INTEGRATING LARGE LANGUAGE MODEL AND PSYCHOMETRIC ANALYSIS TO UNDERSTAND LEARNER'S CODING BEHAVIORS

M. Zhang[1], C. Li[1], H. Guo[1], M. Li[2], A. J. Ko[2], B. Zhou[2]

[1]Educational Testing Service (UNITED STATES)
[2]University of Washington (UNITED STATES)

Abstract

Assessing programming skills effectively remains a significant challenge in computer science education, as traditional methods often prioritize the final code product while overlooking the processes that lead to a solution. This study investigates how integrating both process-based data captured through keystroke logs and product-based data extracted from final code submissions using a pre-trained DeBERTa model can enhance and facilitate the prediction and understanding of student programming performance. Data from 180 undergraduate students performing Python coding tasks were analyzed. Random Forest models demonstrated that process features, such as review and programming duration and number of code test attempts, predict performance more accurately than product features alone, with the combination of both yielding the highest prediction scores. Hierarchical clustering further revealed four distinct patterns of student coding behavior and proficiency, ranging from “proficient and efficient” to “low proficiency but persistent.” These findings revealed the value of process data in programming assessment and offer practical implications for personalized instruction. Limitations and future work are discussed as well.

Keywords: Coding, behavioral process, LLM, psychometrics.

1 INTRODUCTION

The growing interest in learning to code spans across students and professionals of all ages, driven by both career opportunities and the societal impacts of computing. However, educators often find teaching and assessing computer programming challenging [1, 2]. Despite advancements in identifying learning difficulties and developing instructional methods, assessing programming skills remains relatively underdeveloped [3]. Previous research has largely neglected the conflict between evaluating the final program produced by learners and the process they followed to create it. A key challenge in computer science education and assessment is understanding the connection between a learner's process and the resulting programs. Studies on students' programming processes typically examine sequential snapshots of developing programs. For example, Akram et al. [4] and Miao et al. [5] pinpointed snapshot features that can be used to provide constructive feedback during coding. Although these studies highlight potential applications of programming process data, existing literature does not focus on using this data to directly assess programming skills. This research aims to fill this gap by analyzing process data to investigate how programming task complexity and characteristics influence student programming strategies and processes.

In this research, we utilize keystroke logs to capture students' programming processes. In the domain of natural language writing, researchers have demonstrated the benefits of integrating product-based and process-based approaches to enhance writing assessment [6, 7]. Similar to natural language writing, where combining product-based and process-based methods has proven beneficial for assessment purpose [8, 9], we argue that programming also involves a form of writing. Although programming has its distinct characteristics, it shares commonalities with writing in natural languages. For instance, programmers, like writers, must prioritize their goals due to limited working memory. In the context of assessment, the parallels between writing code and natural language are even more pronounced: both activities respond to tasks (or assessment items) that outline expectations, are evaluated against specific criteria, and, when the task is sufficiently complex, require students to develop and follow a plan for production.

Large language models (LLMs), such as GPT-4, have shown promise in understanding and interpreting computer codes [10]. These models leverage vast amounts of training data, including code repositories, documentations, and natural language descriptions, to learn the syntax and semantics of various programming languages [11]. By doing so, LLMs can assist in code comprehension, debugging, and even code generation. They can analyze code snippets, identify errors, suggest improvements, and provide explanations for complex code segments, to name a few. In this study, as an initial attempt, we analyzed students' codes (product) using a pre-trained DeBERTa model (i.e., deberta-v3-xsmall) [12], and combined the information extracted from process data captured by keystroke logs. Specifically, we address the following two research questions: Research Question 1 (RQ1): How well can information extracted from coding process and/or final code product predict students' programming performance? Research Question 1 (RQ2): What patterns did students exhibit in their coding processes and final code products?

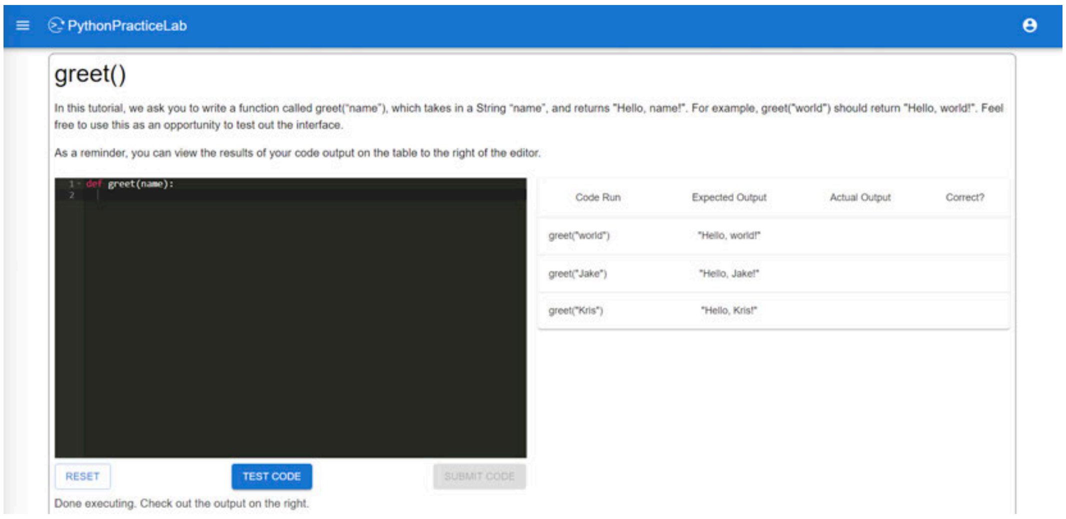
Addressing these research questions has significant practical implications for learners and educators. Understanding how information from the coding process and final code products predicts performance can help educators tailor their teaching methods. By identifying key indicators of success, educators can provide targeted support to students who may struggle, ultimately improving learning outcomes and coding proficiency. Identifying coding patterns can facilitate personalized learning experiences, where instructional strategies can potentially be adapted to fit each student's unique coding style, fostering better engagement and mastery of coding skills.

2 METHODOLOGY

In this section, we describe the data set including the participants, the coding items and the assessment delivery platform, as well as the data analyses applied to address the two research questions.

2.1 Data Set

We used a dataset collected from 180 undergraduate students enrolled in universities in North America. The students were recruited and participated in this study on a voluntary basis and received a moderate monetary compensation. Most participants were either currently enrolled in or had previously taken introductory Python programming courses. Our research team created 21 Python practice coding tasks with varying levels of difficulty and specific task characteristics. Each task included 7 or 8 test cases for students to verify the accuracy, or correctness, of their codes. More detailed descriptions of the recruiting process and participants can be found in Guo et al. [13] and Chen et al. [14]. These tasks were administered through an online learning platform specifically designed for this study, shown in Fig. 1. Students could attempt each task multiple times within a two-week period, and any incomplete work was automatically recorded by the system at the end of this period. We also referred these tasks as coding items hereafter.



greet()

In this tutorial, we ask you to write a function called greet("name"), which takes in a String "name", and returns "Hello, name!". For example, greet("world") should return "Hello, world!". Feel free to use this as an opportunity to test out the interface.

As a reminder, you can view the results of your code output on the table to the right of the editor.

```
def greet(name):
    
```

Code Run	Expected Output	Actual Output	Correct?
greet("world")	"Hello, world!"		
greet("Jake")	"Hello, Jake!"		
greet("Kris")	"Hello, Kris!"		

Done executing. Check out the output on the right.

Figure 1. Coding task interface.

The items were divided into two test forms, enabling a two-stage test design [15]. Students' performance in the first stage determined whether they were routed to a more difficult or easier form in the second stage. Consequently, the items used in the second stage had smaller sample sizes by design. In this study, we analyzed five items from the first test stage, which had larger sample sizes (ranging from 134 to 180) and were of a medium difficulty level as determined by context experts on the research team.

2.2 Data Analysis

To address RQ1, we extracted product features using the last layer embeddings from the DeBERTa model. Specifically, using the pretrained DeBERTa-v3-xsmall model, we processed students' final codes to generate vector representations. This model, with its 12 layers and a hidden size of 384, produces vectors that, hopefully, can encapsulate the nuances of the codes. We focused on the final layer embeddings and performed principal component analysis (PCA), for each item separately, to reduce the complexity of the high-dimensional embedding data from the language model. We analyzed the predictive power for different numbers of principal components (PCs).

In addition, we extracted process features from keystroke logs. Table 1 outlines four key process features derived from keystroke logs, which provide insights into students' coding behaviors.

Table 1. Definitions of Process Feature Variables.

Predictor	***Definition***
Intermediate Review Duration	Sum of all pause durations before checking code accuracy against test cases (in seconds, logged)
Final Review Duration	The pause duration before final code submission (in seconds, logged)
Programming Duration	Total time spent on writing codes (total time duration spent on an item minus the intermediate and final review durations) (in seconds, logged)
Attempt Count	Number of times a student checks the code accuracy against test cases before final code submission

The “Intermediate Review Duration” measures the total time spent pausing before checking code accuracy against test cases. Students may check their codes multiple times over the course of their coding process, and this feature partially reflects a student’s review and debugging process. The “Final Review Duration” feature captures the pause duration before the final submission, showing the time taken for a final review and any last-minute revision. The “Programming Duration” is the total time a student spent writing code, excluding the intermediate and final review durations. This feature highlights the active coding time student spent on a task. The “Attempt Count” feature is simply the count of how many times a student checked their code against test cases, which indicates a student’s iterative testing and debugging efforts. Finally, as an additional product feature, we extracted a “Code Length” variable, calculated as the number of characters in the final submitted code.

For each item, we applied Random Forest models to predict student’s performance using process and/or product features identified above, where student’s performance (prediction target) was determined to be 1 if the final code passed all test cases and 0 otherwise. We implemented a four-fold cross-validation approach and reported average prediction accuracy and F1 score (which represents overall model performance by combining precision and recall) across folds.

For RQ2, we performed hierarchical clustering using Ward’s minimum variance method where variances were defined using squared Euclidean distances between responses [16]. The input variables for clustering included both process and product variables. We inspected the dendrograms resulting from each item as well as between and within cluster variance plots by the number of clusters to decide on a cluster number that make sense across items. It is of note that there is no definitive method or rule that works for all datasets. And determining the optimal number of clusters can be challenging as a result. The decision ought to be guided by the content and context of the data, ensuring that the clusters formed are meaningful and interpretable. All the analyses in this study were conducted using Python.

3 RESULTS

3.1 Predicting Performance Using Process and Product Features (RQ1)

The principal component analysis (PCA) revealed that, consistently across items, the first four PCs explained more than 50% of the variance, the first 10 PCs explained more than 75% of the variance, and the first 20 PCs, over 85%. Fig. 2 shows the cumulative explained variance against the number of principal components.

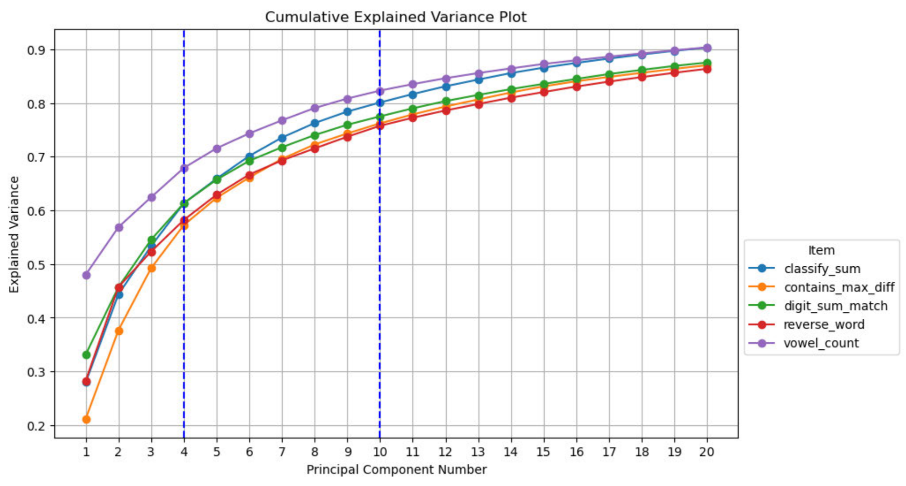


Figure 2. Cumulative explained variance plot.

Fig. 3 and Fig. 4 demonstrate the prediction accuracy and F1 score, respectively, obtained from various predictor sets for each coding item, based on Random Forest models. The prediction target is the dichotomous scores on the final code submission: if the code correctly solves all test cases, the score is 1, otherwise, 0. The results indicate that using only four principal components (PCs) as predictors can achieve comparable prediction accuracy and F1 scores to using 10 or 20 PCs. Adding "Code Length" to the predictor set does not significantly or consistently enhance prediction accuracy and F1 scores. Except for the item "classify_sum," using process features alone (represented by the "Process" bars in lighter blue) resulted in considerably higher prediction accuracy and F1 scores compared to using product features alone. Furthermore, combining process features (as described in Table 1) with product variables as a predictor set led to notably higher prediction accuracy and F1 scores than using product variables (PC and/or Code Length) alone.

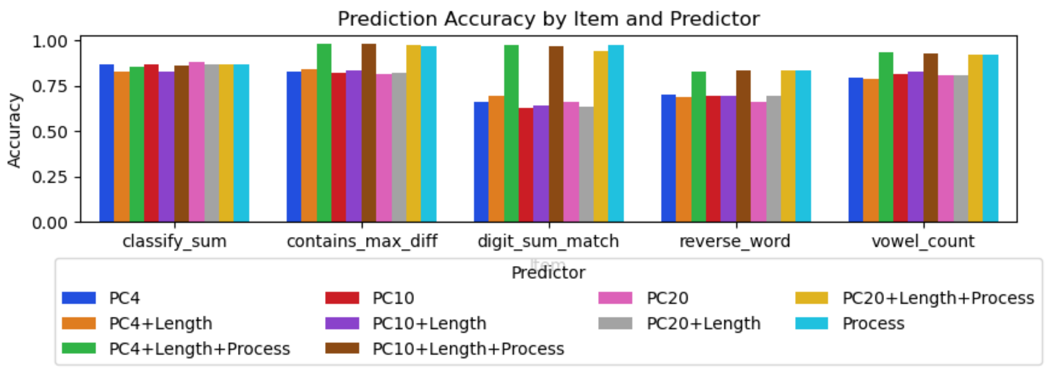


Figure 3. Prediction accuracy by item and predictor set.

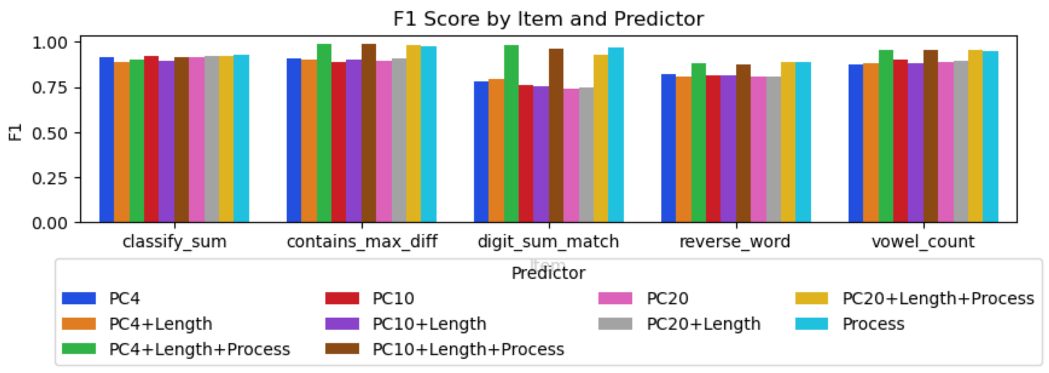


Figure 4. F1 score by item and predictor set.

These results above demonstrated the valuable information that can be gained from students' coding processes, beyond just their final code submissions. The next research question examines the potential strategies students use while coding, which offers deeper insights into their problem-solving skills, debugging techniques, and overall coding habits.

3.2 Clustering of Coding Patterns (RQ2)

Results from RQ1 suggest that process-oriented information are predictive of proficiency. RQ2 investigates whether they can reveal patterns and behaviours that are not evident from the final code alone, which will in turn enable more targeted and effective instructional interventions. Based on the RQ1 results, we moved forward with data analysis using the first 4 PCs. The clustering analysis was conducted separately for each item on the 4 PCs, the process features (listed in Table 1), Code Length, as well as scores. Fig. 5 shows the dendrograms in the hierarchical clustering, where the X-axis represents the students' codes and Y-axis represents the distances between the clusters. The height of each merge (on the Y-axis) in the dendrogram indicates the level of dissimilarity between the clusters being merged, with higher merges representing greater dissimilarity.

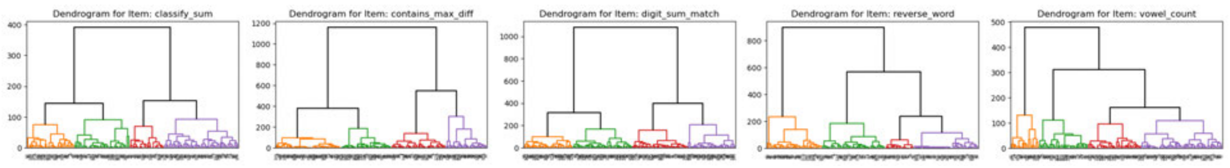


Figure 5. Dendrogram by item.

We identified and selected a four-cluster solution that was consistent and interpretable across items. Table 2 describes the key characteristics for each cluster based on those "explainable" input variables -- the scores, process features and code length. The four clusters represent distinct profiles based on their programming and review patterns. Cluster 1, labelled "Proficient and efficient," is characterized by short intermediate review times, concise code, short programming durations, and high scores, indicating a streamlined and effective approach. Cluster 2, "Proficient and thoughtful," also achieves high scores with short code but takes longer during the final review phase, suggesting a more meticulous, thoughtful and reflective coding process. Cluster 3, "Persistent and adequate proficiency," involves more checking on code accuracy against test cases (more attempts), extended programming times, and longer code, associated with decent scores, highlighting a persistent and thorough effort. Lastly, Cluster 4, "Low proficiency but persistent," features long codes, more attempts, less review time and prolonged programming durations, and associated with lower scores, reflecting a low proficient yet persistent approach.

Table 2. Cluster interpretation.

Cluster	***Profile***	***Characteristic Patterns***
1	Proficient and efficient	Short intermediate review time, short code, short programming time, high score
2	Proficient and thoughtful	Short code, long final review time, high score
3	Persistent and adequate proficiency	More attempts, long programming time, long code, decent score
4	Low proficiency but persistent	Long code, more attempts, less review time, long programming time, low score

For more detailed results, Table 3 shows the means of the clustering input variables used to arrive at these interpretations. It is noted that, while principal components (PCs) were utilized as input variables in the clustering analysis, they were not employed for interpreting the clusters due to their lack of explainability.

Table 3. Means of explainable cluttering variables.

Item	Cluster/ Profile	Cluster Size	Final Score	Intermediate Review Duration (in sec. logged)	Final Review Duration (in sec. Logged)	Programming Duration (in sec. Logged)	Attempt Count	Code Length
classify_ sum	1	30	0.867	-1.421	-4.361	4.103	2.267	113.100
	2	35	0.857	-1.585	-3.909	4.101	2.229	93.657
	3	47	0.851	-1.312	-4.155	4.548	2.660	143.362
	4	22	0.500	-1.847	-5.618	3.845	2.091	160.045
contains_ max_dif	1	35	0.914	-1.893	0.667	4.154	1.543	99.686
	2	44	0.977	-1.949	0.983	4.147	1.364	159.432
	3	37	0.730	-0.937	-0.289	4.792	2.514	227.865
	4	28	0.679	-0.870	-0.288	5.505	2.750	325.071
digit_sum_ match	1	43	0.744	-1.560	0.168	5.057	2.233	125.558
	2	34	0.824	-1.110	0.460	5.381	2.471	306.471
	3	40	0.625	-0.860	-0.833	4.993	2.700	240.450
	4	35	0.486	-1.095	-1.457	5.421	2.914	174.657
reverse_ word	1	39	0.795	-1.571	-0.398	4.009	2.333	162.641
	2	38	0.816	-1.180	-0.133	4.733	2.868	94.737
	3	17	0.765	-1.156	0.033	4.658	2.647	211.412
	4	33	0.636	-0.752	-1.338	5.534	3.121	271.879
vowel_ count	1	32	0.844	-1.803	0.739	4.202	2.156	115.906
	2	34	0.912	-1.287	1.617	4.572	2.647	144.382
	3	53	0.811	-1.527	0.551	4.627	2.453	168.717
	4	18	0.667	-1.035	-0.754	5.339	3.111	229.333

4 CONCLUSIONS

In this study, we addressed the challenges of assessing programming skills among undergraduate students by leveraging both the process (keystroke logs) and product (final code submissions) aspects of coding tasks. Student performance or proficiency, as evaluated based on whether or not successfully passing all test cases, was predicted with high accuracy using Random Forest models: process features alone generally outperformed product features and that the combination of both also yielded high prediction accuracy and F1 scores. Hierarchical clustering of both process and product variables identified four distinct profiles, each characterized by unique coding strategies and proficiency levels: "proficient and efficient," "proficient and thoughtful," "persistent and adequate proficiency," and "low proficiency but persistent."

The findings of this study highlight the useful insights that process data can add to conventional product-based assessment of programming skills. Traditionally, assessment in computer science education has focused almost exclusively on the final code product, potentially overlooking the strategies and behaviors that lead to student success or difficulties. The potential of process-based reporting has been shown valuable in the natural language writing context [17]. Here, our analyses demonstrate that process features, such as time spent reviewing, the number of code check attempts, and programming duration, are at least as informative, and often more predictive, of student proficiency than code product features alone.

The PCA analysis revealed that even a very small number of principal components from code embeddings could capture a large portion of product variance. However, in most cases, process features yielded higher prediction accuracy and F1 scores in predicting student performance. This suggests that students' pathways to a solution, including habits of reviewing and debugging, are essential indicators of their programming proficiency. Practically, this result indicates that integrating both types of data may lead to an assessment framework that is not only more holistic but also more informative for students and educators.

Furthermore, the clustering analysis provides actionable insights into student behaviors. By categorizing students into distinct profiles based on both their processes and final products, instructors may better recognize the individual differences in learning and problem-solving. While this study is an initial attempt to classify coding patterns, the findings show promises to have important pedagogical implications. Educators may use process data to detect struggling students earlier, tailor feedback, and adapt instruction to match distinct coding styles. For example, the "proficient and efficient" group quickly produced successful, concise code, while "proficient and thoughtful" students took more time in their reviews before submission. Persistent students, regardless of proficiency, made more attempts and spent longer coding, suggesting resilience but perhaps a need for additional guidance or feedback to improve effectiveness.

Some limitations should be acknowledged. One key limitation of this work lies in the use of the DeBERTa pre-trained language model for extracting code embeddings. While DeBERTa has shown capability in understanding the syntax and superficial semantics of code, it is not specifically designed for code analysis and may not fully capture deeper structures such as algorithmic efficiency, logical correctness, or problem-solving approaches embedded in the code. Therefore, reliance on DeBERTa embeddings could obscure important facets of programming skill, and future work should explore language models specifically trained for code or incorporate static code analysis tools to enrich product features (such as StarCoder in [18]). Additionally, while clustering offered interpretable profiles, the selection of the optimal number of clusters remains somewhat subjective. Thirdly, only a small number of process features were extracted and used for analysis. Future research is advised to craft more fine-grained process features from keystroke logs such as those used in natural language composition (e.g., phrase and sentence-level editing, keyboarding transcription) [19, 20]. Additionally, the participants were constrained to North American university students familiar with Python, possibly limiting generalizability. Finally, the scope of programming tasks was also relatively narrow, focused on introductory-level problems. Future research is encouraged to explore broader, more diverse datasets, and examine whether these findings generalize to other programming languages and educational contexts.

ACKNOWLEDGEMENTS

This work is supported by the NSF ECR-DUE Awards 2055550 and 2100296.

REFERENCES

[1] M. B. Garcia. “Profiling the Skill Mastery of Introductory Programming Students: A Cognitive Diagnostic Modeling Approach.” *Education and Information Technology*, *30*, 6455-6481. 2025.

[2] B. Xie, D. Loksa, G. L. Nelson, M. J. Davidson… and A. Ko. “A theory of instruction for introductory programming skills.” *Computer Science Education*, *26*, 205-253. 2019.

[3] S. A. Fincher and A. V. Robins. *The Cambridge handbook of computing education research*. Cambridge University Press, 2019.

[4] B. Akram, H. Azizolsoltani, W. Min, E. Wiebe… and J. Lester. “Automated Assessment of Computer Science Competencies from Student Programs with Gaussian Process Regression.” *In Proceedings of the 13th International Conference on Educational Data Mining* (EDM 2020), A. N. Rafferty, J. Whitehill, V.Cavalli-Sforza, and C.Romero (Eds.), pp. 555-560. 2020.

[5] D. Miao, Y. Dong, and X. Lu. “Pipe: Predicting Logical Programming Errors in Programming Exercises.” *In Proceedings of the 13th International Conference on Educational Data Mining (EDM 2020),* A. N. Rafferty, J. Whitehill, V.Cavalli-Sforza, and C.Romero (Eds.), pp. 473-479. 2020.

[6] R. Conijn, C. Cook, M. van Zaanen, and L. Van Waes. “Early Prediction of Writing Quality Using Keystroke Logging.” *International Journal of Artificial Intelligence in Education*, *32*, 835-866. 2022.

[7] M. Zhang, P. Deane, A. Hoang, H. Guo, H. and Li, C. “Applications and modeling of keystroke logs in writing assessments.” *Educational Measurement: Issues and Practice*. Online First. 2025.

[8] M. Zhang and S. Sinharay. “Investigating the Writing Performance of Educationally At-risk Examinees Using Technology.” *International Journal of Testing*, *22*, 312-347. 2022.

[9] R. E. Bennett, M. Zhang, S. Sinharay, H. Guo and P. Deane. “Are There Distinctive Profiles in Essay-Writing Processes?” *Educational Measurement: Issues and Practice*, *41*, 55-69. 2021.

[10] X. Du, M. Liu, K. Wang, H. Wang … Y. Lou. "Evaluating Large Language Models in Class-Level Code Generation." *In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Article 81, 1-13. 2024.

[11] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke… and Y. Zhang. "*Sparks of Artificial General Intelligence: Early Experiments with GPT-4*." Retrieved from https://arxiv.org/abs/2303.12712. 2023.

[12] P. He, J. Gao and W. Chen. "*DeBERTaV3: Improving DeBERTa using ELECTRA-Style Pre-Training with Gradient-Disentangled Embedding Sharing*." Retrieved from https://arxiv.org/pdf/2111.09543. 2023.

[13] H. Guo, M. Zhang, A. Ko, M. Li… and C. Li. "Measuring Students' Programming Skill via Online Practice." *In Proceedings of the 8th Educational Data Mining in Computer Science Education (CSEDM) Workshop*. Atlanta, GA. 2024.

[14] Li, C., Zhang, M., Guo, H., Liu, X., Ko, A., & Li, C. "Leveraging Psychometric Modeling for Enhancing Programming Skill Assessments." *In Proceedings of the 2025 ICAEI Conference*, Suzhou, China. 2025.

[15] M. S. Yigiter and N. Dogan. "Computerized Multistage Testing: Principles, Designs and Practices with R." *Measurement: Interdisciplinary Research and Perspectives*, *21*(4), 254-277. 2023.

[16] Ward, J. H., Jr. "Hierarchical Grouping to Optimize an Objective Function," *Journal of the American Statistical Association*, *58*, 236-244. 1963.

[17] N. Vandermeulen, M. Leijten and L. Van Waes. "Reporting Writing Process Feedback in the Classroom: Using Keystroke Logging Data to Reflect on Writing Processes." *Journal of Writing Research*, *12(1)*, 109-140. 2020.

[18] A. Lozhkov, R. Li, L. B. Allal, F. Cassano… and H. de Vries. "*StarCoder 2 and The Stack v2: The Next Generation*." Retrieved from https://arxiv.org/abs/2402.19173. 2024.

[19] G. Tao, M. Zhang, and C. Li. "Association of Keyboarding Fluency and Writing Performance in Online-Delivered Assessment." *Assessing Writing*, *51*, 100575. 2022.

[20] M. Zhang and P. Deane. "Process Features in Writing: Internal Structure and Incremental Value Over Product Features." *ETS Research Report Series RR-15-27*. Princeton, NJ. 2015.