Journal of Information Systems Engineering and Management
2025, 10(24s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Extracting Error Resolution Patterns for Novice Programming Students using Apriori Algorithm

Niel Francis B. Casillano[1], Rolliedel B. Pajanustan[1]
[1]*Eastern Samar State University*

ARTICLE INFO

Received: 25 Dec 2024

Revised: 13 Feb 2025

Accepted: 27 Feb 2025

ABSTRACT

Introduction: Novice programmers often struggle with error resolution, affecting their learning and performance. This study analyzes error resolution patterns among first-year programming students using the Apriori algorithm.

Objectives: The study aims to identify common programming errors, analyze resolution difficulty and time, and uncover patterns using association rules. It also seeks to provide data-driven recommendations to enhance programming education.

Methods: A dataset of 150 first-year students was analyzed, focusing on error frequency, severity, and resolution time. The Apriori algorithm was applied to identify associations between error type, resolution attempts, and time required.

Results: Syntax errors (319 occurrences) were the most frequent and resolved quickly, while logical (193) and runtime errors (164) were more challenging. Association rules showed that highly difficult errors took over 30 minutes to resolve (80% confidence), whereas low-severity syntax errors were fixed within 30 minutes (75% confidence).

Conclusions: The study revealed the relationship between error type, resolution attempts, and correction time. Findings suggest tiered instructional strategies, such as automated feedback for syntax errors and structured debugging workshops, to improve student proficiency and reduce dropout rates.

Keywords: programming, severity, difficulty, apriori algorithm, association rule

INTRODUCTION

Programming is a core subject in the Information Technology (IT) curriculum in the Philippines [1], serving as a vital skill for various disciplines, including software engineering, data analytics, and artificial intelligence. However, introductory programming courses are often associated with high failure and dropout rates. Recent studies indicate that global passing rates average around 67%, with failure rates exceeding 50% in some institutions [2]. These statistics reveal significant barriers to student success, which can ultimately affect workforce readiness in technology-driven industries. The complexity of learning programming arises from a combination of cognitive, emotional, and technical challenges [3]. In terms of Cognitive load, many students struggle with problem-solving skills and fundamental programming concepts. Research suggests that novice programmers often face difficulties in abstract thinking, logical reasoning, and debugging, all of which are essential for programming proficiency [4][5]. Students frequently experience fear of failure and frustration, particularly in their first year of programming [6]. The need to understand syntax, debugging strategies, and conceptual principles can lead to cognitive overload, further hindering their learning progress.

Among these challenges, debugging is recognized as one of the most difficult aspects of programming [7]. Effective debugging requires not only programming knowledge but also advanced cognitive skills, such as critical thinking [8]. Studies show that while many students acquire basic programming skills, they often lack structured debugging strategies, which are crucial for identifying and resolving errors [9]. Without direct guidance from instructors,

students tend to rely on a trial-and-error approach, which is inefficient and contributes to the high attrition rates in programming courses [10].

Despite being perceived as obstacles, errors are an essential part of the learning process [11]. Research suggests that errors can serve as opportunities for deeper learning, allowing students to refine their understanding of programming concepts [12]. However, for errors to be constructive, students need proper guidance in interpreting and resolving them. Syntax errors, the most common type among beginners, are generally easier to correct when appropriate feedback tools are available. In contrast, logical and runtime errors are less frequent but significantly more challenging due to their conceptual complexity and impact on program execution [13].

While previous research has extensively documented the struggles faced by novice programmers, there has been limited focus on systematically analyzing error patterns and resolution strategies. Existing studies have primarily explored general difficulties in learning to code, but few have examined how specific error types vary in frequency, severity, and difficulty. This gap in the literature underscores the need for research that not only identifies common programming errors but also analyzes how students resolve them. Understanding these patterns can help educators develop more effective teaching strategies.

To address this gap, this study applied the association rule mining technique specifically the Apriori algorithm to analyze programming error data collected from first-year programming students. The research aims to identify patterns in error frequency, difficulty, and severity to provide actionable insights for information technology and computer science educators. By examining how students resolve different types of errors, this study seeks to bridge the gap between error diagnosis and instructional design, ultimately improving programming education outcomes.

Specifically, this study aims to: (1) determine the most frequent types of programming errors made by first-year students, (2) assess the difficulty level of each error type based on student responses and resolutions, (3) evaluate the severity of common errors in terms of their impact on student progress and coding functionality, (4) use the Apriori algorithm to identify patterns in how students resolve errors, highlighting approaches that lead to effective error correction, and (5) propose evidence-based instructional strategies and debugging techniques tailored to the common error patterns identified.

METHODS

Research Design

This study employed the Educational Data Mining (EDM) framework, a specialized branch of data mining used to analyze educational data and derive actionable insights [14]. By integrating techniques from machine learning, statistics, and artificial intelligence, EDM helps enhance learning systems, understand student behaviors, and improve instructional strategies [15]. Figure 1 outlines the structured EDM process, which ensured reliable and meaningful research outcomes. The EDM process begin with data collection, focusing on common programming errors encountered by novice programmers. Data sources included coding logs, assignment feedback, and debugging records, providing a comprehensive view of error types, frequencies, and resolution patterns. Next, data preparation involved cleaning and organizing raw data by addressing inconsistencies, removing duplicates, and handling missing values to ensure dataset reliability.

In the preprocessing stage, data was formatted for compatibility with the Apriori algorithm. Errors were categorized into predefined types: syntax, logical, and runtime, while key variables such as "Time on Task," "Error Frequency," and "Attempts to Resolve" were standardized. This step also involved creating transactional datasets, which were essential for identifying relationships between variables. The data mining phase applied the Apriori algorithm, a widely used association rule mining technique in educational contexts [16]. By analyzing frequent itemsets, the algorithm identified meaningful patterns based on support, confidence, and lift thresholds. Finally, the evaluation phase assessed the instructional value of the generated association rules, prioritizing patterns that highlighted relationships between error severity and resolution attempts. Metrics such as lift and confidence ensured the reliability and relevance of these findings, providing valuable insights for improving programming instruction.

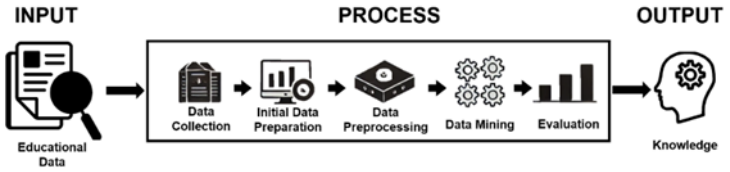

Figure 1. Educational Data Mining Process

Data Collection Methods

The programming language evaluated in this study was C++. Data collection combined manual error logging with the built-in error logging component of Dev-C++. This component systematically recorded error events as students worked on coding tasks, capturing error types and characteristics in real time. The logging component enabled detailed monitoring of the debugging process, documenting interactions within the programming environment. Other details that were not included in the logging component of Dev-C++ was manually written in an error sheet of students along with a predefined set of attributes as shown in Table 1.

Table 1. Data Attributes Collected

Data Gathered	Description
Student ID	A unique identifier assigned to each student
Programming Experience (months)	Indicates the duration of each student's programming experience in months.
Error Type	Categorizes the types of programming errors as syntax, logic, or runtime errors.
Error Frequency	The number of times each type of error occurred for each student,
Error Difficulty	Rates the perceived difficulty level of each error
Error Severity	Measures the impact of each error on the coding task
Time on Task (minutes)	Records the total time each student spent on a coding task
Attempts to Resolve	Counts the number of attempts made by each student to correct each error
Debugging Actions	Describes the specific actions taken by students to debug their code, such as code rewrites or utilizing error messages

Data Analysis

The analysis of programming error data utilized the Apriori algorithm, a common association rule mining method, to identify meaningful patterns and relationships [17]. The process began with data preprocessing, where the raw dataset was cleaned and converted into a binary format suitable for analysis. This involved removing duplicates, addressing missing values, ensuring consistency, and anonymizing student identifiers for privacy. The dataset was further refined through scale compression, abstract description, and consistency processing to ensure a structured format for mining. Following preprocessing, transactions were created for each programming session, aggregating attributes such as error types, debugging actions, and task characteristics. The Apriori algorithm was then applied, using minimum thresholds confidence at 60% to extract patterns that were both frequent and meaningful while filtering out overly common or irrelevant associations.

Association Rule Mining Metrics

Support - This measures how frequently an itemset appears in the dataset.

$$Support = \frac{Frequency\ (X,Y)}{N}$$

Confidence - Measures the likelihood of the consequent occurring, given the antecedent.

$$Confidence = \frac{Frequency\ (X,Y)}{Frequency(X)}$$

Lift - Determines the strength of a rule by comparing its observed support to the expected support if the antecedent and consequent were independent.

$$Lift = \frac{Support}{Support(X) * Support(Y)}$$

Generating and Refining Association Rules

The Apriori algorithm identified frequent itemsets that revealed common co-occurring attributes of programming errors and debugging behaviors, as shown in Figure 2. From these itemsets, association rules were generated and evaluated using support, confidence, and lift metrics. To ensure result quality, an evaluation and pruning process eliminated less significant patterns, retaining only those with actionable insights. The refined rules were then compiled into a structured knowledge base, offering educators data-driven recommendations to enhance debugging instruction.

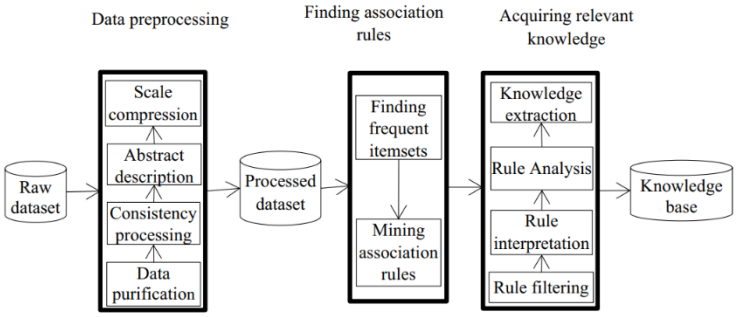

Figure 2. Association Rule Mining Process [18]

Reliability and Validity

To ensure reliability and validity in association rule mining for programming error data, data collection should be standardized so that the error logging consistently captures errors and debugging actions. Testing the Apriori algorithm across various support and confidence thresholds, as well as on different data subsets, can confirm that identified patterns are consistent and not unique to a particular dataset. Clearly defining variables and using metrics like lift and conviction help ensure that the associations found reflect meaningful relationships, rather than random occurrences. Involving programming experts to review the generated rules can enhance the validity by confirming that the patterns align with realistic debugging practices and programming challenges, making the results more applicable for instructional improvement.

Ethical Considerations

When conducting association rule mining on programming error data, several ethical considerations are essential to protect participants and maintain integrity. First, privacy and confidentiality must be safeguarded by anonymizing student identifiers and ensuring that no personally identifiable information is disclosed, especially if findings are

published or shared. Informed consent is crucial, meaning students should be aware that their programming activities are being logged, and they should understand the purpose and potential uses of this data. Additionally, data security measures must be in place to prevent unauthorized access to sensitive information, especially when handling raw error logs and debugging actions. The interpretation and use of results should be approached with care, as findings might influence instructional approaches; thus, any interventions based on the data should aim to support students' learning without unfairly categorizing or penalizing those who encounter more errors.

RESULTS AND DISCUSSION

Frequency of Programming Errors

Figure 3 illustrates the distribution of error types encountered by novice programmers, showing syntax errors (319 occurrences) as the most frequent, followed by logical errors (193 occurrences) and runtime errors (164 occurrences). The predominance of syntax errors aligns with findings by [19], which attribute their frequency to beginners' unfamiliarity with programming language structures and rules. While syntax errors are common, they are often easier to resolve as they typically stem from typographical mistakes or incorrect syntax usage [20]. Logical errors, ranking second in frequency, pose greater cognitive challenges since they require a deeper understanding of program flow and logic. This observation is consistent with Radako [21], who identified logical reasoning as a major hurdle for beginners transitioning from basic syntax to problem-solving. Runtime errors, though least frequent, are particularly difficult as they occur during execution and may result from unexpected user interactions or system configurations. Research by [22] highlights the severity of runtime errors, emphasizing their direct impact on system functionality and the need for robust error-handling mechanisms. The prevalence of syntax errors suggests the need for automated tools or real-time feedback systems to help students quickly identify and correct mistakes [23]. Logical errors, in contrast, require conceptual teaching strategies such as problem-solving workshops and scaffolded debugging exercises [24]. Addressing runtime errors involves training students in debugging techniques and fostering a deeper understanding of program execution, equipping them for more advanced coding challenges [25].

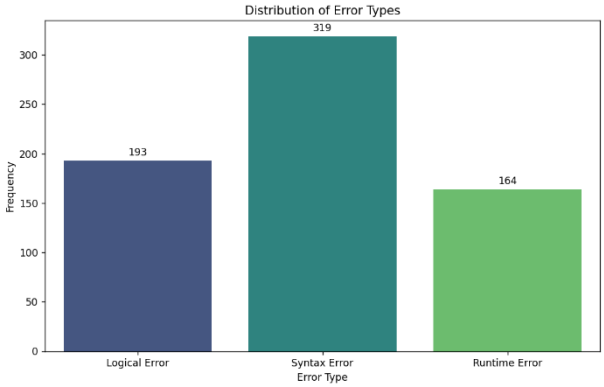

Figure 3. Distribution of Programming Errors

Error Difficulty and Severity

Figure 4 presents the distribution of error difficulty and severity across three programming error types: logical, runtime, and syntax errors. Severity levels range from 1 (least severe) to 5 (most severe), with frequencies represented for each error type. Results indicate that syntax errors occur most frequently across all severity levels, while logical errors display a more balanced distribution, and runtime errors show fewer occurrences but a higher proportion of severe cases. The dominance of syntax errors across severity levels aligns with the findings of [13], who identified syntax errors as the most common among novice programmers. Their high frequency is likely due to students' typographical and syntactical mistakes. However, most syntax errors fall within lower severity levels (1–3), suggesting they are less disruptive and easier to resolve.

Logical errors exhibit a more evenly distributed severity pattern, with higher-severity cases (4 and 5) occurring more frequently than in syntax errors. This supports the research of [26], which highlights that logical errors stem from deeper misunderstandings of program flow and problem-solving, making them more challenging

to correct. Meanwhile, runtime errors, though less frequent, have a significant proportion of high-severity cases. This finding is consistent with [27], who noted that runtime errors often involve complex interactions between code execution and logical structures, posing significant challenges for novices.

The results suggest the need for tiered instructional strategies. For syntax errors, automated tools and integrated development environments (IDEs) with real-time feedback can help students quickly identify and correct mistakes. Logical errors, due to their higher severity and conceptual complexity, require teaching strategies that emphasize program design and logical reasoning, such as pseudocode development and structured problem-solving exercises. Given that runtime errors involve both execution and logic, students should be trained in advanced debugging techniques, including debugging tools and systematic error reproduction. Gradual progression from resolving low-severity errors to tackling more severe issues can help students build confidence and improve problem-solving skills.

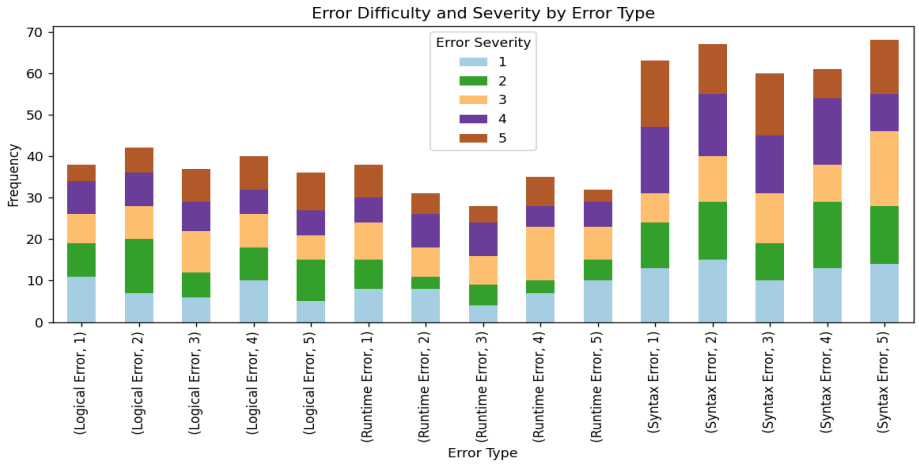

Figure 4. Distribution of Error Difficulty and Severity

Association Rules

The association rules generated by the Apriori algorithm revealed significant patterns in error characteristics and resolution behaviors among novice programmers, as shown in Table 2. Rule 1 indicates that "easy" errors are typically resolved within 30 minutes (75% confidence, lift 1.25), suggesting the efficiency of addressing straightforward issues. This aligns with [28], who found that real-time feedback systems help novices quickly resolve syntax errors, highlighting the value of integrating such tools into learning environments. Rule 2 shows that high-difficulty errors often require over 30 minutes to resolve (80% confidence, lift 1.3), emphasizing the time-intensive nature of complex debugging. This supports findings by [29], which stress the need for advanced debugging strategies. Similarly, Rule 3 links a high number of resolution attempts to difficult errors (70% confidence, lift 1.15), reflecting the trial-and-error approach commonly used by novices when tackling complex issues. Interestingly, Rule 7 suggests that frequent errors are usually of minor severity (60% confidence, lift 1.05), indicating that while they may not significantly hinder progress, addressing them can improve efficiency. Rule 10, which associates "easy" difficulty and "minor" severity with a high likelihood of quick resolution (82% confidence, lift 1.35), reinforces the need for automated feedback tools to handle low-severity errors, freeing up cognitive resources for more complex challenges. Rule 13 links high-difficulty and moderate-severity errors to frequent resolution attempts (74% confidence, lift 1.22), highlighting their resource-intensive nature. These findings underscore the need for targeted interventions, such as step-by-step debugging guidance and conceptual scaffolding, to help students manage complex and severe errors more effectively.

Table 2. Association Rules Generated

Rule No.	Antecedent	Consequent	Support	Confidence	Lift	Interpretation
1	Error Difficulty_Easy	Time on Task (minutes)_<30 mins	0.32	75%	1.25	Errors labeled as easy tend to be resolved within 30 minutes.
2	Error Difficulty_High	Time on Task (minutes)_>30 mins	0.25	80%	1.3	Errors with high difficulty levels often require over 30 minutes to resolve.
3	Attempts to Resolve_High	Error Difficulty_High	0.18	70%	1.15	When high attempts are needed to resolve an issue, it is often a difficult error.
4	Attempts to Resolve_Low	Error Severity_Minor	0.22	68%	1.1	Low attempts to resolve are generally associated with minor severity errors.
5	Error Severity_Moderate	Time on Task (minutes)_>30 mins	0.2	65%	1.12	Errors of moderate severity frequently extend beyond 30 minutes.
Rule No.	**Antecedent**	**Consequent**	**Support**	**Confidence**	**Lift**	**Interpretation**
6	Error Severity_Moderate	Error Difficulty_Medium	0.28	72%	1.18	Moderate severity errors are often of medium difficulty.
7	Error Frequency_High	Error Severity_Minor	0.15	60%	1.05	Highly frequent errors tend to be of minor severity
8	Error Difficulty_Medium	Attempts to Resolve_Medium	0.3	75%	1.2	Medium-difficulty errors generally require a moderate number of attempts to resolve
9	Error Severity_Minor	Time on Task (minutes)_<30 mins	0.34	78%	1.3	Minor severity errors are usually resolved in less than 30 minutes
10	Error Difficulty_Easy, Error Severity_Minor	Time on Task (minutes)_<30 mins	0.27	82%	1.35	Easy errors with minor severity are very likely to take less than 30 minutes
11	Error Severity_High	Error Frequency_Low	0.18	66%	1.2	Errors of high severity tend to occur less frequently
12	Time on Task (minutes)_>30 mins	Error Severity_Moderate	0.21	67%	1.15	When tasks take over 30 minutes, the errors involved are often of moderate severity
13	Error Difficulty_High, Error Severity_Moderate	Attempts to Resolve_High	0.19	74%	1.22	High-difficulty errors of moderate severity often require a high number of attempts to resolve.

3.4. Proposed Debugging and Error Resolution Workshop

Activity Design:

Title: Adaptive Debugging Strategies for Novice Programmers
Theme: Tailored Debugging Approaches for Diverse Learners
Target Participants: First-Year Programming Students

Duration: 2.5 hours
Mode: Blended Learning (face-to-face with digital tools)

Activity Objectives:

1. Classify and analyze common programming errors (syntax, logical, runtime).
2. Apply tiered debugging strategies at varying difficulty levels.
3. Utilize debugging tools (IDEs, debuggers) based on personal skill levels.
4. Foster structured problem-solving skills and adaptive debugging approaches.

Differentiated and Tiered Instruction Strategies:

1. Pre-Assessment: Quick diagnostic quiz to determine student proficiency levels.
2. Tiered Debugging Tasks: Activities are structured at three levels (Beginner, Intermediate, Advanced).
3. Flexible Grouping: Students will work in homogeneous skill-based groups for collaborative learning and heterogeneous mixed-skill groups for peer mentoring.
4. Scaffolded Support: Guided practice for beginners, minimal assistance for advanced students.
5. Adaptive Feedback: Individualized debugging hints and suggestions based on student responses.

Materials Needed:

1. Digital Tools: IDEs (e.g., Visual Studio Code, PyCharm), Debugging Tools
2. Printed Materials: Debugging Playbook with step-by-step debugging techniques
3. Sample Code: Programs embedded with syntax, logical, and runtime errors
4. Laptops/Desktops: For hands-on activities
5. Projector/Whiteboard: For demonstrations

Activity Flow:

Time	Activity	Description	Resources
10 mins	**Pre-Assessment: Debugging Readiness Quiz**	Quick quiz to classify students into skill levels (Beginner, Intermediate, Advanced).	Online/printed quiz
15 mins	**Introduction to Debugging Strategies**	Brief lecture on structured debugging (print statements, debugging tools, tracing errors).	PowerPoint, Sample Code
20 mins	**Tiered Debugging Exercises (Guided Practice)**	Students are grouped based on skill level: - *Beginner*: Step-by-step debugging with instructor guidance. - *Intermediate*: Debugging with guided hints. - *Advanced*: Debugging with minimal instructor intervention.	Pre-made Error Code Samples, IDE
25 mins	**Mixed-Skill Group Debugging Challenge**	Students are regrouped into mixed-skill teams to solve debugging tasks together. Peer mentoring encouraged.	Debugging Playbook, Error Code Samples

20 mins	**Adaptive Individual Debugging Task**	Students work individually at their own pace, receiving tailored hints based on their performance.	IDE, Debugging Tools
20 mins	**Reflection and Personalized Debugging Plan**	Students complete a worksheet analyzing their debugging approach, challenges, and areas for improvement.	Debugging Playbook Worksheets
20 mins	**Wrap-Up, Insights Sharing, and Takeaways**	Group discussion on debugging strategies, sharing of key takeaways. Debugging Playbook is distributed as a resource.	Debugging Playbook

Assessment and Deliverables

1. **Group Challenge:** Number of errors resolved and peer collaboration effectiveness.
2. **Individual Task:** Debugged code submissions with personalized debugging strategies.
3. **Reflection Worksheet:** Self-evaluation of debugging approaches.
4. **Deliverable:**
    - Completed **Debugging Playbook** with personal debugging strategies.
    - Debugged code files with structured solutions.

Expected Outcome:

1. Improved debugging skills tailored to individual competency levels.
2. Increased confidence in error identification and resolution.
3. Development of structured, adaptive debugging approaches for real-world programming tasks.

CONCLUSION

This study successfully analyzed novice programmers' error resolution patterns using the Apriori algorithm, highlighting key challenges faced by first-year programming students. Findings revealed that syntax errors are the most frequent but easier to resolve, while logical and runtime errors, though less common, are more complex and time-consuming. The association rules uncovered meaningful relationships, such as the connection between error difficulty and resolution time, as well as the frequency of attempts made for error correction. The study underscores the need for targeted instructional strategies to enhance programming education. Automated feedback systems can efficiently address frequent syntax errors, while structured problem-solving exercises and debugging workshops are essential for managing logical and runtime errors. By implementing tiered instructional approaches, educators can provide timely support, reducing cognitive overload and dropout rates among novice programmers

REFRENCES

[1] CMO No. 25, Series of 2015. Revised Policies, Standards and Guidelines for Bachelor of Science in Computer Science (BSCS), Bachelor of Science in Information Systems (BSIS), and Bachelor of Science in Information Technology (BSIT) Programs.

[2] Margulieux, L. E., Morrison, B. B., & Decker, A. (2020). Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. International Journal of STEM Education, 7, 1-16.

[3] Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2018). A systematic literature review on teaching and learning introductory programming in higher education. IEEE Transactions on Education, 62(2), 77-90.

[4] Ma, N., Qian, J., Gong, K., & Lu, Y. (2023). Promoting programming education of novice programmers in elementary schools: A contrasting cases approach for learning programming. Education and Information Technologies, 28(7), 9211-9234.

[5] Sun, C., Yang, S., & Becker, B. (2024). Debugging in computational thinking: A meta-analysis on the effects of interventions on debugging skills. Journal of Educational Computing Research, 62(4), 1087-1121.

[6] Atiq, Z., & Loui, M. C. (2022). A qualitative study of emotions experienced by first-year engineering students during programming tasks. ACM Transactions on Computing Education (TOCE), 22(3), 1-26.

[7] Gale, L. (2023, September 18). Debugging: A powerful and dangerous skill to learn. Retrieved from Raspberry Pi Computing Education Research Centre: https://computingeducationresearch.org/debugging-a-powerful-and-dangerous-skill-to-learn/?utm_source=chatgpt.com.

[8] DeLiema, D., Dahn, M., Flood, V. J., Asuncion, A., Abrahamson, D., Enyedy, N., & Steen, F. (2019). Debugging as a context for fostering reflection on critical thinking and emotion. Deeper Learning, Dialogic Learning, and Critical Thinking: Research-based Strategies for the Classroom. Hrsg. von Emmanuel Manalo. New York: Routledge, 209-228.

[9] Michaeli, T., & Romeike, R. (2019, April). Current status and perspectives of debugging in the k12 classroom: A qualitative study. In 2019 IEEE Global Engineering Education Conference (EDUCON) (pp. 1030-1038). IEEE.

[10] Villamor, M. M. (2020). A review on process-oriented approaches for analyzing novice solutions to programming problems. Research and Practice in Technology Enhanced Learning, 15(1), 8.

[11] Sha, A. S., Nunes, B. P., & Shen, J. (2024). Investigating the Role of Errors in Programming Learning. In Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 2 (pp. 797-797).

[12] MacNeil, S., Denny, P., Tran, A., Leinonen, J., Bernstein, S., Hellas, A., ... & Kim, J. (2024, January). Decoding logic errors: a comparative study on bug detection by students and large language models. In Proceedings of the 26th Australasian Computing Education Conference (pp. 11-18).

[13] Shirafuji, A., Watanobe, Y., Ito, T., Morishita, M., Nakamura, Y., Oda, Y., & Suzuki, J. (2023). Exploring the robustness of large language models for solving programming problems. arXiv preprint arXiv:2306.14583.

[14] Shalini, Robinson Joel, B Muthazhagan. (2020). The role of Learning Analytics in Educational Data Mining. International Journal of Computing Algorithm, 9(2), 1-4. DOI: 10.20894/IJCOA.101.009.002.004, ISSN: 2278-2397.

[15] Aulakh, K., Roul, R. K., & Kaushal, M. (2023). E-learning enhancement through educational data mining with Covid-19 outbreak period in backdrop: A review. International Journal of Educational Development, 101, 102814.

[16] Casillano, N.F.B., & Cantilang, K.W. (2024). Employing educational data mining techniques to predict programming students at-risk of dropping out. Indonesian Journal of Electrical Engineering and Computer Science, 35(2), 1219-1226. doi:http://doi.org/10.11591/ijeecs.v35.i2.pp1219-1226

[17] Situmorang, B. H., Isra, A., Paragya, D., & Adhieputra, D. A. A. (2024). Apriori Algorithm Application for Consumer Purchase Patterns Analysis. Komputasi: Jurnal Ilmiah Ilmu Komputer dan Matematika, 21(1), 15-20.

[18] Wang, T., Xiao, B., & Ma, W. (2022). Student behavior data analysis based on association rule mining. International Journal of Computational Intelligence Systems, 15(1), 32.

[19] Kazemitabaar, M., Chyhir, V., Weintrop, D., & Grossman, T. (2023). Scaffolding Progress: How Structured Editors Shape Novice Errors When Transitioning from Blocks to Text. arXiv preprint arXiv:2302.05708. https://arxiv.org/abs/2302.05708.

[20] Dhawan, A. (2024, November 26). The Top 10 Most Common Programming Errors and How to Avoid Them. Retrieved from SchoolMyKids: https://www.schoolmykids.com/education/the-top-10-most-common-programming-errors-and-how-to-avoid-them.

[21] Radaković, D., & Steingartner, W. (2024). Common Errors in High School Novice Programming. IPSI Transactions on Internet Research, 20(1), 47-59.

[22] Osbourn, T. (2023, September 27). The 7 Most Common Types of Errors in Programming and How to Avoid Them. Retrieved from TextExpander: https://textexpander.com/blog/most-common-programming-errors.

[23] Albrecht, E., & Grabowski, J. (2020, February). Sometimes it's just sloppiness-studying students' programming errors and misconceptions. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (pp. 340-345).

[24] Ettles, A., Luxton-Reilly, A., & Denny, P. (2018, January). Common logic errors made by novice programmers. In Proceedings of the 20th Australasian Computing Education Conference (pp. 83-89).

[25] Newby, N. C., Zhang, C., Chidawaya, J., & Dempsey, M. E. (2023, March). Enhancing Feedback Messages for Debugging Runtime Errors in an Introductory Java Programming Course. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2 (pp. 1232-1232).

[26] McCall, D., & Kölling, M. (2019). A new look at novice programmer errors. ACM Transactions on Computing Education (TOCE), 19(4), 1-30.

[27] Albluwi, I., & Zeng, H. (2021, February). Novice difficulties with analyzing the running time of short pieces of code. In Proceedings of the 23rd Australasian Computing Education Conference (pp. 1-10).

[28] Charles, T., & Gwilliam, C. (2023). The Effect of Automated Error Message Feedback on Undergraduate Physics Students Learning Python: Reducing Anxiety and Building Confidence. Journal for STEM Education Research, 6(2), 326-357.

[29] Dupriez, T., Polito, G., & Ducasse, S. (2017). Analysis and exploration for new generation debuggers. Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies.