contained by the struggling change from the first to the third assignment? The differences in the two assignments' similarity to the 20 later problems answer this question. The average similarity between the ten problems in the first assignment and the 20 later problems was 0.40 log-odds ratios, but the average similarity was 1.27 log-odds ratios between the ten problems in the third assignment and the 20 later problems. In terms of normalized weights, the average of the problems in the first assignment was 0.50, but the average of the problems in the third assignment was 1.50. Thus, the struggling change from the first to the third assignment would be reflected in the weighted features. For instance, a student that struggled with a few problems in the first assignment but many problems in the third assignment would have a high weighted proportion of struggling problems because the weights of the third assignment were, on average, three times that of the first assignment. In summary, the particularity of problem arrangement caused that the arrangement captured information about problem struggling similarity. This also explains why the problem order weighting showed similar performance to the problem struggling weighting in many situations (see the second and third rows of Figure 3).

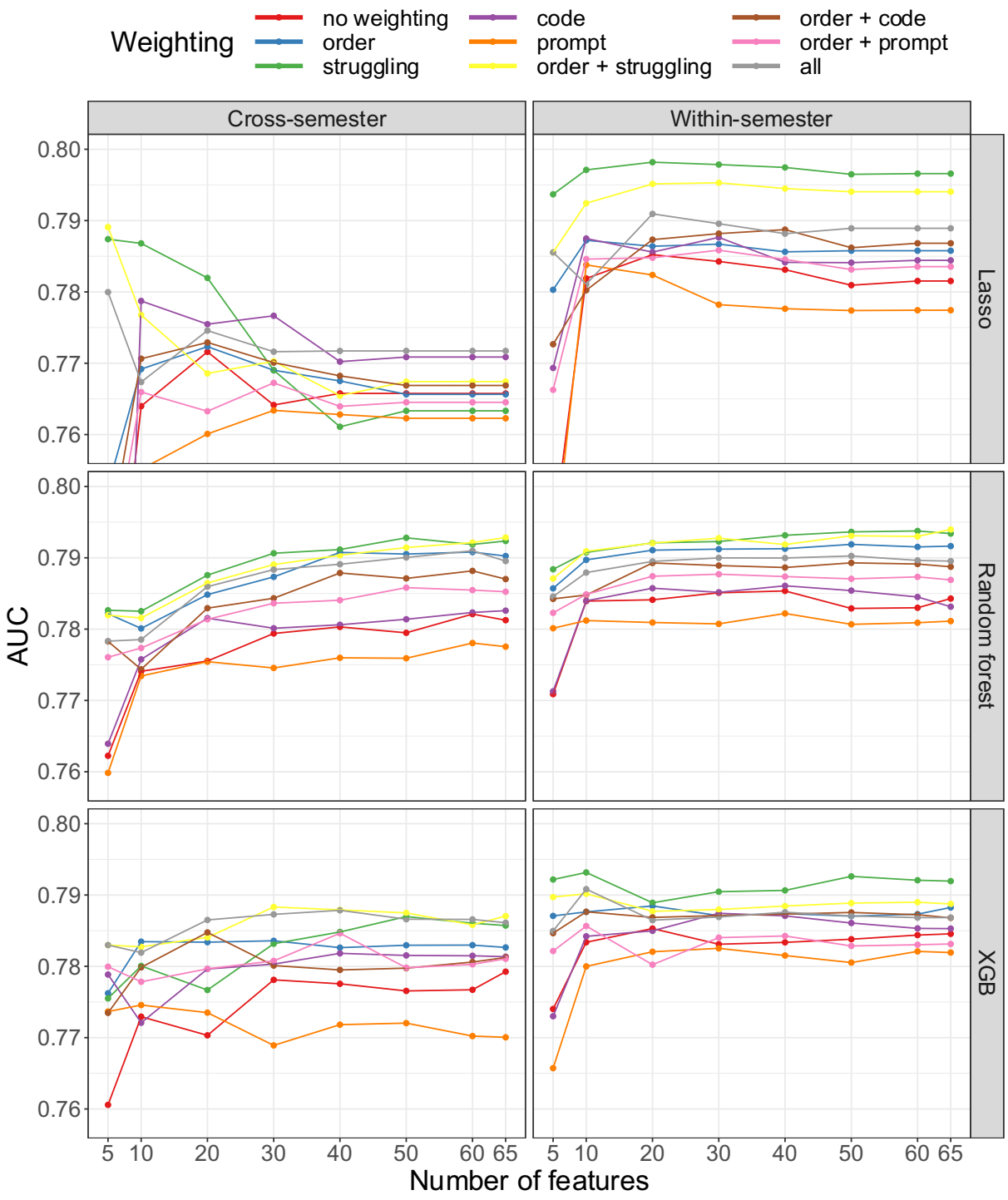

Figure 3: The test AUC changed as the number of features increased.

Note. Values smaller than 0.76 were not displayed for clarity.

3.1.2. The most important features

Table 6 lists the five most important features in the best models with all features for cross- and within-semester predictions. For the within-semester prediction, the rank based on AUC and standardized coefficients were different, and the top five features based on either measure were reported. The interaction between the percentage of problems that a student struggled with (representing students' ability) and the percentage of students struggling with the problem (representing problem difficulty) was the most important feature, regardless of the importance measure and the prediction task (cross- versus within-semester).

The percentage of problems that a student struggled with, whether a student solved problems on the first submission, the percentage of problems on which a student made semantic errors, and its interaction with the percentage of students making errors on the problem were present in the top five features in terms of AUC in cross- and within-semester predictions. The Gini importance rankings were the same as AUC, but the standardized coefficients differed from AUC. Based on the standardized coefficients, the average number of unique test scores and the median number of submissions were present in the top five. The percentage of problems on which a student made semantic errors and its interaction with the percentage of students making semantic errors on the problem had a low rank, perhaps because of their high correlation with the average number of unique test scores (Pearson correlations = 0.86 and -0.79, respectively).

Note that for within-semester prediction, the percentage of problems that a student struggled with and the median number of submissions on a problem had positive coefficients, which is counterintuitive. We provide an explanation in the appendix.

Table 6: Top five features in the best model of cross- and within-semester predictions.

Cross-semester: random forest classifier	**AUC (rank)**	**Gini importance (rank)**
(1 - % problems that a student struggled with) - % students that struggled with the problem	0.777 (1)	0.145 (1)
% problems that a student struggled with	0.762 (2)	0.095 (2)
(1 - % problems on which a student made semantic errors) - % students that made semantic errors on the problem	0.759 (3)	0.079 (3)
% problems on which a student made semantic errors	0.750 (4)	0.064 (4)
% problems that a student solved on the first submission	0.739 (5)	0.053 (5)
Within-semester: Lasso logistic regression	**AUC (rank)**	**Standardized coefficients (rank)**
(1 - % problems that a student struggled with) - % students that struggled with the problem	0.767 (1)	0.747 (1)
% problems that a student struggled with	0.761 (2)	0.220 (5)
% problems on which a student made semantic errors	0.753 (3)	0.000 (64)
(1 - % problems on which a student made semantic errors) - % students that made semantic errors on the problem	0.751 (4)	0.092 (21)

Cross-semester: random forest classifier	AUC (rank)	Gini importance (rank)
% problems that a student solved on the first submission	0.747 (5)	0.264 (4)
Average number of unique test scores on a problem (an approximation to average number of semantic errors on a problem)	0.736 (6)	-0.311 (2)
Median number of submissions on a problem	0.716 (10)	0.276 (3)

Note. The true value of the label means that a student did not struggle with a problem. AUC was the one-feature model AUC.

3.2. COMPARING THE WEIGHTING SCHEME WITH DLKT AND IRT

Table 7 shows the test AUC of lasso logistic regression with and without struggling weighting, along with DLKT and IRT models. Recall that this lasso model only used two features: the proportion of problems that a student struggled with and the proportion of students that struggled with the problem. The lasso with unweighted features showed the lowest test AUC in both cross- and within-semester predictions. With weighted features, the test AUC increased 0.034 and 0.038. In the cross-semester prediction, the lasso with weighted features performed slightly better than DLKT models, with differences in AUC ranging from 0.009 to 0.017. In the within-semester prediction, the lasso with weighted features performed almost the same as DKT and SAKT and slightly better than SAINT. In both predictions, the lasso with weighted features performed better than IRT models, with differences in AUC ranging from 0.024 to 0.036.

Table 7: The test AUC of models with just problem correctness and ID sequences.

Phase	No weighting [a]	Weighting [b]	DKT	SAKT	SAINT	1D IRT	2D IRT
Cross-semester	0.744	**0.778**	0.767	0.761	0.769	0.754	0.751
Within-semester	0.746	0.784	**0.785**	0.782	0.778	0.760	0.748

Note. a: The models in both phases were lasso logistic regression. b: The models in both phases were lasso logistic regression with struggling similarity weighting. These models were chosen based on the average AUC of 5-fold CV in the training data. 1D and 2D IRT: one- and two-dimensional IRT models.

4. DISCUSSION

This study utilized problem similarity and attempt order information to weight behaviors and performance on past problems in the performance prediction on future problems. The main findings are: (1) the source code similarity, struggling similarity, and problem order weighting improved prediction performance consistently regardless of the number of features, but the improvement decreased as the number of features increased; (2) the prompt similarity weighting did not improve prediction performance; (3) combining weighting schemes by taking the average did not improve prediction performance; (4) using the same information,

the struggling similarity weighting made the lasso logistic regression have performance comparable to the SOTA deep learning models.

When only using two features about response correctness and problem difficulties, the model with weighted features showed a test AUC of 0.034 to 0.038 higher than the model with unweighted features, although the differences decreased as the number of features increased. Weighted features generally performed better because the similarity weighting treated past problems differentially according to their relations to future problems, and the problem order weighting made the features capture temporal information of interactions or possible learning. The problem relations and temporal order of interactions are two types of statistical regularities that DLKT can exploit but the methods operating on tabular data have difficulty using (Gervet et al., 2020). To some extent, the weighting schemes helped the latter methods to access the two types of statistical regularities.

For within-semester prediction, our weighting approach performed almost the same as DLKT, which also utilized problem similarity and attempt order to weight past problems but implicitly and in a complex way. However, for cross-semester prediction, our weighting performed slightly better than DLKT. The reason may be that the training sample in the cross-semester prediction was just 1/3 of that in the within-semester prediction. This finding is in line with prior research (Gervet et al., 2020), where DKT performed better than a logistic regression with features about the number of correct and incorrect attempts, student ability, and problem difficulty, but only in the condition of large data size. When the data size was small to medium, the relative performance reversed.

Overall, the results indicate the potential of using problem similarities and problem order as a tool to weight students' historical programming behaviors and performance for learner modeling, especially in situations where KCs are not explicitly defined. The remainder of this section discusses the implications of our results and suggests future directions to address the limitations of using problem similarities and order weightings.

4.1. Implications for learner modeling

With information about response correctness and sequences, the proposed weighting scheme performed better than the model without weighting and IRT models and almost the same as DLKT. This result suggests that using problem relation and order to weight information on past problems in a way simpler than DLKT does not dimmish the prediction accuracy. Moreover, because the computation of the weighting matrix and the process of weighting features were finished before training models, we could incorporate features more than response correctness and sequences and weight these features. With these features, the model with weighting slightly outperformed DLKT (as the best models with all features had an AUC of 0.793 and 0.797 in the cross- and within-semester prediction). This suggests that the flexibility of the simple weighting is worth the cost of not optimizing the weighting matrix. Moreover, computing weighted features is a step of feature engineering and independent of the ML models, so our weighting approaches can be used along with ML models simpler than deep learning. This means that a learner model based on our weighting may be more interpretable but still have accuracy comparable to SOTA deep learning models.

There was not a single weighting scheme that always performed the best, but the top-performers seemed to be struggling similarity and problem order. The struggling similarity between two problems was defined as their similarity in student struggling patterns. This was the performance correlation between the two problems. Computing this performance-based similarity is simple and does not rely on problem prompts or source code, and thus,

performance similarity weighting is applicable to learner modeling for tasks other than programming problems. Similarly, problem order weighting is also applicable in other contexts, although its utility may be limited (which we discuss in the next section).

This study found that when the weighting scheme was used, a few simple features (e.g., problem difficulty, average student ability) seemed sufficient for prediction performance comparable to many features. For instance, when the struggling similarity weighting was used in the within-semester prediction, the test AUC in the condition of five features was close to that of ten or more features, with a difference smaller than 0.005. Note the small difference was not because the added features were not useful. Indeed, when the weighting was not used, the test AUC increased by 0.009 ~ 0.042 as the number of features increased from five to ten. Overall, these findings suggest that the weighting scheme allows a simple model to perform well.

A related finding is that the improvement of the weighting scheme diminished as the feature set became larger. This suggests that the weighting schemes may be most useful for getting more information from a smaller set of features and less beneficial when a comprehensive set of features is available.

Based on past programming process studies, we engineered a set of features related to programming errors, behaviors, and debugging. Prior studies that compared features did not test the generalizability using a train-test split or CV and compared limited features (e.g., Carter et al., 2015; Tabanao et al., 2011). Thus, to some extent, this study provided new information about the relative importance of these features in predicting performance. The features related to semantic errors were the most important, followed by features related to syntax errors. For instance, the proportion of problems on which a student made semantic errors had a test AUC higher than the proportion of problems on which a student made syntactic errors (0.753 versus 0.722 in within-semester prediction; 0.750 versus 0.706 in cross-semester prediction). This result is in line with prior research (Fitzgerald et al., 2008) and suggests that novice programmers' struggle with semantic errors may be more critical in predicting future performance than the struggle with syntactic errors. Overall, the results suggest the importance of information about programming errors in programming performance prediction.

Debugging features were also useful but strongly correlated to error-related features. Interestingly, transition strength between code states showed higher predictive power than the rate of fixing syntactic errors or increasing test scores. For example, the transition from a code state with syntactic errors to a state with test errors had a test AUC higher than the rate of fixing syntactic errors (0.688 versus 0.521 in within-semester prediction; 0.670 versus 0.569 in cross-semester prediction). The result suggests that transition strengths among code states may better indicate novices' debugging skills.

Cechák and Pelánek (2021) suggest that reliable performance-based similarity measures may need 2000 answers per problem. The answers per problem in the current study were no more than 247 in the cross-semester prediction and 740 in the within-semester prediction, much lower than 2000. The finding that the struggling similarity measure performed better than the other similarity measures seemingly conflicts with Cechák and Pelánek's study. However, the inconsistency is likely because of a difference in the evaluation criterion. In Cechák and Pelánek's study, the criterion was the correlation between the performance-based measure and a problem metadata-based measure, which was used as the ground truth. In our study, the criterion was prediction accuracy. The finding in the current and Cechák and Pelánek's studies together suggests that the utility of a performance-based similarity measure

depends on the evaluation criterion. Choosing the problem similarity measure based on the research context and application purpose is critical.

4.2. LIMITATIONS AND FUTURE STUDIES

One main limitation of the struggling and source code weighting schemes is that the relations between past and future problems rely on students' solution or performance data. The problem prompt weighting does not require such data, but it was not useful in this study. Thus, the model based on similarity weighting may decrease when predicting performance on new problems because of missing relations between old and new problems. In this condition, if the mapping between problems and KCs is available, learner modeling methods that use the mapping may be a better choice (e.g., BKT and PFA). Nevertheless, the mapping provides problem metadata, which can also be used to measure problem similarity (Pelánek, 2020). Specifically, we can derive the Q matrix based on the mapping and compute a pairwise problem similarity matrix. The utility of a weighting scheme based on this similarity matrix entails future investigation.

Another limitation of the struggling weighting scheme is that it assumes that students' knowledge changes little between two problems sharing the same KCs. If students learn on these KCs after one problem, their performance on the two problems may not be strongly related. This limitation may be mitigated by the normalization step (see Section 2.3.6) because the normalization assigns final weights based on relative correlation strengths rather than absolute strengths. Although students are learning, the correlation between a past problem *n1* sharing KCs with the future problem may still be stronger than the correlation between another past problem *n2* having no common KC with the future problem. The normalization will assign a higher weight to problem *n1* but a smaller weight to problem *n2*. Nevertheless, if problem *n1* and the future problem have a long interval, students may learn substantially. Consequently, problem *n1* would have a correlation with the future problem the same as problem *n2*.

Our results showed little positive impact from weighting by source code similarity using code2vec embeddings. The reason may simply be that our assumption that similar problems would lead to similar students' solution code is mistaken. Instead, it may be the case that there are more consistent and measurable similarities between code samples from the same student or among students with similar coding backgrounds rather than the final solutions to problems with related programming concepts. It may also be that averaging the embeddings at the problem level—though analogous to accepted practices in natural language embeddings—has caused the model to lose valuable information about students' code submissions. Future research may consider devising novel ways to aggregate code embeddings.

In addition, the number of elements in each code2vec embedding is 384. Having so many elements, as well as the unclear meaning of individual elements, makes it difficult to understand why the source code of two problems is similar or dissimilar. Even if the similarity weighting based on code2vec embeddings could improve prediction, interpretation of problem similarity may be challenging. Thus, future work may consider computing source code similarity based on techniques that can effectively represent source code with a few interpretable features, such as JavaParser (Hosseini and Brusilovsky, 2013).

Finally, we chose to use a pre-trained code2vec model rather than training or fine-tuning on our own data. Our assumption was that the embeddings obtained from such a model would carry relevant information for our purposes, but future work may consider fine-tuning the model to potentially obtain more meaningful embeddings for predicting correctness.

We also found that weighting by prompt similarity led to worse prediction accuracy than not weighting at all. After investigating the text of the problem prompts themselves, this is not very surprising. Each prompt consisted of a short paragraph describing the task, with an average length of 53 words and a range of 13–103 words. The entire vocabulary set was, therefore, very limited. Also, some structural patterns within the prompts may have led to misleading similarity scores. For example, 15 of the 50 prompts began with "Write a function in Java that implements the following logic," though without seeming conceptual similarity between them, and only one of which was in the 20 later problems. It would be beneficial to examine the results of prompt similarity weighting when using a system that has longer, more elaborate prompts.

The method of quantifying problem similarity is flexible, but the flexibility comes at the cost of too many decisions to make. Researchers need to decide which problem properties (e.g., problem performance, solutions, prompt) the similarity is about, how to quantify these properties (e.g., performance as whether a student struggled with the problem versus whether a student made syntactic errors on the problem), and which similarity measure to use (e.g., cosine similarity, Euclidean distance, or Pearson correlation). This study found that weighting based on struggling similarity worked better than weighting based on the other problem properties, but the advantage of struggling similarity may be due to its relevance to the label—whether or not a student struggled with a problem. With a different target label, similarity weighting based on other characteristics may work better. When calculating similarity measures for source code and problem prompts, we did not find a clear difference among cosine similarity, inverse Euclidean distance, and Pearson correlation. However, when the source code is represented via methods other than code2vec (e.g., JavaParser; Hosseini and Brusilovsky, 2013) or the problem prompts are longer and more elaborate, differences among these measures may become clearer. As Pelánek (2020) stated, there is no single answer for which problem property to use, how to quantify a property, and which similarity measure to use. Future work may develop tools that use CV to choose the best options automatically and ease the use of problem similarity weighting in learner modeling.

Weighting by problem order showed a small positive impact. The particular problem arrangement made the problem order weighting contain information about problem relations. Thus, it was challenging to determine the reason for the small improvement. The reason might be because problem order weighting accounted for the decay impact of past problems, or it contained information about problem relations, or both. Besides, the definition of the CSEDM competition task might limit the application of this approach to the task. In the competition, the attempt order on the 20 later problems was kept hidden and unknown for prediction. As such, it was not possible to use the problem order information on the first problems of the 20 later problems when predicting performance on the subsequent problems. Nevertheless, this issue may not exist in practice if the prediction is dynamic. Once a student finishes a problem, their behaviors and performance on this problem can be used to update features to predict performance on the next problem. Thus, problem order weighing may have a larger potential in dynamic prediction and may be worth further investigation.

Another limitation of our problem order weighting scheme is that weights are assigned in the same way across all students (i.e., the first problem has a weight of 1, the second problem 2, etc.). This means that the decay impact of past problems was the same across students, implying that the learning and forgetting rates in programming knowledge are the same across students. This assumption is likely incorrect. The learning rate may vary across populations (Pardos and Heffernan, 2010), such as students with programming experiences versus those without any experience. It may be interesting to examine whether using different problem

order weighting for different populations of students could improve the prediction performance. If so, researchers may be able to use problem order weighting to answer research questions in a manner analogous to the way that studies have used BKT to answer research questions (Beck et al., 2008; Pedro et al., 2013). For example, assume two groups of students in this study: one received feedback when they finished a problem, and one did not. We may be able to use problem order weighting to investigate whether the feedback impacts learning. Specifically, if using different problem order weighting for the two groups could achieve better prediction performance than using the same problem order weighting, we may infer that the two groups have different learning rates. Consequently, we would conclude that the feedback has an impact.

We combined different weighting schemes by averaging them, but this might not be a good practice because a weighting scheme that did not help (e.g., problem prompt weighting) would damage the performance of the average. How to combine different weighting needs a systematic investigation, which is beyond the scope of this study. The best weights of different weighting schemes in the combination weighting may be discovered through grid-searching and CV, analogous to the hyperparameters in ML.

4.3. Concluding remarks

Knowledge tracing and performance prediction are powerful tools in AIED, but the process of defining KCs and mapping them to problems is time- and effort-consuming. This paper proposes a set of non-KC based weighting schemes to improve prediction performance. Specifically, the weighting schemes adjust the contribution of students' behaviors and performance on past problems in predicting performance on future problems. A past problem's weight to a future problem is proportional to the two problems' similarity. Compared to DLKT and the other performance prediction methods that also use problem similarity to weight information on past problems, the proposed weighting schemes are more flexible in capturing problem similarity on various problem properties and weighting various behaviors and performance information on past problems. We measured the similarity between two problems in terms of the source code of students' solutions to the problems, problem prompts, and students' struggling patterns. After applying the weighting schemes to the dataset of the 2nd CSEDM data challenge, we found that similarity weighting based on struggling patterns and source code increased prediction performance, but similarity weighting based on problem prompts did not. In addition, another weighting scheme that aimed to account for the decay impact of past problems also increased prediction performance. While these weighting schemes did not result in particularly large increases in prediction accuracy, they allowed a simple, interpretable model such as logistic regression to perform on par with SOTA deep learning models. Furthermore, the proposed weighting schemes are applicable to fields beyond computer programming.

5. Appendix

5.1. Explanations of features in Table 3

The average number of unique test scores on a problem. The submission traces did not contain the exact semantic errors that the source code had. However, if a submission had semantic

errors, the test score would be smaller than one. Thus, we used the number of unique test scores on a problem as an approximation to the number of semantic errors on the problem.

Rates of fixing syntax errors. This feature is the proportion of pairs of successive submissions where at least one syntax error in the first submission disappeared in the second submission (among all pairs where the first submission had syntax errors).

Rate of making new syntax errors. This feature is the proportion of pairs of consecutive submissions where a syntax error did not exist in the first submission but appeared in the second submission (among all pairs where the first submission had syntax errors).

Rate of improving test scores. This feature is the proportion of pairs of successive submissions where the test score increased from the first to second submissions (among all pairs where the first submission had a test score smaller than one).

The average difference in test scores between two consecutive submissions. This feature is the mean of test score differences between two successive submissions.

Transition strength between code states. We classified each submission into three categories of code states: containing syntax errors (SE), containing semantic errors (test score < 1; TE), and correct (CO). We considered six possible transitions among these code states: SE -> SE, SE -> TE, SE -> CO, TE -> SE, TE -> TE, and TE -> CO. Transitions beginning with CO are not considered because students rarely made more submissions on a problem after they solved it. The transition strength is quantified by the log-odds ratio. For example, the log-odds ratio of SE -> TE is $ln\frac{\left(a+\frac{1}{2}\right)/\left(b+\frac{1}{2}\right)}{\left(c+\frac{1}{2}\right)/\left(d+\frac{1}{2}\right)}$, where *ln* is the natural logarithm. Table A1 explains *a, b, c*, and *d*. For example, *a* is the number of submission pairs where the code states of the first and second submissions are SE and TE, respectively. ½ is used to reduce the bias in the log-odds ratio estimate (Dagne et al., 2002).

Table A-1: Contingency table for the transition SE -> TE.

First state \ Second state	TE	Not TE
SE	*a*	*b*
Not SE	*c*	*d*

The proportion of problems that a student made a submission in less than 15s. Following prior research (Pinto et al., 2021), we considered a submission that had a gap from the last submission shorter than 15 seconds as a quick submission. Typically, consecutive quick submissions may indicate guessing behaviors. In this study, there were quick submissions but hardly consecutive quick submissions. Thus, we only considered whether the student made at least one quick submission on a problem. Note that we did not regard a single quick submission as guessing.

The number of days with at least one submission. Prior research considered the number of days where a student was active as an indicator of spacing (Yeckehzaare et al., 2022). In this study, we defined being active as making at least one submission.

The number of lines added, deleted, and modified. A large value in this feature means that the student tended to change the code a lot in each submission. This feature may be related to the "code-a-little, test-a-little" approach, which has been recommended for novices (Baumstark and Orsega, 2016).

5.2. MODEL CONFIGURATIONS

Table A-2: The configurations of various models.

Model	Hyperparameters	
Lasso logistic	C=1	
Random forest	max_depth=5, min_samples_split=10, n_estimators=500	
XGB	learning_rate = 0.01, max_depth=3, subsample = 0.5, n_estimators = 500	
LSTM-DKT	batch size = 32, optimizer = adam, learning rate = 0.001, loss function = binary crossentropy, LSTM layer hidden units = 10, dropout = 0.3, recurrent dropout = 0.3, training epochs = 50 (+ early stopping), validation fraction = 0.2	
SAKT	batch size = 256, optimizer = adam, loss function = binary crossentropy, sequence length = 31, total number of questions = 50, validation fraction = 0.2	learning rate = 0.001, dimensions of model (embeddings) = 20, number of attention heads = 5, dropout = 0.4, training epochs = 100 (+ early stopping)
SAINT		learning rate = 0.0001, dimensions of model (embeddings, attention, linear layers) = 50, number of attention heads = 5, dropout = 0.2, training epochs = 100 (+ early stopping), number of encoder layers = 4, number of decoder layers = 4, number of unique concept categories = 1 (because we do not know the KCs)
2-dimensional 2PL IRT	Problems on the 1st dimension: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 33. Problems on the 2nd dimension: 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50.	

5.3. THE TRAIN AND TEST AUC OF VARIOUS WEIGHTING SCHEMES AND COMBINATIONS

Table A-3: The train and test AUC of various weighting schemes.

Phase	Weighting	Lasso		Random forest		XGB	
		Train	**Test**	**Train**	**Test**	**Train**	**Test**
Cross-semester (spring sample one)	No weighting	0.798	0.766	0.839	0.781	0.849	0.779
	order	0.798	0.766	0.841	0.79	0.85	0.783
	struggling	0.824	0.763	0.86	0.792	0.873	0.786
	code	0.798	0.771	0.847	0.783	0.855	0.781
	prompt	0.796	0.762	0.845	0.778	0.855	0.77
	order + struggling	0.813	0.767	0.856	**0.793**	0.866	**0.787**
	order + code	0.799	0.767	0.848	0.787	0.855	0.781
	order + prompt	0.797	0.765	0.847	0.785	0.858	0.781
	struggling + prompt	0.812	0.769	0.855	0.787	0.866	0.784
	struggling + code	0.814	0.766	0.854	0.788	0.865	0.781
	prompt + code	0.797	0.768	0.845	0.78	0.856	0.778
	order + struggling + prompt	0.803	0.772	0.853	0.789	0.861	0.787
	order + struggling + code	0.805	0.77	0.853	0.791	0.861	0.786
	order + prompt + code	0.797	0.767	0.848	0.788	0.857	0.783
	struggling + prompt + code	0.807	0.767	0.852	0.785	0.862	0.784
	all	0.801	**0.772**	0.852	0.79	0.86	0.786
Within-semester (Spring samples one and two + fall training)	No weighting	0.776	0.782	0.803	0.784	0.811	0.785
	order	0.776	0.786	0.804	0.792	0.812	0.788
	struggling	0.788	**0.797**	0.811	0.793	0.818	**0.792**
	code	0.775	0.784	0.802	0.783	0.808	0.785
	prompt	0.773	0.777	0.798	0.781	0.805	0.782
	order + struggling	0.782	0.794	0.808	**0.794**	0.815	0.789
	order + code	0.776	0.787	0.803	0.789	0.81	0.787
	order + prompt	0.774	0.784	0.801	0.787	0.808	0.783
	struggling + prompt	0.781	0.79	0.806	0.789	0.815	0.783
	struggling + code	0.782	0.793	0.807	0.79	0.814	0.786
	prompt + code	0.774	0.781	0.8	0.782	0.808	0.783
	order + struggling + prompt	0.778	0.789	0.807	0.791	0.813	0.787
	order + struggling + code	0.779	0.791	0.808	0.792	0.814	0.788
	order + prompt + code	0.775	0.785	0.804	0.788	0.811	0.785
	struggling + prompt + code	0.779	0.788	0.805	0.787	0.813	0.784
	all	0.777	0.789	0.807	0.79	0.813	0.787

Note. XGB: extreme gradient boosting. The similarity measure for code and prompt is cosine similarity.

Table A-4: The AUC of similarity weighting based on inverse Euclidean distance and Pearson correlation.

Phase	Weighting	Lasso		Random forest		XGB	
		Train	Test	Train	Test	Train	Test
Cross-semester (spring sample one)	Code inverse Euclidean	0.798	0.768	0.848	0.784	0.854	0.780
	Code Pearson correlation	0.798	0.771	0.847	0.782	0.855	0.781
	Prompt inverse Euclidean	0.796	0.766	0.847	0.778	0.856	0.777
	Prompt Pearson correlation	0.796	0.762	0.845	0.776	0.855	0.771
Within-semester (spring samples one and two+ fall training)	Code inverse Euclidean	0.776	0.783	0.805	0.784	0.812	0.785
	Code Pearson correlation	0.775	0.784	0.802	0.783	0.808	0.784
	Prompt inverse Euclidean	0.775	0.779	0.801	0.782	0.809	0.783
	Prompt Pearson correlation	0.773	0.777	0.799	0.780	0.805	0.781

Note. XGB: extreme gradient boosting. Inverse Euclidean: the similarity measure based on the inverse Euclidean distance. Pearson correlation: the similarity measure based on inverse Pearson correlation.

5.4. COUNTERINTUITIVE COEFFICIENTS IN TABLE 6

The percentage of problems that a student struggled with had a positive coefficient in the lasso logistic regression (0.220), which seems to suggest that a student struggling with earlier problems was less likely to struggle with later problems. However, the lasso model also contained the interaction between this feature and the percentage of students struggling with the problem (the coefficient was 0.747). As such, the coefficient of the percentage of problems that a student struggled with should not be interpreted on its own. Indeed, if we combine this feature and its interaction, its coefficient would become -0.527 (= 0.220 - 0.747).

The median number of submissions on a problem also had a counterintuitive positive coefficient, suggesting that a student with a higher median number of submissions on early problems was less likely to struggle with later problems. The reason for this counterintuitive coefficient may be that the median number of submissions adjusts the effects of other features related to the number of submissions, such as the percentage of problems that a student solved on the first submission and the average number of submissions on a problem, which had a negative coefficient (-0.183). Struggling with a problem is based on whether the number of submissions on the problem was beyond 75% of peers. Thus, the median number of submissions might also adjust the effect of the percentage of problems that a student struggled with.

REFERENCES

ALEVEN, V., 2010. Rule-based cognitive modeling for intelligent tutoring systems. In *Advances in Intelligent Tutoring Systems*, R. Nkambou, J. Bourdeau, R. Mizoguchi, Eds. Studies in Computational Intelligence, vol 308, Springer, Berlin, Heidelberg, 33-62.

ALON, U., ZILBERSTEIN, M., LEVY, O., AND YAHAV, E., 2019. Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages. 3*, POPL, 1-29.

ANDERSON, J. R., CORBETT, A. T., KOEDINGER, K. R., AND PELLETIER, R., 1995. Cognitive Tutors: Lessons learned. *Journal of the Learning Sciences. 4*, 2, 167-207.

AZCONA, D., ARORA, P., Hsiao, I., AND SMEATON, A., 2019. User2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics and Knowledge (LAK 2019)*. Association for Computing Machinery, 86-95.

BAUMSTARK, L., AND ORSEGA, M., 2016. Quantifying introductory CS students' iterative software process by mining version control system repositories. *Journal of Computing Sciences in Colleges 31*, 6, 97-104.

BECK, J. E., CHANG, K., MOSTOW, J., AND CORBETT, A., 2008. Does help help? Introducing the Bayesian evaluation and assessment methodology. In *Intelligent Tutoring Systems. ITS 2008*, B. P. Woolf, E. Aïmeur, R. Nkambou, S. Lajoie, Eds. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 383-394.

BECKER, B. A., GLANVILLE, G., IWASHIMA, R., MCDONNELL, C., GOSLIN, K., AND MOONEY, C., 2016. Effective compiler error message enhancement for novice programming students. *Computer Science Education 26*, 2-3, 148-175.

BOSCH, N., AND PAQUETTE, L., 2021. What' s next? Sequence length and impossible loops in state transition measurement. *Journal of Educational Data Mining. 13*, 1, 1-23.

CARTER, A. S., HUNDHAUSEN, C. D., AND ADESOPE, O., 2015. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the 11th Annual International Conference on International Computing Education Research (ICER 2015)*. Association for Computing Machinery, 141-150.

CECHÁK, J., AND PELÁNEK, R., 2021. Experimental evaluation of similarity measures for educational items. In *Proceedings of the 14th International Conference on Educational Data Mining (EDM 2021),* S. I. H. Hsiao, S. Sahebi, F. Bouchet, and J. Vie, Eds. International Educational Data Mining Society, 553-558.

CEN, H., KOEDINGER, K., AND JUNKER, B., 2006. Learning factors analysis: A general method for cognitive model evaluation and improvement. In *Intelligent Tutoring Systems. ITS 2006*, M. Ikeda, K. D. Ashley, and T. Chan, Eds. Lecture Notes in Computer Science, vol 4053, Springer, Berlin, Heidelberg, 164-175.

CHALMERS, R. P., 2012. mirt: A multidimensional item response theory package for the R environment. *Journal of Statistical Software,* 48, 6, 1-29.

CHOI, Y., LEE, Y., CHO, J., BAEK, J., KIM, B., CHA, Y., SHIN, D., BAE, C., AND HEO, J., 2020. Towards an appropriate query, key, and value computation for knowledge tracing. In

Proceedings of the 7th ACM Conference on Learning @ Scale (L@S 2020). Association for Computing Machinery, 341-344.

CORBETT, A. T., AND ERSON, J. R., 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction 4*, 4, 253-278.

DAGNE, G. A., HOWE, G. W., BROWN, C. H., AND MUTHÉN, B. O., 2002. Hierarchical modeling of sequential behavioral data: An empirical Bayesian approach. *Psychological Methods 7*, 2, 262-280.

EMBRETSON, S. E., AND REISE, S. P., 2000. *Item Response Theory for Psychologists*. Lawrence Erlbaum Associates.

FITZGERALD, S., LEWANDOWSKI, G., MCCAULEY, R., MURPHY, L., SIMON, B., THOMAS, L., AND ZANDER, C., 2008. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education 18*, 2, 93-116.

GERVET, T., KOEDINGER, K., SCHNEIDER, J., AND MITCHELL, T., 2020. When is deep learning the best approach to knowledge tracing? *Journal of Educational Data Mining 12*, 3, 31-54.

GONG, Y., BECK, J. E., AND HEFFERNAN, N. T., 2011. How to construct more accurate student models: Comparing and optimizing knowledge tracing and performance factor analysis. *International Journal of Artificial Intelligence in Education 21*, 1–2, 27-45.

HOSSEINI, R., AND BRUSILOVSKY, P., 2013. JavaParser: A fine-grain concept indexing tool for java problems. In *The 1st Workshop on AI-supported Education for Computer Science*, N. Le, K. E. Boyer, B. Chaudhry, B. Di Eugenio, S. I. H. Hsiao, L. A. Sudol-Delyser, Eds. CEUR Workshop Proceedings, vol 1009, 60-63.

JADUD, M. C., 2006a. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the 2nd International Workshop on Computing Education Research (ICER 2006)*. Association for Computing Machinery, 73-84.

JADUD, M. C., 2006b. *An exploration of novice compilation behaviour in BlueJ*. Ph.D. thesis, University of Kent, Canterbury, United Kingdom.

JURAFSKY, D., AND MARTIN, J. H., 2000. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Pearson Prentice Hall.

LE, Q., AND MIKOLOV, T., 2014. Distributed representations of sentences and documents. *Proceedings of the 31st International Conference on Machine Learning (ICML 2014)*, E. P. Xing and T. Jebara, Eds. Association for Computing Machinery, 1188-1196.

LEE, Y., CHOI, Y., CHO, J., FABBRI, A. R., LOH, H., HWANG, C., LEE, Y., KIM, S., AND RADEV, D., 2019. Creating a neural pedagogical agent by jointly learning to review and assess. arXiv. https://doi.org/10.48550/arxiv.1906.10910

LIU, Q., HUANG, Z., YIN, Y., CHEN, E., XIONG, H., SU, Y., AND HU, G., 2021. EKT: Exercise-aware knowledge tracing for student performance prediction. *IEEE Transactions on Knowledge and Data Engineering, 33*, 1, 100-115.

LUCKIN, R., HOLMES, W., GRIFFITHS, M., AND FORCIER, L. B., 2016. *Intelligence Unleashed: An Argument for AI in Education*. Pearson Education, London.

MAO, Y., ZHI, R., AND KHOSHNEVISAN, F., 2019. 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), C. F. Lynch, A. Merceron, M. Desmarais, R. Nkambou, Eds. International Educational Data Mining Society, Montréal, Canada, 119-128.

NATTI, A., AND ATHREY, D., 2019. CSEDM 2019 challenge. In *Joint Proceedings of the 2nd CSEDM Workshop at International Conference on Learning Analytics and Knowledge 2019*, D. Azcona, Y. V. Paredes, S. I. H. Hsiao, and T. W. Price, Eds. CEUR Workshop Proceedings.

PANDEY, S., AND KARYPIS, G., 2019. A self-attentive model for knowledge tracing. In *Proceedings of the 12th International Conference on Educational Data Mining (EDM 2019)*, C. F. Lynch, A. Merceron, M. Desmarais, R. Nkambou, Eds. International Educational Data Mining Society, 384-389.

PARDOS, Z. A., AND HEFFERNAN, N. T., 2010. Modeling individualization in a Bayesian networks implementation of knowledge tracing. In *User Modeling, Adaptation, and Personalization (UMAP 2010)*, P. De Bra, A. Kobsa, D. Chin, Eds. Lecture Notes in Computer Science, vol 6075, Springer, Berlin, Heidelberg, 255-266.

PAVLIK JR, P. I., CEN, H., AND KOEDINGER, K. R., 2009. Performance factors analysis – a new alternative to knowledge tracing. In *Proceedings of the 14th International Conference on Artificial Intelligence in Education (AIED 2009)*, V. Dimitrova, R. Mizoguchi, B. du Boulay, A. Graesser, Eds. IOS Press, Amsterdam, Netherlands, 531–538.

PEDRO, M. A. S., BAKER, R. S. J. D., AND GOBERT, J. D., 2013. What different kinds of stratification can reveal about the generalizability of data-mined skill assessment models. In *Proceedings of the 3rd International Conference on Learning Analytics and Knowledge (LAK'13)*. Association for Computing Machinery, 190-194.

PELÁNEK, R., 2020. Measuring similarity of educational items: An overview. *IEEE Transactions on Learning Technology 13*, 2, 354-366.

PELÁNEK, R., EFFENBERGER, T., VANĚK, M., SASSMANN, V., AND GMITERKO, D., 2018. Measuring item similarity in introductory programming. In *Proceedings of the 5th Annual ACM Conference on Learning at Scale (L@S 2018)*. Association for Computing Machinery, Article 19.

PIECH, C., BASSEN, J., HUANG, J., GANGULI, S., SAHAMI, M., GUIBAS, L. J., AND SOHL-DICKSTEIN, J., 2015. Deep knowledge tracing. In *Proceedings of Advances in Neural Information Processing Systems*, vol. 28, S. Becker, S. Thrun, K. Obermayer, Eds. MIT Press, Cambridge, MA, United States, 505–513.

PINTO, J. D., ZHANG, Y., PAQUETTE, L., AND FAN, A. X., 2021. Investigating elements of student persistence in an introductory computer science course. In *Joint Proceedings of the 5th CSEDM Workshop at the International Conference on Educational Data Mining 2021,* T. W. Price and S. San Pedro, Eds. CEUR Workshop Proceedings, vol 3051.

REHUREK, R., AND SOJKA, P., 2010. Software framework for topic modelling with large corpora. In *Proceedings of LREC 2010 Workshop New Challenges for NLP Frameworks*. University of Malta, Valletta, Malta, 46-50.

ROSENTHAL, J. A., 1996. Qualitative descriptors of strength of association and effect size. *Journal of Social Service Research 21*, 4, 37-59.

SAHEBI, S., AND BRUSILOVSKY, P., 2018. Student performance prediction by discovering Inter-Activity relations. In *Proceedings of the 11th International Conference on Educational*

Data Mining (EDM 2018), K. E. Boyer, M. Yudelson, Eds. International Educational Data Mining Society, 87-96.

SAHEBI, S., LIN, Y. R., AND BRUSILOVSKY, P., 2016. Tensor factorization for student modeling and performance prediction in unstructured domain. In *Proceedings of the 9th International Conference on Educational Data Mining (EDM 2016)*, T. Barnes, M. Chi, M. Feng, Eds. International Educational Data Mining Society, 502-506.

SARSA, S., LEINONEN, J., AND HELLAS, A., 2022. Empirical evaluation of deep learning models for knowledge tracing: Of hyperparameters and metrics on performance and replicability. *Journal of Educational Data Mining 14*, 2, 32-102.

SHI, Y., CHI, M., BARNES, T., AND PRICE, T.W., 2022. Code-DKT: A code-based knowledge tracing model for programming tasks. In *Proceedings of the 15th International Conference on Educational Data Mining (EDM 2022)*, A. Mitrovic, N. Bosch, Eds. International Educational Data Mining Society, 50-61.

SHI, Y., MAO, Y., BARNES, T., CHI, M., AND PRICE, T. W., 2021. More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code. In *Proceedings of the 14th International Conference on Educational Data Mining (EDM 2021)*, S. I. H. Hsiao, S. Sahebi, F. C. Bouchet, J. Vie, Eds. International Educational Data Mining Society, 446-453.

SHI, Y., SHAH, K., WANG, W., MARWAN, S., PENMETSA, P., AND PRICE, T. W., 2021. Toward semi-automatic misconception discovery using code embeddings. In *Proceedings of the 11th International Conference on Learning Analytics and Knowledge (LAK'21)*. Association for Computing Machinery, 606-612.

SHUTE, V. J., AND ZAPATA-RIVERA, D., 2012. Adaptive educational systems. In *Adaptive Technologies for Training and Education*, P. J. Durlach, A. M. Lesgold, Eds. Cambridge University Press, 7-27.

STAMPER, J. C., KOEDINGER, K. R., BISWAS, G., BULL, S., KAY, J., AND MITROVIC, A., 2011. Human-machine student model discovery and improvement using datashop. In *Artificial Intelligence in Education. AIED 2011*, G. Biswas, S. Bull, J. Kay, A. Mitrovic, Eds. Lecture Notes in Computer Science, vol 6738, Springer, Berlin, Heidelberg, 353-360.

TABANAO, E. S., RODRIGO, M. M. T., AND JADUD, M. C., 2011. Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the 7th International Workshop on Computing Education Research (ICER 2011)*. Association for Computing Machinery, New York, NY, United States, 85-92.

VILLAMOR, M. M., 2020. A review on process-oriented approaches for analyzing novice solutions to programming problems. *Research and Practice in Technology Enhanced Learning 15*, 1-23.

VON DAVIER, M., 2016. Rasch model. In *Handbook of Item Response Theory, Volume One*, W. J. van der Linden, Eds. Chapman and Hall/CRC, 31-50.

WANG, L., SY, A., LIU, L., AND PIECH, C., 2017. Learning to represent student knowledge on programming exercises using deep learning. In *Proceedings of the 10th International Conference on Educational Data Mining (EDM 2017)*, X. Hu, T. Barnes, A. Hershkovitz, L. Paquette, Eds. International Educational Data Mining Society, 324-329.

WOOLF, B. P., LANE, H. C., CHAUDHRI, V. K., AND KOLODNER, J. L., 2013. AI grand challenges for education. *AI Magazine 34*, 4, 66-84.

YECKEHZAARE, I., MULLIGAN, V., RAMSTAD, G., AND RESNICK, P., 2022. Semester-level spacing but not procrastination affected student exam performance. In *Proceedings of 12th International Conference on Learning Analytics and Knowledge (LAK 2022)*. Association for Computing Machinery, 304-314.

YUDELSON, M., HOSSEINI, R., AND BRUSILOVSKY, P., 2014. Investigating automated student modeling in a java MOOC. In *Proceedings of the 7th International Conference on Educational Data Mining (EDM 2014)*, J. Stamper, Z. Pardos, M. Mavrikis, B. M. Mclaren, Eds. International Educational Data Mining Society, 261-264.

ZHANG, J., SHI, X., KING, I., AND YEUNG, D., 2017. Dynamic Key-Value memory networks for knowledge tracing. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 765-774.

ZHAO, S., WANG, C., AND SAHEBI, S., 2020. Modeling Knowledge Acquisition from Multiple Learning Resource Types. In *Proceedings of the 13th International Conference on Educational Data Mining (EDM 2020)*, A. N. Rafferty, J. Whitehill, C. Romero, and V. Cavalli-Sforza, Eds. International Educational Data Mining Society, 313-324.