Interpreting Latent Student Knowledge Representations in Programming Assignments

Nigel Fernandez, Andrew Lan
University of Massachusetts Amherst
{nigel,andrewlan}@cs.umass.edu

ABSTRACT

Recent advances in artificial intelligence for education leverage generative large language models, including using them to predict open-ended student responses rather than their correctness only. However, the black-box nature of these models limits the interpretability of the learned student knowledge representations. In this paper, we conduct a first exploration into interpreting latent student knowledge representations by presenting InfoOIRT, an Information regularized Open-ended Item Response Theory model, which encourages the latent student knowledge states to be interpretable while being able to generate student-written code for open-ended programming questions. InfoOIRT maximizes the mutual information between a fixed subset of latent knowledge states enforced with simple prior distributions and generated student code, which encourages the model to learn disentangled representations of salient syntactic and semantic code features including syntactic styles, mastery of programming skills, and code structures. Through experiments on a real-world programming education dataset, we show that InfoOIRT can both accurately generate student code and lead to interpretable student knowledge representations.

Keywords

Programming Education, Language Models, Interpretability

1. INTRODUCTION

Open-ended problems, which require students to produce free-form responses, either as short answers or essays [3] or even code [41], serve as a highly meaningful form of assessment and complements closed-form problems such as multiple-choice questions [29]. These questions often require students to detail their reasoning process and offer educators a deeper look into their knowledge states. Past work has shown that students' open-ended responses to such questions contain useful information on their knowledge states, e.g., having misconceptions [5, 11, 45] or generally lacking sufficient knowledge [2]. Until recently, however, research has mostly focused on the automated *scoring* of open-ended responses, either via classification methods [12, 47, 51] or clustering [16] and providing corresponding feedback [14, 15, 18, 26, 37, 40]. However, relatively little has been done towards developing student response models that estimate their knowledge from open-ended responses; existing models such as item response theory (IRT) [48], knowledge tracing [9], and factor analysis [31] primarily analyze close-ended responses or graded ones, which are either binary-valued or nominal/ordinal. These models are fundamentally limited for open-ended problems since they cannot fully extract detailed information on student knowledge contained in their free-form responses. See Section 2 for a detailed discussion on related work.

Recent advances in pre-trained generative large language models (LLMs) [6] provide an opportunity to gain deeper insights into student knowledge by analyzing their free-form responses. Most existing works use text embedding models to summarize open-ended responses only as input into knowledge tracing models [46], not fully utilizing the generative capabilities of LLMs. The only recent work that combines generative LLMs with an underlying student response model is open-ended knowledge tracing (OKT) [23], which uses a knowledge tracing model to track the change in student knowledge state over time, and then injects that knowledge state together with the textual problem statement as input to a generative LLM to predict a student's open-ended response as output. Applied to student-written code to programming problems, OKT shows that learned underlying latent student knowledge states have some correlation with student-written code. Despite some early promise, a key limitation of OKT is the interpretability of the latent student knowledge space; there is no clear way to isolate certain elements in these vectors that capture key aspects of student code: ones that reflect their knowledge of key programming knowledge concepts, ones that reflect certain bugs/misconceptions, or even ones that capture distinct coding styles. Since there is no prior enforced on the structure of the latent knowledge space, the black-box nature of LLMs would likely lead to entangled representations that are highly predictive of student responses but hard to interpret.

1.1 Contributions

In this paper, we present a first attempt at interpreting latent student knowledge states in models of open-ended responses, specifically on code that students write for open-ended programming questions. Our contributions are:

N. Fernandez and A. Lan. Interpreting latent student knowledge representations in programming assignments. In B. Paaßen and C. D. Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 933–940, Atlanta, Georgia, USA, July 2024. International Educational Data Mining Society.

https://doi.org/10.5281/zenodo.12730003

1. We develop InfoOIRT [1], an Information-regularized Open-ended IRT model, which predicts student-written code for open-ended programming questions with a focus on learning interpretable student knowledge representations. Inspired by InfoGAN [7], InfoOIRT *maximizes mutual information between a fixed subset of latent knowledge states enforced with simple prior distributions and generated student code* to encourage the latent factors to learn disentangled representations of salient code features. Although our idea can potentially be applied to other subjects with open-ended questions, we ground our analysis in Computer Science (CS) education.

2. We conduct quantitative experiments on a real-world student code dataset to show that information regularization does not impact the ability of InfoOIRT to accurately predict student code compared to baselines.

3. We conduct qualitative analyses to interpret the learned student knowledge representations. Using a combination of both continuous and discrete latent student knowledge state factors, we present examples of generated student code highlighting the salient syntactic and semantic features captured by these states.

2. RELATED WORK

2.1 Interpretable Representation Learning

There exists prior work in learning interpretable representations for the underlying processes of image [10, 33] and text [27] generation. A seminal method in unsupervised representation learning, InfoGAN [7] aims to learn disentangled representations, one which explicitly represents the salient features of the data as easily interpretable factors (e.g., number, orientation, and stroke thickness in handwritten digits), using an information-based regularization in the training objective. InfoOIRT extends this idea to learn interpretable representations in LLMs for code generation, specifically student responses to programming problems.

2.2 Student Modeling

There exist many models of student knowledge, depending on how they characterize both latent knowledge states and observed responses. For latent knowledge states, the highly interpretable Bayesian knowledge tracing model treats them as binary-valued, i.e., whether a student masters a skill or not. Factor analysis-based methods [8, 31] use a set of handcrafted features to summarize past student activities and represent student knowledge, before relying on IRT models to predict student responses from these features. On the contrary, deep learning-based KT methods [24, 30, 32, 43, 50] treat student knowledge as latent vectors in deep neural networks, resulting in models that excel at future performance prediction but have limited interpretability.

For observed responses, despite most existing models treating them as binary-valued, i.e., correct/incorrect, there exist some models that analyze the exact student response including multiple-choice options [13] and partial credits [49]. In general, one can use polytomous IRT models [28] as the response prediction component in KT methods to predict categorical-valued (such as options in multiple-choice questions) and ordinal-valued (such as partial credit) responses [20]. In the programming domain, [25, 52] use code embedding techniques to convert student-written code into vectors to help student models track their progress. However, they do not use generative LLMs to predict student code.

2.3 Program Synthesis and CS Education

There exist many works applying program synthesis techniques for computer science education to generate (possibly buggy) student code [23, 44], generate new problems [1] with code explanations [38], generate student-code guided test cases [19], provide real-time hints [36], and suggest bug fixes [17]. However, the black-box nature of these models provides limited interpretability.

3. INTERPRETABLE OPEN-ENDED IRT

3.1 Problem Formulation and OIRT

Item response theory (IRT) [4] involves diagnosing a student's mastery of knowledge components/skills/concepts from their responses to problems, where we assume a student's knowledge state is static, i.e., it does not change as they respond to problems. For open-ended item response theory (OIRT), we need two essential components. First, a knowledge estimation (KE) component that estimates a student j's knowledge state from the set of student code submissions c_{ij} to problems p_i denoted by $\{(p_i, c_{ij})\}$, i.e., $h_j = \text{KE}(\{(p_i, c_{ij})\})$. Second and more importantly, a response generation (RG) component that predicts student j's open-ended code submission to a target problem p_k using a generative model, i.e., $c_{kj} = \text{RG}(p_k, h_j)$. This generation model is the key difference between OIRT and traditional IRT: our goal is to predict the code a student would write for an open-ended programming problem via a generative model, rather than simply predicting its correctness.

We denote the student's latent knowledge as a d-dimensional vector h_j for every student j. This setup is similar to learning a multidimensional student ability parameter in IRT. OIRT leverages generative language models and employs a text-to-code finetuned GPT-2 [34] model. A problem p_k is tokenized by GPT-2 into a sequence of M tokens where each token has a 768-dimensional embedding, i.e., $\bar{p}_m \in \mathcal{R}^{768}$ for $m = 1, \ldots, M$ (here, we drop the problem index k). We inject student j's knowledge state h_j by replacing the raw problem token embeddings with knowledge-guided embeddings using a linear alignment function f, i.e., $p_m = f(\bar{p}_m, h_j)$, similar to [23]. The predicted student code is generated autoregressively using GPT-2 given the knowledge-guided problem embeddings as input. OIRT jointly learns the student knowledge states and the fine-tuned GPT-2 parameters together with the linear alignment function. The objective for one student code submission c_{kj} by student j with knowledge state h_j to problem p_k, consisting of N tokens, is: $\mathcal{L}_{\text{OIRT}} = \sum_{n=1}^{N} -\log P_\theta(c_{kj}^n | p_k, h_j, \{c_{kj}^{n'}\}_{n'=1}^{n-1})$, where θ denotes the learnable parameters of the KE and RG components. The final objective is the sum of this loss $\mathcal{L}_{\text{OIRT}}$ over code submissions by all students to all problems.

3.2 InfoOIRT: Information-regularized OIRT

One key limitation of OIRT is that the learned student knowledge states are hard to interpret and associate with

[1]Code: `https://github.com/umass-ml4ed/InfoOIRT`

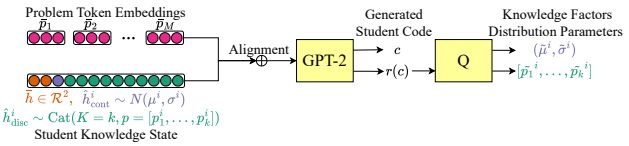

Figure 1: InfoOIRT Model Architecture

different programming skills. We present a simple modification of OIRT, inspired by InfoGAN [7], to learn interpretable and meaningful latent student knowledge states. The idea is to *maximize the mutual information between a fixed subset of the student knowledge state dimensions enforced with simple prior distributions and the generated student code.* These dimensions help us discover semantic and meaningful hidden representations of student code.

We now detail our InfoOIRT model visualized in Figure 1. Following InfoGAN [7], we decompose a latent student knowledge state h into two parts: 1) $\bar{h}$, which represents the incompressible student knowledge state, and 2) $\hat{h}$, which consists of simple and interpretable latent factors $\hat{h}^1, \ldots, \hat{h}^K$, to represent the salient structured semantic features of student written code. However, GPT-2 could *ignore* these additional latent factors $\hat{h}$ and simply generate student code c with a probability distribution satisfying $P(c|\hat{h}) = P(c)$. We, therefore, impose an *information-theoretic regularization* to encourage high mutual information between latent factors $\hat{h}$ and the GPT-2 generator distribution $G(\bar{h}, \hat{h}, p)$ which generates student code c corresponding to a student with knowledge state h. This regularization encourages the latent factors to explicitly contain information that dictates the variation in student code, disentangling the interpretable dimensions from incompressible noise. Specifically, mutual information is defined as $I(\hat{h}, G(\bar{h}, \hat{h}, p)) = H(\hat{h}) - H(\hat{h}|G(\bar{h}, \hat{h}, p))$. Intuitively, mutual information is maximized when the uncertainty, i.e., entropy, in $\hat{h}$ given c is minimized, meaning that the *information in the simple latent knowledge factors $\hat{h}$ should not be lost in the generation of the student code c*, thereby reducing the likelihood of GPT-2 ignoring the latent factors during student code generation. However, maximizing the mutual information directly is hard since it requires access to the posterior $P(\hat{h}|c)$. Therefore, we use a variational lower bound of mutual information following [7]: $I(\hat{h}, G(\bar{h}, \hat{h}, p)) \geq L_I(G, Q) = \mathbb{E}_{\hat{h} \sim P(\hat{h}), c \sim G(\bar{h}, \hat{h}, p)}[\log Q(\hat{h}|c)] + H(\hat{h})$. We treat $H(\hat{h})$ as constant for simplicity and arrive at the regularized InfoOIRT loss for a single student code: $\mathcal{L}_{\text{infoOIRT}} = \mathcal{L}_{\text{OIRT}} - \lambda L_I(G, Q)$. The final objective is the sum of $\mathcal{L}_{\text{infoOIRT}}$ over code submissions by all students to all problems. We refer readers to [7] for details.

Similar to OIRT, InfoOIRT also uses a static knowledge state vector $\bar{h} \in \mathcal{R}^{d_{\text{bar}}}$ for every student to represent incompressible noise. We chose to model the interpretable latent factors $\hat{h}^i$ as having d_{cont} dimensions, each having a simple Gaussian distribution, $\hat{h}^i_{\text{cont}} \sim N(\mu^i, \sigma^i)$, and d_{disc} discrete dimensions, each having a simple categorical distribution with k classes, $\hat{h}^i_{\text{disc}} \sim \text{Cat}(K = k, p = [p^i_1, \ldots, p^i_k])$. We learn the student-specific distribution parameters of each of the latent factors $\hat{h}^i$, namely the mean and standard deviation (μ^i, σ^i) for continuous dimensions, and the categorical distribution parameters $(p^i_1, \ldots, p^i_k)$ for discrete dimensions.

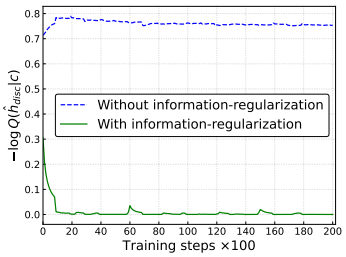

Figure 2: Mutual Information Maximization

To estimate a student's knowledge state, we sample each latent factor $\hat{h}^i$ using the current learned student-specific distribution parameters. We then concatenate the current learned student-specific $\bar{h}$ vector with this sampled $\hat{h}$ vector to get the student's final knowledge state h. We parameterize the auxiliary distribution Q as a fully connected neural network, which takes as input a representation of the generated student code c and outputs the parameters of the distribution $Q(\hat{h}|c)$. We use the mean of the hidden states of the last layer of GPT-2 as a proxy representation $r(c)$ of the generated student code c. The learnable parameters in InfoOIRT include the Q model, the student-specific incompressible knowledge state $\bar{h}$ and the student-specific distribution parameters for each dimension of the latent factors $\hat{h}$, in addition to those in OIRT.

4. EXPERIMENTS AND RESULTS

4.1 Dataset, Metrics, Implementation Details

We ground our analysis on the CSEDM dataset[2], a real-world programming education dataset containing 46,825 student code submissions from 246 college students on 50 open-ended Java programming problems collected over a semester. Following OKT [23], we quantitatively evaluate generated student code using the popular CodeBLEU [35] metric, which measures syntactic and semantic similarity between generated and actual student codes. We report the average test loss of GPT-2 across generated code tokens. To test whether InfoOIRT simply memorizes the training data, we measure diversity in the generated student codes using the dist-N metric [22], which computes the ratio of unique N-grams in the generated codes over all N-grams, with $N = 1, 2, 3$. For OIRT training, we follow the setup in [23]. For InfoOIRT, we chose to model the latent factors in student knowledge states $\hat{h}$, with 1 continuous dimension and 10 discrete dimensions, each having two classes and thereby encouraged to act as binary switches of programming knowledge mastery states/syntactic styles. We learn a student-specific 2-dimensional static knowledge state $\bar{h}$. For a fair comparison with OIRT, we learn a student-specific 23-dimensional $\bar{h}$ in OIRT and use the same hyperparameters in both models (see Appendix A).

4.2 Quantitative Results

We show the performance of both OIRT and InfoOIRT on all metrics on the CSEDM test set in Table 1. We see that InfoOIRT exhibits competitive performance to OIRT. Therefore, the addition of mutual information-based regularization in the objective provides interpretability through

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

Table 1: Experimental results on the CSEDM test set. Our InfoOIRT model is competitive with the baseline OIRT model.

Model	Knowledge State			Code Quality		Code Diversity		
	$\lvert\bar{h}\rvert$	$\lvert\hat{h}_{\text{cont}}\rvert$	$\lvert\hat{h}_{\text{disc}}\rvert$	CodeBLEU ↑	Test Loss ↓	Dist-1 ↑	Dist-2 ↑	Dist-3 ↑
Main Models								
OIRT	23	-	-	0.597	0.200	0.396	0.712	0.825
InfoOIRT	2	1	10	0.601	0.205	0.394	0.712	0.827
Ablation: Increasing Uninterpretable Knowledge Dimensions $\lvert\bar{h}\rvert$								
OIRT	64	-	-	0.609	0.202	0.399	0.717	0.830
OIRT	256	-	-	0.607	0.204	0.404	0.719	0.829
InfoOIRT	64	1	10	0.611	0.199	0.402	0.721	0.832
InfoOIRT	256	1	10	0.613	0.200	0.400	0.718	0.829
Ablation: Increasing Interpretable Knowledge Factors $\lvert\hat{h}\rvert$								
InfoOIRT	2	64	64	0.507	0.213	0.383	0.695	0.812
InfoOIRT	2	256	256	0.510	0.214	0.398	0.710	0.824
Ablation: InfoOIRT with Continuous Knowledge Factors $\lvert\hat{h}_{\text{cont}}\rvert$ or Discrete Knowledge Factors $\lvert\hat{h}_{\text{disc}}\rvert$ Only								
InfoOIRT	2	1	0	0.606	0.201	0.397	0.717	0.831
InfoOIRT	2	0	10	0.539	0.212	0.393	0.706	0.822

simple latent factors $\hat{h}$ without sacrificing code prediction accuracy. We also show various model ablations in Table 1. We observe diminishing performance returns when increasing the number of dimensions in $\bar{h}$. We therefore use a small number of dimensions in $\bar{h}$ in our models to prioritize interpretability without sacrificing performance. We also see performance degradation with a high number of interpretable knowledge factors $\hat{h}$, especially discrete knowledge factors $\hat{h}_{\text{disc}}$. Although $\hat{h}_{\text{disc}}$ provides interpretability, these factors possibly also oversimplify the model by imposing additional constraints, thereby reducing the flexibility of the model. Therefore, our choice of the number of interpretable knowledge factors reflects a balance between performance and interpretability. To test whether mutual information is maximized, in Figure 2, we show the negative log-likelihood of our Q model which is quickly minimized with an information-regularized objective indicating high mutual information between latent factors $\hat{h}$ and generated student code c. However, an equivalent InfoOIRT model without this regularization objective exhibits low mutual information.

4.3 Qualitative Results

4.3.1 *Discrete Latent Knowledge Factors*

We manipulate the learned discrete latent factors $\hat{h}_{\text{disc}}$, each from a binary categorical distribution. For each of the ten factors, we vary the binary class of that factor, keeping the remaining factors constant and set to their learned class, and analyze the resulting variation in generated code. For some discrete factors, these changes reflect different styles (indentation, spacing between function arguments), mastery of programming skills (correct or incorrect codes with different bugs), or code structure (for loops to while loops, if-else with nesting to without nesting). For example, as shown in Table 2, for one student, switching $\hat{h}^9_{\text{disc}}$ from class 0 to 1, resulted in change from code using if-else with nesting to code using if-else without nesting, while for another student, switching $\hat{h}^6_{\text{disc}}$ from class 0 to 1 resulted in change in indentation style. We note that such changes are not found in all students and all problems: problems in the CSEDM dataset cover a wide range of programming skills and many of these changes apply to few problems. For easier problems with shorter student codes, we observe minimal variation since InfoOIRT is often able to predict the exact student code written (which is correct). In these cases, InfoOIRT prioritizes the $\mathcal{L}_{\text{OIRT}}$ loss for generation performance and possibly ignores the information regularization for these student-problem pairs. Compared to InfoGAN [7] showing consistent variations in an unsupervised hand-written digit generation setting, we hypothesize that capturing variations in generated code across multiple problems on different topics is harder. Since different students attempt different problems, InfoOIRT learns a student-specific $\hat{h}_{\text{disc}}$ distribution where the effect of each dimension changes depending on the attempted problems.

4.3.2 *Continuous Latent Knowledge Factors*

We manipulate the learned continuous latent factor $\hat{h}_{\text{cont}}$ (with a range between -3.5 to 3.5) for different ranges and investigate the resulting change in generated code. We do not observe any change for small variations (-2 to 2), showing the robustness of the InfoOIRT, balancing generation accuracy and interpretability. For larger variations (-5 to 5), we see new codes generated with variation in either style, correctness, bug type, or structure, for some students in some problems. For bugs, these changes in the continuous latent variable overlap with flipping certain binary classes among the discrete factors. This observation reflects the nature of code being more discrete rather than continuous. For extreme variations (-10 to 10) that extrapolate $\hat{h}_{\text{cont}}$ beyond learned values, we observe that InfoOIRT still generates coherent student code but interestingly, to a *different problem*. We see that student code for easier problems with conditionals changes to code for harder problems with looping, and vice versa, as shown in Table 2. This observation suggests that $\hat{h}_{\text{cont}}$ could be discretizing the student's knowledge space of unique code constructs across problem types.

4.4 Potential Use Cases in CS Education

Predicting free-form student responses to open-ended problems provides educators a deeper insight into a student's reasoning process [2, 5, 11, 45] through their knowledge states. Doing so can potentially shed light on the typical errors among students before even assigning them, which enables educators to anticipate and prepare corresponding feedback. With informative and interpretable latent states, it can be easier for intelligent tutoring systems to support educators by summarizing common bugs and coding styles among students. Information on latent factors that indicate student bugs can potentially be used to quantize the effectiveness of additional instruction on different topics on helping students correct errors, which may help educators plan their activities. Moreover, for latent codes that we uncover to associate with specific student bugs, we can explore using them to provide progressive edit suggestions, by gradually changing

Table 2: Variation in generated code for variation in $\hat{h}_{\text{disc}}$ and $\hat{h}_{\text{cont}}$.

Switching $\hat{h}^{9}_{\text{disc}}$ from 0 to 1 results in change in if-else nesting on a subset of problems attempted by one student.

$\hat{h}^{9}_{\text{disc}} = 0$ Generates Code With If-Else Nesting

```
public int caughtSpeeding(int speed, boolean isBirthday)
{
    if (isBirthday)
        if (speed <= 65)
            return 0;
        else if (speed >= 66 && speed <= 85)
            return 1;
        else
            return 2;
    else
        if (speed <= 60)
            return 0;
        else if (speed >= 61 && speed <= 80)
            return 1;
        else
            return 2;
}
```

$\hat{h}^{9}_{\text{disc}} = 1$ Generates Code Without If-Else Nesting

```
public int caughtSpeeding(int speed, boolean isBirthday)
{
    if(isBirthday)
        speed -= 5;
    if(speed <= 60)
        return 0;
    else if(speed <= 80)
        return 1;
    else
        return 2;
}
```

```
public boolean cigarParty(int cigars, boolean isWeekend)
{
    if (isWeekend)
        if (cigars >= 40)
            return true;
        else
            return false;
    else
        if (cigars >= 40 && cigars <= 60)
            return true;
        else
            return false;
}
```

```
public boolean cigarParty(int cigars, boolean isWeekend)
{
    if(isWeekend)
        return (cigars >= 40);
    return (cigars >= 40 && cigars <= 60);
}
```

Switching $\hat{h}^{6}_{\text{disc}}$ from 0 to 1 results in change in indentation style on a subset of problems attempted by one student.

$\hat{h}^{6}_{\text{disc}} = 0$ Generates Code with "K&R" Indentation Style

```
public boolean isEverywhere(int[] nums, int val)
{
    for (int i = 0; i < nums.length; i++) {
        if (nums[i]!= val && nums[i+1]!= val) {
            return false;
        }
    }
    return true;
}
```

$\hat{h}^{6}_{\text{disc}} = 1$ Generates Code with "Allman" Indentation Style

```
public boolean isEverywhere(int[] nums, int val)
{
    for (int i = 0; i < nums.length; i++)
    {
        if (nums[i]!= val && nums[i+1]!= val)
        {
            return false;
        }
    }
    return true;
}
```

```
public int makeChocolate(int small, int big, int goal)
{
    int maxBig = goal/5;
    if (maxBig <= big) {
        goal -= maxBig*5;
    }
    else {
        goal -= big*5;
    }
    if (goal <= small) {
        return goal;
    }
    return -1;
}
```

```
public int makeChocolate(int small, int big, int goal)
{
    int maxBig = goal/5;
    if (maxBig <= big)
    {
        goal -= maxBig*5;
    }
    else
    {
        goal -= big*5;
    }
    if (goal <= small)
    {
        return goal;
    }
    return -1;
}
```

Extreme variation in $\hat{h}_{\text{cont}}$ results in code with conditionals for easier problems shifting to code with loops for harder problems, and vice versa.

Student Code for Problems with Conditional Constructs

```
public boolean squirrelPlay(int temp, boolean isSummer)
{
    if(isSummer)
        return (temp >= 60 && temp <= 100);
    return (temp >= 60 && temp <= 90);
}
```

Student Code for Problems with Looping Constructs

```
public boolean xyBalance(String str)
{
    int len = str.length() - 1;
    char ch;
    for(int i = len; i >= 0; i--)
    {
        ch = str.charAt(i);
        if(ch == 'x')
            return false;
        else if(ch == 'y')
            return true;
    }
    return true;
}
```

the continuous latent variables to generate code between a student's original buggy code and correct code, which can be both informative and relatable to the student.

5. CONCLUSIONS AND FUTURE WORK

We presented a first step towards interpreting latent student knowledge states in models of open-ended responses in programming education. We proposed InfoOIRT, an open-ended IRT model that accurately predicts student-written code, validated on the real-world CSEDM dataset, along with interpretable latent student knowledge states. Through qualitative analysis, we presented examples of latent student knowledge states capturing salient syntactic and semantic features including style, mastery of programming skills, and code structure, demonstrating the potential of InfoOIRT in CS education. InfoOIRT should be considered exploratory with limitations and several avenues for future work. First, we can explore adapting InfoOIRT to knowledge tracing and impose constraints on the consistency of these states over time, using ideas from cognitive modeling [42]. Second, we can reduce potential biases toward underrepresented students by minimizing the mutual information between demographic variables and student-written code. Third, we can explore applying InfoOIRT to other domains including language learning [21], and mathematics [39].

6. ACKNOWLEDGEMENTS

We thank Alexander Scarlatos and the anonymous reviewers for their helpful comments. We thank the NSF for partially supporting this work under grants DUE-2215193 and IIS-2237676.

7. REFERENCES

[1] U. Z. Ahmed, M. Christakis, A. Efremov, N. Fernandez, A. Ghosh, A. Roychoudhury, and A. Singla. Synthesizing tasks for block-based programming. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[2] J. R. Anderson and R. Jeffries. Novice lisp errors: Undetected losses of information from working memory. *Human–Computer Interact.*, 1(2):107–131, 1985.

[3] Y. Attali and J. Burstein. Automated essay scoring with e-rater® v. 2. *The Journal of Technology, Learning and Assessment*, 4(3), 2006.

[4] F. B. Baker. *The Basics of Item Response Theory*. ERIC Clearinghouse on Assessment and Evaluation, 2001.

[5] J. S. Brown and R. R. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cogn. sci.*, 2(2):155–192, 1978.

[6] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[7] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[8] B. Choffin, F. Popineau, Y. Bourda, and J.-J. Vie. DAS3H: Modeling student learning and forgetting for pptimally scheduling distributed practice of skills. In *International Conference on Educational Data Mining (EDM)*, pages 29–38, 2019.

[9] A. Corbett and J. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Model. User-adapted Interact.*, 4(4):253–278, Dec. 1994.

[10] G. Desjardins, A. Courville, and Y. Bengio. Disentangling factors of variation via generative entangling. *arXiv preprint arXiv:1210.5474*, 2012.

[11] M. Q. Feldman, J. Y. Cho, M. Ong, S. Gulwani, Z. Popović, and E. Andersen. Automatic diagnosis of students' misconceptions in K-8 mathematics. In *Proc. CHI Conf. Human Factors Comput. Syst.*, pages 1–12, 2018.

[12] N. Fernandez, A. Ghosh, N. Liu, Z. Wang, B. Choffin, R. Baraniuk, and A. Lan. Automated scoring for reading comprehension via in-context bert tuning. In *International Conference on Artificial Intelligence in Education (AIED)*, pages 691–697. Springer International Publishing, 2022.

[13] A. Ghosh, J. Raspat, and A. Lan. Option tracing: Beyond correctness analysis in knowledge tracing. In *International Conference on Artificial Intelligence in Education (AIED)*, pages 137–149. Springer, 2021.

[14] K. Hanawa, R. Nagata, and K. Inui. Exploring methods for generating feedback comments for writing learning. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9719–9730, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics.

[15] H. Heickal and A. Lan. Generating feedback-ladders for logical errors in programming using large language models. *International Conference on Educational Data Mining (EDM)*, 2024.

[16] J. Kolb, S. Farrar, and Z. A. Pardos. Generalizing expert misconception diagnoses through common wrong answer embedding. *Int. Educ. Data Mining Soc.*, 2019.

[17] C. Koutcheme, S. Sarsa, J. Leinonen, A. Hellas, and P. Denny. Automated program repair using generative models for code infilling. In *International Conference on Artificial Intelligence in Education (AIED)*, 2023.

[18] N. A. Kumar and A. Lan. Improving socratic question generation using data augmentation and preference optimization. *Proceedings of the 19th Workshop on Innovative Use of NLP for Building Educational Applications (BEA)*, 2024.

[19] N. A. Kumar and A. Lan. Using large language models for student-code guided test case generation in computer science education. *AI4ED workshop at AAAI Conference on Artificial Intelligence*, 2024.

[20] A. S. Lan, C. Studer, A. E. Waters, and R. G. Baraniuk. Tag-aware ordinal sparse factor analysis for learning and content analytics. In *International Conference on Educational Data Mining (EDM)*, 2013.

[21] J. Lee and A. Lan. Smartphone: Exploring keyword mnemonic with auto-generated verbal and visual cues. In *International Conference on Artificial Intelligence in Education (AIED)*, pages 16–27. Springer Nature Switzerland, 2023.

[22] J. Li, M. Galley, C. Brockett, J. Gao, and W. B. Dolan. A diversity-promoting objective function for neural conversation models. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, pages 110–119, 2016.

[23] N. Liu, Z. Wang, R. Baraniuk, and A. Lan. Open-ended knowledge tracing for computer science education. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[24] T. Long, Y. Liu, J. Shen, W. Zhang, and Y. Yu. Tracing knowledge state with individual cognition and acquisition estimation. In *ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021.

[25] Y. Mao, Y. Shi, S. Marwan, T. W. Price, T. Barnes, and M. Chi. Knowing when and where: Temporal-astnn for student learning progression in novice programming tasks. *Int. Educ. Data Mining Soc.*, 2021.

[26] H. McNichols, M. Zhang, and A. Lan. Algebra error classification with large language models. In *International Conference on Artificial Intelligence in Education (AIED)*, pages 365–376. Springer Nature Switzerland, 2023.

[27] G. Mercatali and A. Freitas. Disentangling generative factors in natural language with discrete variational autoencoders. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2021.

[28] R. Ostini and M. L. Nering. *Polytomous item response theory models*. Sage, 2006.

[29] Y. Ozuru, S. Briner, C. A. Kurby, and D. S. McNamara. Comparing comprehension measured by multiple-choice and open-ended questions. *Canadian Journal of Experimental Psychology*, 67(3):215, 2013.

[30] S. Pandey and J. Srivastava. Rkt: relation-aware self-attention for knowledge tracing. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management (CIKM)*, pages 1205–1214, 2020.

[31] P. Pavlik Jr, H. Cen, and K. Koedinger. Performance factors analysis–A new alternative to knowledge tracing. In *International Conference on Artificial Intelligence in Education (AIED)*, 2009.

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

[33] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *International Conference on Learning Representations (ICLR)*, 2016.

[34] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[35] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[36] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education (IJAIED)*, 27(1), 2017.

[37] R. D. Roscoe, L. K. Varner, S. A. Crossley, and D. S. McNamara. Developing pedagogically-guided algorithms for intelligent writing feedback. *International Journal of Learning Technology 25*, 8(4):362–381, 2013.

[38] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *ACM Conference on International Computing Education Research (ICER)*, 2022.

[39] A. Scarlatos and A. Lan. Tree-based representation and generation of natural and mathematical language. In *61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 3714–3730. Association for Computational Linguistics, July 2023.

[40] A. Scarlatos, D. Smith, S. Woodhead, and A. Lan. Improving the validity of automatically generated feedback via reinforcement learning. *International Conference on Artificial Intelligence in Education (AIED)*, 2024.

[41] Y. Shi, M. Chi, T. Barnes, and T. Price. Code-dkt: A code-based knowledge tracing model for programming tasks. *International Conference on Educational Data Mining (EDM)*, 2022.

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

[43] D. Shin, Y. Shim, H. Yu, S. Lee, B. Kim, and Y. Choi. Saint+: Integrating temporal features for EdNet correctness prediction. In *International Learning Analytics and Knowledge Conference (LAK)*, 2021.

[44] A. Singla and N. Theodoropoulos. From Solution synthesis to Student Attempt synthesis for block-based visual programming tasks. In *International Conference on Educational Data Mining (EDM)*, 2022.

[45] J. P. Smith III, A. A. DiSessa, and J. Roschelle. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The journal of the learning sciences*, 3(2):115–163, 1994.

[46] Y. Su, Q. Liu, Q. Liu, Z. Huang, Y. Yin, E. Chen, C. Ding, S. Wei, and G. Hu. Exercise-enhanced sequential modeling for student performance prediction. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018.

[47] K. Taghipour and H. T. Ng. A neural approach to automated essay scoring. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1882–1891, Austin, Texas, Nov. 2016. Association for Computational Linguistics.

[48] W. J. van der Linden and R. K. Hambleton. *Handbook of Modern Item Response Theory*. Springer Science & Business Media, 2013.

[49] Y. Wang and N. Heffernan. Extending knowledge tracing to allow partial credit: Using continuous versus binary nodes. In *International Conference on Artificial Intelligence in Education (AIED)*, 2013.

[50] J. Zhang, X. Shi, I. King, and D.-Y. Yeung. Dynamic key-value memory networks for knowledge tracing. In *International Conference on World Wide Web (WWW)*, pages 765–774, Apr. 2017.

[51] M. Zhang, S. Baral, N. Heffernan, and A. Lan. Automatic short math answer grading via in-context meta-learning. *International Conference on Educational Data Mining (EDM)*, 2022.

[52] R. Zhu, D. Zhang, C. Han, M. Gao, X. Lu, W. Qian, and A. Zhou. Programming knowledge tracing: A comprehensive dataset and a new model. *International Conference on Data Mining Workshops (ICDMW)*, 2021.

APPENDIX

A. EXPERIMENTS

A.1 Dataset

We ground our analysis on the real-world programming education dataset from the 2nd CSEDM Data Challenge, which we referred to as the CSEDM dataset.[3] This dataset contains 46,825 student code submissions from 246 college students on 50 open-ended Java programming problems collected over an entire semester. We analyze the first submission to each problem and ignore later attempts since this setting captures a student's overall mastery of programming concepts while later attempts also capture debugging skills that we do not analyze in this work. We preprocess the dataset by removing 15% of code submissions that cannot be converted to an abstract syntax tree (AST) and split it into train-validation-test with $80\% - 10\% - 10\%$ proportion.

A.2 Metrics

Following OKT [23], we quantitatively evaluate generated student code using the popular metric CodeBLEU [35], which measures syntactic and semantic similarity between generated and actual student codes. We also report the average test loss of GPT-2 across generated code tokens using the model with the lowest validation loss with a lower test loss being predictive of better student code generation performance. To test whether OIRT simply memorizes the training data, we measure diversity in the generated student codes using the dist-N metric [22], which computes the ratio of unique N-grams in the generated codes over all N-grams, with $N = 1, 2, 3$.

A.3 Implementation Details

For OIRT training, we follow the setup in [23] and use a batch size of 8, an AdamW optimizer with a learning rate of $1 \cdot 10^{-5}$ with linear learning rate scheduler with warmup for GPT-2 model parameters, and the Adam optimizer with a learning rate of $1 \cdot 10^{-3}$ for the alignment function and student specific $\bar{h}$ knowledge states. We finetune OIRT for 50 epochs which takes around 4 hours on a single NVIDIA A100 GPU, and chose the model with the lowest validation loss.

For InfoOIRT, we chose to model the latent factors in student knowledge states $\hat{h}$, with 1 continuous dimension and 10 discrete dimensions, each having two classes and thereby encouraged to act as binary switches of programming knowledge mastery states/syntactic styles. We learn a student-specific 2-dimensional static knowledge state $\bar{h}$. For a fair comparison with OIRT, we learn a student-specific 23 dimensional $\bar{h}$ in OIRT and use the same hyperparameters in both models. Since our goal is to analyze the change in code generated with respect to variation in latent factors $\hat{h}$ only, to remove randomness during inference, we use greedy decoding to generate student code in both models.

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