Federal State Autonomous Educational Institution for Higher Education
National Research University
Higher School of Economics

Faculty of Computer Science
BSc Applied Mathematics and Information Science

BACHELOR'S THESIS

Research project

Profiling of Students' Competencies for Adaptive Learning Systems

Submitted by Dobrosovestnov Ivan Andreevich
student of group 193, 4th year of study,

Approved by Supervisor:
Ignatov Dmitry Igorevich
Department of Data Analysis and Artificial Intelligence

Moscow 2023

Contents

1 Annotation

Online education has emerged as a popular learning style, thanks to the proliferation of the Internet and mobile communication technology. Unlike traditional education, online education offers flexibility and convenience as it enables teaching and learning to occur anytime and anywhere. With the help of online learning systems, high-quality and equitable education can be provided to students, giving them an optimal and adaptive learning experience. Online learning systems continuously record a massive amount of data about student-system interactions, which can be further mined to assess their knowledge levels and learning preferences. Knowledge Tracing (KT) is a crucial research field for online education that utilizes machine learning methods to monitor students' dynamic knowledge states by exploiting educationally related data. KT models enable the development of personalized adaptive learning systems, and students can better understand their learning process, paying attention to skills with poor mastery. This paper presents our method which tracks programming skills of high school students who took the Python programming language course on our educational platform.

2 Keywords

Education, Personalized Learning, Knowledge Tracing, Deep Learning, Transformer, Code Embeddings

3 Introduction

The COVID-19 pandemic has significantly impacted the world, resulting in a growing interest in online learning. However, online learning presents a challenge in effectively tracking a student's progress and analyzing their strengths and weaknesses. Our educational platform has thousands of students who solves programming tasks in different languages and follow different educational tracks.

Teachers of the platform give lessons for pupils, and with the increase in interest in programming and the number of students, it has become increasingly difficult for teachers to track the progress of each pupil. This reduction in personalized feedback impacts not only our school but any educational platform.

With the large digital footprint left by pupils on the platform (code submissions, verdict of the checking system, and task completion time), we have the opportunity to automate the personalization of learning, making it more comfortable and providing more feedback throughout the learning process. To achieve this, we use an adaptive knowledge tracing system based on a transformer neural network.

Several studies have explored the learning process for similar educational platforms, and we plan to conduct research and compare the newest approaches for analyzing student code and task text in detail to present our version of a model for tracking the educational process based on the transformer model. We aim to explore different ways of applying this model to improve personalization. Our educational platform contains a huge amount of data, which is a critical aspect of quality model training. We plan to use this data to develop novel methods for monitoring student knowledge and to provide personalized learning experiences.

In this article, we will present the results of our research and compare our approach to existing methods for tracking student knowledge. We will also discuss the implications of our findings and explore potential future research directions in the field of personalized learning in the context of online education.

4 Related works

4.1 Knowledge Tracing Models

Knowledge tracing (KT) is an increasingly popular field in education technology, with many companies looking to make their learning more personalized and effective [1]. While the field has gained momentum in recent years, its roots

can be traced back to Bayesian methods in the last century [2]. Initially, this task was solved by creating knowledge states within an latent space that dynamically updated as students received new results. As machine learning gained popularity, logistic models [3] emerged, using various student interactions with the platform as features, and then using logistic regression to predict the probability of mastering a particular skill.

The introduction of deep learning, specifically recurrent neural networks (RNN) and long short-term memory (LSTM) models, marked a significant breakthrough in this field. By tracking a student's sequence of actions, the neural network could predict their subsequent results, forming an implicit space of the student's current competencies and updating them with each new result [4, 5].

The next stage in KT technology development was the attention layer [6]. With this layer, each task that a student solves contributes to the development of his competencies, and recurrent models were able to better understand which tasks have a greater impact on enhancing specific competencies [7]. With the advent of transformers and language models, KT models have become more flexible and effective. For example, in [8], the authors went beyond constructing a map of a student's progress and synthesized their future solutions using the GPT-2 model [9]. The interpretation of problems, that students solve, has also played a significant role. While tasks were previously represented by an index, language models can now extract information from the text of the task, including keywords related to what is required of the student and what skills they need to apply to solve the problem. In [10], the authors discussed which features should be collected to improve the quality of KT, including task category, position in the educational process, and time spent by the student solving the task. In [11], an improved version of the method was proposed, adding elapsed and lag embeddings, which showed state-of-the-art performance in knowledge tracing.

4.2 Code Representation Models

As Deep Learning became popular, research on code as vector representations has emerged. These studies cover a broad spectrum of tasks, such as bug detection, vulnerability detection, code efficiency enhancement, docstring summarization, and more. There are numerous approaches to extract semantic features from code.

4.2.1 AST Tree Representation

One of the most common approaches is to represent code as an Abstract Syntax Tree (AST). This representation maintains invariance to variable names in the code and takes into account the nesting of structures. Work [12] used this approach to extract AST paths from the AST tree, which were subsequently encoded using the Bag of Words method. However, AST trees contain a lot of irrelevant information that is not interpreted in semantic code analysis. In response, [13] proposed the ASTNN model, which divides the AST tree into subtrees and then passes them through attention layers to filter them. This approach was adopted by the authors of [8] to transform student solutions into vector representations for Knowledge Tracing.

4.2.2 Control Flow Graph

Another approach involves constructing a static workflow graph for a program. It allows code to be represented as a graph and then applies graph feature extraction methods to it. In [14], the code is represented as a Control Flow Graph (CFG), which is then pass to the DeepWalk process to form embeddings at vertices.

4.2.3 Language Models

With the widespread adoption of the BERT model [15], language models are increasingly being applied to code. In this approach, the code is tokenized

and pass to the language model. The model is trained using masking and token prediction methods. Such models are also used in conjunction with texts [16, 18].

5 Overview

In this article, we focus on the analysis of programming code for profiling the competencies of students on our educational platform. The problem at hand is to utilize the digital footprint left by the students on the platform, with the model predicting the student's future performance while being interpretable for the use of hidden representations in an adaptive learning system (personalized task selection, identifying frequent mistakes, adapting the methodology program).

5.1 Digital Footprint

During the learning process, our educational platform logs the actions of students in the system. While solving tasks, students send multiple submissions that they consider correct. During the task-solving process, a student can send up to 100 submissions to the system. Each submission is saved in the database with the corresponding metadata (ProblemID, UserID, timestamp). Additionally, the system logs the result of the submission check, namely, the compiler's verdict, the number of tests passed, submission time, penalty points for submission, type of work performed (classroom, homework, test), and more. Apart from tasks, the teacher assigns tests to the student, representing a question and multiple-choice answers. On the educational platform, the student can access the presentation for the lesson and view the materials left by the teacher. All of these data are logged in the database for subsequent analysis. We believe that the large volume of heterogeneous data will improve the quality of the KT model.

5.2 Code Analysis

A critical component of the successful profiling of a student's competencies is the qualitative interpretation of their programming code submissions. The semantic interpretation of code is a problematic area, as its invariance allows creating an infinite number of semantically equivalent solutions to a task. With the increasing complexity of the task, the number of possible solutions also increases. Different students solve the task differently, and KT should consider this. Among the code-to-vector approaches, CodeBERT[16], GraphCodeBERT[17], and UniXcoder[18] are the most prevalent. However, even they poorly recognize semantic analysis of code, as they are not trained for this task. The variety of variable names, equivalent constructions in conditions and loops, can be misleading. Among the new methods, one of the most well-known is contrastive learning [19, 20]. The method's essence is to train the model using a contrastive loss to recognize semantically identical code constructions. This training is performed by replacing variable names, transforming conditions and loops, and adding noisy constructions that do not affect the code's workflow.

5.3 Transformer KT model

In this study, we propose a Transformer Knowledge Tracing (TKT) model that is based on the Transformer Encoder-Decoder architecture. This model is capable of taking in two sequences as input: a sequence of condition features, which will be passed through the encoder, and a sequence of student responses, which will be passed through the decoder. We also aim to explore a variation of the model that includes an additional encoder block to capture the intermediate actions of the student, such as viewing a presentation or communicating with a teacher through the educational platform.

To represent code, we will utilize a series of language models such as BERT [15], CodeRetriever[19] and Ast2vec [21]. For the representation of task descriptions, we will use GPT-2[9] and modification - GPTBigCode [22]. The objective

of this model is to predict the outcome of the next task, as well as to generate a vector representation of the student's code for the next task.

6 Methodology

6.1 Data preprocessing

We decided to investigate the Python programming language, as it is one of the most popular languages for researching code semantics. Additionally, Python allows for easy conversion of code to Abstract Syntax Trees (AST) for studying structure. In our study, we used data from students who completed a Python course between 2018 and 2022. The course continues for one year, comprising 60 academic hours. We selected users who fully completed the course and solved at least 50 problems. We also removed rare problems that were assigned to strong or weak students on an individual basis.

For the textual descriptions of the problems, we also provided data cleaning by removing markdown, HTML, and Latex markup. As a result, we obtained approximately 700,000 submissions and 3,000 unique students who took the course during the chosen time period.

Constructing the dataset, we encountered a problem - different students have different behaviors when solving problems. Some students think for a long time and submit a well-analyzed solution, which allows them to get 100 points with just 1-2 submissions per problem. Other pupil submit every code correction without prior analysis, which means they may submit 10-20 submissions per problem. This case poses a problem of consistency in time intervals: for example, for a student who made thoughtful submissions, a history of 500 submissions may cover 200 solved problems, while for a student who relied on luck, only 50 solved problems may be covered. To address this issue, similar to the approach in the Open-Ended Knowledge Tracing article [8], we decided to consider several different data aggregations:

- Take all submissions of a student for a problem
- Take the last 3 submissions of a student for a problem
- Take the last 2 submissions of a student for a problem

We considered not to take the last submission of a student for a problem because students rarely leave problems unsolved and more often try to complete them to get a good grade. Therefore, the last submission of a student almost always receives 100 points.

6.2 Code Processing

To understand the process of material learning by students, it is necessary to understand the semantic component of the code. There are many solutions for all the tasks that our school offers students. These solutions depend on the teacher who conducts the lesson and on the current level of knowledge (the syntactic constructs that students have studied on previous lessons and are able to apply). In addition, the recognition of code is limited by the "failure when running the code" restriction. Students often submit syntactically correct solutions to the system that generate errors after they are run (such as TimeLimit, MemoryLimit, RuntimeError). Therefore, the method chosen by us should recognize code that has such errors, otherwise, a large part of the attempts will simply be removed from the dataset.

In our research, we will consider 3 methods of representing code:

- AST tree - Ast2vec method [21], Unixcoder [18]
- Text - CodeBert [16]
- Graph - GraphCodeBert [17]

Except Ast2vec [21], all models are based on the transformer model [6], but they were trained in different ways on different tasks. The Ast2vec model differs,

that it takes into account only the AST tree and recurrently processes it with GRU layers [23], from the root vertex to the leaf.

All the presented methods are static and will not be trained together with the main model. We decided to do this, firstly, because some models have a large number of parameters, secondly, because the main model will be strongly overfitted to the domain of a specific course that students have taken, and transfer-learning to other courses will become impossible.

In addition, there are many solutions for each task that depend on the teacher and the level of knowledge of the students. Therefore, our methods need to be flexible enough to accommodate variations in the code and accurately recognize the underlying semantic structure.

To determine which model is best suited for our task, we decided to study how well they understand the semantics of the code. To do this, we formed a small dataset of popular student solutions and formulated a subtask solution:

Let M be the observed model, $M(c) \in R_n$, where c is the student's code, and n is the size of the embedding space into which the sequence of code tokens c is projected, C is the set of student solutions. $R(x_{in}, c) = x_{out}$, where R is an executable binary file, and $x_{in} \in X_{in}$ - the set of all possible input data in the program.

Then, we call a pair (c_i, c_j) - isomorphic if

$$R(x_{input}, c_i) = R(x_{input}, c_j) \forall x_{in} \in X$$

To multiply our pairs, we created synthetic data using the Refactory article [27], where the authors describe the rules for isomorphic code constructs. An example of an isomorphic and non-isomorphic pair (figure 6.1)

```
for i in range(10):
    print(i)
```

```
i = 0
while i < 10:
    print(i)
```

(a) Isomorphic pair

```
if a > b:
    print(a)
```

```
if b > a:
    print(a)
```

(b) Non Isomorphic pair

Figure 6.1: Example of pairs

Let $C_{\text{isomorphic}}$ - the set of all isomorphic pairs and $C_{\text{non\_isomorphic}}$ - the set of all non-isomorphic pairs. From the above statements, we formulate the rule by which we want to identify the best model:

$$\text{score}_{\text{positive}} = \sum_i \sum_j \frac{\cos(M(c_i), M(c_j))}{\|C_{\text{izomorf}}\|} \to min \quad \forall (c_i, c_j) \in C_{\text{izomorf}}$$

$$\text{score}_{\text{negative}} = \sum_i \sum_j \frac{\cos(M(c_i), M(c_j))}{\|C_{\text{non\_izomorf}}\|} \to max \quad \forall (c_i, c_j) \in C_{\text{non\_izomorf}}$$

We also chose parameter: the number of trained parameters in the model. This parameter is necessary to evaluate the speed of converting code to embedding representation. For better understanding, we named the positive_score $= 1 -$ $\text{score}_{\text{positive}}$, the value for the isomorphic group and negative_score $= \text{score}_{\text{negative}}$, for the non-isomorphic group. Thus both of scores need to be maximized.

The figure 6.2 shows that the Ast2Vec model slightly lags behind UnixCoder and CodeBert in similarity of isomorphic structures, but significantly outperforms them in distinguishing non-isomorphic structures. We also take into account the fact that the size of the Ast2vec model is several times smaller smaller than in the others, which means that this model is best suited for our task.

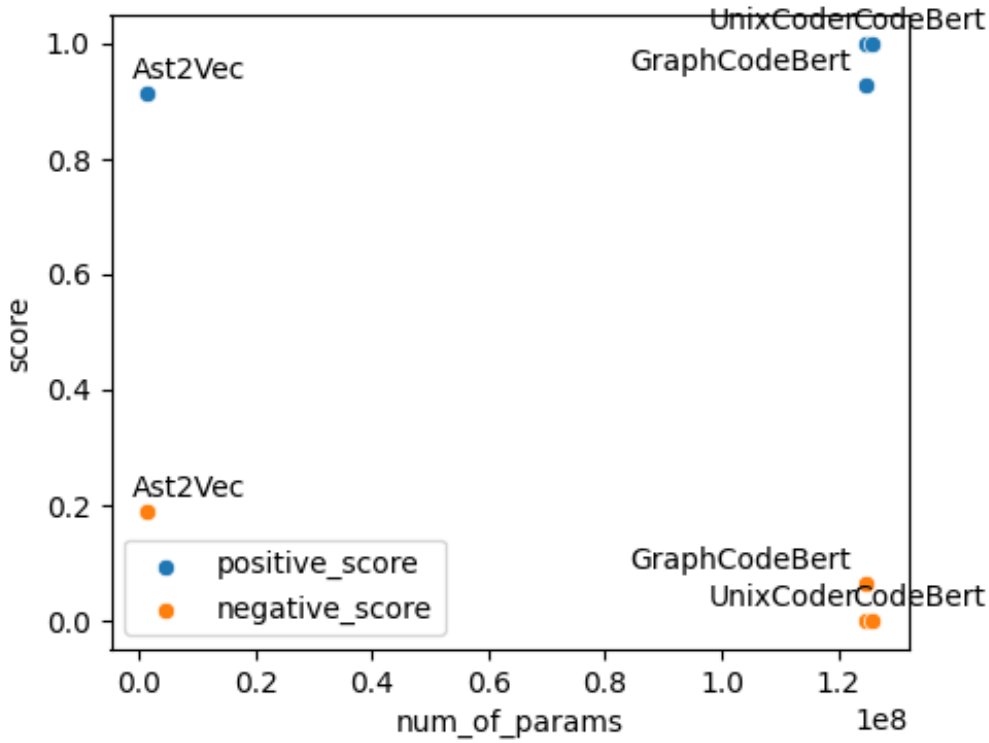

Figure 6.2: Embeddings score

6.3 Text Processing

Text data of problem statements is an important component of the Transformer Knowledge Tracing model. Thanks to this data, we can extract information about the material that the student is studying. In our work, we use two variations of the GPT-2 model: the standard [9] and the GPTBigCode [22]. GPTBigCode differs in that it was trained on the task of text-to-code. In article authors described how they selected repositories from GitHub that had at least 5 stars, assuming that these repositories would contain higher-quality code along with documentation. Since the text of our problems has specific content and contains various entities from the programming field, we believe that using this model will provide a boost compared to the standard model.

We consider that the text description of the problems will allow the model to rely on specific terms from the programming field to better link them to the code that the student submitted. Analyzing the texts of the problems, we found one intricacy in the data: some problems do not contain a direct description of

the algorithm that the student should write. On the contrary, the problem text is written in such a way that it is difficult to understand what algorithm is hidden under the "legend of the problem". This feature imposes a restriction on the use of such problems, as the model will rely not on terms but on the problem headings, trying to memorize the frequency of solutions. To prevent overfitting on such problems, we will remove words from the problems that are least related to the programming field.

By removing words that are less related to the programming field, we can reduce the noise in the problem texts and improve the model's ability to extract relevant information. However, it is important to ensure that we do not remove any important information that may be necessary for solving the problem. According this problem, we can use the ChatGPT explaining technique to transform problem texts into condensed materials containing only the necessary information about the algorithm used. With this technique, we leverage OpenAI's open API to reuse our dataset and generate new descriptions for our problems. In the API prompt, we provide the problem description and add text that asks ChatGPT to explain the given problem in the language of algorithms, explicitly highlighting key programming-related keywords that will help the main model orient itself to the current Knowledge State of the student.

6.4 Knowledge Tracing Model

KT models allow memorizing the current state of the student's knowledge at each moment in memory. There are a number of works in which recurrent KT models were used to extract information about the current stage of student learning [8], [25], [26]. Such models were represented by an RNN, such as LSTM or GRU. These models remember information about past student attempts well, but their structure does not allow describing the true nature of human brain learning.

Recurrent models describe the nature of learning as follows: given x_i - the knowledge state at the i-th stage of learning. For generalization, we will consider

that x_t is the knowledge state of the student after solving t problems. The GRU model assumes that:

$$g(h_t) = P(x_t|x_1, x_2 \dots x_{t-1})$$

We conducted a series of researches in our educational school, communicated with methodologists and teachers, and found out that not all problems play the same role in students' acquisition of skills in the educational process. Each subsequent problem carries a number of skills that the student must be able to apply for successful solution. However, this does not mean that skills for each subsequent problem are developed in all previous ones. Student learning can be represented as a tree: initially, he starts with leaves - small atomic topics that subsequently give new knowledge when synthesized.

On the figure below 6.3, a student is learning five topics: 1 - "input and output", 2 - "integers: sum and difference operations", 3 - "while loop: input in loop", 4 - "integers: division, multiplication operations", 5 - "Euclidean algorithm: finding the GCD". Note that studying topics 1 and 2 does not depend on each other, as well as topics 3 and 4, however, by studying them together, a new topic - 5 based on topics 3 and 4. Thus, when solving problem x_3 (topic 3), the student's result does not depend on their previously solved problem x_2 (topic 2), as the student simply does not apply previously learned knowledge. Therefore, the learning model differs from the one presented below for a recurrent neural network.

$$G(x_t) = P(x_t|x_{t-1}\alpha_{t-1}, x_{t-2}\ \alpha_{t-2}), \alpha_i \in [0, 1], \forall i$$

In the field of neural networks, there is an architecture called Transformer that precisely describes the mathematical model of the educational process. Thanks to attention layers, the Transformer can pay attention to previous topics with different weights α.

The Transformer Encoder-Decoder model is often used to extract feature de-

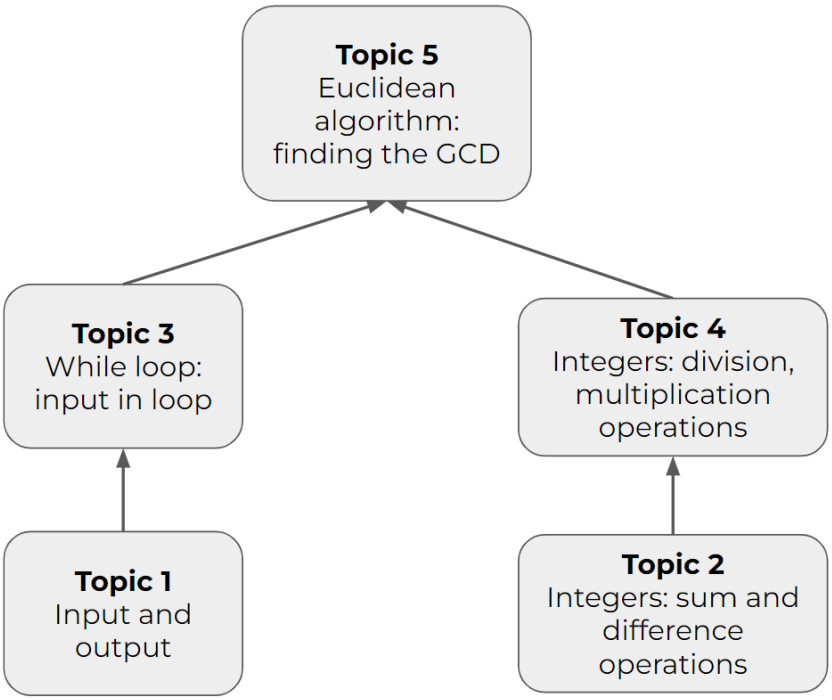

Figure 6.3: Example of topics in Python course

scriptions from sequences. In our research, we use it to combine embeddings of problem texts and student submissions to obtain information about their current knowledge state. Thanks to the attention layer, the model can ignore both previous problems and previous submissions. We create a lower triangular mask for both encoder and decoder layers (unlike models for translating text into another language, where the encoder sees the entire sentence). We did this because information about future solved problems can indirectly indicate where the student has gaps (in problem areas, students will solve more problems to hone their skills).

7 Experiments

7.1 Training

We present experiments, we conducted with our Transformer Knowledge Tracing model (TKT). For comparison with our TKT model, we took the OKT model [8]. The model from the article responds all our requirements and supports input of large sequences due to the long-term memory of the GRU block. We

used GPT2 and GPTBigCode as text encoders for both models and Ast2vec as the code representation model. For each experiment, we used three datasets, each described in the "Data Preprocessing" section.

We divided the sample of students into an 80% training set, 10% validation set for hyperparameter tuning, and 10% test set. Thus, student submissions from different parts did not intersect. We used the Adam optimizer with a learning rate of 0.0002 along with a cosine schedule with warmup for 20 steps. The source code is published on GitHub [1]. We used a maximum submission length of 300. In the architecture of our model, we utilized 3 decoder layers and 1 encoder layer. The output embedding size of Ast2Vec was set to 256. To ensure consistent input embedding sizes, we created an additional trainable projection from the GPT decoder's 768-dimensional embedding to a 256-dimensional embedding. This projection allowed us to make the dimensions of the input embeddings text and code equal.

7.2 Results

For evaluation, we used ROC AUC score. Table 7.1 shows the results of the model training. The results table indicates that the performance of TKT slightly outperforms OKT. However, we observe a significant drop in quality when using the gpt_bigcode text encoder. We believe this behavior is due to the embedding size of the encoder, which is three times larger than ordinal GPT2. Consequently, it would require us to scale up the model architecture. Additionally, we notice that the quality on the last_2 and last_3 datasets is higher that the ordinal dataset. We explain this behavior by the fact that the most crucial information about the student's errors is typically found in their last task's submissions. However, when there are numerous submissions that are very similar, it leads to confusion for the model. In such cases, the model struggles to differentiate between the submissions and may fail to capture the nuances of the student's progress accurately. As a

[1] https://github.com/ivankot88/knowledge_tracing

result, the quality of the predictions and overall performance can be negatively affected.

Overall, our results demonstrate the effectiveness of the proposed Knowledge Tracing model for predicting student performance in programming assignments. By leveraging both text and code representations, our model is able to capture a wide range of student behaviors and provide accurate predictions of their skill levels.

Table 7.1: Results of training

Model Type	Dataset Type	Text Encoder	Test AUC ROC
TKT	ord	gpt2	0.8109
OKT	ord	gpt2	0.7661
TKT	last2	gpt2	0.8243
OKT	last2	gpt2	0.8244
TKT	last3	gpt2	**0.8284**
OKT	last3	gpt2	0.8204
TKT	ord	gpt2_bigcode	0.7312
OKT	ord	gpt2_bigcode	0.7024
TKT	last2	gpt2_bigcode	0.808
OKT	last2	gpt2_bigcode	0.808
TKT	last3	gpt2_bigcode	0.8112
OKT	last3	gpt2_bigcode	0.7971

7.3 Visualization of Knowledge State

7.3.1 Clustering

In order to ensure that our TKT model has learned the necessary information, we decided to visualize the last layer of its embedding representation. This visualization allows us to explore the latent space of each student's knowledge state and perform clustering analysis.

We chose to divide the students into 10 groups. Each student was assigned to a specific group based on their average score per submission: scores ranging from

0.0 to 0.1 placed students in the first bucket, scores from 0.1 to 0.2 placed them in the second bucket, and so on. This division allowed us to categorize students according to their educational outcomes, ranging from weak to strong. For each student, their solution path was passed through the decoder of the KT model, and then all vector representations were projected into a 2D space using the UMAP algorithm. The resulting visualization (fig. 7.1) demonstrates this division. We can observe that the students are divided into several distinct groups, including high-achievers, the main group, and weak students. Additionally, the model has identified several intermediate groups corresponding to different knowledge states.

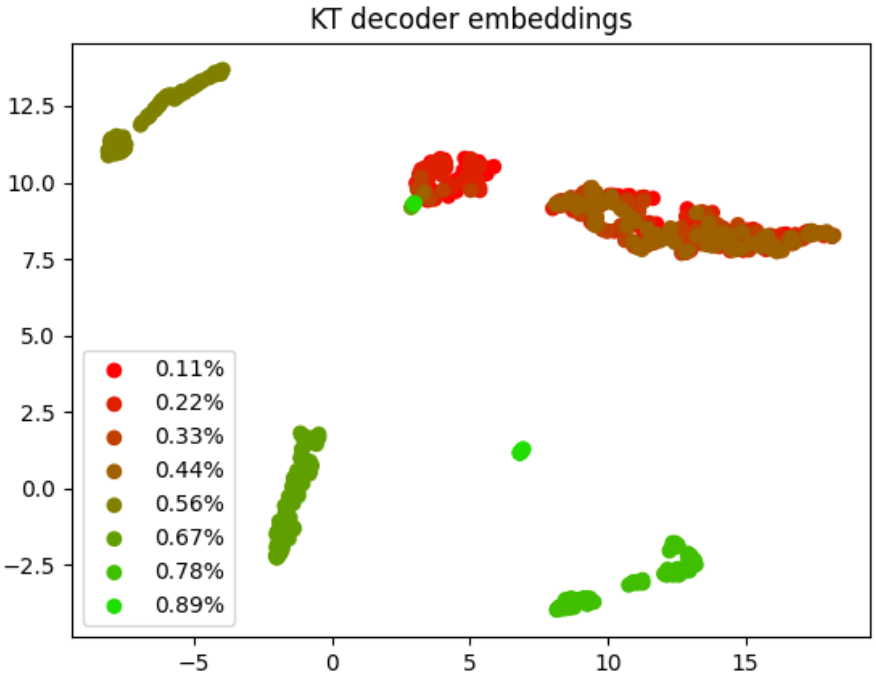

Figure 7.1: Example of topics in Python course

7.3.2 Visualization of Attention Layers

Working with Transformer model provides a significant advantage in data interpretation compared to RNN models. Here, we can explore the attention layers to interpret information about which student submissions (and consequently, the tasks they solved) were more important during their course progress. To accomplish this, we selected a student and visualized the first attention layer of the

code decoder. The heat map of the attention layer is presented in the image 7.2.

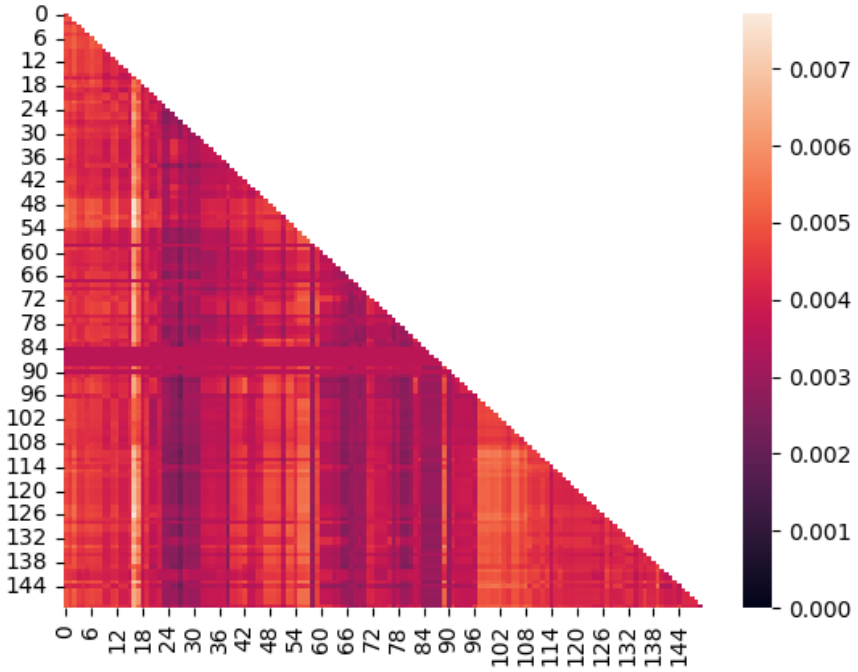

Figure 7.2: Example of topics in Python course

In the upper part of the triangle, there is a white mask overlay, which occurs because the decoder takes input sequences with an infinity upper-triangular mask (to avoid seeing the future). We can observe that the student's 17-th submission was the most important in their solution path, while submissions 23-30 were less significant. Let's examine the problem statement for the 17th submission. The student's submission belonged to ProblemID 441, and they solved it on their first attempt. Here is the text of the problem:

Description:

Write a program that determines the digits of the given four-digit number 'x' and displays them on the screen.

Input:

The input stream contains an integer 'x' ($1000 \leq x \leq 9999$).

Output:

Output the digits of the number 'x' one per line in the following order: thousands, hundreds, tens, and then units."

Submissions 23 to 30 belong to ProblemID 11154, and here is the text of it's problem statement:

Description:

Fedor likes to delay things for later. But every day, his mother asks him to write a report on the progress of an important task she assigned him for the week.

For the first N days, Fedor postponed the task until tomorrow. Then he completed it. In the remaining days, he wrote the phrase "Done long ago" in his daily report. Output Fedor's report entries for the week.

Input:

An integer N: from 1 to 6.

Output:

N lines with the phrase "Will do tomorrow". Then one line "Completed the important task". For the remaining days of the week, display the phrase "Done long ago"

Interpretation of observations.

In this example, we can observe that the first problem involves multiple different tools that a pupil needs to know in order to solve it successfully. These tools include "number input", a "while loop with a condition", "division with remainder", and "division without remainder". In addition, to the tools, the problem hides explicit instructions on which algorithm to apply, making it more

challenging. Solving such a problem on the first attempt indicates that the pupil has a good comprehension of the basic concepts taught in previous lessons. In this example, we can see that the second problem requires fewer tools. It does not involve complex mathematical constructs but focuses on the concepts of "output" and "while loop with a complex condition". The problem explicitly describes the algorithm that needs to be applied. Solving such a problem does not provide much information about the consolidation of knowledge, especially at the end of the course. It is worth considering this problem as an entertaining exercise with elements of material reinforcement.

8 Conclusion

In this article, we explored the use of Transformer Knowledge Tracing (TKT) models in personalized learning. We described how TKT models can be used to predict a student's performance on a given task, based on their previous history.

As a next step, we plan to train our TKT model on student verdicts and analyze its performance in personalizing learning. We identified two specific applications of our TKT model:

1 *Personalized task recommendations.* By inputting a student's history, the model can predict the probability of the student solving the next task. This allows us to select tasks that are of medium difficulty, which are likely to challenge the student, but not overwhelm them.

2 *Clustering students based on their competencies.* By inputting a sequence of tasks completed by a student, the model can generate an embedding of the student's competencies. This embedding can be used to cluster students with similar competencies, allowing us to group students based on their level of difficulty and tailor the curriculum to their needs.

3 *Improvement of instructional materials.* Thanks to attention layer's heatmaps, we will be able to observe which problems have made a greater impact on

the student's development and which ones have had a lesser impact. This allows us to flexibly modify the methodical materials so that each problem contributes significantly to the improving of the student's professional competencies.

References

1. A Survey of Knowledge Tracing Qi Liu, Shuanghong Shen, Zhenya Huang, Enhong Chen, Senior Member, I

2. Albert T. Corbett and John R. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. User Modelling and User-Adapted Interaction, 4(4):253–278, 1995.

3. Pelánek R. Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques //User Modeling and User-Adapted Interaction. – 2017. – T. 27. – C. 313-350.

4. Mohammad Khajah, Robert V Lindsey, and Michael C Mozer. How deep is knowledge tracing? arXiv preprint arXiv:1604.02416, 2016

5. Piech C. et al. Deep knowledge tracing //Advances in neural information processing systems. – 2015. – T. 28.

6. Vaswani A. et al. Attention is all you need //Advances in neural information processing systems. – 2017. – T. 30.

7. Code-DKT: A Code-based Knowledge Tracing Model for Programming Tasks

8. Liu N. et al. Open-ended Knowledge Tracing for Computer Science Education //Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing. – 2022. – C. 3849-3862.

9. Radford A. et al. Language models are unsupervised multitask learners //OpenAI blog. – 2019. – . 1. – №. 8. – C. 9.

10. Choi Y. et al. Towards an appropriate query, key, and value computation for knowledge tracing //Proceedings of the seventh ACM conference on learning@ scale. – 2020. – . 341-344.

11. Shin D. et al. Saint+: Integrating temporal features for ednet correctness prediction //LAK21: 11th International Learning Analytics and Knowledge Conference. – 2021. – С. 490-496.

12. Alon U. et al. code2vec: Learning distributed representations of code //Proceedings of the ACM on Programming Languages. – 2019. – . 3. – №. POPL. – С. 1-29.

13. Zhang J. et al. A novel neural source code representation based on abstract syntax tree //2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). – IEEE, 2019. – С. 783-794.

14. Huo X., Li M., Zhou Z. H. Control flow graph embedding based on multi-instance decomposition for bug localization //Proceedings of the AAAI conference on artificial intelligence. – 2020. – . 34. – №. 04. – С. 4223-4230.

15. Devlin J. et al. Bert: Pre-training of deep bidirectional transformers for language understanding //arXiv preprint arXiv:1810.04805. – 2018.

16. Feng Z. et al. Codebert: A pre-trained model for programming and natural languages //arXiv preprint arXiv:2002.08155. – 2020.

17. Guo D. et al. Graphcodebert: Pre-training code representations with data flow //arXiv preprint arXiv:2009.08366. – 2020.

18. Guo D. et al. Unixcoder: Unified cross-modal pre-training for code representation //arXiv preprint arXiv:2203.03850. – 2022.

19. Li X. et al. CodeRetriever: A Large Scale Contrastive Pre-Training Method for Code Search //Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing. – 2022. – С. 2898-2910.

20. Jain P. et al. Contrastive code representation learning //arXiv preprint arXiv:2007.04973. – 2020.

21. Paassen B. et al. Mapping python programs to vectors using recursive neural encodings //Journal of Educational Data Mining. – 2021. – Т. 13. – №. 3. – С. 1-35.

22. Allal L. B. et al. SantaCoder: don't reach for the stars! //arXiv preprint arXiv:2301.03988. – 2023.

23. Dey R., Salem F. M. Gate-variants of gated recurrent unit (GRU) neural networks //2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS). – IEEE, 2017. – . 1597-1600.

24. Hu Y. et al. Re-factoring based program repair applied to programming assignments //2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). – IEEE, 2019. – . 388-398.

25. Zhang J. et al. Dynamic key-value memory networks for knowledge tracing //Proceedings of the 26th international conference on World Wide Web. – 2017. – . 765-774.

26. Aritra Ghosh, Neil Heffernan, and Andrew S Lan. 2020. Context-aware attentive knowledge tracing. In Proc. ACM SIGKDD, pages 2330–2339

27. Hu Y. et al. Re-factoring based program repair applied to programming assignments //2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). – IEEE, 2019. – . 388-398.