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 or even code, serve as a highly meaningful form of assessment and complements closed-form problems such as multiple-choice questions. 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 or generally lacking sufficient knowledge. Until recently, however, research has mostly focused on the automated scoring of open-ended responses, either via classification methods or clustering, and providing corresponding feedback. 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), knowledge tracing, and factor analysis 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) 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, 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), 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:
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 \(^{29}\). In the programming domain, \(^{23, 52}\) 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) \(^7\) 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’s knowledge state from the set of student code submissions \(c_i\) to problems \(p_j\) 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., \(c_{jk} = \text{RG}(p_k, h_j)\). 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 \(^{24}\) 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., \(\bar{p}_m \in \mathbb{R}^{768}\) 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 \(c_{ij}\) by student \(j\) with knowledge state \(h_j\) to problem \(p_k\), consisting of \(N\) tokens, is: \(\mathcal{L}_{\text{OIRT}} = \sum_{n=1}^{N} -\log P(c_{ij}^{n}\vert p_k, h_j, \{c_{ik}^{m}\}_{m=1}^{n-1})\), where \(\theta\) 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
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) $h$, which represents the incompressible student knowledge state, and 2) $\bar{h}$, which consists of simple and interpretable latent factors $h^1, \ldots, h^K$, to represent the salient structural features of student written code. However, GPT-2 could ignore these additional latent factors $\bar{h}$ and simply generate student code $c$ with a probability distribution satisfying $P(c|h) = P(c)$. We, therefore, impose an information-theoretic regularization to encourage high mutual information between latent factors $\bar{h}$ and the GPT-2 generator distribution $G(\bar{h}, h, p)$ that generates student code $c$ corresponding to a student with knowledge state $\bar{h}$. 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 $I(\bar{h}, G(\bar{h}, h, p)) = H(\bar{h}) - H(\bar{h} | G(\bar{h}, h, p))$. Intuitively, mutual information is maximized when the uncertainty, i.e., entropy, in $\bar{h}$ given $c$ is minimized, meaning that the information in the simple latent knowledge factors $\bar{h}$ 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 $P(h|c)$. Therefore, we use a variational lower bound of mutual information following [7]:

$$I(\bar{h}, G(\bar{h}, h, p)) \geq \mathbb{E}_{\bar{h} \sim P(\bar{h})} \mathbb{E}_{c \sim G(h, \bar{h}, p)} \left[ \log Q(\bar{h}|c) \right] + H(\bar{h}).$$

We treat $H(\bar{h})$ as constant for simplicity and arrive at the regularized InfoOIRT loss for a single student code: $\mathcal{L}_{\text{infoOIRT}} = \mathcal{L}_{\text{OIRT}} - AL_{I}(G, Q)$. 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 $h \in \mathbb{R}^{d_{\text{out}}}$ for every student to represent incompressible noise. We chose to model the interpretable latent factors $\hat{h}^i$ as having $d_{\text{cont}}$ continuous dimensions, each having a simple Gaussian distribution, $\hat{h}_{\text{cont}}^i \sim N(\mu^i, \sigma^i)$, and $d_{\text{disc}}$ discrete dimensions, each having a simple categorical distribution with $K$ classes, $\hat{h}_{\text{disc}}^i = \text{Cat}(K = k, p = [p^1, \ldots, p^K])$. We learn the student-specific distribution parameters of each of the latent factors $\hat{h}^i$, namely the mean and standard deviation $(\mu^i, \sigma^i)$ for continuous dimensions, and the categorical distribution parameters $(p^1, \ldots, p^K)$ for discrete dimensions.

4. EXPERIMENTS AND RESULTS

4.1 Dataset, Metrics, Implementation Details
We ground our analysis on the CSEDM dataset [4], 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 OIRT [23], we quantitatively evaluate generated student code using the popular CodeBLEU [65] 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 $h$, 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

---

1. https://sites.google.com/ncsu.edu/csedm-dc-2021
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 $\hat{h}$. We therefore use a small number of dimensions in $\hat{h}$ 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 $\hat{h}_{disc}$. Although $\hat{h}_{disc}$ 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 $\hat{h}$ 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 $h$ 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 $\hat{h}_{disc}$, 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 $\hat{h}_{disc}$ 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 $\hat{h}_{disc}$ 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 $L_{OIRT}$ 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 $h_{disc}$ 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 $\hat{h}_{cont}$ (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 $\hat{h}_{cont}$ 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 $\hat{h}_{cont}$ 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 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 quantify 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 1: Experimental results on the CSEDM test set. Our InfoOIRT model is competitive with the baseline OIRT model.

<table>
<thead>
<tr>
<th>Model</th>
<th>Knowledge State</th>
<th>Code Quality</th>
<th>Code Diversity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>\hat{h}</td>
<td>$</td>
</tr>
<tr>
<td>OIRT</td>
<td>23</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>InfoOIRT</td>
<td>2</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InfoOIRT</td>
<td>64</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>InfoOIRT</td>
<td>256</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>InfoOIRT</td>
<td>64</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>InfoOIRT</td>
<td>256</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Ablation: Increasing Uninterpretable Knowledge Dimensions $|\hat{h}|$

| InfoOIRT | 2 | 64 | 64 | 0.507 | 0.213 | 0.383 | 0.695 | 0.812 |
| InfoOIRT | 2 | 256 | 256 | 0.510 | 0.214 | 0.384 | 0.719 | 0.824 |

Ablation: Increasing Interpretable Knowledge Factors $|\hat{h}|$

| InfoOIRT | 2 | 1 | 0 | 0.606 | 0.201 | 0.397 | 0.717 | 0.831 |
| InfoOIRT | 2 | 0 | 10 | 0.539 | 0.212 | 0.393 | 0.736 | 0.822 |
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].
7. REFERENCES


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. 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 $\overline{h}$ 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 $\overline{h}$. For a fair comparison with OIRT, we learn a student-specific 23 dimensional $\overline{h}$ 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.

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