Doctorado en Ingeniería

Escuela de Ingeniería de Sistemas y Computación

IMProB-It: Automatic Feedback Model for Iterative Programming Tasks in Introductory Programming Courses.

PhD Thesis

Presented and defended in public the **February 18, 2025**
to obtain the Title of

Doctor in Engineering from the Universidad del Valle

(Specialty in Computer Science)

by

Ginna Viviana Leytón Yela

Advisors : Juan Francisco Díaz Frias, Ph.D., *Universidad del Valle*
Andrés Mauricio Castillo Robles, Ph.D., *Universidad del Valle*

MINCIENCIAS – AVISPA –GUIA

Contents

List of Figures

List of Tables

A la vida que me ha dado tanto, Violeta Parra.

Acknowledgments

First and foremost, I want to express my deepest gratitude to God, the universe, and life itself for presenting me with this incredible challenge. This journey has been transformative, testing my limits and shaping me into the person I am today.

To Professor Juan Francisco Díaz PhD., thank you for unconditionally accepting the role of my mentor. Your unwavering support, patience, and motivation throughout this doctoral journey have been invaluable. I also extend my heartfelt thanks to Professor Andrés Mauricio Castillo PhD., who never doubted my ability to persevere. From the very beginning, he supported and encouraged me to move forward. To both of you, thank you for generously sharing your extensive knowledge and believing in me. I would also like to extend my gratitude to the Universidad del Valle and the Bicentennial Scholarships from MINCIENCIAS, which made it possible for me to pursue my doctoral studies.

Special thanks go to my family, whose unconditional support has been the cornerstone of my success. To my partner, Mauricio Chaves, thank you for always believing in my skills and abilities. Your constant support, patience, and understanding at every step of this process have meant everything to me. To my parents, thank you for your unwavering encouragement. To my father, for supporting me in every possible way and teaching me the value of perseverance in achieving my goals. To my mother, for her constant support through her prayers and uplifting words in every challenging moment. To my brother, thank you for standing by my side through every stage of this journey. To all my relatives, who in one way or another kept me in their thoughts and supported me along this path, my heartfelt thanks.

To my cats, thank you for your quiet companionship during countless long nights. A special mention goes to Chispita, who stayed by my side during those late-night study sessions, providing comfort and warmth.

I am profoundly grateful to my doctoral peers, especially Leidy and Jhon, for always being there to listen, encourage, and share countless moments of coffee and reading that kept me grounded. To my Zumba friends, thank you for showing me how to transform stress into joy. This wonderful hobby became a vital outlet for relaxation, helping me discover talents beyond academia and instilling discipline in exercise. Your friendship and camaraderie helped me grow physically and mentally, building my confidence in every way.

To my friends from Pasto, thank you for your constant encouragement throughout my doctoral studies. To Lilian, for her valuable friendship and support; and to German, for his sense of humor, friendship, and motivation to take up running, which became another rewarding hobby. A special thanks to Hector Mora for sharing his expertise and supporting me with the error classification model. Your contributions made a significant difference, as did the support of Professor Oscar Bedoya, who also generously shared his knowledge.

Finally, I want to thank everyone who has been part of this journey since the moment I embarked on this adventure and moved to Cali. Your kindness, help, and encouragement have left an indelible mark on my life and work.

Thank you all for being part of this incredible journey.

Ginna Leyton
Cali, Noviembre 2024

Abstract

The growing interest in integrating artificial intelligence (AI) in education has spurred the development of tools and platforms that support learning, especially in computer programming. While many of these tools are designed to provide summative assessments and general feedback, only a few are tailored to address programming tasks that involve iteration structures, and an even smaller number incorporate Gries' theory into their design. As a result, few solutions effectively support students' understanding of loops in introductory programming courses. Students often struggle with loop construction, finding it difficult to grasp the scope of a loop, which code segments will repeat, and how many times they will execute.

Automated feedback for loop-based programming can play a key role in improving students' comprehension of these concepts. However, factors such as the type of feedback provided and the intervention level are essential to its effectiveness.

This thesis addresses these difficulties by proposing an automated feedback model to support students' understanding of iterative programming. Grounded in program correctness theory and machine learning (ML), the model evaluates student code, identifies common errors, and delivers targeted feedback to explain mistakes made in loop-based tasks.

This research contributes significantly to the field of automated feedback in programming. We developed a specialized dataset of programming tasks featuring "while" loops, annotated to capture typical student errors, including issues with loop initialization, termination, and state transformation. Using Gries' loop programming theory, we built a detailed taxonomy to categorize and label these errors. Later, we trained ML models on this dataset to classify and predict errors in students' "while" loop tasks. Then, we employed prompt engineering with OpenAI's GPT-4 to generate automated feedback aligned with Gries' theory, tailoring it to the errors detected by the ML classifier. Finally, we integrated these models into INGInious, a learning management system (LMS), through an API named IMProB-It, allowing students to receive specific feedback on programming tasks involving "while" loops.

To evaluate IMProB-It, we conducted a quasi-experimental study with first-semester students in systems engineering and related fields. Divided into experimental and control groups, students in the experimental group received automated feedback on their solutions to "while" loop tasks. The results indicate that students who received specific feedback found it beneficial for understanding loop mechanics, as reflected in survey-measured satisfaction levels. This research demonstrates the potential of ML-driven automated feedback to enhance the learning experience for novice programmers, addressing an essential need in computer science education.

Keywords: Automated feedback, iterative programming, machine learning, loop programming errors, Gries' theory, introductory programming courses, INGInious, IMProB-It API.

Part I

Motivation and State of the Art

Chapter 1

Introduction

Contents

1.1 Motivation.

Currently, we find ourselves in the so-called Digital Age. It is an era in which computer programming gain strength and has permeated different disciplines, being the basis of any technological innovation [1] [2]. Researchers mention in [3] that 4C skills (Creativity and innovation, critical thinking and problem-solving, communication and collaboration) are essential to convert students into knowledge developers instead of consumers, that is, globally competitive students, capable of solving problems and making independent

decisions. However, many different challenges are found in the specific study of computer programming. The dropout and failure rate in the initial stage of programming courses has led some researchers [4] [5] [6] to determine the factors that influence this phenomenon.

According to [7], several problems have been shown to occur in the initial stage of learning computer programming. The author defines four main factors; the first factor is computer programming, which he considers problem-solving and implementation; the second factor includes problem-solving skills such as program design failure, loop selection confusion, and inaccurate mental models; the third factor corresponds to ineffective pedagogy related to teaching materials and strategies; the fourth factor includes prior knowledge, negative perceptions and personal attitudes. We will focus on the difficulties in understanding abstract programming concepts, such as managing control structures, specifically loops. According to Robin [8], programming errors and glitches are more commonly found in conditional statements and loops compared to other programming topics; and [9], he claims that beginning programmers confuse the meaning of a loop, or don't understand how an iteration works.

One of the most powerful influences on learning is feedback. Boud and Molloy [10] define it as the process of obtaining information about the development of a work, producing an evaluation of the qualities of the work itself, and allowing the student to generate a correct solution for it. Similarly, Hattie and Timperley [11] propose a feedback model to improve learning through strategies that reduce the gap between actual performance and achievement of the desired goal. The authors describe four levels at which feedback can be offered, and learning effectiveness varies depending on the level applied. The model distinguishes feedback at the task, process, regulatory, and ego levels, emphasizing the interconnection of effective feedback across all levels. According to [12], a feedback intervention at the task level should include cues that support learning and capture students' attention. In this way, improvements in student performance are likely to occur. Feedback combined with effective classroom instruction can be compelling in improving learning.

In another study, Narciss [13] established categories and subcategories of feedback that can be implemented in computer-based learning environments. The authors, in [14] adapted this feedback model for programming assignments, grouping tools into five feedback types identified by Narciss: knowledge about task constraints (KTC), task-related concepts (KC), task errors (KM), guidance on solving problems or improving a task (KH), and metacognitive knowledge (KMC).

Several studies [15] [16] [17] [18] [19] [20] [21] have used machine learning (ML) to extract source code properties, understand, process, classify and generate feedback within programming learning platforms. Their main contributions focus on identifying patterns in source code to enable predictions and subsequent classification. Other studies [22] [23] [24] [17] [25] have applied clustering to match correct programs with similar incorrect attempts, facilitating suggested changes to help repair a program. However, these approaches tend to emphasize syntactic and structural errors, often neglecting logical errors tied to problem-solving strategies.

They also fall short of addressing the theoretical aspects of iterative programming, which are critical for understanding and mastering programming concepts. For instance, Gries' program correctness theory provides a structured methodology to ensure program correctness, with a particular focus on programs involving iterative constructs. This theory emphasizes key elements within a "while" loop, including the initial state (initial variable), the final state (loop condition), and the state transformation (loop body).

Despite advances in automated feedback for programming courses, two big challenges must be addressed specifically on programming with iterations. First, because, difficulties in teaching and learning loops in programming remain a challenge, mainly due to the need for more understanding of the theory of how to program iterations. Second, because, generating automatic feedback tailored to these needs can be instrumental in guiding students' learning. However, several key issues need to be addressed for this feedback to be helpful, including code specification and ML application in generating feedback in programming with iterations.

Taking into account the above, the object of study of this research is to determine, from the analysis of a solution to a computer programming problem with iterations and the application of the theory of how to program iterations [26], how to evaluate the solutions and what type of feedback to generate [14] that is appropriate and enhances the learning of programming with iterations; and to develop this process automatically using ML techniques.

In this dissertation, we have addressed the following key points: (i) we develop a synthetic dataset including iterative programming tasks, annotated based on the feedback taxonomy and defined error types, to meet the need for a specialized dataset in our study; (ii) we propose a task-level automatic feedback model for programming with iterations, including a Gries-based error classification model for basic loop programming tasks, and feedback generation through the GPT API; (iii) we develop an API for INGInious that

generates automatic feedback for iterative programming tasks in introductory programming courses, using machine learning; and (iv) we evaluate the automatic feedback model through a quasi-experimental study with first-semester systems engineering students.

We present throughout the remainder of this chapter the details of the problems addressed in this dissertation, as well as our derived contributions. Section 1.2 presents the problem statement and addresses key challenges. Section 1.3 describe the objectives of the dissertation on which we base our work. Section 1.5 and Section 1.6 detail our contributions and derived publications, highlighting the relationship between these contributions and the objectives of the dissertation. Finally, Section 1.7 and Section 1.8 explain the organization of the remainder of the dissertation and conclude with a chapter summary.

1.2 Problem Statement and Addressed Challenges.

Problem Statement.

Considering the context introduced in the previous section, we state the main problem addressed in this dissertation as follows:

> *Current research on automatic feedback in programming courses has focused broadly on introductory programming but has yet to fill a significant gap in feedback specific to iterative programming. Students frequently need help understanding the theory of programming with iterations, which affects their learning and performance on these tasks. The complexity of iterative programming and the need for studies on this aspect creates an additional challenge in developing tools to support learning. This problem is exacerbated by the need for automatic and corrective feedback to identify and resolve specific errors in the iterative structure, which are only sometimes addressed by current systems. Therefore, there is a pressing need to explore how loop correction theory can be integrated with machine learning techniques to improve feedback on iterative programming tasks and enrich student learning in this field.*

Addressed Challenges.

As the problem statement is very general, we specify some more detailed challenges derived from the two key challenges at the end of Section 1.1.

C1: Teaching and learning computer programming, particularly loops, present difficulties. Students often struggle to understand program correctness theory and address logical errors that are not detected by the compiler or interpreter in programming tasks with loops.

C2: The field of automatic feedback for iterative programming tasks is underexplored. While there is research on general feedback for programming courses, few studies connect the theory of how to program loops and coding.

C3: Existing repositories of programming exercises, particularly those on competitive programming platforms, rarely specialize in exercises focused on basic iteration concepts. While such repositories offer valuable practice opportunities, they often lack a focus on problems that incorporate program correctness theory as a fundamental aspect of loop-based programming tasks.

C4: Classifying errors in programming tasks with loops based on program correctness theory presents a significant challenge. This requires the development of annotated datasets to train models capable of identifying and predicting common errors in loop programming. Error classification is vital for generating appropriate feedback.

C5: Automatically generating feedback tailored to errors in programming tasks with loops using AI, present a notable challenge. We aim for the generated feedback to identify and explain errors clearly, helping students better understand their mistakes and loop programming.

C6: Ensuring accessibility to programming exercises with loops through an online platform is essential. Such a platform offers practical exercises and allows you to apply program correctness theory makes learning much easier, also serving as a controlled environment for experiments and data collection on programming learning outcomes.

The first challenge (*C1*) corresponds to the difficulties in teaching and learning computer programming, mentioned in [4]; according to the authors there is a lot of confusion regarding the use of loops, since beginner students do not understand the theory of how to program iterations. In iterative programming, solving a problem requires a clear specification that defines the parameters necessary for a loop to function correctly. This involves

describing the problem using a precondition Q and a postcondition R. While students often work on tasks involving iterations, they struggle when faced with issues that are not identified by the compiler. These challenges stem from a lack of understanding of the conceptual foundations, leaving them unable to pinpoint the source of the error or explain why it occurs [27] [28].

For the second challenge (*C2*), it has been observed that there needs to do more research on automatic feedback in programming tasks involving iterations. Only four relevant studies have been identified that address this issue [29], [30], [21], and [31]. Lienardy's Cafe platform [29, 32] offers early feedback using informal loop invariants, focusing on loop property evaluation during execution. Akram et al. [21] and Wu et al. [31] apply rubrics to generate feedback, addressing misconceptions and assessing key programming concepts like loops and conditions. More recent studies, such as Azaiz et al. [33] and Roest et al. [30], leverage large language models to generate feedback, with Roest et al.'s StAP interface delivering step-by-step hints for Python exercises involving arrays, conditionals and loops. Most existing studies focus on general feedback in programming courses, centered on the process and execution of the code rather than on the specific relationship between program correctness theory and coding. Our research focuses on developing feedback specifically designed for iterative programming tasks, grounded in program correctness theory, to strengthen the connection between theoretical understanding and practical application.

Furthermore, although some studies have incorporated specific types of feedback, such as those defined by Narciss [13] and adapted by [14] to programming assignments, most focus on feedback on the localization of errors and problems, such as incorrect output and syntactical errors. Logical, runtime, semantic errors, program performance issues, and stylistic errors are less explored in the reviewed studies.

Advanced research on specific types of feedback in programming course tools remains essential. Constant feedback through assignments, exercises, or assessments plays a crucial role in tracking student progress and identifying areas for improvement.

The third challenge (*C3*) underscores the necessity of a dataset that mirrors the connection between the type of feedback given to students and the errors that prompt such feedback. This dataset is a crucial foundation for our research. To address this, we decided to develop a classification system known as a taxonomy, which organizes the elements into hierarchical categories and defines their relationships.

We reviewed studies such as [34], [35], and [36], which developed taxonomies or cat-

egorizations related to errors in programming tasks. For instance, some studies, such as [34], classify errors based on Bloom's taxonomy of learning objectives. Others, like [35], introduced the SOLO taxonomy, which categorizes errors into four levels, focusing on loops, knowledge of programming building blocks, and compilation mistakes. Additionally, the work in [36] proposes a general taxonomy for classifying syntax, semantic, and logic errors, providing insights into the challenges faced by novice programmers.

However, existing classifications focus primarily on specific errors or problems in basic programming and their relationship to learning objectives without comprehensively addressing the feedback about program correctness theory for programming tasks involving loops.

Identifying and classifying errors based in program correctness theory in basic programs is critical to addressing the (*C4*) challenge and offering innovative, learning-focused and relevant feedback. Artificial intelligence plays a key role in error detection and repair through supervised, semi-supervised, and deep learning techniques, improving feedback when aligned with specific types of errors.

Tools such as ErrorCLR [37], DeepFix [38], and Cafe [29] have been developed to automatically evaluate answers in computer science courses. These tools focus on specific challenges and offer recommendations for improvement. Other studies employed datasets and rubrics to provide detailed feedback on basic concepts and error analysis in programming assignments, but fail to fully integrate programming theory into their feedback mechanisms.

Recent advances in machine learning and large language models (LLMs) have enhanced studies aimed at evaluating the accuracy of error identification in feedback generated for programming tasks, surpassing conventional techniques. However, these approaches tend to focus on providing general feedback about programs, often overlooking the specific conceptual challenges students encounter when working with loops.

This underscores the need for feedback systems that extend beyond mere error detection, integrating program correctness theory to provide feedback that not only identifies mistakes but also enhances students' comprehension of fundamental programming principles.

The challenge (*C5*) explores the need for an automatic feedback system to explain errors in programming tasks with "while" loops based on the classification model from Challenge 4. One potential solution is to use LLMs, artificial intelligence models trained with large volumes of text to process and generate natural language. Using models such as

GPT or Gemini, a prompt can be programmed to generate automatic feedback based on the task statement, the proposed solution, and an error classification taxonomy. These LLMs provide more accurate, detailed and specific explanations of the errors detected, to the extent that the prompt is more precise.

Finally, the (*C6*) challenge addresses the need to evaluate an automatic feedback model in an online environment. This environment should allow students to complete various programming tasks and offer automatic feedback as needed. To do this, we could use learning management tools (LMS) that facilitate the incorporation and evaluation of feedback, ensuring that it is appropriate to the errors found.

In our case, we selected the LMS INGInious, which facilitates the creation of programming tasks, test cases, and summative assessments, providing a foundation for generating feedback through an API. By integrating this API with the platform, it enables error classification in programming tasks and the generation of feedback using LLMs.

1.3 Dissertation Goals and Scope.

This section establishes the scope of our solution to the previously stated problem. For this purpose, we define the general and specific goals pursued in this dissertation. We recall that these goals will, define the scope of our work and focus us on finding appropriate solutions for the specific challenges described in the previous Section. Thus, our general goal can be stated as:

GG: *To develop an automated feedback model for iterative programming tasks in introductory programming courses using program correctness theory and ML.*

Additionally, we foresee five specific goals that will help us accomplish the specific challenges:

G1: *To build a taxonomy of errors in programming tasks involving iterations for introductory programming courses, designed to facilitate the generation of automated feedback.*

G2: *To create a synthetic dataset comprising programming tasks with iterations, including code segments annotated according to the defined error taxonomy.*

G3: *To define an ML-driven automated task-level feedback model for programming tasks involving iterations in introductory programming courses, aligned with the defined error taxonomy.*

G4: *To develop an API for INGInious that generates automated feedback on programming tasks with iterations in introductory programming courses using ML.*

G5: *To evaluate the proposed model through a quasi-experimental study.*

1.4 Possible Ambiguity of Terms.

In this dissertation, we use the term *iterative programming* or *programming with iterations* specifically to refer to programs that use "while" loops to repeat blocks of code. This is distinct from the broader concept of iterative programming, which includes a wider array of control flow techniques and structures. Our study, however, focuses on implementing and managing "while" loops in basic programs and their implications for automated feedback.

1.5 Contribution Overview.

In this dissertation there is a direct relationship between the specific addressed challenges (*C1*, *C2*, *C3*, *C4*, *C5*, *C6*) from Section 1.2 and the specific proposed goals (*G1*, *G2*, *G3*, *G4*, *G5*) from Section 1.3. As a result, our contributions also have a direct relationship with each of the specific goals but always attempting to address their specific challenges. Namely, this dissertation provides a series of five general contributions (*GC1*, *GC2*, *GC3*, *GC4*, *GC5*) that derive into nine specific contributions (*SC1.1*; *SC2.1*, *SC2.2*; *SC3.1*, *SC3.2*, *SC3.3*, *SC3.4*; *SC4.1*, *SC4.2*, *SC4.3*; *SC5.1*, *SC5.2*, *SC5.3*). We describe and organize the contributions of this dissertation in this section, and also outline them in Fig. 1.1. In this figure, we include the dissertation challenges and goals to illustrate their respective relationships.

GC1: To build a taxonomy of errors in programming tasks with iterations:

SC1.1: **We defined a taxonomy of errors in programming tasks involving iterations**, and determined the appropriate types of feedback to provide. This comple-

ments goal *G1*, addressing challenges *C1* and *C2*. Specifically, we classify programming errors according to program correctness theory. The principles of program correctness theory, as outlined by Gries, provides a framework for reasoning about loops and iterations. According to Gries, ensuring the correctness of an iterative process involves verifying three key components: the initial state, the final state, and the transformations that occur during the execution of the loop. Defining this taxonomy allows us to establish the rules for constructing our dataset.

GC2: To create to synthetic dataset that includes programming tasks with iterations:

SC2.1: **We created the dataset by selecting programming tasks with iterations and generating additional solutions.** The dataset was built from 80 cases sourced from both public and private repositories, focusing on basic programming tasks that explicitly use "while" loops. These tasks were selected based on the basic structure outlined by Gries, particularly in relation to functions using "while" loops. Each case includes one or more proposed solutions implemented in Python. This selection aligns with goal *G2* and addresses challenge *C3*.

We then used a script powered by OpenAI's GPT model to generate additional correct and incorrect solutions based on the selected cases (resulting in a dataset of 7000 records). The new solutions were designed to include common errors based on the error taxonomy. This process ensured that the dataset includes diverse solutions and error patterns.

SC2.2: **We annotated the dataset with specific details, focusing on error classification and key code components.** The solutions in the dataset were manually reviewed and annotated according to the error taxonomy. Each solution was broken down into key code components related to the iterative structure, including the initial state, final state, and state transformation within the "while" loop.

For each solution, the errors were identified and classified into three categories based on the program correctness theory: initial state errors, final state errors, and state transformation errors, as well as combinations of these error types. This classification aligns with the taxonomy established

in G1. The annotation process, which was performed manually, allows us to use the dataset effectively in future stages of the error classification model and to generate accurate, taxonomy-driven feedback for students.
Furthermore, to expand the dataset with real examples, we used the INGInious platform to create iterative structure exercises. Students completed these exercises, and their solutions were added to the dataset, further enriching the dataset with authentic student submissions.

GC3: To define an error classification model designed to generate automated task-level feedback in programming exercises with iterations:

SC3.1: **We selected models for data preprocessing to prepare the dataset for classification.** We explored various models for preprocessing the dataset to ensure it was ready for training classification models. To achieve objective G3 and address challenge C4, we first defined the classification of error types specified in the taxonomy and applied them to the dataset. Then, to train the classification model, we implemented a data preprocessing pipeline that includes tokenization and embedding generation for the utterance, solution, and code components to ensure the data is appropriately formatted for the model. We conducted two experiments to identify the best pretrained model that generates the most accurate and comprehensive embeddings for the data.

In the first experiment, we use the BERT pre-trained model for text and CodeBERT for Python code, both with embeddings dimension of 768, achieving acceptable performance in terms of system efficiency. In the second experiment, we employ Xenova's "text-embedding-ada-002" tokenization model and OpenAI's embedding generation model of the same name. Tests show that Ada offers a richer text representation and an acceptable code representation. However, data preprocessing is significantly slower due to its higher embedding dimension of 1536. Finally, we use for preprocessing BERT and CodeBERT, as each model is pre-trained for text and code, respectively.

SC3.2: **We designed experiments with machine learning and deep learning models for error classification in programming tasks with "while" loops.** To meet goal G3 and address challenge C4, we defined an error classification model for programming tasks with "while" loops, evaluating different machine learning

approaches. After tokenizing and generating embeddings for the Python descriptions and programs (*SC3.1*), we split the data into training (80%) and test (20%) sets. Both correct and incorrect solutions were included in the dataset to ensure the model learns to recognize correct solutions as well.

We trained basic machine learning models and neural networks with specific settings. The experiments included models such as multilayer perceptron, random forest classifier, and K-Nearest Neighbors classifier in SkLearn, along with one-dimensional convolutional neural networks (CNN) and fully connected neural networks (FC) in Keras. In addition, we explored two ensemble models: one combining random forest with multilayer perceptron using soft voting and another using logistic regression as the final model. Both ensemble methods show good performance in error classification.

We compared models using accuracy, precision, recall, and F1-score. To analyze performance, we generated loss and F1-score visualizations with Matplotlib, a classification report, and a confusion matrix to assess accuracy in each class.

SC3.3: **We performed experiments to predict errors in new programming tasks with "while" loops.** To complete objective *G3* and address challenge *C4*, we evaluated the performance of the model by testing 70 solutions across 10 exercises from the INGInious platform. INGInious is an LMS designed to assess students' programming skills by providing tasks and automatically grading their solutions. The selected exercises included 7 tasks from the dataset and 3 new ones, with buggy solutions provided by researchers.

After loading the model paths into a script, we fed new data, including statements, solutions, and code parts from the 70 exercises, to generate predictions. We then evaluated the results and identified areas where the models could be further improved. To enhance performance, we experimented with different configurations, including model settings, data balancing, and retraining the models. This process allowed us to select the most effective model for final testing with students, as outlined in objectives *G4* and *G5*.

SC3.4: **We design a prompt using the GPT model to generate automatic feedback on identified errors.** To meet goal *G3* and address challenge *C5*, we create a prompt that takes as input the problem statement, the Python solution, and the error classification provided by the above classification model.

The prompt design process involved carefully tuning the GPT model's instructions to generate accurate and helpful feedback focused exclusively on the errors and program correctness theory. This prompt was designed in Python and hosted on a Flask server to facilitate testing. Throughout this development, we tested different versions of the GPT model, such as GPT-3.5 and GPT-4, along with other models like Gemini, evaluating their capabilities to generate contextualized feedback.

The results showed that GPT-4 provided superior insight and accuracy in generating feedback, particularly in identifying errors. Notably, the feedback generated by GPT-4 aligned with the error predictions made by the classification model, and it included valuable information on whether the errors impacted the problem's invariant. Finally, the prompt's results were stored in a database, allowing for a detailed analysis of the quality of the feedback.

GC4: To develop an API for INGInious that generates automated feedback:

SC4.1: **We designed the functionality of the API for the INGInious LMS platform.** To meet objective *G4* and address challenge *C5*, we designed the ImproB-It API for the INGInious LMS platform, which includes an automatic code evaluator. This evaluator first tests the code with predefined test cases and, after evaluation, ImproB-It generates automatic feedback for basic programming tasks with "while" loops. The API reads the problem statement, extracts the relevant parts of the exercise solution, and performs tokenization and embedding generation. This data serves as input for predicting error types in the model. Subsequently, specific feedback for each solution is generated.

SC4.2: **We built the automatic feedback API for INGInious.** To evaluate error classification for programming assignments with "while" loops, we deployed a Flask server with Gunicorn in production, integrated with the INGInious LMS platform.

The process begins when a student submits a solution through INGInious' automatic code evaluator, which tests the solution against predefined test cases. If the solution passes all tests, the student receives a 100% score. If it passes only some or none, the score is adjusted based on the number of successful tests.

If the solution fails, the student is immediately presented with the option to request feedback. Upon request, the problem statement and the proposed solution are sent to the ImproB-It API for processing. Key code components such as variable initialization, the loop condition, and the body of the "while" loop are extracted, as used during model training.
Using a pre-trained, fully connected neural network model, the system predicts the types of errors in the solution. These predictions, along with the relevant code parts, are processed by a GPT-based prompt to generate tailored feedback on the identified errors. The feedback is sent back to INGInious for the student to view and stored in a database for future analysis, ensuring that API logs are available for later review.
This process aligns with objective *G4* and addresses challenges *C5* and *C6*.

SC4.3: **We conducted functional testing of the external automatic feedback complement.** To achieve goal *G4* and address challenge *C5*, we developed a structured process for functional testing of the automatic feedback API integrated into the INGInious LMS.
The goal of the testing was to ensure that ImproB-It was properly integrated with INGInious to process programming tasks involving "while" loops. These tasks, based on program correctness theory, were selected to evaluate the API's functionality. We established 16 basic tasks: 6 to familiarize students with the LMS and 10 to assess the feedback generated for programming tasks with "while" loops in the experimental group.
The testing scenarios included programming exercises with "while" loops, covering both tasks the error classification model was already trained on and new tasks unfamiliar to the model. This approach ensured an evaluation of the system's ability to generalize and handle novel cases. Functional testing confirmed that the API endpoints were responsive, operating as expected, and providing the required information, thereby validating the overall functionality of the API.

GC5: To evaluate the proposed automated feedback model through a quasi-experimental study:

SC5.1: **We defined the quasi-experimental study and work plan to obtain the results.** To address objective *G5*, we employed a quasi-experimental design

with a nonequivalent control group. This design involved two groups: one experimental group receiving the intervention and a control group that did not. The target population consisted of first-semester students enrolled in the Imperative Programming or Informatics I courses at Universidad del Valle. These courses were selected because they introduce programming fundamentals, including the use of iterative structures such as loops.
The experimental group received feedback through the IMProB-It API integrated into the INGInious LMS, while the control group did not receive this feedback. During the 2024-2 academic period, the study was planned between September 10 and October 1, 2024, focusing on tasks involving iterative programming concepts. This experimento aligned with the syllabus for both courses, ensuring that the intervention was integrated into the natural progression of the curriculum.

SC5.2: **We evaluated the IMProB-It feedback model with a quasi-experimental study.** To complete goal *G5* by addressing challenge *C6*, we assessed the model through a quasi-experimental experiment in which the independent variable was the intervention, feedback provided by the IMProB-It API. By comparing this intervention with the absence of feedback, we aimed to evaluate its impact on student outcomes.
For the experiment, we designed ten basic programming exercises on the IMProB-It platform, each focusing on "while" loops, and equipped them with specific test cases. These test cases, consisting of criteria and conditions, were essential for automatically evaluating the programming solutions submitted by students. In addition to the automatic evaluation, we implemented an option to generate feedback if the solution did not pass the test cases.
For the dependent variables, we measured students' satisfaction with the feedback they received. This was assessed through a survey conducted with both the experimental and control groups.

SC5.3: **We analyzed the results of the quasi-experimental experiment.** We implemented a series of analyses to assess its impact on students' academic performance and perceptions. First, we administered surveys to measure student satisfaction with the automatic feedback provided by the IMProB-It API. The survey incorporated multiple criteria, including clarity, speci-

ficity, level of detail, tone, and the perceived importance of feedback in the course, as outlined in Table 9.2.

The collected responses were analyzed using conventional statistical tests and the Mann-Whitney U test to compare the experimental and control groups. This analysis aimed to determine whether the experimental group reported higher satisfaction and perceived effectiveness of feedback compared to the control group. Additionally, we evaluated other key aspects, such as the clarity of error identification, the usefulness of guidance, and the level of additional information provided. These criteria helped to assess the overall quality and relevance of the feedback.

By addressing these aspects, we were able to meet objective *G5* and address challenge *C6*. The analysis provided critical insights into how automated feedback impacted learning outcomes and student engagement. In summary, the experimental group (EG) showed a positive perception of the feedback, highlighting its utility, accuracy, and impact on their learning experience.

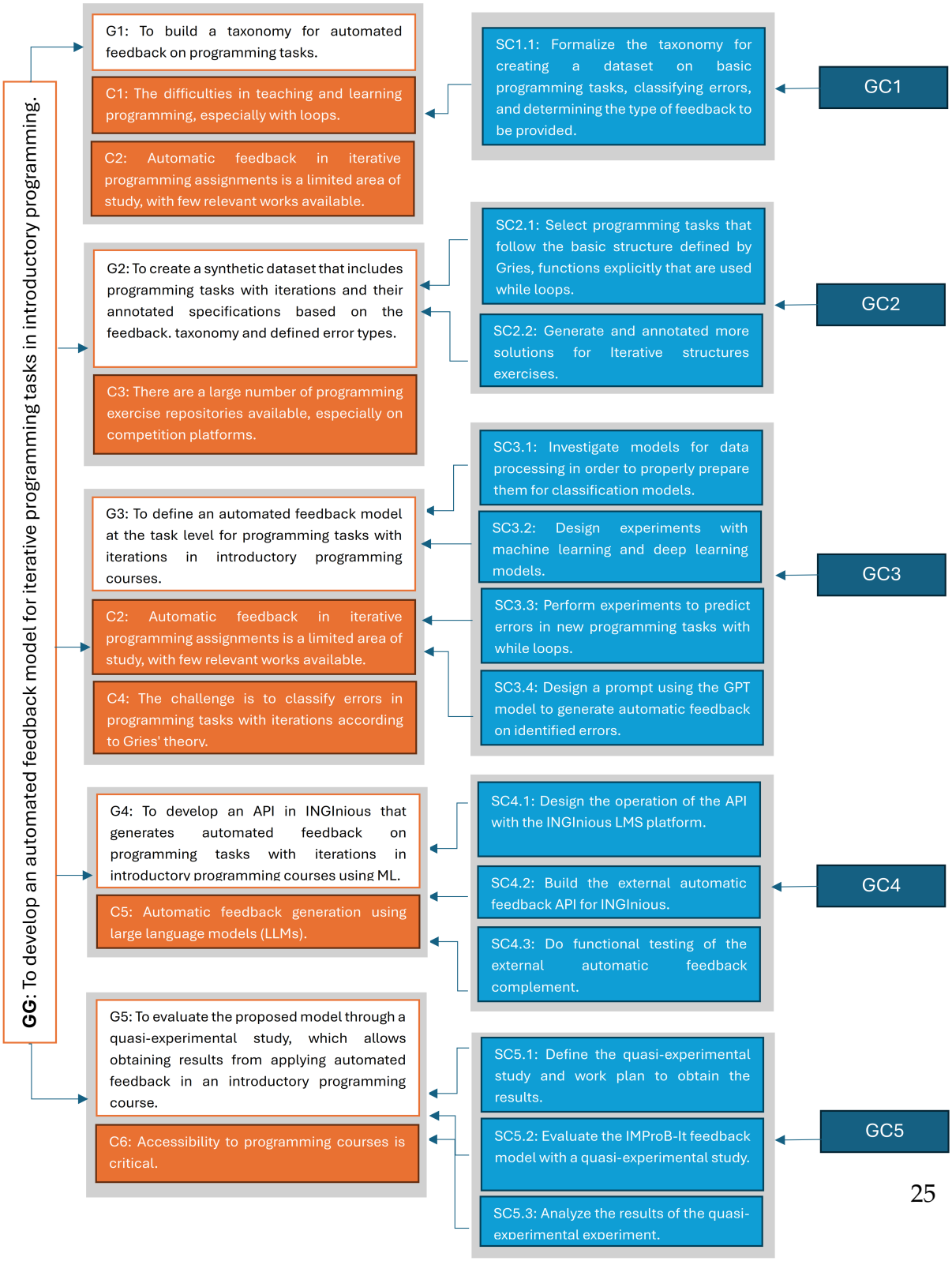

Figure 1.1: Mapping among dissertation challenges, goals and contributions.

1.6 Publications and Resources Derived from this Dissertation.

As part of this research and divulgation process, the results have been published in indexed journals and conference proceedings. Additionally, some of the resources regarding this dissertation are open for public access.

Refereed Journals.

- G. V. Leyton Yela, V. A. Bucheli Guerrero, y H. A. Ordonez Erazo, Systematic literature review: K-12 MOOCs and STEAM., Research and Innovation in Engineering, vol. 9, n. 3, pp. 57 81, dic. 2021. https://doi.org/10.17081/invinno.9.3.5546

- Leyton Yela, G. V., Bucheli Guerrero, V. A., and Ordonez Erazo, H. A. (2022). Tools used for automated formative assessment in computer assisted programming courses. Revista Cientifica, 45(3), 358 - 368. https://doi.org/10.14483/23448350.19662

Conference Proceedings.

- Ginna Viviana Leyton Yela, Victor Bucheli. Smart MOOC model for learning programming in secondary education. SCo2 Masters and Doctorates Symposium - Education and TIC Chapter, 2020.

- Ginna Viviana Leyton Yela, Victor Bucheli. Smart MOOC model for learning programming in secondary education. Semana de la Ingenieria, Universidad del Valle 2020.

- Ginna Viviana Leyton Yela, Juan Francisco Diaz Frias, Andres Mauricio Castillo Robles; IMProB-It: Automatic feedback model for programming tasks with iterations in introductory programming courses. XII Simposio de Investigaciones y Posgrados 2024 "Ingenieria que transforma".

(Related Contribution) & Resource	Description	Rel. Date
(*SC1.1*)Taxonomy document	Taxonomy for automated feedback on programming tasks with iterations.	2022
(*SC2.1* and *SC2.2*) Dataset programming	Dataset of programming tasks with "while" loops, with 7000 records that include the statement, problem, initial state, final state, and state transformation.	2023
(*SC3.1*) Tokenization and Embeddings Program for Programming Loops.	The program is designed to tokenize and generate embeddings using pre-trained models. It uses BERT to tokenize and create embeddings of the statement and CodeBERT to tokenize and generate embeddings of the solution, initial state, final state, and state transformation in Python code.	2023
(*SC3.2* and *SC3.3*) Trained scripts, models for error classification in programming tasks with loops and evaluations.	The deliverable includes the scripts used to train, evaluate, and compare the classification models. In addition, files with the evaluation results, including metrics, visualization charts, and other relevant data, as well as the files with the most competitive results, are provided.	2023
(*SC3.4*) and (*SC4.1*) Automatic feedback API design documentation for INGInious.	This document describes the design of an API for automatic feedback in INGInious, including technical specifications, structure, functionalities, endpoints, input and output formats, and rules for feedback generation.	2024
(*SC4.2* and *SC4.3*) Automatic feedback API for INGInious and evaluations.	This deliverable includes an API for automatic feedback in INGInious and a report with its operation results. It details the API's functionality and performance.	2024
(*SC5.1*, *SC5.2* and *SC5.3*) Quasi-experimental Study Design Document and Results Analysis	This deliverable presents the design of the quasi-experimental study, including the methodology, procedures, and evaluation criteria used. It also provides a detailed analysis of the results obtained, with interpretations and conclusions based on the data collected during the study.	2024

Table 1.1: Dissertation general resources.

Available Resources.

The dataset used in our research was designed to include basic programming tasks that employ iterations, i.e., tasks that use "while" loops. This dataset was built from programming tasks with loops extracted from projects such as CodeNet, INGInious, and other platforms, aiming to obtain various solutions and associated error classifications. Both the researchers and students generated the solutions.

Additionally, large language models (LLMs) such as GPT were used to expand the dataset and generate more examples. The researchers subsequently annotated these LLM-generated solutions, identifying and classifying the types of errors present in each case.

The dataset can be found published on Kaggle (https://www.kaggle.com/datasets/ginnaleytony/dataset-feedbackiterations)

1.7 Dissertation Organization.

The remainder of this dissertation is structured as follows:

In the Part I, Chapter 2 introduces the basic concepts of imperative programming. We explored critical features of structured programming, focusing on iterative programming and common errors related to loops. Chapter 3 shows feedback in programming courses according to existing feedback types and how they apply to programming tasks. We discussed strategies for applying effective feedback in computer programming learning, reviews models proposed in the literature for this purpose, and uses variables to measure the effectiveness of such feedback.

Chapter 4 discusses machine learning and its role in automated feedback. Here, we present the main machine learning algorithms that generate automatic feedback. We describe the algorithms and the metrics used to evaluate their performance, along with the components and tools required for feedback systems. In addition, we give an overview of pre-trained models and specific tools used for error classification in iterative programming tasks.

Part II focuses on taxonomy development and dataset creation. Chapter 5 presents a taxonomy explicitly designed to classify errors in programming tasks with loops. This chapter details the conceptual framework and methodology for defining the properties, relationships, and categories that organize error types and guide the generation of automatic feedback.

Chapter 6 describes the creation of a synthetic dataset focused on iterative programming. We conceived and structured a document of the rules for generating the dataset, its characteristics, and the complete process from its creation to its final implementation.

In Part III, we discuss the development of an automatic feedback model. Chapter 7 defines the scope of this model, describes how we used AI for error classification and subsequent feedback generation with LLMs, and explains the selection of machine learning algorithms, evaluation metrics, and data preprocessing techniques. We also discuss the architectures of the selected models, the training process, hyperparameter configuration,

optimization, and evaluation of the most competitive models. In addition, we explain how feedback is generated using the GPT API based on the error classification model developed for iterative programming tasks.

Chapter 8 focuses on developing an API for the INGInious platform, designed to facilitate automatic feedback generation in programming tasks. This chapter describes the API design, data flow, tools used, and the process of creating and testing endpoints for code extraction, tokenization, embedding generation, and error classification, specifically in programming tasks with "while" loops.

Part IV evaluates the automated feedback model. Chapter 9 presents a quasi-experimental study designed to assess the model's effectiveness. We detailed the study design, target population, study variables, measurement instruments, and plan for executing the test. We discussed data storage, documentation of results, and their interpretation.

Finally, in Part V, Chapter 10 concludes the dissertation by summarizing the main contributions made. It discusses the challenges faced and achievements obtained and offers recommendations for future research, highlighting possible improvements and new directions to follow.

1.8 Chapter Summary.

In this chapter, we have presented the motivation, the problem statement, the challenges addressed, the objectives, and an overview of this research's contributions.

From the motivation perspective (see Section 1.1), automatic feedback in programming tasks with loops offers a promising option to support computer programming learning, providing an efficient tool to assess and guide students in solving computer programming problems.

Regarding the problem statement and the challenges addressed (see Section 1.2), we highlight that the main challenge is the lack of effective mechanisms to provide immediate and valuable feedback in programming exercises with "while" loops according to program correctness theory. This problem is aggravated by the need to correctly evaluate the solutions presented by students and provide clear suggestions for improvement.

For each identified challenge, we define specific objectives (see Section 1.3), such as developing a feedback system that automatically evaluates the code and offers constructive recommendations for common errors in "while" loops.

Finally, we present an overview of our contributions (see Section 1.5), detailing how our solution addresses the challenges and meets the stated objectives. All elements of the dissertation and their relationships (i.e., problem addressed, challenges, goals, and contributions) are described in Fig.1.1.

Chapter 2

Basic Concepts in Computer Programming Fundamentals.

Contents

This chapter presents the fundamental concepts essential to introductory computer programming courses. First, in section 2.1, we introduce structured programming, detailing how control flow structures. Section 2.2 delves into programming with iterations, exploring the use of "while" loops to manage repetitive tasks in alignment with program correctness theory [26]. Following that, section 2.3 highlights typical challenges in iterative programming, such as initial state errors, complex loop conditions, infinite loops, and errors in state transformation that often hinder beginners' progress. Finally, section 2.4 summarizes the chapter, recapping the key ideas and laying the groundwork for the following sections.

2.1 Structured Computer Programming.

Introductory programming, known in the United States as Computer Science 1 (CS1) [6], refers to a course covering basic programming concepts for problem-solving. Failure of this programming fundamentals course affects other courses that must be taken later.

Within this course, the imperative programming paradigm is usually worked on, as show in [39], considered the classic paradigm, which consists on giving orders through a sequence o instructions that specify how to do a task. The source code of imperative languages chains instructions one after another. Control structures such as loops or nested structures are integrated into the source code [28] to manage these instructions.

Imperative programming languages work closely with the system, are concrete, have a simple syntax, and are associated with one of the most used paradigms for the initial teaching of programming [40].

The structured programming style extends the imperative principle for analyzing and designing programs that use concrete control structures: sequences, selection, and iteration [41].

The sequential structure occurs naturally in the language because the statements are executed in the order they appear in the program, one after the other. We find programming concepts within this structure, such as state, data types, data inputs, assignments, expressions, and function calls [28].

One of the relevant concepts is the concept of state [42], considered as a sequence of values in time that contain the intermediate results of a specific computation.

In imperative programming, we find the state as an explicit state. The latter is a state whose lifetime extends beyond a procedure invocation without being present in the procedure arguments [42].

With an explicit state, data abstractions improve considerably in modularity since it is possible to encapsulate it. The operations of the data abstraction limit access to the state.

From the explicit state, the stateful model [42] has been created, which is a modification of the declarative model of computation that adds a mutable store together with the immutable store and the semantic stack, used in imperative sequential programs.

For this document, we will use the term variable assignment. The assignment command has the form:

$$x := e$$

Where x is a variable and e is an expression, the types of x and e are the same. The above command is read as "assign e to x." The execution consists of evaluating e and storing the resulting value in the location called x [26].

As for the difficulties presented in the sequential structure, few are given, and no studies directly compare the effectiveness and efficiency of instruction sequences. There are studies such as [43], which deal with how to avoid sequences that make learning programming difficult, that is, in more complex problems.

The conditional structure relies on executing a statement based on the value assigned to a boolean variable [44]. The conditional statements include if, else, and elif combinations. The statement is as follows:

$$if\langle x\rangle then\langle d\rangle_1 else\langle d\rangle_2 end$$

The behavior of the if statement is subject to whether $\langle x\rangle$ evaluates to true or false. To specify the behavior of a statement, we will use the following notation [26].

Given a precondition P and a postcondition Q, and given a program S, we will say that S is correct concerning P and Q if being in a state that satisfies P, after executing S, we end up in a state that satisfies Q. We will denote it as $\{P\}S\{Q\}$.

Now we can define the behavior of a conditional statement, defining what it means:

$$\{P\}if\langle x\rangle then\langle d\rangle_1 else\langle d\rangle_2 end\{Q\}$$

If x evaluates to true, the statement $\langle d\rangle_1$ executes and should satisfy the following property regarding the preconditions and postconditions P and Q:

$$\{P \wedge \langle x\rangle = true\}\langle d\rangle_1\{Q\}$$

If x evaluates to false, the statement $\langle d\rangle_2$ executes and should satisfy the following property regarding the preconditions and postconditions P and Q:

$$\{P \wedge \langle x\rangle = false\}\langle d\rangle_2\{Q\}$$

The following notation summarizes this rule:

$$\{P \wedge \langle x\rangle = true\}\langle d\rangle_1\{Q\}$$
$$\{P \wedge \langle x\rangle = false\}\langle d\rangle_2\{Q\}$$

$$\frac{}{\{P\}if\langle x\rangle then\langle d\rangle_1 else\langle d\rangle_2 end\{Q\}}$$

The upper section of the horizontal line represents the premises related to the conditional statement, which must hold to reach the conclusion presented below the line.

Research shows that students face fewer challenges with decision structures in learning control structures. However, they still need help to choose the appropriate structures to solve a problem involving decisions [28] [45].

The repetition structure runs one or more statements while a boolean variable remains true. Programming languages use structures such as while and for loops or iterations. Iterative structures are reviewed below [42].

2.2 Programming with "while" iterations.

Programming with iterations, also known as programming with loops, is an essential structure in computer science. This is characterized by the progressive transformation of an initial state S_0 in successive stages until reaching a final state S_{final}. More practically, loops allow a code block to be repeated multiple times until a specific condition is met. This approach is especially useful for automating repetitive tasks and efficiently handling large volumes of data or sequences [26].

An iterative program is then syntactically something like:

$$S := S_0$$

$$\textbf{while not } \text{Final}(S) \textbf{ do } S := T(S) \textbf{ end}$$

Iterative programs are annotated with logical statements that describe their behavior:

- Precondition (Q): Defines the conditions that must be met before the loop starts. It is a requirement to ensure the correct functioning of the loop.
- Invariant (P): It is a logical condition that must remain true during all iterations of the loop. Invariants are essential, guiding the program from precondition Q to postcondition R.

- Postcondition (R): Describes the state of the program after the loop terminates, ensuring that the loop's goal has been reached.

Additionally, we can define a "threshold" (t), which associates each loop state with a natural number that decreases with each iteration, ensuring that the loop eventually terminates.

To understand how an iterative computation works, it is essential to establish a loop specification. This specification is a formal description of the conditions under which the loop runs, the changes made at each iteration, and the expected state at completion.

An iterative program I is correct with respect to a specification $\{Q\}I\{R\}$ if, starting from a state S that satisfies the precondition Q, it ends in a state S_f that satisfies the postcondition R.

However, proving a program's correctness is not trivial. To do so, a logical sentence P called an invariant, helps us verify its correctness.

The following statements can verify the correctness of the iterative program:

1. $\{Q\}\ S := S_0\ \{P\}$: The initial state satisfies the invariant.
2. $\{P\}\ S := T(S)\ \{P\}$: The state transformation maintains the invariant at each iteration.
3. $P \wedge \text{Final}(S) \Rightarrow R$: The postcondition R is guaranteed if the final state is reached.
4. $t(T(S)) < t(S)$ for all Sthat satisfy P: The "threshold" t decreases with each iteration, ensuring the loop's termination.

If the program is incorrect, it is because at least one of the above statements does not hold. From this, we classify the errors as follows:

- Error in the precondition: The initial state does not satisfy the invariant.
- Error in the state transformation: The transformation does not maintain the invariant in some iteration.
- Error in the final state: The final state does not guarantee the postcondition R.
- Termination error: The "threshold" does not decrease correctly, and the loop does not terminate.

From this analysis, the categories of errors that we seek to identify and use in our research emerge. These errors are directly related to the assertions above.

However, in our study, we do not classify a specific error for the "threshold". Instead, the "threshold" is implicitly linked to the final state. This loop structure, together with the derived error categories, allows us to design a conceptual framework for automatic feedback in iterative programming.

Likewise, an iterative computation requires a specification, which defines the description of the parameters necessary for the operation of a loop, that is, the definition of a problem in terms of precondition Q and postcondition R. According to this, an iterative algorithm is correct concerning a specification, if for each input that meets the preconditions Q, the algorithm ends up meeting the postcondition R [42]. The parameters that require a specification are the following:

- Define the input along with its preconditions.
- Specify the output along with its postconditions.
- Describe the iterative approach, detailing how the states evolve from the beginning to the end.
- Specify the structure of each state and present the invariant that holds throughout.
- Establish the initial state by defining its values.
- Define the final state by specifying its values.
- Detail the state transformation, formally describing how each state transitions to the next.

An example that we can take to understand how iterative computation works is factorial calculation. Specifications:

- Input: N $\geq$ 0
- Output: result = N!
- States: states are tuples of the form (index, result) such that result = index! (Invariant).

- Initial state: index = 1, result = 1.
- Final state: index = N.
- State transformation: (index, result) -> (index +1, result * (index+1))
- Idea: an iteration is performed like this, (0,1) -> (1,1) -> (2,2) -> (3,6) -> ... -> (N, N!)

We presented two solutions to the exercise Factorial, the first with the condition "index <= N" (see Fig.2.1); another algorithm could be the one programmed with the output condition "N > 1", as we observed in solution 2 (see Fig.2.2).

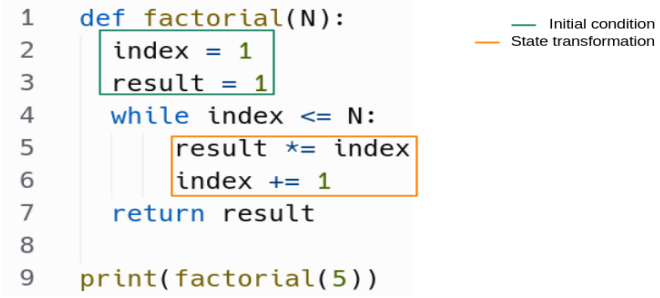

Figure 2.1: Factorial algorithm solution 1.

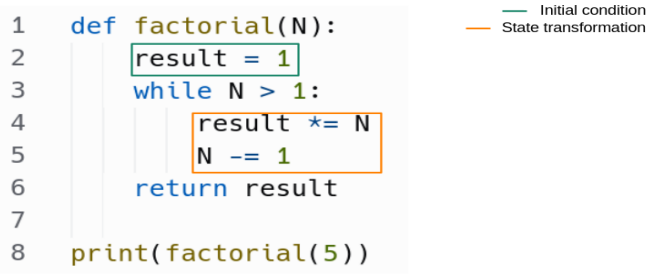

Figure 2.2: Factorial algorithm solution 2.

Similarly, it is possible to create other versions of the same idea that, while slightly different, remain correct along with many others that are similar but incorrect.

2.3 Typical Difficulties in Programming with Iterations.

According to the literature, the mistakes students make when solving programming exercises are primarily associated with the basic mechanics of programming languages; these mistakes are easy to fix. A very different case occurs when the mistakes are due to needing to fully understand how structures such as iterations work, which has been addressed since 1989 by researchers such as Hoc, Kanhey, Kessler & Anderson [46].

The difficulties in programming structures manifest in the study [28]. A survey revealed that 50% of students struggle to grasp the semantics of structures, understand programming concepts, and select the appropriate structures for problem solving.

The iterative solutions can confuse first semester students who often need help understanding the scope of a loop, what lines will repeated, how many times the code within a loop will executed, and so on [27]. For example, suppose there are multiple lines within a loop; one of them is an output statement, such as a print statement. In that case, students may think that the loop only repeats the print statement because they only see the output repeated on the screen.

Additionally, since both loop constructs are present in many programming languages, beginners often mistakenly believe one type of loop is inherently better. For instance, they may assume that "for" loops are superior to "while" loops simply because they are more familiar with them or because of the order in which they were taught, rather than understanding that each loop type has advantages depending on the problem being solved [46]. Another frequent error occurs when loops involve multiple actions. Students sometimes repeat each action individually rather than in the sequence as intended in [47].

Understanding program execution can also be a challenge for beginners. Students may need to understand that program statements are executed sequentially. For example, they may mistakenly believe that a conditional statement inside a loop will executed if the condition is proper, even if it occurs outside the loop [48]. Even understanding a three-line swap is difficult for beginners. Students often struggle to understand the origin of parameter values and the destination of return values when working with functions or methods.

Likewise, studies such as [45] highlight that many misconceptions arise from elements that are not immediately visible but occur within the program's execution, such as loops. The experiment revealed that up to 8% of students made mistakes in exercises involving conditionals in loop control variables, indicating a misunderstanding of these variables'

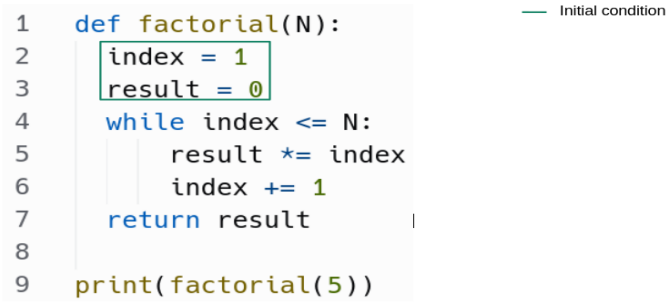

Figure 2.3: Factorial Algorithm solution 3 - Initialization error.

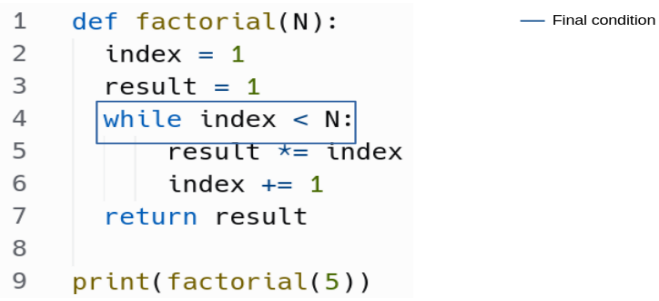

Figure 2.4: Factorial algorithm solution 4 - Final state error.

role in "while" loops.

To understand the difficulties previously addressed, we illustrated below examples of bad implementations of Factorial with typical errors that make a loop not work:

1. Define the initial state incorrectly. In this example (see Fig.2.3), the state variable result starts at 0, which disrupts the state transformation and produces 0 for all factorial calculations.

2. Detect an incorrect final state. The specification defines the final state as $index < N$, but here, the final state aligns with the loop condition, resulting in $N-1$ (see Fig.2.4).

3. Execute an erroneous state transformation. The result remains 0. To obtain the correct outcome, first identify the natural numbers preceding the target number for the factorial calculation, then multiply them all, excluding zero (see Fig.2.5).

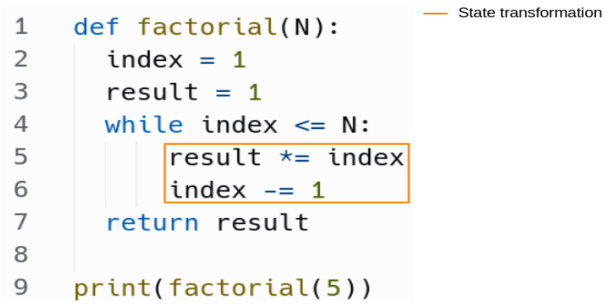

Figure 2.5: Factorial algorithm solution 5 -State transformation error.

2.4 Chapter Summary.

This chapter explores fundamental concepts of computer programming with a focus on the imperative programming paradigm, including structured and iterative programming. It examines the key features of the imperative paradigm, which relies on the sequential execution of instructions and the use of control structures like loops and conditionals. Structured programming is defined by sequences, decisions, and iterations.

The chapter also delves into iterative programming, emphasizing the role of invariants and conditions in ensuring the correctness and efficiency of loops. It addresses common challenges students encounter when learning these control structures and presents techniques to enhance their understanding and improve the design of iterative programs.

Chapter 3

Feedback Essentials.

Contents

In this chapter, we delve into the fundamentals of feedback, exploring its significance and applications, particularly in computer programming education. Section 3.1 defines feedback and its role in learning, which is explained in the following subsections. In subsection 3.1.1, we discuss the advantages of feedback, emphasizing how it promotes reflection, corrects mistakes, and enhances student performance. Subsection 3.1.2 introduces feedback models proposed in the literature, providing an overview of established frameworks that inform effective feedback practices. Following that, subsection 3.1.3 focuses on feedback strategies for programming courses. Subsection 3.1.4 presents variables to measure satisfaction with feedback. Finally, subsection 3.2 summarizes the chapter, synthesizing the key insights and preparing the foundation for subsequent discussions.

3.1 Feedback.

3.1.1 The Advantages of Feedback.

In educational contexts, feedback refers to the information given to students during and after completing a learning task [49]. Feedback is closely tied to two key types of assessment: formative assessment and summative assessment.

Formative assessment is conducted during the learning process and is designed to provide students with actionable insights to improve their understanding and skills. It emphasizes continuous improvement by identifying areas where students need support and enabling iterative refinement of their learning outcomes. This type of assessment is integral to fostering active engagement and aligns closely with self-regulated learning strategies, as described by [50].

Summative assessment, in contrast, is conducted at the end of a learning task or course to evaluate the overall achievement of students. It provides a comprehensive summary of their performance, often in the form of final grades or scores, and serves as a benchmark for their mastery of the subject.

The study [51], describe feedback as an instructional activity aimed at guiding the learning process by integrating foundational knowledge from instructional models with self-regulated learning strategies. Feedback plays a crucial role in enhancing student performance, yet scaling it effectively remains a significant challenge [52]. To address this issue, automated feedback has been introduced. Its key advantages include:

- Automatic feedback helps teachers reduce their workload, complementing the feedback given by the teacher in the classroom. According to [53], these techniques are gaining great popularity.
- The delivery of automatic feedback to students supports the timely development of material and activities in courses [53].
- The delivery of feedback in programming tasks allows monitoring and evaluating students' progress in the course to identify points to strengthen in time [54].

In the study [14], the authors emphasize various motivations behind the development of tools designed to generate feedback and enhance learning in online programming courses:

- Learning programming is a challenge; therefore, students require support to progress, even more so in massive courses.
- Addressing each student's problems individually requires a significant time investment from teachers. Thus, tools that can assist in generating formative feedback during activities provide valuable support.

As can be seen, feedback requires time from the teacher who evaluates the students' activities; it is tough to provide individual feedback to all students enrolled in a course, so alternatives such as automated feedback are sought.

Our research focuses on formative assessment through automated feedback systems.

3.1.2 Proposed Feedback Models in the Literature.

In the literature, there are feedback models proposed by different authors.

For instance, **Hattie and Timperley [11]** introduced a feedback model aimed at minimizing the gap between a learner's current understanding or performance and a desired goal. The effectiveness of the strategies employed by both students and teachers to bridge this gap plays a crucial role in enhancing the learning process.

For feedback to be effective, it must occur within a learning context. Effective teaching involves evaluating students' understanding of the knowledge acquired by addressing three key questions: Where am I going? How am I doing? and Where to next? These questions operate in tandem across the four levels of feedback. Sadler [55] argues that the true power of feedback lies in its ability to close the gap between students' current performance and their learning goals. This feedback model emphasizes that the impact varies depending on the level at which it is applied, with deeper levels resulting in greater effectiveness.

- Feedback on the Task or Product (FT): This type of feedback addresses the correctness of the submitted work. FT is particularly effective in correcting misunderstandings, though less so when dealing with gaps in knowledge. Feedback at this level is most beneficial when it helps students identify and reject incorrect assumptions while offering guidance on how to rectify their mistakes.
- Feedback on the Process of the Task (FP): This feedback focuses on the methods and strategies needed to understand or complete a task. It emphasizes learning processes

such as acquiring, storing, reproducing, and applying knowledge and is essential for fostering cognitive skills that can transfer to more complex tasks.

- Feedback about Self-Regulation (FR): This feedback encourages to continue developing a task. There must be an interaction between commitment, control, and confidence toward the learning goal, including the ability to self-assess, self-management of monitoring, willingness to invest effort in seeking help, and management of feedback information.

- Personal Feedback (FS): This feedback focuses on comments directed to the self, which do not have much information related to the task. FS can impact learning only if it leads to changes in effort or strategies to understand the task.

For feedback to be effective at any level, it must align with well-defined, challenging objectives that direct students' attention to the task at hand.

In other research, **Narciss**, in her study [13], introduces a feedback model that categorizes feedback into various types based on the content of the message, specifically tailored for use in computer-based learning environments. This model addresses multiple aspects, including task rules, errors, conceptual knowledge, procedural knowledge, and metacognitive knowledge. Narciss outlines three main categories of feedback content:

1. Knowledge of Performance for a Set of Tasks (KP): Summative feedback that indicates what has been achieved in the tasks, such as how many tasks were solved correctly.

2. Knowledge of Result/Response (KR): Feedback that communicates whether a solution is correct or incorrect, providing information about the correctness of the current response.

3. Knowledge of the Correct Response (KCR): Descriptions or explanations of the proper solution to the task.

In addition to these core categories, Narciss defines five elaborated feedback concepts that provide more detailed guidance:

- Knowledge about Task Constraints (KTC): Feedback that addresses task rules, constraints, and requirements.

- Knowledge about Concepts (KC): Feedback providing information on the conceptual knowledge necessary for task processing.
- Knowledge about Mistakes (KM): Feedback identifying errors or mistakes made during the task.
- Knowledge on How to Proceed (KH): Feedback offering procedural knowledge on how to continue or correct the task.
- Knowledge on Metacognition (KMC): Feedback fostering awareness of the learner's cognitive processes.

These categories and elaborated concepts of feedback are integral to the design of feedback algorithms. Simple and elaborated feedback are often combined to provide more comprehensive guidance. Narciss's study emphasizes that elaborated feedback, which includes explanations of errors and correct responses, plays a significant role in enhancing learning outcomes. This research guides the development of effective feedback systems in programming tasks, as also highlighted by Keuning et al. [14], who applied elaborated feedback successfully in programming education.

3.1.3 Feedback for Programming Courses.

As mentioned in the introduction, exploring the various types of feedback relevant to programming tasks is crucial. In our case, we focus on introductory programming courses, where studies have adapted the elaborated feedback concepts outlined by [13] for use in automated feedback processes in programming education, as demonstrated by [14]. These concepts are adapted as follows:

Knowledge about task constraints (KTC). This focuses on tasks and divides into two categories:

- Suggestions about task requirements (TR): Outline the specific requirements needed to complete the task successfully.
- Suggestions about task processing rules (TPR): provides general information about the executed exercise.

Studies, such as [14], propose feedback on task requirements (KTC-TR) through tools like INCOM, BASIC BIP, and Fischer06. Other studies, including [14] and [20], explore tools like ADAPT and ACT, which offer suggestions or hints related to task processing rules (KTC-TPR). This feedback is provided at the process level, with the goal of guiding the user in completing or performing a task.

However, we will not focus on this type of feedback in our work, as our primary concern is to address feedback related to error detection and correction within the context of programming tasks, rather than feedback related to task processing or requirement suggestions.

Knowledge about concepts (KC). This distinguishes two categories:

- Exploration on the subject (EXP): As students work on the task, relevant comments about the concepts covered in the resources are provided to support task development.
- Illustrative Examples of Concepts (EXA): Examples that can guide the task.

There are studies such as [20] [21] [56] [57] [58] [59] [60] [61] and [31], which cite tools such as Ask-Elle, Challenge, OverCode, Code.org, among others, which have an additional explanation on the subject (KC-EXP). In the case where suggestions are given with illustrative examples of the concepts (KC-EXA), we fount the studies [14] [18] [62] and [63]. Very few tools contain illustrative examples for automated feedback generation on a task; the examples can serve as a guide for the task. Like the previous feedback classification, this is also delivered at the process level to perform or complete the task.

The feedback we aim to generate is intended to address errors and procedural guidance after the student has finished coding, making real-time conceptual feedback or examples less relevant to our approach.

Knowledge about errors (KM). These types of messages have a type and level of detail that can be a numeric value, location in the code, or a detail identifier such as "compilation error". The subtypes are as follows:

- Testing failures (TF): a failed test indicates that a program does not produce the expected output.

- Compilation errors (CE) are syntactical or semantic errors that a compiler can detect.
- Bug fixing (SE): these can occur in programs that do not have the required behavior; they can be runtime errors, logical errors, or alternative algorithms that are not accepted.
- Style problems (SI): formatting and documentation problems, structural problems.
- Performance problems (PI): issues with execution times or use of more resources than necessary.

The literature presents messages that highlight compilation errors through failure testing (KM-TF) using tools such as OverCode, JPLAS, OeLE, Travis-CI, Cafe, and DeepFix, among others, mentioned in [14] [58] [59] [60] [61] [54] [64] [65] [66] [67] and [19]. Others are compilation errors (KM-CE) that produce feedback through syntactical and semantic errors, such as the feedback presented in CodeHunt, edX, PythonTutor, Annete, Codewebs, DrRepair, RLAssist, among others, cited in [14] [68] [16] [69] [22] [70] [71] [72] [73] [38] [74] [75] [76] [77] [23] [78] y [79].

Error-solving feedback (KM-SE) presents feedback on logical or runtime errors, such as the case of the AutoLEP, AnalyseC, CSTutor tools, among others [14]. This type of feedback should provided at the task level, specifically when the student completes the task. It relies on the compilation process and includes error tests that validate the expected output.

In our research, we worked with this type of feedback (TF, SE, and CE). INGInious was responsible for validating test cases (TF) to confirm the expected output. The tool also detected compilation errors (CE). Since the information provided by the compiler is often limited, ImproBIt complemented this data with additional comments generated by GPT. Additionally, ImproBIt tackled logical errors (SE) by utilizing a classification model to predict specific error types, providing detailed insights into the issues within the code. For each identified error, GPT was used to generate feedback that explained the error's location, suggested potential solutions, and indicated whether it affected the program's invariant. This approach aligned with our objective of delivering feedback after the programming task was completed, ensuring that error identification and resolution were directly linked to the submitted solution. Style issues (SI) and performance problems (PI), while important, were beyond the scope of our research.

Knowledge on how to proceed (KH). Feedback can be suggestions, questions, or examples that help students correct errors or proceed to the next step. The following subtypes:

- Error-related suggestions for correction (EC): this type of feedback identifies errors and focuses on what the student should do to correct them.
- Task processing steps (TPS): contain information about the next step the student should follow to reach a solution.
- Improvements (IM): suggestions for improving the solution.

Error-related feedback for correction (KH-EC) appears in tools like TipsC, MindReader, LISP, Clara, Sarfgen, and Grok Learning. These tools provide specific feedback aimed at helping students correct identified errors, as demonstrated in various studies [14] [80] [18] [67] [70] [23] [24] [81] [82] [83] [84] [17] [85] [86] [87] and [25].

When students require more advanced feedback, the task processing steps subtype (KH-TPS) offers hints, guides, or information to help them achieve successful solutions. Tools such as Ask-Elle, Eclipse Che, Hint Factory, and CodeHunt, among others, provide this type of feedback, as noted in the literature [14] [56] [71] [75] [81] [88] [89] [90] [91] and [92].

Feedback for improvement (KH-IM) offers suggestions that enhance solutions. Tools like Tracer, Code.org, and edX provide this feedback, as referenced in studies by [31], [63], [15], [93], and [94]. This type of feedback can occur at two levels: during the task-solving process or after completion. The system can automatically validate the code as students work in their coding environment and generate feedback upon compiling a task. According to the literature, students can receive hints automatically during compilation or request them, as demonstrated by UNCODE [95].

Although some suggestions may be provided to aid error correction, the primary goal is not to guide students through a complete problem-solving process but to highlight and explain the identified issues in their code.

Knowledge about Metacognition (KMC). Metacognition involves the strategies students employ to solve problems. It reflects their awareness of the task at hand and their understanding of how effectively they have performed it. Only two projects developed with this type of feedback [14], [88]. This type of feedback is within the level of self-regulation defined by [11].

However, in our research, we opted not to incorporate KMC, as it focuses a more complex area that goes beyond error correction and task-level feedback. Addressing metacognition requires delving into cognitive and self-regulatory processes, which involve broader interdisciplinary considerations and extend beyond the primary scope of our study.

3.1.4 Variables for Measuring Feedback Satisfaction.

The literature review identifies several variables used to measure feedback satisfaction in programming courses. We defined four variables: feedback, programming, grading, and teaching status. Each type has a set of variables classified and referenced in Table 3.1.

Most studies [31] [21] [56] [18] [64] [65] [16] [69] [22] [70] [38] [74] [80] [83] [25] [57] [96] [97][98] [99] [100] [101] [67], interviews and surveys were used to assess the satisfaction with feedback. For example, in the [57] study, 71% of students appreciated feedback on programming assignments and were willing to receive feedback by providing positive comments on the feedback. Similarly, in the [58] study, there was greater consistency in submitting assignments per week due to the feedback received. On the other hand, interviews are conducted with students after an assignment to investigate whether the interface is useful and to what extent the feedback works [99].

Researchers increasingly use large language models (LLMs) to enhance programming learning. The study [102] included a questionnaire to collect programming students' opinions on using ChatGPT for learning purposes, investigating both this tool's perceived benefits and limitations.

Likewise, the study by [33] focused on evaluating feedback generated by GPT in the context of introductory programming tasks. This study included a manual evaluation by experts, considering variables such as the feedback length, the accuracy of the correction performed by GPT-3.5, and the quality of the feedback text generated.

There are other types of variables concerning metadata or information obtained from programming course platforms; we will call this set of variables "programming." This variable type allows compilation errors in new code submissions to be compared with previous relevant pre-existing errors submitted by other students. This comparison shows the variables of some attempts and time taken to solve a task [103] [63][104] [67].

The variable rating evaluates by the percentage of students who passed a programming course compared to previous courses or from other institutions that have standard

curricula [105] [106] [107] [108] [100]. In addition, the study [92] takes into account the higher grades obtained by students who used the feedback tool in programming courses.

The study [105] shows the variable of improvement in teaching conditions, in which observed that the amount of teacher participation was reduced from more than 700 hours to only 300 hours, including the deadlines for tutorials and the preparation of programming assignments. The time required for manually correcting programming assignments submitted by studios was reduced to 80 hours.

No.	Variable type	Variables	References
1	Feedback.	Usefulness of high-level suggestions, usefulness of suggestions on the next step to follow, usefulness in accessing elaborated solutions, ease in understanding feedback texts, use of problem solvers, usefulness of problem solvers, feedback length, accuracy of correction performed by GPT 3.5, quality of generated feedback text, type of feedback, level of detail, personalized, appropriate, specific, misleading information, tone and length.	[56] [18] [64] [65] [16] [69] [22] [70] [38] [74] [80] [83] [25] [57] [96] [97][98] [99] [100] [101] [67] [21] [31] [30]
2	Programming.	Execution time, time to decrease in compilation errors, number of attempts, number of lines changed, test cases that pass, time taken by the tool to generate the Feedback, percentage of submissions with automatic fixes.	[103] [63][104] [67]
3	Rating.	Percentage of students who passed the course, higher grades obtained.	[105] [106] [107] [108] [100]
4	Teaching Condition.	Reduction of teaching load in hours.	[105]

Table 3.1: Variables for measuring Feedback.

In our study, we focus on the feedback variable due to its direct relationship with students' satisfaction when receiving feedback on programming tasks. This variable en-

compasses elements such as the type of feedback, level of detail, additional information, specificity, potential misleading information, as well as the tone and length of the feedback all of which contribute to evaluating the quality of the feedback. We also take into account the importance of feedback within the course context, as its perceived value influences the improvement of feedback strategies.

Furthermore, we incorporate the variable programming experience within the programming category, as students' prior programming experience may affect how they perceive and apply the feedback they receive, thereby influencing their learning process.

3.2 Chapter Summary.

The chapter on feedback in programming courses addresses its advantages, theoretical models, and strategies applied in teaching programming. It highlights that feedback is essential to improve learning and regulate student performance, but its large-scale implementation can be challenging, prompting the development of automated tools.

Different feedback models are presented, such as those proposed by Hattie and Narciss, feedback strategies implemented in programming courses, and the variables to measure feedback satisfaction. In the context of programming, strategies such as knowledge about task constraints, concepts, and errors are explored, which allow for detailed feedback tailored to students' needs, facilitating their progress in learning programming.

Chapter 4

Machine Learning and Feedback.

Contents

In this chapter, we explore the use of machine learning algorithms for generating automated feedback in computer programming tasks. Section 4.1 introduces various machine learning methods, including supervised, unsupervised, and deep learning, along with the performance metrics used to evaluate these algorithms. Section 4.2 discusses the machine learning algorithms implemented explicitly for code classification and automated

feedback generation in programming, highlighting their practical applications and evaluation. Section **??** outlines the components and tools required for building automated feedback systems, including data sources and pre-trained models used in programming courses. Finally, section 4.5 summarizes the chapter's key points, emphasizing the role of machine learning in error classification and feedback generation for programming tasks.

4.1 Machine Learning Algorithms.

This subsection presents an overview of machine learning algorithms, categorizing them into supervised, unsupervised, and deep learning approaches.

The following algorithms are specifically designed for classification tasks in machine learning. Detailed explanations are provided to serve as references when discussing their application in generating automatic feedback.

4.1.1 Discriminative AI.

4.1.1.1 Supervised learning.

Supervised learning algorithms rely on a set of labeled training data to learn patterns and relationships. These labeled datasets are used to train the algorithm to correctly assign labels that match the ones previously defined [109].

The algorithm's performance is iteratively refined by evaluating how often it fails to correctly label the test data. Based on these failures, the algorithm adjusts its behavior to improve accuracy in subsequent runs. Once the training phase with labeled data is complete, the algorithm is tested on a new dataset with known labels to evaluate its ability to generalize and make accurate predictions [110].

Supervised learning is often used for solving:

- Classification problems, that is, problems in which the result is a discrete label categorized according to a set of parameters.
- Regression problems, that is, problems where the result is a continuous numerical value within an infinite set of possible results.

Types of Algorithms in Supervised Learning.

Decision trees. A decision tree is an algorithm that learns based on observations and logical constructions. It has a graphical representation of a tree containing a set of nodes, leaves, and branches [111]. The classification process begins with a central node, from which the child nodes emerge according to a particular attribute of the problem; the branches have labels with attribute values, and the leaves correspond to final nodes with conditions for solving the problem.

Support Vector Machines (SVM). SVMs are a set of supervised learning algorithms that classify a data set in a dimension n; Support vectors are points that define a maximum separation margin between the hyperplane that separates the classes. When the way to separate two classes with the hyperplane in a linear way is not found, a kernel function is used, which consists of inventing a new dimension in which a hyperplane can be found to separate the classes [112].

Case-based reasoning (CBR). This type of algorithm emulates the reasoning process that a human being performs to solve a problem. The algorithm solves a problem by using other solutions similar to the one posed; that is, it uses a database of cases of problems solved in the past. To solve a new problem, the system retrieves the most similar instances from the case database and proposes a newly solved case to enhance the database [113].

Naive Bayes Classifier. It is a class of machine learning classification algorithms based on a statistical classification technique called "Bayes' theorem." The algorithm consists of training a data set from which a model is built that allows predicting the nominal value belonging to a class; the predictor variables are independent of each other. This classifier requires a small amount of data [114].

K-Nearest Neighbor. The k-nearest neighbors (KNN) algorithm is a non-parametric supervised learning technique for classification. It classifies data points based on proximity rather than clustering, operating on the assumption that similar points tend to be located near each other [17].

4.1.1.2 Unsupervised learning.

Unsupervised algorithms have an exploratory behavior; they do not have labeled data to train and try to find some organization or pattern in the input data they receive [109].

For example, when facing a clustering problem, these algorithms try to group the data based on similar characteristics.

Types of Algorithms in Unsupervised Learning.

Clustering (k-means). An unsupervised classification algorithm groups objects into k groups according to specific characteristics. The clustering minimized the sum of distances between each object and the centroid of its group; this process is iteratively, and for each of its repetitions, the aim is to reduce the sum of distances between the object and the centroid [115].

The types of problems that this algorithm solves are optimization problems, in which the centroids are initially assigned, the sets are created taking into account the elements closest to the centroid, and the centroids update in each repetition [116].

Principal component analysis (PCA). This technique serves as part of exploratory data analysis, aiming to describe data in terms of new, uncorrelated components [117].

4.1.1.3 Deep Learning.

Deep learning is a subset of machine learning based on neural networks comprising input, output, and hidden layers. This process is inspired by biology, which includes units that behave similarly to neurons in the human brain, connected and capable of transmitting signals between them [109].

Each neural network layer contains units that transform data into useful information to make predictions, learn from experience, generalize rules from previous examples, and abstract critical patterns from the input data [118].

Deep learning algorithms within the supervised learning paradigm are trained on labeled data, enabling them to tackle tasks such as classification and regression. The following models are particularly noteworthy in this context:

Multilayer Perceptron (MLP), is a fully connected artificial neural network where every node in one layer connects to all nodes in the subsequent layer. This structure proves particularly effective for classification and regression tasks.

Recurrent Neural Networks (RNN), these networks are intriguingly designed for sequential tasks, such as time series analysis or natural language processing. Their unique feature of maintaining an internal memory allows them to take into account temporal dependencies in the data.

Long Short-Term Memory (LSTM), this is a type of RNN designed to address the problem of long-term dependencies. This capability makes LSTMs especially valuable for applications like machine translation and speech recognition.

Convolutional Neural Networks (CNN), these networks are impressively versatile, effectively processing data with grid-like structures, such as images. Their versatility has led to widespread use in fields like computer vision, image processing, and object recognition.

Transformers, these architectures have revolutionized natural language processing by using self-attention mechanisms. They allow the modeling of long-distance dependencies without the need for recursion. Transformers are essential in machine translation, text generation, and natural language analysis tasks [119].

4.1.2 Generative AI

Large Language Models (LLMs), belong to the deep learning family and leverage advanced architectures like Transformers. Trained on extensive datasets, these models excel at understanding and generating coherent, contextually relevant text. Notable examples include GPT (Generative Pre-trained Transformer), BERT (Bidirectional Encoder Representations from Transformers), and their various specialized variants used in different natural language processing applications [102].

4.1.3 Performance metrics in classification algorithms.

Below are some metrics used to measure the performance of trained algorithms. In this case, we will explain the metrics that measure the performance of classification algorithms and cite them later. The metrics allow us to determine how good the predictions made by the models are and to improve them since they can lead to erroneous predictions.

Accuracy. Accuracy is a performance metric that allows us to evaluate the percentage of correct classifications made by an algorithm [120], according to the following equation:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

Where:

- TP = True Positives.
- TN = True Negatives.
- FP = False Positives.
- FN = False Negatives.

Precision. Precision is a metric that obtains the rate of true positives within all the data classified by the [120] algorithm. The equation for its calculation is the following:

$$Precision = \frac{TP}{TP + FP} \tag{4.2}$$

Where:

- TP = True Positives.
- FP = False Positives.

Exhaustiveness (Recall). This metric deals with the rate of true positives identified in the correct classification made by the [120] algorithm. Its equation is the following:

$$Exhaustiveness = \frac{TP}{TP + FN} \tag{4.3}$$

Where:

- TP = True Positives.
- FN = False Negatives.

F1 score. The F1 score metric combines the precision equation and exhaustiveness (recall). The algorithm obtains false positives and negatives [120]. Its equation is given by:

$$F1 = 2\frac{Precision * Recall}{Precision + Recall} \tag{4.4}$$

4.2 Machine Learning Algorithms Used for Feedback in programming.

4.2.1 Discriminative AI.

4.2.1.1 Algorithms Implemented for Automated Feedback - Supervised learning.

Decision trees. Projects like [18] and [19] use decision trees for program analysis. These projects apply decision trees in educational settings, leveraging their visual representation to make them easily understandable for individuals without a background in machine learning. Preprocessed data trains the decision tree models, identifying the skill structures influencing student performance in programming tasks. The models also detect students at risk of failing and provide adaptive feedback to support their learning. In massive online courses, systematic decision trees analyze previous assignments to categorize questions effectively. This categorization helps generate tailored suggestions and examples, guiding students through programming challenges.

In the specific case of [19], features extracted from students' programming activities, including command logs and source code collected every minute during a programming language course, were used as input data. Key attributes such as the frequency of "while" or "for" loops, the number of function arguments, and indentation styles were processed and stored in a CSV file to train a decision tree model. The model visualized the skill structures influencing students' scores and performed regression to predict performance, offering an interpretable framework for educators.

Support Vector Machines (SVM). Support Vector Machines have been implemented to solve programming problems, where time efficiency and algorithm design capacity are determined. It is graded based on a rubric to measure a programmer's ability to solve a problem; the rubric provides objective feedback obtained through SVMs [15].

The system extracts key features from the student's code, such as programming construct frequency, complexity metrics, and coding standard adherence, to capture its functionality. These features are input to a machine learning model, which outputs a grade reflecting the program's quality and correctness.

Another work with SVM is [121], which applies linear and Gaussian SVM to learn and classify students according to the grades obtained; this is to determine which students are at risk of failing and generate adaptive feedback, but additionally, a performance comparison is made in this work with other techniques such as random forests, logistic regression, and k-means.

Case-based reasoning (CBR). This algorithm was implemented in the study [94], seeking to improve the quality of feedback for computer science students, using CBR to enhance the quality of feedback provided by automatic grading systems. The CBR system allows for the measurement of potential impact using an incorrectness similarity measure, where two programs are declared equally incorrect if some test cases fail; the presentations are grouped based on the similarity of incorrectness, and the feedback is generated accordingly.

The system takes features from students' code, such as programming constructs, complexity, and coding standards, as inputs. It compares these to a database of past submissions and feedback to identify similar cases.

Bayes Classifier (Naive Bayes). This algorithm has been implemented in [20] for the classification of two levels of feedback in programming tasks, one corresponding to the first level with sufficient or insufficient comments and the other second-level classification with subcategories including concepts, reflections, organization, and types of literal comments in generating feedback for programming tasks.

The system's inputs are comments extracted from students' source code, which are processed using natural language techniques like stop word removal, tokenization, and lemmatization. The trained machine learning model classifies each comment as "sufficient" or "insufficient" based on its usefulness in understanding the code.

K-Nearest Neighbor. The University of Dublin implemented the K-Nearest Neighbor algorithm in a virtual learning environment (VLE) [17], where students can submit their programming lab work and receive real-time feedback by running a set of test cases specified on the classification server.

The classifier computes uniform weights and uses Euclidean distance to measure similarity between the current interactions of students and previous instances in the database. It then identifies the "k" nearest neighbors those previous instances most similar to the current one. The model's output is based on the majority labels or rankings from these closest neighbors, enabling the prediction of the student's performance by comparing it to the behavior of similar students.

It can also be used for regression, but in this case, the average of the k nearest neighbors is taken to predict a classification [122].

4.2.1.2 Algorithms Implemented for Automated Feedback - Unsupervised learning.

Clustering (k-means). The K-Means Clustering technique, as implemented in [62], is utilized for educational data mining to automate the classification and analysis of programming tasks. This approach aids in providing constructive feedback through equality testing and alignment distances. The system analyzes students' problem-solving methods by monitoring the execution traces of their programs, which serve as the input to the algorithm. K-means clusters similar execution traces to identify recurring patterns in errors or solution strategies, helping to generated feedback.

Another study [71] developed a system to detect error types by clustering incorrect student solutions. The system takes previously submitted incorrect solutions as input

and applies clustering techniques to identify common error patterns. These clusters are then labeled, and the model uses them to classify new submissions based on the identified error type. The output is a classification that pinpoint the specific error in a student's new solution, offering targeted feedback to help correct the code. Similarly, in [93], an approach based on clustering the solutions given by students is presented to easily describe or exemplify the solution to a problem or provide feedback.

TipsC [23] attempts to find programs similar to a user's attempt and then suggests changes to the user's program with runtime logic error corrections. The input to the algorithm consists of the solutions submitted by students, which are analyzed and normalized. These solutions are then grouped based on common patterns of errors. Correct programs that match the incorrect attempts are listed, and changes are proposed to fix the user's program. This is done by analyzing the normalized submissions and grouping them according to their similarity, considering the edit distance and normalization metrics. This same technique is also applied in [25] and [93].

Principal component analysis (PCA). It is a technique applied as part of the exploratory data analysis, which seeks to describe them in terms of new uncorrelated components. It is implemented in Elivate [117], where data is taken from other platforms to analyze each student's progress. The system's inputs include detailed data on students' learning activities, such as code review progress, login times, and assignment submissions. Elivate processes this data using an educational taxonomy with PCA to reduce the dimensionality of the data. The system's output is an estimation of each student's progress and a prediction of the time needed to complete tasks, enabling the generation of personalized feedback.

4.2.1.3 Algorithms Implemented for Automated Feedback - Deep learning in Supervised Learning.

Long-term memory network (LSTM). One way to generate automated feedback is through source code repair. In [38], a programming language correction framework using deep learning is proposed. The program is represented through a sequence of tokens, which are processed by a Long Short-Term Memory (LSTM) network. The LSTM takes as input sequences of tokens from the student's code with syntactical errors and outputs corrected sequences of tokens, effectively repairing the errors. The agent can access and

modify the program text, not only identifying errors but also making the necessary edits to ensure the program compiles successfully. These error repair techniques, through the detection of syntactical errors, help inform the student about their mistakes and provide the correct solution.

On the other hand, the study [123] uses LSTM AttM, a model that combines Long-Short-Term Memory (LSTM) type neural networks with attention mechanisms to classify and detect errors in the source code. The inputs to the system are sequences of tokens representing the student's source code, which are processed by the LSTM network. The output is an estimated probability of error, classifying the code as either correct or incorrect based on the detected errors. These errors are evaluated at different levels, such as syntax and logic, providing a probabilistic assessment of the code's correctness.

Multilayer perceptron neural network. In another research [69], Annete is proposed. This tool generates intelligent feedback to assist students in a university Java programming course using a multilayer perceptron neural network. A sigmoid transfer function is used for propagation, an input layer with data collected from the students' programming assignments, a hidden layer, and an output layer representing the feedback given to the student. The input data consists of the students' code submissions, including syntax and logical errors, as well as interactions with the development environment. The network classifies the code based on these inputs, outputting feedback in the form of error categories such as correct or incorrect, and providing specific suggestions on how to fix identified mistakes. The output layer helps deliver targeted feedback to guide students in improving their code.

Convolutional neural network (CNN). In [124], NeuronalBugLocator focuses on locating semantic errors in programs through a convolutional neural network (CNN). The tool uses information from failed tests to identify specific semantic errors in student code. This technique is beneficial for identifying errors that are not readily apparent and that affect program functionality. The system processes program representations from student submissions along with corresponding tests. The output is a binary prediction indicating whether the program passes or fails the test. To pinpoint the specific lines of code contributing to the prediction, a neural attribution technique is used, providing a more detailed understanding of the error's location.

InferCode [125] is a tool that uses tree-based convolutional neural networks (TBCNN)

for natural language processing applied to abstract syntax trees (AST) of source code. InferCode specializes in predicting automatically identified subtrees, allowing practical training of code representations. Annotated trees provide a solid foundation for bug detection and classification, facilitating a more detailed understanding of code syntax and semantics.

Another study [126] employs a convolutional neural network (CNN) to classify code program errors using a tokenization-based representation and one-hot encoding. The process begins by transforming the code into a sequence of structural features, which is then converted into a one-hot binary matrix. This structural representation of the code allows the CNN to classify the code into one of six categories based on the algorithm it implements such as computational geometry problems (CGP), number theory problems (NTP), flow network problems (FNP), shortest path problems (SPP), query for data structures problems (QDSP), and combinatorial optimization problems (COP), . These categories are the output of the classification process, where the CNN analyzes the structural features of the code to identify its corresponding algorithm. The CNN works by extracting patterns from the one-hot matrix and learning to recognize distinct algorithmic structures. This approach not only aids in classifying code but also improves the efficiency and accuracy of identifying program behaviors, helping to analyze and correct errors more effectively.

Recurrent Neural Network (RNN). In the study [31], a recurrent neural network is implemented to train an l-dimensional classifier on tokens. Tokenization is performed through probabilistic grammars for programs based on synthetic blocks; the composition of terminal nodes constitutes a program, while non-terminal nodes constitute the labels or feedback. Feedback is generated through unit tests that emit pop-up suggestions with the programming assignment theme or compiler messages.

The authors, U. Z. Ahmed et al. [63], implemented recurrent neural networks to provide feedback to students who have compilation errors in the form of code repairs, using the Tracer tool to generate code repairs for a buggy program. Initially, the tool locates the error, abstracts the variables and literals with their types to predict the corresponding repair, and converts it into an abstract code representation.

On the other hand, Tegcer was implemented to provide automated feedback in the form of examples of programming tasks. If a program is erroneous, Tegcer locates the error lines, abstracts each line, predicts the error repair class to which the error line belongs, and suggests frequently observed training examples for that error. A dense neural

network is used to achieve 87% accuracy in prediction.

PIPE [127] is a tool designed to classify errors in software programs. Using a Recurrent Neural Network (RNN), PIPE evaluates precision, recall, and F1 measures to categorize mistakes in code. The process begins with the manual labeling of programs, where ten main types of errors are identified and annotated. PIPE then considers both the Abstract Syntax Tree (AST) representation of programs and the representation in embeddings to improve the accuracy in error classification. The RNN takes as input the processed program code, represented as a sequence of tokens or abstract syntax tree (AST) structures, and outputs a classification of errors. These errors are categorized into ten types: 1. Incorrect input variables, 2. No output, 3. Incorrect output format, 4. Incorrect initialization, 5. Incorrect data types, 6. Incorrect data precision, 7. Incorrect loops, 8. Incorrect branches, 9. Incorrect logic, and 10. Incorrect operators. This methodology allows for a detailed and accurate assessment of the errors present in the code.

Wang Ke. et al. [128], implemented SARFGEN, focuses on classifying errors made by students in programming exercises. Using a recurrent neural network, SARFGEN focuses on the accuracy of classifying errors in three specific types of exercises: conditionals and loops. The tool employs hybrid embeddings to generate detailed representations of errors, allowing for a deeper and more specific analysis of the difficulties students face in programming. This approach facilitates a more complete understanding of errors and helps improve educational feedback.

Abstract Syntax Tree Neural Network (ASTNN). In a comparative study between ASTNN and Code2Vec [129], both methods for error classification in basic Java programs, it has been observed that Code2Vec can outperform ASTNN when the amount of training data is small due to its ability to generate effective representations with less information. However, when large volumes of data are available, ASTNN proves to be more effective thanks to its ability to take advantage of the complexity of the AST structure for a more accurate classification.

Transformers. TFix [130] is a tool that focuses on generating automatic corrections for errors in source code. It uses a neural network-based approach with transformers, specifically the T5 (Text-to-Text Transfer Transformer) model, to analyze and correct errors in programs written in JavaScript. The tool employs static code analysis, processing the Abstract Syntax Trees (AST) of the code to identify different types of errors.

The tool takes code fragments with errors as input, represented as text, and generates corrected code as output. It is designed to address a range of error types, such as syntax errors, which involve issues in the structure of the code that prevent it from being correctly interpreted. It also handles semantic errors, which affect the logic and behavior of the program, and style errors, which concern inconsistencies in code format and style that can make the code less readable and maintainable.

These categories of errors are essential for improving the overall quality of code through automated debugging and correction, making TFix a valuable tool for software development.

The literature has addressed the use of deep learning in other activities involving the analysis of errors.

4.2.2 Generative AI.

4.2.2.1 LLMs.

Over the years, much research has focused on automatically generating feedback for programming tasks. Historically, these processes were more time-consuming and resource-intensive due to the need for large amounts of data and the complexity of the training processes. However, developing large language models (LLMs) has revolutionized this field. Today, with models such as OpenAI's GPT-3, GPT-4, Codex, and Google's Gemini-1.5, it is possible to generate accurate and efficient feedback for specific tasks such as creating text, code, and images.

For example, a recent study [131] used OpenAI Codex to generate code and provide feedback on Python programming exercises. The feedback was evaluated using a rubric considering reasonableness, novelty, problem and solution preparation, context, test coverage, and solution execution. In addition, the researchers injected errors into student solutions using GPT-4 to expand the dataset, highlighting Codex's capabilities to improve programming teaching and identifying the need to refine contextual understanding and detect more complex errors.

The authors provided textual prompts containing keywords and specific instructions that described programming concepts and contextual details relevant to the exercises. These inputs guided the language model in generating meaningful programming tasks and corresponding explanations. By leveraging large-scale pretraining, the model could

interpret these prompts and produce structured outputs aligned with educational objectives.

Another study [132] explored the use of GPT-4 Turbo for generating feedback on programming exercises. The model received as input both the problem specification and student solutions, producing feedback that was later evaluated based on several dimensions. The evaluation criteria included the content and structure of the feedback, the representation of code, the correctness and types of corrections suggested, proposed optimizations and coding style, as well as inconsistencies and redundancies in the generated feedback. This analysis aimed to assess the effectiveness of GPT-4 Turbo in providing meaningful and pedagogically useful feedback to support students in improving their programming skills.

An automatic feedback system using GPT-3.5 has also been developed to suggest corrections and improvements to student code submissions in Java [133]. This system is characterized by its ability to correct logical and code style errors and manually verify the syntactical and functional correctness of submissions using unit tests. However, it was observed that the order of suggestions can appear random, suggesting the need to improve the sequence of suggestions and the detection of inner classes and encapsulation in Java.

Another work [134] examines how GPT-3.5, implemented in the ChatGPT platform, responds to feedback requests for Java programming exercises involving calculations, classes, methods, constants, conditionals, and enumerations. Using Keuning's taxonomy [14], feedback is evaluated in content, quality, style, examples, causes of errors, solutions, misleading information, and motivational messages. The findings highlight the challenges current language models face in adapting to diverse student needs and the importance of personalization and accuracy in the suggestions generated.

In addition, web interfaces such as StAP [30] for Python practice have been created, which integrate feedback generated by GPT-3.5-turbo. This system is designed to help students improve their programming skills by providing tailored feedback on conditionals and loops. Feedback is evaluated on criteria such as level of detail, personalization, appropriateness, specificity, tone, and length. The study highlights differences in feedback perception between teachers and students, underlining the need to tailor feedback to maximize its effectiveness in learning.

4.2.3 Performance Metrics for Evaluation and Feedback Generation.

This section classifies the machine learning algorithms implemented in some studies on automatic feedback. Two types of algorithms are found: classification and regression. In the literature review, it was observed that the most used algorithms are classification algorithms, and there are several studies in which this type of algorithm has been implemented in convolutional and recurrent neural networks, such as [135] [136] and [137].

Classification algorithms have generated significant contributions to this research. In the study [107], the AutoGrader tool measures performance using the accuracy metric, defined as the portion of correctly classified cases, where a 92.80% accuracy was obtained. Likewise, in the tests carried out, no false negatives were found. Still, concerning false positives, 11 submissions with errors were found, labeled as correct because the test cases were incomplete.

In other research [82] [138], the accuracy metric is used to determine how many correct classifications the algorithm obtained. The PEX4FUN tool obtained a 64% accuracy by sending feedback to the incorrect presentations submitted by the students in approximately 10 seconds.

In the evaluation of IQCheck [65], precision and recall metrics were applied to assess its effectiveness in identifying appropriate code identifiers. The study measured both correctness and completeness, finding that IQCheck correctly identified most of the identifiers deemed appropriate by instructors. When comparing automatic and human assessments, the results showed a precision of 75.0% and a recall of 82.6%, indicating that completeness (recall) was higher than correctness (precision). Additionally, confidence interval analysis confirmed these findings, with precision ranging from 63.7% to 83.5% and recall from 71.8% to 90.4% at a 95.0% confidence level. These values were derived using the ordinary non-parametric bootstrap method, resampling identifier names 2000 times, and computing confidence intervals through the bias-corrected and accelerated method.

Another research that measures completeness is [139], which evaluates false positive and false negative rates for each programming task in Web-CAT. Specifically, the study reports the false positive rates (clues revealed as feedback but not gained) and false negative rates (clues hidden but gained) for two projects: Minesweeper and Queues. For Minesweeper, with 1,604 submissions, the average false positive rate was 0.72 (SD = 0.41), while the false negative rate was 0.19 (SD = 0.33). In the Queues project, with 2,084 submissions, the average false positive rate was 0.84 (SD = 0.32), and the false negative rate

was 0.21 (SD = 0.30). These results indicate that high rates of false positives were observed, suggesting that many of the clues revealed during the feedback process were not ultimately gained by the students, irrespective of the task at hand. This indicates that, on average, the feedback system has a fairly good ability to detect errors in the code, with a recall close to 80% in both projects.

In the study [19], the decision tree's purpose was to visualize the skill structures that influence students' scores, providing an interpretable model for educators. Additionally, regression was performed using the decision tree to predict students' scores based on the extracted features. The model was validated by calculating the coefficient of determination (R^2) to measure how well the predictions aligned with the actual scores. Through this process, the study aimed to identify key programming behaviors and patterns that impact student performance while ensuring the results were accessible and interpretable for educators unfamiliar with machine learning techniques. The model's performance was further evaluated using precision, recall, and the F-measure. Specifically, precision and recall were calculated by combining microaverages, given the multiclass classification nature of the problem. The model was considered good if all predicted values for precision were 0.0 and for recall were 1.0. The F-measure, which considers both precision and recall, was used as the primary evaluation metric. The precision calculated for the model's performance was 0.714 in F-measure.

The summary of metrics used in automatic feedback algorithms found in the literature is shown in Table 4.1

No.	Algorithm type	Metrics	References
1	Classification.	Precision, recall, accuracy, F1-score, bleu.	[82] [138] [65] [139] [140] [119] [125] [127] [129] [123] [130]
2	Regresion.	R^2, F-measure.	[19]
3	Other techniques.	Fuzz test, Friedman test.	[139][141]

Table 4.1: Metrics for automatic feedback algorithms.

On the other hand, there is the fuzz test, which is not a metric used in machine learning but has been used in the AutoGrader tool to generate automatic feedback [141]. Fuzz testing is a random technique to verify the behavior of programs. This research identifies when the solution to a task submitted by a student differs in behavior from the reference

implementation, generating random inputs in a program and verifying the properties of the output or program.

4.3 Automated Feedback Platforms for Supporting Programming Learning in Programming Courses.

We identified 74 different platforms supporting programming learning between 2010 and 2024. Among them, edX stands out as the most widely implemented platform offering programming courses with personalized feedback [58] [73] [83] [87] [93]. In addition to edX, other platforms like Code.org [31], Coursera [72], Codewebs [75], as well as platforms developed by various educational institutions, have also been explored for providing similar learning experiences (see Table 4.2).

To develop this research, it is necessary to determine the platforms that have implemented automatic feedback. To do so, the classification of platforms or tools cited in the articles [14] [80] [82] was taken. The following are defined:

Intelligent Tutoring System ITS Its objective is to provide personalized support to students, finding errors in programming and providing adequate feedback so that students can correct their programs without requiring direct help from the teacher [142]. To do this, AI techniques are applied with the implementation of extensive problem-solving modeling for the domain of this type of application; as an example, there is Ask-Elle [56], an adaptive programming tutor for the Haskell language, which generates model tracking through programming strategies derived from model solutions that generate clues about what to do to solve a task. In the same way, there are other platforms aimed at intelligent tutoring in different programming languages, such as TipsC for C language, Lips for the language named with this same name, and J-LATTE and Fit for Java language [14].

Automatic Program Repair APR. As its name implies, it fixes software compilation errors automatically; it requires a set of tests that drives this repair process [80]. Typically, what has been implemented is a partial repair, where students can identify the failures and approvals of a program, allowing the student to understand why their program fails and encouraging them to modify their program, taking into account the partial repair instead of blindly accepting.

No.	Platform type	AI techniques	Tools	Contribution
1	APR.	Decision trees, Neural networks.	CodeHunt, edX, RLAssist, DeepFix, Tegcer, Tracer.	Corrects software compilation errors automatically, using a hierarchical search engine, a comparator, and a repair minimizer [80].
2	ITS.	K-means Clustering, K-neighbors, CBR, SVM, Random forest, Markov, Bayes, Neural networks.	Challenge, AskElle, Feedback for Rule, Grok Learning, MindReader, PythonTutor, TipsC, Annete, MOOC Stepnik.	They allow us to find errors in the programs or presentations delivered by the students and provide the appropriate feedback. These types of tools have a task description, model solution with feedback messages, and the properties that a solution must have [56] [14].
3	Domain-specific programming techniques.	Neural networks, SVM, K-means Clustering, PCA, Random forest, Bayes.	OeLE, OverCode, JPLAS, Junit, ICDE on edX, IIT Kanpur, RIT, Sketch, Clara, Code.org, IAPAGS, Engage, Pycaprser, IQCheck, TMC, Cod Hunt, Hint Factory, PABS, APEX, Elivate, PA3P, JACK, Travis-CI.	Basic static analysis of programs or through dynamic code analysis using automated testing. Unit testing and property-based testing are implemented to evaluate the code and generate comments [14].

Table 4.2: Tools with automatic feedback.

Domain-specific programming techniques. They analyze a program statically and dynamically by examining the source code without executing and after executing to compare the expected result. The most commonly used techniques in this type are unit tests and property-based tests, such as in the case of platforms based on JUnit, TMC, and AutoGrader, now called Sketch, which run tests to evaluate the source code and generate comments. In addition, they perform intent diagnostics through internal or external testing tools such as standard compilers or code analyzers [14].

Other strategies. Corresponds to platforms or tools that do not fit the characteristics above.

Our analysis reveals that the most commonly used types of tools to generate automatic feedback are the intelligent tutoring system (ITS) and domain-specific programming techniques, each with a 41% implementation rate. This information can inspire further research and development in these areas, leading to more effective feedback generation tools.

4.4 Background on Machine Learning and Automated Feedback in Iterative Programming.

In the literature review, three research works have made significant contributions to learning and providing automatic feedback in programming tasks involving iterations. Lienardy, in his studies [29] [32], discusses the online platform Cafe, which is designed to evaluate and provide automated feedback on student progress in a Computer Science (CS1) course. The platform specifically focuses on a series of programming exercises centered around loops, offering feedback both during the process and at the end of each exercise. Cafe's feedback mechanism is aimed at providing early insights to help students refine their solutions.

The research utilizes an informal version of the invariant loop to assess the correctness of program loops with each iteration, specifically evaluating the loop condition at every step. The invariant is used to express the loop's property through a logical statement before coding begins, following Dijkstra's approach of dividing the code into three sections: Zone 1, where the loop condition is evaluated; Zone 2, which contains the loop body along with the invariant; and Zone 3, the code executed after the loop.

In terms of learning outcomes, the research highlights that the Cafe platform, inspired by the Assessment for Learning (AfL) framework [143], provides students with feedback on their progress, empowering them to take corrective actions to improve their performance. A survey conducted at the end of the course revealed that 45.5% of students recognized that the feedback helped them identify learning gaps, particularly noticing deficiencies in three or more challenges. Another key finding from the survey was that students felt more aware of their learning gaps due to the feedback they received, illustrating the platform's impact on self reflection and improvement.

In another research, Akram et al. [21] evaluate the areas of strengths and weaknesses regarding basic concepts of Computer Science, such as loops and conditionals. In the course studied, effective and personalized scaffolding and feedback are generated to support students' mastery of essential computer science concepts for programming. The dataset used for this project had to be annotated. Code annotation is done with an assessment rubric focused on benchmarks such as Effectiveness and Conciseness. The rubric assesses the design and implementation of practical and generalizable algorithms, the appropriate use of loop statements, conditional statements, and a combination of loops and conditions. Likewise, a hypothesis-based learning analytics approach is used through an evidence-centered design to identify the core computer science concepts highlighted in an environment and assess students' competencies concerning each of the target concepts.

Also, another study, such as [31], presents a rubric approach to address feedback in a programming fundamentals course. The Zero-Shot challenge is to infer the misconceptions that led to student response errors. The tool can attribute feedback to specific parts of the code, track learning throughout the course, and generate synthetic data sets. Students' understanding of loops and geometry is assessed at the end of the course.

The research conducted on accelerating the automated evaluation of programming exercises introduced an innovative approach that utilizes caching to identify programming assignments similar to those submitted by students. When a match is found, the system generates corresponding correction comments. If no match is found, the system performs a static analysis of the code to detect syntax errors. Additionally, incorrect programs are subjected to unit tests for verification. The approach was applied to basic programming tasks involving loops [144].

Other more recent models use machine learning (ML) and large language models (LLM), trained on large amounts of unlabeled text using machine learning.

The researches in [33] evaluated the quality of feedback generated by GPT-3.5 for

students' programming assignment submissions. The programming tasks evaluated included an exercise that required using the "while" loop constructor and another exercise focused on list management. Regarding the comments and suggestions generated by GPT, these covered various types of errors, including aspects of style, syntax, and logic. Regarding classification between correct and incorrect tasks, GPT achieved an accuracy rate of 73%. However, in terms of pertinent feedback, it obtained 47%, evidencing that errors or suggestions that needed to be adequately aligned with the specific instructions of the task were presented.

In another study [30], the authors developed the StAP web interface, designed for students to practice their Python skills with step-by-step hints. Using CodingBag's "clumpCount" exercise, which includes conditionals and loops and is solved in multiple steps, feedback generated by GPT-3.5-turbo was implemented. Feedback was requested for each step of the problem solution and then evaluated using the feedback types defined by Keuning [14].

During the experiment, students were asked to make several requests for hints and evaluate them without focusing on their level of programming experience. The students were encouraged to request hints at the start of each step. In addition, each hint was evaluated using three key questions: whether the answer was clear, whether it helped solve the concern, and whether it fit the student's work.

4.5 Chapter Summary.

Based on the information collected in the initial phase of this research, there are few studies focused specifically on automated feedback for programming tasks involving iterations. Most existing work revolves around generating general automated feedback for programming tasks, such as evaluating whether the code is correct or incorrect, with a predominant emphasis on the functional aspects of the code. However, these studies rarely integrate the theory of how to program iterations, as outlined by Gries in The Science of Programming (1987). On the other hand, it is crucial to define the type of feedback that supports learning, as described by Keuning et al. (2018), aligning with the intervention levels suggested by Hattie and Timperley (2007). Regarding machine learning (ML) techniques, most studies have primarily used supervised and unsupervised learning methods for feedback generation, with only a few exploring reinforcement learning or deep learn-

ing approaches. Additionally, with the rise of generative AI, several studies have started to leverage these technologies to generate comments, produce code, and develop new functionalities. While it is promising to take advantage of these advances, it is still essential to have a deep understanding of prompt engineering to ensure that AI delivers the required outputs effectively.

In contrast, our approach combines the use of discriminative artificial intelligence (AI) by applying deep learning algorithms for error classification with generative AI, utilizing large language models (LLMs) to generate feedback tailored to the specific errors detected in the code. This approach allows for the provision of more precise and contextualized feedback that not only addresses the functional aspects of the code but also incorporates a deeper understanding of the theory behind iterations.

Part II

Taxonomy Development and Dataset Creation

Chapter 5

Taxonomy of Errors in Iterative Programming Tasks for Automated Feedback.

Contents

This chapter develops a taxonomy of errors in iterative programming tasks, providing a structured framework for identifying and classifying common errors in programming, particularly in tasks involving loops. Section 5.1 introduces the taxonomy, reviewing related work and placing the study in the context of previous research. It also outlines the most frequent error categories in iterative programming, such as errors in while loops, and

the feedback types that address these issues. Section 5.2 details the methodology used to create the taxonomy, including the definition of properties, conceptual relationships, and error categories to ensure a clear and consistent classification. Section 5.3 presents the finalized version of the taxonomy, which serves as the foundation for to classify of errors in iterative programming tasks in educational technologies. Section 5.4 concludes the chapter, summarizing the key aspects and laying the groundwork for applying the taxonomy in future systems and experiments.

5.1 Overview of the Taxonomy.

5.1.1 Related Work.

Beginning programmers often need help understanding how loops or iterations work, and many of their errors are concentrated in conditional structures and loops. Adequate feedback is essential to improve computer programming learning, but a specific approach is needed to address the mistakes in iterative programming tasks.

It's worth noting that there is a dearth of literature exclusively addressing feedback on errors in iterative programming tasks. However, some studies have proposed taxonomies to classify errors in introductory courses. For instance, in [34] developed a task-based classification aligned with the levels of Bloom's taxonomy, identifying patterns of syntactic, semantic, and pragmatic errors. Similarly, another authors [36] focused on the difficulties of beginning students in understanding programming syntax and semantics, especially with concepts such as state transformation in loops and control variables.

Taking another approach [35] used the "SOLO" taxonomy to analyze specific errors in tasks involving loops and vectors. They classified errors into levels that reflected the complexity of learning, from basic errors, such as empty or unedited loops, to more advanced relational errors, such as incorrect index initialization and confusion in vector manipulation.

5.1.2 Conceptual Framework for Error Categories in Iterative Programming and Feedback Types.

Establishing a clear conceptual framework is essential for properly defining the taxonomy of errors in iterative programming tasks. This framework focuses on three fundamental

aspects: the types of errors commonly found in these tasks, the characteristics of programming tasks involving iterations, and how these errors can serve as the foundation for generating targeted and meaningful feedback for students.

First, the **Programming Concepts with Iterations** must be understood, as detailed in Section 2.2. From this analysis, the categories of errors that we seek to identify and classify in our research are derived. These categories align closely with the assertions presented in Chapter 2, where we discuss the challenges and characteristics of iterative programming tasks.

From this foundation, we classify errors into three main categories:

- Error in the initial state: The precondition is not satisfied.
- Error in the state transformation: The invariant is not maintained during some iteration.
- Error in the final state: The postcondition is not guaranteed in the final state.

While the termination error (related to the "threshold") is traditionally included in similar frameworks, our study does not consider it as a separate category. Instead, we treat the "threshold" implicitly as part of the final state. This decision simplifies the taxonomy and aligns it more closely with our focus on providing actionable feedback for iterative programming tasks.

Accurately identifying these three types of errors plays a crucial role in generating specific and targeted feedback. This feedback not only addresses the detected issues but also supports the student's learning process by fostering a deeper understanding of programming concepts related to iterations.

Second, it is crucial to analyze the **Feedback** that can be generated in the programming tasks.

As mentioned above, according to [11], educational feedback can be classified into different levels, such as feedback on the task (Feedback Task, FT), on the process of completing the task (Feedback Process, FP), of self-regulation (Feedback Self-regulation, FR), and personal feedback (Feedback Self, FS). Each type of feedback has a different impact on learning, and it is essential to identify which one is more relevant according to the educational context and learning objectives.

Our research focuses on feedback at the task level (FT), which provides direct feedback on students' errors when completing a specific task. This type of feedback is precious for

correcting misinterpretations or conceptual failures that arise not from a lack of information but from an incorrect understanding. Feedback at this level is effective because it allows students to identify and correct erroneous hypotheses, offering clear instructions to rectify their errors and improve their performance in future tasks.

We have combined this approach with the Narciss feedback model, which has been adapted to the programming field. We have also followed the adaptations proposed by [14]. These adaptations focus on knowledge of mistakes (KM) and validation through failure tests (TF).

Failure tests are particularly relevant in programming, as they allow the validation of code by comparing the expected inputs and outputs with the real ones. When a test fails, that is, specific feedback is generated when the program does not produce the expected output. This feedback can not be just any feedback; it must be precise and focused on the specific error. This feedback facilitates the learning process and improves the student's ability to write correct code in future tasks.

5.2 Taxonomy creation and development process.

5.2.1 Methodological approach for taxonomy creation.

Four steps will be taken into account to develop the taxonomy, as cited in the literature review on the taxonomy of knowledge by [145]:

1. Determine the requirements of the taxonomy: Before starting, it is essential to define the scope, purpose, and content formats that the taxonomy cover and identify the target audience.

   The main requirement of the taxonomy is to identify and classify the most common errors in iterative programming, with a specific focus on the use of "while" loops. The purpose is to facilitate the generation of particular feedback tailored to these errors, supporting both students and instructors in introductory programming courses. This feedback will allow students to understand and correct their mistakes.

   Regarding the format of the content, the taxonomy should include clear error categories, practical feedback examples, and a definition of the relationships between key concepts, such as logical errors. The structure should also be intuitive to allow

easy navigation through the different categories, ensuring that users can apply the feedback appropriately in their programming assignments.

This taxonomy's target audience is students and instructors of introductory programming courses, primarily those working with Python. The tool will be designed to enhance the teaching and learning of programming with iterations by accurately identifying errors and delivering specific, relevant feedback.

2. Identify the concepts within the taxonomy: To identify the key concepts, we use a three-pronged approach: determine the content's location and nature and conduct a complete inventory of it.

   A three-pronged approach was adopted to identify key concepts within the taxonomy: determining the location and nature of the content and conducting a comprehensive inventory of the content. Searching and analyzing relevant literature allowed us to identify fundamental concepts and everyday difficulties in teaching introductory programming. Studies that analyze syntax, loops, strategic knowledge, and pedagogical challenges were examined, allowing us to build an organized inventory of relevant content.

   The study [146], classifies introductory programming concepts into three main categories: syntactic knowledge, which includes keywords, operators, and delimiters; conceptual understanding, which covers sequential, conditional, and iterative structures such as loops; and strategic knowledge, related to planning, writing, and debugging programs. These categories allow us to address common difficulties such as cognitive complexity, conceptual errors, and inappropriate strategies.

   In [4], the difficulties encountered are categorized into three distinct stages. In the programming stage, challenges such as confusion in the use of loops and flaws in mental models are emphasized. During the problem-solving stage, the focus shifts to the impact of ineffective pedagogical materials. Finally, in the implementation stage, barriers such as negative attitudes and limited transfer of prior knowledge are identified.

   In another study [6], the authors identified problematic areas in introductory programming, such as the abstract formulation of problems, the correct expression of the solution through control structures, and the evaluation of solutions with debugging techniques. These studies highlight additional challenges for teachers, such as

the choice of programming language, the scalability of problems, and the need to offer effective feedback.

To develop the taxonomy, we prioritize common errors in loops due to their significant relevance in introductory programming courses [7] [4]. These errors include uninitialized variables, improper use of global or non-local variables, unintentional modifications to loop variables, incorrect or missing end conditions, and infinite loops. Each of these errors reflects underlying conceptual misunderstandings of iterative programming, such as the failure to properly initialize or update variables, mismanaging scope, or incorrectly formulating conditions that control loop termination. Addressing these errors requires both identifying their presence in the code and providing conceptual explanations that help students understand why these logical errors occur and how to correct them.

This taxonomy specifically incorporates these common logical errors in loops because they are critical to developing a deeper understanding of iteration in programming. While previous studies have proposed broader classification schemes, such as those focusing on syntax, strategic knowledge, or abstract problem-solving, these are not explicitly included in our taxonomy. Syntax errors, however, are indirectly addressed through the integration of a compiler in the tool, and these comments are supported by IMProB-It.

By focusing on the most frequent logical errors in iterative programming, this taxonomy remains streamlined and directly aligned with our objective: enabling targeted and conceptually grounded feedback for iterative programming tasks.

3. Develop and refine the taxonomy by organizing the top levels into main categories. Limit the subject areas to a maximum of ten to ensure easier navigation within the hierarchy.

   To simplify navigation within the hierarchy, the focus was placed on the types of errors detected in tasks based on failures identified through specific tests. Following these failures, feedback was generate to address the type of error, detailing its cause and whether it impact the problem's invariant. This approach avoids unnecessary complexities and enables a precise classification of the types of feedback applicable to errors in programming tasks involving iterations.

   The preliminary concepts were validated through a review with some professors

from the Universidad del Valle, who contributed their experience on the most common errors students make when working with loops. They also provided recommendations for offering appropriate feedback to facilitate the correction of such errors. This validation stage allowed the taxonomy to be refined, adapting it to the specific needs of introductory programming courses and ensuring its usefulness in real educational contexts.

Following the drafting, an iterative review process was undertaken in which improvements, adjustments, and potential errors were identified. This stage provided essential information to guide the refinement of the taxonomy.

The refinement also followed an iterative approach, incorporating feedback from users who reviewed the taxonomy and observations made by the working team. This validation phase allowed the taxonomy to be adapted to the specific needs of introductory programming courses, ensuring its usefulness in natural educational environments. The result is a structure aligned with the practical and theoretical needs of the context in which it will be applied, ensuring its relevance and functionality.

4. Apply the taxonomy to content: Users must understand the structure of the taxonomy and the relationships between concepts for its proper application.

   To ensure proper application, the structure of the taxonomy and the relationships between its concepts must be understood. This understanding facilitates its consistent and accurate use in organizing and classifying errors.

   The taxonomy was applied in two key works:

   - Creation of the dataset: The defined structure served as a guide to organize and label the records, ensuring that each entry was aligned with the established categories.
   - Definition of error categories: The taxonomy was used to define the classification model with artificial intelligence, precisely the types of errors in programming tasks with iterations, which subsequently facilitates the generation of adapted and pertinent feedback.

   Applying the taxonomy in practice demonstrates its ability to maintain conceptual coherence while offering tangible value for structuring and analyzing data on errors

in programming tasks.

5.2.2 Definition of properties, concept relationships, and categories.

The taxonomy categorizes error types in programming tasks with loops and establishes a clear connection with the feedback tailored to address each type of error. Based on the framework from [11], which identifies four levels of feedback-task, process, self-regulation, and self this taxonomy narrows its focus exclusively to the task level. By doing so, it ensures that feedback remains directly aligned with the detected errors, providing precise and actionable guidance to address issues in iterative programming tasks.

This study focuses on task-level feedback, specifically analyzing how programming tasks involving loops are executed. At this level, task-level feedback often referred to as corrective or outcome based feedback aims to identify correct and incorrect solutions while providing additional information or alternative guidance to enhance students' understanding and support the correction of errors.

At a second level, the types of errors that will be evaluated at the task level are addressed. Here, Narciss [13] defines errors as those related to coding or the result obtained in a task. Knowledge feedback messages about errors (KM) can vary in detail, from a numerical value of total errors to a detailed description of the error or the specific location in the code. Errors are classified into five main types: Test Failures (TF), Compilation Errors (CE), Solution Errors (SE), Style Problems (SI), and Performance Errors (PI).

At the third level, according to the types of error feedback, the following levels are established in the taxonomy for feedback in programming tasks with iterations:

- **Test Failures (TF) and Solution Errors (SE):** Combines test failures and solution errors to evaluate whether a program produces the expected output and complies with the required behavior according to logical errors or execution times. It focuses on:

  - **Initial state:** According to the given or identified specification, it is determined whether the initial state corresponds to the one required for the scheduling of the iteration.

    * Does not satisfy the invariant: It must be shown that the initial state does not satisfy P. That means that $-(\{Q\}\, S := S_0\, \{P\})$.

  - **State transformation:** According to the given or identified specification, whether the formal definition of the state transformation is correct is determined.
    * The invariant is not satisfied from a new state: The new state resulting from an iteration does not satisfy P. That means that $-(\{P \wedge B\}\, D\, \{P\})$.
    * Does not advance to the completion of a state: The new state resulting from an iteration does not advance towards completion. This means that $-(\{P \wedge B \wedge S_1 = S\}\, D\, \{t(S) < t(S_1)\})$.
    * Missing steps to finish the iteration: The final state is reached, but there are still steps to finish. This means that $P \wedge \neg B \wedge t(S) > 0$ is true.
  - **Final state:** To determine that the final state is correct, stopping tests can be applied. These tests allow for the determination of whether the loop should stop because it has already met the desired condition and the program continues beyond the loop. To verify this, the following stopping tests are defined:
    * Does not meet the postcondition: That means $P \wedge \text{Final}(S) \not\Rightarrow R$ is true.

- **Compilation errors (CE):** Refers to syntactical or semantic errors that a compiler can detect and that are not specific to an exercise. In the syntax of the `"while"` loop:
  - The `while` keyword.
  - A condition that determines whether the loop will continue execution or not, based on its value (`True` or `False`).
  - The opening brace ({) at the end of the first line.
  - The sequence of instructions or statements to be repeated. This block of code is called the "body" of the loop and must be indented within the opening and closing braces of the loop body.
  - The closing brace (}) at the end of the iteration.

- **Style problems (SI):** Related to messy formatting, inconsistent names, or missing comments, which, although not serious, affect the readability and maintenance of the code.

- **Performance errors (PI):** Occur when the program uses more resources than necessary, such as excessive execution time.

According to the methodology, the concepts and taxonomy in programming with iterations are refined after carrying out an inventory of content. Professors from the Universidad del Valle who have taught introductory programming courses were interviewed to gather their experience in teaching and learning iterations.

5.3 Final Taxonomy.

Figure 5.1 shows a taxonomy that classifies common errors in programming tasks, specifically in the context of automated task-level feedback. This taxonomy is organized into several main categories and divided into subcategories to describe further the different types of errors that can occur during programming, especially when working with loops.

- **Task Errors:** This is the main category under which all errors are grouped. This category is further divided into four main subcategories:
  - Test Errors
  - Compilation Errors
  - Style Errors
  - Performance Deficiencies
- **Test Errors:** Errors in this category occur when the code fails predefined test cases, signaling issues in logic or implementation. For these errors, we provide targeted feedback to help address and resolve them.
  - **Initial state:**
    - Does not satisfy invariant: This refers to the initial state of the program not meeting the expected invariant condition, which must remain true during the execution of the loop.
  - **State transformation:**
    - Formal representation of the transformation: Indicates how the program states should be transformed during the execution of the loop.
    - New state does not satisfy invariant: After an iteration, the new state of the program does not satisfy the invariant, indicating an error in the transformation logic.

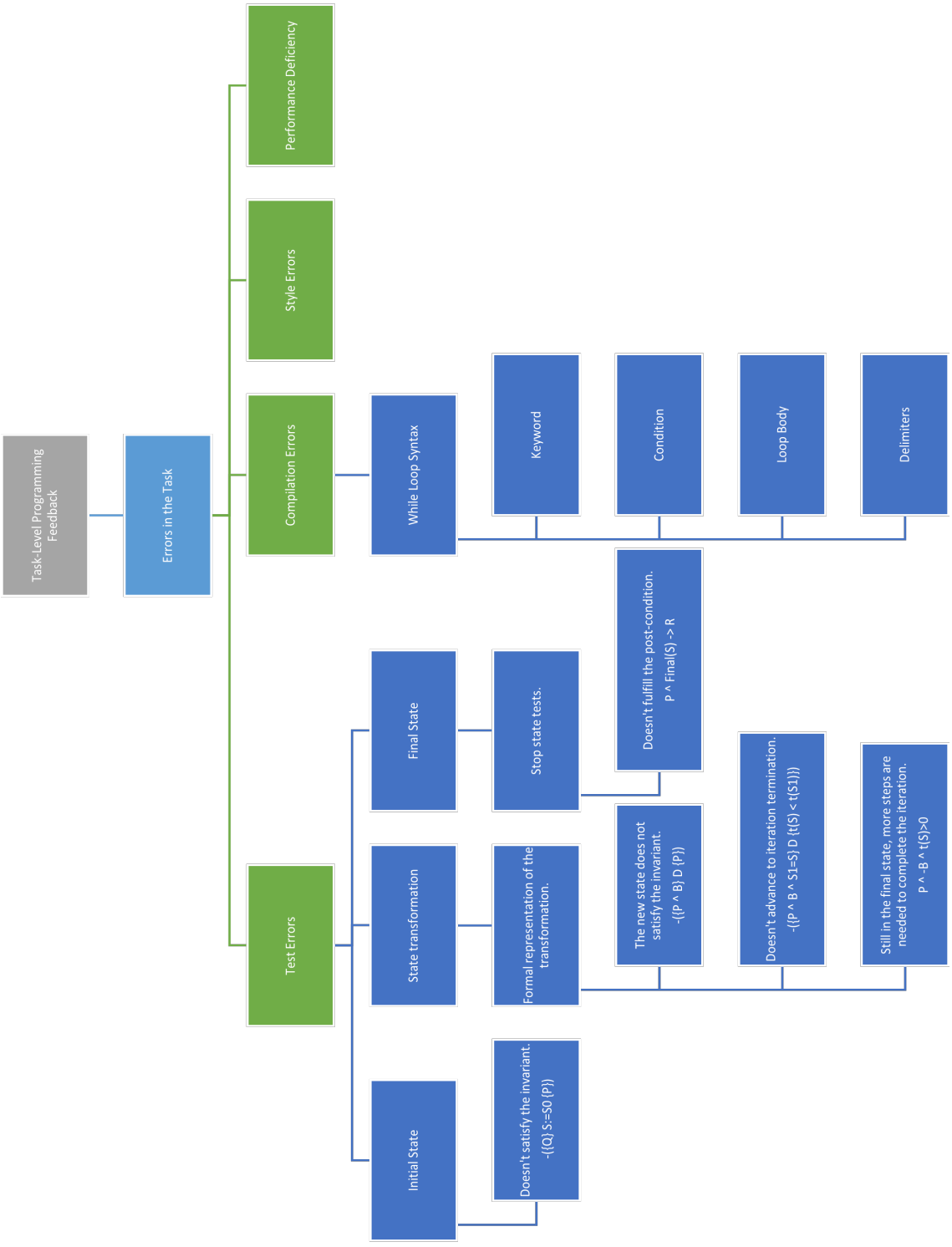

Figure 5.1: Final taxonomy.

    - Does not advance to the end of the iteration: The loop does not advance towards its termination condition, which can lead to infinite or incorrect loops.
    - In the final state, steps must still be completed to end the iteration: Despite having reached a state that should be final, the loop does not stop properly, suggesting a problem with the stopping condition.

- **Final state:**
    - Stopping state tests: Evaluate whether the final state of the program meets the expected termination condition.
    - Does not meet postcondition: The final state does not meet the postcondition, which is the condition that must be true at the end of the loop's execution.

5.4 Chapter Summary.

This chapter focuses on the development of a taxonomy of programming errors, with a particular emphasis on iteration errors involving "while" loops, grounded in program correctness theory. It includes a review of the literature on taxonomies and categorizations of common loop-related errors, establishing a clear connection to the feedback generated for these errors. The methodology employed to create this taxonomy involves identifying key concepts and classifying errors at various levels, including syntactic, semantic, and pragmatic. The primary objective of this taxonomy is to define the types of errors commonly made in programming tasks with loops, serving as a foundation for generating automated feedback to these specific errors in iterative programming tasks.

Chapter 6

Creation of a Synthetic Dataset for Iterative Programming Tasks.

Contents

This chapter describes creating a synthetic dataset for iterative programming tasks to support automatic feedback generation in introductory programming courses. Section 6.1 defines the purpose of the dataset and presents the specific rules for its creation, accompanied by illustrative examples that demonstrate its application. Section 6.2 details the creation process, describing the structure and key features of the dataset, as well as each step followed from its conception to implementation. Section 6.3 introduces the final dataset, highlighting its key features and information relevant to its use in experiments and automatic feedback systems. Finally, Section 6.4 summarizes the chapter, synthesizing the most critical points and setting the stage for its practical application in error analysis and feedback generation in educational settings.

The dataset presented in this chapter was specifically developed as a dedicated resource for basic programming tasks involving iterations, particularly those using "while" loops, to support the implementation of machine learning models. It was created using basic programming tasks with iterations from projects such as CodeNet, INGInious, and other platforms. From these exercises, additional solutions both correct and incorrect were generated manually and with the assistance of GPT.

6.1 Rules for Creating the Synthetic Dataset.

We established the following rules for the creation of the data set:

6.1.1 Provide a detailed description of what to solve in a computer programming problem.

Each programming task must include a well-crafted and detailed problem statement. The problem statement is crucial for the creation of the dataset, as it defines the scope and requirements of the task to be solved.

6.1.2 Get a "while" loop's initial condition, end condition, and state transformation.

Identifying the initial and final conditions of a "while" loop is crucial for determining the range of values over which the loop operates. Understanding the state transformation the way variables change within the loop plays a key role in detecting potential infinite loops or logic errors. This analysis provides essential insights into the loop's behavior, helping to identify issues early and ensure the correctness of the code.

Determining these components for any loop code is essential. This process enhances understanding of the code, its limitations, and potential issues. Additionally, this information is valuable for testing the code and verifying that it meets the problem statement requirements.

6.1.3 Provide specific test cases.

We defined the expected result to guarantee consistency across all solutions and alignment with the desired outcome. Additionally, by verifying that the answer is correct by testing the code against various inputs, we can ensure that it works as expected. If the code doesn't produce the desired result, we can use the test cases to identify errors in the code, isolate where the problem is occurring, and work on a fix.

In general, providing a detailed description of what to solve in a programming problem and providing test cases helps gain confidence that the solution is clear, consistent, and correct.

6.1.4 Generate both correct and incorrect solutions to programming tasks using "while" loops .

Creating both correct and incorrect solutions is crucial for training a model to classify errors in source code. This approach enables the model to distinguish between valid and faulty code effectively. By providing diverse examples of correct and incorrect solutions, the model can identify common patterns and typical errors in iterative programming.

By offering a range of correct and incorrect solutions, it becomes possible to classify errors systematically and develop a more robust automated feedback model, capable of delivering accurate and valuable feedback on errors to users.

6.1.5 Categorize incorrect solutions according to the error types specified in this investigation.

The dataset features a detailed error classification based on the taxonomy outlined in Chapter 5, aligned with program correctness theory [26], focusing on the initial state, final state, and state transformations that define the loop invariant. This classification is essential for identifying errors stemming from incorrect handling in iterative programming tasks.

6.1.6 Examples Illustrating the Established Rules.

Understanding and correctly applying control structures, such as "while" loops, is fundamental to learning computer programming. Establishing a clear framework for designing assignments and evaluating solutions is essential to helping students develop strong skills in implementing these loops. This section provides examples of programming problems involving "while" loops created following the established guidelines for crafting assignments.

These examples demonstrate the application of these rules and serve as a reference for building datasets to train models. Each example includes a detailed problem description, specific test cases, and an analysis of the loop's initial condition, final condition, and state transformation. Correct and incorrect solutions are also provided to highlight common errors and support continuous improvement in the programming teaching-learning process.

a) Calculation of the Factorial of a number.

Problem description: Write a Python function that returns the factorial of a number using a "while" loop. Remember that the factorial of a number results from multiplying that number by all the numbers that exist from it to 1.

Test cases:

- **Input:** 5, **Expected output:** 120
- **Input:** 0, **Expected output:** 1

Code:

```
def calcular_factorial():
    n = int(input(""))
    factorial = 0 # Error in initial state
    i = 0 # Error in initial state

    while i <= n:
        factorial *= i
        i += 1

    return factorial
print(calcular_factorial())
```

Initial condition: i = 0, factorial = 0
Final condition: i <= n
State transformation: factorial *= i, i += 1

b) Prime Number Verification.

Problem Description: Write a Python function to check if a number is prime using a "while" loop. Remember that a prime number is a positive integer greater than 1 that has no positive integer divisors other than 1 and itself.

Test cases:

- **Input:** 7, **Expected output:** True
- **Input:** 10, **Expected output:** False

Code:

```
def is_prime():
    n = int(input(""))
    if n < 2:
        return False
    else:
        i = 2
        while i < n // 2: # Error in loop condition
            if n % i == 0:
                return False
            i += 1
    return True

print(is_prime())
```

Initial condition: i = 2
End condition: i < n // 2
State transformation: if n % i == 0:, return False, i += 1

c) Checking for Even Numbers.

Problem description: Write a Python function that uses a "while" loop to check whether a number is even. The function should return True for even numbers and False for odd numbers.

Test cases:

- **Input:** 4, **Expected output:** True
- **Input:** 7, **Expected output:** False

Code:

```
def even():
    n = int(input(""))
    i = 0
    is_even = False

    while i < n:
        if n % 2 != 0: # Error in state transformation
            is_even = True
            break
        i += 1

    return is_even
print(even())
```

Initial condition: i = 0, is_even = False
End condition: i < n
State transformation: if n % 2 != 0:, is_even = True, break, i += 1

d) Checking Odd Numbers.

Problem Description: Write a Python function that uses a "while" loop to check whether a number is odd. The function should return True for odd numbers and False for even numbers.

Test cases:

- **Input:** 5, **Expected output:** True
- **Input:** 8, **Expected output:** False

Code:

```
def odd():
    n = int(input(""))
    i = 0
    is_odd = False

    while i <= n: # Error in loop condition
        if n % 2 == 0: # Error in state transformation
            is_odd = True
        break
        i += 1

    return is_odd

print(odd())
```

Initial condition: i = 0, is_odd = False
Ending condition: i <= n
State transformation: if n % 2 == 0:, is_odd = True, break, i += 1

e) Addition of Numbers from 1 to N.

Problem description: Write a Python function that calculates the sum of all integers from 1 to a given number N using a "while" loop.

Test cases:

- **Input:** 5, **Expected output:** 15
- **Input:** 10, **Expected output:** 55

Code:

```
def sum_num():
    n = int(input(""))
    sum = 1 # Error in initial state
    i = 1

    while i < n: # Error in loop condition
        sum += n # Error in state transformation
        i += 1

    return sum

print(sum_num())
```

Initial condition: sum = 1, i = 1
End condition: i < n
State transformation: sum += n, i += 1

6.2 Definition of the Synthetic Dataset Creation Process.

This study features a dataset comprising 7000 basic programming tasks involving "while" loops in Python. Each task is designed to address specific problems and includes a range of correct and incorrect solutions. The data was collected and generated through a series of structured steps.

6.2.1 Initial data.

We downloaded data in several public repositories containing problem statements and their corresponding solutions [147, 148, 149]; these repositories were filtered to retain only entries that mentioned "while" loop in the statement or contained `"while"` loops in the implementation.

6.2.2 Entering the description and correct solution of the programming task with a "while" loop.

The first step was to enter a detailed description of the programming task involving a "while" loop and a correct version of the solution in Python. We developed this solution to ensure it met the task's requirements and followed the theoretical guidelines for iterative programs according to the program correctness theory.

6.2.3 Generating correct versions.

To explore alternative solutions, we utilized GPT to generate correct variations of the programming task, specifically using "while" loops in Python. The question was, "Is there another solution to this problem using a `while` loop in Python?" We reviewed GPT's responses to ensure they met the problem requirements, correctly used the "while" loop, and adhered to the iterative structure defined by Gries. We sometimes requested new solutions when the answers did not meet these criteria or when other loops, such as `for` or `do-while`, were used.

6.2.4 Generating incorrect versions.

To generate incorrect versions of the programming task in Python, we prompted GPT with the request: "Provide an incorrect solution to this exercise using a `while` loop." The generated solutions were labeled as incorrect and subsequently classified based on the error types taxonomy outlined in Chapter 5. These incorrect solutions were categorized into three main error types:

- **Error in initial state:** The initial state does not match the one required for iteration scheduling.
- **Error in state transformation:** The formal definition of state transformation does not lead to the proper completion of the state.
- **Error in final state:** When the loop stops, the final state does not yet have the correct answer.

When GPT generated incorrect solutions that repeated the same types of errors, we took the original exercise and manually added specific errors. We then input these manually generated incorrect versions into GPT for feedback. In some cases, GPT correctly identified the errors, while in others, it failed to detect any logical or syntactical issues. In these instances, we provided the feedback manually.

6.2.5 Program specification definition.

Finally, we developed Python functions to extract the iteration's specification for both the correct and incorrect solutions within the dataset. This specification encompasses each iteration's initial state, final state, and state transformation. An iterative specification outlines the parameters essential for the functioning of a loop. This specification ensures that iterative programs align with established theoretical and practical expectations.

A specific Python function split and labeled each program into its different components:

- Variable initialization.
- Loop condition `while`.
- Loop body `while`.

This additional information was used as part of the model inputs during training and inference and has been shown to improve model performance in all cases.

6.3 Generation of the Synthetic Dataset.

6.3.1 Final Dataset: Key Characteristics and Relevant Information.

As previously mentioned, our final dataset focuses on fundamental programming tasks that involve "while" loops. For the final corpus, we selected exercises from repositories such as INGInious, CodeNet, and others that explicitly referenced "while" loops in their problem statements or included "while" loops in their implementations. This resulted in a selection of 80 fundamental programming problems utilizing "while" loops. The final corpus consists of 7,000 entries: 1,000 correct solutions and 6,000 incorrect solutions addressing the 80 programming problem statements.

The incorrect solutions were generated using a custom prompt in GPT-3.5 Turbo, with detailed configuration provided in the supplementary material. In some cases, both correct and incorrect solutions were manually created to ensure greater diversity. Each entry in the dataset was carefully reviewed and curated to maintain high quality and relevance.

Furthermore, each incorrect solution was labeled according to the error types defined by the taxonomy for programming iterations presented in Chapter 5, as described by [26]. These solutions were carefully reviewed and balanced to ensure consistency in the distribution of solutions exhibiting specific errors or combinations of errors in their implementation (see Table 6.1).

Additionally, to evaluate the classification model, we generated 21 additional incorrect solutions for exercises not included in the training dataset. This evaluation set comprises three solutions for each error type and their combinations (seven error types).

Classes	**Count**
`['Correct']`	1000
`['Initial state']`	1024
`['Final state']`	1016
`['State transformation']`	1203
`['Initial state', 'State transformation']`	692
`['Final state', 'State transformation']`	692
`['Initial state', 'Final state']`	692
`['Initial state', 'Final state', 'State transformation']`	681
Total	**7000**

Table 6.1: Distribution of labels and their counts in the dataset.

This dataset provides a rich source of information to analyze and classify different types of errors in programming with iterations, facilitating a detailed study of how these errors occur and are corrected in basic programming tasks. To preprocess later the data set and load it into a data frame, we worked with a CSV file consisting of nine columns. The structure of the CSV file is as follows:

- Column 1: Identification of the programming task with iterations.
- Column 2: Description of the problem to be solved.
- Column 3: Python solution provided for the programming task.
- Column 4: Definition of the initial state of the loop, where the initial variable declaration is located.
- Column 5: Definition of the final state of the loop, where the loop condition is defined.
- Column 6: Definition of the loop state transformation, also known as the loop body.
- Column 7: Task status. It is denoted as one if the solution is correct and 0 if it is incorrect.
- Column 8: The types of errors are defined in the taxonomy, which includes the following error categories: Initialization State, Final State, and State Transformation.

6.4 Chapter Summary.

This chapter describes the creation of a synthetic dataset for programming tasks with while loops, including detailed descriptions, loop conditions, test cases, and correct and incorrect solutions. Errors are classified according to a taxonomy (Chapter 5) based on program correctness theory [26]. The final dataset consists of 7000 tasks, extracted from repositories and generated with the support of GPT.

This dataset will be used to train a bug classification model using machine learning techniques, with the aim of generating automated feedback in introductory programming courses.

Part III

Automatic Feedback Model

Chapter 7

Automated Feedback Model for Iterative Programming Tasks.

Contents

In this chapter, we present the automated feedback model designed for iterative programming tasks. Section 7.1 defines the scope of the feedback model, detailing the objectives and limitations. Section 7.2 focuses on the selection of machine learning algorithms and evaluation metrics, covering aspects such as data preprocessing techniques, model architectures, and the training and optimization process. It also provides the results obtained using the defined metrics. Section 7.3 outlines the variables used to assess the type of feedback, explaining how error classification models integrate with feedback generation. Additionally, it describes the strategy for generating and evaluating feedback. Finally, section 7.4 summarizes the key points discussed throughout the chapter.

7.1 Defining the Scope of the Automated Feedback Model.

The automatic feedback model developed in this research assesses solutions to basic programming problems with iterations, following Gries' approach to loop programming [26]. It identifies common coding errors and provides targeted feedback to enrich student learning.

Designed for introductory programming courses that use Python, the model evaluates tasks built around the "while" loop structure. It identifies logical errors in three specific areas:

- Errors in the initial state.
- Errors in the final state.
- Errors in state transformation.

These errors often overlap, and an incorrect solution may exhibit multiple issues. To address this, we developed a model that leverages machine learning techniques to classify these errors. The model is trained on a carefully curated dataset designed to capture the unique patterns of each error type and their possible combinations. This approach enhances the model's ability to provide adequate feedback, helping students identify and correct mistakes.

Once we classify the error, the model uses the GPT API to generate detailed textual feedback. This feedback identifies the error and explains the problem according to pro-

gram correctness theory in iterative programming, helping students understand the computer programming concepts.

We have integrated this modeling environment into an LMS platform like INGInious, making it an invaluable resource for introductory programming courses. This integration offers meaningful support to first-semester students, enhancing their learning experience through automated feedback in programming tasks.

7.2 Error classification model in programming tasks with iterations.

7.2.1 Description of preprocessing techniques for data.

Embeddings are representations of complex data such as words, phrases, or images as fixed sized vectors, that encodes into continuous spaces the semantic of the data, making it easier for machine learning algorithms to process them as [150] [151] [152].

In this work, we focus on tokenizing and generating embeddings for problem statements, Python solutions, and specific segments of Python code from the dataset. To achieve this, we evaluated three pre-trained models: Google's BERT for English text, Google's CodeBERT specifically designed for natural language and source code, and OpenAI's Ada model.

For problem descriptions, we utilized a pre-trained BERT model specifically designed for English text, which generates 768-dimensional embeddings to effectively capture semantic information.

To tokenize and generate embeddings in Python source code, we employ the CodeBERT model, which has been specifically trained to understand and process natural language and source code. This model is well-suited for tasks like code generation, classification, labeling, and searching for similar code. The 768 dimensions of CodeBERT embeddings effectively capture the semantic context of natural language and the hierarchical structures and patterns of code, striking a good balance between representation quality and computational efficiency.

In addition, we tested, OpenAI's "text-embedding-ada-002", packaged by Xenova. This model is primarily designed to generate embeddings from natural language text. Although its main focus is text processing, some experiments have explored its application

in source code, given that code shares certain structural features with natural language.

The experiments confirmed that CodeBERT outperforms text-embedding-ada-002 for code related tasks, as it incorporates code structures and semantics into its encoding features that general purpose embeddings lack. These tests involved tokenizing and generating embeddings for the training dataset. Subsequently, the dataset was then used to train the model, and metrics such as precision, recall, accuracy, and F1 score were evaluated for each case to determine the model that achieved the best performance based on the preprocessing approach.

Integrating these embeddings improves the dataset's quality and symbolic depth, enabling more effective analysis, modeling, and downstream tasks. This approach supports building machine learning models and conducting natural language processing on problem descriptions, code solutions, and related data.

After defining the error encoding scheme, we first apply label encoding to convert error categories into unique integer labels. This process assigns each combination of errors a specific integer, as shown in Table 7.1.

Label	Description
0	Correct program (No errors)
1	Error in the initial state
2	Error in the end state
3	Error in state transformation
4	Errors in the initial state and end state
5	Errors in the initial state and state transformation
6	Errors in the end state and state transformation
7	Errors in the initial state, end state, and state transformation

Table 7.1: Label convention used for dataset classification.

7.2.2 Model experimentation for error classification in programming tasks with iterations.

We performed a series of experiments to evaluate the performance of basic machine learning algorithms such as the Random Forest Classifier, and the KNeighbors Classifier, implemented in the SkLearn library [153]. We selected these algorithms for their ease of

implementation and tuning, which is essential to obtain fast and practical results, allowing us to iterate and tune the training process with less complexity.

Each of the selected algorithms approaches classification from a different perspective:

- **Random Forest Classifier**: This ensemble model, built on decision trees, effectively resists overfitting and manages datasets with noisy or irrelevant features. It also provides interpretability through feature importance, allowing users to understand the impact of each feature on the model's decisions [19].

- **KNeighbors Classifier**: This model relies on the proximity of data points within the feature space. Its simplicity makes it practical for problems where similar data often share the same class. This local similarity-based approach excels at uncovering patterns within the data [17].

The selection of these algorithms allowed us to explore different classification approaches and provided a solid starting point for future optimizations and improvements.

Additionally, we included more complex models capable of detecting more subtle non-linear patterns and relationships, which is crucial for accurate error classification in programming tasks. Therefore, we used fully connected (FC) neural network models and convolutional neural networks (CNN) implemented with the Keras library.

- **The fully connected neural network**: was chosen for its unique ability to capture complex non-linear relationships between input features. This network, deployed using the Keras library, typically consists of an input layer, where each neuron represents a feature, one or more hidden layers, and an output layer. In our research, each neuron in the output layer represents a specific error class, enabling the model to efficiently classify the identified errors [154].

- **On the other hand, 1D convolutional networks (CNNs)**: also implemented with Keras, were selected for their ability to process sequential data and detect local patterns in one-dimensional series, such as sequences of tokens in source code. CNNs apply convolutional filters that extract relevant features from inputs, which is beneficial for identifying errors in specific code fragments. This property of CNNs makes them well-suited for error classification, where the sequential structure of code is critical to identifying error patterns[154].

In addition to assessing individual models, we explore the effectiveness of ensemble models. This technique integrates multiple models to enhance performance and generalization, capitalizing on the strengths of various approaches to achieve more accurate and robust predictions. In this research, we develop two ensemble models:

- The first ensemble model fuses a random forest model with a multilayer perceptron model using a soft voting method, where the predictions are averaged to make the final decision.
- The second ensemble model integrates a random forest model with a multilayer perceptron model as base estimators, using logistic regression as the final decision model.

The metrics used to evaluate model performance were:

- Accuracy.
- Precision.
- Recall.
- F1 score.

7.2.3 Training, architecture, and optimization process in the error classification model.

After tokenizing and generating embeddings, we split the data, 80% for training and 20% for testing. We evaluated the classification process using a variety of techniques, including both traditional machine learning algorithms and deep learning models, by exploring different hyperparameter configurations.

The table 7.2 presents the ranges of values tested for the hyperparameters across the different machine learning techniques used in this study. The hyperparameter calibration process in this study involved two distinct approaches. First, for some hyperparameters, we defined value ranges and manually tuned them through iterative experimentation. This method allowed us to identify the configurations that achieved the best performance metrics for each model. For example, learning rates, batch sizes, and the number of epochs were tested across various ranges to optimize model convergence and generalization.

Hyper parameter	Fully Connected	Convolutional	Random Forests	K Neighbors	Voting Classifier	Stacking Classifier
Optimizer	Adam, RMSProp	Adam, RMSProp	N/A	N/A	N/A	N/A
Loss Function	Sparse categorical crossentropy, Categorical Crossentropy	Sparse categorical crossentropy, Categorical Crossentropy	N/A	N/A	N/A	N/A
Learning Rate	$[0.00001, 0.001]$	$[0.00001, 0.001]$	N/A	N/A	N/A	N/A
Epochs	$[100, 200, 300]$	$[100, 200, 300]$	N/A	N/A	N/A	N/A
Batch Size	$[50, 100, 200]$	$[50, 100, 200]$	N/A	N/A	N/A	N/A
Number of Estimators	N/A	N/A	$[10, 200]$	N/A	N/A	N/A
Max Depth	N/A	N/A	$[5, 20]$	N/A	N/A	N/A
Number of Neighbors	N/A	N/A	N/A	$[3, 15]$	N/A	N/A
Voting Type	N/A	N/A	N/A	N/A	soft	N/A
Final Estimator	N/A	N/A	N/A	N/A	N/A	Logistic Regression
Estimators	N/A	N/A	N/A	N/A	ann, rf	ann, rf

Table 7.2: Ranges of values tested for hyperparameters in different techniques.

Second, other hyperparameters were configured based on the specific nature of the data and the requirements of the machine learning algorithms employed. For instance, sparse categorical crossentropy was chosen as the loss function given the categorical nature of the classification task, while Adam was selected as the optimizer due to its proven effectiveness in handling sparse gradients. Finally, the configuration of the neural network architectures and the hyperparameters used in the "fully connected" and convolutional neural network models are specified in Table 7.3.

The training and evaluation of these models were conducted on a Google Colab Pro instance, leveraging a T4 GPU for efficient computation. This combination of manual tuning and informed configuration ensured that the selected hyperparameters were well-suited to the dataset and tasks, ultimately enhancing the overall performance of the models.

Figure 7.1 shows the architecture of the Fully Connected neural network implemented using the Keras' Sequential API. We included Batch Normalization and Dropout to avoid over-fitting.

Hyperparameters	Fully Connected	Convolutional
Optimizer	Adam.	Adam.
Loss Function	Sparse categorical crossentropy.	Sparse categorical crossentropy.
Learning Rate	0.00001	0.00001
Epochs	200.	200.
Batch size	100.	100.

Table 7.3: Hyperparameters.

Figure 7.2 describe the arquitecture used for the Convolutional Neural Network. The main difference with the previous model is the use of Convolutional layers rather than Fully Connected layers for the first blocks of the network. Batch Normalization, Dropout and MaxPooling is used to avoid over-fitting.

7.2.4 Performance evaluation.

7.2.4.1 Model Performance Evaluation on Training and Test Data.

The dataset used for this research is cited in Chapter 6, which includes basic programming tasks with "while" loops.

Table 7.4 and Figure 7.3 summarize the performance of the six methods above when used to classify the proposed solutions for the test data.

Model	Loss	Accuracy	Precision	Recall	F1
RandomForest Classifier	0.61	85%	89%	87%	88%
KNeighbors Classifier	1.53	76%	82%	80%	81%
Ensemble Voting	0.53	85%	89%	87%	88%
Ensemble Stacking	0.42	87%	90%	89%	89%
FC Keras	0.42	88%	90%	90%	90%
Conv Keras	0.40	86%	89%	88%	88%

Table 7.4: Model performance on test data. Highlighted cells indicate the highest (green) and lowest (orange) values for each metric.

The Fully Connected Network delivered the best performance, achieving a 90% F1 score on the test set (see Figure 7.4). These results demonstrate that deep learning models excel in handling this complex task, although they demand more computational power

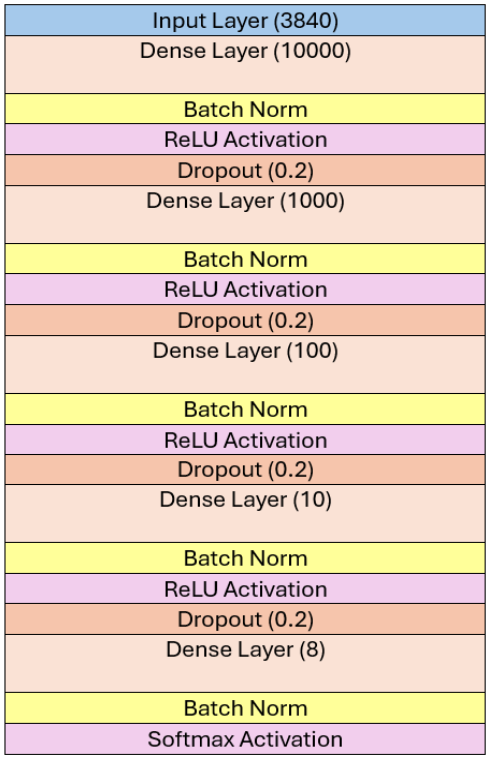

Figure 7.1: FC Keras model architecture.

and offer limited interpretability. In contrast, classic models also performed well and may be preferred when the goal is to extract insights from the model.

The model performs quite well, especially when multiple errors occur in an iterative programming task. The label "Initial state, Final state, State transformation" shows high performance with 95% precision, 96% recall, and 96% F1-score, indicating a solid ability to classify instances with this type of mistake.

In contrast, the "Correct" class, which identifies solutions without logical errors, achieves a performance of 87% across precision, recall, and F1-score.

The inclusion of the "Correct" category in the model serves to ensure that it can learn from and validate correct solutions for each exercise. The "Correct" class prediction is applied immediately after running the test cases. If the tool identifies a solution as correct based on the test cases, the model is used to confirm that there are no logical errors in the code. But, syntax errors unrelated to logic might arise. To address this, an additional verification is included using a large language model (LLM) to generate specific feed-

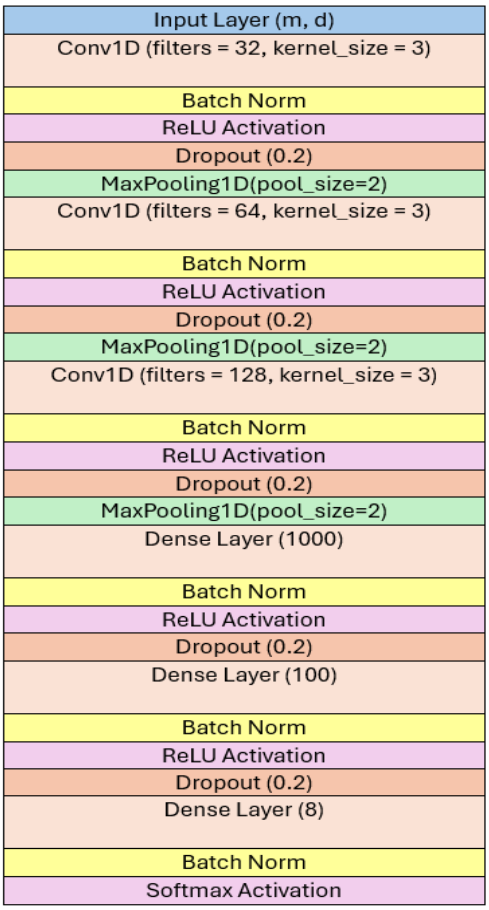

Figure 7.2: CV Keras model architecture.

back. This component complements the compiler's notifications by providing detailed comments on the syntax errors identified in the student's solution.

To provide a comprehensive analysis of the best model's performance, we include a scikit-learn confusion matrix, which compares predicted values with actual classifications (see Figure 7.5).

As we mentioned before, class 0 of correct solutions, called "Correct", presented a satisfactory classification of 176 cases in the test; however, notable challenges we observed when classifying correct tasks as if they gave errors in the initial state, final state, and transformation, of states. Similarly, class 3 ("State Transformation") presented satisfactory overall performance with 182 instances correctly classified. However, 32 notable errors were made in classifying class 3 instances as correct solutions and misclassifying class 3

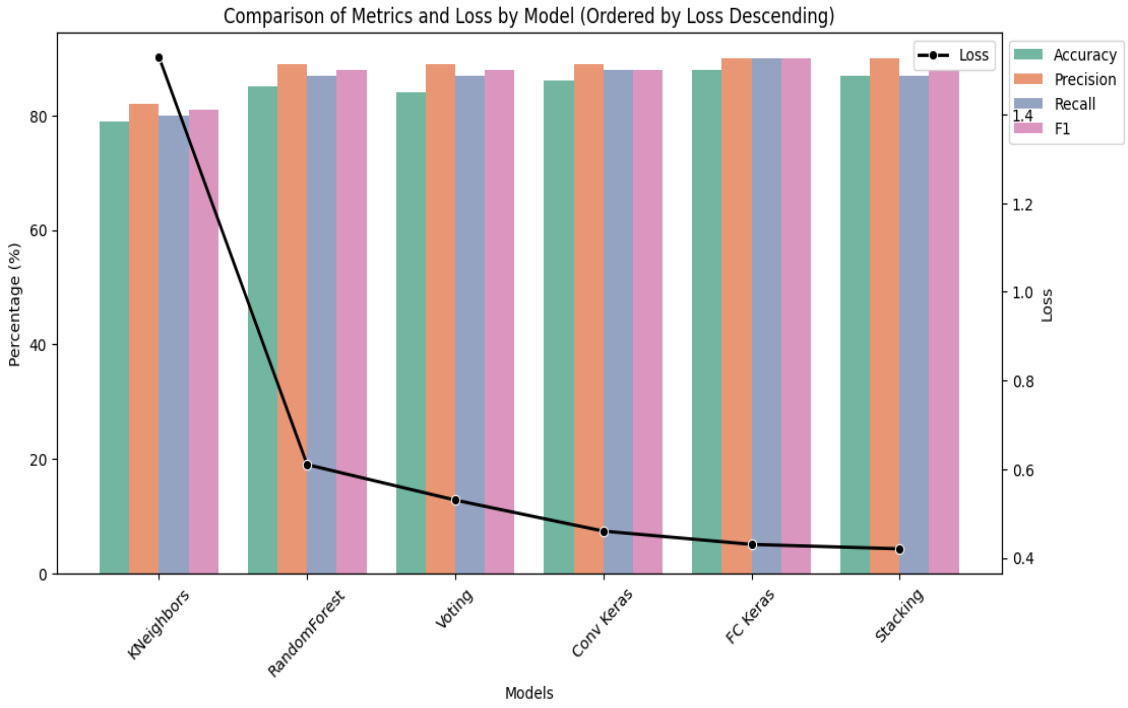

Figure 7.3: Comparison of Metrics and Loss by Model.

with errors in the initialization of the state or final state, respectively.

Moreover, as mentioned earlier, the model achieves better performance when predicting error combinations; one possible reason for this difference is that individual errors may be more ambiguous or less distinctive in the data. For example, a mistake in the initial state might share characteristics with an error in the final state. If the control variable i is initialized to 0 and it must iterate five times, it must end at i=4, fulfilling the condition i<=4. If I make a mistake in state initialization at i = 1 not 0, the model could predict that the error is in the loop condition since it would not be i<=4 but i<=5. Not in state initialization. As a result, distinguishing individual errors can be more challenging due to their overlapping features, which often blur the boundaries between categories. In contrast, error combinations typically exhibit more distinct and specific patterns, allowing the model to develop clearer and less ambiguous representations.

The confusion matrix reveals that 22+10+32+4+1 = 69 instances with errors were incorrectly classified as Correct, which can be attributed to several factors. One possibility is that some incorrect solutions closely resemble the correct ones, especially when the er-

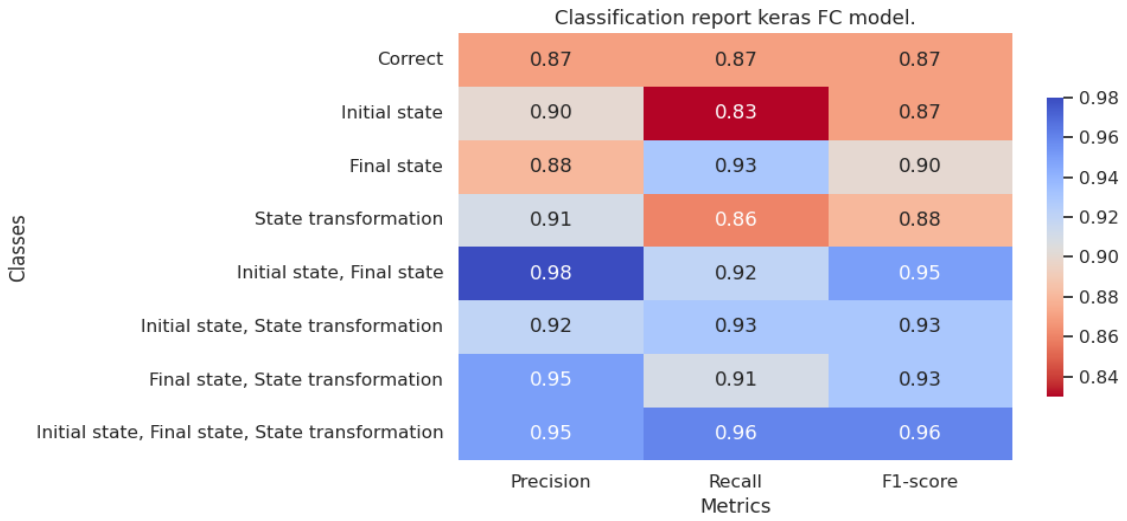

Figure 7.4: Classification report FC keras model.

rors are subtle and do not significantly affect the overall structure of the code. In such cases, the model may misinterpret an erroneous solution as correct due to their similarity in features.

Another contributing factor could be the model's inability to accurately capture the specific characteristics of certain types of errors, such as those in initial or final states, or in state transformations.

The confusion matrix also shows that hay 8+7+8+1+1+1+1 = 27 correct solutions were misclassified as incorrect, indicating a false positive issue. This could be due to noise in the training data, where some solutions labeled as correct might contain minor errors, causing the model to learn incorrect patterns. Furthermore, some correct solutions may have structural similarities to erroneous ones, making them difficult to distinguish.

In the confusion matrix, errors are observed according to different classes, based on which we conducted an analysis to relate the problem's difficulty with the model's incorrect predictions. Later, in Chapter 8, subsubsection 8.3.5.2, we discussed the classification of feedback based on a rubric. Additionally, we examined whether the ten proposed exercises used in the tests exhibit any correlation between difficulty and incorrect error classification.

In addition, Appendix A includes the training of the fully connected model with a dataset that excludes records categorized as 'Correct.' This approach was taken to train

the model solely on errors and their combinations as defined in the taxonomy.

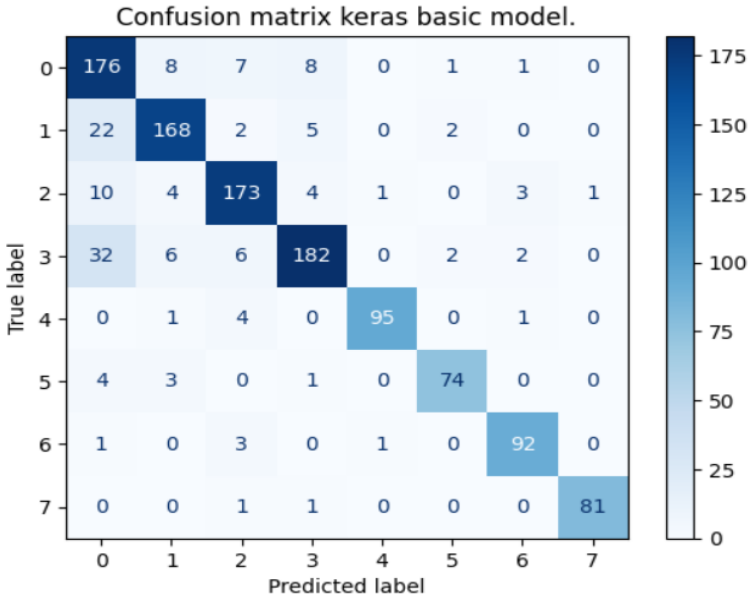

Figure 7.5: Confusion matrix FC keras model.

However, the model has limitations related to bias caused by the high representation of correct exercises in the dataset, which affects the classification of other categories. Additionally, it struggles to distinguish individual errors, such as "Initial State" and "Final State," due to overlapping features. It would be beneficial to explore regularization techniques or adjustments in the prediction thresholds to mitigate the incidence of false positives and negatives, thus improving the overall accuracy of the model. Finally, its scope is limited, as it only addresses errors in solutions using basic "while" loops, without considering more complex logical errors.

Subsequently, we ventured to experiment with ensemble neural network models. For the first set, a random forest model and a multilayer perceptron model using a soft voting method, thus achieving an accuracy rate of 84%. For the second set, we experimented with a random forest model and a multilayer perceptron model as estimators and Logistic Regression as the final prediction using a Stacking method; we achieved an accuracy rate of 87%. Furthermore, these assembly approaches have generated notable precision, recall, and F1 score metrics, suggesting that this strategy has the potential to improve overall performance.

7.2.4.2 Performance Evaluation on Non training Data.

A total of 21 Python solutions with errors were taken from three basic exercises that use while loops, which were not included in the training and testing process described in the previous subsection. These exercises were extracted from the Imperative Programming material from Universidad del Valle.

No.	Exercise Description	Expected Behavior/Output
1	Write a program that calculates the sum of the squares of all odd numbers less than 10. Use a `while` loop to iterate through the odd numbers, starting from 1. Finally, print the total sum of the squares.	The program should output the total sum, which is $1^2 + 3^2 + 5^2 + 7^2 + 9^2 = 165$.
2	Write a program that calculates the cumulative sum of positive integers starting from 1 to 4. Use a `while` loop to perform the sum and print the accumulated value at each iteration.	The program should output the accumulated sums: 1, 3, 6, 10.
3	Write a program that uses a `while` loop to iterate over the integers from 3 to 6. In each iteration, if the number is even, 2 should be added to the accumulator variable y, and if it is odd, 1 should be subtracted. At the end of the loop, the program must calculate and print the final value of y.	The program should output the final value of y, based on the operations: $y = -1 + 2 - 1 + 2 = 2$.

Table 7.5: Programming Exercises with `while` Loops (Model evaluation.)

These exercises were used to evaluate the model's ability to classify errors in programming tasks with while loops, specifically in contexts that were unknown or not previously included in its training. In general terms, correct and incorrect predictions were recorded, with variations in accuracy depending on the label analyzed. The model was correct in 9 cases, indicating that the prediction was correct. This result reflects the system's reasonable ability to identify certain types of errors.

In the incorrect cases, different degrees of deviation were observed. Some errors were classified as partial, suggesting that the model managed to correctly identify some aspects of the problem but was not able to cover all the elements necessary for a completely accurate prediction. One particular error, "1 error out of 2", was frequent, appearing 6 times, which stands out as a key area for improvement for the system. Furthermore, we can ob-

served that the model was unable to identify some or all errors in 6 solutions (See Figure 7.6).

Model performance varied across labels. Some, such as labels 6 ('Final state', 'State transformation') and 7 ('Initial state', 'Final state', 'State transformation'), showed consistent and correct performance, with several predictions classified correctly. In contrast, other labels, such as 1 ('Initial state', 'Final state') and 4 ('Initial state', 'Final state'), showed a higher incidence of errors, which could indicate that the system is having difficulty handling certain features associated with these categories.

A relevant pattern identified is the presence of "partially correct" errors, where the model recognizes some aspects of the problem but does not cover all those necessary for an accurate prediction. This indicates that while the system has a basic understanding of errors in while loops, it still requires fine-tuning to improve its ability to identify more complex combinations of errors and categories with a single error type.

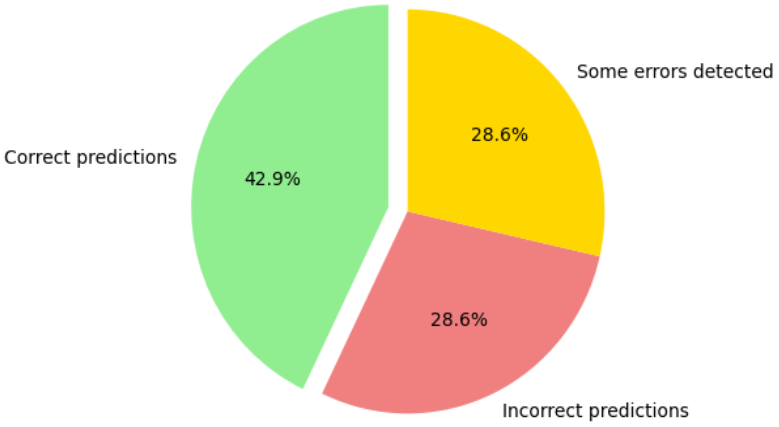

Figure 7.6: Error Detection in Unseen Data.

This analysis highlights the model's limitations in accurately predicting certain types of errors. The more challenging areas, such as partial errors and low-performing labels, require focused attention in future improvement efforts. Potential strategies include integrating a more diverse and representative training dataset and refining the algorithm to better manage complex error combinations and the unique features of individual error categories.

The fully connected network yielded the best results for classifying errors in program-

ming tasks involving "while" loops. Building on this result, the next step is to generate feedback for the identified errors. To do so, we use a dedicated model tailored to provide automated feedback. In the next section, we outline the key variables needed to create and evaluate the appropriate type of feedback, which will be further analyzed in the subsequent stages of the study. Additionally, in Chapter 9, we addressed several questions regarding the impact of feedback on students, including a dedicated subsubsection (9.3.3.1) that explores the effects of receiving feedback when the classification model makes an error.

7.3 Prompt engineering for automatic feedback generation.

7.3.1 Definition of variables for determining and assessing the type of feedback to provide.

This section focuses on identifying and explaining the key variables that determine the feedback we provide to students based on the errors detected in their programming solutions. These variables guide the design and adaptation of the feedback, ensuring it meets the students' level and needs.

Below, we describe the main variables influencing the feedback generation presented in this research (Figure 7.7):

1. Variables associated with the student's code.

   These variables are directly related to the student's solution in Python code. Their analysis allows for an accurate evaluation of the errors and the subsequent customization of the feedback.

   - **Type of error**: Our research relies on the taxonomy of errors at the task level, as described in previous chapters. Specifically, we focus on logical errors that the compiler or interpreter cannot automatically detect. We categorize these errors into three main types based on program correctness theory in iterative programming:
     - Errors in the initial state.
     - Errors in the final state.

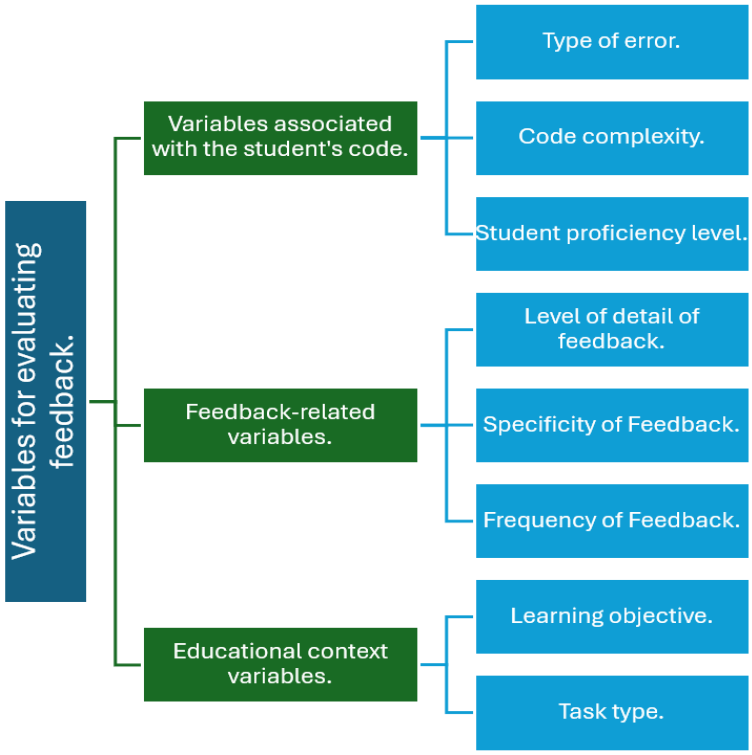

Figure 7.7: Variables for evaluating feedback.

  - Errors in state transformation.

  We defined these error types in Chapter 5. The errors often combine two or more categories, complicating diagnosis and feedback generation.

- **Code complexity**: Since we work with first-semester students, the solutions delivered are relatively simple, limited to a "while" loop. This makes it easy to identify common errors and design feedback focused on correcting basic errors and building a solid understanding of iterative structures.
- **Student proficiency level**: We consider students to be beginners in computer programming, so they can submit their solutions for evaluation as often as they wish. This feature requires feedback that corrects errors and motivates continuous learning.

2. Feedback-related variables.

These variables determine the characteristics and format of the feedback students receive, which they seek to be clear, specific, and valuable for their development.

- **Level of detail of feedback**: Feedback is tailored to the type of error detected by the error classification model. This approach ensures that feedback specifies the errors that need to be corrected.
- **Specificity of feedback**: Based on program correctness theory in iterative programming, feedback focuses on the general structure of essential programs and addresses errors affecting the "while" loops logic.
- **Frequency of feedback**: Feedback is immediate and is generated each time the student submits their code for evaluation. This feature allows for an iterative learning loop, where the student can continuously correct their errors and receive real-time feedback on their errors.

The following example shows a solution submitted by a student on the INGInious platform, using ImproBIt to generate feedback (See Figure 7.8). This solution corresponds to one of the exercises on prime numbers. The error classifier's prediction is accurate and detects an error in the initial state.

The feedback is appropriate to the error identified and complies with the variables defined for its evaluation. It details the type of error according to program correctness theory, described in the corresponding taxonomy. In addition, it includes comments on the error, indicating where it was identified, what steps the student must follow to correct it, and whether or not it affects the problem's invariant.

It is important to note that after receiving feedback, the student can revise their solution and request evaluation and feedback again as often as needed, enabling continuous improvement of their response.

3. Educational context variables.

   The learning process's context is crucial to adapting the feedback to the specific course objectives and assigned tasks.

   - **Learning objective**: The imperative programming course in which this study is framed includes the teaching of iterative structures, with a focus on "while"

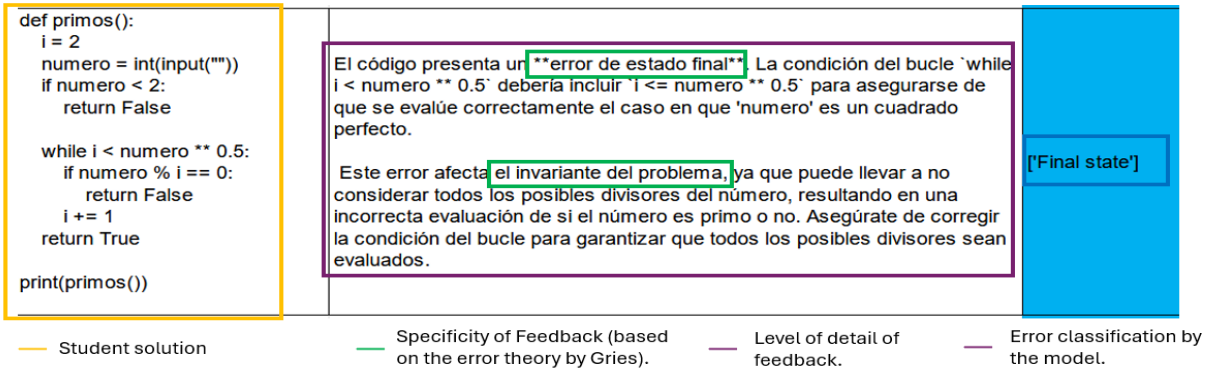

Figure 7.8: Example of a solution from a student with the accurate error prediction and adequate feedback.

loops. The feedback is specifically designed to support the learning of this concept, providing students with guidance on how to structure and debug "while" loops correctly.

- **Task type**: The assigned tasks are closed exercises that must comply with certain predetermined test cases. These test cases evaluate the correctness of the student's solution based on specific inputs and outputs.

7.3.2 Integration of Error Classification Models and Feedback Generation.

As previously discussed, defining a robust method for generating and evaluating feedback is essential to ensure it genuinely enhances student learning. With a model already developed to classify errors in programming tasks involving "while" loops, the next crucial step was to integrate targeted feedback for the errors identified by this model. This feedback needed to address the specific mistakes made, focusing on the types of errors categorized according to program correctness theory, as outlined in Chapter 5.

Given the rapid advancements in Extended Language Models (LLMs), we decided to leverage these technologies to generate feedback. Our initial work involved testing various models, including Gemini, LLaMA, and multiple versions of OpenAI's GPT. The goal was to identify the model that produced the clearest, most accurate, and pedagogically appropriate feedback for first-semester programming students.

We began with Gemini, LLaMA, and GPT-3.5. Gemini demonstrated potential in creating relevant feedback but lacked the precision to address the logical errors students often make with "while" loops. It provided general explanations but struggled to emphasize specific issues within the required theoretical framework. Similarly, LLaMA generated coherent responses but often failed to adhere strictly to prompt instructions, leading to less effective feedback for educational purposes.

Next, we experimented with OpenAI's GPT models, starting with GPT-3.5 and progressively testing newer variants. These models excelled in understanding programming errors and generating well-structured feedback aligned with the program correctness theory. After several iterations and comparative analyses, GPT-4.0 mini emerged as the optimal choice. It provided high-quality feedback and was more cost-effective than other models. To fully leverage GPT-4.0 mini's capabilities, we refined the prompt to be more concise and precise.

The prompt created for GPT-4.0 mini was a pivotal element of the system (See Figure 7.9. It was thoughtfully organized into three main parts:

1. We defined the model's role with a system message stating that it should act as a helpful assistant.

2. We provided detailed instructions for analyzing the student's code in the context of "while" loops, classifying errors based on program correctness theory. Errors were divided into three categories: initial state errors, final state errors, and state transformation errors. This categorization helped the model grasp and articulate the specific logical flaws in the code.

3. The model received essential inputs, including the problem statement, the student's solution, and our model's error classification, as described in Section 7.2.

With these inputs, GPT-4.0 mini generated precise feedback, ensuring it was relevant and actionable for the student.

The prompt also included instructions to deliver feedback in Spanish, given that our target student population is native Spanish speakers. We emphasized that the feedback should address only the identified errors without discussing unrelated potential issues. This approach prevented truncated responses and ensured the feedback was concise and practical. GPT-4.0 mini's response structure included specific feedback for each type of

error and, if necessary, reassessed the classification to ensure the accuracy of the feedback before delivering it.

```
# Construct the prompt
prompt_parts = [
    {
        'role': 'system',
        'content': 'You are a helpful assistant.'
    },
    {
        'role': 'assistant',
        'content': """Analyze the student's code with respect to "while" loops. The model classifies errors as (Gries's theory):

          - Initial State Error: Initialization issues.
          - Final State Error: Loop condition issues.
          - State Transformation Error: Issues within the loop body.

          Provide concise feedback in Spanish only for the errors and evaluate if these affect the problem's invariant.
          Ensure the response is not truncated; summarize if necessary."""
    },
    {
        'role': 'user',
        'content': f"Problem: {data['statement']}\nSolution: {data['code']}\nClassification: {data['classification_result']}"
    }
]
```

Figure 7.9: Custom prompt with error classification model, problem statement and student code.

This carefully crafted prompt allowed the system to automate code analysis and provide tailored feedback on programming tasks involving "while" loops. The prompt's structure was standardized across different LLMs, with model-specific adjustments as needed, making the system adaptable.

Our implementation was built in a Flask environment, with GPT-4.0 mini's responses stored in Firebase to maintain efficient records of interactions. Model settings, such as max_tokens for controlling response length and temperature for balancing creativity and precision, were fine-tuned to maximize performance. This streamlined design resulted in a highly effective feedback system, proving GPT-4.0 mini to be the superior model for our research objectives.

To provide feedback to students, we integrated a feature within the LMS that enables students to request feedback whenever their assignment fails the test cases. This feedback generation and visualization process will be discussed in detail in Chapter 8.

7.3.3 Strategy for Feedback assessment in prompt development.

Detailing the process of generating and evaluating feedback is crucial to our research. This process is vital to our overall strategy to ensure that feedback enriches student learning.

The previous subsection detailed how an exit prompt was designed to generate automatic feedback based on error classification for GPT, a crucial component in our process. This feedback considers the utterance, the solution provided by the student, and the model's error classification, which is delivered through an LMS.

After designing the prompt, we recognized the importance of testing its behavior to ensure the feedback provided aligns with the error types identified in programming tasks using "while" loops. Specifically, we wanted to verify that the feedback adhered to program correctness theory and matched the errors detected by the classification model. To achieve this, we conducted tests across three scenarios:

- Feedback generated using an error classification model with a customized prompt.
- Feedback generated with a customized prompt but without an error classification model.
- Feedback generated with a generic prompt.

For this evaluation, we selected an exercise, outlined as follows:

Calculate the factorial.

- **Problem statement:** Write a Python function that returns the factorial of a number using a "while" loop. Remember that the factorial of a number results from multiplying that number by all the numbers that exist from it to 1.
- **Proposed solution:**

```
def factorial(n):
    result = 0
    i = 1
    while i <= n:
        result *= i
        i += 1
    return result
print(factorial(5))
```

- **Automatic feedback results:**

  The comparative result of the Table 7.6, shows that the feedback generated in the first case, which integrates an error classifier and a detailed prompt, is the most effective. This approach provides precise and concise corrections, directly focused on the specific error identified, in this case, the initial state. In addition, it conforms to program correctness theory, providing the student with focused and valuable feedback regarding the theory of how to program interactions.

  In contrast, the second case, which omits the error classification, offers extensive but less precise feedback, which can disperse the student's focus. The third case, which employs a generic prompt, generates broad remarks that need more focus on the specific error, limiting them to a general view of the problem, not to the theory.

  The Appendix B section contains the prompt we used in all three cases. These tests were performed with other exercises, and we can conclude that designing a specific prompt that works in conjunction with the error classifier is the optimal approach. This method improves the accuracy of the feedback, guiding the student in a concise manner aligned with program correctness theory on iterations.

In addition, we included other examples of solutions submitted by students (see Figures 7.10 and 7.11). The first solution corresponds to an exercise on the factors of a number. In this case, ImproBIt incorrectly classified errors and generated inaccurate feedback. The correct error classification should be a 'Final state' error and a 'State transformation' error, meaning there are two errors in the solution.

The second solution corresponds to an exercise on the summation of numbers. ImproBIt also misclassified the error and generated incorrect feedback. The correct classification

Error Classification	Feedback with Classifier and Custom Prompt	Feedback with Custom Prompt but Without Classifier	Feedback Without Classifier or Custom Prompt

Initial State	The error in the initial state refers to the incorrect setup of variables or data. In this proposed solution, the initial value of "result" should be 1 instead of 0, as the product of a number and 0 will always be 0. This affects the problem invariant because the factorial result will be incorrect due to this error in the initial state.	`result`: Initial State Error: The error in this solution arises because the variable "result" is initialized with the incorrect value of 0 instead of 1. This affects the factorial calculation as the result should start at 1 and be multiplied by all numbers from the given number down to 1.
		State Transformation Error: No error of this type in the provided solution.
		Final State Error: No error of this type in the provided solution.
		These errors affect the problem invariant since the factorial calculation is incorrectly executed in both the initial state and the transformation.
			The issue with the proposed solution is that it does not correctly calculate the factorial of the number. Instead, it adds numbers from 0 to n. To calculate the factorial, you should multiply numbers from n down to 1, not add them. Here's an improved way to calculate the factorial using a "while" loop:
			
			`def factorial(n):`
			`result = 1`
			`i = n`
			`while i >= 1:`
			`result`

Table 7.6: Comparison of Feedback Scenarios.

should have been a 'State transformation' error, rather than an 'Initial state' error, as predicted by the model.

The second solution corresponds to an exercise on the summation of numbers. ImproBIt also misclassified the error and generated incorrect feedback. The correct classification should have been a 'State transformation' error, rather than an 'Initial state' error, as predicted by the model.

In both cases, ImproBIt failed to accurately predict the errors present in the solutions. However, GPT generated feedback based on the detected errors and, in some instances, delivered feedback that was almost accurate for specific solutions.

For example, consider the solution shown in Figure 7.12. In this case, ImproBIt misclassified the error, but GPT generated feedback based on the errors it detected in the solution, such as an error in the 'Initial state'.

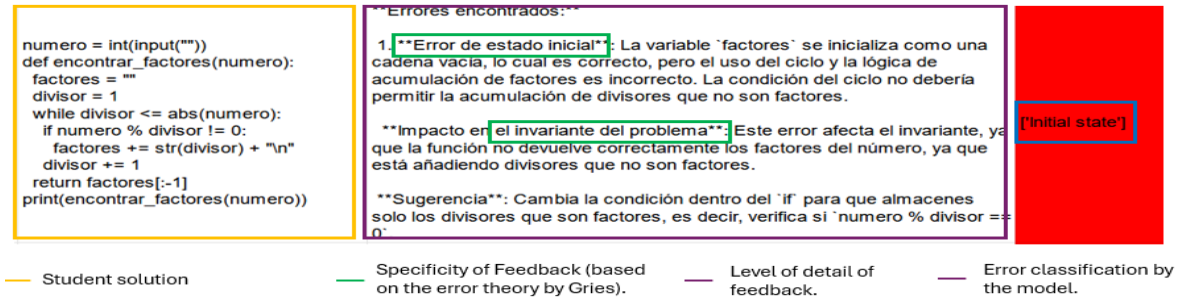

Figure 7.10: Example 1 of a solution from a student with the bad accurate error prediction and bad feedback.

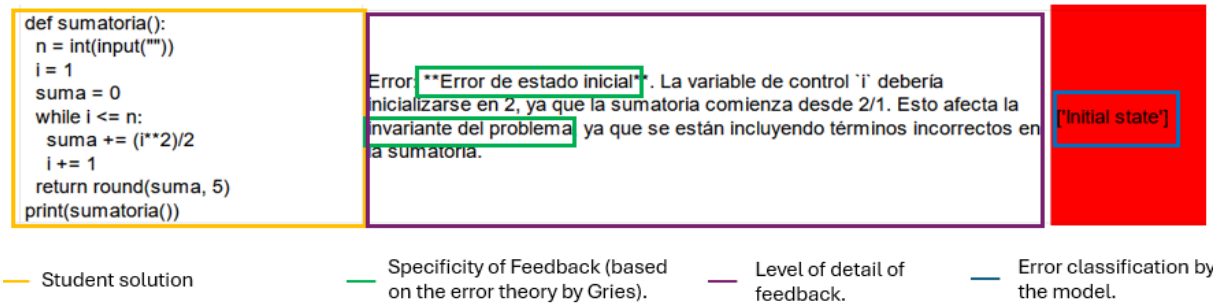

Figure 7.11: Example 2 of a solution from a student with the bad accurate error prediction and bad feedback.

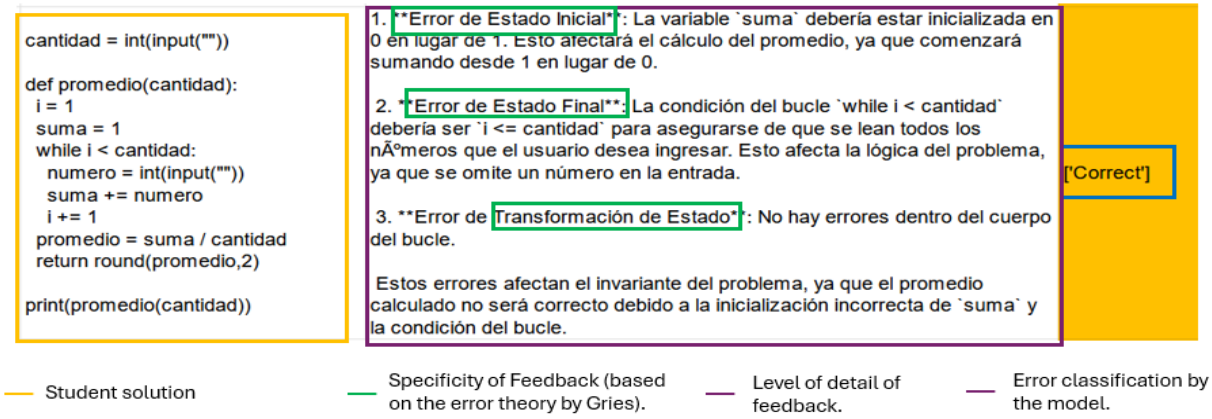

Figure 7.12: Example 3 of a solution from a student with the bad accurate error prediction and correct feedback.

7.4 Chapter Summary.

In this chapter, we evaluated the performance of various machine learning models and neural networks for classifying errors in programming tasks involving "while" loops. The dataset was processed through tokenization and embedding generation, and subsequently split into 80% for training and 20% for testing.

The results showed that the fully connected neural network achieved the best performance, with an F1 score of 90% on the test set, demonstrating its strong capacity to classify errors effectively. Deep learning models, in particular, outperformed classical models, excelling in the classification of complex error combinations, such as those involving the initial state, final state, and state transformation, with high precision and recall. However, challenges emerged when distinguishing between certain error types that shared overlapping characteristics, such as errors related to the initial and final states. Furthermore, 21 exercises outside the training and testing sets were introduced to evaluate the model's generalization to unseen data. While 28.6% of the errors were not predicted accurately, the model still provided partial or complete predictions for the remaining exercises.

To enhance the error classification process and achieve the research goals, we incorporated automatic feedback generation into the system. By utilizing GPT, the model not only generates feedback for predicted errors but also cross-checks them using program correctness theory of iteration programming, ensuring that any overlooked errors are flagged for further review. Despite these advancements, limitations remain, and further improvements are needed in error prediction accuracy. In the future work will focus on enhancing the model's ability to generalize to new exercises and refining the feedback process.

Chapter 8

Developing an API for INGInious to Enable Automated Feedback Generation.

Contents

In this chapter, we describe the development of an ImproB-It API for INGInious to enable automatic feedback generation based on the model presented in Chapter 7. Section 8.1 presents the design of the automated feedback API, including the resources, tools, and components required for feedback generation, accompanied by diagrams explaining its functionality. Section 8.2 details the API's construction, focusing on creating endpoints for code extraction, tokenization, embedding generation, error classification, and feedback requests. It also covers integrating these components with INGInious. Section 8.3 focuses on the API's functional testing, outlining testing scenarios, case designs, and input-output expectations. It includes setting up the testing environment and executing tests, followed by analyzing the results. Finally, section 8.4 provides a summary of the chapter's key content.

INGInious is an open-source learning management system (LMS) designed explicitly to evaluate real-time programming exercises. It supports various programming languages, making it ideal for automated assessments. We can leverage INGInious for this project because it provides a robust framework for automating code assessments, allowing us to focus on automated feedback generation rather than building an evaluation system from scratch.

Using INGInious requires configuring exercise environments, defining tasks, and specifying the criteria for evaluation. To enable automated feedback, we develop an API that integrates seamlessly with INGInious, adding custom endpoints for tasks like code extraction, tokenization, embedding, error classification according to Gries, and feedback generation. This setup ensures that students receive instant, targeted feedback on their coding exercises, supporting an interactive learning experience.

8.1 Design of the Automated Feedback API.

8.1.1 Definition of Resources and Tools for the API.

The development of the IMProB-It API takes place within a robust technical environment that includes hardware resources, software, development platforms, and learning management systems (LMS). Below are the main components that make up this environment:

1. Hardware Resources or Cloud Execution.

- Google Cloud Services: Google Cloud hosts the INGInious platform and the IMProB-It API, providing storage and computing capacity. It also facilitates the execution of AI scripts and models.
- Google Colab Pro: Used for the programming and training of machine learning models and the preprocessing of data related to the classification of errors in programming tasks with "while" loops.

2. Software Resources.

   a) Programming Languages.

      - Python: Main language used to develop the AI models, data preprocessing, and the IMProB-It API.

   b) Frameworks and Development Environments.

      - Flask: Web framework used to develop the API that provides the automatic feedback functionality.
      - Gunicorn: The WSGI server deploys the Flask application in production on the Google Cloud server, ensuring efficient execution in the production environment.
      - Visual Studio Code (VSCode): Open source IDE used for endpoint coding, source code extraction, tokenization, embedding generation, and error prediction in programming tasks with loops.

   c) Databases.

      - MongoDB is a structured database that stores information related to INGInious, such as users, courses, assignments, and student submission statistics.
      - Firebase: A non-relational database stores the feedback generated by IMProB-It, the problem statement, the student's solution, and the AI model's error classification.

   d) Data and Processing.

      - Dataset: The dataset created for this project consists of programming exercises involving "while" loops, created in Chapter 6 based on the error taxonomy defined in Chapter 5. It was curated from public repositories and further supplemented with solutions manually created by the researchers,

as well as additional solutions generated and validated with the help of GPT.

- Data Preprocessing: Preprocessing includes data cleaning, manual error classification, tokenization of statements and solutions, and generation of embeddings. Embedding generation is also part of the preprocessing performed with BERT and CodeBERT.

8.1.2 Explanation of Component Contributions to Feedback Generation.

The architecture diagram of the IMProB-It API (see Figure 8.1) provides a visual representation of the system's components, detailing the interactions between them and outlining the inputs and outputs for each element.

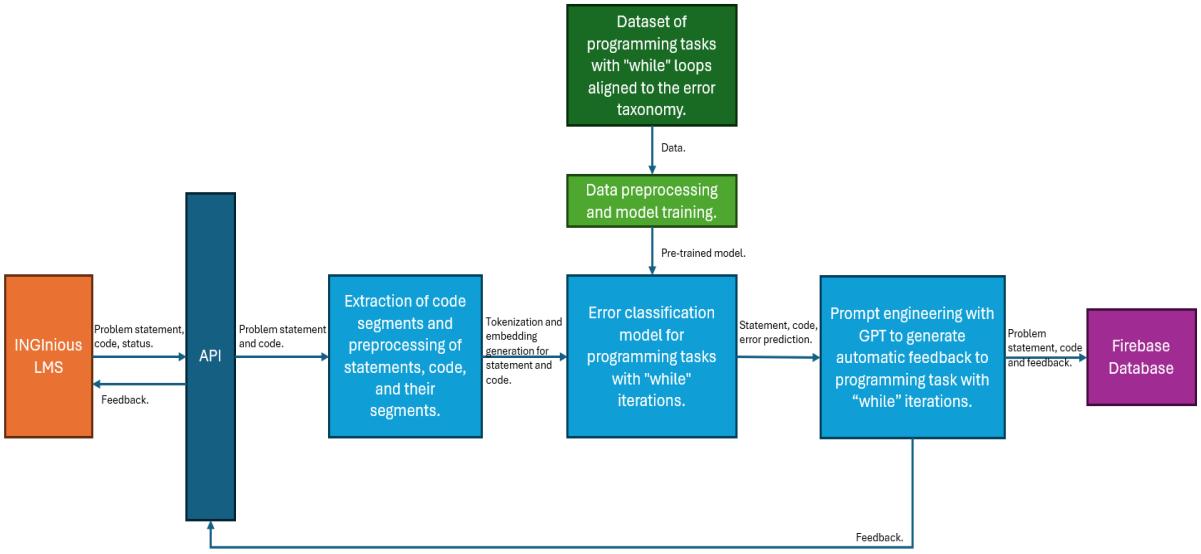

Figure 8.1: ImproB-It architecture diagram.

8.1.3 Diagrams and Functionality of the Automated Feedback API.

We have made a C4 diagram to explain the ImproB-It API more fully (See Figure 8.2). This diagram allow us to identify how communication flows between the API's different

elements and INGInious.

As previously mentioned, INGInious is an LMS platform equipped with an automatic source code evaluator. It is specifically designed to assess programming tasks in an educational context.

INGInious communicates seamlessly with the API through a custom function we developed. Students submit their Python code solutions for evaluation by pressing a designated button. This button triggers a request to the API, sending programming tasks involving loops that have encountered errors in their summative evaluation (Autograder).

The Autograder evaluates submissions based on two possible outcomes:

- "Success": This status corresponds to programming tasks that have not presented syntax errors and have passed all the established test cases.
- "Failed" - This status corresponds to programming tasks that failed test cases or have syntax errors.

Based on these states, we focus specifically on the "Failed" state, as it allows us to identify any errors present in programming tasks involving loops. This research has identified eight types of errors, as listed in Table 7.1.

The "Correct" class is not included because it is not an error but one of our classes.

The Autograder generates a JSON file containing the submission ID, the student's solution, and the submission status, indicating whether it failed or not. This JSON file is sent through the API to the first service, which handles data preprocessing. During this process, essential components of the source code are extracted, as defined in Chapter 6 regarding dataset creation. These components include the initial state, the final state (determined by the loop's stopping condition), and the state transformation (the loop's body). Extracting these specific parts of the code is crucial for generalizing the classification model. Then, the information is packaged into a JSON file for further analysis.

In the subsequent step, the problem statement is tokenized, and embeddings are generated for the statement, the student's Python solution, and the extracted code components. To achieve this, a pre-trained BERT model is employed for the problem statement, while a pre-trained CodeBERT model is used for the source code and its components. The output of this stage is a set of embeddings that serve as inputs for further processing. These preprocessing steps are clearly defined in Chapter 7.

Once the embeddings are prepared, they are sent to the second service of the API, which is responsible for error classification in programming tasks involving loops. This

classification process adheres to the taxonomy defined in the Chapter 5, specifically for programming errors and builds on the theory of how to program iterations [26]. Using the pre-trained error classification model described in Chapter 7, the service processes the inputs to classify errors in the student's solution. The problem statement and Python code are also utilized in this step to generate tailored prompts for later stages.

The third service leverages OpenAI's GPT-4.0 model, configured with the "OPENAI_API_KEY." This pre-trained language model is specifically adapted to generate automatic feedback. A detailed and customized message is crafted to meet the unique requirements of providing feedback for programming tasks involving loops. This feedback is generated based on the classification results and is further refined to ensure it addresses the specific errors identified, as described in Chapter 7.

Finally, the generated feedback is sent via the API to the INGInious Autograder, making it available to students. Simultaneously, the feedback, along with other relevant data, is stored in a database to support future review and analysis, ensuring that the system continuously improves and adapts to meet educational needs.

8.2 Construction of the Automated Feedback API.

8.2.1 Development of API endpoints.

This subsection describes three endpoints designed to extract, analyze, tokenize, generate code embeddings, classify errors in programming tasks, and generate automatic feedback. The construction of each is detailed below:

1. Code Extraction Endpoint.

   As mentioned earlier, in Chapter 5, we defined the error taxonomy for programming tasks involving "while" loops to specify the errors that the classification model will address. This taxonomy also guided the creation of the dataset that will be used to train the model for error prediction. Also, in Chapter 6, the rules for dataset creation are outlined, including the extraction of key code components from Python solutions. Thus, this final endpoint extracts and analyzes essential elements of the provided code, such as the initial state, the loop conditions, and the state transformations within the loop. It utilizes regular expressions (regex) and the AST module

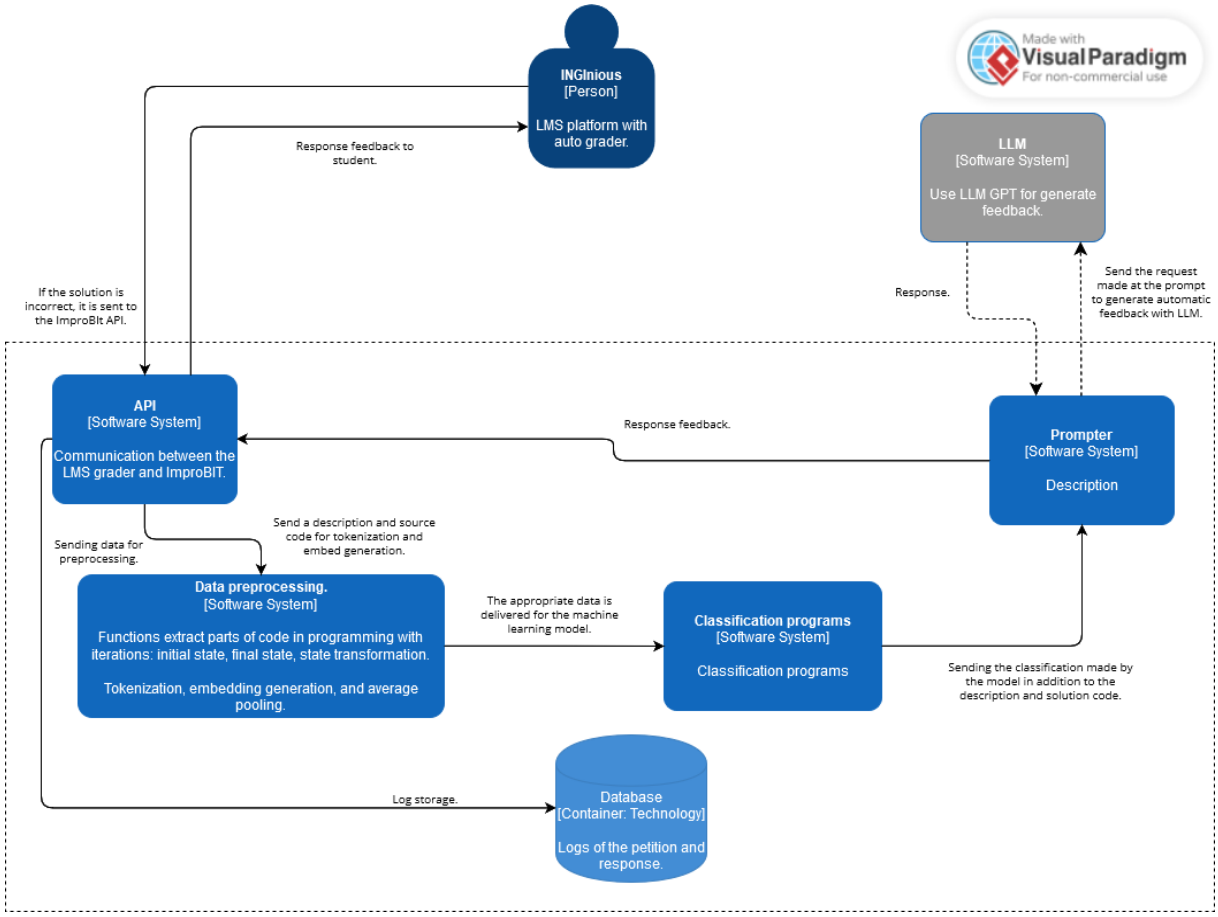

Figure 8.2: ImproB-It C4 diagram.

to evaluate the code's structure. The implementation runs on a server that processes POST requests containing both the code and the problem description.

This endpoint's goal is to centralize the code's structural analysis, preparing it for subsequent stages such as tokenization, embedding generation, and error classification.

2. Tokenization, Embedding Generation, and Error Classification Endpoint.

   This second endpoint focuses on tokenization and embedding generation of source code and problem descriptions, leveraging two language models: BERT for natural language descriptions and CodeBERT for source code. The goal of this endpoint is to capture the essence of the problem statement and key elements of the code, such

as variable initialization, the final state, and state transformations. After processing this information, the error classification model is trained to predict errors when a student submits their task for evaluation. This entire process is described in detail in Chapter 7.

This step plays a critical role in transforming the code and descriptions into appropriate mathematical representations, enabling the proper training of the machine learning model and the subsequent classification of new exercises.

3. Feedback Prompting Endpoint.

   The third endpoint generates automatic feedback based on the results of the code classification. It constructs and sends a prompt to the GPT-4 mini model to provide targeted feedback on the errors identified in the student's code. The feedback specifically addresses elements such as the initial state, final state, and state transformation, in line with program correctness theory, as outlined in the error taxonomy in Chapter 5.

   The prompt includes the problem description, the student's code, and the identified error before being sent to the GPT API. The feedback generated, along with the error classification and the original code, is then stored in a database for future analysis. This entire process is described in detail in Chapter 7.

   This endpoint is essential for delivering accurate, automated feedback based on a thorough analysis of the student's code.

8.2.2 Test of API endpoints.

Unit testing performed on the code extraction, tokenization, embedding generation, classification, and feedback generation endpoints was essential to ensure the accuracy and reliability of each feature. Below are some of the test cases implemented and the environment setup in which the tests were conducted:

- **Test case 1:**
  - Endpoint: Code Extraction.
  - Description: Verify the correct extraction of variable initialization, "while" loop conditions, and state transformation.

  - Input: Python code containing a simple "while" loop.
  - Expected result: Proper extraction of variable initialization, "while" loop conditions, and state transformation.
  - Actual result: The elements were accurately extracted, confirming the functionality of the code analysis.

- **Test case 2:**

  - Endpoint: Tokenization and Embedding Generation.
  - Description: Ensure proper tokenization of the code and problem descriptions using BERT and CodeBERT, and validate the generation of accurate embeddings.
  - Input: Problem description in natural language, Python code, and extracted code components.
  - Expected result: Successful generation of representative embeddings for both the problem statement and source code.
  - Actual result: Embeddings were generated accurately, providing coherent and suitable values for input into the classification model.

- **Test case 3:**

  - Endpoint: Error Classification After Tokenization and Embedding Generation.
  - **Description:** Evaluate the system's capability to accurately classify errors using embeddings generated from the problem statement and Python code.
  - **Input:** Embeddings derived from the problem statement and Python code.
  - **Expected Output:** Accurate classification of the identified error(s).
  - **Actual Output:** The system accurately classified one or more errors, highlighting its ability to effectively utilize embeddings and the pre-trained model for error classification.

- **Test case 4:**

  - Endpoint: Feedback Generation.

- Description: Ensure the prompt is constructed correctly for generating feedback based on classified errors.
- Input: Code containing an error in a "while" loop and its error classification (e.g., an issue with the loop condition).
- Expected result: A well-structured prompt that describes the problem, includes the code, and highlights the identified error.
- Actual result: The prompt was constructed successfully and sent to GPT, which generated accurate feedback.

We developed an API for these endpoints and deployed it on a Google Cloud server running Debian 11. We set up a virtual environment on this server and installed all necessary dependencies, including libraries like Flask, transformers, and pytest-cov. Although we used Flask during development, we configured the more robust Gunicorn server for production, efficiently handling multiple simultaneous requests.

We set up the required ports for each endpoint and exposed the server through port 80 by configuring DNS and port forwarding, ensuring seamless web access. We conducted testing directly on the production server, including the unit tests mentioned earlier, to confirm that all features operated correctly in a real-world environment.

Using pytest and pytest-cov, we achieved over 90% code coverage, reflecting the thoroughness of our functionality validation. We also tested edge cases, such as incomplete or complex code structures, to ensure the system's robustness.

Bug tracking tools and rapid continuous integration loops were essential in our agile development process, enabling early detection and swift resolution of issues. This approach significantly impacted the system's continuous improvement and successful deployment.

8.2.3 Integration of INGInious with Feedback API Endpoints.

IMProB-It is an automated feedback system that uses an external API and follows the RAP (Request-Answer-Process) methodology. The system seamlessly integrates with INGInious through a script that intercepts responses from submissions failing automated tests. These responses are sent to IMProB-It API endpoints, where the information is processed, errors in the student code are classified, and personalized feedback is generated.

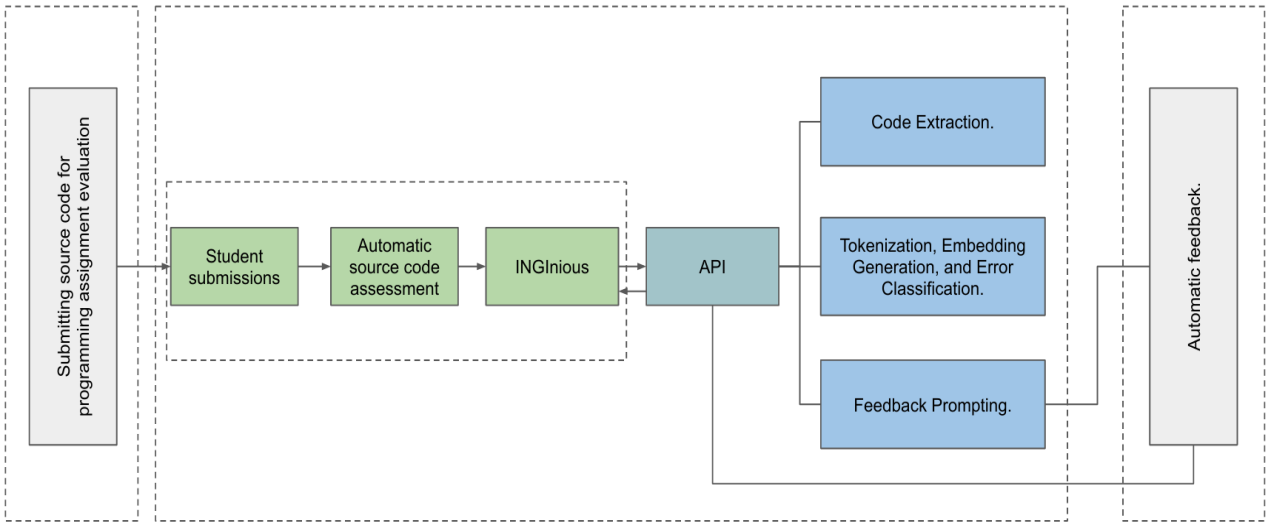

Figure 8.3: Functioning of the External Feedback API (IMProB-It) in INGInious.

The feedback is then sent back to INGInious, providing students with specific guidance on the detected errors and suggestions for improvement.

INGInious manages courses, tracks response attempts, performs automatic summative assessments, assigns grades, and monitors the status of student-submitted activities. This platform offers numerous advantages for learning computer programming, including the flexibility to incorporate new functionalities like automatic feedback, enhancing the overall educational experience (see Figure 8.3).

8.3 Functional Testing of the Automated Feedback API.

We conducted functional testing of the ImproB-It automatic feedback API integrated into the INGInious LMS for programming tasks with iterations through a structured process. This process involved identifying test scenarios, designing test cases, and creating comprehensive test cases to ensure accurate functionality.

8.3.1 Description of Test Scenarios for the API.

Our approach focuses on INGInious LMS integrated with the ImproB-It automatic feedback API. This API is fundamental in several aspects, including extraction of code segments from programming tasks involving "while" loops, tokenization, embedding generation, error prediction through a classification model, and subsequent automatic feedback generation using the GPT LLM model, as mentioned above.

Programming tasks, specifically those involving "while" loop iterations are basic. They adhere to the principles outlined in program correctness theory of iteration programming methodologies. These tasks serve as an ideal basis for evaluating the functionality of the ImproB-It API.

8.3.2 Design of Test Cases Covering All Identified Scenarios.

Before designing the test scenarios, we need to define the specific steps a student user follow when interacting with the API to give or receive feedback. The steps are as follows:

1. Enter the INGInious Univalle link created for this experiment (`http://inginious.gramcybersec.com/`).
2. Verify the assigned username and password.
3. Enroll in the "Imperative Programming (Python)" course.
4. Review the tasks established for its solution.
5. Each task has a statement, specification and requirements, input and output cases, and a workspace to write the code.
6. Once the code is written, it is sent for evaluation.

When the code contains errors, the input and output validation toggle appear, along with the option to generate feedback. Students can select the number of words they want for the feedback and must wait for it to display.

We have established sixteen basic programming tasks. The first six tasks help students familiarize themselves with the LMS and understand how to submit their code for evaluation, along with the submission requirements. The remaining ten tasks focus on "while" loop iterations and tested with the experimental group.

From the identified scenario, we created a list of test scenarios that encompass a wide range of situations. This includes various environments and conditions, such as different error types, task in Python programming language, tasks from the previously pre-trained dataset, and new programming tasks not included in that dataset.

The test scenarios serve to:

1. Test the extraction of code segments from programming tasks with "while" loops.

2. Validate the tokenization and embedding generation process.

3. Evaluate the error prediction using the classification model.

4. Evaluate the relevance of the automatic feedback generated by the GPT LLMs model, which was generated according to the identified errors. Verify the API's capability according to the complexity of the tasks.

8.3.3 Creation of Test Cases in INGInious.

After establishing the test scenarios and their design, we created the actual test scenarios. Each scenario requires meticulous design, detailing the specific data, expected results, and steps to replicate the testing process.

We organized the testing into two phases. The first phase includes six programming tasks that involve iterations, aiming to help students become familiar with the INGInious LMS.

In the second phase, we developed ten additional programming tasks for researchers to test. The feedback verification process is essential, as this process enables us to evaluate the feedback provided to students and fosters the continuous improvement of the tool.

We showed how we define and configure the test cases for ten exercises in INGInious. In the accompanying Table 8.1, we outline the instances defined for the programming task of calculating the factorial. This table includes the problem statement, specifications, and requirements necessary for correct coding and validation by the autograder, along with the input and output cases for each scenario. We ensured that multiple input and output cases were considered based on each exercise's unique nature.

To view all the test cases, refer to Appendix C.

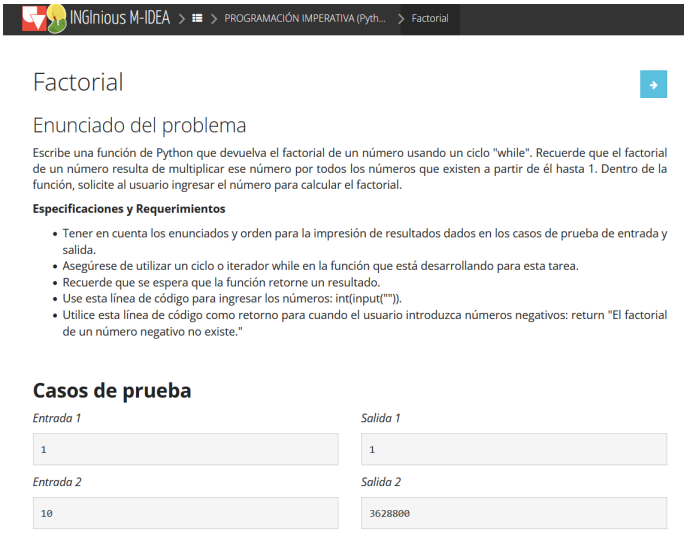

Figure 8.4: INGInious - Problem statement and specifications and requirements.

Figure 8.4 shows the Factorial calculation exercise on the platform, presented in Spanish. The exercise includes the statement, specifications, and requirements, followed by the test cases in the Table above 8.1.

Students can view two examples test cases; the system evaluates the remaining cases when they submit their code. If the code contains errors, a section displaying the failed test cases appears when they click the "Toggle diff" button, as shown in Figure 8.5. This feature highlights the differences between the calculated and actual inputs and outputs.

Additionally, students can request automatic feedback from IMProB-It to better understand the errors in their code, as demonstrated in Figure 8.6.

The student selects the desired word count for the feedback and presses "Continue." The system then provides feedback in Spanish, tailored to the errors made and based on the predictions from the error classification model.

8.3.4 Testing Environment Configuration, Test Data Preparation, and INGInious Setup.

The test scenarios we created correspond to basic programming tasks that utilize "while" loops. We configured these exercises on the INGInious platform, hosted on a Google

Description Task	Specifications and Requirements	Inputs	Outputs
Write a Python function that returns the factorial of a number using a "while" loop.	- Follow the order and format for input-output as given in the test cases. - Use a "while" loop for this task. - The function should return a result. - Use this code for input: `int(input(""))`. - For negative numbers, return "The factorial of a negative number doesn't exist."	Test case when the number is equal to 1:	1
		Test case with numbers yielding large results: 10	3628800
		Test case when the number is 0: 0	1
		Test case with small known numbers: 4	24
		Test case for negative numbers: -5	The factorial of a negative number doesn't exist.

Table 8.1: Test cases for scenario "Calculate the Factorial".

Cloud server. This infrastructure allows us to access platform settings through the teacher profile, where we can manage the creation and assignment of courses. In this case, we set up the Imperative Programming course at the Universidad del Valle, creating 10 test scenarios that include statements, instructions, and specific requirements (see Figure 8.7).

Each task includes two visible test cases for students, allowing partial validation of their solutions before final submission. Additionally, we configured several "hidden" test cases that rigorously evaluate the performance of the submitted solutions. These hidden cases cover different predefined input and output scenarios, ensuring the correct implementation of "while" loops in each task (see Table 8.1).

We also specify the required programming language, which in this case is Python. We accept solutions developed in Python versions from 3.5 to the latest releases, so students must ensure that their programs are compatible with these versions.

On the other hand, IMProB-It acts as an external API also deployed on a Google Cloud server. We designed IMProB-It to receive and process requests made by students from INGInious, offering automatic feedback based on errors detected in their solutions (see Figure 8.6). The API remains active and ready to respond to requests from INGInious.

Before conducting tests in a natural environment, we verified that both INGInious and IMProB-It operate correctly. This verification included checking the redirection to the servers through the configured domain and validating access for administrators, teachers,

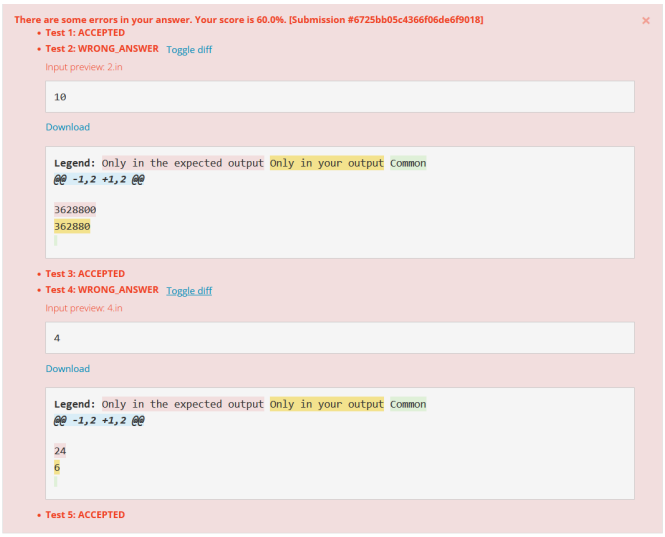

Figure 8.5: INGInious - Solution with errors Cases Test.

and students. We also confirmed that the connections between INGInious and IMProB-It work seamlessly, ensuring that responses and feedback are generated and delivered correctly to students.

8.3.5 Execution of Tests and Analysis of Results.

8.3.5.1 Feedback assessment in prompt testing.

This criterion was color-coded to improve interpretation based on its technical importance and impact on system performance, as shown in Table 8.2. Each color corresponds to a specific level of accuracy and quality of the classification and feedback:

- Red (Poor, 1): Neither classification nor feedback is correct. The model fails to identify any errors in the code or provide meaningful feedback.
- Yellow (Acceptable, 2): Misclassification of the model and generic feedback. The model may misclassify some errors, and the feedback provided is not sufficiently tailored to the identified issues.
- Orange (Good, 3): The classifier has identified at least one error, and/or the feedback is consistent with the error type. This indicates that the model is providing more relevant feedback, though it may not be fully comprehensive.

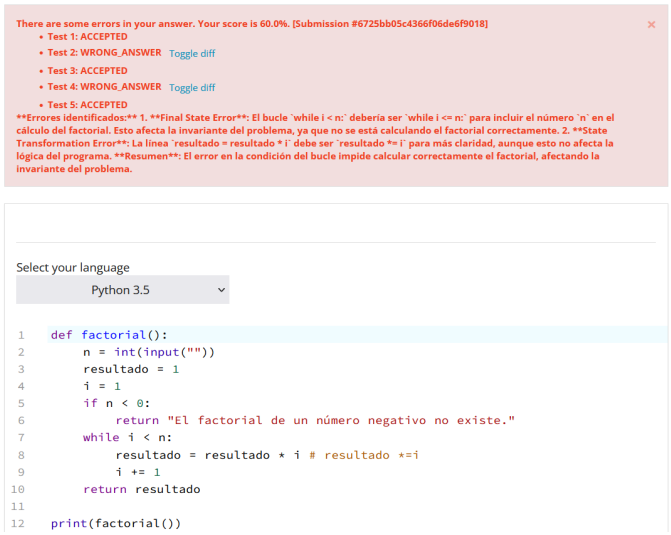

Figure 8.6: INGInious - Solution with errors.

Criteria	Poor (1)	Acceptable (2)	Good (3)	Excellent (4)
Model classification	Neither classification nor feedback is correct.	Misclassification of the model and generic feedback.	The classifier has identified at least one error, and the feedback is consistent.	Correct model classification and adequate feedback.

Table 8.2: Integrated rubric for evaluation of feedback and model classification.

- Blue (Excellent, 4): Correct model classification and/or adequate feedback. The feedback is highly accurate and aligns with the classification model, providing precise and actionable guidance on the errors in the code.

To better understand the classification process, we will examine four representative examples, each corresponding to a specific case and color. In the first example (Blue), a student's solution to a factorial calculation exercise is analyzed. An "Initial State" error is present and is correctly identified by the error classification model. The feedback generated by the system aligns appropriately with the identified error (See Figure 8.8).

In the second example (Orange), the solution refers to an exercise that calculates the sum of the cubes from 1 to n. This solution contains two errors: one related to the "Initial

Figure 8.7: INGInious Course Imperative programming.

State" and another to the "State Transformation." However, the error classification model only identified the "Initial State" error, and the feedback generated was limited to addressing this single issue. GPT also failed to detect the second error (See Figure 8.9).

The third example (Yellow) evaluates a student's solution to an exercise that determines prime numbers. In this case, a "State Transformation" error is present. However, the error classification model incorrectly classified the solution as correct. In contrast, GPT generated feedback for three types of errors, including the "State Transformation" error, which is the only actual issue in the solution (See Figure 8.10).

Finally, the fourth example (Red) assesses a student's solution to an exercise converting an octal number to a decimal. This solution includes an "Initial State" error, but the error classification model mistakenly identified it as a "Final State" error, which does not reflect the actual problem. The feedback generated by GPT was based solely on the error identified by the classification model (See Figure 8.11).

This color-coded classification highlights the strengths and limitations of the error classification model and the feedback it generates, providing valuable insights into its performance.

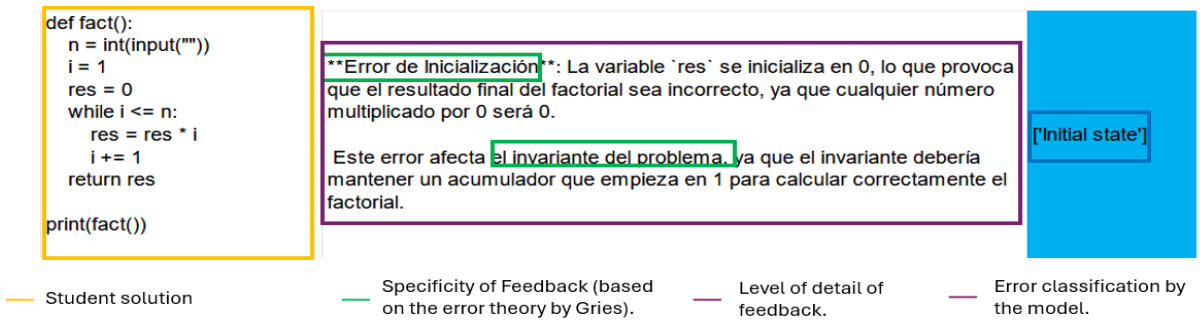

Figure 8.8: An example of a solution from a student with the classification color 'Blue'.

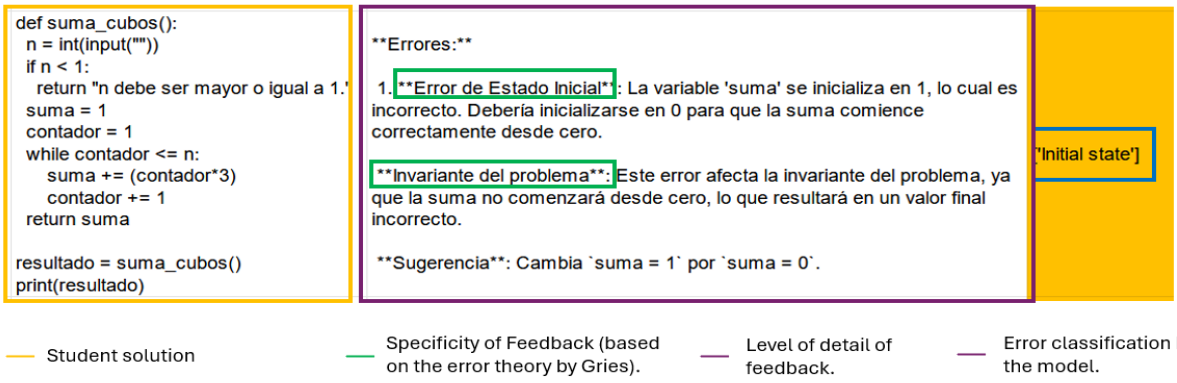

Figure 8.9: An example of a solution from a student with the classification color 'Orange'.

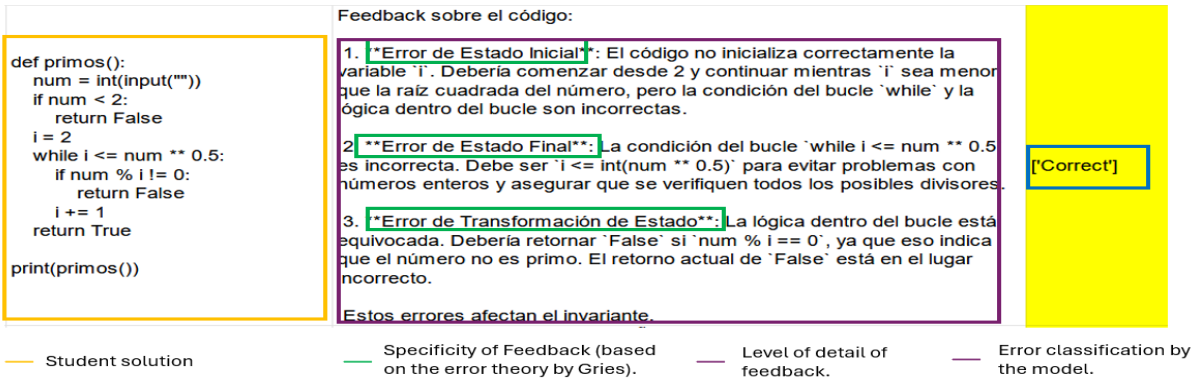

Figure 8.10: An example of a solution from a student with the classification color 'Yellow'.

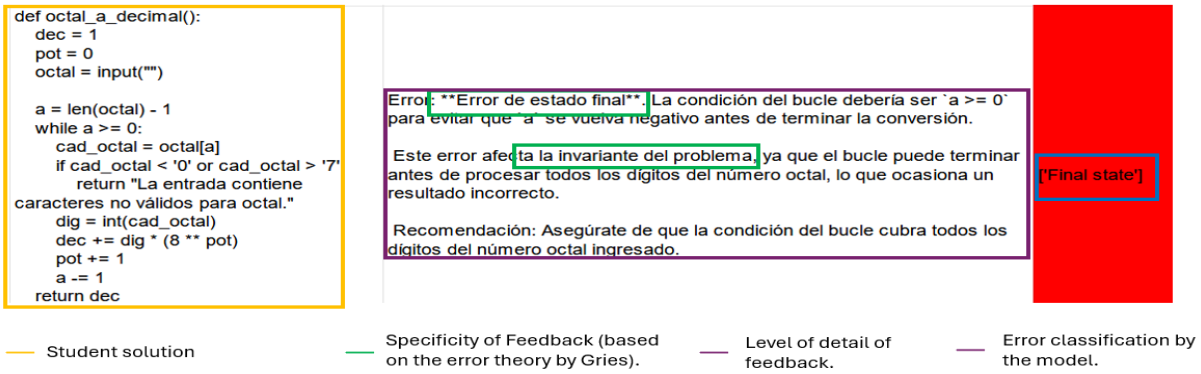

Figure 8.11: An example of a solution from a student with the classification color 'Red'.

8.3.5.2 Analysis of evaluation results feedback in testing.

The tests involved 136 solutions selected from programming exercises with "while" loops developed by the authors and some students. These solutions, correct and incorrect, cover ten basic programming exercises. Each solution underwent a meticulous manual review for error classification.

We extracted the results from the Firebase database. We processed them to analyze the correspondence between the manual classification (corresponding to the review conducted by the researchers on the exercises used for evaluating the error classification model) of the model prediction and the feedback generated by GPT. The review focused on verifying the consistency between the feedback generated and the error classification assigned to each solution. The results are categorized using the following color scheme in the Table 8.2.

We tested GPT-3.5-turbo, Gemini, and Llama in the initial phase with 136 exercises. GPT-3.5-turbo and Gemini struggled with error reduction and feedback precision, with Gemini needing to improve in more complex tasks and often generating misleading information with incorrect recommendations. When run locally, Llama required significant computing resources and produced responses based on an old model.

We found that GPT-4 Mini provided the best response among all the models tested, achieving superior results, as shown below. We also tested the GPT-4 model, which demonstrated enhanced performance. It is superior because it combines greater computational power, context management, and the ability to identify and correct errors. As previously mentioned, these tests are already conducted in the INGInious environment with the ImproB-It API. Therefore, they are included in this section because the functionality of each component is evaluated together, rather than separately as in Chapter 7.

Of the 136 exercises, 100 were correctly classified (blue), 25 contained partial errors (orange), 5 presented generic feedback (yellow), and 6 were incorrectly classified (red), as shown in Figure 8.12.

Most of the 136 solutions were classified correctly (blue), but there are still areas for improvement in more complex exercises (Table 8.3). On simple arithmetic problems such as Least Common Multiple (LCM) and Greatest Common Divisor (GCD), GPT-4 showed high accuracy. However, on more complicated exercises such as prime number verification and cube addition (EJ2, EJ9), more partial classifications were observed (orange), indicating difficulties in identifying specific errors. On summation or numerical conver-

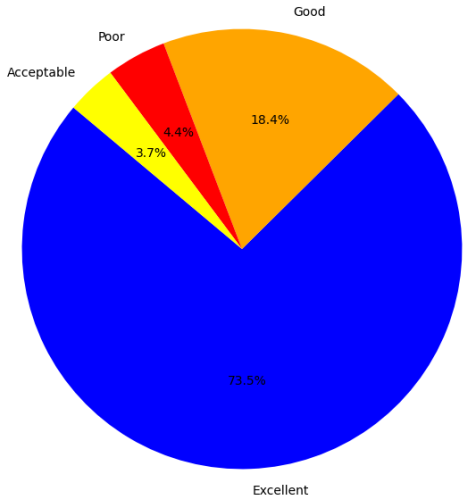

Figure 8.12: Results model error classification and feedback - GPT4.

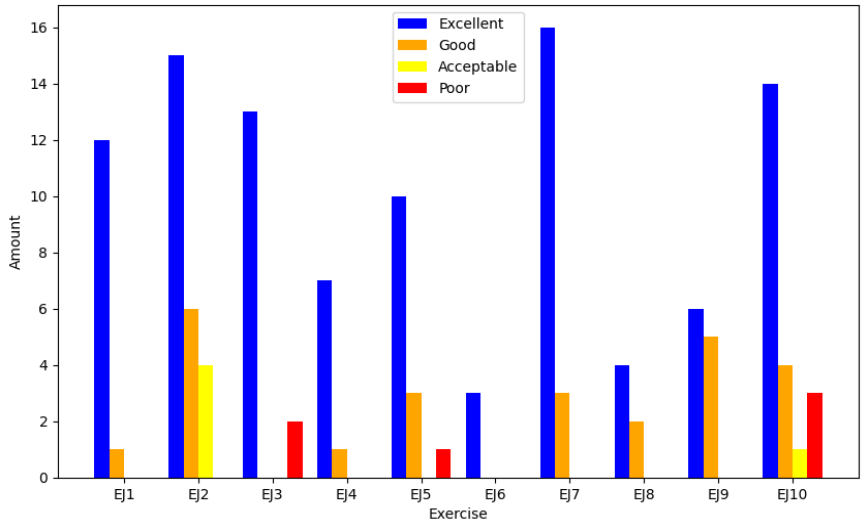

Figure 8.13: Results model error classification and feedback - GPT4 for tasks.

No.	Description	Complexity
1	Factorial using "while" loop	Basic loop and accumulation
2	Sum of cubes of numbers	Basic loop and mathematical operations
3	Prime number verification	Loop with condition checking
4	Factors of a number	Loop to check divisibility
5	Least Common Multiple (LCM)	Understanding LCM and loops
6	Print numbers with increments	Looping through ranges
7	Average of a variable number of integers	Loop and accumulation
8	Mathematical summation (2/n$\hat{2}$)	Fractions and mathematical series
9	Octal to decimal conversion	Base conversion and loops
10	Greatest Common Divisor (GCD)	Advanced algorithm (Euclid's method)

Table 8.3: Exercises and Their Complexity.

sion exercises (EJ3, EJ10), performance was mixed, with more incorrect (red) or incomplete (orange) classifications, revealing limitations in advanced calculations and loops (Figure 8.13). Among the exercises, problems 9 (octal to decimal conversion), 4 (factors of a number), and 8 (mathematical summation $\frac{2}{n^2}$) showed the highest number of prediction errors by the model. Although these exercises could be considered more complex or elaborate, no clear relationship was found between the problem's difficulty and the model's classification errors.

On the other hand, on more straightforward problems, such as calculating averages or printing numbers with increments (EJ1, EJ5), GPT-4 performed correctly with very few errors.

In summary, GPT-4 has shown notable improvements in classification and feedback.

We conducted two additional evaluations on the feedback results. First, we tested iterative programming exercises that were not part of the dataset, using only the problem statement or solution to observe how the model would respond. We also evaluated new solutions for exercises that already had predefined statements and solutions in the dataset.

For the initial evaluation, three of the ten exercises were outside the dataset. We analyzed thirty-one solutions to these exercises, contributed by researchers and students.

The outcomes were grouped into three categories:

- **Excellent (Blue):** Out of the thirty-one solutions, twenty were classified as correctly classification. This indicates that the model correctly identified errors and provided

feedback aligned with those errors. In other words, either the solution was accurately classified, or the feedback directly addressed the actual issues.

- **Good (Orange):** Ten solutions were categorized as containing partial errors, meaning they contained multiple error types. Although the classifier detected at least one error, the feedback only partially matched the identified problems. Despite not being fully comprehensive, the feedback still offered insights that were relevant to one or more detected errors.
- **Poor (Red):** Only one solution received a Red classification. In this instance, the model failed to identify the correct errors, and the feedback was neither accurate nor relevant. Both the classification and feedback were significantly misaligned with the actual issues, making it the least effective outcome.

The model performed well in identifying and classifying errors across most solutions, with 64.5% (20 out of 31) receiving accurate classifications and feedback (Blue). Additionally, 32.2% (10 out of 31) fell into the Orange category, indicating that the model detected some errors but lacked complete accuracy in its feedback. Only one instance (3.2%) was classified as Red, reflecting a rare occurrence of total misalignment between the model's output and the actual errors, highlighting overall reliability (see Figure 8.14).

This distribution shows that the model is effective at providing accurate feedback for common or simpler errors (Blue). However, it tends to struggle with more complex solutions that contain multiple error types (Orange). The single Red case points to occasional instances where the model's interpretation significantly deviated from the expected results.

In the second phase of the investigation, we focused on analyzing iterative programming exercises where the feedback model had prior knowledge of the problem statements and other solutions but lacked access to the specific answers provided by researchers and students (see Figure 8.15).

The classification of 51 out of 136 solutions across ten exercises revealed valuable insights into the performance of the automatic feedback model. Using the established color scheme to evaluate feedback quality, 74% (38 solutions) received a Blue classification. This indicates that the model accurately identified correct answers and/or provided appropriate feedback.

Orange classifications accounted for 16% (8 solutions), signaling cases where multiple errors were present but at least one was correctly detected. This suggests that the model's

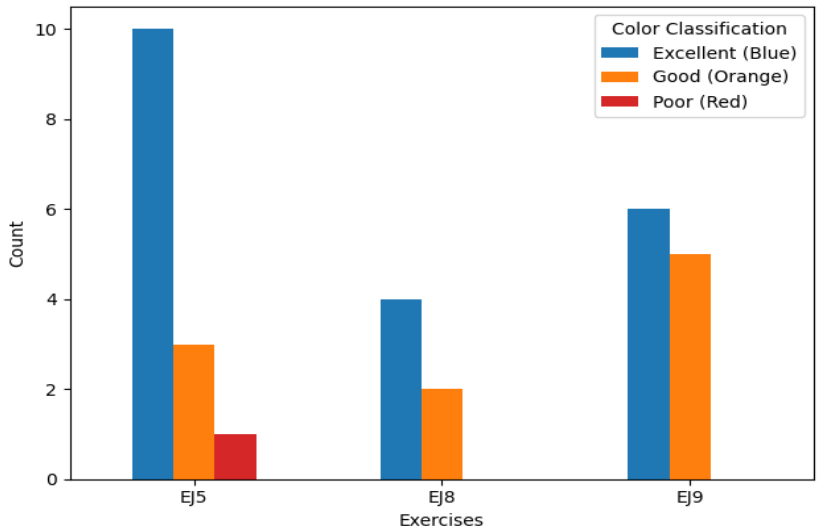

Figure 8.14: Evaluation of Model Performance on Unseen Programming Exercises.

feedback was partially effective and opens the door for deeper analysis into the types of errors identified. For example, exercises EJ2 and EJ4 featured minor issues, highlighting an opportunity to refine feedback and better guide students in improving their understanding.

Yellow classifications were rare, making up just 2% (1 solution), where the model misclassified the errors, resulting in generic feedback. Exercise EJ10 exemplifies this scenario and warrants a detailed review to pinpoint the causes of misclassification. Addressing these instances could enhance the model's precision in providing targeted and relevant feedback.

Lastly, Red classifications, representing 8% (4 solutions), signify cases where the model completely failed to identify errors or deliver accurate feedback. Exercises like EJ3, which included Red responses, require further investigation to understand the contributing factors. Future research should consider revising problem statements and fine-tuning the model to better address frequent errors, ensuring more reliable and effective feedback.

From the analysis of the results, the automatic feedback model demonstrates positive performance, even on exercises that were not part of its training dataset. Although the model was not previously aware of the solutions provided by the researchers and students, 90% of the answers were classified as blue (74%) or orange (16%), suggesting that

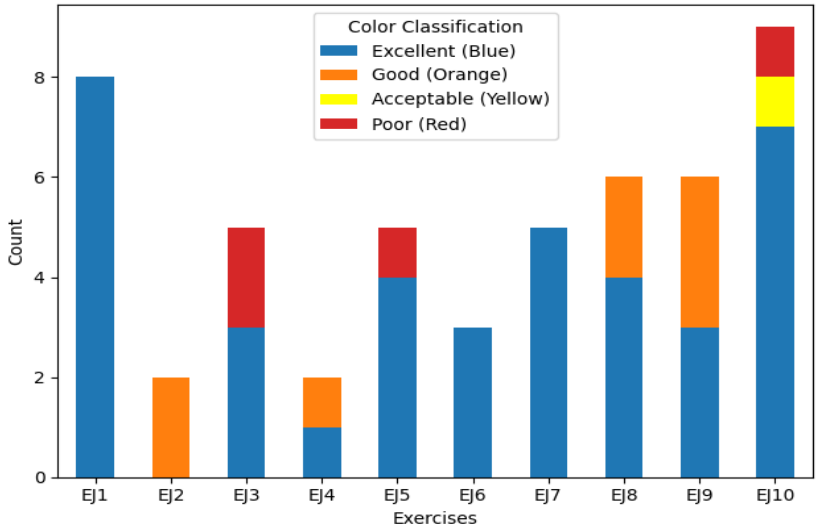

Figure 8.15: Evaluating model performance in programming exercises with unseen solutions.

the model was able to correctly identify valid solutions or provide relevant feedback in most cases. This high percentage reflects the robustness of the model in generalizing its feedback capability beyond pre-trained examples, reinforcing its usefulness in scenarios where new or unseen solutions are presented.

To see the results obtained from the other models, refer to Appendix D.

8.4 Chapter Summary.

This chapter describes the development of the IMProB-It API, integrated with the INGInious platform to provide automatic feedback on programming tasks with "while" loops. The API evaluates students' solutions and generates personalized feedback using AI models, ensuring specific feedback on identified errors.

Its implementation used tools such as Google Cloud, Google Colab Pro, Flask, and databases such as MongoDB and Firebase. The API extracts elements from the code (initial state, condition, and loop transformation), generate embeddings with BERT and CodeBERT and classifies errors for GPT4 to produce accurate feedback.

Finally, the chapter details the functional tests that validated the extraction, tokeniza-

tion, error classification, and automatic feedback, ensuring adequate performance in different scenarios. GPT4 stood out for its accuracy, outperforming GPT3.5 and Gemini in several tasks. Finally, the best-performing classification and feedback generation models were established on the server for subsequent testing with students.

Part IV

Evaluation of the automated feedback model

Chapter 9

Evaluation of the Automated Feedback Model with a Quasi-Experimental Study.

Contents

In this chapter, we present the evaluation of the automated feedback model through a quasi-experimental study. Section 9.1 defines the study design, including the target population, study variables, and the development of the measurement instrument. Section 9.2 outlines the work plan for the evaluation, detailing the application of tests and the testing schedule. Section 9.3 focuses on the evaluation process, including data collection, documentation of results, and interpretation of findings in relation to the study objectives according the variables. The results indicate that implementing automatic feedback in computer programming tasks was beneficial for students, aiding both their progress in the activities and helping them avoid repeating mistakes in the future. Finally, section 9.5 provides a summary of the chapter's main points.

9.1 Definition of the Quasi-Experimental Study.

9.1.1 Considerations in Quasi-Experimental Research.

A quasi-experiment is a research design that typically involves pre-existing groups, allowing researchers to examine the effects of an intervention in a natural setting [155]. In this study, we conduct a quasi-experimental evaluation of an automatic feedback model for programming tasks involving "while" loops. We have one control group and one experimental group, both composed of students from Systems Engineering, Electrical Engineering, and other engineering disciplines (See Table 9.1). The control group includes 40 students: 18 enrolled in the Imperative Programming course and 22 in the Informatics I course. The experimental group comprises 41 students: 16 from Imperative Programming and 25 from Informatics I.

9.1.2 Definition of Target Population and Study Variables.

To test the feedback model previously described, we design an experiment with undergraduate students taking a first programming course at the beginning of their undergraduate program. To do so, it is essential to clearly define the variables to be measured. These variables allow for structuring the analysis and obtaining relevant conclusions about the intervention's results.

The independent variable in this study is the model intervention, that is, automatic

Course Name	Imperative Programming Course / Informatic I
Institution	Universidad del Valle
Location	Cali, Colombia
Program	Systems Engineering and another Engineering
Class Hours	3 hours per week (2 theoretical and 1 practical)
Independent Work Hours	6 hours per week
Credits	3
Course Objective	Learn and implement data structures and control structures, develop algorithmic thinking to solve computational problems
Programming Language	Python

Table 9.1: Course Information for Imperative Programming

feedback in basic programming tasks with loops. This feedback is based on Gries' theory of how to program, which provides a systematic approach to the design and verification of loops. In this case, the goal is to provide students with specific, targeted feedback on errors made in tasks involving "while" loops, enhancing their understanding of iterative structures.

The dependent variable focuses on students' perception of the feedback's usefulness and impact on learning. A questionnaire assesses critical aspects, including the feedback's specificity, relevance, effectiveness, and level of detail. It also gauges how students perceive the influence of feedback on their learning and problem-solving process. Additionally, the questionnaire determines whether students feel the feedback effectively helps them correct specific errors and enhance their programming skills, specifically when working with loops.

Defining the independent and dependent variables clearly enables a more precise analysis of how automatic feedback impacts student learning.

During the 2024-1 academic term, we conducted testing with Group 1 from the Imperative Programming course to ensure the proper functioning of both the INGInious platform and the IMProB-It tool, as well as to make any necessary improvements.

In the 2024-2 academic term, we conducted formal testing by applying the automatic feedback system to an experimental group of 41 students from engineering disciplines, including Systems and Electronic Engineering. These students used the IMProB-It model, received an introduction to program correctness theory for programming loops, and com-

pleted foundational exercises using "while" loops on the INGInious LMS platform. Meanwhile, a control group of 40 students from the same engineering programs received traditional feedback, meaning feedback provided by the instructor. These students only had access to the default evaluation of test cases for each exercise, as provided by INGInious.

9.1.3 Developing a Measurement Instrument for the Study.

Designing the measurement instrument plays a crucial role in evaluating the effectiveness of the automatic feedback provided by the IMProB-It API in an imperative programming course (See Appendix E). To achieve this, we developed a structured survey with multiple sections to capture various dimensions related to the perception and usefulness of the feedback. The questions in the survey are formulated based on the feedback evaluation criteria established by [14] [156], which we will discuss next.

The first section focuses on gathering **personal and general information from students**, including their previous programming experience and their use of AI-assisted tools for learning programming. This data plays a crucial role in contextualizing the results and gaining insights into the variations in students' perceptions of automatic feedback.

The second section examines specific aspects related to the criteria established for evaluating feedback. We apply eight criteria sourced from the literature, particularly from [156], to assess the quality of feedback. These criteria include **feedback type, specificity, level of detail, additional information, tone, comment length, non-misleading information and feedback importance in course**. We emphasize the importance of providing concise yet informative feedback, as longer responses do not necessarily improve clarity.

We classify feedback as "appropriate" when the suggestions accurately address the errors made by students, in line with established programming theory. Table 9.2 shows the concrete rubric we used to evaluate the study.

Finally, the last section assesses students' engagement with their **learning**. To identify their degree of proactivity in the learning process, they are asked whether they consider it important to learn the concepts of the subject and whether they turn to monitors or teachers when they have doubts.

Overall, this measurement instrument offers a comprehensive view of students' perception of the automatic feedback provided by IMProB-It, ranging from its quality and specificity to its impact on students' learning and motivation.

Criteria	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Feedback Type	Feedback lacks clarity, doesn't identify errors or guide corrections.	Feedback is relevant but lacks clarity and provides limited help.	Feedback is somewhat clear, identifies some errors, and offers partial guidance.	Feedback is clear, identifies major errors, and gives useful guidance.	Feedback is very clear, identifies all relevant errors, and provides precise instructions.
Additional Information	No additional information is provided.	Information provided is limited or ambiguous.	Information adds moderate value.	Additional information is provided but lacks detail.	Additional information is thorough and valuable.
Level of Detail	Feedback is very general, with no useful details.	Feedback includes some details but is too basic or generic.	Feedback offers acceptable detail, but lacks specificity.	Feedback provides appropriate detail, specific to the task.	Feedback includes full and specific details on how to correct errors.
Specificity	The feedback is broad and unspecific.	The feedback covers multiple steps and is not very specific.	The feedback is moderately specific.	The feedback focuses on a single next step.	The feedback is precise and limited to a specific step.
Misleading Information	The feedback contains clearly incorrect information.	The feedback is partially incorrect or confusing.	The feedback is somewhat correct but ambiguous.	The feedback is correct but could be clearer.	The feedback is completely correct and precise.
Tone	The tone is inappropriate or unprofessional.	The tone is appropriate but not very motivating.	The tone is neutral or slightly motivating.	The tone is friendly or neutral, appropriate for the situation.	The tone is kind and motivating, promoting a positive attitude.
Length	The feedback is too brief or excessively long.	The feedback is slightly insufficient or excessive in length.	The feedback is of adequate length but could be improved.	The feedback is appropriately sized and provides sufficient information.	The feedback is concise and clear, with the perfect length.
Feedback Importance in Course	Feedback did not support my learning or improve performance.	Feedback was minimally helpful for my learning.	Feedback was somewhat helpful but limited in impact.	Feedback generally supported my learning and performance.	Feedback significantly enhanced my learning and academic performance.

Table 9.2: Integrated rubric for evaluating feedback in the survey.

9.2 Work Plan for Evaluating the Automated Feedback Model.

9.2.1 Test Administration and Target Participants.

We selected the INGInious automatic source code evaluation environment because it has an appropriate interface for managing computer programming courses and assignments. It has a virtual judge that allows students to submit source code solutions for evaluation, and we included the ImproB-It API for automatic feedback for programming assignments with iterations. Automatic code evaluation is based on a comparison method between test cases (inputs) and expected test cases (outputs), and our API performs a subsequent error evaluation with the classification model and subsequently generates feedback on these errors. In addition, the environment allows keeping records of the grades obtained, number of response attempts, average times, grade score, and status indicator for each activity submitted by the students.

We created a course with 10 test exercises that students in the experimental group must complete. They submit their source code to the automatic evaluator, which provides feedback if errors are detected or test cases fail. If students make mistakes, they can revise their code and resubmit it as many times as needed. Also, they may request feedback as often as they wish.

For the experiment, we conducted two interventions: the first with first-semester systems engineering students from two groups, and the second with electronic engineering and other engineering students from two groups, as mentioned in Section 9.1.

9.2.2 Schedule for Tests Applications.

In the term, 2024-2, tests were conducted with two groups of first-semester students, focusing specifically on programming with iterations. These tests took place between September 23 and October 1, 2024.

The control group accessed INGInious, the validation of test cases, and the automated feedback generated by IMProB-It, offering an opportunity to thoroughly evaluate the performance of the automated feedback model on programming tasks involving loops.

In contrast, the control group received traditional feedback from the teacher and validation of test cases in INGInious for each solution submitted by the students. These

validations provided information on the discrepancies between the expected and actual inputs and outputs for each exercise.

9.3 Evaluation of the Automated Feedback Model.

9.3.1 Data Storage Details.

We store the data collected from student submissions in two separate databases, each tailored to manage different aspects of the feedback and analysis process.

The first database is a MongoDB non relational database hosted on the same server where the system operates. This database holds essential course information, including group and class details, as well as student data. We also keep records of the students' code submissions, encompassing both correct and incorrect solutions. This setup ensures efficient querying and simplifies the tracking of student progress and performance throughout the course.

The second database is a non-relational Firebase database, hosted in the cloud. With real-time access and high scalability, it is designed to store data processed by the API. This includes details about programming problems presented to students, their submitted solutions, error classifications identified by the model, and the feedback automatically generated.

Storing this data in Firebase enables swift and efficient analysis, allowing us to evaluate the model's performance in error detection and the quality of the feedback provided.

9.3.2 Study Results Documentation.

The results showed a strong positive response from experimental group (EG) students to the automatic feedback model, with 82% indicating they either "totally agree" or "agree" that the IMProB-It feedback met their needs.

The first three questions characterize students in the quasi-experimental study across both groups: experimental (EG) and control (CG). EG students used the INGInious LMS platform with the IMProB-It API, which provided automatic feedback on errors in "while" loop exercises. The exercises focused on the initial state, final state, and state transformation to support core concept understanding. In contrast, CG students relied on traditional resources and teacher feedback.

We assessed prior **programming experience, tool usage, and feedback perception**. About 17% of EG and 5% of CG students had programming-related work experience, though most lacked any (83% in EG, 95% in CG). For programming knowledge, 49% of EG students were new to programming, while 48% of CG students reported superficial prior knowledge. **Regarding using LMS or AI-supported tools**, 68% of EG students used such tools for the first time, compared to 50% of CG students.

We evaluated the perception of feedback across several dimensions. First, we analyzed the **type of feedback** provided. In question 4, regarding **whether the feedback explains why a program did not produce the expected output**, 70.73% of students in the EG agreed or strongly agreed. This high satisfaction rate reflects their belief that the feedback provides clear explanations of errors. In contrast, only 42.50% of the control group (CG) shares this positive view, indicating a lack of clarity in the feedback they receive.

For question 5, which asks **whether the feedback addresses the most significant errors**, 75.61% of EG students feel that the feedback effectively focuses on critical issues. This highlights the relevance and utility of the feedback in helping them improve their programs. Conversely, only 55.00% of the CG believes the feedback sufficiently targets important aspects, suggesting room for improvement.

In question 6, concerning **whether the feedback helps students understand the necessary changes in their code to correct errors**, an impressive 85.36% of EG participants feel supported. This demonstrates the effectiveness of the feedback in guiding them toward required modifications. In contrast, only 37.50% of the CG reports a similar level of understanding, indicating a significant opportunity for improvement in the quality of feedback they receive.

Test cases play a crucial role as they highlight the discrepancies between the expected and actual results that may occur in an exercise. As shown in the figure 9.2, by accessing the button, we can review the inputs and outputs for each test case recorded for the exercise when it fails the evaluation. Incorrect outputs are highlighted in red, while expected outputs are highlighted in yellow. Additionally, beyond the test cases, I can request automated feedback that helps me better understand why these test cases are failing the evaluation.

In this first section, we can analyze that the control group (GC) does not hold a favorable perception of the type of feedback provided by the teacher. Several factors may explain why an expert teacher might not have been able to deliver effective feedback to their students, as revealed in the last section of the survey. In this section, the question

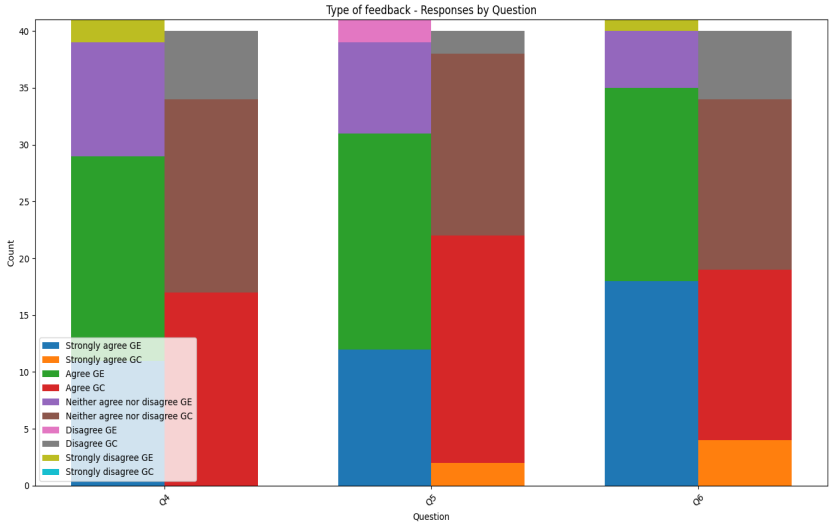

Figure 9.1: Comparison of Responses (GE vs GC) - Type of feedback.

was left open-ended: *"What aspects of feedback do you consider most useful and why?" and "What suggestions do you have to improve the feedback provided by the system?"*

First, the pace at which the course progresses and the speed of topic coverage might have been too fast for some students, particularly those with limited programming experience. One student mentioned, *"Even though I already have some basic programming knowledge, I feel that the topics are moving quickly for my classmates who might not have any programming background."* This comment suggests that the rapid pace of the course may have hindered the professor's ability to provide detailed and personalized feedback, especially for students struggling to keep up.

Second, another contributing factor could be students' perceptions of their progress in relation to their peers. As one student explained, *"I felt a bit behind because I noticed that everyone knew how to program, but I have been informing myself and will keep asking for help if I need it."* This sense of being left behind may have affected the effectiveness of the teacher's feedback, as some students may have required more individual support or clarification, which could not be delivered due to time constraints or resource limitations.

It is also important to consider the significant portion of responses from the CG that selected the "Neither agree nor disagree" option, with a smaller percentage choosing "Disagree." Notably, there were no responses in the "Totally disagree" category.

Also, based on the responses from the experimental group, it can be concluded that

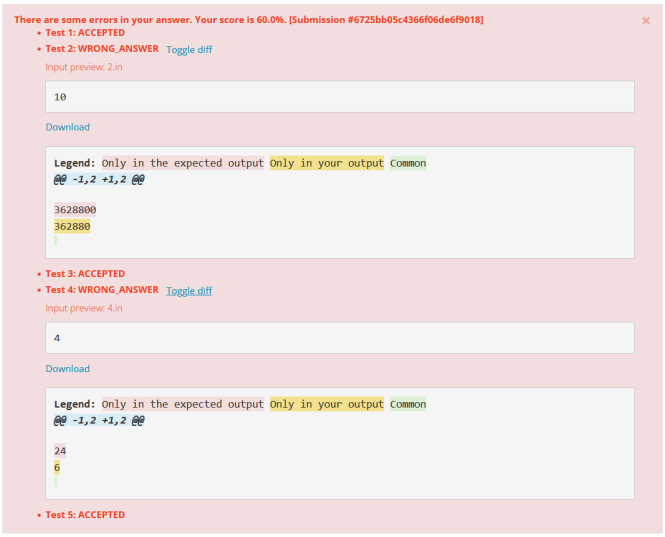

Figure 9.2: INGInious - Solution with errors Cases Test.

providing direct solutions to students might not be the most effective approach for fostering learning. The responses to the survey questions suggest that while students generally feel the feedback they receive helps them understand their errors and how to correct them, there is a clear emphasis on learning from their mistakes rather than simply having the solution given to them.

For example, when asked whether the feedback helped them understand what changes were needed to fix their code, 43.90% strongly agreed, and 41.46% agreed. This indicates that students are benefiting from feedback that clarifies the error and guides them towards the solution, rather than just providing the solution outright. Similarly, 51.22% of students felt that the feedback helped them develop a deeper understanding of fundamental programming concepts, indicating that feedback should focus on fostering conceptual understanding rather than simply fixing the current error.

Regarding the test cases, it's important to note that the system does not provide the direct solution to the exercise. Instead, it presents the expected output for each test case, allowing the student to compare it with their program's output. This approach helps students identify discrepancies and understand what went wrong, but it does not offer the exact solution.

Additionally, the feedback provided serves as a complement to the test case evaluation. It highlights the error the student made, offers suggestions for how to correct it,

and explains whether the mistake affects the invariant of the problem. Importantly, the feedback does not give the solution in code form.

We explored the perception of **additional information** within the feedback provided to students. When asked in question 7 if the feedback provides valuable and relevant information for improving their solutions, a remarkable 92.68% of EG students agree or strongly agree. This high percentage underscores their perception that the feedback contributes significantly to their learning and problem-solving skills. In comparison, only 62.50% of CG participants share this sentiment, suggesting that many do not find the feedback as useful. In Chapter 7, subsection 7.3, the importance of feedback detail is discussed.

Question 8 addresses whether the **feedback helps students develop a deeper understanding of fundamental programming principles**. Here, 90.24% of the EG agrees, demonstrating the feedback's role in enhancing their comprehension of essential concepts. In contrast, only 42.50% of the CG feels this way, indicating challenges in connecting feedback to their knowledge.

In question 9, we assessed whether **the feedback helped students better understand concepts such as initial state, final state, state transformation, and invariants in "while" loops**. Here, 85.37% of EG students agree or strongly agree, showcasing the feedback's effectiveness in clarifying these essential concepts. In contrast, only 27.50% of CG students share this view, with 12.50% disagreeing. This disparity underscores the need for targeted improvements in the feedback given to CG students, as they may not fully grasp the fundamental programming concepts with iterations.

We examined **the level of detail in the feedback**, focusing on the clarity and precision with which he addressed errors. Regarding the clarity of the feedback in question 10, 75.61% of EG students agree that the feedback was clear and precise. This significant score suggests that students in the EG found the feedback more effective in pinpointing errors and offering clear guidance for corrections. In comparison, only 45.00% of the CG feels similarly, indicates that many students struggled to find the feedback as clear or helpful for understanding and fixing their mistakes.

We assessed **the specificity of the feedback** provided to students. In question 11, when asked whether the feedback accurately indicates which aspects of the code need correction, 78.05% of EG students feel satisfied. This highlights the feedback's specificity according the errors types. In contrast, only 62.50% of the CG shares this perception, suggesting that they could benefit from more specific feedback.

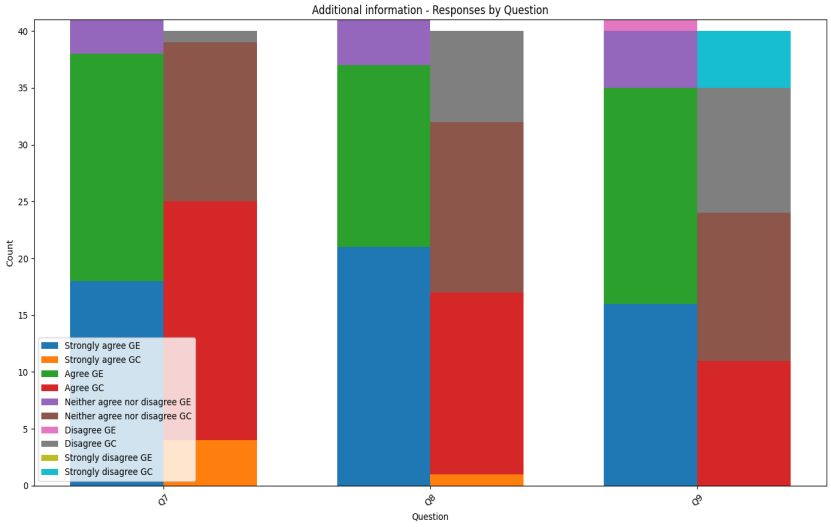

Figure 9.3: Comparison of Responses (GE vs GC) - Additional information.

Regarding whether **the feedback provides valuable information to avoid making the same mistakes**, 87.80% of EG students consider it beneficial. This indicates that the feedback fosters long-term learning. In contrast, only 52.50% of the CG feels the same way. This discrepancy emphasizes the value of specific, forward-looking feedback that equips students with the knowledge to improve continuously.

In question 13, concerning whether suggestions and explanations in **the feedback help correct errors and improve solutions**, 82.93% of EG students report positive experiences. This emphasizes the feedback's utility in the learning process, while only 35.00% of the CG feels the same way.

We focused on evaluating the clarity of the feedback, ensuring it was free from misleading or incorrect information. In question 14, 70.73% of EG students believe the feedback was consistently correct. This perception reinforces their view of the feedback's accuracy. In the CG, 70.00% has a similar opinion. This indicates that both groups generally perceived the feedback as accurate and free of misleading content. The difference between the groups was not substantial, these findings highlight the need to further refine the feedback to better address students' needs and minimize any remaining ambiguity.

Regarding **the tone of the feedback** in question 15, 48.78% of EG students believe it was friendly and motivating. This suggests that the feedback felt supportive and encouraging, helping them maintain a positive attitude when addressing errors. No students

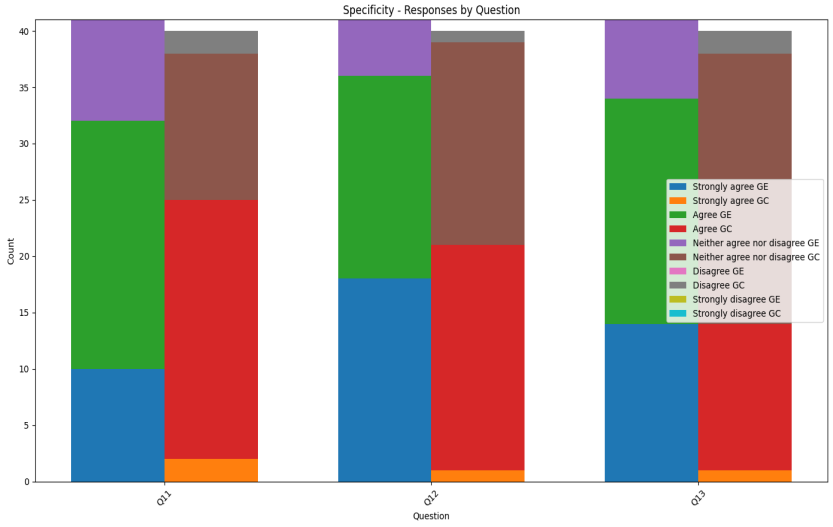

Figure 9.4: Comparison of Responses (GE vs GC) - Specificity.

in the EG expressed disagreement.. In contrast, only 10.00% of the CG shares this view, indicating a need for improvement in the tone of their feedback.

We examined **the feedback length** to assess whether it offered sufficient information without overwhelming students. In question 16, 43.90% of EG students felt the feedback length was appropriate, indicating that many found a good balance in the information provided. In contrast, only 10.00% of CG students considered the feedback length ideal.

We explore **the significance of feedback** through two questions. In Question 17, only 7.50% of control group (CG) students strongly agreed, and 47.50% agreed that the feedback they received helped improve their academic performance. In contrast, 36.59% of experimental group (EG) students strongly agreed and 39.02% agreed, revealing that while some students recognize the value of feedback, this indicates a positive perception of feedback's impact on their learning.

An analysis of the limitations of IMProB-It reveals several factors that may contribute to the fact that a portion of the experimental group did not fully recognize the value of the feedback. One issue mentioned by students is the visibility of the interface, with some reporting that *"the color of the text is not very visible."* This issue could hinder the legibility of the feedback, making it harder for students to fully absorb the information provided.

Additionally, some students felt that the organization of the feedback could be improved. One student suggested that *the comments should be displayed below the validation of*

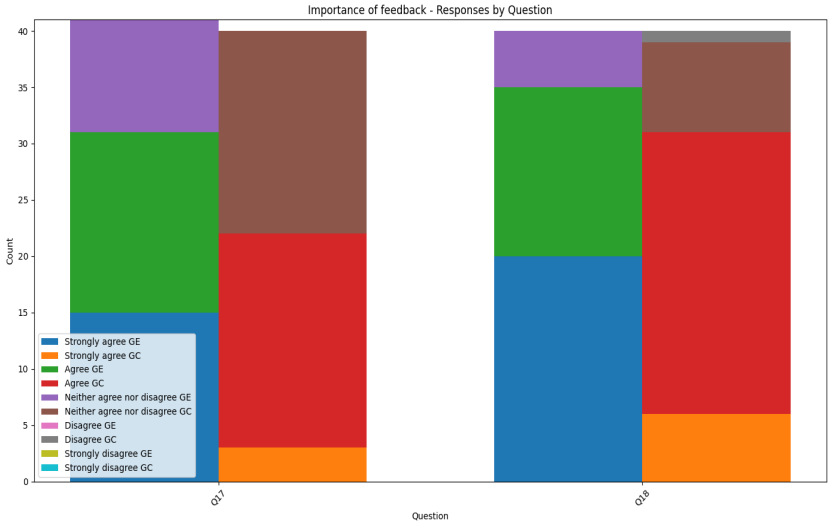

Figure 9.5: Comparison of Responses (GE vs GC) - Importance Feedback.

input and output, rather than being shown together. This indicates that students may benefit from a clearer separation of feedback elements, allowing them to focus more easily on the relevant information.

Another limitation noted by students was the lack of detail in the some feedback. While students appreciated the identification of errors, some felt that the feedback could be more specific about the nature of the mistakes. One student commented that the feedback could be *"a little more detailed when specifying the errors*." This suggests that a more thorough explanation of the mistakes, along with clearer guidance on how to correct them, would enhance the feedback's effectiveness.

Lastly, *the structure of the feedback* was seen as potentially confusing, as some students found that the explanations given by the AI were difficult to follow due to the numbering sequence. A more organized presentation of the feedback, such as in a list format, might improve clarity and help students better understand the provided explanations.

In question 18, concerning whether LMS tools and online judges support their learning, 48.78% of EG students agree that these tools benefit their education. This indicates strong enthusiasm for these tools compared to the CG, where only 15.00% agrees.

Finally, we discuss **commitment to learning**. Question 19 focused on the perceived importance of learning the subject matter. Regarding commitment to learning the material, 43.90% of EG students feel dedicated. This indicates greater engagement compared

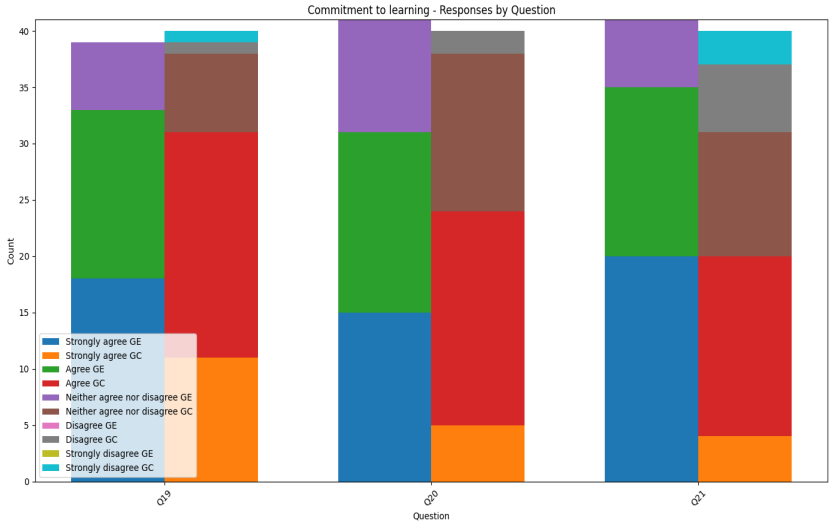

Figure 9.6: Comparison of Responses (GE vs GC) - Commitment to learning.

to 27.50% of the CG.

In question 20, concerning the review of notes and resources, 36.59% of EG students feel committed to understanding concepts. This highlights their commitment compared to only 12.50% of the CG. These results suggest that EG students are more dedicated to deeply understanding the material than their CG peers.

In question 21, regarding willingness to ask for help, 48.78% of EG students are inclined to seek assistance, demonstrating a proactive approach to their learning, underscoring their commitment to overcoming learning challenges. In contrast, only 10.00% of the CG feels the same way.

In summary, the experimental group (EG) demonstrates a more favorable perception of the utility, accuracy, and detail of the feedback they received. This suggests that the ImproB-It automatic feedback model positively impacts their learning experience, promoting greater motivation and willingness to engage with the subject matter and utilize technological tools for learning.

To view the complete survey results, see Appendix E.

9.3.3 Interpretation of the Results in Relation to the Study Objectives.

In our research, we hypothesize that automated feedback, based on the theory of loop correction, enriches student learning. To address this hypothesis, we formulate the research question:

How can program correctness theory, particularly loop theory, be used to provide adequate feedback on iterative programming tasks using machine learning techniques?

To assess the validity of our hypothesis, we conduct a series of analyses, including comparisons of central measures and standard deviation evaluations. Additionally, we employ formal statistical tests, such as the Mann-Whitney U test, to substantiate our study's findings.

The Mann-Whitney U test is a non-parametric statistical test designed for ordinal data or continuous data that do not conform to a normal distribution. This test evaluates whether there is a significant difference between the distributions of two independent groups by comparing the ranks of the values in each group, rather than their raw scores. By focusing on ranks, the Mann-Whitney U test is less affected by outliers and skewed data, relying instead on the relative positions of values within the datasets [157].

9.3.3.1 Effectiveness and Impact of Automated Feedback on Student Learning.

In the experimental group (EG), 34.1% of the students strongly agreed that the suggestions and explanations received in the feedback allowed them to correct their errors and improve their solutions effectively, while 48.8% agreed. Only 17.1% were neutral, and none disagreed. This indicates that a large majority of the students perceived the feedback as specific and useful for correcting errors, suggesting a positive impact on their ability to improve their solutions.

Regarding the accuracy of the feedback, 29.3% of the EG students strongly agreed that the feedback was always correct and did not include confusing or incorrect information, while 41.5% agreed. Additionally, 24.4% were neutral, and 4.9% disagreed. Although most considered the feedback to be accurate, the presence of a small percentage who perceived the information as confusing or incorrect suggests that some errors in the model's classification may have affected the clarity of the feedback.

Concerning the improvement of academic performance, 36.6% of the EG students strongly agreed that the feedback allowed them to improve their performance in the subject, while 39% agreed. Additionally, 24.4% were neutral, with no reports of disagreement.

These results indicate that the feedback was recognized by the majority as a positive factor in their learning and academic performance.

Overall, the analysis of these responses suggests that the feedback provided was mostly effective and specific, helping students to correct their errors and improve their academic performance. However, the perception of confusing or incorrect information by some students indicates that classification errors in the model could negatively impact the clarity and usefulness of the feedback, potentially affecting the learning process of those students.

9.3.3.2 Comparison of Central Measures (Means and Medians).

The results show that the means of the experimental group (EG) are consistently higher than those of the control group (CG) on all questions. It indicates that, on average, students in the EG who used the ImproB-It automatic feedback system received more positive feedback than students in the CG. This finding suggests that the use of ImproB-It positively impacted students' perception of their learning process.

Furthermore, the EG medians remain at 5, while the CG medians range between 3 and 4. This difference highlights the higher positive evaluation of the EG students toward the ImproB-It tool. The core measures suggest that EG students have a more favorable perception of learning and automated feedback's effectiveness than CG students.

For example, in Question 8, which assesses feedback as a tool to improve understanding of fundamental programming principles, the EG mean was 4.41, while the CG mean was 3.25. It shows that EG students perceived higher effectiveness in feedback to strengthen their understanding of key concepts.

Similarly, in Question 9, related to understanding the concepts of initial state, final state, and state transformation in "while" loops, the EG mean was significantly higher (4.22) than the CG mean (2.75). The automated feedback provided by ImproB-It was particularly helpful to EG students in applying and understanding these fundamental concepts. To examine the distribution of means, refer to Figure 9.7 and Figure 9.8.

9.3.3.3 Dispersion Analysis (Standard Deviation).

The EG standard deviations are lower than the CGs for most questions. It indicates that EG students had more consistent perceptions of the feedback they received. The lower dispersion in EG responses suggests that the feedback provided by ImproB-It was more

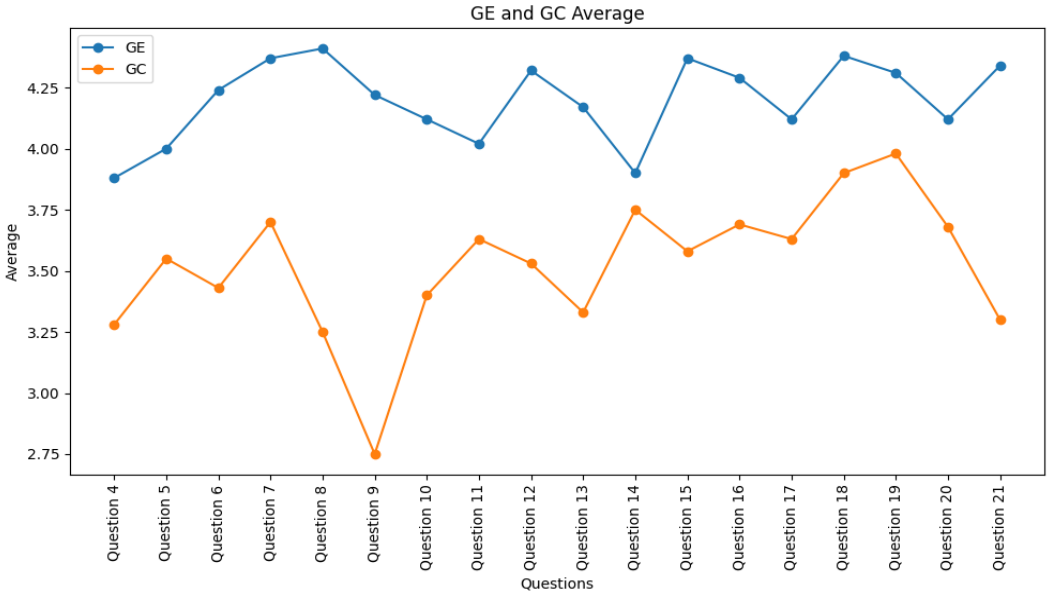

Figure 9.7: GE and CG Average.

precise and uniform for students. In contrast, the CGs exhibit more significant variability in their responses, which may reflect different levels of understanding or satisfaction with the feedback received in that group.

For example, in Question 9, the EG's standard deviation is 0.76, while the GC's is 0.99, suggesting that students in the EG shared a more consistent view on the usefulness of feedback in understanding "while" loops, compared to the GC, which showed a greater disparity in their responses.

In other questions, such as Questions 5, 6, and 12, the GC's standard deviations are more significant than those in the EG. It can be interpreted as an indicator that students in the CG experienced more varied perceptions regarding the effectiveness of feedback. The greater spread suggests that some students in the CG found helpful feedback while others did not, which could be due to a lack of clarity or personalization in the input.

For example, in Question 12, which assesses whether feedback helps avoid making the same mistakes in the future, the CG has a standard deviation of 0.59 versus 0.69 in the EG. It reflects greater consistency in the EG's responses regarding the usefulness of feedback in correcting future errors.

For a comparison of means along with their standard deviations, see Figure 9.8.

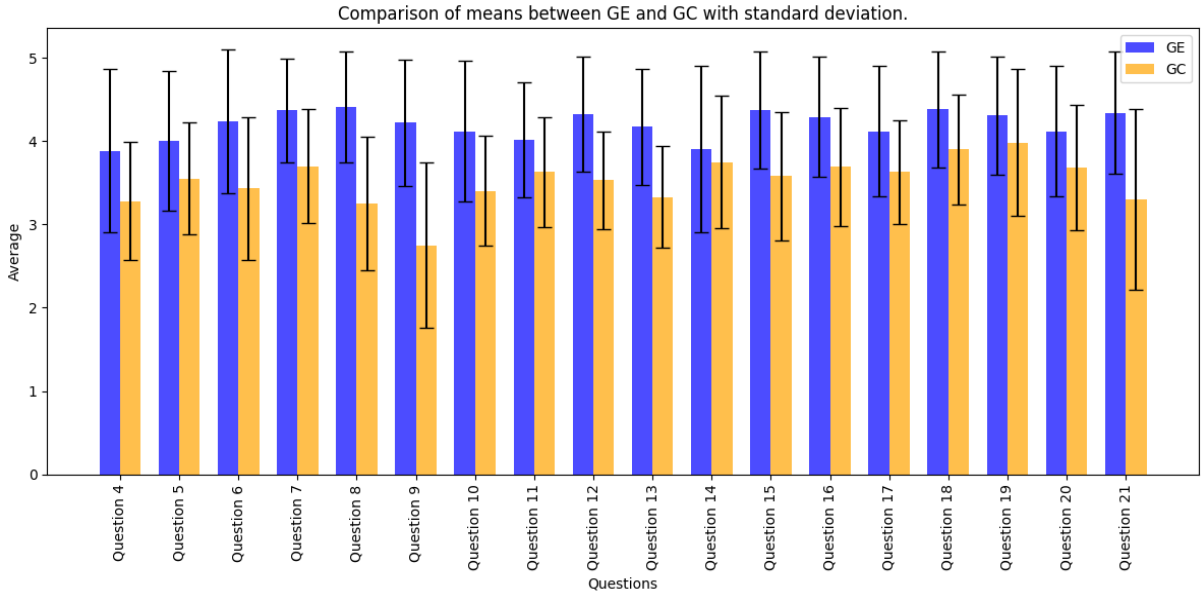

Figure 9.8: Comparison of means between GE and CG with standard deviation.

9.3.3.4 Notable Differences in Specific Questions.

Further analysis reveals significant differences in critical questions. For example, in Question 9, the EG's mean (4.22) is notably higher than the CG's (2.75), indicating a significant disparity in the perception of feedback related to programming concepts. The standard deviations also reflect this difference: the CG has a considerably higher standard deviation (0.99) than the EG (0.76), suggesting greater variability in the CG's experience with the feedback received.

Questions 14 and 21 also show higher means in the EG, while the CG has higher standard deviations. It reinforces the idea that CG students experience more diverse and less uniform perceptions regarding feedback, which could affect their learning.

9.3.3.5 Questions without Significant Differences.

In questions such as Question 14, where the means and standard deviations between EG and CG do not differ substantially, it can be concluded that both groups have similar perceptions regarding certain aspects of feedback, such as its accuracy and clarity. It suggests

that traditional teacher-provided and automated feedback can be equally effective.

For example, in Question 14, the EG mean is 3.90, while the CG mean is 3.75, with similar standard deviations. Both groups had an almost equivalent perception of feedback accuracy.

The results suggest that the ImproB-It automatic feedback positively impacts students' perception of the GE. Not only do students rate the feedback received more favorably, but they also show less dispersion in their responses, indicating a more uniform experience.

9.3.3.6 Test Mann-Whitney U.

The Mann-Whitney U test is a nonparametric statistical test used to compare two independent groups, in this case, the experimental group (EG) and the control group (CG), to determine whether there are significant differences between them in terms of the variables measured (in this case, students' responses to feedback questions). This test is practical when the data do not follow a normal distribution or when the sample size is small.

The Mann-Whitney U test compares the median of the two samples and assesses whether one tends to have higher values than the other. Suppose the p-value associated with the test is less than a predefined significance level (usually 0.05). In that case, the null hypothesis that the two groups come from the same distribution is rejected, suggesting a significant difference between the groups [157].

Below in Table 9.3 are the results of the Mann-Whitney U test for the study questions:

Overall, the lower p-values indicate that, in most cases, there are significant differences between the experimental group (EG) and the control group (CG). It suggests that the use of ImproB-It produced different perceptions in the students, reflecting a positive impact on those who used this tool. However, in questions 14 and 19, there was insufficient evidence to reject the null hypothesis, implying that both groups had similar perceptions in these aspects.

Most questions present statistically significant differences, with p-values less than 0.05. It means that the EG students who used ImproB-It perceived the feedback received more favorably than those in the CG group. In particular, the questions with the lowest p-values, such as questions 8, 9, and 21, indicate that the EG students felt that the automated feedback not only helped them develop a deeper understanding of the fundamental principles of programming but also better understand key concepts, such as "while" loops. Furthermore, these students positively valued the support provided by the teacher or

Criteria	Q No.	GE Mean	GE Std. Dev.	CG Mean	CG Std. Dev.	Mann-Whitney p-value
Feedback Type	Q4	3.88	0.98	3.28	0.71	0.0008
	Q5	4.00	0.84	3.55	0.67	0.0076
	Q6	4.24	0.86	3.43	0.86	0.000036
Additional Information	Q7	4.37	0.62	3.70	0.68	0.000042
	Q8	4.41	0.67	3.25	0.80	0.000000018
	Q9	4.22	0.76	2.75	0.99	0.000000004
Detail Level	Q10	4.12	0.84	3.40	0.66	0.00012
Specificity	Q11	4.02	0.69	3.63	0.66	0.0159
	Q12	4.32	0.69	3.53	0.59	0.0000025
	Q13	4.17	0.70	3.33	0.61	0.0000011
Misleading Information	Q14	3.90	1.00	3.75	0.80	0.3071
Tone	Q15	4.37	0.70	3.58	0.77	0.000021
Length	Q16	4.29	0.72	3.69	0.71	0.0006
Importance of Feedback	Q17	4.12	0.78	3.63	0.62	0.0036
	Q18	4.38	0.70	3.90	0.66	0.0025
Commitment to Learning	Q19	4.31	0.71	3.98	0.88	0.0936
	Q20	4.12	0.78	3.68	0.75	0.0156
	Q21	4.34	0.73	3.30	1.08	0.0000096

Table 9.3: Mann-Whitney U Test Results Comparing GE and GC.

monitor in the learning process.

The students in the experimental group (EG) also highlighted the usefulness of feedback in understanding and correcting errors. Questions 6, 7, and 12, which focus on the clarity and relevance of feedback, show significant differences between the groups, reinforcing the conclusion that the EG students found the feedback provided by ImproB-It to be a key resource for their learning. This positive perception of the feedback is a reassuring sign of ImproB-It's effectiveness.

On the other hand, in question 14, which assesses the accuracy of feedback, the p-value of 0.3071 suggests no significant differences between the groups. This implies that both the EG and the CG similarly perceived the accuracy of the feedback.

In conclusion, the results of the Mann-Whitney U test demonstrate the positive impact of ImproB-It on student perceptions in the experimental group. The feedback generated

by this automated tool was instrumental in understanding concepts and correcting errors, suggesting that ImproB-It is a valuable resource to enrich the programming learning process. While no significant differences were observed in aspects such as feedback accuracy and motivation regarding the course topics, the overall positive impact of ImproB-It is a testament to its potential (See Figure 9.9).

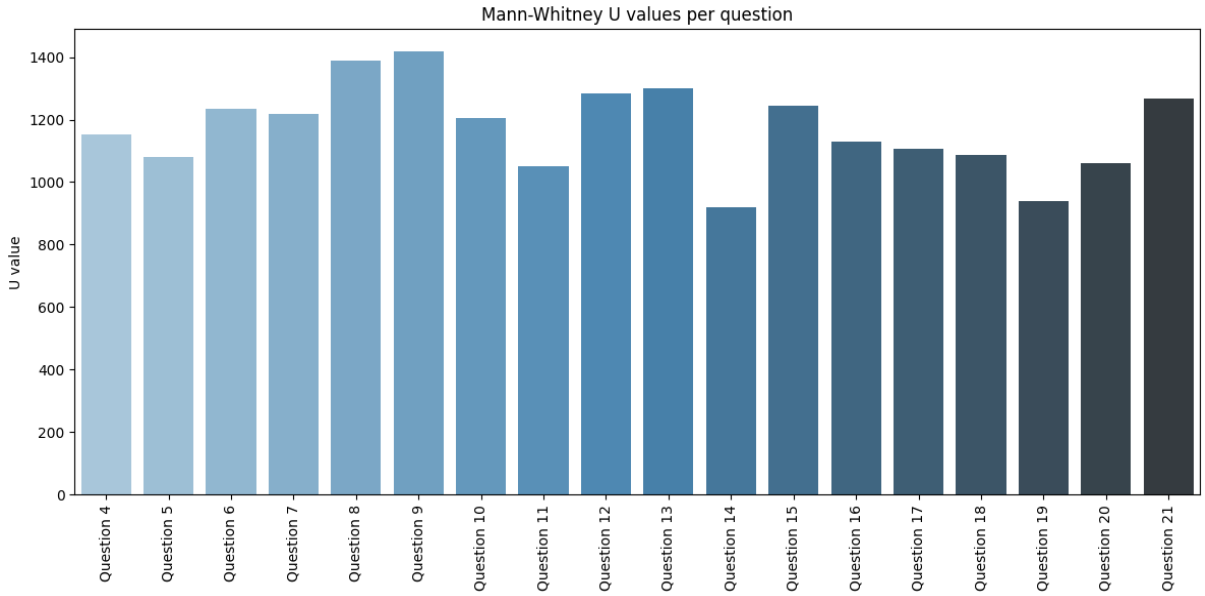

Figure 9.9: Mann-Whitney U values per question.

9.3.3.7 Analysis of Results by Criterion.

Below is a detailed analysis of the results by criterion, highlighting the significant differences between the experimental group (EG) and the control group (CG) based on the Mann-Whitney U test results. This analysis groups the survey questions based on critical aspects of feedback in programming tasks.

The first criterion, Feedback Type (See Figure 9.10), explores how students perceive the explanation provided about errors in their programs and the help that feedback offers them in understanding what changes to make to their code. Students in the EG rated the clarity and usefulness of the feedback significantly higher (Mean GE = 4.00; CG = 3.55) than in the CG, with statistically significant differences ($p < 0.01$). The ImproB-It tool was more effective in explaining errors and pointing out the changes needed to correct

them. The ability of feedback to focus on critical errors was also rated higher by the EG, reinforcing the idea that ImproB-It helped students focus on the most vital issues in their code.

In the Additional Information criterion, which assesses the usefulness of feedback in learning fundamental concepts, the EG reported a much more positive perception (Mean EG = 4.41; CG = 3.25), with highly significant differences ($p < 0.000001$). EG students considered that feedback helped them improve their solutions and deepened their understanding of fundamental programming concepts, especially regarding "while" loops. It highlights a considerable positive impact of ImproB-It, which allowed EG students to assimilate fundamental principles better.

Regarding the Level of Detail, EG students perceived that the feedback provided was more precise in describing errors and how to correct them (Mean EG = 4.12; CG = 3.40, $p < 0.001$). It shows that the tool helped students receive more precise instructions on improving their code, facilitating the correction and learning process.

Regarding the Specificity criterion, EG rated the feedback's accuracy more highly in pointing out which aspects of the code needed correction (Mean EG = 4.02; CG = 3.63, $p < 0.02$). This difference suggests that ImproB-It was able to more effectively identify and communicate specific errors in students' code, thus improving the learning experience.

On the other hand, regarding Misleading Information, no significant differences were found between the two groups ($p > 0.30$), which indicates that both the EG and the CG perceived the accuracy of the feedback similarly, without seeing it as confusing or incorrect.

The criterion of Commitment to Learning did not show statistically significant differences between the groups ($p > 0.09$). However, the EG students tended to value learning in the Subject more. It indicates that although ImproB-It did not generate a drastic increase in general commitment, there was a positive trend.

Finally, in the Importance of Feedback in the Subject criterion, the EG students reported a greater willingness to ask the teacher or monitor for help when they did not understand a topic (Mean EG = 4.34; CG = 3.30, $p < 0.00001$). The feedback provided by ImproB-It gave them more confidence to ask for additional support when needed, a positive indicator of the tool's effect on their learning process.

In conclusion, the analysis shows that the EG who used ImproB-It had a significantly better experience in most of the criteria evaluated. This highlights the clarity, usefulness, and accuracy of the feedback received, reinforcing the effectiveness of ImproB-It in im-

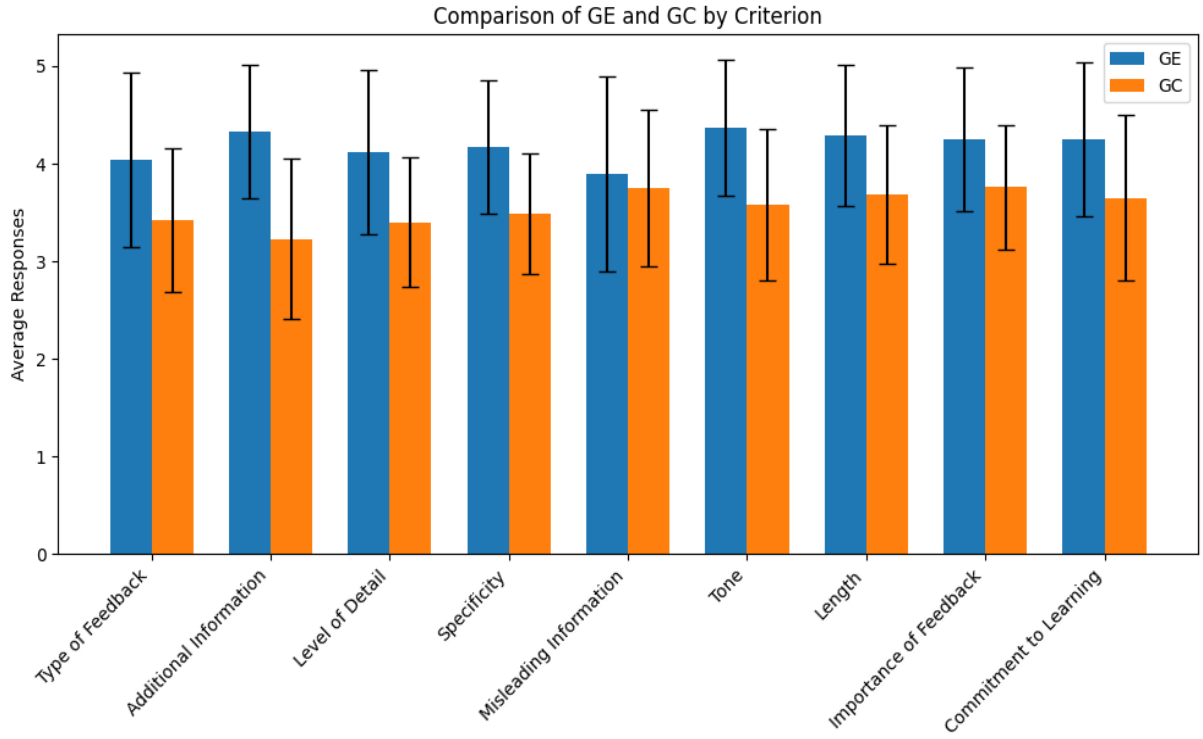

Figure 9.10: Comparison GE and CG Criterion.

proving the feedback and learning process in programming tasks.

9.4 Comparative Analysis of IMProB-It with Other Studies and Limits.

IMProB-It shares several similarities with the works mentioned in Chapter 4, as both utilize automated techniques to evaluate student code, aiming to support instructors in the assessment and feedback generation for students' submitted solutions. Both IMProB-It and these studies strive to provide quick and accurate feedback to support the learning process. However, there are key differences in their approaches and outcomes.

IMProB-It specializes in offering detailed textual feedback on specific errors, explaining how to correct them and their impact on the problem's invariant without directly providing the solution in code. This approach is crucial for fostering critical thinking

and autonomous problem-solving among students. The results show that students highly value IMProB-It's feedback, stating that it helps them better understand their mistakes, correct them, and avoid repeating them in the future.

In contrast, the study [144] focuses on automatic error detection using static analysis and machine learning models, enhancing the efficiency of the evaluation process by predicting program correctness before executing tests. Researchers in this study reported that rapid feedback allowed for more efficient error correction, though they also noted that the lack of detailed feedback limited their understanding of the errors.

On the other hand, in the study [30] employs LLMs to generate high-level hints, which, although initially helpful, may become less effective as students progress. In this study, students noted that the hints were useful for general guidance but were sometimes confusing or not directly applicable to their specific issues.

Feedback quality is another important point of comparison. IMProB-It provides specific feedback on the committed error, how to correct it, and its impact on the program's logic, which is essential for deep learning in loop comprehension. In contrast, [144] offers binary feedback on whether the code will pass the tests, which is less detailed. Researchers in this latter study appreciated the speed but mentioned that the lack of specificity in the feedback made correcting more complex errors difficult. The study [30] approach can help guide students initially but does not always offer the clarity and specificity needed to address particular mistakes.

In terms of the type of feedback, IMProB-It focuses on specific errors, their solution, and their effect on the program's logic, integrating invariant-based evaluations, which are crucial for understanding loop logic. Conversely, the research [144] provides more general feedback on code correctness before executing tests, without a detailed logic analysis. In [30] gives high-level hints, which may not be directly applicable to specific error corrections.

Although IMProB-It has clear advantages in the quality and detail of feedback, it also presents certain limitations. Its dependency on LMS platforms, such as INGInious, can limit the number of requests handled simultaneously, resulting in delays in receiving feedback compared to tools that use machine learning or AST analysis for quicker and more efficient assessments.

The simplicity of IMProB-It API may also restrict its ability to identify a broad range of errors and provide detailed feedback in complex contexts. Regarding generalization, other studies utilizing large datasets to train their models may offer more consistent re-

sults across various contexts, whereas IMProB-It could face challenges if its dataset is limited to tasks involving "while" loops in programming.

In conclusion, while IMProB-It is effective in providing detailed and specific feedback essential for learning iteration programming theory, it could benefit from improvements in coverage, scalability, personalization, and generalization to compete with more advanced tools in different educational contexts. Researchers and students in the various studies expressed appreciation for both rapid and detailed feedback, though with diverse critiques depending on the approach and quality of the feedback provided.

9.5 Chapter summary.

This chapter presents the evaluation of the automated feedback model through a quasi-experimental study aimed at determining its impact on learning in programming tasks with loops. It focuses on first-semester students in introductory programming courses, identifying key variables related to automatic feedback in error correction and students' perceptions of that feedback. The chapter details the development of measurement instruments, including Likert-scale surveys and programming tests, ensuring data validity, and outlines a work plan for testing both experimental and control groups under comparable conditions.

The results from the test reveal significant differences between the experimental group (EG) and the control group (CG) in most criteria, indicating that the automated feedback model positively influences student perceptions and understanding of programming concepts. The EG reported higher clarity, usefulness, and specificity of feedback. Although no significant differences were observed in feedback accuracy and learning engagement, overall trends suggest that the ImproB-It tool generates adequate feedback and enriches programming learning with iterations, the feedback positively contributes to the learning experience by providing valuable information for error correction and conceptual understanding.

Part V

Summary

Chapter 10

Conclusions and Future Work

Contents

In this dissertation, we investigate the role of automated feedback in computer programming education, particularly in tasks involving iterative constructs like while loops. We examine how machine learning error classification models and large language models (LLMs) can enrich the learning experience by delivering targeted feedback that addresses common student errors. This concluding section outlines the key challenges encountered in implementing automated feedback systems, details the objectives pursued during the research, and highlights the significant contributions made to the field. Additionally, we consider potential avenues for future research, emphasizing the continuous evolution of technology in educational contexts and its implications for enrich computer programming learning.

10.1 Dissertation Summary and Contributions.

Automatic feedback in computer programming education has become an essential tool for supporting student learning. This approach seeks immediate and specific assistance

in resolving errors, especially in tasks involving loops, such as "while" loops. With the increasing use of artificial intelligence technologies, tools such as LLMs allow for generating feedback tailored to errors identified in students' code, thus facilitating more dynamic and practical learning.

In particular, programming problems involving loops present unique challenges due to the logical nature of iterations. Many students need help understanding concepts such as initial, final, and state transformations within a loop. Generating targeted feedback can address these difficulties by identifying common errors and offering explanations that help students rectify their approaches.

Furthermore, implementing machine learning models to classify errors in programming tasks allows for a deeper analysis of the nature of loops. By classifying errors according to well-defined categories, more targeted feedback can be provided that not only points out the existence of an error but also guides the student to the correct solution. This transforms feedback from a simple error statement into an educational resource that fosters understanding and supports problem-solving ability.

Finally, integrating these tools into learning platforms can support learning. Combining automatic feedback with error classification creates a learning environment that not only helps students understand the theory of how to program iterations but also gives them the skills necessary to tackle more complex programming problems in the future.

10.1.1 Addressed Challenges and Goals.

This dissertation tackles important challenges and goals associated with implementing automated feedback systems in computer programming education, particularly for tasks involving iterations, such as while loops. A significant challenge is the wide range of errors students encounter while learning to program, which complicates the design of adequate feedback mechanisms that incorporate programming theory. Automated feedback must identify errors and provide meaningful explanations that guide students toward understanding and correcting their mistakes.

Another challenge lies in integrating machine learning models and large language models (LLMs) into existing educational frameworks. These models require careful tuning and adaptation to analyze student submissions effectively and generate relevant feedback. Additionally, the rapid pace of technological advancements necessitates ongoing evaluation and refinement of these systems to ensure they remain effective and aligned

with pedagogical goals.

A central goal of this research is to enhance the learning experience by leveraging Gries's theory of programming iterations, which emphasizes understanding the principles underlying iterative processes. We aim to develop automated feedback mechanisms that accurately classify errors in programming tasks and provide tailored support, thereby enriching the learning process. By integrating Gries's theoretical framework, we show how automated feedback can clarify common pitfalls in student solutions and reinforce foundational programming concepts. Through this work, we contribute to the field by demonstrating how technology and robust pedagogical theories can improve programming education, ultimately making learning more accessible and effective for students, we propose five challenges and their corresponding goals:

C1/C2/G1: The challenges in teaching and learning computer programming, particularly with loops, combined with the limited research on automated feedback for iterative programming tasks, underscore the necessity to *identify common types of errors that arise in programming tasks involving loops and develop a comprehensive taxonomy for automated feedback specifically designed for these tasks in introductory courses. This approach aims to bridge the existing gap by focusing on the connection between loop theory and appropriate feedback.*

C3/G2: The limited availability of programming exercises focused on iteration theory in existing repositories underscores the necessity to *create a synthetic dataset that includes programming tasks with iterations and their annotated specifications, based on the feedback taxonomy and defined error types.*

C2/C4/G3: The limited research on automatic feedback for iterative programming tasks, along with the necessity of generating feedback from error classification based on Gries theory, highlights the importance of *defining an automated feedback model at the task level for programming tasks involving iterations in introductory programming courses, focusing on how to generate feedback from the classification of errors according to Gries theory.*

C5/G4: The goal of automatically generating error-adapted feedback in iterative programming through AI emphasizes the need to *develop an API for INGInious that utilizes machine learning to generate automated feedback on programming tasks with iterations in introductory programming courses. This feedback aims to clearly identify and explain er-*

rors, thereby enhancing students' understanding of their mistakes and their grasp of loop programming.

C6/G5: The availability of an online platform for programming courses emphasizes the importance of *integrating the automated feedback API with a platform for practical exercises. This connection will enable the evaluation of the proposed model through a quasi-experimental study, allowing for the collection of results from the implementation of automated feedback in introductory programming courses.*

10.1.2 Contributions.

This research makes substantial contributions to programming education and artificial intelligence, specifically focusing on automated feedback for iterations of programming tasks. Each chapter of this dissertation addresses critical aspects of this topic, contributing to the overarching goal of enriching programming learning with loops.

Exploring programming theory emphasizes the critical role of control structures, particularly loops, in programming education. By identifying the common challenges students face when learning iterative programming concepts, this research contributes to a deeper understanding of these fundamental principles. This understanding is essential for addressing the difficulties learners encounter.

In the realm of feedback, this dissertation reviews various feedback models and strategies, highlighting the importance of tailored feedback in supporting student learning. By underscoring the necessity for automated tools that facilitate large-scale implementation of feedback mechanisms, the research lays a theoretical foundation for effective feedback practices in programming courses. This is crucial for fostering student progress and ensuring relevant and constructive feedback.

The investigation into machine learning techniques reveals significant gaps in the existing literature regarding integrating iteration theory with practical feedback mechanisms. By identifying these gaps, the research points to machine learning's potential to transform feedback generation, paving the way for innovative solutions that can significantly improve programming instruction.

A significant contribution of this research is the development of a taxonomy of errors in iterative programming tasks. This taxonomy is based on Gries' theory of iterations, which provides a conceptual framework for identifying the errors that may arise when students work with these structures. Through a comprehensive review of the existing literature,

this study not only categorizes the most common errors in iterative programming but also establishes a connection between these errors and the type of feedback to be provided.

In this case, we opted to deliver feedback at the task level, as it is the most appropriate level for providing students with information when they submit their solutions for evaluation. Once this level was determined, we identified the types of feedback that could be offered, with a particular emphasis on error-correction feedback based on the proposed taxonomy. This type of feedback was then configured in the GPT prompt.

Later, The meticulous creation of a dataset tailored explicitly for learning programming with while loops represent another critical aspect of this work. This dataset is designed to support the training of machine learning models focused on classifying errors in basic programming tasks. It includes a carefully curated selection of programming exercises, each accompanied by extracted code snippets organized into three key components: initial state, final state, and state transformation.

Following the established taxonomy, this dataset systematically analyzes each exercise to identify and classify the errors present. The errors are meticulously annotated and reviewed to ensure accuracy and relevance, drawing on Gries' theory of programming iterations as a guiding framework. This rigorous process enhances the dataset's quality and ensures it aligns with the theoretical foundations underpinning programming instruction.

Developing an automated feedback model tailored for iterative programming tasks bridges the gap between theoretical knowledge and practical application. This model consists of two distinct yet interconnected components: the first focuses on classifying errors in programming tasks involving "while" loops. In contrast, the second generates automated feedback for these identified errors using GPT-4.

The initial phase of the model entails a comprehensive error classification process that systematically identifies common mistakes students make when working with while loops. This classification is grounded in the theoretical framework established by Gries, ensuring that the identified errors are relevant and appropriately categorized. Following the classification of errors, the model transitions into its second phase, utilizing GPT-4 to provide automated feedback tailored to the identified mistakes. By crafting prompts that focus on the particular errors classified in the first phase, the model generates feedback that is not only relevant but also actionable, enabling students to understand their mistakes and learn from them.

Implementing the IMProB-It API, which integrates the automated feedback model with the INGInious platform, showcases the practical application of artificial intelligence

in providing automated feedback. This technical development validates the feedback generation process and ensures that students receive clear and actionable insights into their programming errors, further enhancing their learning experience.

Finally, the quasi-experimental study evaluates the impact of the automated feedback model on students in iterative programming tasks. Analyzing the data collected from first-semester students provides empirical evidence of the model's effectiveness, demonstrating significant improvements in student perceptions of feedback and understanding of programming concepts. This contribution validates the practical application of the IMProB-It tool and highlights its potential to enrich the learning experience in introductory programming courses.

In conclusion, this research's contributions are significant. We provide valuable insights into programming theory, feedback's role, and machine learning's application in generating automated educational support. By developing a structured approach to feedback and a robust dataset, along with the innovative IMProB-It API, this research lays a strong foundation for supporting teaching and learning in programming courses. The empirical evidence gathered through the quasi-experimental study reinforces the model's effectiveness, suggesting that integrating automated feedback mechanisms can support and enrich students' programming learning.

10.2 Future Work.

This research opens multiple avenues for future work that can further enrich the field of programming learning and the use of artificial intelligence in education.

- **Expanding the Error Taxonomy:** Develop and validate a broader taxonomy that includes not only common errors in tasks with while loops but also errors in other control structures, such as for loops, conditionals, and functions. This would allow for more comprehensive feedback in programming learning.
- **Improving the Feedback Model:** Research and develop new machine learning techniques to enhance the accuracy and relevance of error classification and the feedback generated. This could include incorporating more advanced models.
- **Research in Different Programming Languages:** Adapt and validate the automated feedback model in other programming languages, such as Java, C++, or JavaScript.

This would help assess the generality of feedback strategies and the model's effectiveness in various educational contexts.

- **Long-Term Evaluation of the Impact on Learning:** Conduct longitudinal studies that evaluate the long-term impact of automated feedback on students' programming skills, motivation, and confidence in their problem-solving abilities.
- **Studying the Effect of Feedback Personalization:** Investigate how personalizing feedback based on individual student progress and preferences can impact learning and retention of programming knowledge.
- **Developing a Prototype for Advanced Learners:** Create versions of the model that address more complex and advanced programming errors, targeting intermediate or advanced-level learners, which would broaden the model's applicability.

Bibliography

[1] I. Eteng, S. Akpotuzor, S. O. Akinola, and I. Agbonlahor, "A review on effective approach to teaching computer programming to undergraduates in developing countries," vol. 16, p. e01240, 2022-07-01. 9

[2] H. B. JoÃ£o and A. C. d. Francisco, "Active learning in the context of the teaching/learning of computer programming: A systematic review," vol. 20, pp. 201–220, 2021-05-11. 9

[3] The University of Georgia, Tbilisi, Georgia, S. Abesadze, and D. Nozadze, "Make 21st century education: The importance of teaching programming in schools," pp. 158–163, 2020. 9

[4] C. S. Cheah, "Factors contributing to the difficulties in teaching and learning of computer programming: A literature review," vol. 12, no. 2, p. ep272, 2020-05-08. Publisher: Bastas. 10, 13, 83, 84

[5] A. Luxton-Reilly, "Learning to program is easy," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pp. 284–289, Association for Computing Machinery, 2016-07-11. 10

[6] R. P. Medeiros, G. L. Ramalho, and T. P. FalcÃ£o, "A systematic literature review on teaching and learning introductory programming in higher education," vol. 62, no. 2, pp. 77–90, 2019. Conference Name: IEEE Transactions on Education. 10, 32, 83

[7] C.-S. Cheah, "Screencasting: how effective is it in developing positive attitude towards the learning of c++ computer programming," vol. IX (LXXI), no. 2, 2019. Place: Ploiesti, Romania Publisher: PG University Ploiesti Publishling House. 10, 84

[8] A. V. Robins, "Novice programmers and introductory programming," in *The Cambridge Handbook of Computing Education Research* (A. V. Robins and S. A. Fincher, eds.), Cambridge Handbooks in Psychology, pp. 327–376, Cambridge University Press, 2019. 10

[9] K. M and A. B, "The impact of different teaching approaches and languages on student learning of introductory programming concepts," 2016-01-18. Publisher: ACM PUB27 New York, NY, USA. 10

[10] E. M. David Boud, "Feedback in higher and professional education: Understanding it and doing it well," 2013. 10

[11] J. Hattie and H. Timperley, "The power of feedback," vol. 77, no. 1, pp. 81–112, 2007-03-01. Publisher: American Educational Research Association. 10, 43, 48, 81, 86

[12] A. N. Kluger and A. DeNisi, "The effects of feedback interventions on performance: A historical review, a meta-analysis, and a preliminary feedback intervention theory," vol. 119, pp. 254–284, 1996. Place: US Publisher: American Psychological Association. 10

[13] S. Narciss, "Feedback strategies for interactive learning tasks," pp. 125–144, 2008-01-01. 10, 14, 44, 45, 86

[14] Keuning Hieke, Jeuring Johan, and Heeren Bastiaan, "A systematic literature review of automated feedback generation for programming exercises," 2018-09-28. Publisher: ACM PUB27 New York, NY, USA. 10, 11, 14, 42, 45, 46, 47, 48, 67, 70, 71, 72, 74, 82, 166

[15] S. Srikant and V. Aggarwal, "A system to grade computer programming skills using machine learning," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '14, pp. 1887–1896, Association for Computing Machinery, 2014-08-24. 11, 48, 60

[16] B. Paassen, B. Mokbel, and B. Hammer, "Adaptive structure metrics for automated feedback provision in intelligent tutoring systems," vol. 192, pp. 3–13, 2016-06-05. 11, 47, 49, 50

[17] D. Azcona, I.-H. Hsiao, and A. F. Smeaton, "Detecting students-at-risk in computer programming classes with learning analytics from studentsâ digital footprints," vol. 29, no. 4, pp. 759–788, 2019-09-01. 11, 48, 55, 61, 111

[18] M. Lepp, T. Palts, P. Luik, K. Papli, R. Suviste, M. SÃ¤de, K. Hollo, V. Vaherpuu, and E. Tonisson, "Troubleshooters for tasks of introductory programming MOOCs," vol. 19, no. 4, 2018-09-26. 11, 46, 48, 49, 50, 59

[19] S. Oeda and M. Chieda, "Visualization of programming skill structure by log-data analysis with decision tree," vol. 159, pp. 582–589, 2019-01-01. 11, 47, 59, 60, 69, 111

[20] P. Beck, M. J. Mohammadi-Aragh, and C. Archibald, "An initial exploration of machine learning techniques to classify source code comments in real-time," 2019-06-15. 11, 46, 61

[21] B. Akram, H. Azizolsoltani, W. Min, E. Wiebe, A. Navied, B. W. Mott, K. Boyer, and J. C. Lester, "A data-driven approach to automatically assessing concept-level CS competencies based on student programs," in *CSEDM@EDM*, 2020. 11, 14, 46, 49, 50, 73

[22] J. Krugel, P. Hubwieser, M. Goedicke, M. Striewe, M. Talbot, C. Olbricht, M. Schypula, and S. Zettler, "Automated measurement of competencies and generation of feedback in object-oriented programming courses," in *2020 IEEE Global Engineering Education Conference (EDUCON)*, pp. 329–338, 2020-04. ISSN: 2165-9567. 11, 47, 49, 50

[23] S. Sharma, P. Agarwal, P. Mor, and A. Karkare, "TipsC: Tips and corrections for programming MOOCs," in *Artificial Intelligence in Education* (C. Penstein RosÃ©, R. MartÃnez-Maldonado, H. U. Hoppe, R. Luckin, M. Mavrikis, K. Porayska-Pomsta, B. McLaren, and B. du Boulay, eds.), Lecture Notes in Computer Science, pp. 322–326, Springer International Publishing, 2018. 11, 47, 48, 62

[24] S. Suhailan, M. Mat Deris, S. Abdul Samad, and M. Burhanuddin, "A recommended feedback model of a programming exercise using clustering-based group assistance," vol. 7, 2019. 11, 48

[25] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani, "Semi-supervised verified feedback generation," 2016-03-15. Number: arXiv:1603.04584. 11, 48, 49, 50, 62

[26] D. Gries, *The Science of Programming*. 1987-02-01. 11, 31, 33, 34, 94, 102, 103, 108, 138

[27] D. Sleeman, R. T. Putnam, J. Baxter, and L. Kuspa, "Pascal and high school students: A study of errors," vol. 2, no. 1, pp. 5–23, 1986-02-01. Publisher: SAGE Publications Inc. 14, 38

[28] S. Xinogalos, "Designing and deploying programming courses: Strategies, tools, difficulties and pedagogy," vol. 21, no. 3, pp. 559–588, 2016-05-01. 14, 32, 34, 38

[29] S. LiÃ©nardy, L. Leduc, D. Verpoorten, and B. Donnet, "CafÃ©: Automatic correction and feedback of programming challenges for a CS1 course," in *Proceedings of the Twenty-Second Australasian Computing Education Conference*, ACE'20, pp. 95–104, Association for Computing Machinery, 2020-02-03. 14, 15, 72

[30] L. Roest, H. Keuning, and J. Jeuring, "Next-step hint generation for introductory programming using large language models," in *Proceedings of the 26th Australasian Computing Education Conference*, pp. 144–153, 2024. 14, 50, 67, 74, 187

[31] M. Wu, M. Mosse, N. Goodman, and C. Piech, "Zero shot learning for code education: Rubric sampling with deep learning inference," 2018-12-16. Number: arXiv:1809.01357. 14, 46, 48, 49, 50, 64, 70, 73

[32] S. LiÃ©nardy, "Learning computer programming around a CAFÃ," in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, pp. 318–319, Association for Computing Machinery, 2020-08-10. 14, 72

[33] I. Azaiz, O. Deckarm, and S. Strickroth, "Ai-enhanced auto-correction of programming exercises: How effective is gpt-3.5?," *arXiv preprint arXiv:2311.10737*, 2023. 14, 49, 73

[34] A. Böttcher, V. Thurner, and D. Zehetmeier, "Alignment of teaching and electronic exams and empirical classification of errors for an introductory programming class," in *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*, pp. 1–10, IEEE, 2020. 14, 15, 80

[35] C. Izu, A. Weerasinghe, and C. Pope, "A study of code design skills in novice programmers using the solo taxonomy," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pp. 251–259, 2016. 14, 15, 80

[36] A. Ebrahimi, D. Kopec, and C. Schweikert, "Taxonomy of novice programming error patterns with plan, web, and object solutions," *ACM Computing Surveys*, vol. 38, no. 2, pp. 1–24, 2006. 14, 15, 80

[37] S. Han, Y. Wang, and X. Lu, "Errorclr: Semantic error classification, localization and repair for introductory programming assignments," in *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '23, (New York, NY, USA), p. 1345â1354, Association for Computing Machinery, 2023. 15

[38] R. Gupta, A. Kanade, and S. Shevade, "Deep reinforcement learning for syntactic error repair in student programs," vol. 33, no. 1, pp. 930–937, 2019-07-17. Number: 01. 15, 47, 49, 50, 62

[39] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, and C. Szabo, "Introductory programming: a systematic literature review," in *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018 Companion, pp. 55–106, Association for Computing Machinery, 2018-07-02. 32

[40] E. Golan, "Multiple paradigms in cs1- cs2." 32

[41] R. Segura, F. Pino, C. OgÃ¡yar, and A. Rueda, "VRâOCKS: A virtual reality game for learning the basic concepts of programming," vol. 28, pp. 31–40, 2020-01-01. 32

[42] P. Van Roy, "Concepts, techniques, and models of computer programming," 2004. 32, 34, 36

[43] D. A. Kranch, "Teaching the novice programmer: A study of instructional sequences and perception," vol. 17, no. 3, pp. 291–313, 2012-09-01. 33

[44] J. L. Antonakos and K. C. Mansfield, *Programación estructurada en C*. Prentice Hall Iberia, 1997. Google-Books-ID: 2iPJAAAACAAJ. 33

[45] T. SirkiÃ¤ and J. Sorva, "Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises," in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12, pp. 19–28, Association for Computing Machinery, 2012-11-15. 34, 38

[46] Y. Qian and J. Lehman, "Studentsâ misconceptions and other difficulties in introductory programming: A literature review," *ACM Transactions on Computing Education (TOCE)*, vol. 18, no. 1, pp. 1–24, 2017. 38

[47] M. Mladenović, I. Boljat, and Z. Zanko, "Comparing loops misconceptions in block-based and text-based programming languages at the k-12 level," vol. 23, no. 4, pp. 1483–1500, 2018-07-01. 38

[48] R. D. Pea, "Language-independent conceptual bugs in novice programming," vol. 2, no. 1, pp. 25–36, 1986-02-01. Publisher: SAGE Publications Inc. 38

[49] R. Teusner, T. Hille, and T. Staubitz, "Effects of automated interventions in programming assignments: Evidence from a field experiment," in *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*, pp. 1–10, 2018-06-26. 42

[50] R. Morris, T. Perry, and L. Wardle, "Formative assessment and feedback for learning in higher education: A systematic review," *Review of Education*, vol. 9, no. 3, p. e3292, 2021. 42

[51] K. Li and D. R. Moore, "Motivating students in massive open online courses (MOOCs) using the attention, relevance, confidence, satisfaction (ARCS) model," vol. 2, no. 2, pp. 102–113, 2018-12-01. 42

[52] J. W. Harrold and J. Sandland, "The influence of immediate homework feedback on student performance and satisfaction in an engineering MOOC," in *2018 Learning With MOOCS (LWMOOCS)*, pp. 90–93, 2018-09. 42

[53] J. Zambrano, L. Cano, and K. Presiga, "Virtualidad y MOOC desde la perspectiva de estudiantes universitarios," vol. 8, no. 15, pp. 106–119, 2017-12-23. Number: 15. 42

[54] W. Xing, C. Li, G. Chen, X. Huang, J. Chao, J. Massicotte, and C. Xie, "Automatic assessment of studentsâ engineering design performance using a bayesian network model," vol. 59, no. 2, pp. 230–256, 2021-04-01. Publisher: SAGE Publications Inc. 42, 47

[55] D. R. Sadler, "Formative assessment and the design of instructional systems," vol. 18, no. 2, pp. 119–144, 1989. 43

[56] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-elle: an adaptable programming tutor for haskell giving automated feedback," vol. 27, no. 1, pp. 65–100, 2017-03-01. 46, 48, 49, 50, 70, 71

[57] A. K. Dominguez, K. Yacef, and J. Curran, "Data mining to generate individualised feedback," in *Proceedings of the 10th international conference on Intelligent Tutoring Systems - Volume Part II*, ITS'10, pp. 303–305, Springer-Verlag, 2010-06-14. 46, 49, 50

[58] E. Glassman, L. Fischer, J. Scott, and R. Miller, "Foobaz: Variable name feedback for student code at scale," 2015-11-01. 46, 47, 49, 70

[59] G. L, ScottJeremy, SinghRishabh, G. J, and M. C, "OverCode," 2015-03-10. Publisher: ACM PUB27 New York, NY, USA. 46, 47

[60] L. Cheniti Belcadhi, "Personalized feedback for self assessment in lifelong learning environments based on semantic web," vol. 55, pp. 562–570, 2016-02-01. 46, 47

[61] M. d. M. SÃ¡nchez-Vera, J. T. FernÃ¡ndez-Breis, D. Castellanos-Nieves, F. Frutos-Morales, and M. P. Prendes-Espinosa, "Semantic web technologies for generating feedback in online assessment environments," vol. 33, pp. 152–165, 2012-09-01. 46, 47

[62] B. Paassen, J. Jensen, and B. Hammer, "Execution traces as a powerful data representation for intelligent tutoring systems for programming," in *EDM*, 2016. 46, 61

[63] U. Z. Ahmed, N. Srivastava, R. Sindhgatta, and A. Karkare, "Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '20, pp. 139–150, Association for Computing Machinery, 2020-06-27. 46, 48, 49, 50, 64

[64] W. Zhou, Y. Pan, Y. Zhou, and G. Sun, "The framework of a new online judge system for programming education," in *Proceedings of ACM Turing Celebration Conference - China*, TURC '18, pp. 9–14, Association for Computing Machinery, 2018-05-18. 47, 49, 50

[65] M. Nascimento, E. AraÃºjo, D. Serey, and J. Figueiredo, "The role of source code vocabulary in programming teaching and learning," in *2020 IEEE Frontiers in Education Conference (FIE)*, pp. 1–8, 2020-10. ISSN: 2377-634X. 47, 49, 50, 68, 69

[66] N. Funabiki, T. Mohri, and S. Yamaguchi, "Toward personalized learning in JPLAS: Generating and scoring functions for debugging questions," in *2016 IEEE 5th Global Conference on Consumer Electronics*, pp. 1–4, 2016-10. 47

[67] Q. Hao, D. Smith, L. Ding, A. Ko, C. Ottaway, J. Wilson, K. Arakawa, A. Turcan, T. Poehlman, and T. Greer, "Towards understanding the effective design of automated formative feedback for programming assignments," 2021-01-31. 47, 48, 49, 50

[68] J. L. Reguera and Y. F. Leiva, "A learning methodology for object oriented programming with effective support from the PA3p automatic evaluation platform," in *2017 36th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–8, 2017-10. 47

[69] M. Day, M. R. Penumala, and J. Gonzalez-Sanchez, "Annete: An intelligent tutoring companion embedded into the eclipse IDE," in *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, pp. 71–80, 2019-12. 47, 49, 50, 63

[70] H. M. Jamil, "Automated personalized assessment of computational thinking MOOC assignments," in *2017 IEEE 17th International Conference on Advanced Learning Technologies (ICALT)*, pp. 261–263, 2017-07. ISSN: 2161-377X. 47, 48, 49, 50

[71] A. Lobanov, T. Bryksin, and A. Shpilman, "Automatic classification of error types in solutions to programming assignments at online learning platform," in *Artificial Intelligence in Education* (S. Isotani, E. MillÃ¡n, A. Ogan, P. Hastings, B. McLaren, and R. Luckin, eds.), Lecture Notes in Computer Science, pp. 174–178, Springer International Publishing, 2019. 47, 48, 61

[72] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: Scalable homework search for massive open online programming courses," pp. 491–502, 2014-04-07. 47, 70

[73] K. Wang, B. Lin, B. Rettig, P. Pardi, and R. Singh, "Data-driven feedback generator for online programing courses," in *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pp. 257–260, Association for Computing Machinery, 2017-04-12. 47, 70

[74] J. Broisin and C. Herouard, "Design and evaluation of a semantic indicator for automatically supporting programming learning," in *12th International Conference on Educational Data Mining (EDM 2019)*, pp. 270–275, 2019-07. 47, 49, 50

[75] Z. Chen, A. Nguyen, A. Schlender, and J. Ngiam, "Real-time programming exercise feedback in MOOCs," in *EDM*, 2017. 47, 48, 70

[76] M. PÃ¤rtel, M. Luukkainen, A. Vihavainen, and T. Vikberg, "Test my code," vol. 5, no. 3, pp. 271–283, 2013-01. Publisher: Inderscience Publishers. 47

[77] C. Wilcox, "Testing strategies for the automated grading of student programs," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pp. 437–442, Association for Computing Machinery, 2016-02-17. 47

[78] C. Wang, W. Zhang, H. Zhao, and Z. Jin, "Towards a fictional collective programming scenario: an approach based on the EIF loop," vol. 25, no. 5, pp. 3671–3710, 2020-09-01. 47

[79] R. Suzuki, G. Soares, E. Glassman, A. Head, L. D'Antoni, and B. Hartmann, "Exploring the design space of automatically synthesized hints for introductory programming assignments," in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '17, pp. 2951–2958, Association for Computing Machinery, 2017-05-06. 47

[80] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 740–751, Association for Computing Machinery, 2017-08-21. 48, 49, 50, 70, 71

[81] S. Chow, K. Yacef, I. Koprinska, and J. Curran, "Automated data-driven hints for computer programming students," in *Adjunct Publication of the 25th Conference on User Modeling, Adaptation and Personalization*, UMAP '17, pp. 5–10, Association for Computing Machinery, 2017-07-09. 48

[82] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pp. 15–26, Association for Computing Machinery, 2013-06-16. 48, 68, 69, 70

[83] V. J. Marin, T. Pereira, S. Sridharan, and C. R. Rivero, "Automated personalized feedback in introductory java programming MOOCs," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 1259–1270, 2017-04. ISSN: 2375-026X. 48, 49, 50, 70

[84] A. Annamaa, R. Suviste, and V. Vene, "Comparing different styles of automated feedback for programming exercises," in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Calling '17, pp. 183–184, Association for Computing Machinery, 2017-11-16. 48

[85] A. Gordillo, "Effect of an instructor-centered tool for automatic assessment of programming assignments on studentsâ perceptions and performance," vol. 11, no. 20, p. 5568, 2019-01. Number: 20 Publisher: Multidisciplinary Digital Publishing Institute. 48

[86] S. Buyrukoglu, F. Batmaz, and R. Lock, "Improving marking efficiency for longer programming solutions based on a semi-automated assessment approach," vol. 27, no. 3, pp. 733–743, 2019. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22094. 48

[87] D. Joyner, "Intelligent evaluation and feedback in support of a credit-bearing MOOC," in *Artificial Intelligence in Education* (C. Penstein RosÃ©, R. MartÃnez-Maldonado, H. U. Hoppe, R. Luckin, M. Mavrikis, K. Porayska-Pomsta, B. McLaren, and B. du Boulay, eds.), Lecture Notes in Computer Science, pp. 166–170, Springer International Publishing, 2018. 48, 70

[88] T. J. Tiam-Lee and K. Sumi, "Adaptive feedback based on student emotion in a system for programming practice," *Intelligent Tutoring Systems Springer*, 2018. 48

[89] T. Price and T. Barnes, "An exploration of data-driven hint generation in an open-ended programming problem," in *EDM*, 2015. 48

[90] P. Antonucci, C. Estler, D. NikoliÄ, M. Piccioni, and B. Meyer, "An incremental hint system for automated programming assignments," in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pp. 320–325, Association for Computing Machinery, 2015-06-22. 48

[91] D. Perelman, J. Bishop, S. Gulwani, and D. Grossman, "Automated feedback and recognition through data mining in code hunt," 2015-07-01. 48

[92] P. Ardimento, M. Bernardi, and M. Cimitile, "Software analytics to support students in object-oriented programming tasks: An empirical study," vol. 8, pp. 132171–132187, 2020-07-17. 48, 50

[93] H. Yin, J. Moghadam, and A. Fox, "Clustering student programming assignments to multiply instructor leverage," in *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, pp. 367–372, Association for Computing Machinery, 2015-03-14. 48, 62, 70

[94] A. Kyrilov and D. C. Noelle, *Using Case-Based Reasoning to Improve the Quality of Feedback Provided by Automated Grading Systems*. International Association for the Development of the Information Society, 2014-07. Publication Title: International Association for Development of the Information Society. 48, 60

[95] J. Ramirez-Echeverry, F. Restrepo-Calle, and F. GonzÃ¡lez, "UNCODE: interactive system for learning and automatic evaluation of computer programming skills," 2018-07-02. 48

[96] W. Wang, R. Zhi, A. Milliken, N. Lytle, and T. Price, "Crescendo: Engaging students to self-paced programming practices," pp. 859–865, 2020-02-26. 49, 50

[97] S. Marwan, J. Jay Williams, and T. Price, "An evaluation of the impact of automated programming hints on performance and learning," in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, pp. 61–70, Association for Computing Machinery, 2019-07-30. 49, 50

[98] O. Mirmotahari, Y. Berg, S. Gjessing, E. Fremstad, and C. Damsa, "A case-study of automated feedback assessment," in *2019 IEEE Global Engineering Education Conference (EDUCON)*, pp. 1190–1197, 2019-04. ISSN: 2165-9567. 49, 50

[99] S. Marwan, P. Shabrina, A. Milliken, I. Menezes, V. Catete, T. Price, and T. Barnes, "Promoting studentsâ progress-monitoring behavior during block-based programming," pp. 1–10, 2021-11-18. 49, 50

[100] B. Grawemeyer, J. Halloran, M. England, and D. Croft, "Feedback and engagement on an introductory programming module," pp. 17–20, 2022. 49, 50

[101] A. Rump, "Automated assessment of learning objectives in programming assignments," 2020-07-03. Publisher: University of Twente. 49, 50

[102] R. Yilmaz and F. G. Karaoglan Yilmaz, "Augmented intelligence in programming learning: Examining student views on the use of chatgpt for programming learning," *Computers in Human Behavior: Artificial Humans*, vol. 1, no. 2, p. 100005, 2023. 49, 57

[103] U. Z. Ahmed, R. Sindhgatta, N. Srivastava, and A. Karkare, "Targeted example generation for compilation errors," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 327–338, 2019-11. ISSN: 2643-1572. 49, 50

[104] I. RadiÄek, "Automated feedback generation in introductory programming education : a dynamic program analysis approach," 2020. Accepted: 2020-06-27T20:32:47Z. 49, 50

[105] L. IfflÃ¤nder, A. Dallmann, P.-D. Beck, and M. Ifland, "PABS-a programming assignment feedback system," 2015-11-06. 50

[106] J. Gao, B. Pang, and S. S. Lumetta, "Automated feedback framework for introductory programming courses," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pp. 53–58, Association for Computing Machinery, 2016-07-11. 50

[107] S. Parihar, Z. Dadachanji, P. K. Singh, R. Das, A. Karkare, and A. Bhattacharya, "Automatic grading and feedback using program repair for introductory programming courses," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pp. 92–97, Association for Computing Machinery, 2017-06-28. 50, 68

[108] X. Liu, S. Wang, P. Wang, and D. Wu, "Automatic grading of programming assignments: An approach based on formal semantics," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 126–137, 2019-05. 50

[109] B. Mahesh, "Machine learning algorithms-a review," *International Journal of Science and Research (IJSR).[Internet]*, vol. 9, pp. 381–386, 2020. 54, 56

[110] G. Bonaccorso, *Machine Learning Algorithms*. Packt Publishing Ltd, 2017-07-24. 54

[111] A. Roche, "Arboles de decisión y series de tiempo.," 2009. 55

[112] S. Parsons, "Introduction to machine learning.," vol. 25, no. 3, pp. 353–353, 2010-09. Publisher: Cambridge University Press. 55

[113] B. LÃ^3$pez, "$Case - basedreasoning : Aconciseintroduction,"$ $vol.\ 7, no.\ 1, pp.\ 1 - -103, 2013 - 04 - 30. Publisher : Morgan\&ClaypoolPublishers.$ 55

[114] B. Chandra, M. Gupta, and M. Gupta, "Robust approach for estimating probabilities in naive-bayes classifier," vol. 4815, pp. 11–16, 2007-11-28. 55

[115] S. Na, L. Xumin, and G. Yong, "Research on k-means clustering algorithm: An improved k-means clustering algorithm," in *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*, pp. 63–67, 2010-04. 56

[116] U. de Oviedo, "El algoritmo k-means aplicado a clasificaciÃ^3$nyprocesamientodeimgenes.."$ 56

[117] S. Kim, J. W. Kim, J. Park, and A. Oh, "Elivate: A real-time assistant for students and lecturers as part of an online CS education platform," pp. 337–338, 2016-04-25. 56, 62

[118] lgayhardt, "Aprendizaje profundo frente a aprendizaje automÃ¡tico - azure machine learning." 56

[119] J. Cheng, I. Fostiropoulos, and B. Boehm, "GN-transformer: Fusing sequence and graph representation for improved code summarization," 2021-11-16. Number: arXiv:2111.08874. 57, 69

[120] R. Borja-Robalino, A. Monleón-Getino, and J. Rodellar, "Estandarización de métricas de rendimiento para clasificadores machine y deep learning," *Revista Ibérica de Sistemas e Tecnologias de Informação*, no. E30, pp. 184–196, 2020. 58, 59

[121] C. B. Pantaleon, L. S. Feliscuzo, and C. L. C. S. Romana, "MOOC-ready system for a course on fundamentals of programming using c: Development and analysis," in *Proceedings of the 2019 The 3rd International Conference on Digital Technology in Education*, ICDTE 2019, pp. 212–216, Association for Computing Machinery, 2019-10-25. 60

[122] "What is the k-nearest neighbors algorithm? | IBM." 61

[123] M. M. Rahman, Y. Watanobe, and K. Nakamura, "Source code assessment and classification based on estimated error probability using attentive lstm language model and its application in programming education," *Applied Sciences*, vol. 10, no. 8, p. 2973, 2020. 63, 69

[124] R. Gupta, A. Kanade, and S. Shevade, "Neural attribution for semantic bug-localization in student programs," *Advances in Neural Information Processing Systems*, vol. 32, 2019. 63

[125] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1186–1197, IEEE, 2021. 63, 69

[126] Y. Watanobe, M. M. Rahman, M. F. I. Amin, and R. Kabir, "Identifying algorithm in program code based on structural features using cnn classification model," *Applied Intelligence*, vol. 53, no. 10, pp. 12210–12236, 2023. 64

[127] D. Miao, Y. Dong, and X. Lu, "Pipe: Predicting logical programming errors in programming exercises.," *International Educational Data Mining Society*, 2020. 65, 69

[128] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: data-driven feedback generation for introductory programming exercises," in *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, pp. 481–495, 2018. 65

[129] Y. Shi, Y. Mao, T. Barnes, M. Chi, and T. W. Price, "More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code.," *International Educational Data Mining Society.*, 2021. 65, 69

[130] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*, pp. 780–791, PMLR, 2021. 65, 69

[131] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic generation of programming exercises and code explanations using large language models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, pp. 27–43, 2022. 66

[132] I. Azaiz, N. Kiesler, and S. Strickroth, "Feedback-generation for programming exercises with gpt-4," in *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, pp. 31–37, 2024. 67

[133] I. Azaiz, O. Deckarm, and S. Strickroth, "Ai-enhanced auto-correction of programming exercises: How effective is gpt-3.5?," 2023. 67

[134] N. Kiesler, D. Lohr, and H. Keuning, "Exploring the potential of large language models to generate formative programming feedback," in *2023 IEEE Frontiers in Education Conference (FIE)*, pp. 1–5, IEEE, 2023. 67

[135] L. Qiu, Y. Liu, Q. Hu, and Y. Liu, "Student dropout prediction in massive open online courses by convolutional neural networks," vol. 23, no. 20, pp. 10287–10301, 2019-10-01. 68

[136] W. Wang, H. Yu, and C. Miao, "Deep model for dropout prediction in MOOCs," in *Proceedings of the 2nd International Conference on Crowd Science and Engineering*, ICCSE'17, pp. 26–32, Association for Computing Machinery, 2017-07-06. 68

[137] W. Feng, J. Tang, and T. X. Liu, "Understanding dropouts in MOOCs," vol. 33, no. 1, pp. 517–524, 2019-07-17. Number: 01. 68

[138] S. Gulwani, I. RadiÄek, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 41–51, Association for Computing Machinery, 2014-11-11. 68, 69

[139] K. Buffardi and S. Edwards, "Reconsidering automated feedback: A test-driven approach," pp. 416–420, 2015-02-24. 68, 69

[140] J. Wang, H. Xie, F. L. Wang, L.-K. Lee, and O. T. S. Au, "Top-n personalized recommendation with graph neural networks in MOOCs," vol. 2, p. 100010, 2021-01-01. 69

[141] S. Sridhara, B. Hou, J. Lu, and J. DeNero, "Fuzz testing projects in massive courses," in *Proceedings of the Third (2016) ACM Conference on Learning @ Scale*, L@S '16, pp. 361–367, Association for Computing Machinery, 2016-04-25. 69

[142] C. G. Hidalgo Suarez, V. A. Bucheli, F. Restrepo-Calle, and F. A. Gonzalez, "A strategy based on technological maps for the identification of the state-of-the-art techniques in software development projects: Virtual judge projects as a case study," in *Advances in Computing* (J. E. Serrano C. and J. C. MartÃnez-Santos, eds.), Communications in Computer and Information Science, pp. 338–354, Springer International Publishing, 2018. 70

[143] C. A. Jones, *Assessment for Learning*. Learning and Skills Development Agency, 2005. Google-Books-ID: PqKaAAAACAAJ. 73

[144] S. Sarsa, J. Leinonen, C. Koutcheme, and A. Hellas, "Speeding up automated assessment of programming exercises," in *Proceedings of the 2022 Conference on United Kingdom & Ireland Computing Education Research*, UKICER '22, (New York, NY, USA), Association for Computing Machinery, 2022. 73, 187

[145] A. Pellini and H. Jones, "A study of adb's knowledge taxonomy," *ADB*, 2011. 82

[146] Y. Qian and J. Lehman, "Using an automated assessment tool to explore difficulties of middle school students in introductory programming," vol. 0, no. 0, pp. 1–17, 2021-01-26. Publisher: Routledge _eprint: https://doi.org/10.1080/15391523.2020.1865220. 83

[147] IBM, "Project CodeNet," 2021. 99

[148] "Codeforwin," 2023. 99

[149] "Pynative," 2023. 99

[150] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," 2018. 109

[151] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," *CoRR*, vol. abs/2009.08366, 2020. 109

[152] W. Koehrsen, "Neural network embeddings explained," 2018. 109

[153] F. Nelli, *Machine Learning with scikit-learn*, pp. 259–287. Berkeley, CA: Apress, 2023. 110

[154] T. Google, "Keras | TensorFlow core," 2022. 111

[155] L. G. Agudelo Viana and J. M. Aigneren Aburto, "Disenos de investigación experimental y no-experimental," Accepted: 2015-06-03T19:27:37Z Publisher: Universidad de Antioquia, Facultad de Ciencias Sociales y Humanas. 164

[156] L. Roest, H. Keuning, and J. Jeuring, "Next-step hint generation for introductory programming using large language models," in *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, (New York, NY, USA), p. 144â153, Association for Computing Machinery, 2024. 166

[157] R. Turcios, "Prueba de wilcoxon-mann-whitney: mitos y realidades," *Rev Mex Endocrinol Metab Nutr*, vol. 2, pp. 18–21, 2015. 178, 182

Part VI

Appendices

Appendix A

Training and testing results of the FC model without the "Correct" category.

This appendix presents the results of the fully connected model's training without the "Correct" category., including key performance metrics. The model achieved an accuracy of 0.9749 and a loss of 0.3217 on the training set, while the validation accuracy was 0.9104 with a corresponding validation loss of 0.4436 (see Figure A.1). Additionally, the confusion matrix in Figure A.2 and classification report in Figure A.3 are provided to give further insight into the model's performance across different classes.

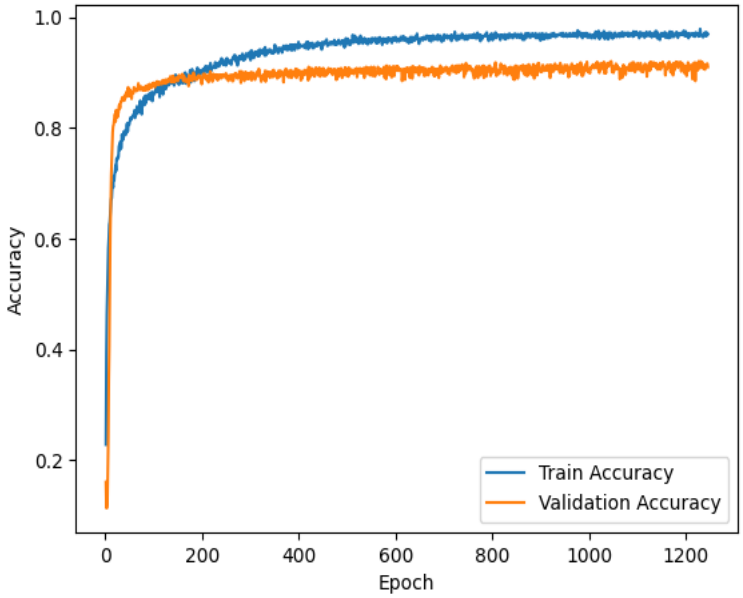

Figure A.1: Results model error classification without the "Correct" category (Accuracy).

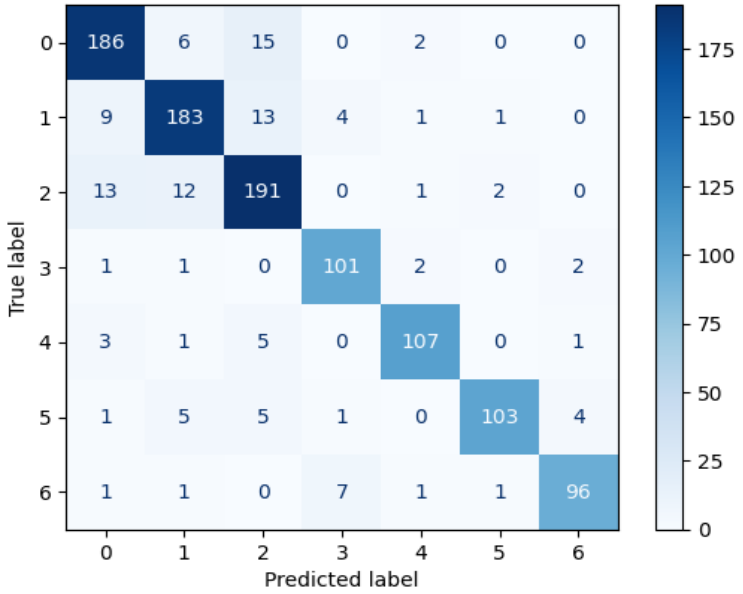

Figure A.2: Results model error classification without the "Correct" in confusion matrix.

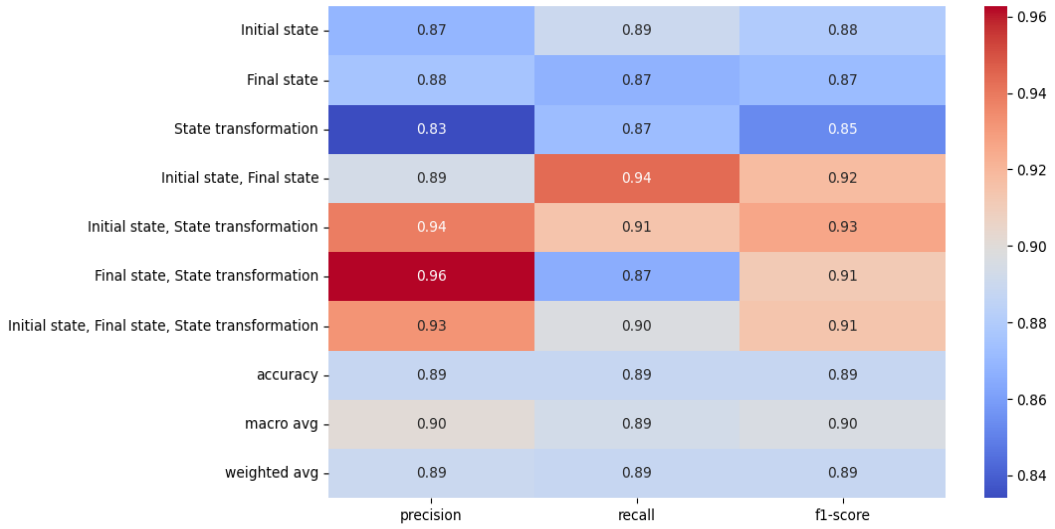

Figure A.3: Results model error classification without the "Correct" in classification report.

Appendix B

Prompt Variations and Their Impact on Feedback.

This appendix includes the three prompt variations used during testing: "Feedback with Classifier and Custom Prompt," "Feedback with Custom Prompt but Without Classifier," and "Feedback Without Classifier or Custom Prompt." These tests were performed with a range of exercises to assess the effectiveness of each prompt design. The results indicate that using a custom prompt in combination with the error classifier provides the most accurate and concise feedback, guiding students in alignment with the program correctness theory on iterations.

```
# Construct the prompt
prompt_parts = [
    {
        'role': 'system',
        'content': 'You are a helpful assistant.'
    },
    {
        'role': 'assistant',
        'content': """Analyze the student's code with respect to "while" loops. The model classifies errors as (Gries's theory):

          - Initial State Error: Initialization issues.
          - Final State Error: Loop condition issues.
          - State Transformation Error: Issues within the loop body.

          Provide concise feedback in Spanish only for the errors and evaluate if these affect the problem's invariant.
          Ensure the response is not truncated; summarize if necessary."""
    },
    {
        'role': 'user',
        'content': f"Problem: {data['statement']}\nSolution: {data['code']}\nClassification: {data['classification_result']}"
    }
]
```

Figure B.1: Custom prompt with error classification model, problem statement and student code.

```
# Construct the prompt
prompt_parts = [
    {
        'role': 'system',
        'content': 'You are a helpful assistant.'
    },
    {
        'role': 'assistant',
        'content': """Analyze the student's code with respect to "while" loops. The model classifies errors as (Gries's theory):

          - Initial State Error: Initialization issues.
          - Final State Error: Loop condition issues.
          - State Transformation Error: Issues within the loop body.

          Provide concise feedback in Spanish only for the errors and evaluate if these affect the problem's invariant.
          Ensure the response is not truncated; summarize if necessary."""
    },
    {
        'role': 'user',
        'content': f"Problem: {data['statement']}\nSolution: {data['code']}"
    }
```

Figure B.2: Custom prompt Without error classification model, problem statement and student code.

```
# Construct the prompt
prompt_parts = [
    {
        'role': 'system',
        'content': 'You are a helpful assistant.'
    },
    {
        'role': 'assistant',
        'content': """Analyze the student's code with respect to "while" loops."""
    },
    {
        'role': 'user',
        'content': f"Problem: {data['statement']}\nSolution: {data['code']}"
    }
]
```

Figure B.3: Without Custom Prompt and Without Error Classification Model: Problem Statement and Student Code.

Appendix C

Test cases for each test scenario (basic programming tasks with loop).

This appendix presents a table lists the test scenarios defined in Chapter 8, which consist of basic programming exercises using "while" loop iterations. Each exercise includes test cases specific input and output scenarios designed to evaluate the solution. The test cases cover all relevant possibilities to ensure the solution is validated accurately based on the exercise's requirements.

Task.	**Description**	**Specifications and Requirements**	**Inputs**	**Outputs**
Octal to decimal conversion	Write a Python function that returns the octal to decimal conversion using a while loop. Inside the function, ask the user to enter the number to convert.	- Note the statements and order for printing results given in the input and output test cases. - Make sure to use a while loop or iterator in the function you are developing for this task. - Remember that the function is expected to return a result. - Use this line of code to input the number: int(input("")). - Use this line of code to validate when non-allowed octal numbers are entered: return "The input contains characters that are not valid for octal."	Test case for single octal number: 17	15
			Test case for octal number with leading zeros: 076	62
			Test case for large octal number: 777	511
			Test case for octal number with trailing zeros: 1000	512

			Test case for octal number with mixed digits: 3274	1724
			Test case for octal number with zero as the only digit: 0	0
			Test case for negative octal number (not allowed): -2	The input contains characters that are not valid for octal.
			Test case for octal number not allowed: 8	The input contains characters that are not valid for octal.
Calculate LCM	Develop a Python function that calculates the Least Common Multiple (LCM) between two positive integers. Inside the function, ask the user to input the numbers to calculate the LCM.	- Follow the instructions and order for printing results given in the input and output test cases. - Make sure to use a while loop or iterator in the function you are developing for this task. - Remember that the function is expected to return a result. - Use this line of code to input the numbers: int(input("")).	Test case with the larger of two numbers: Test with two numbers where one is larger than the other. 2, 4	4
			Test case with small, simple numbers: 12, 18	36

			Test case for multiplication of numbers: Two numbers whose product equals their LCM. 4, 5	20
			Test case with prime numbers: Two numbers with no common factors other than 1. 6, 35	210
			Test case with identical numbers: 7, 7	7
			Test case with negative numbers: How negative numbers behave in LCM calculation. -8, 12	24
Prime Numbers	Write a Python function that checks if an integer is prime. Remember, a prime number is a positive integer greater than 1 that has no divisors other than 1 and itself. Inside the function, ask the user to input the number to evaluate whether it is prime.	- Follow the statements and order for printing results given in the input and output test cases. - Ensure to use a while loop or iterator in the function. - The function should return "True" if the number is prime and "False" otherwise. - Validate inputs less than 2 to return "False".	Test case with large known primes and composites: 97, 121	True, False
			Test case with small known primes and composites: 4, 2	False, True
			Special cases: 0 and 1 are not primes. 0	False
			Special case: Negative numbers should not be prime. -1	False

			Test case with composite numbers: 4	False
Factors of Number	Write a Python function that returns all the factors of a given integer, ensuring to handle negative numbers. The result should be a string where each factor is separated by a newline. Inside the function, ask the user to input the number.	- Ensure to follow the input-output format and order in the test cases. - Use a while loop or an iterator for the task, and return a string as the result with newline characters separating the factors. - Do not use lists.	Test case with small numbers having multiple factors: 12	1, 2, 3, 4, 6, 12
			Test case with a negative number: -36	1, 2, 3, 4, 6, 9, 12, 18, 36
			Test case where the number is 1: 1	1
			Test case where the number is prime: 13	1, 13
			Test case where the number is 0: 0	Empty
			Test case with large numbers having multiple factors: 120	1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120
Average of n Numbers	Write a Python function that calculates the average of a variable number of integers. Inside the function, ask the user to input the quantity of numbers to be averaged and the numbers themselves.	- Follow the order and format for input-output as given in the test cases. - Use a while loop or iterator for this task. - The result must be rounded to two decimal places.	Basic test case with positive numbers: 5, 1, 2, 3, 4, 5	3
			Test case with large numbers: 5, 1000, 2000, 3000, 4000, 5000	3000
			Test case with mixed positive and negative numbers: 4, -12, 2, -5, 1	-3.50

			Test case with mixed positive and negative numbers, including 0: 6, 0, 1, -6, 4, 21, -9	1.83
			Test case with a single number: 1, 100	100.00
			Test case with negative numbers: 4, -10, -15, -20, -25	-17.50
Series Value 2	Write a Python function that calculates the following sum of numbers from $\frac{2}{1^2}$ to $\frac{2}{n^2}$, where $n \geq 1$. Inside the function, ask the user to input the value of n.	- Follow the order and format for input-output as given in the test cases. - Use a while loop or iterator for this task. - Ensure that the result is rounded to five decimal places.	Test case with a small number: 5	2.92722
			Test case with a large number: 20	3.19233
			Test case with $n = 0$: 0	0
			Test case with negative numbers: -5	0
Greatest Common Divisor (GCD)	Write a Python function that uses a "while" loop to calculate the greatest common divisor (GCD) between two positive integers. Inside the function, ask the user to input the numbers to compute the GCD.	- Follow the order and format for input-output as given in the test cases. - Use a while loop or iterator for this task. - The function should return a result. - Use this code for input: int(input("")).	Test case where one number is a multiple of the other: 24, 12	12
			Test case with small known numbers: 12, 18	6
			Test case with large known numbers: 998, 1024	2

			Test case with prime numbers: 15, 7	1
			Test case with equal numbers: 20, 20	20
			Test case with numbers having multiple factors: 48, 36	12
Factorial	Write a Python function that returns the factorial of a number using a "while" loop.	- Follow the order and format for input-output as given in the test cases. - Use a while loop for this task. - The function should return a result. - Use this code for input: int(input("")). - For negative numbers, return "The factorial of a negative number doesn't exist."	Test case when the number is equal to 1: 1	1
			Test case with numbers yielding large results: 10	3628800
			Test case when the number is 0: 0	1
			Test case with small known numbers: 4	24
			Test case for negative numbers: -5	The factorial of a negative number doesn't exist.
Summation	Write a Python function that calculates the sum of the cubes of numbers from 1 to n, where n is greater than or equal to 1, using a while loop.	- Follow the order and format for input-output as given in the test cases. - Ensure to use a while loop. - The function return a result. - Ensure that n is greater than or equal to 1. Otherwise: print("n must be greater than or equal to 1."). - Use this code for input: int(input("")).	Test case with a small number: 5	225
			Test case with a large number: 20	44100
			Test case with n equal to 0: 0	n must be greater than or equal to 1.

			Test case with negative numbers: -2	n must be greater than or equal to 1.
Numbers from n to m	Develop a Python function that prints numbers from an initial value 'n' to a final value 'm', both inclusive, with positive increments of 'x', where 'm' is greater than 'n'. Use a while loop. Inside the function, prompt the user to input the numbers for evaluation.	- Follow the order and format for input-output as given in the test cases. - Ensure to use a while loop. - The function should return a result. - Ensure that 'm' is greater than 'n'. - Use this code for input: int(input("")). - Only integer numbers should be entered. - Print a message if 'm' is less than 'n': print("The final value m must be greater than the initial value n for positive increments."). - The output should be a string with newlines, not a list.	Basic test case, positive increment: 1, 10, 2	1, 3, 5, 7, 9
			Test case, increment of 1: 0, 5, 1	0, 1, 2, 3, 4, 5
			Test case with negative increment where initial value is greater than final: 6, 4 2	The final value m must be greater than the initial value n for positive increments.
			Test case where initial value equals final value: 4, 4, 3	The final value m must be greater than the initial value n for positive increments.
			Test case with negative numbers: -2, 10, 3	-2, 1, 4, 7, 10

Table C.1: Test cases for each test scenario (basic programming tasks with loop).

Appendix D

Classification and feedback results generated by the ImproBIt API with other LLMs.

This appendix presents a detailed analysis of the results generated by the ImproBIt automatic feedback model when integrated with different LLMs. The error classification and feedback results generated by ImproBIt when employing the GPT-3.5 Turbo, Gemini, and LLaMA models are shown below. Each section contains graphs illustrating the performance in terms of accuracy and quality of feedback each model provides.

1. GPT3.5 Turbo.

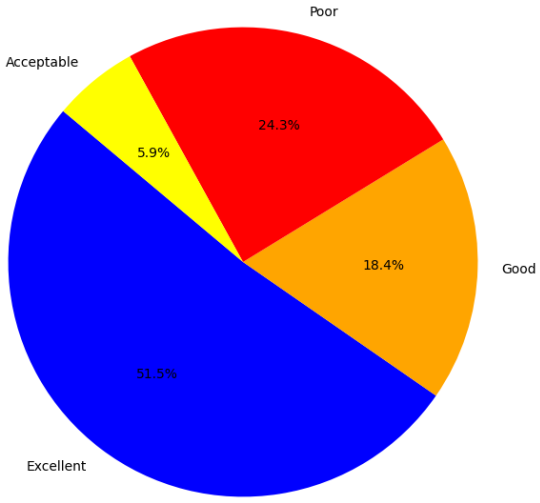

Figure D.1: Results model error classification and feedback - GPT3.5.

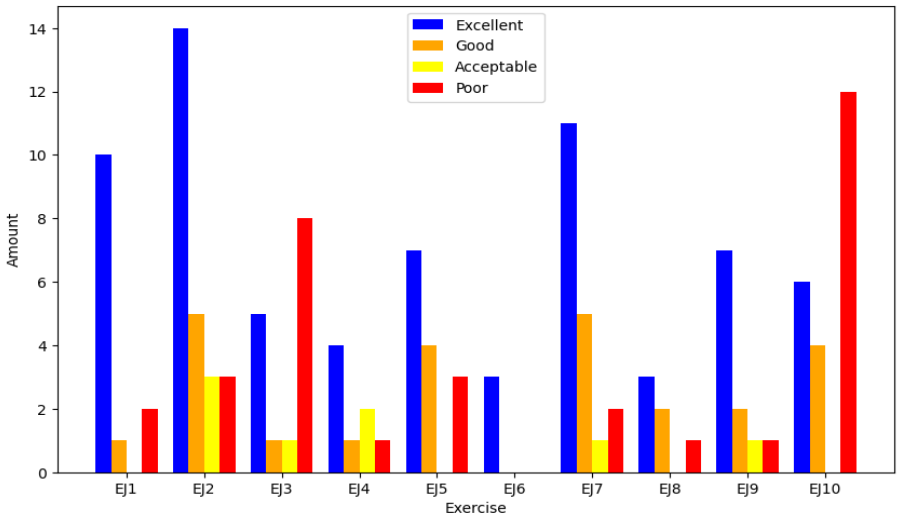

Figure D.2: Results model error classification and feedback - GPT3.5.

2. Gemini.

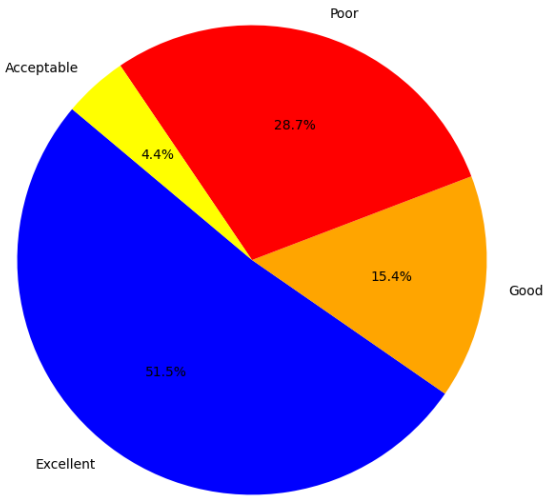

Figure D.3: Results model error classification and feedback - Gemini.

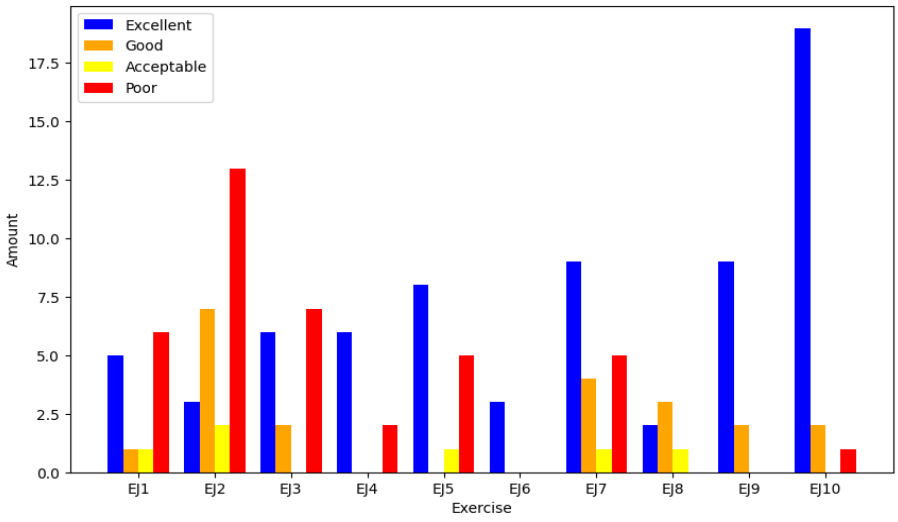

Figure D.4: Results model error classification and feedback - Gemini.

3. LLAma.

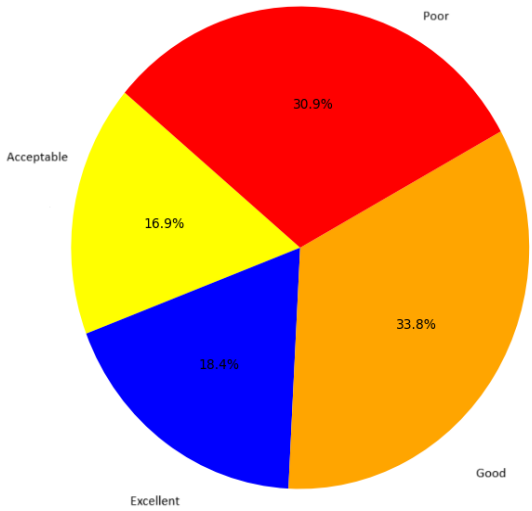

Figure D.5: Results model error classification and feedback - LLAma.

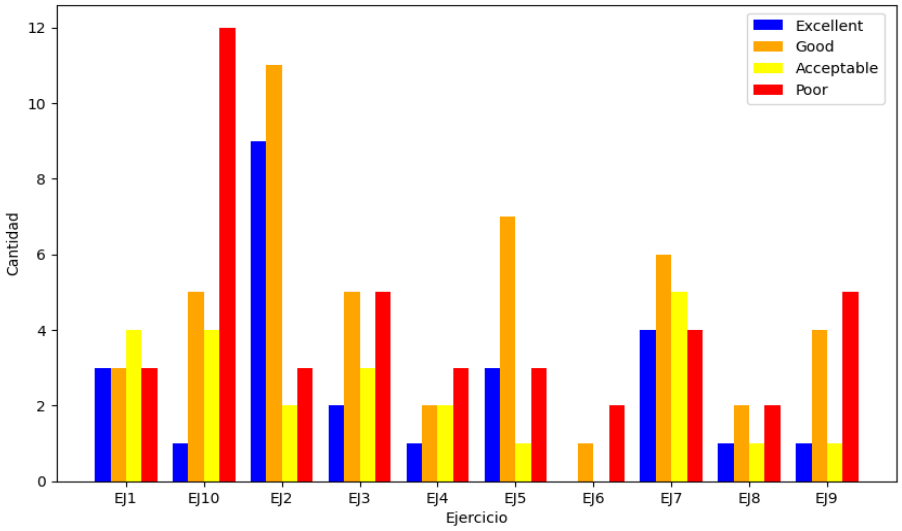

Figure D.6: Results model error classification and feedback - LLAma.

Appendix E

Survey

Introduction

This survey aims to evaluate the impact of task-level automated feedback on the learning process in programming, following the principles outlined by Hattie and Timperley (2007). It specifically focuses on error feedback in basic programming tasks.

We appreciate your willingness to participate in our experiment. Before we begin, please review the following key points:

Personal Data Processing:

- **Confidentiality of Personal Data:** The survey is anonymous.
- **Name Usage:** It will only be used to accept the form.
- **Right to Data Deletion:** You can request your data to be deleted at any time.
- **Voluntary Participation:** Participation is voluntary.

Personal Information

- Full Name. (*)
- Email Address.

Questionnaire

Please answer the following questions.

1. I currently work in a computer programming-related field:

- Yes
- No

2. Before taking this course, I had knowledge in the field of computer programming:
   - Yes, I had taken programming courses.
   - Yes, I learned programming on my own.
   - Yes, but it was very superficial knowledge.
   - No, this is my first time approaching this subject.
3. I have used LMS tools, online judges, and/or interpreters with AI support to learn programming:
   - Strongly Agree
   - Agree
   - Neither Agree nor Disagree
   - Disagree
   - Strongly Disagree

Type of Feedback

For each statement, indicate your level of agreement or disagreement.

4. I believe the feedback I receive explains why my program did not produce the expected output in failed tests.
   - Strongly Agree
   - Agree
   - Neither Agree nor Disagree
   - Disagree
   - Strongly Disagree
5. I find that the feedback I receive focuses on the most important errors that caused my program to fail testing.
   - Strongly Agree
   - Agree
   - Neither Agree nor Disagree
   - Disagree

- Strongly Disagree

6. I find that feedback helps me understand what changes I needed to make to my code to fix the failing tests.
    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

Additional information

For each statement, indicate your level of agreement or disagreement.

7. I consider that the feedback I receive on programming tasks includes useful and relevant information to improve my solutions.
    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree
8. I find that the feedback I receive about my mistakes in programming assignments helps me develop a deeper understanding of the fundamental principles of programming and put them into practice.
    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree
9. I believe the feedback I received helped me better understand the concepts of initial state, final state, and state transformation in "while" loops.
    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

Level of detail

For each statement, indicate your level of agreement or disagreement.

10. I find the feedback I receive to be timely and provide clear details about my mistakes and how to correct them.

    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

Specificity

For each statement, indicate your level of agreement or disagreement.

11. I find the feedback I receive to be accurate and tells me exactly what aspects of my code need to be fixed, helping me understand where and why I went wrong.

    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

12. I find that the feedback I receive helps me understand my mistakes and provides me with useful information to avoid making the same mistakes in the future.

    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

13. I find that the suggestions and explanations I receive in feedback allow me to correct my mistakes and improve my solutions effectively.

    - Strongly Agree

- Agree
- Neither Agree nor Disagree
- Disagree
- Strongly Disagree

Misleading information

For each statement, indicate your level of agreement or disagreement.

14. I believe the feedback I receive is always accurate and does not include misleading or incorrect information.
    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

Tone

For each statement, indicate your level of agreement or disagreement.

15. I find the feedback I receive to be friendly and encouraging, which helps me maintain a positive attitude towards my mistakes.
    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

Length

For each statement, indicate your level of agreement or disagreement.

16. I find the feedback I receive to be of an appropriate length and provide the necessary information.

- Strongly Agree
- Agree
- Neither Agree nor Disagree
- Disagree
- Strongly Disagree

Importance of feedback in the subject

For each statement, indicate your level of agreement or disagreement.

17. I believe that the feedback, suggestions or explanations received on an assignment allowed me to improve my academic performance in the subject.

    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

18. I believe that working with LMS tools or online judges that evaluate my performance in programming tasks and generate summative and/or formative feedback supports my learning.

    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

Commitment to learning

For each statement, indicate your level of agreement or disagreement.

19. I consider it very important for me to learn the topics of this subject.

    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree

- Disagree
- Strongly Disagree

20. I consider that I thoroughly review my class notes and the learning resources of this subject until I assimilate the important concepts.
    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree
21. I consider that I ask the teacher or the monitor for help when I am unable to assimilate a subject in a good way.
    - Strongly Agree
    - Agree
    - Neither Agree nor Disagree
    - Disagree
    - Strongly Disagree

Open Comment

- What aspects of the feedback do you find most useful and why?
- What suggestions do you have to improve the feedback provided by the system?

Appendix F

Results of the Quasi-Experimental Study on ImproBIt: Complete Data Table.

This appendix includes the complete table of results from the quasi-experimental study on the use of ImproBIt, measured through a data collection instrument. The table presents each question, the criterion it pertains to, and the responses obtained for each question. The response options for both the control and experimental groups are broken down, allowing for a detailed comparison of perceptions and outcomes between the groups.

Criteria	Question	Response	Quantity GE	Quantity GC
Work experience in programming	I currently work in a computer programming-related field	Yes	7	2
		No	34	38
Prior knowledge in programming and use of AI tools	Before taking this course, I had knowledge in the field of computer programming	No, this is my first approach to the topic.	20	18
		Yes, but the knowledge was very superficial.	13	19
		Yes, I had already taken programming courses.	5	2

Criteria	Question	Response	Quantity GE	Quantity GC
		Yes, I learned to program empirically.	3	1
	I have used LMS tools, online judges, and/or interpreters with AI support to learn programming	No, this is my first time using this type of tool.	28	20
		Yes, I had already used this type of tool.	13	20
Type of feedback	I believe the feedback I receive explains why my program did not produce the expected output in failed tests	Strongly agree.	11	0
		Agree.	18	17
		Neither agree nor disagree.	10	17
		Disagree.	0	6
		Strongly disagree.	2	0
	I find that the feedback I receive focuses on the most important errors that caused my program to fail testing	Strongly agree.	12	2
		Agree.	19	20
		Neither agree nor disagree.	8	16
		Disagree.	2	2
		Strongly disagree.	0	0

Criteria	Question	Response	Quantity GE	Quantity GC
	I find that feedback helps me understand what changes I needed to make to my code to fix the failing tests	Strongly agree.	18	4
		Agree.	17	15
		Neither agree nor disagree.	5	15
		Disagree.	0	6
		Strongly disagree.	1	0
Additional information	I consider that the feedback I receive on programming tasks includes useful and relevant information to improve my solutions	Strongly agree.	18	4
		Agree.	20	21
		Neither agree nor disagree.	3	14
		Disagree.	0	1
		Strongly disagree.	0	0
	I find that the feedback I receive about my mistakes in programming assignments helps me develop a deeper understanding of the fundamental principles of programming and put them into practice	Strongly agree.	21	1

Criteria	Question	Response	Quantity GE	Quantity GC
		Agree.	16	16
		Neither agree nor disagree.	4	15
		Disagree.	0	8
		Strongly disagree.	0	0
	I believe the feedback I received helped me better understand the concepts of initial state, final state, and state transformation in "while" loops	Strongly agree.	16	0
		Agree.	19	11
		Neither agree nor disagree.	5	13
		Disagree.	1	11
		Strongly disagree.	0	5
Level of detail	I find the feedback I receive to be timely and provide clear details about my mistakes and how to correct them	Strongly agree.	16	1
		Agree.	15	17
		Neither agree nor disagree.	9	19
		Disagree.	1	3
		Strongly disagree.	0	0

Criteria	Question	Response	Quantity GE	Quantity GC
Specificity	I find the feedback I receive to be accurate and tells me exactly what aspects of my code need to be fixed, helping me understand where and why I went wrong	Strongly agree.	10	2
		Agree.	22	23
		Neither agree nor disagree.	9	13
		Disagree.	0	2
		Strongly disagree.	0	0
	I find that the feedback I receive helps me understand my mistakes and provides me with useful information to avoid making the same mistakes in the future	Strongly agree.	18	1
		Agree.	18	20
		Neither agree nor disagree.	5	18
		Disagree.	0	1
		Strongly disagree.	0	0
	I find that the suggestions and explanations I receive in feedback allow me to correct my mistakes and improve my solutions effectively	Strongly agree.	14	1
		Agree.	20	13

Criteria	Question	Response	Quantity GE	Quantity GC
		Neither agree nor disagree.	7	24
		Disagree.	0	2
		Strongly disagree.	0	0
Misleading information	I believe the feedback I receive is always accurate and does not include misleading or incorrect information	Strongly agree.	12	5
		Agree.	17	23
		Neither agree nor disagree.	10	10
		Disagree.	0	1
		Strongly disagree.	2	1
Tone	I find the feedback I receive to be friendly and encouraging, which helps me maintain a positive attitude towards my mistakes	Strongly agree.	20	4
		Agree.	16	18
		Neither agree nor disagree.	5	15
		Disagree.	0	3
		Strongly disagree.	0	0
Length	I find the feedback I receive to be of an appropriate length and provide the necessary information	Strongly agree.	18	4
		Agree.	17	21
		Neither agree nor disagree.	6	12

Criteria	Question	Response	Quantity GE	Quantity GC
		Disagree.	0	2
		Strongly disagree.	0	1
Importance of feedback in the subject	I believe that the feedback, suggestions or explanations received on an assignment allowed me to improve my academic performance in the subject	Strongly agree.	15	3
		Agree.	16	19
		Neither agree nor disagree.	10	18
		Disagree.	0	0
		Strongly disagree.	0	0
	I believe that working with LMS tools or online judges that evaluate my performance in programming tasks and generate summative and/or formative feedback supports my learning	Strongly agree.	20	6
		Agree.	15	25
		Neither agree nor disagree.	6	8
		Disagree.	0	1
		Strongly disagree.	0	0
Commitment to learning	I consider it very important for me to learn the topics of this subject	Strongly agree.	26	11

Criteria	Question	Response	Quantity GE	Quantity GC
		Agree.	13	20
		Neither agree nor disagree.	1	7
		Disagree.	1	1
		Strongly disagree.	0	1
	I consider that I thoroughly review my class notes and the learning resources of this subject until I assimilate the important concepts	Strongly agree.	9	5
		Agree.	17	19
		Neither agree nor disagree.	12	14
		Disagree.	3	2
		Strongly disagree.	0	0
	I consider that I ask the teacher or the monitor for help when I am unable to assimilate a subject in a good way	Strongly agree.	7	4
		Agree.	17	16
		Neither agree nor disagree.	10	11
		Disagree.	6	6
		Strongly disagree.	1	3

Table F.1: ImproBIt evaluation survey results.