PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ

DAVI BERNARDO SILVA

A CLUSTERING-BASED COMPUTATIONAL MODEL TO GROUP STUDENTS WITH SIMILAR PROGRAMMING SKILLS FROM AUTOMATIC SOURCE CODE ANALYSIS USING NOVEL FEATURES

SCIENTIA, VITA ET FIDES

DOUTORADO EM
INFORMÁTICA
PUCPR

CURITIBA
2023

DAVI BERNARDO SILVA

A CLUSTERING-BASED COMPUTATIONAL MODEL TO GROUP STUDENTS WITH SIMILAR PROGRAMMING SKILLS FROM AUTOMATIC SOURCE CODE ANALYSIS USING NOVEL FEATURES

Thesis presented to the Programa de Pós-Graduação em Informática (PPGIa) of the Pontifícia Universidade Católica do Paraná as a partial requirement to obtain the title of Ph.D. degree in Computer Science.

Pontifícia Universidade Católica do Paraná - PUCPR

Programa de Pós-Graduação em Informática - PPGIa

Supervisor: Prof. Dr. Carlos N. Silla Jr.

Co-supervisor: Profa. Dra. Deborah Ribeiro Carvalho

Curitiba – PR, Brasil

2023

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central
Luci Eduarda Wielganczuk – CRB 9/1118

S586c
2023

Silva, Davi Bernardo
A clustering-based computational model to group students with similar programming skills from automatic source code analysis using novel features / Davi Bernardo Silva ; supervisor: Carlos N. Silla Jr. ; co-supervisor: Deborah Ribeiro Carvalho. – 2023.
172 f. : il. ; 30 cm

Tese (doutorado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2023
Bibliografia: f. 155-172

1. Informática. 2. Análise de conglomerados. 3. Ensino auxiliado por computador. 4. Programação (Computadores) – Estudo e ensino. I. Silla Júnior, Carlos Nascimento. II. Carvalho, Deborah Ribeiro. III. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. IV. Título.

CDD. 21. ed. – 004

PUCPR
GRUPO MARISTA

Pontifícia Universidade Católica do Paraná
Escola Politécnica
Programa de Pós-Graduação em Informática

Curitiba, 9 de novembro de 2023.

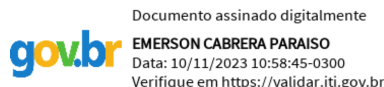
87-2023

DECLARAÇÃO

Declaro para os devidos fins, que **DAVI BERNARDO SILVA** defendeu a tese intitulada **A Clustering-Based Computational Model to Group Students with Similar Programming Skills From Automatic Source Code Analysis Using Novel Features"**, na área de concentração Ciência da Computação no dia 06 de julho de 2023, no qual foi aprovado.

Declaro ainda, que foram feitas todas as alterações solicitadas pela Banca Examinadora, cumprindo todas as normas de formatação definidas pelo Programa.

Por ser verdade firmo a presente declaração.



Prof. Dr. Emerson Cabrera Paraiso
Coordenador do Programa de Pós-Graduação em Informática

Acknowledgements

I write with gratitude to everyone who somehow participated in this construction process (of the person involved).

First of all, I thank my mother, Giovana, a warrior woman and the most important person in my life. I thank my father, Francisco, the main enthusiast of his children's personal growth. I thank my grandmothers who played the role of mother so many times. To my grandmother *Tele* who never measured efforts for my good. To my grandmother Maria, a giant woman, who even in her simplicity taught the wisest.

I thank my family for understanding that this time was not easy for me either. I would like to highlight my uncle Geraldo, who supported my studies. To my godparents, Marcos and Tereza, who welcomed me with their children, like a son. To my confirmation godfather, Dalton, who with his trajectory became my inspiration. To my sisters, Dalila and Mariana, blood of my blood. To my brother Rafael, who has taught me many things about life. To my brother Leandro, who introduced me to this fascinating world of computing. I thank my girlfriend, Lidiane, who was by my side, even in the most difficult moments.

I thank my former advisors. To professors Francisco Pereira Junior (*Theskão*) and Henrique Shishido (*Shidão*), great friends who played a fundamental role in my academic choices. To the professor, Dr. André Takeshi Endo, who with his teachings enabled me to face a new challenge.

I thank my advisors. To professors Dr. Carlos Nascimento Silla Jr. and Dra. Deborah Ribeiro Carvalho, who guided me during my studies and led me to this thesis. I am grateful for the trust and bond of friendship that we have developed over this time.

I thank my doctoral examining committee. To professors Dra. Simone Lima, Dr. Clodis Boscarioli, Dr. Júlio César Nievola and Dr. Emerson Paraíso for their insightful comments, which enriched and brought a new horizon to the research.

I thank the professors at PUCPR, who shared their precious knowledge during the development of their classes. I thank the friends of the IFSC, for their support and participation in this very important stage of my life.

Thank you all ♡

Abstract

In a programming course, students develop various source code tasks. Using these tasks to track student progress can provide valuable insights into their strengths and weaknesses in each learning topic. This practice allows the teacher to intervene in learning from the first weeks of class and thus maximize student gains. However, manually analyzing these tasks poses a significant challenge for teachers due to the time and effort required. To address this challenge, our research aims to automatically group students with similar programming skills based on the automatically analysis of their submitted source codes. To achieve this objective, we propose novel features that represent students' progress in different learning topics. Our research follows an applied approach and employs an experimental procedure. Firstly, we prepared a database of over 650 real-world source code tasks written in the C language. These tasks covered five learning topics taught throughout an academic period. Next, we defined a set of 21 source code features to explicitly extract for each learning topic. In the subsequent step, we pre-processed the database and extracted the proposed features. Finally, we employed a hierarchical clustering algorithm to group the students based on their source code tasks. Our model expects as input the source code tasks that the students developed. Besides, our model allows the teacher to input the solution to each task, enabling analysis and comparison. As a result, we obtained four distinct groups of students. Our findings provide individualized analysis of the student distribution within each group and calculate the midpoint of each cluster. The midpoint calculation offers a summarized evaluation of the strengths and weaknesses exhibited by each group of students. This approach supports continuous student monitoring throughout the academic period, empowering teachers to design tailored tasks without relying on specific programming environments. We believe that our results will assist teachers in making informed pedagogical decisions and addressing the specific needs of each group of students. By automating the analysis of source code tasks, our approach reduces the burden on teachers, enhances student progress tracking, and promotes effective educational interventions.

Keywords: Classroom feedback systems, clustering, computer science education, feature engineering, teaching programming.

Resumo

Em um curso de programação, os estudantes desenvolvem diversas tarefas de código-fonte. Utilizar essas tarefas para acompanhar o progresso dos estudantes pode fornecer valiosas informações sobre os pontos fortes e os pontos fracos dos estudantes em cada tópico de aprendizagem. Essa prática permite que o professor intervenha no aprendizado desde as primeiras semanas de aula, maximizando assim os ganhos dos estudantes. No entanto, analisar manualmente essas tarefas representa um desafio significativo para os professores devido ao tempo e esforço envolvidos. Para enfrentar esse desafio, o objetivo desta pesquisa é agrupar automaticamente os estudantes com habilidades de programação semelhantes com base na análise dos códigos-fonte enviados pelos estudantes. Para atingir esse objetivo, foi necessário propor novas características que representam o progresso dos estudantes em diferentes tópicos de aprendizagem. Esta pesquisa segue uma abordagem aplicada e emprega um procedimento experimental. Primeiramente, foi preparado um banco de dados com mais de 650 tarefas de código-fonte do mundo real escritas na linguagem C. Essas tarefas abrangem cinco tópicos de aprendizagem ensinados ao longo de um período acadêmico. Em seguida, foi definido um conjunto de 21 características de código-fonte a ser extraído especificamente de cada tópico de aprendizagem. Na etapa subsequente, foi realizado o pré-processamento do banco de dados e as características propostas anteriormente foram extraídas. Por fim, foi utilizado um algoritmo de agrupamento hierárquico para agrupar os estudantes com base em suas tarefas de código-fonte. O modelo computacional espera como entrada as tarefas de código-fonte que os estudantes desenvolveram. Além disso, o modelo permite que o professor insira a solução de cada tarefa, que pode ser usada para fins de análise e comparação. Como resultado, obteve-se quatro grupos distintos de estudantes. As descobertas fornecem análises individualizadas da distribuição dos estudantes em cada grupo e calculam o ponto médio de cada cluster. O cálculo do ponto médio oferece uma avaliação sumária dos pontos fortes e os pontos fracos de cada grupo de estudantes. Essa abordagem suporta o monitoramento contínuo dos estudantes ao longo do período acadêmico, capacitando os professores a projetar tarefas personalizadas sem depender de ambientes de programação específicos. Acredita-se que estes resultados auxiliarão os professores na tomada de decisões pedagógicas embasadas e no atendimento às necessidades específicas de cada grupo de estudantes. A automação da análise das tarefas de código-fonte reduz a carga de trabalho dos professores, aprimora o acompanhamento do progresso dos estudantes e promove intervenções educacionais eficazes.

Palavras-chave: Sistemas de retorno de sala de aula, agrupamento, educação em computação, engenharia de características, ensino de programação.

List of Figures

Figura 1 – Distribution of content covered by years 36
Figura 2 – Distribution of pedagogical strategies by years 37
Figura 3 – Distribution of teaching support tools by year 41
Figura 4 – An example of k-means clustering algorithm 49
Figura 5 – Representation of the hierarchical tree structure 50
Figura 6 – An example of representing a hierarchical clustering 52
Figura 7 – Defining proximity between clusters 52
Figura 8 – Clustering representation using Single Linkage 53
Figura 9 – Clustering representation using Complete Linkage 53
Figura 10 – Clustering representation using Average Linkage 53
Figura 11 – Clustering representation using ward link 54
Figura 12 – Source code example that averages three grades 61
Figura 13 – Control Flow Chart for *calc_result()* function 65
Figura 14 – Control Flow Chart for the other functions of the source code . . . 66
Figura 15 – Example of a nesting tree 67
Figura 16 – Mathematical description of the cohesion metrics set 70
Figura 17 – Source code example that averages three grades 89
Figura 18 – Execution flow of our computational model 90
Figura 19 – Conceptual diagram of the teaching process 93
Figura 20 – An example of a source code statement from T1 96
Figura 21 – Example of source code solution 121
Figura 22 – Source code example where the student does not meet the topic . 122
Figura 23 – Relationship between clusters and features for the T1 130
Figura 24 – Relationship between clusters and features for the T2 131
Figura 25 – Relationship between clusters and features for the T3 132
Figura 26 – Relationship between clusters and features for the T4 134
Figura 27 – Relationship between clusters and features for the T5 135

List of Tables

Table 1 – Number of publications selected and classified by year 33
Table 2 – Definition of ICs for selection of publications 34
Table 3 – Overview of Results 35
Table 4 – Cyclomic Complexity Scale defined in Anderson (2004) 65
Table 5 – Weighted Methods per Class Scale 67
Table 6 – Where is this research compared to the state of the art? 84
Table 7 – Description of the source code task database 95
Table 8 – Correspondence between learning topics and features 104
Table 9 – Example of feature extraction for Figure 21 121
Table 10 – Example of feature extraction for Figure 22 122
Table 11 – Internal validation between clusters 126
Table 12 – Submission rate by cluster 127
Table 13 – Model output after extracting features for the 41 students 129
Table 14 – Clusters quality validation for a sample of students in T1 136
Table 15 – Summary of the *Kruskal-Wallis H-test* statistical test 137
Table 16 – Summary of the *Spearman's rho* correlation between the features . 139
Table 17 – Midpoint obtained from the grouping in T1 140
Table 18 – Midpoint obtained from the grouping in T2 141
Table 19 – Midpoint obtained from the grouping in T3 142
Table 20 – Midpoint obtained from the grouping in T4 143
Table 21 – Midpoint obtained from the grouping in T5 144

List of abbreviations and acronyms

AST	Abstract Syntax Tree
BoW	Bag-of-Words
CC	Cyclomatic Complexity
CFC	Control Flow Chart
CS1	Introductory Computer Science Courses
CS2	Basic Data Structures Courses
G	Group
HC	Halstead Complexity
IDE	Integrated Development Environment
IR	Information Retrieval
LOC	Lines of Code
CGPA	Cumulative Grade Point Average
MLOC	Lines of Code by Method
NAdd–s	Number of Simple Address
NAdd–ds	Number of Array Address
NBD	Nested Block Depth
NC	Number of Calls
NCRec	Number of Recursive Functions Calls
NF	Number of Functions
NFRec	Number of Recursive Functions
NFree	Number of Free
NIFPar	Number of Parameters in Conditional
NMalloc	Number of Malloc

NOA	Number of Attributes
NOM	Number of Methods
NOP	Number of Parameters
NOSM	Number of Static Methods
NOSA	Number of Static Attributes
NOC	Number of Classes
NOPK	Number of Packages
NOI	Number of Interfaces
NP	Number of Parameters
NPt–s	Number of Simple Pointers
NPt–ds	Number of Array Pointers
NRRec	Number of Recursive Returns
NRNRec	Number of Non-Recursive Returns
NSizeof	Number of Sizeof
NStr	Number of Structs
NStrC	Number of Struct Calls
NStrI	Number of Struct Instances
NStrM	Number of Struct Members
NStrT	Number of Struct Typedefs
OOP	Object-Oriented Programming
RQ	Research Question
T	Learning Topic
TF	Terms Frequency
TFIDF	Term Frequency-Inverse Document Frequency
WMC	Weighted Methods per Class

Contents

1 INTRODUCTION **23**
1.1 Problem Statement and Justification for the Research **24**
1.2 Objectives **26**
1.3 Thesis Hypothesis **27**
1.4 Outcomes and Contributions **27**
1.4.1 Scientific Contributions 27
1.4.2 Technical Contributions 28
1.4.3 Social Contributions 28
1.5 Document Outline **29**

2 RECENT STUDIES ABOUT TEACHING ALGORITHMS AND DATA STRUCTURES **31**
2.1 Study Setting **32**
2.1.1 Definition of Research Questions 32
2.1.2 Selection of Publications 33
2.1.3 Data Extraction 34
2.1.4 Threats to Validity 34
2.2 Analysis of Results **35**
2.2.1 Covered Contents 35
2.2.2 Pedagogical Strategies 36
2.2.3 Support Tools 40
2.3 Final Remarks **43**

3 DATA CLUSTERING ANALYSIS **45**
3.1 Distance Measures between Objects **45**
3.2 Types of Clustering Algorithms **47**
3.2.1 Partitive Clustering 47
3.2.2 Hierarchical Clustering 49
3.2.2.1 Divisive hierarchical clustering 49
3.2.2.2 Agglomerative hierarchical clustering 51
3.2.2.3 Hierarchical clustering graphical representation 51
3.2.2.4 Distance measures between hierarchical clusters 52
3.3 Model Evaluation **54**
3.4 Final Remarks **55**

4 FEATURE ENGINEERING FOR SOURCE CODE ANALYSIS **57**
4.1 Fundamentals of Source Code Analysis **58**
4.1.1 Information Retrieval 58
4.1.2 Term Frequency Normalization 59
4.1.3 Regular Expressions 60
4.2 Software Quality Metrics **60**
4.2.1 Size Metrics 62
4.2.2 Complexity Metrics 64
4.2.2.1 Cyclomatic Complexity 64
4.2.2.2 Weighted Methods per Class 65
4.2.2.3 Nested Block Depth 67
4.2.2.4 Halstead Complexity 68
4.2.3 Coupling Metrics 68
4.2.4 Cohesion Metrics 69
4.3 Best Coding Practices **69**
4.3.1 Refactoring Functions and Variables 70
4.3.2 Simplifying Conditional Logic 72
4.3.3 Refactoring Parameters 73
4.4 Final Remarks **73**

5 RELATED WORK **75**
5.1 Students' Grade **76**
5.2 Students' Performance **77**
5.3 Students' Grouping **79**
5.4 Aspects of the Approaches **80**
5.5 Final Remarks **85**

6 METHODOLOGICAL APPROACH **87**
6.1 Research Characterization **87**
6.2 Research Structuring **88**
6.2.1 Data Selection 89
6.2.2 Data Preprocessing 91
6.2.3 Data Transformation 91
6.2.4 Data Mining 92
6.2.5 Interpretation and Evaluation 92
6.3 Experimental Settings **93**
6.3.1 Research Questions 94
6.3.2 Database Description 95
6.3.3 Clustering Algorithm 96

6.3.3.1 Distance measures between students . . . 97
6.3.3.2 Distance measures between hierarchical clusters . . . 97
6.3.4 Model Interpretation and Evaluation . . . 97
6.3.4.1 Interpretation of extracted features . . . 97
6.3.4.2 Cluster internal validation . . . 98
6.3.4.3 Definition of the number of clusters. . . . 98
6.3.4.4 Statistical analysis . . . 99
6.4 Final Remarks . . . 100

7 PROPOSED FEATURES . . . 103
7.1 Features for T1 (Functions by Value) . . . 103
7.1.1 Number of Functions . . . 104
7.1.2 Number of Parameters . . . 105
7.1.3 Number of Calls . . . 106
7.1.4 Number of Returns . . . 107
7.2 Features for T2 (Functions by Reference) . . . 108
7.2.1 Number of Simple Pointers . . . 108
7.2.2 Number of Simple Address . . . 109
7.2.3 Number of Array Pointers . . . 109
7.2.4 Number of Array Addresses . . . 110
7.3 Features for T3 (Data Structures) . . . 111
7.3.1 Number of Structs . . . 112
7.3.2 Number of Struct Members . . . 112
7.3.3 Number of Struct Typedefs . . . 113
7.3.4 Number of Struct Instances . . . 113
7.3.5 Number of Struct Calls . . . 114
7.4 Features for T4 (Recursive Functions) . . . 115
7.4.1 Number of Recursive Functions . . . 115
7.4.2 Number of Recursive Functions Calls . . . 116
7.4.3 Number of Parameters in Conditional . . . 117
7.4.4 Number of Recursive Returns . . . 117
7.4.5 Number of Non-Recursive Returns . . . 119
7.5 Features for T5 (Dynamic Allocations) . . . 119
7.5.1 Number of Malloc . . . 120
7.5.2 Number of Sizeof . . . 120
7.5.3 Number of Free . . . 120
7.6 Feature Extraction Example . . . 121
7.7 Final Remarks . . . 122

8 ANALYSIS OF RESULTS . . . 125

8.1 Clustering Settings **125**
8.2 Students' Grouping **126**
8.2.1 Individual programming skills for the T1 130
8.2.2 Individual programming skills for the T2 131
8.2.3 Individual programming skills for the T3 132
8.2.4 Individual programming skills for the T4 133
8.2.5 Individual programming skills for the T5 133
8.2.6 Clusters quality validation 135
8.3 Statistical Test Results **136**
8.4 Correlation between the Features **138**
8.5 Overview of Students' Grouping **138**
8.5.1 Overview for the T1 140
8.5.2 Overview for the T2 141
8.5.3 Overview for the T3 142
8.5.4 Overview for the T4 143
8.5.5 Overview for the T5 144
8.6 Final Remarks **145**

9 CONCLUSIONS **149**
9.1 Research Limitations and Future Work Directions **151**

REFERENCES **155**

Chapter 1

INTRODUCTION

Nowadays, computational software permeates the daily lives of virtually every individual, either directly or indirectly. From the most specialized professions to routine activities like using a smartphone or installing electronic devices, computing technology plays a pivotal role. Concurrently, the influence of computing extends its impact on employment opportunities, as emphasized by Barr and Stephenson (2011). The ubiquitous presence of computational software underscores its significance as a driving force in modern society, shaping various aspects of human life and professional endeavors. Research studies by Schwab (2018) show that in 2025 machines will perform more than half of the tasks humans perform in their current jobs. The impact is that a large part of the paid jobs today will be replaced by new jobs requiring technological skills. The computational power and rapid evolution of algorithms in the workplace can create 133 million new functions in the next two years. However, emerging challenges are related to the qualification of these professionals to adapt to the new job market.

This popularization of access to technology and the various employment opportunities have made higher education in computing a preference for a considerable part of future university students (CAMP et al., 2017). In this context, programming courses have been present since the beginning of studies and characterize fundamental student success skills in subsequent phases. Introductory subjects to programming are conventionally referred to by the term Introductory Computer Science (CS1) course, designed by the computing association ACM Digital Library (HERTZ, 2010; ACM, 2013). This approach is usually the student's first contact with programming environments and languages. Therefore, algorithmic concepts and logical reasoning are the first subjects to be studied. The term algorithm is commonly defined as a sequence of ordered and executable steps that lead to the resolution of a given problem (CORMEN, 2012). The algorithms can initially be expressed in natural language for educational purposes, helping beginners better understand. However, natural language has innate ambiguities and inaccuracies; therefore, programming languages are used to develop software solutions. Typically, source code related

tasks are proposed for students to develop problem-solving skills, learn basic programming concepts, and translate the problem description into a programming language, following the necessary syntax and semantics (IEEE, 2013). The common programming languages used at this learning stage are C, Python, and Java (SPACCO et al., 2015).

Teaching introductory programming generally requires students to be able to design programs and solve simple problems. Throughout the course, students acquire different levels of knowledge. Some students readily understand the elementary concepts and can receive more advanced learning topics, while others still need assistance in key topics (CARTER et al., 2010). Both groups need attention. In the first case, the teacher should direct the learning to continue progressing and prevent the student from becoming disinterested in the course. In the second case, which is the most critical, it will require even more effort for the student to keep learning and not give up on the course.

Then, the new course's excitement is often interrupted by the obstacles present in these early experiences with programming for many students. The difficulty in developing the expected logical reasoning can lead to the course's failure or drop out. Research shows that approximately 30% of students fail the course (BENNEDSEN; CASPERSEN, 2007; WATSON; LI, 2014; LUXTON-REILLY et al., 2019). Regardless, drop out rates can reach 50% of students (KINNUNEN; MALMI, 2006). The most frequent reasons for students to make this decision are lack of time and lack of motivation, both directly affected by the difficulty of the course. For these reasons, teachers are concerned about making the learning experience more engaging, and researchers are increasingly investigating methods (VIHAVAINEN; AIRAKSINEN; WATSON, 2014), and tools to support these teachers (KEUNING; JEURING; HEEREN, 2018).

1.1 Problem Statement and Justification for the Research

Given the academic and professional importance programming learning to students, as well as all the challenges involved in the process, giving a comprehensive review of all the related work is beyond the scope of this research. However, some very interesting and inspiring papers provide a good overview of this area while discussing important issues. Over the years, different authors have made joint efforts to disseminate experiences and discoveries in teaching programming. Robins, Rountree and Rountree (2003) investigated programming learning processes for beginners and practical implications for teachers. Pears et al. (2007) researched the teaching of introductory programming with a specific focus on curriculum, pedagogy, languages, and tools. Since then, differences in learning have been debated in terms of levels of programming understanding, programming paradigms, and knowledge dissemination strategies.

In this sense, teachers also face many inherent challenges in teaching programming to beginning students. Among the challenges observed, the biggest is the lack of methods and tools available to understand student development and support teaching strategies (MEDEIROS; RAMALHO; FALCãO, 2019). So a greater effort is devoted to motivational theories, learning topics, and methods used for teaching programming (SCAICO; SCAICO; QUEIROZ, 2018). For example, Maia, Serey and Figueiredo (2017) investigated how teaching styles affect programming learning. They highlighted that styles that present active, reflective, and intuitive characteristics positively affect learning. For a comprehensive review of automated feedback generation for programming exercises, we refer the reader to the paper (KEUNING; JEURING; HEEREN, 2018). Research carried out in Brazil shows that active approaches are predominant, such as gamified classes, the use of robotics, and block language, to encourage student motivation (SANTOS; ARAUJO; BITTENCOURT, 2018).

There has been a growing concern about how to engage students more during classes. In this context, several novel pedagogical approaches have been developed under the "Active Learning Pedagogical Approaches". Felder and Brent (2009) define active learning as everything students do in a course that goes beyond just watching, listening, and taking notes. There are several approaches of this type, such as Think-Pair-Share (KOTHIYAL et al., 2013), 300 (FRANGELLI, 2015), Pair-Programming (HANKS et al., 2011), Peer Instruction (FAGEN; CROUCH; MAZUR, 2002), Problem-Based Learning (HMELO-SILVER, 2004), Project-Based Learning (KOKOTSAKI; MENZIES; WIGGINS, 2016), Gamification (ZAINUDDIN et al., 2020), to name a few. To apply some of these methodologies in programming classes is necessary to have a way of grouping students by some criteria, such as their programming skills. However, to the best of our knowledge, no approach or tool would allow us to do that using the students' source code in an automatic way.

The increase in the number of students enrolled, for example, leads to overcrowded classes that limit the monitoring of activities and make personalized intervention difficult (SZABO; FALKNER, 2017). Some research carried out indicates that immediate teacher intervention is vital for student performance (SZABO; FALKNER, 2017) (FONSECA; MACEDO; MENDES, 2018) (PETERSEN et al., 2016). In contrast, obtaining information about student progress from manually grading source code tasks is humanly challenging. The amount of source code produced by students during the term requires several hours of work by the teacher to analyze them.

For these reasons, researchers have continuously tried to create methodologies and develop tools to support teaching programming (SILVA et al., 2019). Related literature presents tools for automatic correction (ALFARO; SHAVLOVSKY, 2014) and plagiarism detection (YAN et al., 2018) to platforms for block programming (RODRÍGUEZ; PRICE;

BOYER, 2017), in addition to other technological approaches (ZAVALA; MENDOZA, 2018). Recent studies have started to employ machine learning techniques within the developed tools. One technique used in machine learning to make discoveries from data is clustering. Data clustering is an unsupervised machine learning technique for creating groups of objects. One example of such an approach in teaching programming is Aottiwerch and Kokaew (2018) work, where a clustering algorithm is used to suggest programming pairs from a qualitative questionnaire answered by students. Another example is the research of Ahadi et al. (2017), where they use a clustering model to analyze performance and consistency in programming tasks.

However, to the best of the author's knowledge, there seems to be no report in the literature about an approach that allows monitoring students' progress in such a way that it can guide groups with similar difficulties and challenges based on the analysis of their source code solutions. If the teacher can identify the programming skills the students developed in the first few weeks of class, he can make specific pedagogical decisions. With the support of machine learning techniques, the students' source code solutions can be automatically grouped according to the similarity of the lines of source code. After that, the teacher can identify the knowledge levels of each group and provide targeted assistance. An example would be the choice of a teaching methodology that would adapt to the needs identified for each group of students. Therefore, we observe high research potential in this theme.

1.2 Objectives

Our main research objective is to automatically group students with similar programming skills based on the analysis of their submitted source codes. In order to achieve this, we also had to propose novel features for the source code analysis that are related to the learning topics. This main objective refers to the following specific objectives:

(a) To define machine learning techniques to analyse the students' source code tasks.

(b) To propose source code analysis features that consider the different learning topics.

(c) To develop a solution to monitor student progress based on the techniques studied.

(d) To evaluate the use of this solution in different experiments.

1.3 Thesis Hypothesis

The development of this doctoral thesis is motivated by the lack of scientific studies that use machine learning techniques to support monitoring the progress of programming students. In this context, we present our hypotheses.

- Null Hypothesis (H_0): It is possible to group students with related programming skills from the automatic analysis of their source code tasks.
- Alternative Hypothesis (H_1): It is not possible to group students with related programming skills from the automatic analysis of their source code tasks.

It is important to note that the definition of the best clustering algorithm is not part of the scope of this research. Also, predicting students' final grades is not in our best interest, but we want to support the teacher's pedagogical decisions regarding the development of each group of students better.

1.4 Outcomes and Contributions

The research developed in this thesis has lead to the following contributions: *(i)* scientific contribution in filling knowledge gaps with the proposition of new features; *(ii)* technical contribution in developing new technology; and *(iii)* social contribution in problem-solving, and improvement of analysis conditions.

1.4.1 Scientific Contributions

The scientific relevance of the research is creating a set of new source code features specific to the learning topics of a programming course. From the features created, the application of machine learning techniques allows the grouping of students. In addition, the definition of a midpoint as a reference value to evaluate the clustering results.

Throughout the development of this doctoral thesis, some full papers have been written and published in scientific events in the area.

- BERNARDO SILVA, D.; DECONTO, D. S.; AGUIAR, R. L., and SILLA, C. N.. 2019. **Recent studies about teaching algorithms (CS1) and data structures (CS2) for computer science students.** In Proceedings Frontiers in Education. 49th Annual Conference (FIE '19). IEEE, Cincinnati, OH, USA.

- BERNARDO SILVA, D.; SILLA, C. N.. 2020. **Evaluation of students programming skills on a computer programming course with a hierarchical clustering algorithm.** In Proceedings Frontiers in Education. 50th Annual Conference (FIE '20). IEEE, Uppsala, Sweden, UK.

- BERNARDO SILVA, D.; RIBEIRO CARVALHO, D.; SILLA, C. N.. 2023. **A clustering-based computational model to group students with similar programming skills from automatic source code analysis using novel features.** In IEEE Transactions on Learning Technologies (TLT).

1.4.2 Technical Contributions

The technical relevance of the research is developing a computational model to group students according to their programming skills. The computational model can automatically group students based on their submitted source code tasks and allows the teacher to track students' progress across the learning topics of a programming course.

The source code of our computational model is available in a public repository for greater detail of how its construction occurred[1]. This source code also helps to understand the technical aspects our computational model. Considering that it is difficult to find available implementations to handle the preprocessing of source code tasks and student grouping, this can be considered as another technical contribution of this research.

1.4.3 Social Contributions

Considering the use of the proposed method by teachers, it has the added contribution of allowing the teacher to better visualise and identify the limitations of the learning process based on the organization of groups of students, and thus define the focus of the learning topics of a programming course.

Our approach presents a visualization of what is happening within each group of students. Programming classes are often large, and this makes it difficult to understand what skills students are developing. Our model allows the teacher to visualize these skills without opening any of the students' source code assignments. Based on this, the teacher can define different pedagogical strategies to level the students.

[1] The public repository of the computational model is available at: https://github.com/davibernardos/SCAFX.git

1.5 Document Outline

This doctoral thesis is organized into eight chapters, including this introductory chapter that presents a contextualization of teaching-learning programming. Also, we describe some of the problems about the teaching-learning programming that present as research opportunity. Next, we present our objectives and our thesis hypothesis. Finally, we present the contributions of our research, such as social, scientific, and technological.

Chapter 2 presents a mapping of recent studies in SIGCSE focusing on the teaching of programming (CS1) and data structures (CS2) to university students in computer science courses. Three key findings emerged from this review: the characterization of course contents, the identification of pedagogical strategies, and the categorization of supporting tools. This concise literature review situates us within the field of programming education and provides valuable insights to attack approaches to clustering and feature extraction.

Chapter 3 presents part of the fundamental bibliography, where we explain the main concepts inherent in unsupervised machine learning. Initially, the data clustering task is contextualized, and then the main clustering algorithms are described. In the sequence, two categories of distance measures are presented that can be used in the data clustering. The first category is used to calculate the distance between objects in a cluster, and the second category is used to calculate the distance between clusters. Finally, the possible assessment models for a cluster are presented.

Chapter 4 presents the second part of the fundamental bibliography, where we explain the main concepts inherent to feature engineering. Initially, the basic definitions of information retrieval, information extraction, word processing, and regular expressions are described. In the sequence, some of the main numerical data normalization techniques are presented. Subsequently, we present a set of techniques for measuring software quality. The software metrics described are size, complexity, coupling, and cohesion. Finally, some best coding practices are presented. These practices are divided into refactoring functions and variables, simplifying conditional logic, and refactoring parameters.

Chapter 5 presents the related works that inspired and guided the construction and development of this doctoral thesis. First, we present the works that used approaches to make students' grading in teaching programming. Next, we present works that used approaches to verify students' performance in teaching programming. Subsequently, we present the works that used students' grouping approaches and techniques in teaching programming. Overall, we have tried to put some intonation for these approaches' coding features. Also, we tried to rescue aspects of the works such as the programming language used, population, information from the database, and information about the experiment execution, among other study characteristics. Finally, we present an analysis of these aspects and how they are related to our research.

Chapter 6 presents the methodological approach, where we present the research method used to develop this doctoral thesis. First, we present the characterization and classification of our research. Next, we present the structure of our research that was guided by the steps of the knowledge discovery process in data mining. We describe how our approach works, the steps that must be followed to replicate our experiments, and how we collect and analyze our results. Finally, we present the configuration of our experiments. We define our research questions, describe our database, and present the machine learning techniques chosen for use and how they were evaluated.

Chapter 7 describes our main scientific contribution that addresses the area of feature engineering, specifically the creation and extraction of features. Sections of this chapter have been divided between learning topics. In this sense, we present the new source code features that we have developed to represent each learning topic. First, we introduce the learning topic and present its respective features. In addition, we relate these features to their respective learning outcomes. Then, we describe the feature and explain how the extraction process of this feature occurs at the source code level. In addition, we present the importance of the new feature regarding the learning topic and how this feature can help the teacher in the classroom. Finally, we discuss the potentials and weaknesses identified in the features defined and extracted from the source code tasks.

Chapter 8 presents the analysis of the results obtained from grouping students with similar programming skills. First, we present the results obtained in the different configurations tested for the clustering algorithm. Next, we present a descriptive analysis of our database. Then, we extract the features, which were previously proposed, from the source codes of our database and perform the grouping of students. We present and statistically analyze the output of our computational model that addresses programming skills individually. Subsequently, we present an overview of the student groups based on the midpoint of the features obtained in each grouping. The overview supports the teacher in understanding the skills developed by the students. Finally, we compare the group's overview with the benchmark defined for each learning topic.

Chapter 9 presents the final considerations of this doctoral thesis. We revisited the contributions achieved, which were published in annals of events and are available for consultation by the scientific community. Besides, we summarize the limitations present in our research and outline guidelines for future research.

Chapter 2

RECENT STUDIES ABOUT TEACHING ALGORITHMS AND DATA STRUCTURES

This chapter present the outcomes of a survey conducted with the primary objective of gaining deeper insights into the realm of programming teaching. The comprehensive findings and insights presented here are derived from a research that has been previously published in the Proceedings of the 49th Annual Conference on Frontiers in Education (SILVA et al., 2019).

In the view of the relevance of this subject, many studies are being conducted in order to disseminate the experiences and discoveries in the teaching of algorithms and data structures. New approaches are being developed and consolidated approaches have been tested in different environments. Some computing events have specific tracks for the publication of education related papers. However, there are whole events destined to the teaching of computing, such as the ACM Technical Symposium on Computer Science Education (SIGCSE).

An important part of the initial research of this thesis was to map the recent advances in teaching programming and to understand what approaches were being used and developed by other reserachers. Given that doing a complete survey about the topic of teaching programming was infesable given the amount of existing works, we used as a starting point the papers published in the ACM SIGCSE during the period of 2014 to 2018, which would be equal to the last five years, when the mapping was being conducted. In this sense, the aim of this study wass to map the approaches employed in the teaching of algorithms and data structures, published in SIGCSE, as well as their contributions and limitations. We were interested in answering the following research question:

$\mathbf{RQ}_1$	What are the recent approaches to teaching programming and data structures?

2.1 Study Setting

In this section, we elucidate the methodology employed for the construction of this mapping. Our research was structured around three distinct research subquestions, each designed to guide our investigative approach. Additionally, we expound upon the criteria and systematic procedures utilized for the selection of pertinent publications and the organization of data collection. To maintain transparency and scholarly rigor, we also address the inherent limitations that may impact the scope and outcomes of our mapping. In this section, we elucidate the methodology employed for the construction of this mapping. Our research framework was structured around three distinct research subquestions, each designed to guide our investigative approach. Additionally, we expound upon the criteria and systematic procedures utilized for the selection of pertinent publications and the organization of data collection. To maintain transparency and scholarly rigor, we also address the inherent limitations that may impact the scope and outcomes of our mapping.

2.1.1 Definition of Research Questions

The main question of research to be investigated in this study is as follows: What are the recent approaches to teaching programming and data structures? To do this, we define three other secondary research questions.

$\mathbf{RQ}_{1.1}$	What type of course content is covered?

In $\mathbf{RQ}_{1.1}$, our subsequent aim is to discern the specific content being imparted, whether it pertains to introductory programming (CS1) or data structures (CS2). Also we want to ascertain the primary programming languages employed in these educational contexts. This inquiry will enable us to gain a comprehensive understanding of the instructional landscape and the prevalent languages in the programming courses.

$\mathbf{RQ}_{1.2}$	What pedagogical teaching strategies are used?

In $\mathbf{RQ}_{1.2}$, our subsequent aim is to provide a detailed exposition of the pedagogical strategies that are currently in use for teaching programming. We want to elucidate not only what these strategies encompass but also how they are effectively implemented within the instructional framework. By delving into these pedagogical methodologies, we can offer valuable insights into the diverse approaches employed in programming education, thereby enhancing our understanding of their efficacy and impact on student learning outcomes.

RQ$_{1.3}$	What teaching support tools are used?

In **RQ$_{1.3}$**, our subsequent aim is to map the landscape of tools and frameworks that are under development and undergoing testing to support the teaching of programming. We endeavor not only to identify these tools but also to delve into their functionalities, applications, and their potential to enhance the educational experience.

2.1.2 Selection of Publications

In order to search for the papers we have performed the search using two different Databases. This is particularly important in order to get a good number of primary related studies (PETERSEN et al., 2008). The search process occurred in September 2018. A summary view of quantities selected and classified as primary studies, separated by year, can be seen in the Table 1.

Table 1 – Number of publications selected and classified by year

Year	DBLP		ACM		Final
	selected	*classified*	*selected*	*classified*	
2014	25	**11**	25	**+4**	**15**
2015	24	**9**	20	**+2**	**11**
2016	17	**7**	28	**+0**	**7**
2017	31	**13**	26	**+2**	**15**
2018	23	**11**	33	**+1**	**12**
Total	120	**51**	131	**+9**	**60**

The first database was the DBLP[2] computer science bibliography list. In this database the search was performed manually by visual inspection of the titles of the papers published in the SIGCSE conference in the years 2014, 2015, 2016, 2017 and 2018. All papers published between 2014 and 2018 (783 papers in total) were reviewed by pairs of researchers and selected according to the inclusion criteria (IC) presented in Table 2. The selection consisted of reading the title and keywords. The potentially related papers were registered in a spreadsheet and, in cases of doubt, the abstract of the paper was read. 120 papers were selected. The selected papers were read completely. When the uncertainty about the classification prevailed, the authors of this study met to establish a consensus. Its the end 51 papers were classified as primary studies.

The second round of selections was done using the ACM Digital Library[3]. The search was performed by filtered by event (SIGCSE), year of publication (2014–2018) and keywords (CS1 or CS2). The second selection resulted in 131 papers. The same procedure

[2] DBLP computer science bibliography, available in https://dblp.uni-trier.de
[3] ACM Digital Library, available in https://dl.acm.org/dl.cfm

Table 2 – Definition of ICs for selection of publications

Criteria	Description
CI 1	publications in SIGCSE
CI 2	categorized as Full Paper
CI 3	between 2014 and 2018
CI 4	focus on teaching programming (CS1 or CS2)
CI 5	aimed at computer science
CI 6	according to research questions

used for the selection of primary studies in the DBLP database was used to select the papers from the ACM Digital Library. The total number of papers classified as primary studies in this second search was 60. It should be noted that out of these 60 papers there were only nine new papers that were added to the pool of primary studies.

2.1.3 Data Extraction

After the final classification of the papers, the process of collecting and tabulating the results was initiated. The collected characteristics were grouped according to their similarity and organized into three categories: *(i)* content coverage; *(ii)* pedagogical strategy and, finally, *(iii)* support tools. A spreadsheet with the list of classified papers and the characteristics collected is available for online consultation[4].

2.1.4 Threats to Validity

The possible threats to the validity of this study are the following. The SIGCSE is an important event in computer teaching, but it is not the only one. Other events also contain papers with relevant contributions to the teaching programming area. However, in order to present more detailed results and in-depth discussions, we opted to restrict the scope of the review. To cover a larger slice of the literature, we recommend that other revisions equal to this, with the specific and well-defined scope, be performed.

We know that the symposium has happened since the decade of 80 and, therefore, many primary studies, prior to 2014, were not considered. However, the interest in this paper was to identify what is most recent in teaching programming. It may have happened that, due to the negligence of the authors, primary studies related in the range of 2014 and 2018 remained outside the selection. To circumvent this threat, peer reviews were performed. Given that this technique can still generate losses, we performed a second search in the ACM Digital Library database. We were able to add nine more papers.

[4] List of classified publications, available in goo.gl/ymL7Bp

2.2 Analysis of Results

In our mapping we found 60 related papers. The data were extracted, tabulated and analyzed in order to answer the research questions of this study. An overview of the results can be observed in Table 3. The results presented in Table 3 refer to the number of pedagogical strategies and support tools that appear within all papers. For example, in the paper (LATULIPE; LONG; SEMINARIO, 2015) three approaches are presented. In the years 2014 and 2017, the highest numbers of publications were found, with 15 papers (25.0%) for each year. In 2014 it was also the year that had the most publications on pedagogical strategies, 10 papers (16.7%). And, 2017 was the largest publication of support tools, 11 papers (18.3%). The year 2016, had the smallest number of published papers and the smallest proportions of pedagogical strategies and support tools.

Table 3 – Overview of Results

Year	Contents	Strategies	Tools
2014	15 (25,0%)	10 (16,7%)	7 (11,7%)
2015	11 (18,3%)	9 (15,0%)	4 (6,7%)
2016	7 (11,7%)	5 (8,3%)	2 (3,3%)
2017	15 (25,0%)	6 (10,0%)	11 (18,3%)
2018	12 (20,0%)	8 (13,3%)	6 (10,0%)

The following results will be classified among the course content, strategies and teaching support tools that have been identified.

2.2.1 Covered Contents

The first programming disciplines are usually divided between the fundamental concepts of the algorithms (CS1) and the advanced data structures (CS2). We verify the coverage of the content of the publications found. As can be seen in Figure 1, the 60 papers found were classified among CS1, CS2 or both. The x axis indicates the year of publication, and the bars of the y axis represent the grouping of the papers found. The publications that only depicted the teaching of CS1 were the majority, with 37 approaches (61.7%). The teaching of CS2 was found in 14 papers (23.3%). Nine other publications addressed both contents (15.0%).

The programming language is used to put the covered content into practice. Normally, this choice is at the discretion of the teacher, according to their personal preference. Among the programming languages, the most recurrent was *Java*, which was present in 24 publications (40.0%). Then Python appeared 18 times, *C++* 10 times and the language *C* was found in five publications. Other languages such as *C#*, *Matlab*, and *Ruby* were also found, but in smaller proportions.

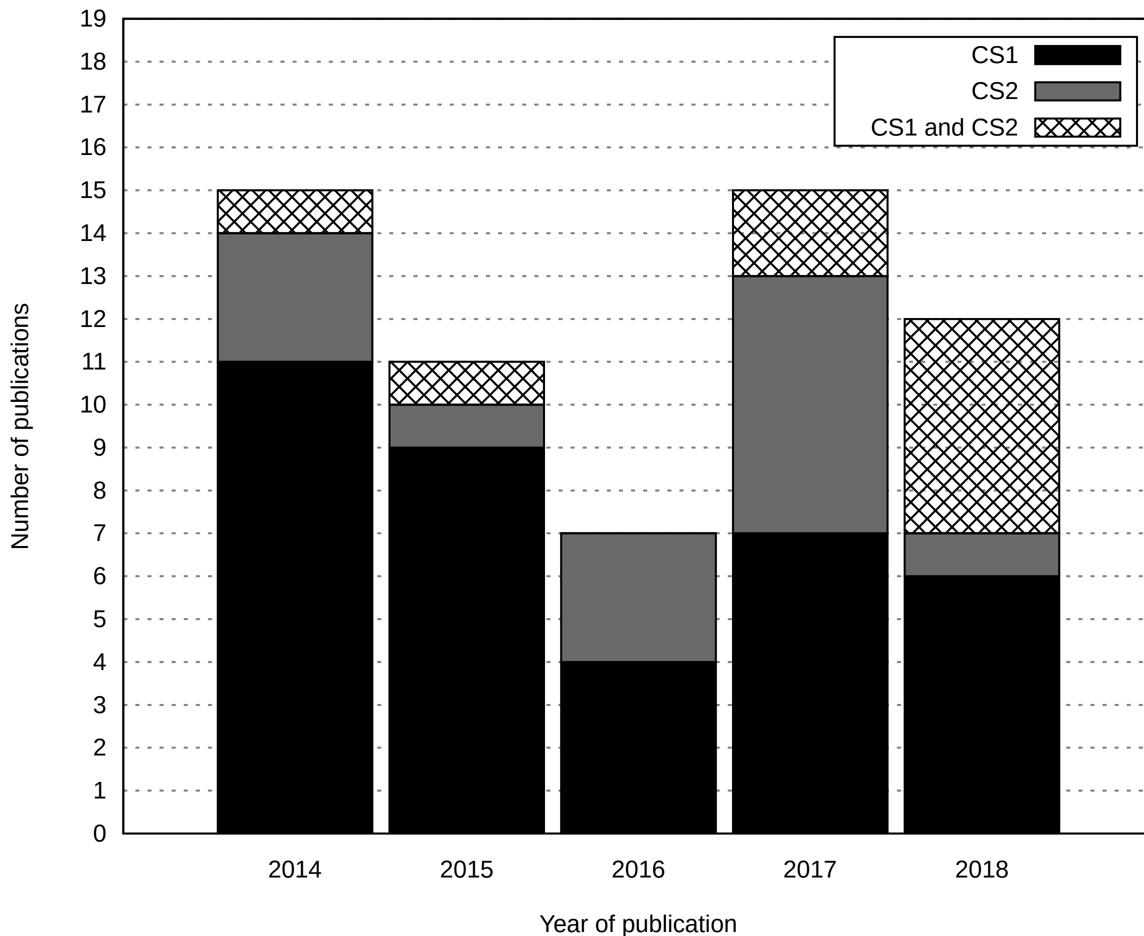


Figure 1 – Distribution of content covered by years

In the classroom, teachers use teaching methodologies as strategies to transmit content and provide learning to students. Such strategies are usually related to increased involvement and content fixation. The following will be presented with the strategies we have discovered.

2.2.2 Pedagogical Strategies

While the content covered by the course defines what needs to be taught, teaching strategies represent how the teacher will accomplish the transfer of knowledge. Traditionally, the teaching of programming mixes between theoretical classes and laboratory practices. Traditional teaching methods add privilege for the transmission of information by teachers regardless of what the students gain is. Whereas, the active methodologies put the student as the main focus and the teacher as a mediator of the teaching. As can be seen in Figure 2, we identified 32 papers that specified the teaching strategy used. The x axis indicates the year of publication, and the bars of the y axis represent the grouping of the pedagogical strategies found. Next, a brief description of the pedagogical strategies discovered and an

overview of their respective publications will be presented.

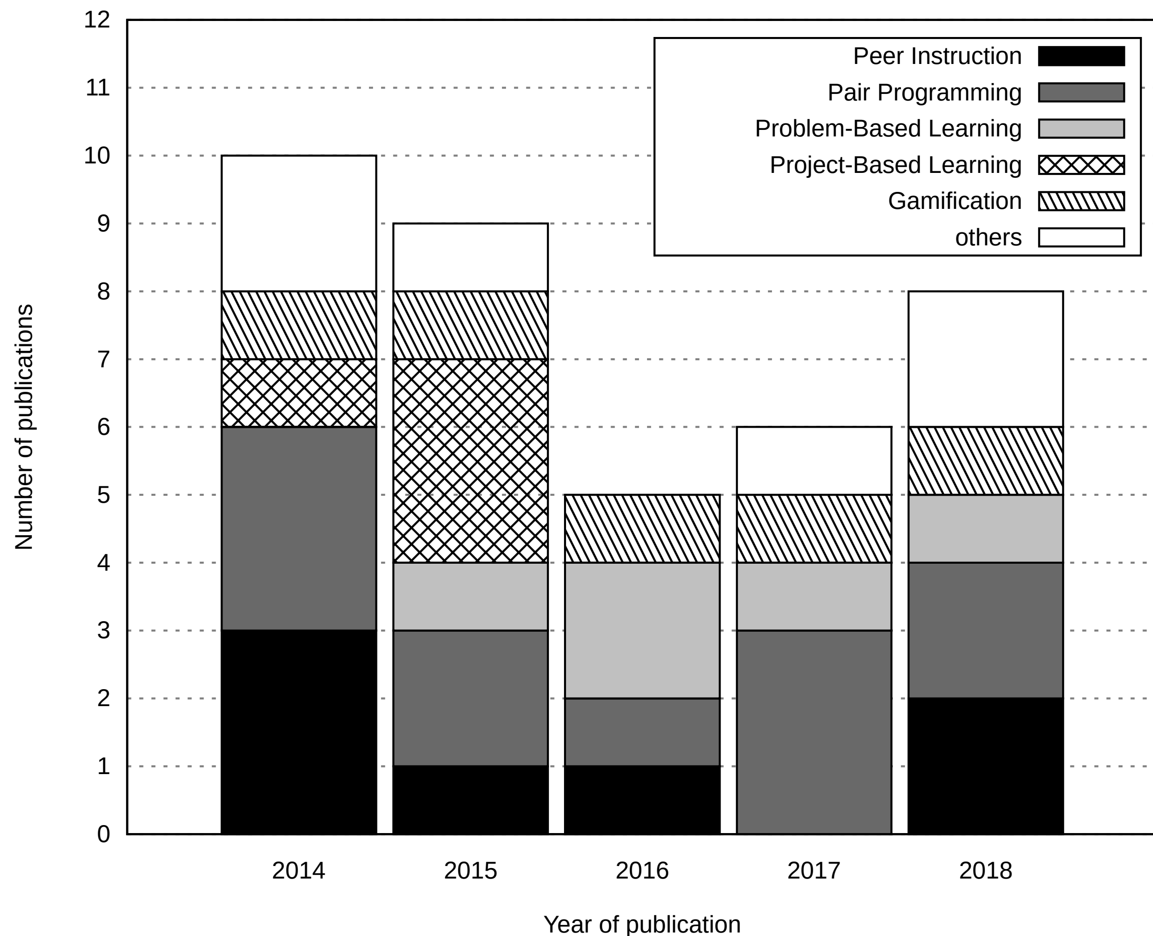


Figure 2 – Distribution of pedagogical strategies by years

Peer Instruction (PI) — is a pedagogical approach characterized by the provision of preparatory materials to students, allowing them to engage with the course content prior to attending class sessions (CROUCH; MAZUR, 2001). At the beginning of each class, students are presented with brief knowledge assessments, where they individually respond to questions. Subsequently, students engage in collaborative discussions within small groups to deliberate upon these questions, and, guided by their enhanced comprehension, they formulate revised responses. Throughout the instructional sessions, instructors intersperse the presentation of course content with interactive clicker questions (BRUFF, 2009).

The pedagogical strategy PI was found in seven papers (11.7%). The year 2014 presented the highest number of publications, with three papers. Although in 2017 there was no paper found. Usually, publications about PI involve their comparison with traditional methodologies (ZINGARO, 2014; PORTER et al., 2016; CACEFFO; GAMA; AZEVEDO, 2018; HAO et al., 2018), differing only in the configuration of the environment. In general, the results describe more satisfied and better-income students in the disciplines that use

PI. On the other hand, the preparation of classes usually involves more work the teacher (CACEFFO; GAMA; AZEVEDO, 2018). Important attention for the implementation of PI is the elaboration of challenging, but not very complex issues. Also, the teacher must make sure that is resetting sufficient time for reading and understanding the questions (PORTER et al., 2016).

Pair Programming (PP) — is a collaborative software development practice stemming from the Agile methodology, specifically, Extreme Programming (XP), wherein two individuals, designated as the Driver and the Navigator, collaborate closely at a single computer for brief programming sessions (BECK; ANDRES, 2004). The Driver is primarily responsible for in-depth source code composition, while the Navigator engages in ongoing collaboration involving code quality assurance and strategic coding planning (BRYANT; ROMERO; BOULAY, 2008). The functions are switched according to the previously defined protocol.

The PP pedagogical strategy was found in 11 papers (18.3%). In the years 2014 and 2017, three publications were found for each year. In 2016, only one publication was found. Experiments conducted in the classroom prove the benefits of using PP in teaching programming (MCCHESNEY, 2016; HARSLEY et al., 2017). In addition to the students developing more relevant questions, Li and Kraemer (2014) report that PP is a collaborative approach that promotes early engagement of critical thinking in the process of resolving the task. However, a key point in its use is the communication between the pair. Zarb, Hughes and Richards (2014), Zarb, Hughes and Richards (2015) used industry guidelines to verify the communication experience between peers in debugging and troubleshooting programming tasks. The pairs that were exposed to the guidelines obtained higher scores and more significant contribution rates. In the same direction, Rodríguez, Price and Boyer (2017) discover that collaboration is more effective when both partners contribute to the dialogue. Finally, in order to avoid the wear of the methodology, something to be observed and coordinated is the Union of equally qualified partners (CELEPKOLU; BOYER, 2018).

Problem-Based Learning (PBL) — is an instructional approach rooted in problem-solving, where educational themes are derived from genuine or simulated issues (KöLLING; BARNES, 2004). The student's engagement in the PBL process involves problem identification, evaluation of prospective solutions, and collaborative teamwork towards resolution (WOOD, 2008). Teachers introduce practical scenarios to direct the course content effectively. It is worth noting that activities encompassing source code comprehension, maintenance, refactoring, and adaptation or extension closely align with real-world software development practices in the industry.

The PBL pedagogical strategy was found in five papers (8.3%). The highest number of publications was found in the year 2016. In 2014 there were no publications. In order

to increase student engagement, Anderson et al. (2015) set up a CS1 course that uses real data to solve problems encountered in practice, such as DNA analysis, indication of friends in a social network, prediction of the outcome of the election, among others. Sheth et al. (2016) describes a course that uses four steps to solve problems: logical reasoning, implementation, analysis and presentation of results to the class. Something interesting is that there is an incentive to exchange ideas and share the source code, as long as the owner is referenced. Lovellette et al. (2017) investigate whether contextualized problems to the detriment of only numerical problems, make a difference in learning. On the other hand, Morrison et al. (2016) have opted to reduce the cognitive burden in solving problems. First, resolutions of similar problems are provided. Subsequently, students define smaller goals to reach the solution of their own tasks.

Project-Based Learning (PjBL) — is an educational approach that centers on students' active engagement in real-world projects and challenges. This methodology promotes deeper learning by encouraging students to investigate, explore, and find solutions to authentic problems (JAZAYERI, 2015). Key features of PBL include its emphasis on real-world relevance, active and collaborative learning, longer-term engagement in projects, and interdisciplinary application of knowledge. PBL helps students develop critical thinking, problem-solving, and teamwork skills while fostering a deeper understanding of subject matter.

The PjBL pedagogical strategy was found in four papers (6.7%). In the year 2015, 3 publications were found. However, between the years 2016 and 2018 there were no publications. One goal to be achieved with PjBL is to deepen the student's involvement with the learning task. Wood and Keen (2015) have opted to challenge students to create virtual worlds in a large comprehensive programming project. Lucas (2015) presents a sequence of projects that integrated the study of algorithmic paradigms with an illustration of how the choice of data structures significantly impacts an algorithm.

Gamification — is a pedagogical strategy that seeks to integrate competitive elements into non-gaming environments (IBANEZ; DI-SERIO; DELGADO-KLOOS, 2014). More recently, this approach has found application within the educational domain, where it leverages reward and challenge mechanisms to facilitate content delivery, while concurrently enhancing student engagement and motivation (DOMINGUEZ et al., 2013). This educational strategy capitalizes on elements often found in games to create an immersive and participatory learning experience.

The gamification pedagogical strategy was found in five papers (8.3%). For each year there was a publication. The idealization of tournaments favors the effort to develop more robust solutions, as occurs in the last phase of the methodological cycle of (SHETH et al., 2016). The use (or creation) of the games are initiatives that instill the students' competitive interest. Dicheva and Hodge (2018) presented *Stack Game*, a game for the

teaching of concepts, implementation and manipulation of stacks applied in a CS2 course. In the experiment, the students received two videos and didactic material for a previous study. In class, the teacher made a brief summary on the subject. Butler, Bezakova and Fluet (2017) encourage the approach of pencil puzzles to convey the content of CS1 and CS2.

Other pedagogical strategies were identified in 6 papers (10.0%). The elaboration of instructional videos was an alternative to fix the content and minimize the burden of teachers in a CS1 course (FRANK-BOLTON; SIMHA, 2018). VanDeGrift (2017) describes the creation and evaluation of activities using POGIL (Process-Oriented Guided Inquiry Learning) in the teaching of CS2. POGIL is based on constructivist and collaborative learning theories, in which students work in teams through a set of activities that guide the construction of the content. Battestilli, Awasthi and Cao (2018) conducted an experimental study based on a Two-Stage Project. In the first step, students submit their programming tasks individually. Followed by a second step, where they are paired to work on an enhanced version of the same task.

2.2.3 Support Tools

The tools to support teaching programming have the important role of optimizing and enriching the student's experience, as well as automating and making the teaching processes more dynamic. In Figure 3, we have identified 28 papers that specified the use of support tools. The x axis indicates the publication year and the y axis bars represent the grouping of the found tools. Next, a brief description of the discovered tools and an overview of their respective publications will be presented.

Automatic grade — is a mechanism used by teachers to automate the process of correcting programming tasks. This approach is particularly advantageous when dealing with extensive class sizes, characterized by heightened workload demands. Consequently, the time saved through automation allows for the development of more intricate instructional materials and heightened focus on addressing student challenges.

Studies on automatic evaluation tools were found in 6 papers (10.0%). Alfaro and Shavlovsky (2014) present *CrowdGrader*[5] that explores learning through collaborative assessment. Pettit et al. (2015) present a set of metrics to evaluate the interactive modifications in coding activities. Estey, Keuning and Coady (2017) perform an evaluation based on the amount of builds and hints requested. Castro-Wunsch, Ahadi and Petersen (2017) use machine learning models to predict student performance.

Learning environment — is a programming environment specifically tailored for novice programmers and serves as a platform for teaching beginners. In contrast, Integrated

[5] CrowGrader, available in https://www.crowdgrader.org

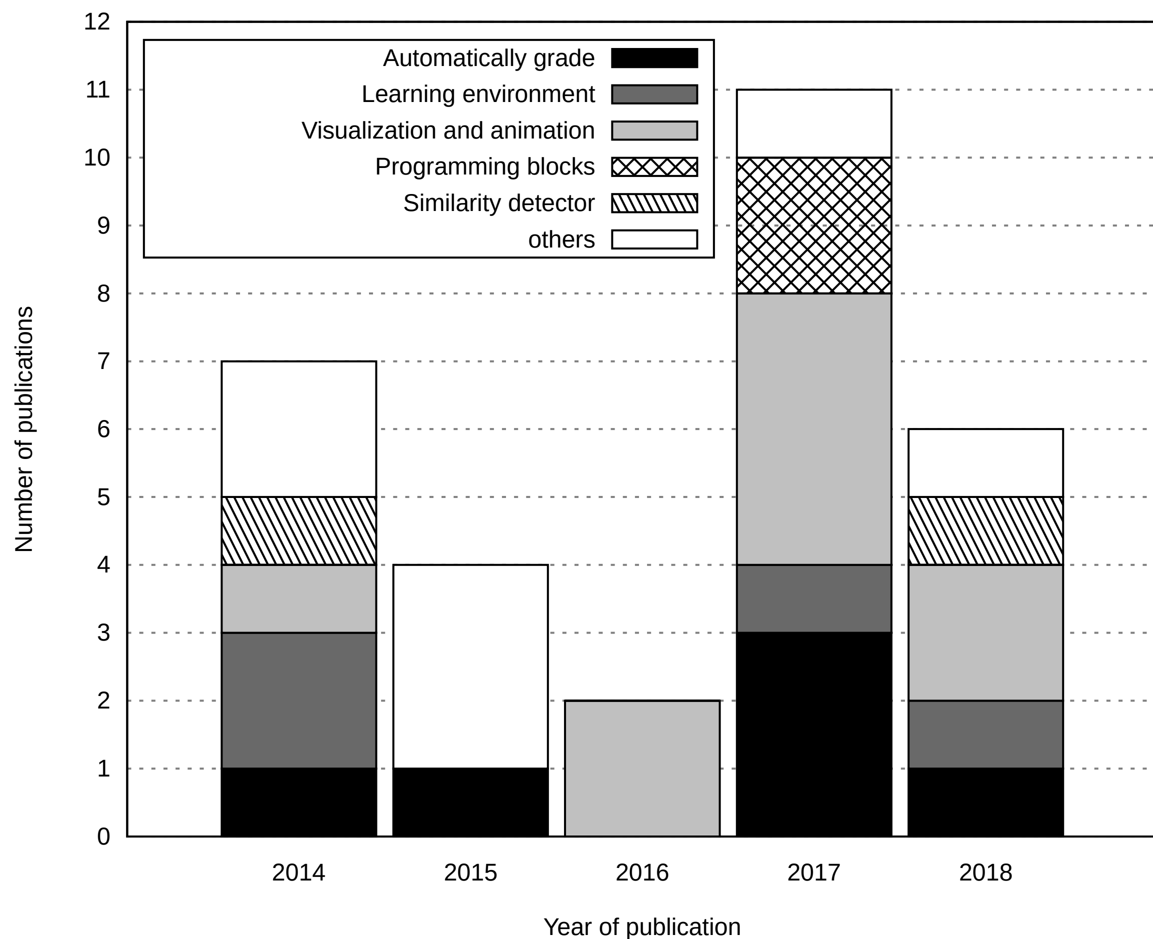


Figure 3 – Distribution of teaching support tools by year

Development Environments (IDEs) encompass a multitude of advanced programming features. Nevertheless, these comprehensive tools can appear intricate and provide limited guidance for individuals new to programming.

Studies on learning environments were found in four papers (6.7%). *CodeSkulptor*[6], used for teaching CS1, allows you to configure visual events for debugging and tracking of source code at run time (TANG; RIXNER; WARREN, 2014). *Mumuki*[7], an environment in which the theory arises from the exercises (BENOTTI et al., 2018). *Pythy*[8], a Web environment for the learning of the *Python* language (EDWARDS; TILDEN; ALLEVATO, 2014).

Visualization and animation — are mechanisms used to support learning, offering a graphical depiction of programming processes. This resource aims to supplant traditional textbooks by presenting visual information and interactive content, which serves to elucidate abstract and complex concepts.

[6] CodeSkulptor, available in http://www.codeskulptor.org/viz
[7] Mumuki, available in https://mumuki.io/home
[8] Pythy learning environment, available in https://pythy.cs.vt.edu

Studies on visualization and animation tools were found in nine papers (15.0%). The main application was in the teaching of CS2, in which students have great difficulties to understand the functioning of the advanced structures (CROSS et al., 2014; FäRNQVIST et al., 2016; BURLINSON et al., 2016; SCHREIBER; DOUGHERTY, 2017; FARGHALLY et al., 2017; YOUNG; WALKINGSHAW, 2018; MCQUAIGUE et al., 2018). The interactive books were to support learning are found in (FäRNQVIST et al., 2016; FARGHALLY et al., 2017). In addition, some studies allow the monitoring of learning (EDGCOMB et al., 2017; DEB; FUAD; KANAN, 2017; MCQUAIGUE et al., 2018). Cross et al. (2014) and Burlinson et al. (2016) presented studies in which the student can generate the visualization of their own coding.

Blocks programming — is a pedagogical approach that shares a striking resemblance to LEGO construction, characterized by its intuitive, visual, and modular nature. This innovative method has gained substantial prominence as a foundational tool for cultivating and nurturing the logical thinking and problem-solving skills of students, especially those embarking on their initial programming learning journey. Much like the step-by-step instructions in computer programming, blocks programming utilizes a repertoire of drag-and-drop visual elements, often referred to as "blocks." These blocks are designed to encapsulate discrete programming commands, creating an abstract but remarkably comprehensible representation of code. By employing these blocks, students are afforded an experiential learning process, where they can effortlessly assemble logical structures and sequences by interlocking these visual elements.

Studies on tools using block programming were found in two papers (3.3%). In order to avoid syntactic errors, Rodríguez, Price and Boyer (2017) used *Snap!*[9] to teach introductory programming. Price, Dong and Lipovac (2017) presented *iSnap*, a *Snap!*, that records the student's actions and generates contextualized tips.

Similarity detector — is a mechanism designed to evaluate and compare various activities, primarily aimed at identifying instances of plagiarism or duplication within the context of education, research, or content creation. This essential mechanism operates by scrutinizing the content of different files, with a primary focus on textual data, and cross-referencing it to identify potential matches or similarities. A platform that successfully makes this functionality is *MOSS*[10].

Studies on similarity detection tools were found in two papers (3.3%). Yan et al. (2018) presented *TMOSS*[11], an extension of *MOSS* that allows intermediate analysis of the accomplishment of a programming task. The development environment has been modified so that at the time of any compilation the current state of the task is sent to a versioner.

[9] Snap! Build Your Own Blocks, available in https://snap.berkeley.edu
[10] MOSS, available in https://theory.stanford.edu/ aiken/moss
[11] TMOSS, available in https://github.com/yanlisa/tmoss

Other seven techniques were tested by Gaudencio, Dantas and Guerrero (2014), among them the Jaccard coefficient, distance between texts and variations of the Terms Frequency (TF). The authors found that there may be greater concordance, in the evaluation of similarity, between a teacher and a tool than between two professors.

Tools of other natures were found in seven papers (11.7%). Zavala and Mendoza (2018) present a semantic-based approach to automatically generate a diversity of contextualized exercises. Heinonen et al. (2014) developed *CodeBrowser*[12], a tool for analyzing snapshots of source code.

2.3 Final Remarks

In a limited scope of five years of single-event publications, we find a large number of publications related to teaching programming. Most of them were directed to the CS1 course, using in 41.3% of cases the *Java* programming language. Probably, this demand is the answer to the difficulties faced by the novices in their first contact with the programming. The choice of *Java* language seems to be tied to the industry demand. However, we note that *Python* has also been a widely used programming language as it was present in 39.1% of CS1 publications. We recommend special attention in this initial phase, especially in the first weeks of the course. Students motivated from the beginning, are more able to follow the pace of classes and willingness to seek help when needed. In addition, we emphasize the importance in developing other didactic mechanisms to recover those students subject to disapproval and withdrawal. Few publications addressed the teaching of CS2, which may be a concern. The most widely used programming language is *Java*, which is present in 34.8% of CS2 publications. The course of data structures carries complex and abstract concepts that require prior knowledge and close attention.

It seems that the way to promote student engagement has been to alter teaching strategies. In programming education, 63.3% of the publications presented pedagogical strategies in the classroom. Active teaching methodologies are increasingly being applied at different levels of education. Among the publications that involve pedagogical strategies, 78.1% are related to CS1 teaching and only 34.4% with CS2.

The teaching-learning process is still very laborious for teachers, from the preparation of good materials to the correction of large volumes of exercises. Among the publications involving support tools, 78.6% are related to the teaching of CS1. This must happen, mainly, by the large scale of students registered in introductory programming courses. Given the large number of students enrolled in introductory programming courses, the improvement of existing tools and the development of new ones are important to

[12] CodeBrowser, available in https://github.com/codebrowser/web-client

improve the teaching and learning of CS1 and CS2.

The next chapter presents a detailed background on unsupervised learning techniques, with a specific focus on clustering algorithms. These algorithms constitute a pivotal component of our research, as they hold the potential to significantly enhance the landscape of programming education and streamline the process of knowledge discovery.

Chapter 3

DATA CLUSTERING ANALYSIS

This chapter presents the main concepts of unsupervised machine learning techniques focusing on data clustering approaches. Machine learning is the field of study that, from a large volume of data, gives computers the ability to learn a pattern without being programmed (HARRINGTON, 2012). There are two main approaches to machine learning. In supervised learning, each object comprises a set of features and a label that identifies the object's class. On the other hand, in unsupervised learning, only the input data is presented for the algorithm to discover the outputs. This means that the dataset has no previously known label information (ZHU; GOLDBERG, 2022).

In this doctoral thesis, we are particularly interested in unsupervised learning. This task focuses on exploratory analysis and is used to discover new patterns in the data. In this case, clustering algorithms are the most common. They aim to organize the data groups according to the distance between the objects. A practical example of applying a clustering algorithm is the discovery of companies whose characteristics diversify an investment portfolio on the stock exchange (PAI; MICHEL, 2009). Unsupervised machine learning techniques can be very useful for the exploratory analysis of textual objects, such as source code files.

3.1 Distance Measures between Objects

Clustering algorithms need a measure to calculate the distance between two objects that can be classified as similarity or dissimilarity measures (ROKACH; MAIMON, 2005). The first defines the degree of similarity; the higher the result, the more similar the objects are. The second is used to check the difference; the higher the result, the less similar the objects are. A brief description of the main measures will be presented in this section, where the distance between two numerical vectors u and v is calculated.

- Manhattan distance (or City Block) is used to calculate the distance it would take to get from one point to another within a city (SINWAR; KAUSHIK, 2014). The distance between two points is the sum of the absolute differences between a pair of objects' coordinates. In Equation 3.1, *manhattan* is the distance between u and v in an n-dimensional vector space, and i are the coordinates to be compared.

$$manhattan(\overrightarrow{u}, \overrightarrow{v}) = \sum_{i=1}^{n} |u_i - v_i| \tag{3.1}$$

- Euclidean distance is one of the most common measures to calculate the distance between two multidimensional space points (SINWAR; KAUSHIK, 2014). Once the coordinates of two points in the Cartesian plane are known, it is possible to use the Pythagorean theorem to calculate their distance. The calculation is defined as the sum of the square root of the difference between two objects' coordinates. In Equation 3.2, *euclidean* is the distance between u and v in an n-dimensional vector space, and i are the coordinates to be compared.

$$euclidean(\overrightarrow{u}, \overrightarrow{v}) = \left(\sum_{i=1}^{n} (u_i - v_i)^2\right)^{1/2} \tag{3.2}$$

- Cosine distance is often used to determine the similarity between two text analysis documents (SAHU; MOHAN, 2014). Each word in the document corresponds to a dimension in a multidimensional space. The similarity of Cosine verifies the orientation between two vectors through its angle. Two vectors with the same orientation have a cosine similarity of 1 and two vectors oriented at 90 degrees have a similarity of 0. That is, the smaller the angle, the greater the correspondence between the vectors. In Equation 3.3, *cosine* is the distance between u and v in an n-dimensional vector space, and i are the coordinates to be compared.

$$cosine(\overrightarrow{u}, \overrightarrow{v}) = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2}\sqrt{\sum_{i=1}^{n} v_i^2}} \tag{3.3}$$

- Bray-Curtis distance (or Sorensen) is a measure of dissimilarity commonly used in the study of environmental sciences (MICHIE, 1982). The calculation involves the absolute difference divided by the sum of the two samples. In Equation 3.4, *braycurtis* is the distance between u and v in an n-dimensional vector space, and i are the coordinates to be compared. If all coordinates are positive, their value will be between 0 and 1.

$$braycurtis(\overrightarrow{u}, \overrightarrow{v}) = \frac{\sum_{i=1}^{n} |u_i - v_i|}{\sum_{i=1}^{n} (u_i + v_i)} \tag{3.4}$$

- Chebyshev distance (or Maximum Value distance) is often used in cases where a high execution speed is required in the calculation (SOUZA; CARVALHO, 2004). It

calculates the absolute magnitude of the differences between coordinates between two objects. In Equation 3.5, *chebyshev* is the distance between the two objects u and v in an n-dimensional vector space, and i are the coordinates to be compared.

$$chebyshev(\overrightarrow{u}, \overrightarrow{v}) = max_i \, |u_i - v_i| \tag{3.5}$$

- Canberra distance was proposed in Lance and Williams (1967) and is often used to calculate the distance between two objects. It is similar to Manhattan distance, but the absolute difference between the two objects' variables is divided by the sum of the values of the absolute variables before the sum (BOUGUETTAYA et al., 2015). In Equation 3.6, *canberra* is the distance between the two objects u and v in an n-dimensional vector space, and i are the coordinates to be compared. Each fraction term has a value between 0 and 1, but the Canberra distance is not between 0 and 1.

$$canberra(\overrightarrow{u}, \overrightarrow{v}) = \sum_{i=1}^{n} \frac{|u_i - v_i|}{|u_i| + |v_i|} \tag{3.6}$$

3.2 Types of Clustering Algorithms

Data clustering analysis consists of an exploratory data analysis, in which a set of clusters share common characteristics (HARRINGTON, 2012). Each cluster has an indefinite number of objects that are assembled according to their similarity. The constituted clusters must obey a division in which the cluster's objects must be as similar as possible. Regardless, the distinct clusters must have a high dissimilarity between their objects (JAIN, 2010). There are two main categories of data clustering algorithms, which are the partitive and hierarchical algorithms.

3.2.1 Partitive Clustering

The partitioning clustering algorithms aim to discover exclusive clusters present in the data from a set of initial seeds (AGGARWAL; REDDY, 2014). These seeds are the central points of each cluster and are called centroids. The number of centroids is defined before executing the algorithm, and its coordinates are configured randomly. The centroid is calculated by averaging the values of the objects contained in each cluster and is improved iteratively to become more representative of the clusters present in the data. Then, a specific purpose function is optimized, and the quality of partitions is improved iteratively.

The K-means algorithm is the most popular (MACQUEEN et al., 1967; LLOYD, 1982; JAIN, 2010), and its pseudocode is presented in Algorithm 1. The only parameters required are the dataset (D) and the number of clusters (k) chosen arbitrarily by the user.

Then, each object in the dataset is assigned to a cluster. The assignment takes effect when finding the cluster closest to the selected object. The algorithm uses a function to calculate the error, which is the distance from the cluster centroid to the selected object. Finally, the centroids are updated according to the average value of all objects in the cluster in question. This process will be repeated until the objects stop changing clusters or a given number of iterations is run.

Algorithm 1: K-MEANS

```
  Input: dataset D ∈ ℝ and the value for k
  Output: Partitive clustering D in k groups
1 select k points as the initial centroids;
2 begin
3 |  while the centroids change do
4 |  |  form k clusters by assigning all points to the closest centroid;
5 |  |  recompute the centroid of each cluster;
6 |  end
7 end
```

The function used to calculate the proximity of centroids is called the Sum of Squared Errors (SSE) (XU; WUNSCH, 2008). This function is also used to calculate the quality of the partitive clustering. SSE is defined as the sum of the squared distance between the centroid and each cluster object. The error is the distance to the nearest cluster for each object (ROKACH; MAIMON, 2005). Given a dataset $D = \{x_1, x_2, x_3, ..., x_n\}$, where n is the total number of objects, clustering is formed from $C = \{C_1, C_2, C_3, ..., C_n\}$. In Equation 3.7, the sum of cluster C's square errors is calculated, where d is the distance measure, and c_i is the centroid of cluster C_i.

$$SSE(C) = \sum_{i=1}^{K} \sum_{x \in C_i} d(x, c_i)^2 \tag{3.7}$$

where c_i is calculated as:

$$c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x \tag{3.8}$$

The resulting clusters for an illustration of the application of K-means are shown in Figure 4. In Figure 4a, the input data are presented on a 2-dimensional dataset with three clusters. In Figure 4b, three random seed points were defined as centroid, and the objects were assigned to their respective clusters. In Figure 4c and 4d, the centroids change position until similar objects are in the same cluster. In Figure 4e, the final clustering obtained by the K-means algorithm is presented.

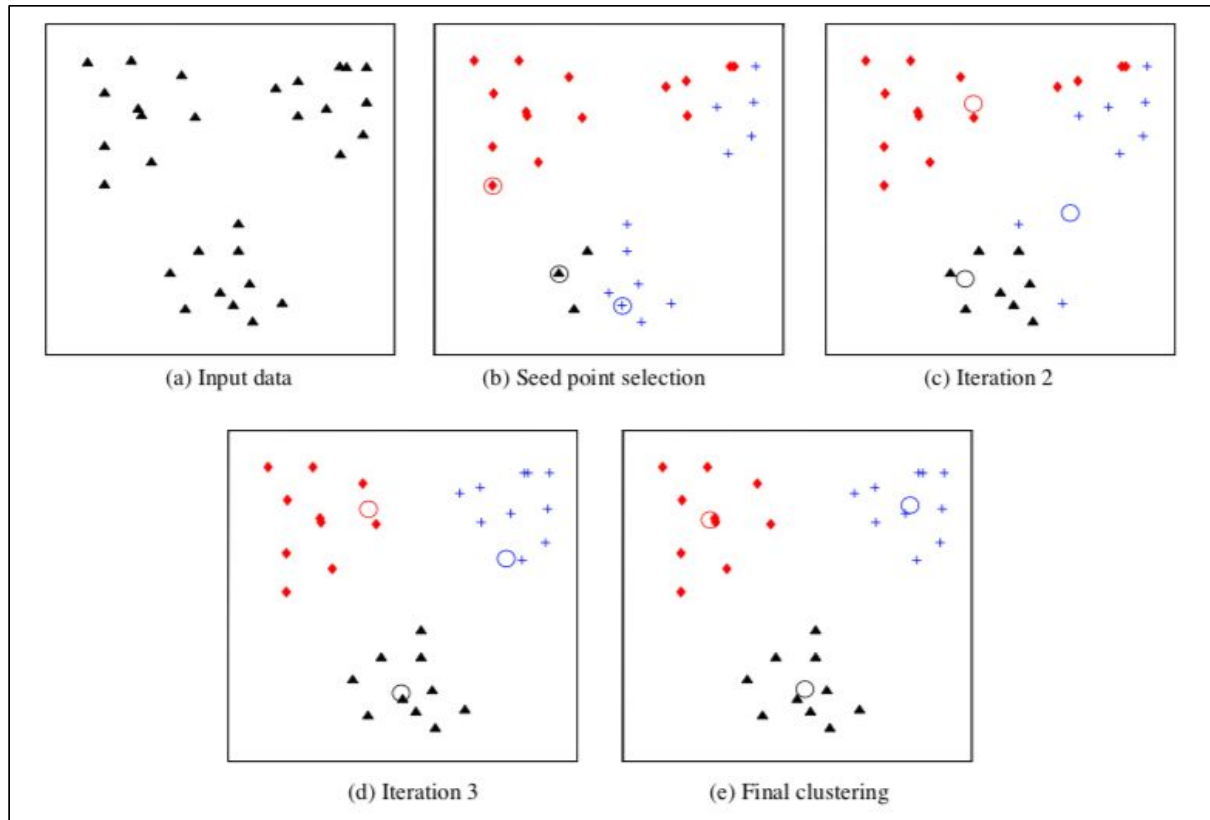


Figure 4 – An example of k-means clustering algorithm

Source: Jain (2010)

This type of algorithm does not work well with outliers in the data and presents undesirable characteristics when the groups are of different sizes, densities, or have non-globular shapes (AGGARWAL; REDDY, 2014).

3.2.2 Hierarchical Clustering

While in partitive clustering, the number of clusters needs to be defined in advance, hierarchical clustering is generally used when the number of clusters is unknown. The hierarchical clustering corresponds to a set of nested groups organized as a hierarchical tree that helps to determine the ideal number of clusters. This clustering can occur following the approach of divisive or agglomerative (MULLNER, 2011).

3.2.2.1 Divisive hierarchical clustering

The divisive hierarchical clustering approach is when the algorithm starts with a cluster that includes all the objects (EVERITT et al., 2011). This large cluster is then successively divided into smaller clusters until each cluster contains only one object. This approach is also called *top-down* because it divides objects from the root to the tree leaves

(XU; WUNSCH, 2008). An illustration is shown in Figure 5. The direction of the arrows indicates the sequence in which iterations occur, and the numbers are the hierarchy level of the tree. In the divisive approach at level zero, the root cluster contains all objects: $\{a, b, c, d, e\}$. As the tree navigates, the large clusters of objects dissolve into the leaves, where each object has its cluster. This approach is generally advantageous when it is desired to identify large cluster structures.

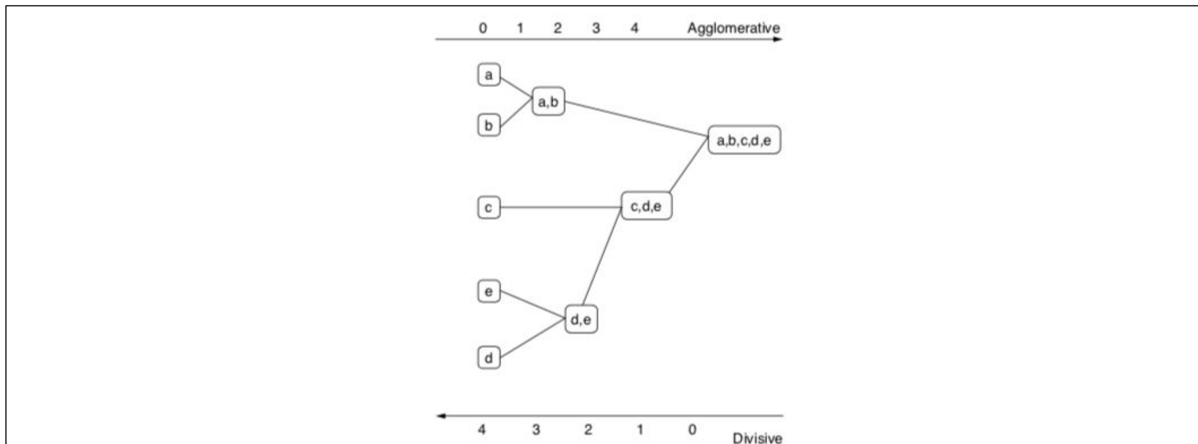


Figure 5 – Representation of the hierarchical tree structure

Source: Adapted from Everitt et al. (2011)

The divisive hierarchical clustering algorithmic representation is presented in Algorithm 2 (AGGARWAL; REDDY, 2014). The input parameter is an S matrix that contains the dissimilarity measures for all objects in the dataset. Then, a looping is started with a cluster containing all objects. At each iteration, the distance measurements between the clusters are updated, and the algorithm searches for the most distant object to create a new cluster. The looping stop condition is the complete isolation of objects.

Algorithm 2: Divisive hierarchical clustering

```
   Input: matrix of dissimilarity S ∈ ℝ
   Output: Divisive hierarchical clustering S
1  begin
2  |   Start with the root cluster that consists of all data objects;
3  |   while all objects are not in different groups do
4  |   |   update the distances between the clusters;
5  |   |   find the most distant objects;
6  |   |   separate into a new cluster;
7  |   end
8  end
```

3.2.2.2 Agglomerative hierarchical clustering

The agglomerative hierarchical clustering approach is when the algorithm starts with as many clusters as the number of objects (*singleton*) (EVERITT et al., 2011). Then, the closest clusters are merged successively until there is a single cluster with all objects. This approach is also called *bottom-up* because it brings together objects from leaves to the tree's root (XU; WUNSCH, 2008). In the illustration in Figure 5, the agglomerative approach starts at level zero with five clusters: $C_1 = \{a\}, C_2 = \{b\}, C_3 = \{c\}, C_4 = \{d\}, C_5 = \{e\}$. As navigation in the tree occurs, the individual clusters are joined to the root, where only one cluster contains all objects. In general, this approach is advantageous when small cluster structures need to be identified.

The algorithmic representation of agglomerative hierarchical clustering is presented in Algorithm 3 (AGGARWAL; REDDY, 2014). The input parameter is an S matrix that contains the dissimilarity measures for all objects in the dataset. Then, a looping is started to place all objects in the same group. At each iteration, the distance measurements between the clusters are updated, and the algorithm searches for the closest cluster to join a new cluster.

Algorithm 3: AGGLOMERATIVE HIERARCHICAL CLUSTERING

```
  Input: matrix of dissimilarity S ∈ ℝ
  Output: Agglomerative hierarchical clustering S
1 begin
2   while all objects are not in the same group do
3     update the distances between the clusters;
4     find the nearest clusters;
5     join in a new cluster;
6   end
7 end
```

3.2.2.3 Hierarchical clustering graphical representation

The hierarchical clustering methods provide a graphical tree representation of the results, called a dendrogram diagram. This diagram shows the order and distances between the groups formed during the clustering (KOREN; HAREL, 2003). Figure 6 presents an example of the tree representation of hierarchical clustering. The dendrogram nodes represent clusters, and the lengths of the stems represent the distances at which the clusters are first fused. Level zero represents the tree's root, where all the data objects to be grouped are. Subsequent levels are the child nodes that represent the clusters present in the data subsets. In this sense, it is possible to obtain different clusters for the same dataset, depending on the level at which the dendrogram is cut (AGGARWAL; REDDY, 2014).

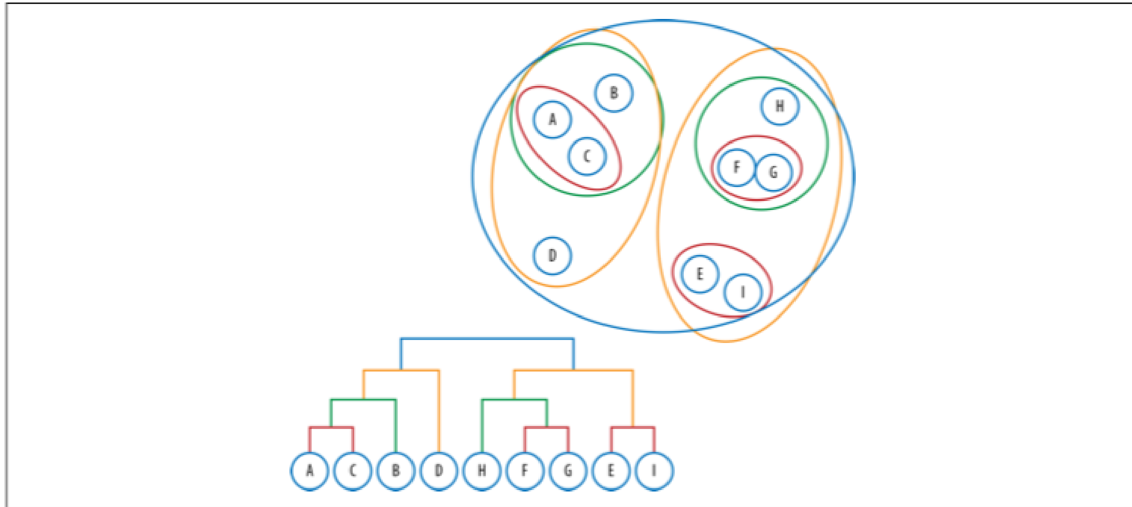


Figure 6 – An example of representing a hierarchical clustering

Source: Bengfort, Bilbro and Ojeda (2018)

3.2.2.4 Distance measures between hierarchical clusters

The linkage method defines how the distance between the clusters will be measured (MULLNER, 2011). The three main methods for calculating the proximity between two clusters are: Single Linkage, Complete Linkage, and Average Linkage. Their behavior is shown in Figure 7 and will be described in this subsection. Besides, there is a method defined by an objective function that is based on the quadratic error.

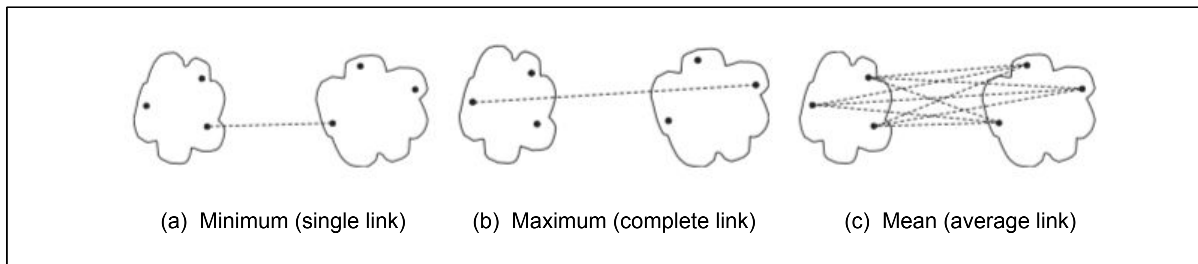


Figure 7 – Defining proximity between clusters

Source: Adapted from Tan, Steinbach and Kumar (2016)

- Single Linkage or Minimum (Figure 7a) defines the similarity between two clusters as the shortest distance between any two objects in different clusters (XU; WUNSCH, 2008). This means that the similarity between the clusters is greater than in other methods. In Figure 8, it is possible to observe that the Single Linkage tends to produce longer clusters. The result is independent of the normalization of the data.
- Complete Linkage or Maximum (Figure 7b) defines the similarity between two clusters as the longest distance between any two objects in different clusters (XU; WUNSCH, 2008). This means that the similarity between the clusters is lower than

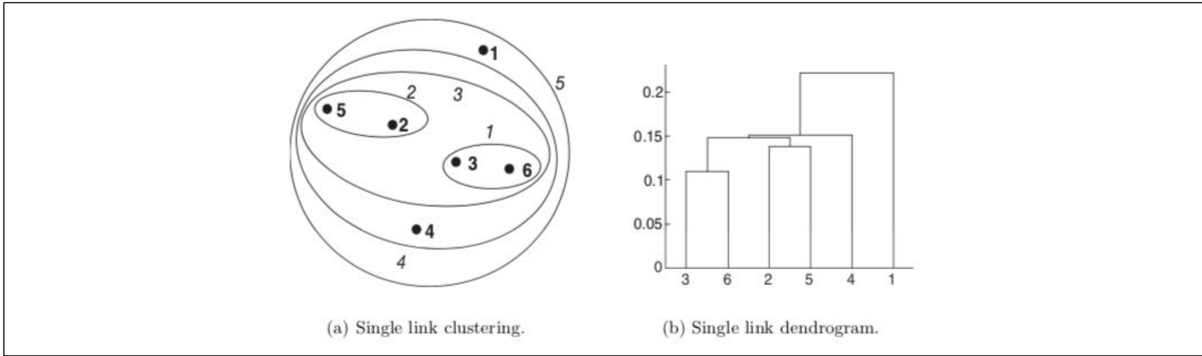


Figure 8 – Clustering representation using Single Linkage

Source: Tan, Steinbach and Kumar (2016)

in other methods. In Figure 9, the Complete Linkage tends to produce more compact clusters. The result is independent of the normalization of the data.

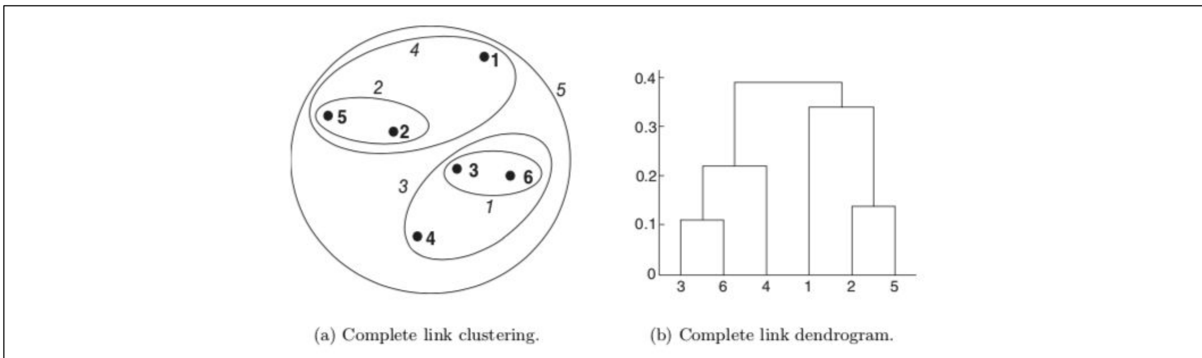


Figure 9 – Clustering representation using Complete Linkage

Source: Tan, Steinbach and Kumar (2016)

- Average Linkage or Mean (Figure 7c) defines the similarity between two clusters as the average distance between pairs of objects in one cluster and another (XU; WUNSCH, 2008). Figure 10 shows an example of clustering using this method. The result is sensitive to the scale of the data.

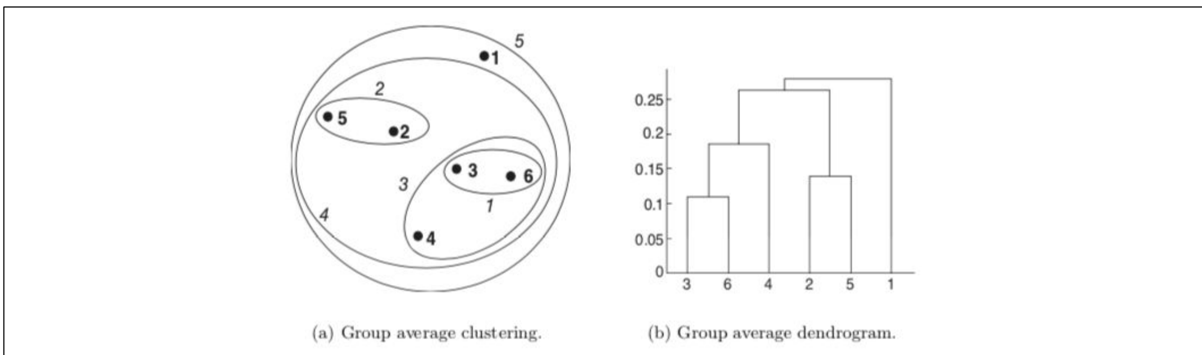


Figure 10 – Clustering representation using Average Linkage

Source: Tan, Steinbach and Kumar (2016)

- Ward link or Ward's minimum variance defines the similarity between two clusters as the sum of the pairs of objects' squared distances in different clusters (XU; WUNSCH, 2008). Figure 11 presents an example of clustering using this method. As a K-means algorithm, Ward's method tries to minimize the sum of the square distances from its cluster centroid points. The result is sensitive to the data scale and tends to minimize the total variation within the cluster.

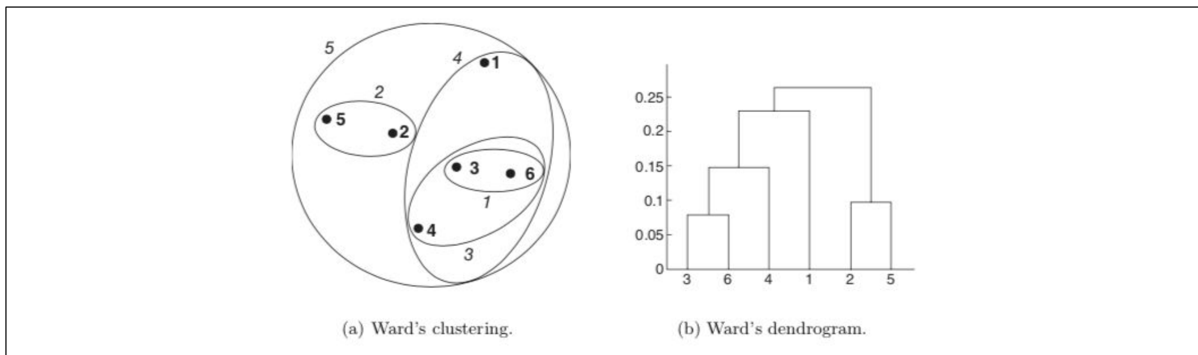


Figure 11 – Clustering representation using ward link

Source: Tan, Steinbach and Kumar (2016)

3.3 Model Evaluation

Choices in configuring the clustering approach can generate different clusters for the same dataset. The reliability of the obtained clusters depends on the validation of the model. According to Bonner (1964), a universal definition of what a good clustering is controversial. This evaluation is related to the criteria and requirements of the evaluator. However, the three main validation strategies recognized in the literature are presented in this subsection (THEODORIDIS; KOUTROUBAS, 1999).

- The external validation index is used to evaluate an algorithm's results based on the structure of the cluster compared to some predefined classification of objects (HALKIDI; BATISTAKIS; VAZIRGIANNIS, 2001). An example is checking the correspondence between the labels found in the clustering and the classes' labels provided externally. This index reflects the expert's intuition about the cluster structure and serves as a criterion to test the cluster's trend and the dataset (REZAEI, 2016).

- The internal validation index measures the degree of agreement between the cluster found, and the data set presented in the similarity matrix (HALKIDI; BATISTAKIS; VAZIRGIANNIS, 2001). This index can be approached from the point of view of intra-cluster similarity or inter-cluster similarity (REZAEI, 2016). The intra-cluster

measures the similarity between objects within a cluster; in this case, the compression between objects is the most important criterion for spherical clusters. The inter-cluster measures the distance between the clusters; in this case, a cluster is expected to provide clusters with objects well-separated (LEGÁNY; JUHÁSZ; BABOS, 2006). In the k-means algorithm, the internal validation index is usually measured using the sum of the quadratic error (SSE), used as an object of group optimization (ROKACH; MAIMON, 2005). Regardless, an existing internal validation index for hierarchical clustering is the Copenetic Coefficient (CARDONA et al., 2013).

- The relative validation index compares two different clusters (HALKIDI; BATISTAKIS; VAZIRGIANNIS, 2001). It is common to use internal and external validation measures for this validation.

3.4 Final Remarks

In this chapter, we present aspects that involve unsupervised machine learning that is fundamental for understanding this doctoral thesis. We focused on data clustering algorithms, such as the partitive and hierarchical algorithms. We describe how each of them works and provide practical examples. Besides, the main existing measures to calculate the distance between the objects in the cluster were described. Finally, we present the means to validate the quality of the formed clusters.

In our research, we strategically utilized the agglomerative hierarchical clustering algorithm, an approach that prioritizes the creation of clusters characterized by enhanced granularity. This algorithmic choice was guided by the desire to extract intricate patterns and capture nuanced relationships among data points, ensuring a comprehensive representation of the underlying dataset.

To establish the proximity between objects within the dataset, we selected the Euclidean distance as our distance measure. This metric has proven to be a robust and widely adopted method for calculating the dissimilarity or similarity between vectors in multidimensional space. By employing the Euclidean distance, we sought to precisely capture the spatial relationships between data points, fostering a comprehensive understanding of their similarities and differences.

Furthermore, to effectively merge and delineate the boundaries between clusters, we employed the Average Linkage method. This approach evaluates the distances between clusters based on the average distance between their constituent data points, offering a balanced and cohesive representation of cluster interrelatedness. By employing the Average Linkage method, we aimed to yield clusters that are coherent and representative of the underlying data structure, facilitating insightful interpretations and meaningful

explorations.

The next chapter explores the formation of the dataset provided as an input to these clustering algorithms. We present the fundamental concepts and the different ways to extract and create data features from source code.

Chapter 4

FEATURE ENGINEERING FOR SOURCE CODE ANALYSIS

This chapter presents the main concepts and potential features for source code analysis. Choosing which features to use is a critical step in the success of machine learning techniques. The ability to predict or explain a model's results is closely linked to the defined set of features. A set of features are attributes used to describe and distinguish objects (ANDERSON et al., 2013). Defining a feature set can seem simple, although it is performed on an extensive dataset – generally, the more features, the better (LEVY, 2010).

Feature engineering is a crucial aspect of data analysis and machine learning, with the primary objective of augmenting the set of features to gain deeper insights into the interactions among existing features, as emphasized by Heaton (2016). This process involves introducing new features that are derived from the original ones, leading to a more comprehensive representation of the data. However, it should be noted that this endeavor often necessitates a series of iterative tests and adjustments to identify which characteristics hold significance in representing the raw data, as highlighted by Bengio, Courville and Vincent (2013). These repeated evaluations are crucial for refining the feature set and ensuring that it encapsulates the most relevant information, ultimately contributing to the effectiveness and accuracy of the subsequent data analysis or machine learning tasks. By skillfully engineering features, researchers can unleash the full potential of the underlying data, leading to improved model performance and enhanced decision-making processes. As the field of machine learning continues to advance, feature engineering remains a fundamental and indispensable component for unlocking valuable insights from complex datasets.

4.1 Fundamentals of Source Code Analysis

A feature can exhibit various data types, with the primary ones being categorical, ordinal, and numeric, as outlined by Dong and Liu (2018). Categorical features are discrete values, such as eye color, covering the following domain: {black, blue, green, brown}. Ordinal features are values that follow a hierarchical order, for example, the position in a company covering the following domain: {intern, assistant, supervisor, manager, boss}. Numerical features are numerical, quantitative, or continuous values, such as the height covering the following domain: {1.60, 1.70, 1.80, 1.90}.

4.1.1 Information Retrieval

Information Retrieval (IR) is a critical area within the field of computing, aimed at verifying and extracting pertinent information from a given set of data from the user's perspective, as proposed by Mooers (1951). The process of information retrieval involves navigating through the dataset, seeking out relevant and valuable data that aligns with the user's specific needs and requirements. This entails employing sophisticated algorithms and techniques to efficiently identify and extract the most relevant information. The dataset used for information retrieval is called a corpus. In turn, a corpus consists of a set of objects, called documents (BENGFORT; BILBRO; OJEDA, 2018). For example, documents can be phrases, texts, people, music, or source codes.

In preprocessing a corpus, a common practice is to perform the tokenization of the text. This is the process of decomposing the document into each term that composes it – each word is transformed into a token (BENGFORT; BILBRO; OJEDA, 2018). The most used delimiters are white spaces, line breaks, or tabs. Then, cleaning the corpus is recommended. It is part of this procedure to remove words irrelevant to the application's context, called stopwords. For the textual context, the list of stopwords usually consists of articles, prepositions, adverbs, numbers, pronouns, and punctuation. For example, in English, this list could contain[13]: *I, me, my, myself, we*. This list could contain braces, semicolons, parentheses, blanks, and line breaks for programming languages. However, it is important to address each situation in the particularity of its application.

This corpus is later transformed into a Bag-of-Words (BoW). The BoW represents each document in a numeric vector (ZHENG; CASARI, 2018). This vector's words do not respect any hierarchy of sentence organizational structure or synonyms. The BoW can be binary (whether or not the word exists in the document) or store the term count. Each position in the vector references a term in the corpus vocabulary. The vocabulary is the set

[13] The Python NLTK library has a standard list of stopwords for both English and Brazilian Portuguese. Available in: http://www.nltk.org/howto/portuguese_en.html

of all possible words present in the corpus. Every time one of these terms is found in the document, its corresponding position in the vector increases. The purpose of this vector is to show the frequency of terms in the document. This application commonly produces sparse vectors, when many positions have values equal to zero (DONG; LIU, 2018).

To effectively navigate through the corpus and extract the desired information, information retrieval systems rely on a combination of search algorithms, indexing techniques, and relevance ranking methodologies. These components work in tandem to ensure the efficient and accurate retrieval of information that aligns with the user's query.

4.1.2 Term Frequency Normalization

The frequency with which a term occurs in a document is used to check the term's representativeness for the learning content. However, a term appearing in several documents becomes less representative of the data analysis task. Therefore, the Term Frequency – Inverse Document Frequency (TFIDF) is used to measure the importance of a term in a document present in a collection of documents (corpus) (JONES, 1972).

- Frequency of Terms (TF) is the number of times each term occurs in a corpus document. The result is given by comparing the total number of terms in the same document. The TF calculation is given by Equation 4.1; where $n_{i,j}$ is the frequency of a term i in a document j.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}} \tag{4.1}$$

- Inverse Document Frequency (IDF) is the weight of each term in the collection of documents. Terms that rarely occur in the corpus score high on the IDF. The IDF calculation is given by Equation 4.2; where N is the total number of documents in the corpus and df_t is the number of documents containing the term t.

$$idf_{(w)} = \log\left(\frac{N}{df_t}\right) \tag{4.2}$$

- Term Frequency – Inverse Document Frequency (TFIDF) is a statistical measure that indicates the importance of a term within the corpus. The TFIDF value of a term increases proportionally as the number of term occurrences in a document increases. However, this value is relative to the number of documents that this term appears. The TFIDF of a term in a corpus document is the previous equation's product (4.1 and 4.2). The TFIDF (w) calculation is given by Equation 4.3; where $tf_{(i,j)}$ is the number of occurrences of a term i in a document j, $df_{(i)}$ is the number of documents containing i, and N is the total number of documents.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right) \tag{4.3}$$

4.1.3 Regular Expressions

One way to retrieve information is to use regular expressions. Regular expressions are a formal language for specifying strings and are the simplest ways to process text. They define standards that a text must match (GOYVAERTS; LEVITHAN, 2012). A practical application is to verify that the user's given value is a valid email address. Other applications include searching for a word in a text, extracting specific parts (such as a telephone or postal code), replacing words or parts of the text, and dividing a text into smaller parts using delimiters.

Regular expressions are made up of two categories: literals and special characters (LÓPEZ; ROMERO, 2014). Literals are the simplest form of pattern matching in regular expressions. They are common letters (a to z) or numbers (0 to 9) and will succeed whenever that literal is found. The special characters are known as metacharacters. Metacharacters are symbols with a specific function that can change depending on the context. They can be combined to form more complex constructions. Some examples of metacharacters are: backslash, dollar sign, plus sign, opening and closing parenthesis, dot, and question mark.

4.2 Software Quality Metrics

In this section, we introduce some software quality metrics. These metrics are related to the software's size, complexity, coupling, and cohesion. Whenever necessary, we will use the source code example in Figure 12 to illustrate. The purpose of this source code is to calculate a student's final average. For this, one main function and three auxiliary functions were created. Three values are received as an argument in the main function, representing the student's three grades. Then, the sum of the three grades is performed using a specific function. Then, the previous operation results are used as a parameter to access the function that calculates the grades' average. Finally, the average of the three grades is used to verify whether or not the student has reached the minimum final grade established. The variables used in the source code are of the primitive data type *double*, which is more suitable for operations as values belonging to the set of real numbers. Also, two libraries were used, and two constants were defined.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MEAN 7.0
5 #define NUM_GRADES 3
6
7 // Calculates the sum of three grades
8 double calc_sum(double gd1, double gd2, double gd3){
9     return (gd1 + gd2 + gd3);
10 }
11
12 // Calculates the sum of three grades
13 double calc_average(double sum_grades){
14     return (sum_grades / NUM_GRADES);
15 }
16
17 // Checks wheter or not a student has reached the average
18 double calc_result(double average_grades){
19     if(average_grades >= MEAN){
20         return 1;
21     }else{
22         return 0;
23     }
24 }
25
26 // Main function of calculating the average of three grades
27 int main(int argc, char *argv[]){
28     double sum, average;
29
30     sum = calc_sum(atof(argv[1]), atof(argv[2]), atof(argv
          [3]));
31     average = calc_average(sum);
32     printf("%.2f", calc_result(average));
33     return 0;
34 }
```

Figure 12 – Source code example that averages three grades

Source: The author

4.2.1 Size Metrics

Size metrics were the first to be created and are generally used to calculate other metrics (ANDERSON, 2004). They are associated with the size of the source code. Usually, these metrics with higher values represent more complex software (OLIVEIRA et al., 2008). The biggest impact is on the computer's memory storage and processing consumption. Most size metrics are commonly used in source codes written in structured and object-oriented languages.

- The Lines of Code (LOC) is the most common metric (ANDERSON, 2004). LOC assesses the complexity of the software through the total volume of lines in the source code. This count includes only the lines of instructions that the processor will execute. That is, blank lines or comment lines for source code are disregarded. In the case of a real programming project involving several classes and files, all lines are counted. For example, the line count for the source code in Figure 12 is $LOC = a$, such that a is equal to 24.

- The Lines of Code by Method (MLOC) is an alternative to LOC (OLIVEIRA et al., 2008). In this metric, the counting scope is restricted to the source code block of the method. Only the lines of instructions that the processor will execute are counted. Following the common pattern, blank lines and comments are disregarded. MLOC is often used in object-oriented languages (OLIVEIRA et al., 2008), but it can be adapted to structured languages' function blocks. For example, the line count for the function *calc_result()* in Figure 12 is $MLOC = a$, such that a is equal to 5.

Results with high LOC and MLOC metrics values can reduce the source code's readability and cause more memory filling and processing consumption. Both metrics represent a simple and quick way to measure software. The fact that they are independent of the programming language also makes them flexible. However, the main problems are inherent to its practicality (BHATIA; MALHOTRA, 2014). First, the gross count of lines of code neglects the complexity of programming tied to each line. Second, each programming structure's meaning is not considered (conditionals, looping, branches, libraries). The more advanced language features are used, the tendency is to lower the LOC. This situation can lead to a misunderstanding from a programming point of view. Finally, the best coding solution for the same software can vary, in the number of lines, between different programmers.

Still, other metrics are related to the gross count applied to object-oriented languages, which can be adapted for structured languages (OLIVEIRA et al., 2008).

- The Number of Attributes (NOA) counts the total number of attributes within a class. This metric is equivalent to a structured language program's total number of variables. In structured language, NOA should be counted at the level of the global and local variables. For example, the attributes count for the function *calc_sum()* in Figure 12 is $NOA = a$, such that a is equal to 3.

- The Number of Methods (NOM) counts the total number of class methods. This metric is equivalent to a structured language program's total number of functions. For example, the functions count for the source code in Figure 12 is $NOM = a$, such that a is equal to 4.

- The Number of Parameters (NOP) counts the total number of parameters defined in the class's method signature. This metric is equivalent to the total number of parameters defined in the signature of functions within a structured language program. For example, the parameters count for the function *calc_sum()* in Figure 12 is $NOP = a$, such that a is equal to 3.

- The Number of Static Methods (NOSM) counts the total number of static methods defined within a class. This metric is equivalent to using libraries declared in the header of a program in a structured language. For example, the static functions count for the source code in Figure 12 is $NOSM = a$, such that a is equal to 2. The declared libraries were *stdio.h* to allow textual output, and *stdlib.h* to enable converting the received arguments in character format to the floating-point format.

- The Number of Static Attributes (NOSA) counts the total number of static attributes defined within a class. This metric is equivalent to the total number of constants declared in a program's header or library constants used in a structured language. For example, the static attributes count for the source code in Figure 12 is $NOSA = a$, such that a is equal to 2.

Global variables tend to cause more memory filling. On the other hand, high values for the local and NOP variables characterize a more local source code with strong indications that the memory space will be emptied sooner. However, a very high number of parameters can indicate that the code block needs to be divided into sub-blocks. In general, high values reduce the source code's readability and increase the complexity of the software.

Other metrics that require specific characteristics of object-oriented languages were presented in (OLIVEIRA et al., 2008). Therefore, they cannot be adapted for structured languages.

- The Number of Classes (NOC) counts the total number of classes defined in the software.

- The Number of Packages (NOPK) counts the total number of packages defined in the software.
- The Number of Interfaces (NOI) counts the total number of purely abstract classes defined in the software.

These metrics generally refer to the modularity and flexibility of the software. Results with high values for these metrics provide signals of a more organized, reusable, and easy-to-main source code. However, they make the software more complex.

4.2.2 Complexity Metrics

Complexity metrics are responsible for verifying the difficulty of understanding or expressing a source code. They are usually related to the flow of execution, granularity, or the level of nesting of the source code. Flow structures are powerful tools that allow the development of advanced logical solutions that are effectively and easily implementable.

4.2.2.1 Cyclomatic Complexity

The Cyclomatic Complexity (CC) software metric is the most common and was initially proposed by McCabe (1976). This metric can be applied in structured and object-oriented languages (ANDERSON, 2004; OLIVEIRA et al., 2008). CC measures how many tests need to be performed to verify all possible flow the code can have (MUNSON; KHOSHGOFTAAR, 1992). Conditional structures, repetition structures, and logical operators represent these different flows. The higher the values obtained with this metric, the more complex the software is. In order to compute the CC, it is necessary to use a Control Flow Chart (CFC). A CFC visualizes sequential procedural instructions (SARWAR; SHAHZAD; AHMAD, 2013). The mathematical representation of CC is presented in Equation 4.4. Its result is calculated from a CFC, where E is the number of edges and N is the number of nodes in graph G.

$$V(G) = E - N + 2 \tag{4.4}$$

To illustrate the CC, we present an example using the *calc_result()* function of the source code shown in Figure 12. We create the CFC, as shown in Figure 13. The structure formed was a graph with five edges and five nodes.

1. Each node in the graph corresponds to a line of the source code numbered in red. Thus, the total number of nodes E is equal to 5.

2. The edges of the graph represent the flow possibilities of the source code. There is only one path from Line 1 to Line 2, but in Line 2, there is a conditional that can lead to Line 3 or Line 4. Finally, any conditionals lead to the end of the algorithm in Line 5. In the example shown, the total number of edges N is equal to 5.

3. Therefore, the result of the CC calculation is equal to 2 ($V(G) = 5 - 5 + 2$).

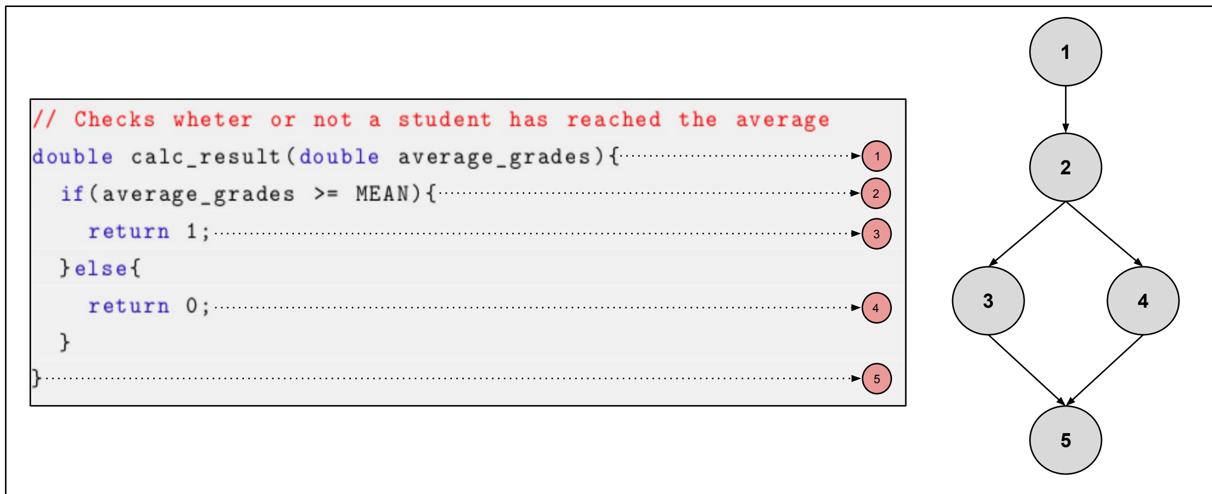


Figure 13 – Control Flow Chart for *calc_result()* function

Source: The author

Anderson (2004) defined a scale to classify the results of CC about the assessment of risks associated with software development. Table 4 shows this scale. Values between 01 and 10 represent simpler and less complex source code structures. Values between 11 and 20 represent source code structures with moderate complexity. Values between 21 and 50 represent highly complex source code structures. Finally, values above 50 represent source code structures with very high complexity.

Table 4 – Cyclomic Complexity Scale defined in Anderson (2004)

Cyclomatic Complexity (CC)	Risk Evaluation
1-10	A simple function, without much risk
11-20	More complex, moderate risk
21-50	Complex, high risk function
greater than 50	Untestable program (very high risk)

4.2.2.2 Weighted Methods per Class

The Weighted Methods per Class (WMC) is a variation of CC. WMC is the sum of the Cyclomatic Complexity for all class methods (OLIVEIRA et al., 2008). High-value results for this metric make the source code more complex.

To illustrate the WMC, we present an example using the source code shown in Figure 12. We created the CFC for each of the functions. The graph of the function *calc_result()* has already been presented in Figure 13. The other three functions are represented in Figure 14. Since there are no conditionals or loops, the flow is simple. Then, we present the CC calculation for each function and the WMC result.

1. Result for the function *calc_result()* is equal to 2 ($V(G) = 5 - 5 + 2$)
2. Result for the function *calc_sum()* is equal to 1 ($V(G) = 2 - 3 + 2$)
3. Result for the function *calc_average()* is equal to 1 ($V(G) = 2 - 3 + 2$)
4. Result for the function *main* is equal to 1 ($V(G) = 6 - 7 + 2$)
5. The WMC result, which is the sum of all cc results, is equal to 5

Table 5 presents the risk scale associated with software development defined in (OLAGUE; ETZKORN; COX, 2006).

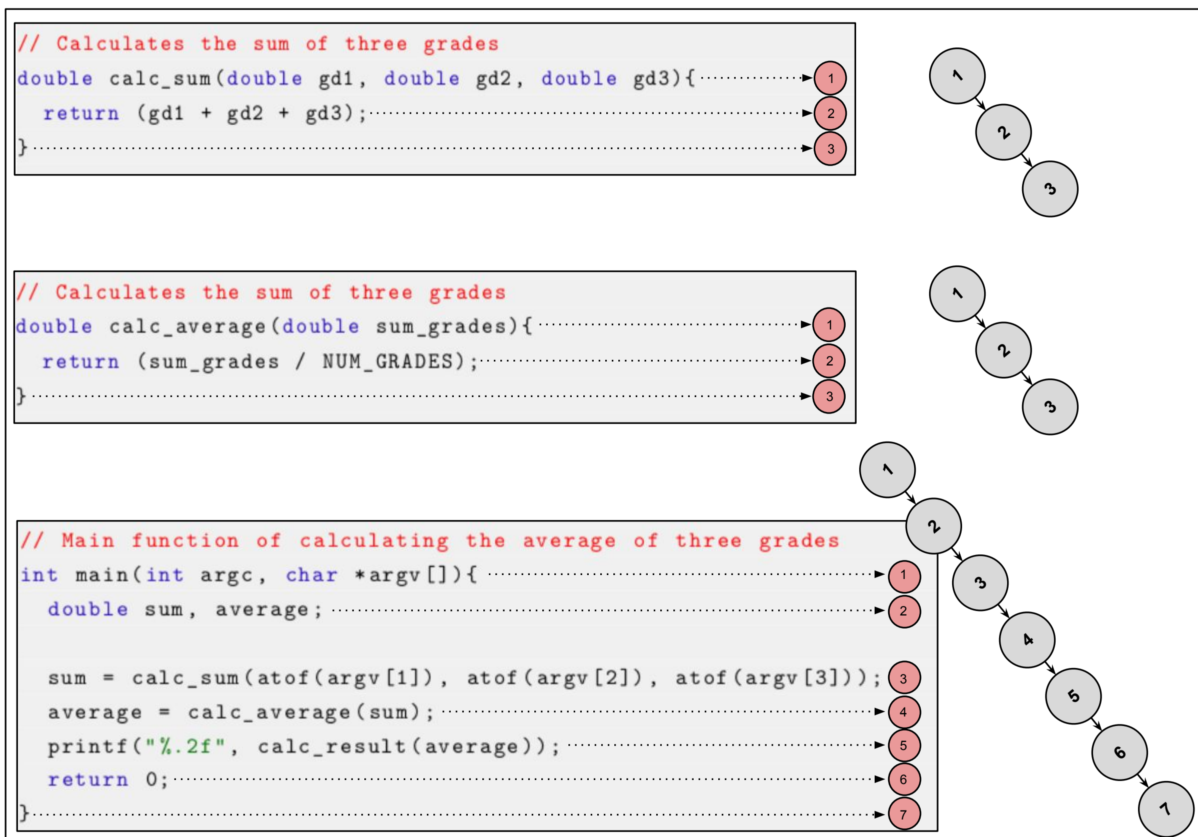


Figure 14 – Control Flow Chart for the other functions of the source code

Source: The author

Table 5 – Weighted Methods per Class Scale

Weighted Methods per Class (WMC)	Risk Evaluation
1-20	Good value of class complexity
20-100	Moderately high value of class complexity
greater than 100	High class complexity, cause for investigation

4.2.2.3 Nested Block Depth

A limitation of CC-based metrics is the non-differentiation between the breadth and depth of the structures' nesting. A high level of breadth or depth of these structures can result in unwanted software in terms of quality Alrasheed and Melton (2014). Nesting amplitude is the number of nested structures at the same level. In Figure 15(a), a tree with two nodes is presented, where the amplitude is equal to 2, and the depth is equal to 1. Nesting depth is the number of structures chained together at different levels (one within the other). In Figure 15(b), a tree with two nodes is shown, where the amplitude is equal to 1, and the depth is equal to 2.

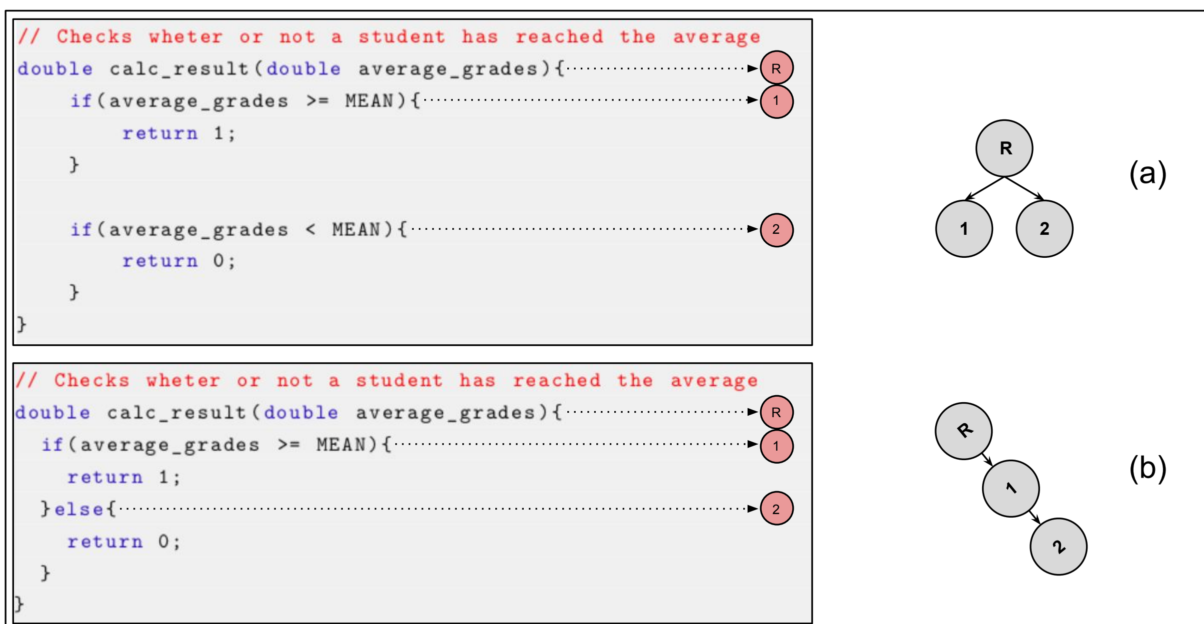


Figure 15 – Example of a nesting tree

Source: The author

The depth of nesting is an aspect that attributes more complexity than the breadth of nesting. Very high levels of depth can result in incomprehensible source code and extremely complex software. Therefore, a specific metric is defined in this situation.

Nested Block Depth (NBD) calculates the depth of nested code blocks. This metric's high-value results lead to worse readability and more complex solutions (OLIVEIRA et al., 2008). For example, the nested block's depth for the function *calc_result()* in Figure 12 is $NBD = a$, such that a is equal to 2.

4.2.2.4 Halstead Complexity

The Halstead Complexity (HC) software metric was initially proposed in Halstead (1977) and can be applied in both structured and object-oriented languages. HC is calculated from the count of operators and operands present in the scope of the source code. The operators are the reserved words of the programming language, as well as the arithmetic, logical and relational operators. On the other hand, operands are the logical units to be operated, for example, explicitly defined variables, constants, and numerical values. This metric makes it possible to measure the approximate size of the software execution in bits (NURMINEN, 2003). In Equation 4.5, *halstead* is the result of Halstead Complexity.

$$halstead(volume) = OPs_occur * \log_2 OPs_unique \tag{4.5}$$

Where *OPs_occur* is the sum of the total number of occurrences of the operands (*opd_occur*) and the total number of occurrences of the operators (*opt_occur*); and, *OPs_unique* is the sum between the unique operands (*opd_unique*) and the unique operators (*opt_unique*). In order to illustrate the HC, we present an example using the *calc_result()* function of the source code shown in Figure 12.

1. To calculate the number of distinct operators (*opt_unique = 10*)
   - set of operators found: *double, if, else, return, {, }, (,), >=, ;*
2. To calculate the number of distinct operands (*opd_unique = 4*)
   - set of operands found: *average_grades, MEAN, 0, 1*
3. To calculate the total number of occurrences of the operators (*opt_occur = 17*)
4. To calculate the total number of occurrences of the operands (*opd_occur = 5*)
5. To calculate the sum between different operators and operands (*OPs_unique = 14*)
6. To calculate the sum between the total of operators and operands (*OPs_occur = 22*)
7. To calculate complexity (*halstead(volume) = 83.76*)

4.2.3 Coupling Metrics

In the context of software metrics, the coupling is the level of dependency between the source code modules. This can be considered the number of source code blocks that access the same part of the global data in a structured language. Object-oriented languages

divide the coupling between afferent and efferent (MARTIN, 1995). Afferent coupling is when the classes of other packages depend on the classes of a given package. Efferent coupling occurs when classes in one package depend on classes in other packages.

Another coupling measure refers to the number of calls to the functions (ANDERSON, 2004). The name is *fan-in* for the number of calls made to a given function. The number of times a specific function calls other functions is called a *fan-out*. When software is tightly coupled, its maintenance and reuse become more complex.

4.2.4 Cohesion Metrics

Cohesion metrics are used to verify the degree of relationship between blocks of code (ANDERSON, 2004). High cohesion means that the code blocks perform only one task and do not share their variables with other blocks (HENDERSON-SELLERS, 1996). A function is cohesive in the structured language if its elements are designed to perform a single specific task. If the function is being used to solve many tasks, it may be necessary to divide it. Modularization promotes reusability and makes the source code less complex, easier to understand, and test (YOURDON; CONSTANTINE, 1979).

Weiser (1984) developed the first metrics to slice the complexity of the source code. These metrics were subsequently improved by Ott and Thuss (1993). The set of metrics that were defined to verify the cohesion of the source code are: coverage, minimum coverage, maximum coverage, overlap, and tightness. In Figure 16, these metrics are described mathematically. Such equations were defined by Green et al. (2009), and the meaning of the symbols is presented as follows:

- M is the function.
- V_M are all the variables in the function.
- V_0 are the output variables in the function.
- i is one of the output variables.
- SL_i is a slice on output variable i.
- SL_{int} is the intersection of slices for each of the output variables.

4.3 Best Coding Practices

The development of source code goes beyond compilation according to its purpose of operating the software. There are countless ways to implement a source code solution,

Metric	Formula	Description	Meaning
Coverage(M)	$\frac{\sum_{i=1}^{\|V_O\|} \frac{\|SL_i\|}{\|M\|}}{\|V_O\|}$	$\frac{sum\ of\ slice\ sizes}{\#(output\ variables) \times module\ size}$	Mean of the ratios of each slice size to the size of the module.
MinCoverage(M)	$\frac{min_i \|SL_i\|}{\|M\|}$	$\frac{minimum\ slice\ size}{module\ size}$	The ratio of the size of the smallest slice to the module size.
MaxCoverage(M)	$\frac{max_i \|SL_i\|}{\|M\|}$	$\frac{maximum\ slice\ size}{module\ size}$	The ratio of the size of the largest slice to the module size.
Overlap(M)	$\frac{\sum_{i=1}^{\|V_O\|} \frac{\|SL_{int}\|}{\|SL_i\|}}{\|V_O\|}$	$\frac{\sum_i \frac{intersection\ size}{slice\ size_i}}{\#(output\ variables)}$	Mean of the ratios of intersection size to the size of each slice in the module.
Tightness(M)	$\frac{\|SL_{int}\|}{\|M\|}$	$\frac{size\ of\ slice\ intersection}{module\ size}$	The ratio of the slice intersection size to that of the module.

Figure 16 – Mathematical description of the cohesion metrics set

Source: Green et al. (2009)

which is done according to each person's programming logic (GAUDENCIO; DANTAS; GUERRERO, 2014). While all solutions may be correct, some may be more complex than others. Consequently, the coding characteristics will be different, and the quality of the software will also change (LEFFINGWELL, 2007). Therefore, it is interesting to establish a minimum of rules that are called best coding practices. This set of rules can help decrease the software's complexity and improve its readability.

Some practices already consolidated include the proper source code indentation, intuitive nomenclature of variables and functions, and using comments, among others. Fowler (2018) created a catalog of refactorings listing the main coding best practices. Most of these practices apply to both structured and object-oriented languages, some of which will be presented in this section.

4.3.1 Refactoring Functions and Variables

Although it is related to fundamental knowledge, functions and variables are items that structure the source code and occur all the time in development. Therefore, writing these blocks of source code is important in the most readable way possible.

- Extract Function is one of the most common refactorings. It should check the purpose

of a code fragment and then move it to its own function (LI, 2011; LIEBIG et al., 2015). Some practical experience tips are: the functions must not be larger than the monitor screen. Any code used more than once must be placed in a function, but the code used only once must remain in the program's body (FOWLER, 2018). In general, roles should be large enough to perform a single task. Anything more than that, it starts to smell. There was a time when it was concerned with building small functions to save computational costs. Currently, this processing is negligible for the computer. Now the concern is with the readability and reuse of the source code.

- Inline Function is the inverse of the Extract Function. When the function's body is as simple as its name, then that code snippet can become part of the main function again (LIEBIG et al., 2015). The excess of functions that perform very simple tasks can confuse those who read the source code (FOWLER, 2018). In this case, the ideal is to combine all these small functions and then think about new refactoring of the Extract Function. It is important to find all references to the function and replace them with the function body to do this refactoring.

Another aspect related to the functions is nomenclature. These names need to add the intention with which the role was created. They should be named according to "*what it does*" and not "*how it does it*" (FOWLER, 2018). If it cannot think of a good name, maybe that code snippet should not become a function. The better the names, the more self-documented the code will be. Many source code comments, written to explain a function, can be condensed into the function name. The source code calls and instructions can be so well written that writing comments are less and less necessary.

- Change Function Declaration is mainly concerned with writing the function signature (FOWLER, 2018). This signature must have a name consistent with the function's task and must have good parameter names. The parameters determine how a function can fit the rest of the code. This should happen so that it is possible to understand the function's implementation only when reading its signature. Depending on the context, it may be more interesting to receive data primitive to the function parameters' data structure. In addition to allowing a function to be used more often, this practice allows for reducing coupling.

A code fragment cut from the main function to an auxiliary function must be independent of its variables. Therefore, it is important to pay special attention to the variables. All local variables within the scope of the main function, that are used in the auxiliary function must be passed by parameter.

- Extract Variable is used for the complexity inherent in lines with many self-contained tasks (REIMANN et al., 2012). In this sense, creating local variables can simplify a

given line of code's complexity. This makes the code more manageable and makes it easier to understand what is going on. Besides, these variables also favor debugging the source code. When the function's logic is broken into variables with good nomenclature, the code becomes more self-explanatory, and comments become unnecessary (FOWLER, 2018).

- Inline Variable is the inverse of the Extract Variable. When the attribution value is free of side effects, it is a sign that the new variable may not be playing a significant role in understanding the code (FOWLER, 2018). Instead, this variable may confuse the programmer and take up unnecessary memory space. When eliminating the declaration of a variable, it is important to refactor the code by assigning the variable in all references to the variable (REIMANN et al., 2012).

4.3.2 Simplifying Conditional Logic

Conditional structures are the first to bring more complexity to the source code. Therefore, it the importance to apply techniques to simplify these structures.

- Decompose Conditional is the transformation of the entire logical content body of the conditional into functions (PRETE et al., 2010). A function is created for each part of the conditional structure (*if-elif-else*) using the Extraction Function rule(FOWLER, 2018). Each piece of code will receive a function name that shows the intention. This decomposition procedure gives clarity to a code that was previously complex to read.

- Consolidate Conditional Expression is the use of logical operators (*and, or*) to aggregate conditionals with the same resulting action (COUNSELL et al., 2013; FOWLER, 2018). For example, in some cases, a sequence of independent *ifs* conditions are tested with the same resulting value. Then, it can add all these conditionals using the "*or*" operator. Another case would be to have two nested conditionals, so using the "*and*" operator would be appropriate.

- Replace Nested Conditional with Guard Clauses is a way to make a block of conditional structures more emphatic about its purpose (COUNSELL et al., 2013). For example, all clauses appear equally important in a structure that uses *if-then-else*. In cases where all clauses are equally likely, this method is appropriate. However, guard clauses would be more elucidating if the *else* clause were less important. The guard clause indicates the core of the conditional structure, and if it is triggered then, it must perform its task and leave the function (FOWLER, 2018). The implementation

rule is that a guard clause replaces the outermost clause (*else*). One observation is that if the custody clauses return the same value, it will be possible to implement the practice of Consolidate Conditional Expression.

4.3.3 Refactoring Parameters

- Parameterize Function is joining two functions that perform similar tasks (FOWLER, 2018). This will make it possible to increase the function's usefulness, as it can be applied in different locations. In some cases, it is necessary to adapt the parameters received to make the function more generic.

- Remove Flag Control means removing an argument that the function uses to indicate which logic should be executed. (FOWLER, 2018). This technique makes use of conditional structures to test the values. Boolean flags can be even more confusing – they do not convey meaning to the reader (COUNSELL et al., 2013). Removing the signaling arguments makes the code clearer and better prepares the code for tool analysis. Therefore, it is indicated that an explicit function replaces each flag value. This can be done using the practice of Decompose Conditional. For all times that the literal value of the flag was used, the function call is performed.

4.4 Final Remarks

In this chapter, we present the aspects that involve feature engineering, which is fundamental for understanding this doctoral thesis. We summarized the basic concepts related to Information Retrieval, and the operation of regular expressions. We also did a comprehensive review of the different software quality metrics applied in structured and object-oriented languages. Four categories of metrics were addressed: size, complexity, coupling, and cohesion. Finally, we describe some best practices for developing source code that can be applied in structured languages.

Software quality metrics are important features to characterize source code tasks and make discoveries regarding student clustering. There are tools like Sonar and SonarQube[14] that provide comprehensive features to evaluate and improve code quality in software projects. These tools are designed to continuously scan and analyze source code, detect bugs, security vulnerabilities, violations of coding standards, and other issues that may affect the maintenance and performance of the software. However, based on the bibliographic research carried out, software quality metrics have been used so far to characterize developers working in the industry (KOZIK et al., 2019; SILVA et al., 2019). Bearing in mind that

[14] The SonarQube is available at: https://www.sonarqube.org/

the characteristics of professional developers' software are more advanced than students who are starting in programming, we observe a potential contribution to the adaptation of these metrics to verify students' progress throughout the learning topic.

Another important point in software development is the best coding practices. They are taught in CS1 courses from the first days of class and are from the simplest to the most advanced (BUSE; WEIMER, 2010), but they are all fundamental to the organization and readability of the source code. To the best of the authors' knowledge, the best coding practices are evaluated manually in a qualitative way (POSNETT; HINDLE; DEVANBU, 2011). Therefore, we did not find automatic metrics to measure the readability of the source code developed by students. We believe that it would be relevant, in the future, to create features that verify simple programming aspects, such as indentation and source code comments if students are in the first weeks of class. More advanced coding practices should follow the student's progress in the course, for example, aspects related to the nesting of conditionals and loopings.

The next chapter explores the works related to this thesis. In particular, we report some literature reviews that put computer education in evidence. Then, we introduce the existing tools and approaches to support teaching-learning programming. We seek to identify the advantages, disadvantages, and context in which each tool operates. Special attention was paid to machine learning tools for student monitoring and assessment. Finally, we report research that uses software quality metrics as features.

Chapter 5

RELATED WORK

According to the Survey by Douce, Livingstone and Orwell (2005), tools that support students in learning programming have been developed since the 1960s. Nowadays, the majority of the tools can be classified into the following categories (SILVA et al., 2019): Learning Environment, Visualization and Animation (of the functioning of the algorithms), Block Programming, Similarity Detection, Students' Feedback, Students' Grade and Students' Performance. Regarding the existing tools, it should be noted that the approach proposed in this research does not fit any of these existing categories because it was developed with a different purpose in mind. Our approach was developed to support teachers in their pedagogical decisions in the programming classroom.

The obstacles faced by students in their experiences with programming cannot always be closely monitored by the teacher. An aggravating factor is an increase in the number of students enrolled, which makes it difficult for the teacher to provide a personalized intervention. Like other areas of research, teaching-learning programming is challenging and constantly evolving. The use of technology can make the teaching-learning process more efficient in the sense that learning happens and is meaningful to the student. The literature presents a diversity of tools that help students to evolve in their programming tasks (KEUNING; JEURING; HEEREN, 2018). However, these tools are mainly focused on student feedback. Teachers are far from adapting these tools to their needs.

This chapter presents the works related to the research developed in this doctoral thesis. In this context, we will discuss how our research relates to the "closest" existing tools from the different tools categories. Then, we present three research fronts to discuss the related works. First, students' grading approaches provide a means to grade tasks automatically. Second, students' performance approaches provide a means of verifying student outcomes. Third, students' grouping approaches provide a means of grouping students using some criteria.

5.1 Students' Grade

The evaluation of source code is a critical aspect in software development and programming education. Two primary methodologies employed for source code assessment are static analysis and dynamic analysis. Static analysis entails evaluating the code without execution, thereby providing insights into its structural characteristics and potential issues. This evaluation encompasses a spectrum of techniques, including the utilization of syntactic analysis, structural analysis, similarity detection, and software quality metrics (BASILI; BRIAND; MELO, 1996). By analyzing the code's structure, complexity, and adherence to coding standards, static analysis aids in identifying potential vulnerabilities, code smells, and adherence to best practices. Conversely, dynamic analysis necessitates the execution of the program to assess its behavior during runtime. This approach evaluates how the program responds to various inputs and scenarios, thereby revealing performance bottlenecks, memory leaks, and runtime errors (ERNST et al., 1999). By simulating real-world conditions, dynamic analysis provides valuable insights into the code's runtime behavior and efficiency, facilitating the identification of potential issues that might not be apparent through static analysis alone.

The pursuit of automated methods for grading programming tasks has been a subject of extensive research, aiming to alleviate the burden on educators and streamline the assessment process. Among the prevalent approaches, the utilization of test cases has emerged as a common method for evaluating source code tasks. Test case-based assessment leverages predefined input scenarios and expected outputs to objectively evaluate the correctness and functionality of the provided code (SHARMA; SAHA, 2018). Edwards and Perez-Quinones (2008) introduced *Web-CAT*, an automated grading tool designed to assess source code tasks by employing test cases. A noteworthy feature of *Web-CAT* is its encouragement of student involvement in crafting their own test cases for the tasks they complete. This collaborative aspect not only enhances students' understanding of the material but also contributes to a comprehensive evaluation of their programming skills. Similarly, Polito, Temperini and Sterbini (2019) devised *2TSW*, a gamified tool tailored for teaching C Language programming. In *2TSW*, teachers can incorporate new tasks, each necessitating the creation of corresponding test cases. Students' interaction with these tasks serves to advance their in-game accomplishments, while the central focus of the study lies in automated task correction and the provision of personalized error feedback. Notably, the evaluation process in these approaches hinges upon validated test cases, underscoring the importance of adhering to predefined input-output standards.

In contrast to the prevalent test case-based evaluation approaches, Liu et al. (2019) present an innovative perspective through the *AutoGrader* tool. *AutoGrader* employs a reference implementation strategy, automatizing the assessment of programming tasks by

focusing on the semantics of the program. This approach diverges from the conventional binary evaluation criterion, allowing for a more nuanced assessment that considers the overall program behavior and logic. The endeavor to assign partial credit and rectify minor syntax errors in source code has also garnered attention (PARIHAR et al., 2017). Furthermore, modern machine learning techniques, such as convolutional neural networks, have been harnessed to validate grades assigned manually by instructors, exemplifying the integration of methodologies into the assessment domain (SOUZA; ZAMPIROLLI; KOBAYASHI, 2019).

An alternative technique that emerged in the automated grading landscape is Parsing-based Automated Assessment (*PAAA*), proposed by Tianyi et al. (2019). *PAAA* harnesses parsing techniques to automate the grading process of source code tasks, with a particular focus on tasks involving the Python programming language. Remarkably, the adaptability of *PAAA* to various programming languages showcases its potential applicability across diverse educational contexts. Abstract Syntax Tree (AST), a key component of programming language processing, is effectively employed by Porfirio, Pereira and Maschio (2021) to automate the assessment of students' programming skills. AST-based grading facilitates a deeper understanding of the code's structural intricacies, enabling a more comprehensive evaluation of students' comprehension and implementation. For an in-depth exploration of tools and techniques encompassing static, dynamic, and hybrid approaches to automated assessment, the works by Ullah et al. (2018) and Galan et al. (2019) provide valuable insights into the landscape of automated grading methodologies and their implications.

5.2 Students' Performance

In addition to the aforementioned concerns, another pertinent matter revolves around the identification of students in need of assistance throughout the programming course. Substantiating this, existing evidence highlights the correlation between initial and final course performance, implying the persistence of early trends throughout the learning journey (OZTURK; BONFERT-TAYLOR; FUGENSCHUH, 2018). Moreover, the attainment of successful outcomes in tests has been leveraged as a means to ascertain student performance.

Fonseca, Macedo and Mendes (2018) introduced *CodeInsights*, an online tool that uses a *plug-in* installed in the programming environment to receive coding information (PHP, Java, or Python) and provides performance notifications of the students to teachers. Source code tasks are compiled and tested using the automated *black-box* testing software engineering procedure (NIDHRA; DONDETI, 2012). Information was collected, such as an unusual number of code lines, compilation errors, attempts per assignment, assignments not

attempted, and unfinished assignments. Performance is calculated based on the percentage of source code tasks completed correctly over a while – the student is given a label related to their pace (slow, intermediate, or fast).

Benotti et al. (2018) presented *Mumuki*, an open-source online editor, in a process that, for each source code task, the system explains the theory and a programming example that involves the concepts needed to solve the source code task. The teacher is responsible for manually defining the test cases and can also define auxiliary functions. The tool includes an interactive console on which solutions and reusable functions can be performed, and automatic feedback is provided according to the tests. The system supports 17 languages, including Python, JavaScript, and C.

Ihantola et al. (2015) mapped using machine learning techniques and analysis techniques for teaching-learning, to assist teachers and students in programming. Ahadi et al. (2015) used decision trees and Bayesian networks to predict student performance, while Castro-Wunsch, Ahadi and Petersen (2017) used neural networks for the same purpose. A diverse set of 10 features was used by (AHADI et al., 2015) to implement their classifier. Among the features are the student's average in other courses, the passage through test cases, the number of steps used to solve the tasks, and a categorical feature that indicates students' major. Castro-Wunsch, Ahadi and Petersen (2017) focused on just two features: *Steps*, which is the number of submissions for each task, and *Correctness*, which is the success fraction in the test cases.

The utilization of the linear regression technique has garnered considerable attention in the realm of educational assessment. It has been adeptly employed to discern students exhibiting weaker academic performance (MUNSON; ZITOVSKY, 2018) and to prognosticate the likelihood of student attrition from the course (OZTURK; BONFERT-TAYLOR; FUGENSCHUH, 2018). Munson and Zitovsky (2018) conducted a comprehensive study encompassing the incorporation of over 100 distinct features, meticulously curated to encapsulate critical facets. These factors encompass a spectrum of variables, including the extent and frequency of academic engagement, resolution of compilation errors, source code dimensions, temporal patterns of coding activities, frequency and duration of interludes, among other influential parameters. In parallel, Ozturk, Bonfert-Taylor and Fugenschuh (2018) harnessed the potential of a keystroke-level analysis tool, facilitating the meticulous capture of an expansive set of 300 source code features. A prominent highlight of their investigation resides in the identification of a significant correlation existing between two key features: the temporal investment of students in task completion and the efficacy of task resolution rates. This interplay highlights the dynamics governing students' coding behavior and its subsequent implications on task accomplishment.

5.3 Students' Grouping

Unsupervised machine learning techniques have gained widespread utilization across diverse educational domains, serving as a potent tool to unearth insights (YADAV et al., 2014; ILIC et al., 2016; RANA; GARG, 2016; PURBA; TAMBA; SARAGIH, 2018). These techniques harness student-related attributes as features, encompassing demographic information, class attendance (ALFIANI; WULANDARI et al., 2015), historical grades (POLYZOU; KARYPIS, 2019), and indicators of academic performance (OYELADE; OLADIPUPO; OBAGBUWA, 2010). The burgeoning interest in leveraging unsupervised machine learning techniques to unveil intricate patterns and relationships within educational data highlights the potential for novel insights. However, the extant body of literature addressing the confluence of student clustering and programming in the pedagogical context appears to be relatively limited.

In the endeavor to establish programming pairs, Aottiwerch and Kokaew (2018) developed a web platform that suggests the pairs from a questionnaire that students answer in advance. This questionnaire covers three themes: *(i)* programming skills that consist of a basic programming test; *(ii)* learning behavior that involves issues of attitude, motivation, planning, knowledge-seeking and *(iii)* self-assessment (KHALIL et al., 2017), and finally, behavioral interoperability involving communication and teamwork skills (RODRÍGUEZ; PRICE; BOYER, 2017). The information collected was used as features for a partition clustering algorithm that resulted in four clusters: good and bad students in all respects, students with high, and low performance in programming skills.

Regardless, Ahadi et al. (2017) used clustering to predict student performance in the final example of the CS1 course. The course was taught in Java, and the learning content taught was basic commands, variables, conditionals, looping, methods, lists, and objects. A software unit test was used to verify the students' average performance. The average performance was considered the degree of correctness in the students' weekly source code tasks. It was analyzed whether or not the students' performance was consistent. For this, static analysis was performed to quantify students' weekly performance. Finally, it was verified whether or not the students' performance changed from one week to another. For this, a transition diagram was used based on the student's average performance. Clustering was then applied to verify which students would be more likely to drop out of the course. These three features were used: average performance, consistent performance, and transition. As a result, it was identified that student performance decreases over the course.

Anand et al. (2018) intended to group programming students based on previous student performance. They used the K-means clustering algorithm and extracted three features. The features used were the students' grades in the prerequisite courses, Cumulative

Grade Point Average (CGPA), and the grades in the current course. The interesting point is that they fixed three groups (below average, average, and above average). Given the importance of grouping students according to their programming skills to support the teacher's pedagogical decisions, a student grouping survey was carried out. Silva and Silla (2020) processed a database similar to ours in a computational model of hierarchical clustering. Text-based features were used that were linked to programming language keywords. However, the extracted features were too generic to represent the learning topics. In addition, all the source code tasks of a term were processed at once. The main limitation was the inability of the teacher to intervene in the teaching-learning process before the end of the academic period.

5.4 Aspects of the Approaches

Table 6 compares our research and the related works, listing the main learnings found in each work. In the first six columns, we present and summarize some characteristics of the approaches described above. These characteristics address: *(i)* the bibliographic identification of the research; *(ii)* the type of application targeted by the research, which we classified into one of the three fronts discussed above: grade, performance, or grouping; *(iii)* the number of students involved in the experiment; *(iv)* the programming language used to experiment, even if the approach supports other languages; *(v)* the number of features used, which in some cases can be very comprehensive and involve all programming language keywords or use software tests as a feature; *(vi)* and the database used. In the database, we identify the number of submissions of a task or the number of compilations that may involve many occurrences, as a task can be corrected and submitted or compiled numerous times. We identified databases that use the number of different assignments, regardless of student submissions. Finally, the number of source codes is the same as what we considered in our experiments.

In the last 10 columns, we present the approaches classified as having (✓) or not having (✗) a particular aspect. Therefore, we raise potentials and limitations observed in these approaches under the focus of this thesis.

- **Provides solution** is when the tool offers the correct answer to a problem after a series of unsuccessful attempts by the student. Conversely, our model's purpose does not encompass direct correction, thereby allowing students the autonomy to iteratively develop and rectify their assignments without external intervention. These assignments can be submitted through any designated platform as per the teacher's guidelines. Furthermore, we advocate for a restricted access policy to the clusters, ensuring that only the teacher possesses the privilege of accessing the clustering

results.

- **IDE freedom** is the autonomy of teachers to select a development environment without being bound by specific tools or constraints. Within the framework of our model, the chosen integrated development environment remains inconsequential, as our processing methodology operates directly on the underlying source codes, which are presented in plain text format. As such, our approach transcends the influence of IDE preferences, emphasizing the core aspects of code analysis and evaluation. This highlights the versatility of our model in accommodating diverse pedagogical preferences, fostering an environment where teachers can exercise their discretion in choosing IDEs.

- **New tasks** are the ability for teachers to generate their own source code tasks autonomously, free from tool-specific constraints. In alignment with this principle, our model possesses the capability to autonomously identify and extract predetermined features from the source code of any programming challenge crafted in the C Language. This extends the flexibility of our approach beyond the confines of a particular tool, enabling integration with a diverse array of programming problems. As a result, teachers are empowered to curate tailored tasks without being constrained by tool dependencies, ultimately fostering an adaptable and versatile pedagogical environment.

- **Compilation** is when the tool compiling the code task to facilitate the analysis. In contrast, our model is designed to operate effectively regardless of whether the code compiles or whether it is deemed correct or incorrect. Our model possesses the capability to group students based on their programming skills, irrespective of the compilation outcome. Given that weaker students might struggle to generate compilable code, our focus is primarily directed towards identifying programming skills within the code that can aid and support the student's learning process.

- **Partial evaluation** is the process wherein the tool undertakes a nuanced assessment of the source code task, taking into account partial completion. Within the framework of our model, we possess the capacity to meticulously analyze and categorize students, even in instances where their assignments are not fully completed. This approach acknowledges the potential educational value of assessing partial efforts, providing a comprehensive understanding of students' progress and skills, regardless of task completion status. By incorporating partial evaluation, our model contributes to a more holistic and inclusive evaluation process that highlights the significance of incremental learning and individualized growth.

- **Lots of data** is when the tool necessitates an extensive volume of data to execute the analysis. However, our model distinguishes itself by its minimal data requirement

to conduct the analysis effectively. A notable advantage of this approach is that teachers can swiftly garner a comprehensive overview of the class's performance early in the academic period, immediately following students' completion of a source code task. This expedited insight into students' programming capabilities facilitates timely and informed instructional decisions, enabling teachers to tailor their teaching strategies according to the collective skill level exhibited by the classroom.

- **Different languages** is the capability to accommodate a variety of programming languages. In the context of our model, the utilization of language-specific keywords serves as a mechanism for locating and extracting relevant features, establishing a pronounced association with the C Language. The prospect of extending our model's applicability to alternative programming languages entails recognizing analogous patterns and subsequently implementing the requisite extractor module tailored to the new language's syntax and semantics. This process hinges on the identification of parallel constructs and the translation of our feature extraction mechanism to ensure the transferability of our model's functionalities to diverse programming languages.

- **Semantic independence** is when the tool dissociates itself from semantic considerations while identifying code similarity. In contrast, our model focuses on analyzing the sequence of commands and syntactic patterns generated by the student. Instances of task replication, where deliberate modifications are made to variable names or additional line breaks are introduced to distinguish the tasks, are treated equivalently by our model. This approach ensures a consistent evaluation, where the primary emphasis remains on syntactical alignments rather than semantic nuances, offering a streamlined and robust assessment process.

- **Student's history** is the ability to construct a chronological representation of a student's development over the course of various learning topics. In alignment with this principle, our model possesses the capacity to construct a comprehensive timeline that charts the evolution of students' skills across different programming tasks. This temporal perspective enables us to monitor the trajectory of students as they transition between different groups, discerning whether their programming skills are converging towards or diverging from established benchmarks. By affording us the means to track students' progression within the context of their learning history, our model enriches our understanding of their growth and achievements, facilitating informed pedagogical interventions and tailored support.

- **Reference value** is when the tool relies on or permits the input of a reference source code for processing students' code submissions. In contrast, our model operates independently of a reference value to yield its results. However, the option to input a source code or parameterize feature values provides teachers with the opportunity

to conduct a more comprehensive analysis of our outcomes. This facet within our model is referred to as the "Gold-Standard", signifying a deeper level of exploration and evaluation enabled by this resource.

Table 6 – Where is this research compared to the state of the art?

Research	Application[*]	# Stud.	Language	Features	Database	Provides solution	IDE freedom	New tasks	Compilation	Partial evaluation	Lots of data	Different languages	Semantic independence	Student's history	Reference value
Edwards and Perez-Quinones (2008)	GD	-	Java, C++	test case	-	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓
Parihar et al. (2017)	GD	410	C	55	15.613 submissions	✓	✗	✓	✓	✓	✗	✗	✓	✓	✗
Polito, Temperini and Sterbini (2019)	GD	11	C	test case	49 codes	✓	✗	✓	✓	✗	✗	✗	✓	✓	✓
Liu et al. (2019)	GD	-	C	key words	10.270 submissions	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓
Souza, Zampirolli and Kobayashi (2019)	GD	150	Java	key words	938 codes	✗	✓	✓	✓	✓	✓	✗	✗	✓	✗
Porfirio, Pereira and Maschio (2021)	GD	-	C	27	3977 codes	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗
Tianyi et al. (2019)	GD	61	Python	key words	61 codes	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗
Ahadi et al. (2015)	SP	296	Java	10	2.000 codes	✗	✗	✗	✗	✓	✓	✗	✓	✓	✗
Castro-Wunsch, Ahadi and Petersen (2017)	SP	146	Java, Python	23	1416 codes	✗	✗	✗	✗	✓	✓	✓	✓	✓	✗
Ozturk, Bonfert-Taylor and Fugenschuh (2018)	SP	95	C	300	68.248 submissions	✓	✗	✗	✓	✓	✓	✗	✓	✓	✗
Munson and Zitovsky (2018)	SP	86	Java	100	61.684 compilations	✓	✗	✓	✓	✓	✓	✗	✗	✓	✗
Fonseca, Macedo and Mendes (2018)	SP	27	Java, Python	5	51 tasks	✗	✗	✓	✓	✗	✗	✓	✓	✓	✗
Benotti et al. (2018)	SP	114	Haskell	test case	82 tasks	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓
Moresi, Gómez and Benotti (2021)	SP	75	Haskell	4[**]	67 tasks	✗	✓	✓	✗	✓	✓	✗	✓	✗	✗
Ahadi et al. (2017)	GR	89	Java	9	-	✗	✓	✓	✓	✓	✗	✗	✓	✓	✗
Anand et al. (2018)	GR	200	C++	3	-	✗	✓	✓	✗	-	✗	✗	✗	✓	✗
Silva and Silla (2020)	GR	34	C	46	630 codes	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗
This research (2023)	GR	41	C	24	713 codes	✗	✓	✓	✗	✓	✗	✗	✓	✓	✓

[*] Application legend: Grading (GD), Student Performance (SP), Grouping (GR).
[**] They also used a Bag-of-Words (BoW) to perform word count in natural language processing.

5.5 Final Remarks

In this chapter, we present the works related to this doctoral thesis. We present several aspects in related works, but our main interest is the source code features. It is well known that building a good machine learning model depends fundamentally on selecting good features (GARRETA; MONCECCHI, 2013). Within software engineering research, some studies attempt to define metrics that characterize software development. An example is Bassi et al. (2018), which presented a set of software quality metrics to measure the contribution of development team members. The metrics are related to the source code's complexity, inheritance, coupling, and size. Other works have also explored software quality metrics as features for programming solutions (KASTO; WHALLEY, 2013; WHALLEY; KASTO, 2014). Regardless, the relationship between source code readability and software quality was explored by Buse and Weimer (2010). We approach state-of-the-art on three research fronts involving teaching programming: students' grade, students' performance, and students' grouping.

These are just some teaching-learning programming research perspectives that report the challenges over time. Overall, the common purpose of all related works is to identify students who need assistance in the course. In contrast, their differences are usually related to the information collected and the techniques used to produce the results. Our main focus of study is the grouping of programming students. While the integration of unsupervised machine learning techniques into educational research is discernible, the specific intersection of student clustering with programming pedagogy is relatively underexplored. Some studies found try to predict the students' final grade (AHADI et al., 2017), and those try to form programming pairs as an alternative to accelerate learning (AOTTIWERCH; KOKAEW, 2018).

In our computational model, we wanted to develop an approach that would be helpful to teachers to have insights into the student's programming skills since the class's first task. Based on those insights, the teacher would then be able to use different pedagogical approaches, such as active learning-based pedagogical approaches, to help improve student learning. Most of the different pedagogical approaches involve pairing at least two students. However, these students must be similar or dissimilar in some aspect (depending on the pedagogical approach). Therefore, the teacher could make pairs or groups using the students from the same group (when similar skills are necessary) or from different groups (when dissimilar skills are required).

To the best of the author's knowledge, there seems to be no report in the literature on a model that uses clustering algorithms to monitor student progress from their source code solutions designed to support the teacher in his pedagogical decisions throughout the academic period. Therefore, we observe research potential in this theme. To find

practical use in teachers' daily lives, we developed a survey to consider student progress across learning topics. In this way, we present our hypothesis: "It is possible to develop a computational approach to monitoring student progress from their source code solutions in a programming course". To achieve our objective, we created features based on software metrics that represent each learning topic present in the exercise lists in our database.

The next chapter presents the research methodology used to develop this doctoral thesis. We present the characterization and structure of our research, and we detail the configuration of our experiments.

Chapter 6

METHODOLOGICAL APPROACH

This chapter presents the methodological approach used to conduct the research and achieve the objectives proposed in this doctoral thesis. First, the research is characterized by aligning its conceptual underpinnings with the well-established definitions and concepts in the literature. This initial step serves as a robust foundation for delving into the methodological approach employed to undertake the research and accomplish the predetermined objectives outlined in this doctoral thesis. Next, the research structure is defined. We used the Knowledge Discovery in Databases (KDD) guidelines to explain the tools used to implement the method and the data collection process. This entails a detailed explication of the mechanisms and protocols put in place to procure the requisite data that fuels the empirical inquiry. Finally, the configurations used to carry out the experiments of this doctoral thesis are detailed. In this context, paramount emphasis is placed upon elucidating the interplay of features, settings, and variables that were selected and aligned with the overarching research objectives.

6.1 Research Characterization

The scientific method is an instrument formed by procedures to formulate scientific problems and examine scientific hypotheses (GALLIANO, 1979). In our research, we defined the characterization of the scientific method based on the concepts presented in Jung (2004). The scientific method is classified according to three points (JUNG, 2004): *(i)* in terms of nature, *(ii)* in terms of objectives, and *(iii)* in terms of procedures. In this sense, this research is characterized as applied in nature, with descriptive and explanatory objectives that use experimental research procedures and strategies. Figure 17 helps to understand the methodological choices made in this research.

As for the nature of research, applied (or technological) research is used to acquire theoretical knowledge and obtain a solution for practical application, such as, for example,

a new product, a process, or a patent. In our case, we use the knowledge acquired in computing to apply it in education to develop a computational model to monitor students' progress in different programming learning topics.

As for the research objectives, descriptive research aims to explore a certain topic and classify the relationship between the variables, expanding its understanding. For this, as much information as possible is collected to infer a population statistically, such as obtaining percentages, averages, frequency, and correlation coefficient, among other possibilities (GILL, 2002). In explanatory research, the objective is to understand the causes and effects of a given phenomenon from the ideas identified. In our case, we identified a set of features related to each learning topic. In addition, we carried out a descriptive analysis of the groups of students that were formed after the grouping.

In terms of research procedures, experimental research endeavors to methodically manipulate relevant variables in a comprehensive, systematic, and impartial manner to explore novel techniques and approaches. In our research, we specifically manipulated the source code features to establish their correlation with students' progress over the academic period.

6.2 Research Structuring

The structure of our research follows the Knowledge Discovery in Databases (KDD) process guidelines. We use KDD to support the development of our experiments methodologically. KDD is an iterative process that aims to extract knowledge and identify understandable patterns from large databases (HAN; PEI; KAMBER, 2006). Knowledge discovery involves a sequence of five steps, where one step outcome depends on the other, and each step can be repeated over and over again.

The structure of our research addresses: *(i)* Data Selection organizes the database and defines the Gold-Standard; *(ii)* Data Preprocessing prepares the data; *(iii)* Data Transformation includes feature extraction; *(iv)* Data Mining defines the machine learning technique; and, finally *(v)* Interpretation and Evaluation occurs the outputs interpretation and the model evaluation. Figure 18 presents the necessary steps to run our computational model and obtain groups of students with their respective skills. The description of each step of the execution flow of our computational model is presented in the following subsections.

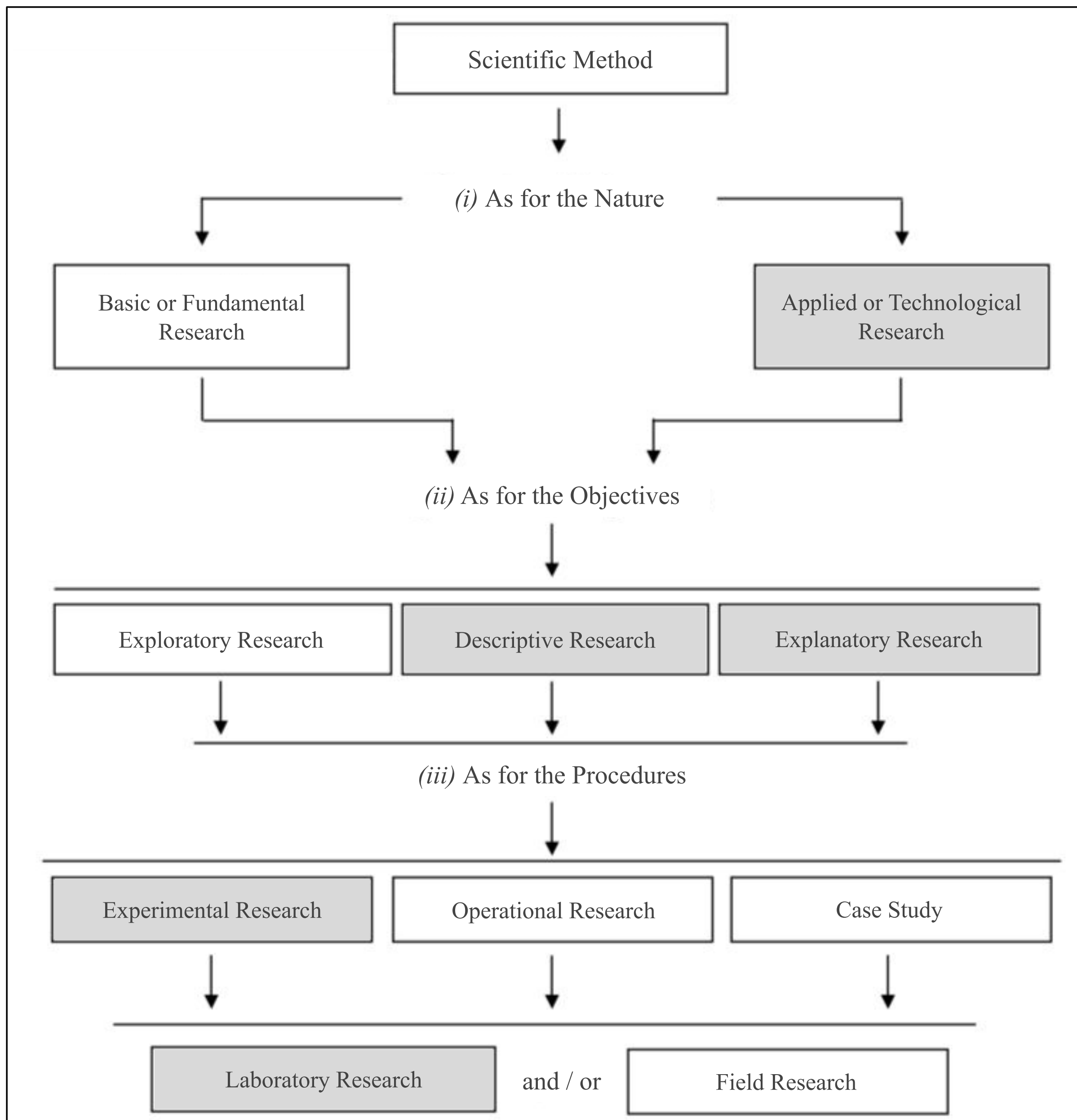


Figure 17 – Source code example that averages three grades

Source: Adapted from Jung (2004)

6.2.1 Data Selection

In the Data Selection step, the application domain is learned. Also, in the same step occurs the selection and segmentation of the data to be analyzed. In this sense, the data can come from different sources and formats.

In this step, we prepare the database in our computational model. The database must essentially consist of source code tasks. As expected by the programming language compiler, these tasks must be in plain text format (e.g., *task1.c*). The operating system directory structure that stores these tasks must also respect a hierarchical organization.

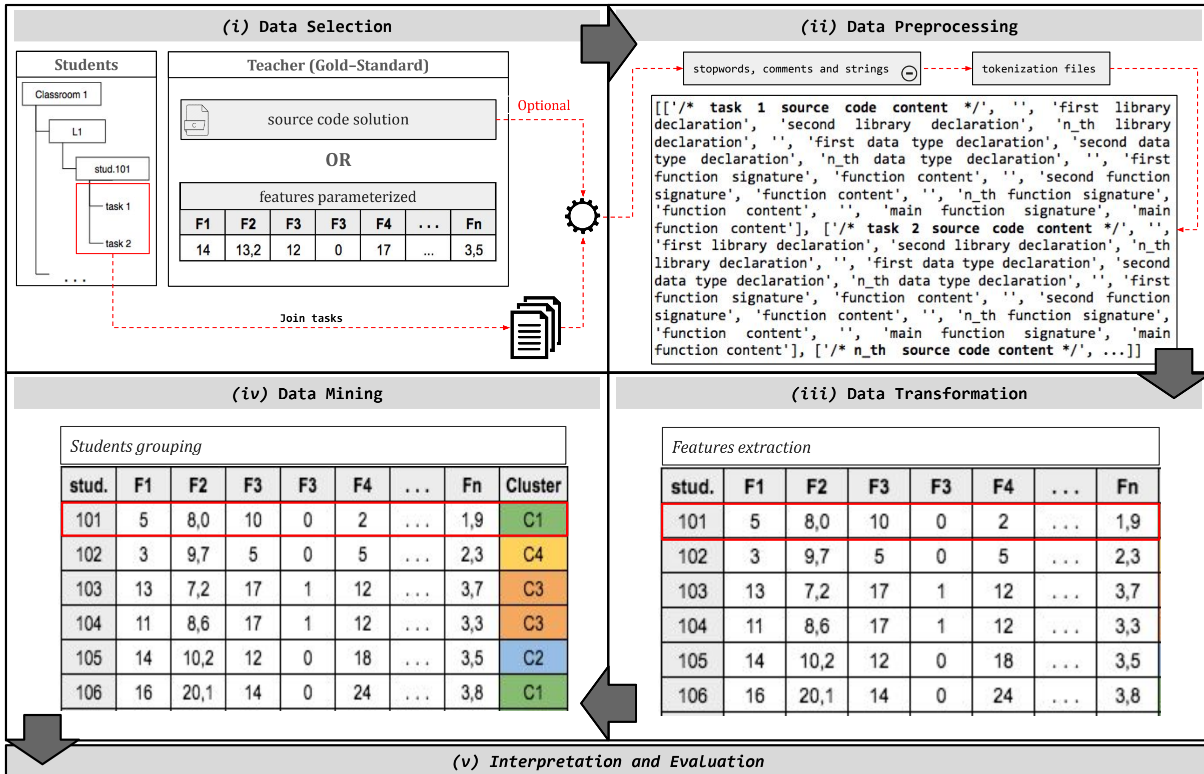


Figure 18 – Execution flow of our computational model

Source: The author

This structure allows the model to maintain the link between students and tasks, as well as the learning stage within the course.

The root of this structure is the classroom taught, that is, the class of students, while the child nodes of a classroom are the learning topics that reflect the lists of exercises developed in the course. Then, the students that make up a classroom are nested within each exercise list. Finally, student source code solutions for the exercise lists are found in the leaf nodes of the directory structure.

After organizing the database, we define the reference values for the tasks in the learning topic. In addition to the source code solutions submitted by students, our model can receive a unique reference implementation for each task. This reference solution is a Gold-Standard that the course teacher developed. This way, the Gold-Standard is the set of results the teacher expects for the tasks in a learning topic. There are two ways the teacher can send this reference to our model. The first is the source code solution written by the teacher. Then, our model does the code processing and extracts the features to use as Gold-Standard. In the second, the teacher informs each feature value to the model in a parameterized way. Our model works with or without the Gold-Standard. However, this reference provides a general evaluation of source code solutions developed by students. Values above or below the expected may represent positive or negative conditions based on the teacher's interpretation. We performed this analysis in our experiment and compared

it to the learning outcomes established for our course.

6.2.2 Data Preprocessing

In the Data Processing step, the quality of the stored data is checked. Among the operations performed in this step, the main one is cleaning the stored data. In this sense, some data can be removed if they are identified as inconsistent or discrepant (outliers). This step can also make strategic decisions to fill in missing data fields.

In this step, we preprocess the source code solutions in our computational model. Each learning topic is preprocessed separately and sequentially. This preprocessing policy is important to preserve the temporal order in which each learning topic occurs. The preprocessing is performed directly on the directory structure, that is, on the source code solutions developed by each student. Each student's solutions (from a learning topic) are concatenated into a single large text file. After that, source code comments are identified and removed to maintain student anonymity. Then the words from this large file are identified and tokenized to be stored in a linked list. Then, the feature extractor uses this linked list of source code tokens. If there is any empty source code solution or with the occurrence of features that go beyond normality, this task can be considered an outlier. The teacher can identify these outliers after the Data Transformation step. When observing an outlier, one returns to this stage, and the source code solution must be removed from the database not to generate noise in the grouping.

6.2.3 Data Transformation

In the Data Transformation step, data is reduced and transformed to facilitate its use and navigation. The data must be standardized and thus adapted to Data Mining techniques. Among the applied transformations, normalization, aggregation, creation of new attributes, reduction, and data synthesis stand out. In this sense, the most important attributes of the application domain must be identified and classified. In addition, reducing the number of attributes performed in this step can increase the model's effectiveness.

In this step, we perform feature extraction from source code solutions in our computational model. The source code features were defined based on the subjects taught in each learning topic. In this way, the features of each learning topic were identified in the lists of exercises solved by the students. The important point is that another learning topic can reuse the features defined for a learning topic. The teacher predefined the criterion for reusing features, respecting the importance of the learning topic. Then, the features are counted according to the feature extraction criteria that we present in the proposed features. Afterward, the features are stored in a term-document matrix. Each feature

is a term that represents a programming skill, and each student is a new vector space document. The term-document matrix with the features is used to group students.

6.2.4 Data Mining

In the Data Mining step, machine learning techniques are constructed or applied to obtain new knowledge. Some aspects involved are the choice of algorithm for data processing and the definition of settings related to the technique, such as distance measurements. In clustering, choices must be made so that the cluster represents the selected data.

In this step, we apply machine learning techniques to group students according to their programming skills in our computational model. Our model aims to provide insight into students' abilities without the teacher having to open source code solutions. We use an unsupervised learning algorithm because these codes were not previously classified and did not have a label. A distance calculation that uses the extracted features verifies the distance between students in vector space. Students are grouped according to the proximity of this distance, which refers to the similarity between their programming skills. By default, our model creates four clusters, each representing the student's level of learning topic: very skillful, skillful, medium, and unskilled.

6.2.5 Interpretation and Evaluation

In the Interpretation and Evaluation step, the knowledge extracted from the data is carefully examined, interpreted, and validated. The primary objective is to gain meaningful insights and ensure the accuracy and reliability of the computational model.

During the interpretation phase, experts or domain specialists closely analyze the results obtained from the data mining process. They leverage their expertise and domain knowledge to interpret and understand the patterns, trends, and relationships discovered by the model. This interpretation helps to extract valuable knowledge and insights that can be applied to real-world problems or decision-making processes.

Validation, on the other hand, aims to assess the quality, reliability, and generalizability of the computational model. It involves various evaluation techniques to measure the performance, effectiveness, and robustness of the model. These techniques can include statistical measures, cross-validation, hypothesis testing, and comparing the model's predictions against real-world data or known outcomes.

By conducting thorough interpretation and validation, the KDD process ensures that the extracted knowledge is accurate, meaningful, and useful for making informed decisions or solving complex problems. It helps to establish the credibility and reliability of the computational model, providing confidence in its applicability and effectiveness.

6.3 Experimental Settings

Figure 19 contextualizes how we approach the teaching process in our research. First, the teacher presents a theoretical part of the learning topic in question. Also, part of this step is solving source code tasks as an example for students. Then, a list of exercises is proposed for the students to fix the subjects of the learning topic. Students solve tasks and submit them on a teaching platform used by the teacher. These tasks are used as input to our computational model that extracts the features and groups the students. A table with the features obtained per student and their respective clusters is the output of our computational model. From this table, the teacher can make decisions regarding teaching.

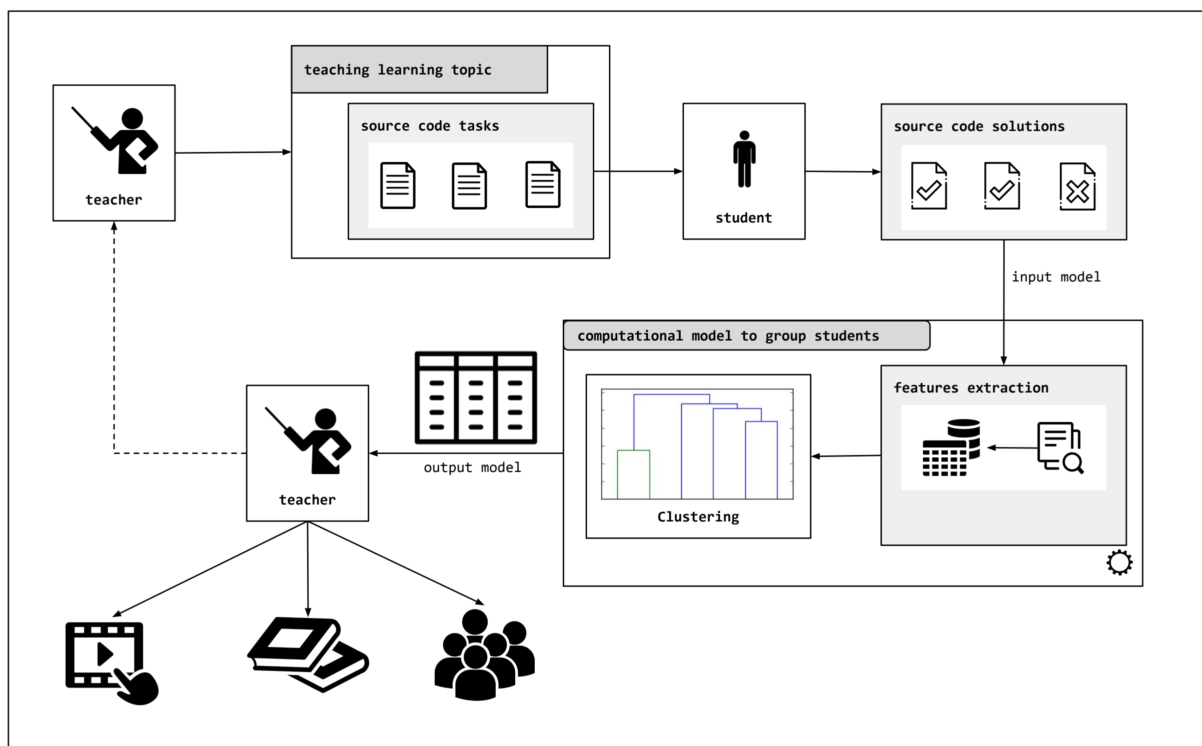


Figure 19 – Conceptual diagram of the teaching process

Source: The author

The clustering task has been used in many areas to join related elements (JAIN, 2010). Usually, the dataset is not labeled, so the clustering approach is classified as an unsupervised machine learning technique. In this way, the elements are clustered according to the similarity of the information available (ZHU; GOLDBERG, 2022). Knowledge of the application domain is imperative to the success of the model. The results are highly sensitive to the input parameters, such as the measure of similarity, the chosen clustering algorithm, and the employed feature set. In this section, we will explore the process of constructing our model and address these topics.

6.3.1 Research Questions

In this research, we present an approach to clustering students according to their respective programming skills that were extracted from previously submitted source code tasks. Our experiments were divided into five categories, which are represented by their respective research questions:

RQ$_1$ How to group students with similar skills using automatic analysis of the source code with machine learning?

In **RQ**$_1$, we aim to describe the application of a clustering algorithm to a set of source code solutions and analyze their results. For this, we use an agglomerative hierarchical clustering algorithm present in the machine learning library *Scikit-learn*[15], available for Python programming language.

RQ$_2$ How to assess students' individual programming skills?

In **RQ**$_2$, we intend to present the strengths and weaknesses of each student from the extracted features. Also, we want to see if the clusters represent related ways of thinking. To enrich the results, we performed a descriptive statistical analysis of the submission information of the source code solution.

RQ$_3$ Were the extracted features significantly different in each learning topic?

In **RQ**$_3$, we aim to determine if there are significant differences in the extracted features across different learning topics. To evaluate this, we conducted statistical tests to assess the statistical significance of our findings. Specifically, we applied the *Kruskal-Wallis* test to determine if there is a significant difference among the clusters generated by our clustering algorithm. We also employed the *Mann-Whitney* statistical test to identify specific differences between the clusters (CORDER; FOREMAN, 2014). By using these statistical tests, we can provide evidence of whether the extracted features exhibit statistically significant variations across the learning topics, enhancing the validity and reliability of our results.

RQ$_4$ What is the correlation between the features extracted in each learning topic?

[15] Available in https://scikit-learn.org

In $\mathbf{RQ}_4$, we apply a correlation coefficient to the extracted features to assess the level of linkage between them. This analysis aims to explore the existence of relationships or dependencies among the features and understand how they relate to each other. By using the correlation coefficient, we seek to identify whether the extracted features are positively, negatively, or insignificantly correlated. This investigation is crucial for understanding the interactions and interdependencies among the features and can provide valuable insights for our study.

$\mathbf{RQ}_5$	How to get an overview of programming skills from student groups?

In $\mathbf{RQ}_5$, we want to present an overview of the strengths and weaknesses of each student group. The aim is to verify if the clusters are being organized from the evolution of learning. In this way, the teacher will identify the clusters of students who need more attention on a particular learning topic.

6.3.2 Database Description

Our database includes source code solutions written by students in the C Language. These source codes are organized by learning topics, representing a real-world environment where students progress throughout the academic period.

Table 7 presents the characterization of the database used in this research. Our database contains real-world data from a programming course organized at an educational institution in southern Brazil in 2016.

Table 7 – Description of the source code task database

ID	Topics	# Tasks	# Stud.	# Codes
T1	functions (by value)	6	35	189
T2	functions (by reference)	4	26	94
T3	data structures	5	32	141
T4	recursive functions	4	26	89
T5	dynamic allocation	6	28	160
Total source code solutions submitted				**673**

The classroom had 16 weeks of face-to-face classes, including theoretical and practical classes in the computer lab. Five learning topics were taught during the course, following the chronological order presented in Table 7. To fix the subject, the students solved the learning topics tasks (T1 to T5). Each learning topic comprises a set of source code tasks ranging from 4 to 6 problems. Thus, 25 source code tasks were proposed for the classroom. The difficulty level of the tasks progressively increases as the learning topic

progresses. An example of a statement for a function with passing parameters by value task is presented in Figure 20.

> *"Write a program that takes two integers. Then, create a function to calculate the sum of these two numbers. Finally, print the result in the main function".*

Figure 20 – An example of a source code statement from T1

Source: The author

The number of students who submitted at least one of the source code solutions and the total amount of source code processed for each topic is also shown in Table 7. It is important to note that a student can submit at least one solution to a learning topic without submitting any solution to another topic. In all, the classroom had 41 students enrolled who developed a total of 673 source code solutions. These students were in Higher Education and had already taken an introductory module where they learned about variables, conditionals, control flow, and loops.

6.3.3 Clustering Algorithm

Data cluster analysis consists of an exploratory data analysis, where a set of clusters share common features (HARRINGTON, 2012). The constituted clusters must obey a division where the cluster points must be as similar as possible.

According to Halkidi, Batistakis and Vazirgiannis (2001) and Jain, Murty and Flynn (1999) the main clustering approaches can be classified into Partitional clustering, Hierarchical clustering, Density-Based clustering, and Grid-Based clustering. Partitive clustering algorithms require that calculations involving the initialization of centroids be performed repeatedly. They are more efficient in terms of memory usage and are faster when the database is large. On the other hand, hierarchical clustering algorithms are more flexible, as they allow data points to be clustered according to their features without the need to initialize the centroids. Density-Based clustering, as explored by Ester (2018), focuses on identifying clusters with regions of high data density. On the other hand, Grid-Based clustering, explored by Cheng, Wang and Batista (2018), proves to be highly effective with large multidimensional datasets.

In summary, each clustering approach offers distinct advantages and is best suited for specific types of datasets, making them valuable tools for different clustering scenarios. Considering the size of our database, which is relatively small, we opted for the agglomerative hierarchical clustering algorithm. This choice was driven by the limitations encountered with partitional methods and the inherent ease of visualization offered by the dendrogram, allowing for a clearer understanding of the data's hierarchical structure.

6.3.3.1 Distance measures between students

Clustering algorithms need a measure to calculate the distance between two objects. Distance measures between clusters are updated with each iteration, and the algorithm looks for the closest point to join a new cluster. They are called similarity or dissimilarity measures (ROKACH; MAIMON, 2005).

We tested two measures to calculate the distance between objects used to define the cluster structure (RANI; SAHU, 2017): the *Cityblock distance* and the *Euclidean distance*. Each metric was applied from the data matrix, in which each row represents an object. Then, the second distance matrix is calculated; each matrix element corresponds to a quantitative measure of the proximity between two pairs of objects.

6.3.3.2 Distance measures between hierarchical clusters

Hierarchical clustering algorithms require the configuration of the linkage method that defines how the distance between clusters will be calculated. The three main methods to calculate the proximity between two clusters are: Single Linkage, Average Linkage, and Complete Linkage.

We calculated the similarity between the clusters using the three linkage methods. To evaluate the best behavior configuration, we interleaved each linkage method with the two distance measures (Cityblock and Euclidean).

6.3.4 Model Interpretation and Evaluation

As cluster analysis is used to discover patterns in the data, the database usually does not include the labels with the true data classification. In this sense, the evaluation of the model is carried out through specific methods to verify the quality of the resulting clusters. However, cluster evaluation essentially depends on the data interpretation by a professional who knows the application domain.

6.3.4.1 Interpretation of extracted features

In the feature extraction step, our model generates a term-document matrix where it is possible to carry out a detailed investigation of each student's abilities. However, in our final results, we provide the teacher with an overview of each group of students. This overview is the midpoint index that calculates the average of the feature values that were obtained in each cluster. Subsequently, we used the Gold-Standard reference provided by the teacher to interpret the midpoint.

There is a comprehensibility cost in normalizing the features (i.e., performing the experiments with and without feature normalization). We have run a preliminary investigation using *L2-normalization*, and the student clusters did not change. However, it has become harder to understand what is happening with each feature due to the normalization.

For this reason we decided not to normalize the extracted features as the obtained visualization has become of the research's contribution. The analysis of actual values provides the teacher with the possibility to obtain valuable insights.

6.3.4.2 Cluster internal validation

We apply an internal validation index to check the quality of the resulting clusters. In this sense, the *Cophenetic Coefficient Correlation* (CARDONA et al., 2013) was used, which correlates linearly with similarity matrices and Cophenetic Coefficient. Equation 6.1 defines the *Cophenetic Coefficient cp*, where x is the distance and t is the node's height where the points meet.

$$cp = \frac{\sum_{u<v} \left(x\left(u,v\right) - \bar{x}\right)\left(t\left(u,v\right) - \bar{t}\right)}{\left(\left[\sum_{u<v} \left(x\left(u,v\right) - \bar{x}\right)^2\right]\left[\sum_{u<v} \left(t\left(u,v\right) - \bar{t}\right)^2\right]\right)^{1/2}} \tag{6.1}$$

The resulting matrix must correlate with the similarity matrix for a valid dendrogram. The coefficient value can range from 0.00 to 1.00, the closer to 1.00, the higher the internal quality of the cluster.

6.3.4.3 Definition of the number of clusters.

The hierarchical algorithms allow the definition of the number of clusters after obtaining the result of the clusters. However, the resulting output from our model requires that a number of clusters have been chosen. With this number of clusters, the model can automatically generate the students' individual skills and their respective discovered clusters. Therefore, we set four clusters as the default value for our model's number of clusters automatically generated. This value was chosen based on four groups of expected students: very skilled, skilled, average, and unskilled.

In addition to dividing the clusters into four groups based on classroom experience, we interpret a dendrogram to confirm the acceptable number of clusters in the dataset. To verify this division's consistency, we apply the *Elbow method*, which is used to examine the percentage variation between clusters (LIU et al., 2010). The purpose is to add new clusters to the model to the point where the information gain is no longer significant. This point is identified on the graph as an "elbow".

Furthermore, the hierarchical clustering algorithms provide a graphical tree representation of the results that shows the order and distances between the clusters formed during clustering. This feature allows the teacher to track how the student groups are hierarchically organized. In this way, the teacher can analyze the difference between students within the same cluster to see how heterogeneous the students can be.

It is important to note that our computational model allows the number of clusters to be parameterized. In this way, the number of clusters can be changed according to the teacher's strategy. Students who did not submit a source code solution for a given learning topic were not considered in the grouping.

6.3.4.4 Statistical analysis

We use a nonparametric statistical test to perform our statistical analyses. The nonparametric test was chosen because of the characteristics of our database. In some learning topics, our database has a sample smaller than 30 students ($N < 30$). In addition, our database meets the prerogatives of the nonparametric test as it is a non-homogeneous sample and does not have a normal distribution. Since students from one cluster cannot belong to another, we apply the test for k-independent samples. We used IBM SPSS Statistics Software[16] to perform the statistical analysis.

The *Kruskal-Wallis* test is used when we have three or more independent groups and want to determine if there are significant differences in the medians of these groups. It is a non-parametric extension of the 1-way ANOVA test, which is used to compare means of independent groups (MCKIGHT; NAJAB, 2010). The *Kruskal-Wallis* test is appropriate when the assumptions of the ANOVA test are not met, particularly when the data does not follow a normal distribution or when the data is measured on an ordinal scale. We applied the *Kruskal-Wallis H-test* statistical test to verify whether the groups of students were significantly different. To run the test, we check the total features of each student to use as a test item and use the clusters as test group identifiers. We run the test for each learning topic separately. Finally, we compare the samples individually by applying *Mann-Whitney U-tests* (MCKNIGHT; NAJAB, 2010).

Besides, we compared the samples individually using a post-hoc test to find differences between groups. One way to do the post-hoc test is to apply multiple *Mann-Whitney U-tests* (MCKNIGHT; NAJAB, 2010). The problem with doing multiple *Mann-Whitney U-tests* is that it increases the chance of a type 1 error. Thus, correcting the value of p by the *Bonferroni* factor is necessary. We apply the *Kruskal-Wallis H-test* to avoid this problem by making two-by-two comparisons. The post-hoc test using multiple comparisons returns the adjusted p for the number of comparisons performed.

[16] The IBM SPSS Statistics Software is available at: https://www.ibm.com/spss

Finally, we used the bivariate (2-tailed) correlation to verify the relationship involving the two variables. Given the characteristics of our database, we need to apply a nonparametric correlation coefficient. With the correlation, we intend to verify if there is a relationship between the features defined for each learning topic. The coefficient used was *Spearman's rho* correlation.

The *Spearman's* rank correlation coefficient, also known as *Spearman's rho*, provides a measure of the correlation between two variables on an ordinal scale. The coefficient ranges from -1 to +1 and indicates the strength and direction of the correlation (SIEGEL, 1956). The resulting output of *Spearman's* rank correlation is a single value within this range. A value close to +1 indicates a strong positive correlation, while a value close to -1 indicates a strong negative correlation. On the other hand, a value close to 0 suggests a weak or no linear correlation between the variables. In interpreting the *Spearman's* rank correlation coefficient, Cohen (1988) suggests that a correlation value of 0.50 or higher can be considered large, indicating a substantial relationship between the variables. An average correlation falls between 0.30 and 0.50, suggesting a moderate relationship. Conversely, a correlation value of 0.30 or less suggests a small or negligible relationship.

6.4 Final Remarks

In this chapter, we present the methodological approach that has been employed in this doctoral thesis. Our aim is to provide a comprehensive understanding of the methodology adopted to guide our research endeavors. We commence by delineating the fundamental characteristics of our research, including its nature, objectives, and procedural aspects. By doing so, we set the stage for a comprehensive comprehension of the foundational principles that underpin our study.

Moreover, we elucidate the structure that governs our research, aligning our approach with the systematic stages of the Knowledge Discovery in Databases (KDD) process. Within this framework, we outline the steps undertaken to construct our computational model and elucidate the process through which our data were collected and organized. This provides a transparent overview of the rigor in our research efforts.

Subsequently, we turn our attention to the configuration of our experiment, an essential facet that underpins the acquisition of meaningful results. This section delves into the specifics of how our experiment was designed, highlighting the factors that were considered to ensure the accuracy and reliability of our findings. Additionally, we offer insights into the dataset employed, shedding light on the bedrock upon which our analyses were conducted.

The next chapter presents the synthesis of the feature set corresponding to each

learning topic. Before running our experiments, it was imperative to construct a comprehensive set of source code features. These defined features were tailored for every individual learning topic, with the overarching objective of extracting relevant insights from the students' source codes. In the subsequent sections, we delve into the intricacies of this constructed feature set, expounding on the rationale that underpins its architecture and the strategic decisions that guided its formulation.

Chapter 7

PROPOSED FEATURES

This chapter presents the definition of a set of features that are directly related to the subjects in each learning topic. The proposed features are presented in Table 7 and described in this chapter. The goal is that these features represent the programming skills reflected in the source code instructions programmed by the students. In this way, the teacher will be able to identify the students' strengths and weaknesses without manually grading the source code solutions.

In all, we defined 21 features for the course's five learning topics. Each learning topic is represented by exercise lists applied during the course. Learning topics include: functions with parameter passage by value (T1), functions with parameter passage by reference (T2), struct (T3), recursion (T4), and dynamic allocation (T5). We will use the acronym "*topic.feature*" to simplify identifying features within the learning topic whenever necessary – the acronym T1.NF is an example to refer to the number of functions of the first learning topic.

We discuss each feature's importance in the practice of teaching programming and present how each feature contributes to learning. The distribution of features among the learning topics is shown in Table 8. In addition, we describe how the features were extracted at the source code level. Our computational model was developed in Python, but we formalized the main features in pseudocode and presented them in this chapter.

7.1 Features for T1 (Functions by Value)

The learning topic corresponding to T1 allows students to understand the functions with passing parameters by value. We have defined four new proprietary features to investigate student's strengths and weaknesses in the T1 learning topic. We adapted two of these features used in the object-oriented paradigm (HENDERSON-SELLERS, 1996) (OLIVEIRA et al., 2008) for the context of structured programming.

Table 8 – Correspondence between learning topics and features

ID	Feature	T1	T2	T3	T4	T5	Adapted from
1	NF	✓					Henderson-Sellers (1996)
2	NP	✓					Oliveira et al. (2008)
3	NC	✓					
4	NR	✓					
5	NPt–s		✓				
6	NAdd–s		✓				
7	NPt–ds		✓				
8	NAdd–ds		✓				
9	NStr			✓			
10	NStrM			✓			
11	NStrT			✓			
12	NStrI			✓			
13	NStrC			✓			
14	NFRec				✓		
15	NCRec				✓		
16	NIFPar				✓		
17	NRRec				✓		
18	NRNRec				✓		
19	NMalloc					✓	
20	NSizeof					✓	
21	NFree					✓	

The extracted features are desirable and complementary criteria to the learning topic and meet the following learning outcomes: 1) Number of Functions: the ability to create functions; 2) Number of Parameters: the ability to understand passing parameters by value; 3) Number of Calls: the ability to call student-created functions; and 4) Number of Returns: the ability to understand return types.

7.1.1 Number of Functions

The Number of Functions (NF) represents the number of functions the student developed. This feature allows the teacher to verify the student's understanding of the functioning of the functions and, consequently, the degree of organization of the source code. The more functions are developed, the greater the chance that the source code is modularized. According to good programming practices, modularizing source code increases the possibility of source code being readable and reusable (FOWLER, 2018).

Algorithm 4 shows how our computational model obtained the NF feature. This algorithm inputs the source code task that the student has developed and a list of data types. These data types correspond to the function's return, which appears at the beginning of a function's signature in C Language. The list of data types is parameterized and can

be redefined according to teacher preferences. We default use the following data types: void, int, float, double, and char. The algorithm's output counts the number of functions found in the source code. In the execution of this algorithm, a looping goes through all the lines of the source code. For each line, we ignore leading spaces and look for the data types that have been defined. The data types are checked on the first word found on the current line. If any data type is found, we look for the characters "open parentheses" and "close parentheses" on the same line where the data type was found. Finally, we check for "open braces" on the same or next line. We identify a function when all these criteria are met. All content found between the braces represents the structure's function and is stored in a list for future use (ls_functions). Then the current line is stored in a signature list. The important point is that we only consider the new functions created by the students and disregard the main function (which must exist in any source code in C Language).

Algorithm 4: GET_NF: NUMBER OF FUNCTIONS

```
   Input: code //source code the student developed
   Input: return_types //set of data type
   Output: counter ∈ ℕ //number of functions
 1 begin
 2 |  initialize counter → 0;
 3 |  while do not finish the code lines do
 4 |  |  while do not finish the return_types do
 5 |  |  |  if type in line then
 6 |  |  |  |  if parentheses in line then
 7 |  |  |  |  |  if braces in line or braces in next line then
 8 |  |  |  |  |  |  increments the counter;
 9 |  |  |  |  |  end
10 |  |  |  |  |  break; //exit return_types looping
11 |  |  |  |  end
12 |  |  |  end
13 |  |  end
14 |  end
15 |  return counter;
16 end
```

7.1.2 Number of Parameters

The Number of Parameters (NP) represents the number of parameters present in the functions the student developed. This feature allows the teacher to check the student's understanding of the variable's scope. It is necessary to transfer this variable to the context of the subroutine in question to use the value of a local variable in a subroutine. A smaller number of parameters than expected may mean that the code needs to meet the minimum requirements for the correct functioning of the program. On the other hand, the student

may use global variables to make them accessible to all the source code. Another possibility is for the student to use a data structure to store more than one value and thus save parameters.

Algorithm 5 shows how our computational model obtained the NP feature. This algorithm inputs the source code task that the student has developed and a list of data types. The algorithm's output counts the number of parameters found in the source code. In C Language, parameters are normally found on the same line as the function declaration. For this reason, we used the Algorithm pseudocode 4 to obtain the function signatures in the student's source code. Then, a looping goes through each function signature. Finally, we identify the arguments inside the parentheses. We use the commas as the delimiter of these arguments and perform the raw count of occurrences.

Algorithm 5: GET_NP: NUMBER OF PARAMETERS

```
   Input: code //source code the student developed
   Input: return_types //set of data type
   Output: counter ∈ ℕ //number of parameters
 1 begin
 2     initialize counter → 0;
 3     list_signatures → get_NF(code, return_types);
 4     while do not finish the list_signatures do
 5         if not empty within the parentheses then
 6             counter accumulates the total of parameters delimited by comma;
 7         end
 8     end
 9     return counter;
10 end
```

7.1.3 Number of Calls

The Number of Calls (NC) represents the number of calls to the functions the student developed. This feature allows the teacher to check if the functions created by the student are being used. A number of calls less than the number of functions may mean that more than necessary functions have been created. On the other hand, a much higher number of calls means that a function is being used more than once.

Algorithm 6 shows how our computational model obtained the NC feature. This algorithm inputs the source code task that the student has developed and a list of data types. The algorithm's output counts the number of calls to functions found in the source code. First, we retrieve the list of function signatures as in Algorithm 4. Then, a looping goes through each function signature. From the signature, we identify the function name, which is delimited by the first space to the first parenthesis of the function signature. We

search each line of source code for the function name. We discard all lines that are the function signature itself. We also discard occurrences that refer to function prototypes. Prototypes are characterized by a ";" at the end of the line instead of an "open braces". We identify a call to the function when all these criteria are met.

Algorithm 6: GET_NC: NUMBER OF CALLS

```
   Input: code //source code the student developed
   Input: return_types //set of data type
   Output: counter ∈ ℕ //number of calls
1  begin
2      initialize counter → 0;
3      list_signatures → get_NF(code, return_types);
4      while do not finish the list_signatures do
5          while do not finish the code lines do
6              if function name in line then
7                  if line is not a prototype then
8                      if line is not a signature then
9                          increments the counter;
10                     end
11                 end
12             end
13         end
14     end
15     return counter;
16 end
```

7.1.4 Number of Returns

The Number of Returns (NR) represents the number of return commands the student used in his source code. This feature allows the teacher to identify the student's understanding of the variable's scope. Variables declared locally are valid only within the function in which they were declared. For this reason, it is necessary to return to the main function to retrieve the value of a local variable that has been modified.

Algorithm 7 shows how our computational model obtained the NR feature. This algorithm inputs the source code task that the student has developed and the command that will be searched. The algorithm's output counts the number of return statements found in the source code. A looping goes through the source code. We search each line of source code for the return keyword. We identify a return command when these criteria are met.

Algorithm 7: GET_COMMANDS: GENERIC FUNCTION FOR COMMANDS

```
    Input: code //source code the student developed
    Input: command //keyword command
    Output: counter ∈ ℕ //number of commands
1 begin
2     initialize counter → 0;
3     while do not finish the code lines do
4         if command in line then
5             increments the counter;
6         end
7     end
8     return counter;
9 end
```

7.2 Features for T2 (Functions by Reference)

The learning topic corresponding to T2 enables the student to understand the functions with parameter passage by reference. We defined four new proprietary features to investigate the weaknesses and strengths of students in the T2 learning topic.

The extracted features are desirable and complementary criteria to the learning topic and meet the following learning outcomes: 5) Number of Simple Pointers: the ability to create pointers; 6) Number of Simple Address: the ability to pass parameters by reference; 7) Number of Array Pointers: the ability to understand the use of array pointers; and 8) Number of Array Addresses: the ability to pass parameters by reference of array-type.

7.2.1 Number of Simple Pointers

The Number of Simple Pointers (NPt–s) is the number of primitive data type pointers the student used in his source code. This feature allows the teacher to identify if the student is declaring pointers in the source code writing. Understanding pointers is a determining factor for future topics learning such as dynamic memory allocation. In addition, pointers are used to return more than one value in the function and are references to lists, stacks, and trees in the data structure.

Algorithm 8 shows how our computational model obtained the NPt-s feature. This algorithm inputs the source code task that the student has developed and a list of data types. The algorithm's output counts the number of pointers of the primitive data type found in the source code. First, we retrieve the list of function signatures as in Algorithm 4. Then, a looping goes through each function signature. Later, we identify the placeholder for passing parameters within the signature of each function. Then, within the parameters, we find the prefix referring to the pointer operator (*) and perform the raw count of

occurrences.

Algorithm 8: GET_NPT-S: NUMBER OF SIMPLE POINTERS

```
   Input: code //source code the student developed
   Input: return_types //set of data type
   Output: counter ∈ ℕ //number of simple pointers
1  begin
2  |  initialize counter → 0;
3  |  list_signatures → get_NF(code, return_types);
4  |  while do not finish the list_signatures do
5  |  |  increments the counter with the number of asterisk in the line;
6  |  end
7  |  return counter;
8  end
```

7.2.2 Number of Simple Address

The Number of Simple Address (NAdd–s) is the number of addresses for pointers of the primitive data type the student used in his source code. This feature allows the teacher to identify if the student is passing the address of the pointers created in the function calls of its source code. The NAdd-s result must be equal to or higher than the NPt-s result. The value is higher when a function with passing parameters by reference is called more than once.

Algorithm 9 shows how our computational model obtained the NAdd-s feature. This algorithm inputs the source code task that the student has developed and a list of data types. The algorithm's output counts the number of addresses for pointers of the primitive data type found in the source code. First, we retrieve the function call list as it was in the Algorithm 6. Then, a looping goes through each call to the function. Later, we identify the placeholder for passing parameters within the signature of each function. Then, within the parameters, we find the prefix referring to the address operator (&) and perform the raw count of occurrences.

7.2.3 Number of Array Pointers

The Number of Array Pointers (NPt–ds) is the number of composite data type pointers the student used in his source code. In this case, the data type refers to the homogeneous data structure, such as arrays. This feature allows the teacher to identify if the student uses data arrays to declare pointers.

Algorithm 10 shows how our computational model obtained the NPt-ds feature. The extraction process for this feature is similar to NPt-s. The difference is that we loop

Algorithm 9: GET_NADD-S: NUMBER OF SIMPLE ADDRESS

```
   Input: code //source code the student developed
   Input: return_types //set of data type
   Output: counter ∈ ℕ //number of simple address
 1 begin
 2     initialize counter → 0;
 3     list_calls → get_NC(code, return_types);
 4     while do not finish the list_calls do
 5         increments the counter with the number of ampersand in the line;
 6     end
 7     return counter;
 8 end
```

through a list of calls to the functions. Then, we identify the suffix referring to the array identifier in the function parameters instead of the pointer operator.

Algorithm 10: GET_NPT-DS: NUMBER OF ARRAY POINTERS

```
   Input: code //source code the student developed
   Input: return_types //set of data type
   Output: counter ∈ ℕ //number of array pointers
 1 begin
 2     initialize counter → 0;
 3     list_signatures → get_NF(code, return_types);
 4     while do not finish the list_signatures do
 5         while there are words between commas in the signature do
 6             if open square brackets in word and close square brackets in word then
 7                 increments the counter;
 8             end
 9         end
10     end
11     return counter;
12 end
```

7.2.4 Number of Array Addresses

The Number of Array Addresses (NAdd–ds) is the number of addresses for composite data type pointers the student used in his source code. In this case, the data type refers to the homogeneous data structure, such as arrays. This feature allows the teacher to identify if the student is passing data arrays in function call arguments. The NAdd-ds result must be equal to or higher than the NPt-ds result. The value is higher when a function with passing parameters by reference is called more than once.

Algorithm 11 shows how our computational model obtained the NAdd–ds feature. The extraction process for this feature is more complex than the previous ones because the

array-type arguments do not receive a specific markup. We use the relationship of function signatures and calls to find out the exact name of the function. Later, we identify if there is any array between the local variables. Then, we check if any of the identified arrays are used in the arguments of the function calls to perform the raw count of occurrences.

Algorithm 11: GET_NADD-DS: NUMBER OF ARRAY ADDRESSES

```
   Input: code //source code the student developed
   Input: return_types //set of data type
   Output: counter ∈ ℕ //number of array addresses
 1 begin
 2     initialize counter → 0;
 3     list_signatures → get_NF(code, return_types);
 4     list_parameters → get_NP(code, return_types);
 5     while do not finish the code lines do
 6         if line is an array-type local variable then
 7             while do not finish the list_signatures do
 8                 while do not finish the list_parameters do
 9                     if array-type in parameter then
10                         increments the counter;
11                     end
12                 end
13             end
14         end
15     end
16     return counter;
17 end
```

7.3 Features for T3 (Data Structures)

The learning topic corresponding to T3 enables the student to understand heterogeneous data structures. We defined five new proprietary features to investigate the weaknesses and strengths of students in the T3 learning topic.

The extracted features are desirable and complementary criteria to the learning topic and meet the following learning outcomes: 9) Number of Structs: the ability to create heterogeneous data structures; 10) Number of Struct Members: the ability to understand how heterogeneous data structures work; 11) Number of Struct Typedefs: the ability to rename heterogeneous data structures; 12) Number of Struct Instances: the ability to instantiate new heterogeneous data structures; and 13) Number of Struct Calls: the ability to call heterogeneous data structures.

7.3.1 Number of Structs

The Number of Structs (NStr) is the number of heterogeneous data structures the student developed. This feature allows the teacher to identify if the student is developing their own set of variables within a data structure.

Algorithm 12 shows how our computational model obtained the NStr feature. We use the source code the student developed to find lines that start with the keyword *struct* to extract this feature. Afterward, we look for "braces open" on the same line or the next line where the structure was identified. All content found between the braces represents the structure's body and is stored in a list for future use (ls_structsBody). Finally, we perform the raw count of occurrences for the items in the list.

Algorithm 12: GET_NSTR: NUMBER OF STRUCTS

```
   Input: code //source code the student developed
   Output: counter ∈ ℕ //number of structs
 1 begin
 2   initialize counter → 0;
 3   while do not finish the code do
 4     if word struct in line then
 5       if line ends with open braces or next line then
 6         increments the counter;
 7       end
 8     end
 9   end
10   return counter;
11 end
```

7.3.2 Number of Struct Members

The Number of Struct Members (NStrM) is the number of elements of the heterogeneous data structure the student declared. This feature allows the teacher to identify the size and consistency of the heterogeneous data structures defined by the student.

Algorithm 13 shows how our computational model obtained the NStrM feature. We relate the heterogeneous data structures found in the student source codes to extract this feature. Then, we ignored the lines referring to the structure's signature to find only the declaration of the variables. Finally, we perform the raw count of each variable declared within the structure.

Algorithm 13: GET_NSTRM: NUMBER OF STRUCT MEMBERS

```
   Input: code //source code the student developed
   Input: ls_structsBody //body each struct saved in Algorithm 12
   Output: counter ∈ ℕ //number of struct members
 1 begin
 2 |  initialize counter → 0;
 3 |  while do not finish the list_structsBody do
 4 |  |  if line is not struct declaration then
 5 |  |  |  if line is not starts with open braces or not ends with close braces then
 6 |  |  |  |  while there are words between commas in the line do
 7 |  |  |  |  |  increments the counter;
 8 |  |  |  |  end
 9 |  |  |  end
10 |  |  end
11 |  end
12 |  return counter;
13 end
```

7.3.3 Number of Struct Typedefs

The Number of Struct Typedefs (NStrT) is the number of typedef commands the student used. This feature allows the teacher to identify if the student renames existing data structures. Sources become more readable and portable, as only *typedef* commands need to be changed if the structure is changed.

Algorithm 14 shows how our computational model obtained the NStrM feature. We use source codes the student developed to extract this feature. We look at the beginning of each line for the keywords "typedef struct". Then, we check if there is an end-of-statement indication (;) as the last letter in the line. Afterward, we capture the alias of each typedef and store it in a list. Finally, we perform the raw count of occurrences for the items in the list.

7.3.4 Number of Struct Instances

The Number of Struct Instances (NStrI) is the number of instances of the heterogeneous data structure the student used. This feature allows the teacher to identify if the student is instantiating structures defined in the source code.

Algorithm 15 shows how our computational model obtained the NStrI feature. To extract this feature, we list the aliases of the typedefs found in the student source codes. We then look for each alias at the beginning of each line of code and do the raw count of hits.

Algorithm 14: GET_NSTRT: NUMBER OF STRUCT TYPEDEFS

```
   Input: code //source code the student developed
   Output: counter ∈ ℕ //number of struct typedefs
1  begin
2      initialize counter → 0;
3      while do not finish the code do
4          if typedef struct in line then
5              if line ends with semicolon then
6                  while there are words between spaces in the line do
7                      increments the counter;
8                  end
9              end
10         end
11     end
12     return counter;
13 end
```

Algorithm 15: GET_NSTRI: NUMBER OF STRUCT INSTANCES

```
   Input: code //source code the student developed
   Output: counter ∈ ℕ //number of struct instances
1  begin
2      initialize counter → 0;
3      ls_typedefs → get_NStrT(code);
4      while do not finish the ls_typedefs do
5          while do not finish the code do
6              if typedef in line then
7                  increments the counter;
8              end
9          end
10     end
11     return counter;
12 end
```

7.3.5 Number of Struct Calls

The Number of Struct Calls (NStrC) is the number of accesses to instances of the heterogeneous data structure the student performed. This feature allows the teacher to identify if the student is using the instance structures. A call value less than the number of instances may mean that more structures were created than necessary.

Algorithm 16 shows how our computational model obtained the NStrC feature. We list the instances found in the student's source codes to extract this feature. Then, we look for each instance through the source code and perform the raw count of occurrences.

Algorithm 16: GET_NSTRC: NUMBER OF STRUCT CALLS

```
   Input: code //source code the student developed
   Output: counter ∈ ℕ //number of struct calls
 1 begin
 2     initialize counter → 0;
 3     ls_instances → get_NStrI(code);
 4     while do not finish the ls_intances do
 5         while do not finish the code do
 6             if intance in line then
 7                 increments the counter;
 8             end
 9         end
10     end
11     return counter;
12 end
```

7.4 Features for T4 (Recursive Functions)

The learning topic corresponding to T4 enables the student to use recursive functions. We defined five new proprietary features to investigate the weaknesses and strengths of students in the T4 learning topic.

The extracted features are desirable and complementary criteria to the learning topic and meet the following learning outcomes: 14) Number of Recursive Functions: the ability to create recursive functions; 15) Number of Recursive Functions Calls: the ability to call recursive functions; 16) Number of Parameters in Conditional: the ability to implement stop conditions in recursive functions; 17) Number of Recursive Returns: the ability to return values recursively; and 18) Number of Non-Recursive Returns: the ability to implement stopping conditions in recursive functions.

7.4.1 Number of Recursive Functions

The Number of Recursive Functions (NFRec) is the number of recursive functions the student developed. This feature allows the teacher to check if the student is implementing recursive functions. Recursive functions sometimes allow the student to create more readable and simpler source code.

Algorithm 17 shows how our computational model obtained the NFRec feature. We retrieved the function signature list and the function body list to extract this feature. Then, we collect the role name from the role signature. Finally, we look in the function body for the function's name. Bearing in mind that this verification will be used other times, we have developed Algorithm 18. If an occurrence is identified, we carry out the

count.

Algorithm 17: Get_NFRec: Number of Recursive Functions

Input: ls_functions //body each function saved in Algorithm 4
Output: counter ∈ ℕ //number of recursive functions

```
1 begin
2     initialize counter → 0;
3     while do not finish the list_functions do
4         if is_recursive_function(function) then
5             increments the counter;
6         end
7     end
8     return counter;
9 end
```

Algorithm 18: is_recursive_function: checks a recursive function

Input: function body
Output: boolean variable

```
1 begin
2     initialize counter → 0;
3     ls_signatures → get_NF(function);
4     while do not finish the function from line two do
5         if signature in line then
6             return True;
7         end
8     end
9     return False;
10 end
```

7.4.2 Number of Recursive Functions Calls

The Number of Recursive Functions Calls (NCRec) is the number of calls to recursive functions used by the student. This feature allows the teacher to check if the student is using the recursive functions that were created.

Algorithm 19 shows how our computational model obtained the NCRec feature. We retrieve the list with the body of the functions to extract this feature. Then we check which functions are recursive and collect the name of those functions. Next, we look through the source code for the lines containing the name of the recursive function, except the recursive function itself. Finally, we perform the raw count of occurrences.

Algorithm 19: Get_NCRec: Number of Recursive Functions Calls

```
   Input: code //source code the student developed
   Input: ls_functions //body each function saved in Algorithm 4
   Output: counter ∈ ℕ //number of recursive functions calls
 1 begin
 2     initialize counter → 0;
 3     while do not finish the list_functions do
 4         if is_recursive_function(function) then
 5             while do not finish the code do
 6                 if name function in line and line is not a signature then
 7                     increments the counter;
 8                 end
 9             end
10         end
11     end
12     return counter;
13 end
```

7.4.3 Number of Parameters in Conditional

The Number of Parameters in Conditional (NIFPar) is the number of conditional structures linked to the parameters of the recursive functions the student used. This feature allows the teacher to check if the student implements the stopping conditions in their recursive functions. Recursive functions need to have a stop condition to not run indefinitely. This condition is represented by the *IF* command and must return a value other than the recursive function.

Algorithm 20 shows how our computational model obtained the NIFPar feature. We rescued the list with the function's body to extract this feature. Then, we check whether the functions are recursive and store the function parameters. The stop condition must test some of the parameters that were used as input to the recursive function, so we carry out this check. When these criteria are met, we perform a raw count of occurrences.

7.4.4 Number of Recursive Returns

The Number of Recursive Returns (NRRec) is the number of *return* statements that call the recursive function the student used. This feature allows the teacher to verify if the function developed by the student works recursively. The recursive return needs to include the function's name and change some function parameters so that the function is recursive.

Algorithm 21 shows how our computational model obtained the NFRRec feature. We retrieve the list with the body of the functions to extract this feature. Then we check

Algorithm 20: GET_NIFPAR: NUMBER OF PARAMETERS IN CONDITIONAL

```
   Input: ls_functions //body each function saved in Algorithm 4
   Output: counter ∈ ℕ //number of parameters in conditional
 1 begin
 2     initialize counter → 0;
 3     while do not finish the list_functions do
 4         if is_recursive_function(function) then
 5             while do not finish the function lines do
 6                 if there are conditional in line then
 7                     if there are parameter in line then
 8                         increments the counter;
 9                         break;
10                     end
11                 end
12             end
13         end
14     end
15     return counter;
16 end
```

which functions are recursive and collect the name of those functions. Next, we look for *return* statements that contain the function name on each line of the function body. Finally, we perform the raw count of occurrences.

Algorithm 21: GET_NRREC: NUMBER OF RECURSIVE RETURNS

```
   Input: ls_functions //body each function saved in Algorithm 4
   Output: counter ∈ ℕ //number of recursive returns
 1 begin
 2     initialize counter → 0;
 3     while do not finish the list_functions do
 4         if is_recursive_function(function) then
 5             while do not finish the fuction lines do
 6                 if name function in line and line starts with return then
 7                     increments the counter;
 8                 end
 9             end
10         end
11     end
12     return counter;
13 end
```

7.4.5 Number of Non-Recursive Returns

The Number of Non-Recursive Returns (NRNRec) is the number of return statements that do not call the recursive function the student used. This feature allows the teacher to check if the student has implemented a return to exit the recursive function.

Algorithm 22 shows how our computational model obtained the NRNRec feature. We retrieve the list with the body of the functions to extract this feature. Then we check which functions are recursive and collect the name of those functions. Next, we look for return statements that do not contain the function name on each line of the function body. Finally, we perform the raw count of occurrences.

Algorithm 22: GET_NRNREC: NUMBER OF NON-RECURSIVE RETURNS

Input: code //source code the student developed
Input: ls_functions //body each function saved in Algorithm 4
Output: counter $\in \mathbb{N}$ //number of non-recursive returns

```
1  begin
2      initialize counter → 0;
3      while do not finish the list_functions do
4          if is_recursive_function(function) then
5              while do not finish the fuction lines do
6                  if name function is not in line and line starts with return then
7                      increments the counter;
8                      break;
9                  end
10             end
11         end
12     end
13     return counter;
14 end
```

7.5 Features for T5 (Dynamic Allocations)

The learning topic corresponding to T5 enables the student to use dynamic memory allocation. We defined three new proprietary features to investigate the weaknesses and strengths of students in the T5 learning topic.

The extracted features are desirable and complementary criteria to the learning topic and meet the following learning outcomes: 19) Number of Malloc: the ability to dynamically allocate memory; 20) Number of Sizeof: the ability to calculate the size of any data type; and 21) Number of Free: the ability to free memory that has been allocated.

7.5.1 Number of Malloc

The Number of Malloc (NMalloc) represents the number of *malloc* functions the student used in his source code. This feature facilitates the teacher to identify the student's ability to engage in dynamic memory allocation. Invoking the *malloc* function yields a pointer, thereby enabling the acquisition of memory space.

Algorithm 7 shows how our computational model obtained the NMalloc feature. The algorithm's output counts the number of *malloc* functions found in the source code. A looping goes through the source code. We search each line of source code for the *malloc* function. We identify a *malloc* function when these criteria are met.

7.5.2 Number of Sizeof

The Number of Sizeof (NSizeof) represents the number of *sizeof* operators the student used in his source code. This feature allows the teacher to identify if the student knows how to verify the number of bytes occupied by a certain type of data. For every memory allocation with *malloc*, having one instance of the *sizeof* operator would be consistent. The balance between these two features assures that the memory allocated follows the type of data used.

Algorithm 7 shows how our computational model obtained the NSizeof feature. The algorithm's output counts the number of *sizeof* operators found in the source code. A looping goes through the source code. We search each line of source code for the *sizeof* operator. We identify a *sizeof* operator when these criteria are met.

7.5.3 Number of Free

The Number of Free (NFree) represents the number of *free* functions the student used in his source code. This feature allows the teacher to identify if the student can release previously allocated memory. Every memory allocation with *malloc* would be consistent with having one instance of the free function. The balance between these two features ensures that allocated memory is returned to the system after it is used.

Algorithm 7 shows how our computational model obtained the NFree feature. The algorithm's output counts the number of *free* functions found in the source code. A looping goes through the source code. We search each line of source code for the *free* function. We identify a *free* functions when these criteria are met.

7.6 Feature Extraction Example

Figure 21 encapsulates a exemple of a source code solution corresponding to the task presented in Figure 20. This illustrative example showcases the practical application of the task and provides a tangible reference for the subsequent discussions.

```
1 #include <stdio.h>
2 
3 int sumNumbers(int n1, int n2) {
4     return (n1 + n2);
5 }
6 
7 void main() {
8     int number1, number2;
9     int sum;
10    printf("\nEnter two integer numbers: ");
11    scanf("%d %d", &number1, &number2);
12    sum = sumNumbers(number1, number2);
13    printf("\nThe sum of the values is: ");
14    printf("%d", sum);
15 }
```

Figure 21 – Example of source code solution

Source: The author

For Figure 21, our model extracted the features presented in Table 9. As we chose to disregard the main function counting, the only function created by the student was *sumNumbers()* (line 3). Therefore, NF received the value 1. The number of parameters accounts for *n1* and *n2* (line 3). Therefore, NP received the value 2. The *sumNumbers()* function was called once (line 12). Therefore, NC received the value 1. We observe that the *sumNumbers()* function makes a return (line 4). Therefore, NR received the value 1.

Table 9 – Example of feature extraction for Figure 21

NF	NP	NC	NR
1.0	2.0	1.0	1.0

Figure 22 presents an example in which the student uses a different approach than expected to solve the task. In this case, the student declares two global variables (*number1* and *number2*) instead of creating a function and passing parameters by value to the function.

In Table 10, we present the extracted features for Figure 22. The features were all zeroed because no new function was created. One of the advantages of the proposed

```
1 #include <stdio.h>
2
3 int number1, number2;
4 int sum;
5
6 void main() {
7     printf("\nEnter two integer numbers: ");
8     scanf("%d %d", &number1, &number2);
9     sum = number1 + number2;
10    printf("\nThe sum of the values is: ");
11    printf("%d", sum);
12 }
```

Figure 22 – Source code example where the student does not meet the topic

Source: The author

method is that the instructor can verify at a glance when something unexpected happens, such as having all the features zeroed or even only one particular feature zeroed, which was related to the learning topic.

Table 10 – Example of feature extraction for Figure 22

NF	NP	NC	NR
0.0	0.0	0.0	0.0

7.7 Final Remarks

In this chapter, we present a set of 21 source code features that were created specifically for five learning topics. First, we present how the features relate to each learning topic and their respective learning outcomes. Next, we describe each feature. We cover from how we perform the feature extraction to the contribution that this feature provides to the teacher in the classroom.

We must note that we tested other features until we arrived at this set that suits each learning topic. First, we extracted text-based features that counted the occurrences of keywords from the programming language (SILVA; SILLA, 2020). The problem with these features was that they were simple and had no connection to the learning topics. We could identify that the students were using decision structures and loops, for example. However, that did not tell us anything about the specific programming skills being learned in each learning topic.

Later, we tested features such as whether the source code compiles or not, but we found that this was not a feature that our computational model should address. Our main

purpose was to create features that helped explain the learning topics, and that would be a reason for the build to be left out. Also, the compilation is a "binary test" with only two possibilities: to compile or not to compile. For example, the lack of a semicolon at the end of a statement would be enough for the source code not to work. Our model is broader than that. We want the teacher to have an overview of the skills developed by the students, regardless of whether the solution is correct or the code is functional.

Also, we tested features such as cyclomatic complexity and the average of lines per function. These features are interesting because they add information to the coding in a summarized way. However, we chose to leave them out. In addition to compromising the originality of our features and not representing the specificity of the learning topic, they were not statistically significant. Finally, we tested different configurations, for example, using all features on all learning topics. However, this caused the learning topic at hand to lose the focus of the analysis. We also consider that clustering algorithms ask for fewer variables to avoid the "curse of dimensionality" (HINNEBURG ALEXANDER E KEIM, 1999).

In order to test this set of defined features, the next chapter presents the analysis of our results. We organized the chapter based on our research questions and perfomed experiments to answer each one. We present the experiments' results and highlight the main findings. When possible, we present an interpretation of the clusters from the teacher's point of view. In addition, we performed a descriptive statistical evaluation and non-parametric tests to verify the significance of our results.

Chapter 8

ANALYSIS OF RESULTS

This chapter presents the conduct of experiments and evaluations aimed at providing comprehensive answers to the research questions posited in Chapter 6. In Section 8.1 addressed $\mathbf{RQ}_1$, where we experimented with the different clustering algorithm configurations. Moving forward, Section 8.2 addressed $\mathbf{RQ}_2$, where we experimented with individual student features. Subsequently, Section 8.3 addressed $\mathbf{RQ}_3$, where we checked the statistical validity of our experiment. Then, Section 8.4 addressed $\mathbf{RQ}_4$, where we checked the correlation between features. Finally, Section 8.5 addressed $\mathbf{RQ}_5$, where we experimented with the cluster overview.

8.1 Clustering Settings

$\mathbf{RQ}_1$ How to group students with similar skills using automatic analysis of the source code with machine learning?

To answer $\mathbf{RQ}_1$, we grouped the students by testing the different distance settings for the agglomerative hierarchical clustering algorithm. In Table 11, we present the internal validation index obtained for each configuration of the algorithm using the *Cophenetic Coefficient Correlation*. The results obtained are divided by learning topics (T1 to T5). We tested the combination between the distances (Euclidean and Cityblock) and the Linkage Method (Single, Average, Complete) for each topic.

The best results were found with the Average Linkage method in the Euclidean distance configuration. The only exception was in T2, where there was a tie (0.73) between the distances. In this case, we opted for the Euclidean distance, which was also the best in the other experiments. According to the data, the results from *Kruskal-Wallis H-test* indicated that one or more of the four clusters are significantly different ($p < 0.05$).

Table 11 – Internal validation between clusters

Topic	Distance	Linkage Method		
		Single	Average	Complete
T1	Euclidean	0.62	0.80	0.62
	Cityblock	0.62	0.77	0.65
T2	Euclidean	0.58	0.73	0.73
	Cityblock	0.55	0.72	0.72
T3	Euclidean	0.86	0.88	0.82
	Cityblock	0.87	0.85	0.62
T4	Euclidean	0.74	0.81	0.80
	Cityblock	0.68	0.80	0.78
T5	Euclidean	0.73	0.82	0.64
	Cityblock	0.70	0.70	0.68

8.2 Students' Grouping

$\mathbf{RQ}_2$	How to assess students' individual programming skills?

To answer $\mathbf{RQ}_2$, we use agglomerative hierarchical clustering with the Euclidean Distance and the Average Linkage method because of $\mathbf{RQ}_1$. In addition, we used the previously defined features for each learning topic that was presented in Table 8.

In our default definition, we get four clusters resulting from the grouping, which will be called: GA, GB, GC, and GD. Students who have not submitted any solution for a given list will be identified by NA (not applicable). It is important to note that the names of the clusters do not represent any evaluative concept, nor do they represent an order of development quality. The ascending order in the naming of the groups follows the single pattern of the largest number of features found. However, depending on the teacher's interpretation, this can be a strength or a weakness.

In Table 12, we present the task submission rate. The submitted tasks were separated by cluster and for each learning topic. In the first column, cluster identification is displayed. From the second to the fourth column, the information for T1 is presented. The subsequent columns represent information from the other learning topics (T2 to T5). Each learning topic has the number of students involved for each cluster and the respective amounts of tasks sent. In the last line, we present the number of students who did not submit tasks for each learning topic. In addition, we calculate the percentage of submissions, which is relative to the total expected tasks ($codes * Stud.$). Students have a high submission rate, above 90% on average. GA had the best subimission rates but also had the lowest number of students on average. A highlight point is that in T1 (functions with parameter passage by value), GC grouped 24 students (60% of the total

approximately) and had a submission rate of 94% of the tasks. Something similar happened in T3 (struct), in which GB grouped 20 students (approximately 50% of the total) and had a submission rate of 89% of the tasks. The GD had the lowest submission rates in all learning topics. In T4 (recursion), GD concentrated its largest number of students and had the lowest rate of submissions.

Table 12 – Submission rate by cluster

Cluster	T1			T2			T3			T4			T5		
	# Stud.	# codes	rate	# Stud.	# codes	rate	# Stud.	# codes	rate	# Stud.	# codes	rate	# Stud.	# codes	rate
GA	1	6	100.0%	4	16	100.0%	1	5	100.0%	2	8	100.0%	1	6	100.0%
GB	1	6	100.0%	1	4	100.0%	20	89	89.0%	10	40	100.0%	13	78	100.0%
GC	24	136	94.4%	9	36	100.0%	8	40	100.0%	2	8	100.0%	12	66	91.7%
GD	9	41	75.9%	12	38	79.2%	2	7	70.0%	12	33	68.8%	2	10	83.3%
NA	6	-	-	15	-	-	10	-	-	15	-	-	13	-	-

Students classified as NA did not submit tasks for the learning topic. Therefore, they were not considered in the grouping. There is a chance that these students will master the learning topic of study and refuse to perform the tasks. However, most likely, they have not learned any of the subjects taught in the learning topics. The first learning topic (T1) had only six students who did not submit tasks. The other learning topics had an average of 13 students who did not submit tasks. These values can be explained by why students are more enthusiastic at the beginning of the course, so the submission rates are higher. However, as the course progresses and the difficulties increase, more students stop participating.

In Table 13, we present the cluster output for the five learning topics. The first column represents the 41 students' identification, which is solely used to maintain traceability across learning topics. The number that precedes the student identification corresponds to the identification of the classroom in question (e.g., #101 corresponds to Classroom 1, and Student 01). The four subsequent columns represent the features extracted from T1. The next column (sixth) is the cluster obtained from the clustering algorithm. The following subdivisions represent the extracted features and the cluster obtained for the other learning topics (T2 to T5).

The teacher can verify each student's weaknesses and potential by comparing the Gold-Standard reference values with the results obtained in Table 13, the teacher can verify each student's weaknesses and potentials. We will present some possible insights for the output obtained in our database.

We used boxplot plots to verify the distribution of features extracted in each cluster. The boxplot graphically describes the quantitative distribution of the data to facilitate the comparison between features. The graph is divided into quartiles. The bottom line of the box indicates the first quartile representing 25% of the data distribution. The top line

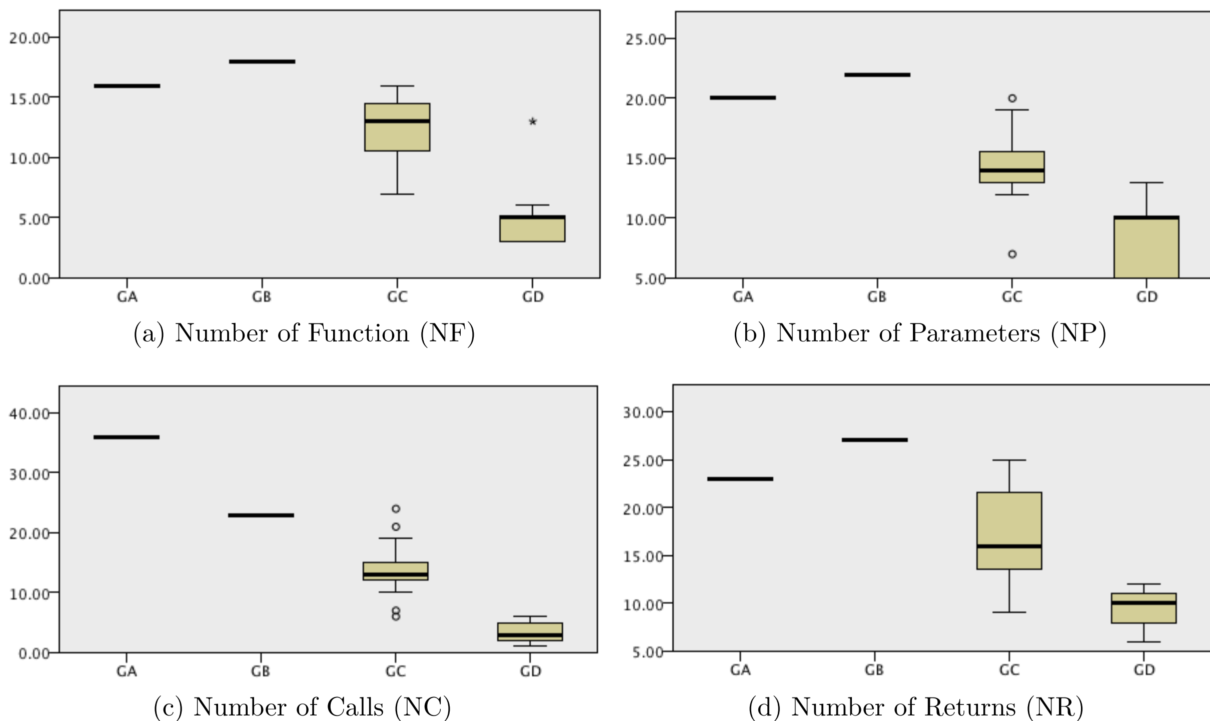
of the box indicates the third quartile, representing 75% of the data distribution. The line that crosses the box represents the median of the data. Also, we have the lower bound and upper bound of the data represented by the ends of both sides of the box. Finally, outliers in the rest of the distribution are called outliers. Thus, with the boxplot, it is possible to visualize the more concentrated data and if there are outliers outside the quartiles. We present a more detailed analysis and some possible insights into each learning topic in the following subsections. The graph's *x-axis* presents the cluster identification (e.g., GA, GB, GC, and GD), and the graph's *y-axis* represents the value obtained for each feature.

Table 13 – Model output after extracting features for the 41 students

Stud.	NF	NP	NC	NR	Cluster	NPt–s	NAdd–s	NPt–ds	NAdd–ds	Cluster	NStr	NStrM	NStrT	NStrI	NStrC	Cluster	NFRec	NCRec	NIFPar	NRRec	NRNRec	Cluster	NSizeof	NMalloc	NFree	Cluster
101	-	-	-	-	NA	-	-	-	-	NA	5.0	26.0	5.0	3.0	17.0	GB	2.0	4.0	2.0	2.0	2.0	GD	2.0	2.0	1.0	GC
102	3.0	5.0	5.0	10.0	GD	-	-	-	-	NA	5.0	26.0	5.0	3.0	16.0	GB	2.0	6.0	2.0	2.0	2.0	GD	2.0	2.0	2.0	GC
103	13.0	17.0	12.0	15.0	GC	2.0	2.0	6.0	6.0	GC	3.0	12.0	3.0	3.0	18.0	GB	4.0	9.0	6.0	0.0	4.0	GC	3.0	3.0	2.0	GB
104	11.0	17.0	10.0	15.0	GC	3.0	3.0	6.0	9.0	GA	5.0	26.0	5.0	3.0	20.0	GB	2.0	5.0	3.0	2.0	2.0	GD	3.0	3.0	2.0	GB
105	14.0	12.0	10.0	18.0	GC	2.0	2.0	6.0	6.0	GC	5.0	26.0	5.0	3.0	20.0	GB	3.0	8.0	5.0	4.0	3.0	GB	2.0	2.0	1.0	GC
106	-	-	-	-	NA	-	-	-	-	NA	5.0	29.0	5.0	3.0	16.0	GB	2.0	4.0	3.0	2.0	2.0	GD	-	-	-	NA
107	16.0	14.0	24.0	21.0	GC	2.0	0.0	5.0	0.0	GA	5.0	26.0	4.0	2.0	14.0	GB	3.0	6.0	5.0	2.0	3.0	GB	3.0	3.0	3.0	GB
108	9.0	7.0	13.0	16.0	GC	1.0	1.0	9.0	9.0	GD	5.0	26.0	0.0	0.0	0.0	GC	4.0	9.0	6.0	0.0	3.0	GC	3.0	3.0	4.0	GB
109	5.0	10.0	2.0	11.0	GD	-	-	-	-	NA	5.0	26.0	0.0	0.0	0.0	GC	3.0	6.0	3.0	0.0	1.0	GD	0.0	1.0	2.0	GD
110	13.0	13.0	1.0	8.0	GD	2.0	0.0	3.0	0.0	GD	-	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	NA
111	12.0	14.0	13.0	22.0	GC	2.0	2.0	7.0	10.0	GA	5.0	26.0	0.0	0.0	0.0	GC	-	-	-	-	-	NA	3.0	3.0	2.0	GB
112	14.0	15.0	13.0	9.0	GC	2.0	2.0	7.0	7.0	GC	5.0	26.0	0.0	0.0	0.0	GC	3.0	7.0	6.0	2.0	3.0	GB	3.0	3.0	2.0	GB
113	-	-	-	-	NA	2.0	1.0	0.0	0.0	GD	2.0	8.0	0.0	0.0	0.0	GD	-	-	-	-	-	NA	-	-	-	NA
114	5.0	10.0	2.0	7.0	GD	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	NA
115	15.0	20.0	15.0	22.0	GC	5.0	5.0	3.0	7.0	GB	4.0	19.0	4.0	3.0	18.0	GB	4.0	12.0	5.0	2.0	2.0	GA	3.0	3.0	3.0	GB
116	8.0	13.0	6.0	13.0	GC	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	-	-	NA	3.0	3.0	1.0	GC
117	-	-	-	-	NA	2.0	1.0	0.0	0.0	GD	-	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	NA
118	15.0	15.0	13.0	10.0	GC	2.0	2.0	6.0	7.0	GC	4.0	20.0	4.0	3.0	16.0	GB	3.0	5.0	6.0	3.0	3.0	GB	1.0	1.0	1.0	GD
119	6.0	8.0	5.0	6.0	GD	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	NA
120	-	-	-	-	NA	2.0	2.0	3.0	3.0	GD	3.0	10.0	3.0	3.0	16.0	GB	-	-	-	-	-	NA	2.0	2.0	1.0	GC
121	11.0	13.0	7.0	15.0	GC	2.0	2.0	4.0	2.0	GD	4.0	19.0	4.0	3.0	18.0	GB	4.0	9.0	4.0	4.0	4.0	GB	3.0	3.0	1.0	GC
122	10.0	13.0	13.0	24.0	GC	2.0	2.0	6.0	6.0	GC	5.0	26.0	5.0	3.0	20.0	GB	3.0	6.0	4.0	0.0	3.0	GD	3.0	3.0	3.0	GB
123	12.0	14.0	21.0	14.0	GC	-	-	-	-	NA	5.0	23.0	1.0	0.0	0.0	GC	4.0	12.0	5.0	4.0	4.0	GA	3.0	3.0	3.0	GB
124	15.0	14.0	15.0	16.0	GC	2.0	2.0	3.0	0.0	GD	5.0	25.0	5.0	3.0	16.0	GB	4.0	8.0	5.0	4.0	3.0	GB	3.0	3.0	2.0	GB
125	16.0	14.0	15.0	25.0	GC	2.0	0.0	6.0	0.0	GD	3.0	13.0	0.0	0.0	0.0	GD	2.0	4.0	2.0	2.0	2.0	GD	3.0	3.0	0.0	GC
126	13.0	13.0	15.0	10.0	GC	2.0	2.0	6.0	3.0	GC	5.0	26.0	4.0	2.0	8.0	GC	4.0	9.0	6.0	5.0	4.0	GB	3.0	3.0	1.0	GC
127	-	-	-	-	NA	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	-	-	NA	2.0	2.0	2.0	GC
128	10.0	14.0	13.0	16.0	GC	2.0	2.0	6.0	6.0	GC	3.0	13.0	2.0	3.0	16.0	GB	2.0	4.0	3.0	0.0	0.0	GD	2.0	2.0	2.0	GC
129	10.0	14.0	12.0	23.0	GC	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	NA
130	3.0	5.0	3.0	12.0	GD	1.0	1.0	3.0	3.0	GD	3.0	13.0	3.0	3.0	16.0	GB	2.0	5.0	4.0	3.0	2.0	GD	-	-	-	NA
131	12.0	13.0	10.0	18.0	GC	2.0	2.0	3.0	1.0	GD	5.0	26.0	5.0	3.0	38.0	GA	2.0	4.0	2.0	2.0	2.0	GD	2.0	2.0	0.0	GC
132	16.0	20.0	36.0	23.0	GA	-	-	-	-	NA	5.0	29.0	6.0	3.0	24.0	GB	-	-	-	-	-	NA	3.0	3.0	1.0	GC
133	14.0	19.0	19.0	12.0	GC	-	-	-	-	NA	5.0	26.0	3.0	0.0	0.0	GC	4.0	9.0	6.0	4.0	4.0	GB	3.0	3.0	3.0	GB
134	5.0	10.0	6.0	11.0	GD	-	-	-	-	NA	5.0	26.0	5.0	3.0	20.0	GB	2.0	4.0	3.0	2.0	2.0	GD	-	-	-	NA
135	14.0	13.0	13.0	12.0	GC	2.0	2.0	6.0	3.0	GC	5.0	26.0	5.0	3.0	20.0	GB	4.0	8.0	6.0	4.0	4.0	GB	3.0	3.0	2.0	GB
136	18.0	22.0	23.0	27.0	GB	4.0	4.0	8.0	10.0	GA	-	-	-	-	-	NA	2.0	4.0	2.0	2.0	2.0	GD	5.0	3.0	3.0	GA
137	7.0	12.0	12.0	21.0	GC	2.0	2.0	6.0	5.0	GC	5.0	26.0	0.0	0.0	0.0	GC	-	-	-	-	-	NA	-	-	-	NA
138	5.0	10.0	4.0	10.0	GD	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	NA
139	3.0	5.0	1.0	9.0	GD	1.0	1.0	3.0	3.0	GD	3.0	13.0	3.0	1.0	13.0	GB	-	-	-	-	-	NA	-	-	-	NA
140	15.0	16.0	16.0	20.0	GC	2.0	2.0	3.0	3.0	GD	5.0	26.0	5.0	3.0	20.0	GB	4.0	8.0	4.0	3.0	3.0	GB	3.0	3.0	2.0	GB
141	14.0	16.0	15.0	25.0	GC	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	-	-	NA	-	-	-	NA

8.2.1 Individual programming skills for the T1

In Figure 23, we graphically present the distribution of T1 features. The *x-axis* represents the four defined clusters, while the *y-axis* indicates the occurrence of the corresponding features. In Figure 23a, we present the NF distribution. Analyzing this boxplot, we noticed that the median of the GA, GB, and GC clusters are close. This means that most of the data from these three clusters are similar to the NF feature. However, students #132 and #136 from the GA and GB clusters, respectively, showed the greatest similarity for all features. In Figure 23c, we can see that the NC feature was decisive in separating student #132 from student #136.



(a) Number of Function (NF)

(b) Number of Parameters (NP)

(c) Number of Calls (NC)

(d) Number of Returns (NR)

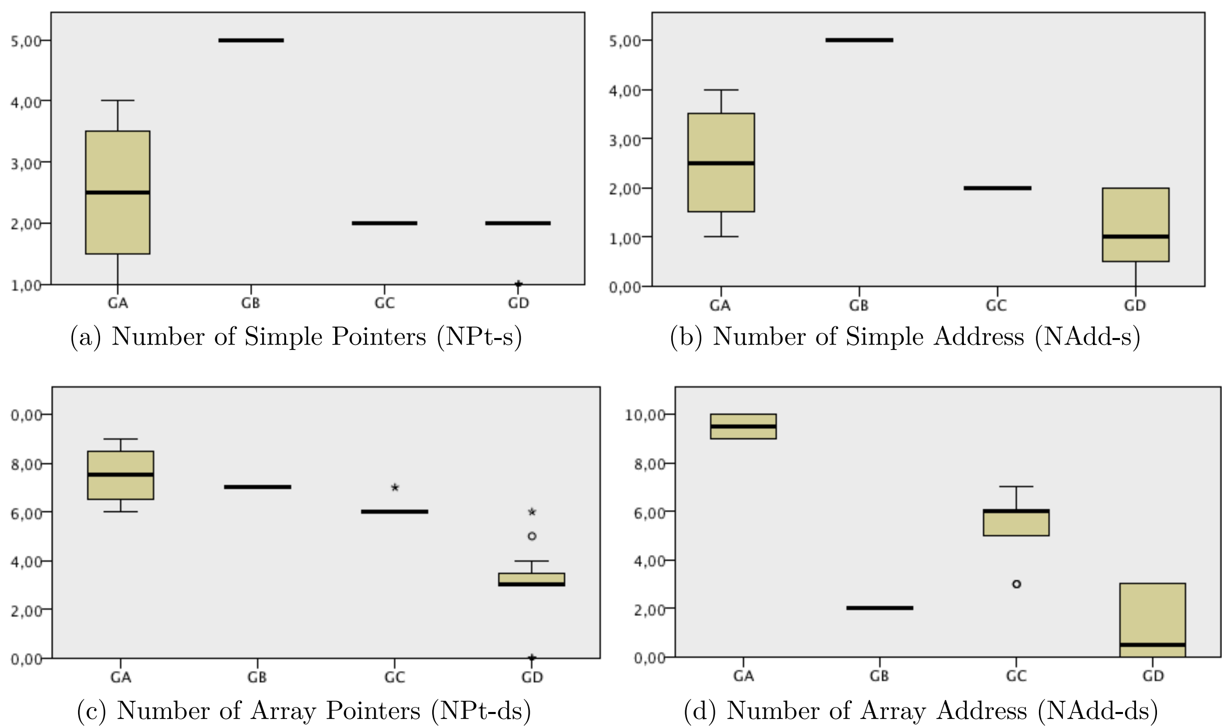
Figure 23 – Relationship between clusters and features for the T1

Overall, we noticed that GC occupies a larger region of the boxplot. This also happens because GC is the cluster with the largest number of students. The GC boxplot demonstrates that several students are about to migrate from the cluster. This happens for students about to be upgraded, such as students #107, #115, and #141 – Just as it happens for students about to be downgraded, such as students #108, #116, and #137. Therefore, the GC cluster also had more outliers for the NP and NC features. It is possible to notice that DG students are struggling to progress, but in all cases, not enough features are developed. In Figure 23a, student #110 excels at creating more functions than his cluster mates. This also happens more leniently with the same student for the NP feature. In Figure 23c, we observe that for NC, the students are more concentrated in a small range of values, indicating that the data are similar. On the other hand, in Figure 23b, we

observe that the students are more dispersed for NP.

8.2.2 Individual programming skills for the T2

In Figure 24, we graphically present the distribution of T2 features. The *x-axis* represents the four defined clusters, while the *y-axis* indicates the occurrence of the corresponding features. Overall, the GA boxplot occupied a more extensive region, while the GC data were concentrated. In the simple type structures (figures 24a, 24b, and 24c), we observed that the GA cluster was quite comprehensive in encompassing other clusters in its minimum and maximum limits. What differentiated GB from GA were the NAdd-ds features, where student #115 presented only three occurrences.



(a) Number of Simple Pointers (NPt-s)

(b) Number of Simple Address (NAdd-s)

(c) Number of Array Pointers (NPt-ds)

(d) Number of Array Address (NAdd-ds)

Figure 24 – Relationship between clusters and features for the T2

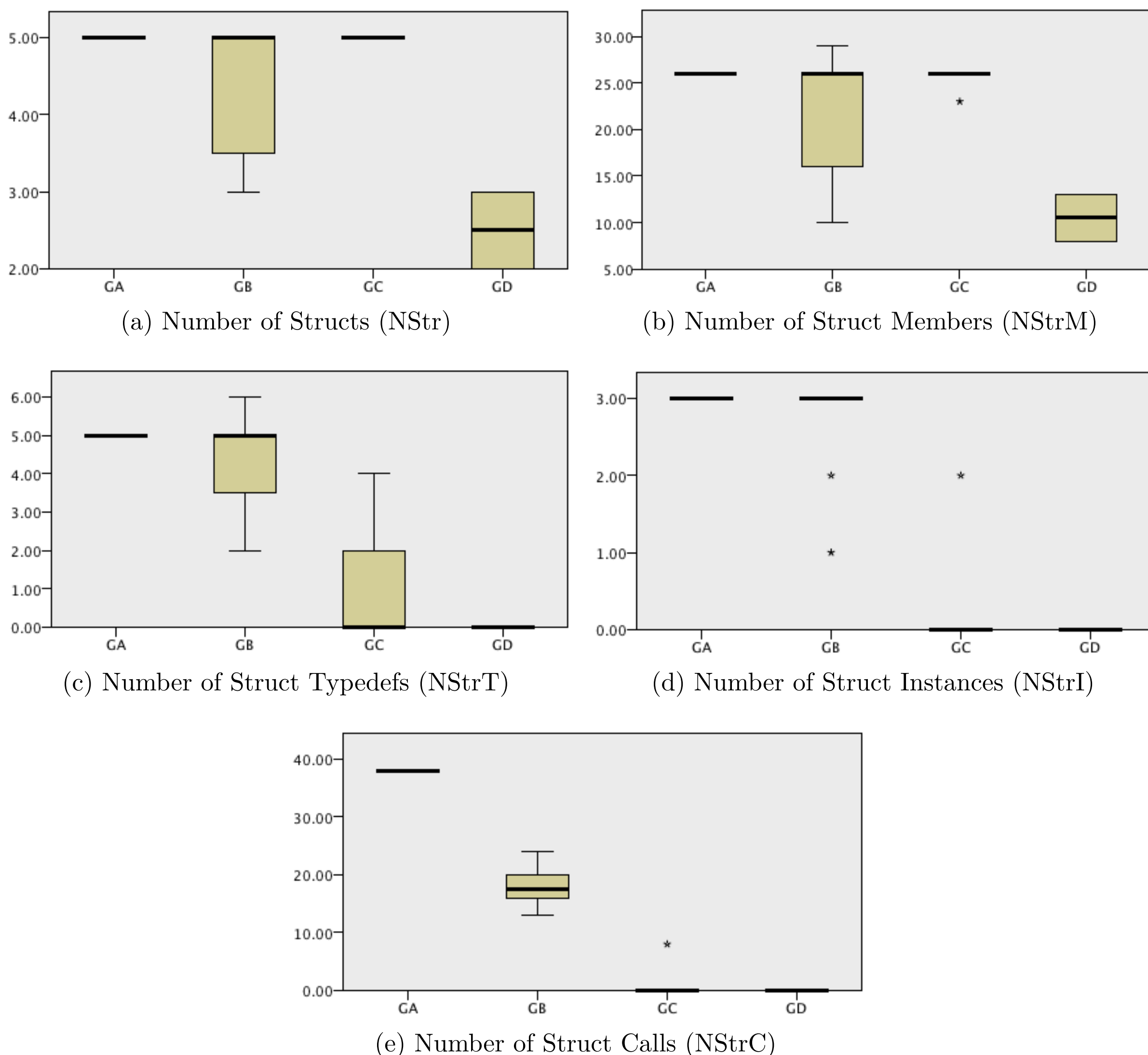
Outliers occurred mainly in array-like structures. In Figure 24c, we observe that student #112 used more array pointers than the average of his peers. The same is true for students #121 and #125 in the GD cluster. On the other hand, two GD students were highlighted for not developing skills related to the NPt-ds feature. In Figure 24d, students #126 and #135 did not develop the NAdd-ds features as much as their peers. A highlight point is that the median of GC is equivalent to the third quartile, which gathers 75% of the students.

From the T2, it is possible to verify how the migration of students between the clusters occurred along the learning topics. An example is student #107, who was in T1.GC, close to migrating from the cluster, moved to T2.GA. On the other hand, student

#108, who was also in T1.GC, but with features below his peers, moved to T2.GD. It is also interesting to note that 67% of students in GD.T1 moved to NA.T2.

8.2.3 Individual programming skills for the T3

In Figure 25, we graphically present the distribution of T3 features. The *x-axis* represents the four defined clusters, while the *y-axis* indicates the occurrence of the corresponding features. Overall, the GB boxplot occupied a larger region, except for GB.NStrI. GB also grouped most students, about 50% of enrolled students. The interesting point is that the median obtained by GB was close to student #131 of the GA cluster, except for the NStrC feature. Half of the GB students are aligned with the GA student. The GC also had its data similarly distributed in the NStr and NStrM features. However, GC.NStrT (Figure 25c), GC.NStrI (Figure 25d), GC.NStrC (Figure 25e) had their data more similar to the GD cluster.



(a) Number of Structs (NStr)

(b) Number of Struct Members (NStrM)

(c) Number of Struct Typedefs (NStrT)

(d) Number of Struct Instances (NStrI)

(e) Number of Struct Calls (NStrC)

Figure 25 – Relationship between clusters and features for the T3

In Figure 25d, NStrI was the feature that most concentrated the data. Outliers occurred for features NStrM, NStrI, and NStrC. In Figure 25d, students #107 and #139 trend downgrade in GB.NStrI. On the other hand, student #126 is trending upward in GC.NStrC.

In the migration of students from T2 to T3, it is possible to observe a large number of students from different groups assuming GB, approximately 49% of the sample. A positive point is that 67% of GD.T2 students moved up in level when they moved to T3. Only two GD.T2 students remained in the GD in T3. Another two GD.T2 students moved on to NA.T3. Furthermore, only student #136, who had submitted tasks in T2, did not turn in any task in T3. These insights allow observing greater stability among students in T3.

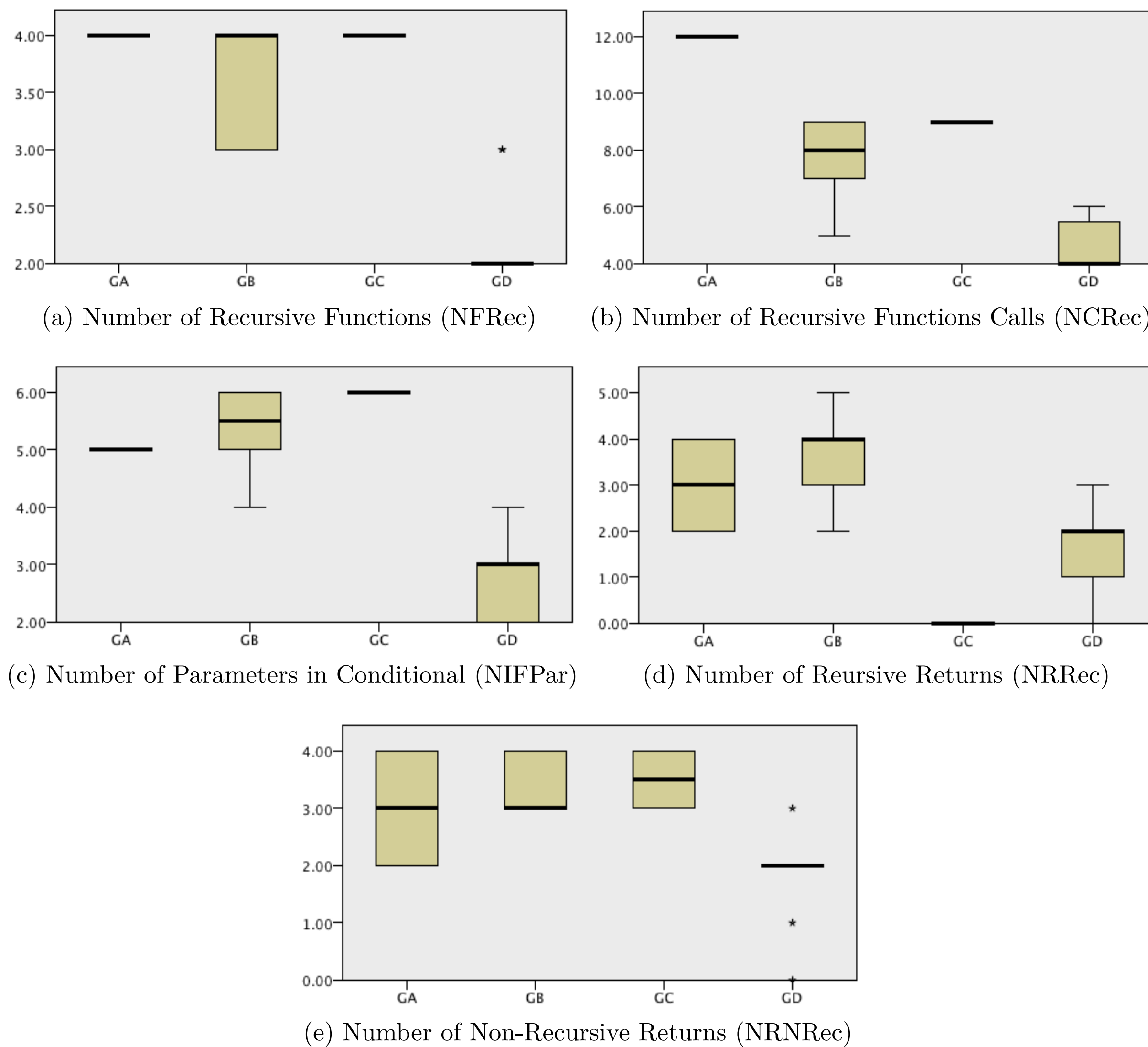
8.2.4 Individual programming skills for the T4

In Figure 26, we graphically present the distribution of T4 features. The *x-axis* represents the four defined clusters, while the *y-axis* indicates the occurrence of the corresponding features. What separated GA from the other clusters was the NCRec feature. GB has a lower limit coinciding with the GD cluster in NIFPar, NCRec, and NRRec. The outliers occurred in GD.NFRec with an upward trend for students #109 and #122, who had three occurrences of recursive functions. Other students who presented discrepant data were #109, #122, and #128 in GD.NRNRec.

In the migration of students from T3 to T4, the results were somewhat negative. There was a 12% increase in students who did not submitted any tasks. In addition, it increased 24% of students from GD.T3 to GD.T4. Although T4 was not exciting in terms of leveling up students, a positive point was student #123 who came from GC.T1 to GA.T4. The same happened with student #115 who went from GC.T1 to GA.T4, but in this case the journey was even better because he went through GB.T2 and GB.T3.

8.2.5 Individual programming skills for the T5

In Figure 27, we graphically present the distribution of T5 features. The *x-axis* represents the four defined clusters, while the *y-axis* indicates the occurrence of the corresponding features. Overall, GA and GB presented similar features. What differentiated GA from GB was mainly NMalloc, shown in Figure 27a. Although GB grouped 12 students, its data was concentrated on the NMalloc and NSizeof features. On the other hand, student #108 outperformed all other students in GB.NFree. Regardless, GC was closer to GB and what differentiated them was mainly the NFree features, shown in Figure 27c. Interestingly, the GD was inferior to the other clusters, except in NFree. In this way, the lower limit of



(a) Number of Recursive Functions (NFRec)

(b) Number of Recursive Functions Calls (NCRec)

(c) Number of Parameters in Conditional (NIFPar)

(d) Number of Reursive Returns (NRRec)

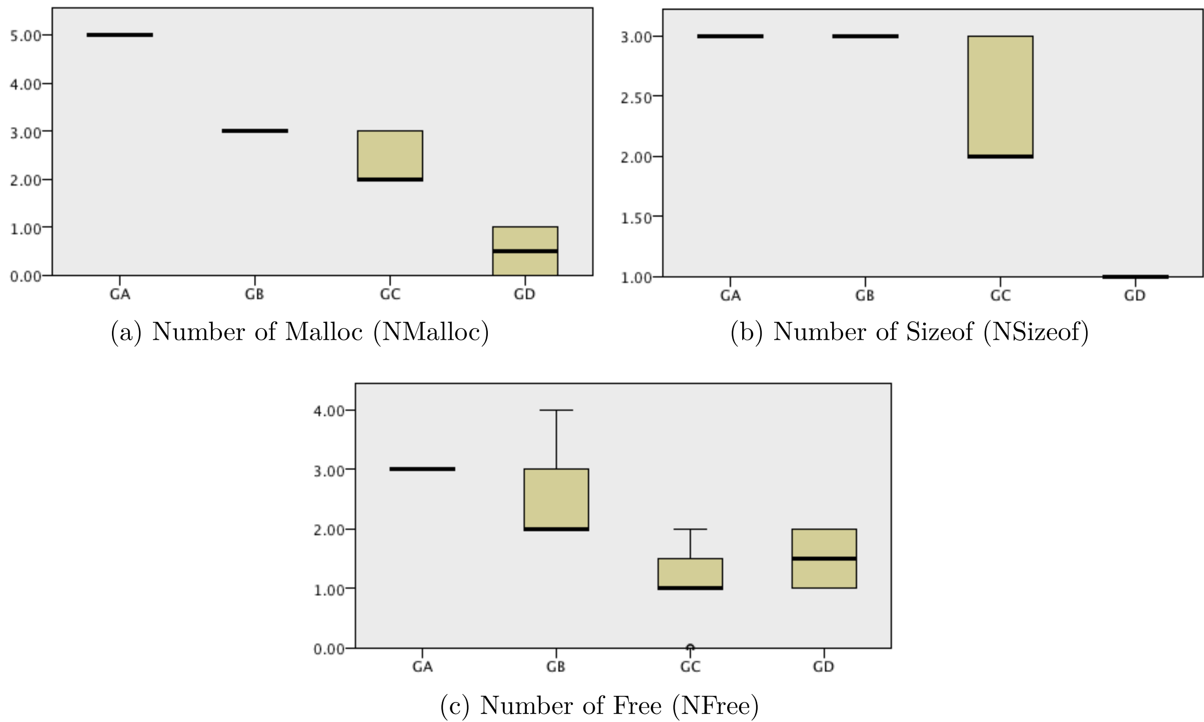
(e) Number of Non-Recursive Returns (NRNRec)

Figure 26 – Relationship between clusters and features for the T4

GD.NFree even surpassed the median of GC.NFree.

One event that occurred several times in T2 was that the median coincided with the first quartile and the lower limit of the clusters. This happened in GC.NMalloc in Figure 27a, GC.NSizeof in Figure 27b, and in Figure 27c for GB.NFree and GC.NFree. Outliers occurred twice, only in GC.NFree. Students #131 and #125 never used the *free* function to free memory.

In the migration of students from T4 to T5, it is possible to observe only two students in GD. In addition, there was less dropout among students. Only two students who submitted in T4 did not submit any tasks in T5 (#106 and #134). Another interesting insight is the students' complete journey through the learning topics. For example, #106 showed a possible recovery in GB.T3, when it could have been redeemed course. This happened in the case of student #101, who started to submit tasks from T3 onwards.



(a) Number of Malloc (NMalloc)

(b) Number of Sizeof (NSizeof)

(c) Number of Free (NFree)

Figure 27 – Relationship between clusters and features for the T5

8.2.6 Clusters quality validation

In order to validate the quality of the generated clusters, we manually checked the source code of a sample of students for learning topic 1. Our sample procedure consisted of selecting examples of students closest to each cluster midpoint. Therefore, from GA cluster we analyzed the source codes from student (#132). From GB Cluster, we only had one student (#136). From GC Cluster the students closest to the cluster midpoint were #122 and #131. And, from GD Cluster the selected students were #102 and #138. The result of our manual analysis is presented in Table 14. We used the following criteria to identify the level of the quality specification: fully meets (●), partially meets (◒), and does not meet (○). While it may appear that certain students across different clusters exhibit similarities in certain source code tasks, it is crucial to bear in mind that the clusters were established based on the entirety of the learning topic's tasks (six tasks in the T1 case).

Although the order in which the cluster names are defined does not imply superiority, we noticed that the student in the GA cluster were more careful in their source code solutions. One example would be in task 01, whose statement is presented in Figure 20. Apart from receiving integer values, student #132 utilizes the resources of the *limits.h* library to verify whether or not the received values fall within the minimum and maximum numerical values that an integer data type can assume. Another example is GB student #136, who verifies if the input data is an integer, when it is not, a new value is requested from the user. In addition, students in the GA and GB cluster enclose the program within

Table 14 – Clusters quality validation for a sample of students in T1

	Cluster	GA	GB	GC	GC	GD	GD
	Stud.	#132	#136	#122	#131	#102	#138
task 01	statement	●	●	●	●	●	◒
	compile	●	●	●	●	●	○
	logic	●	●	●	●	●	◒
	validations	●	●	○	○	○	○
	comments	●	●	○	○	○	◒
	clear buffer	●	●	○	○	○	○
	run again	●	●	○	○	○	○
	variables names	●	●	●	●	●	◒
	external library	●	●	○	○	○	○
	own library	○	○	○	○	○	○
task 02	statement	●	●	◒	◒	●	●
	compile	●	●	○	○	●	●
	logic	●	●	◒	◒	◒	◒
	validations	●	●	○	○	○	○
	comments	●	●	◒	◒	◒	◒
	clear buffer	●	●	○	○	○	○
	run again	●	●	○	○	○	○
	variables names	●	●	◒	◒	◒	◒
	external library	●	●	○	○	○	○
	own library	●	●	○	○	○	○
task 03	statement	●	●	◒	●	◒	◒
	compile	●	●	●	●	○	○
	logic	●	●	◒	◒	◒	◒
	validations	●	●	○	○	○	○
	comments	●	●	◒	◒	◒	◒
	clear buffer	●	●	○	○	○	○
	run again	●	●	○	○	○	○
	variables names	●	●	◒	◒	◒	◒
	external library	●	●	○	○	○	○
	own library	●	●	◒	●	○	●
task 04	statement	●	●	●	●	◒	◒
	compile	●	●	●	●	○	●
	logic	●	●	●	◒	◒	◒
	validations	●	●	●	◒	○	◒
	comments	●	●	●	◒	◒	◒
	clear buffer	●	●	○	○	○	○
	run again	●	●	○	○	○	○
	variables names	●	●	●	◒	◒	◒
	external library	●	●	●	◒	○	○
	own library	○	○	○	○	○	○
task 05	statement	●	●	●	●	○	○
	compile	●	●	●	●	○	○
	logic	●	●	●	◒	○	○
	validations	●	●	●	◒	○	○
	comments	●	●	●	◒	○	○
	clear buffer	●	●	●	○	○	○
	run again	●	●	●	●	○	○
	variables names	●	●	●	◒	○	○
	external library	●	●	●	●	○	○
	own library	○	○	○	○	○	○
task 06	statement	●	●	●	◒	○	○
	compile	●	●	●	○	○	○
	logic	●	●	●	◒	○	○
	validations	●	●	●	○	○	○
	comments	●	●	●	◒	○	○
	clear buffer	●	●	●	○	○	○
	run again	●	●	●	○	○	○
	variables names	●	●	●	◒	○	○
	external library	●	●	●	●	○	○
	own library	○	○	○	○	○	○

● The student fully meets the quality specification.
◒ The student partially meets the quality specification.
○ The student does not meet the quality specification.

a loop so that the user can execute it more than once, they also perform buffer cleaning before receiving a value, and they also execute operating system commands using the *system()* function from the *stdlib.h* library. These quality specifications are some examples that differentiate students #132 and #136 from other student clusters.

The students in GC and GD clusters demonstrate a more limited understanding of the subject compared to the students in the other clusters. The quality requirements were either partially or not met in most cases. However, what seems to distinguish students in the GC cluster from students in the GD cluster is the completion of tasks 05 and 06, which were not submitted by the students in the GD cluster.

8.3 Statistical Test Results

RQ$_3$	Were the extracted features significantly different in each learning topic?

In Table 15, we present the statistical test result performed to compare the distri-

butions of the different features and verify if the features are significantly different. We run a test for each learning topic and present the results separately. In the first column, we identify the features. In the second column, we present the result of the *Chi-Square* statistic with the Degree of Freedom ($df = 3$). Furthermore, in the last column, we present the value of p. In tables 15a, 15d and 15e, we observed that all features showed significant differences ($p < 0.05$). In Table 15b, we observe that only the NPt-s feature is not statistically different ($p = 0.067$). Finally, in Table 15c, we observe that only the NStrM feature is not statistically different ($p = 0.079$).

Table 15 – Summary of the *Kruskal-Wallis H-test* statistical test

(a) Statistical test for T1

Feature	$H_{(3)}$	p
NF	18.84	0.000
NP	19.43	0.000
NC	22.37	0.000
NR	17.54	0.001

(b) Statistical test for T2

Feature	$H_{(3)}$	p
NPt-s	7.16	0.067
NAdd-s	10.90	0.012
NPt-ds	19.91	0.000
NAdd-ds	19.92	0.000

(c) Statistical test for T3

Feature	$H_{(3)}$	p
NStr	10.53	0.015
NStrM	6.78	0.079
NStrT	17.69	0.001
NStrI	25.69	0.000
NStrC	21.92	0.000

(d) Statistical test for T4

Feature	$H_{(3)}$	p
NFRec	19.92	0.000
NCRec	19.24	0.000
NIFPar	19.27	0.000
NRRec	15.19	0.002
NRNRec	16.99	0.001

(e) Statistical test for T5

Feature	$H_{(3)}$	p
NMalloc	17.63	0.001
NSizeof	16.17	0.001
NFree	16.99	0.001

The statistical analysis is relevant for understanding learning topics, enabling the identification of significant differences among features, detecting patterns in source code, and providing support for decision-making in the development of new features. This analysis helps identify features that can be improved and highlights aspects of source code that may form the basis for the course. Consequently, it enables a more informed and data-driven approach to enhancing the computational model.

8.4 Correlation between the Features

$\mathbf{RQ}_4$	What is the correlation between the features extracted in each learning topic?

In Table 16, we present the result of the correlation between the features. With these results, we verify if there is a relationship between the features that were defined for each learning topic. We run a test for each learning topic and present the results separately. In the first column, we identify the features. In subsequent columns, we repeat each feature to cross them in pairs. The table rows show the resulting value of the obtained Spearmans' rho coefficient. In Table 16a, we observe a high correlation between all the features of T1. We can highlight the relationship between the number of functions (NF) with parameters (NP) and the calls (NC) that had the highest coefficients for T1 ($\rho = 0.80$). In Table 16b, we observe a low correlation between simple type pointers and arrays. We can highlight the relationship between simple pointers and array addresses ($\rho = 0.08$). In Table 16c, we observe features with the lowest correlations. The number of structs (NStr) and the number of struct instances (NStrI) had a negative correlation ($\rho = -0.06$). That is, the higher the frequency of NStr, the lower the frequency of NStrl. Finally, in Table 16d, we highlight recursive functions (NFRec) and recursive function calls (NCRec) that are highly correlated ($\rho = 0.92$). The same happens in Table 16e, with the commands of *Malloc* and *Sizeoff* ($\rho = 0.97$).

The statistical correlation is relevant in understanding learning topics, enabling the identification of non-linear relationships among source code features, reducing the influence of outliers, and considering scale independence. In source code features, relationships between different features can be complex and may not follow a linear pattern, especially due to programming errors or unique code conditions. The use of correlation allows for the identification and quantification of such relationships, even when they do not exhibit a direct association.

8.5 Overview of Students' Grouping

$\mathbf{RQ}_5$	How to get an overview of programming skills from student groups?

To answer $\mathbf{RQ}_5$, we present the midpoint calculation for each feature within a cluster of students. We present these results in the next subsections, divided into five different tables (tables 17 to 21) representing each learning topic. The five tables follow a common organization. In the first column, cluster identification is presented. In the second column, the number of students grouped is presented. The subsequent columns

Table 16 – Summary of the *Spearman's rho* correlation between the features

(a) Correlation for T1

	NF	NP	NC
NP	0.80		
NC	0.80	0.74	
NR	0.50	0.52	0.62

(b) Correlation for T2

	NPt-s	NAdd-s	NPt-ds
NAdd-s	0.68		
NPt-ds	0.30	0.47	
NAdd-ds	0.08	0.27	0.72

(c) Correlation for T3

	NStr	NStrM	NStrT	NStrI
NStrM	0.91			
NStrT	0.40	0.46		
NStrI	-0.06	0.02	0.80	
NStrC	0.12	0.21	0.85	0.88

(d) Correlation for T4

	NFRec	NCRec	NIFPar	NRRec
NCRec	0.92			
NIFPar	0.79	0.73		
NRRec	0.40	0.38	0.34	
NRNRec	0.80	0.74	0.77	0.56

(e) Correlation for T5

	NSizeof	NMalloc
NMalloc	0.97	
NFree	0.41	0.38

represent the features of the learning topic. For each feature, the calculation of the midpoint and its respective standard deviation are presented. In addition, the expected reference value (Gold-Standard) of each feature is presented in the last row of the tables. The Gold-Standard was defined according to the teacher's solution logic.

In general, GD students presented results much lower than expected and needed special monitoring from the teacher. Students developed at least the first exercises on the list, either completely or almost all of them incompletely. These students are close to dropping out of the course or are likely to retake the course. As learning topics advance, the possibility of recovering GD students becomes increasingly remote. Promoting tasks in pairs (for example, pair programming (HANNAY et al., 2009)) with members of different groups could help GD students. Another option is to have stronger students teach weaker ones, as in the 300 methodology (FRANGELLI, 2015). Individually, the teacher can direct easy-level role tasks to reinforce topic understanding and restore student confidence.

8.5.1 Overview for the T1

We present the resulting midpoint for each T1 cluster in Table 17. A total of 35 students participated in the T1 tasks submissions, corresponding to 85% of the enrolled students. The other six enrolled students did not participate in the T1 tasks submission.

Table 17 – Midpoint obtained from the grouping in T1

Cluster*	# Stud.	NF	NP	NC	NR
GA	1	16.0 ±0.0%	20.0 ±0.0%	36.0 ±0.0%	23.0 ±0.0%
GB	1	18.0 ±0.0%	22.0 ±0.0%	23.0 ±0.0%	27.0 ±0.0%
GC	24	12.5 ±2.5%	14.3 ±2.5%	13.5 ±3.9%	17.2 ±4.8%
GD	9	5.3 ±2.9%	8.4 ±2.7%	3.2 ±1.7%	9.3 ±1.9%
Gold-Standard		16.0	16.0	16.0	28.0

* There were no applied (NA) in the grouping of six students who did not submit any source code solution for this learning topic.

Only one student was labeled with GA. He used the same number of functions defined in the reference value. The number of returns was lower than expected. In terms of functions with passing parameters by value, a higher frequency is expected in NR. On the other hand, the number of parameters was higher than expected. The results of these features are within an acceptable range of variation. However, the main point of attention was using the created functions represented by NC. The student made twice as many calls to functions as anticipated. The functions created are being called over and over again, probably in an exaggerated way. In this case, the NC feature does not represent a threat to the structure of the developed solutions. In general, additional instructions are being implemented, and less objective logic is being developed. This group of students needs the teacher's help to make their source codes more uncomplicated and objective.

Only one student was labeled with GB. He used features above the reference value for all features except NR. The first feature that draws the teacher's attention is the number of functions, as the student may fragment the source code beyond what is necessary. On the other hand, the NP and NC features point to coherence in coding: the more functions developed, the higher the exchange of parameters, and the higher the number of function calls. Reviewing the scope of the variable is the first activity indicated for students to be able to manipulate functions. The understanding that the variables sent by parameter need to work again in the *main* function with new values is fundamental for the evolution of GB students. Second, NR is a feature that generates an alert for the teacher, but it should not be a concern. As the student used more functions and parameters than the reference, the returns of the functions were expected to the same extent. Overall, the GB generated enough features to solve the source code tasks and presented coherence between the features. The GB represents a possibly self-taught group of students capable of quickly

learning different logic to solve the same exercise. Correcting the list of classroom exercises would be enough for the GA to understand the possible optimizations in their source code.

The largest cluster was GC, which grouped 24 students. Overall, the cluster midpoint was close to the reference values. However, attention should be on the resulting standard deviation that was higher than 2.4% for all features. A higher heterogeneity of solutions is also expected when the cluster is large. This cluster can reach extremes for the collected features. The resulting standard deviation confirmed the diversity of solutions. Probably, some students produced tasks closer to the reference value, while others developed simple tasks to the point of almost switching clusters. By looking at these details, the teacher could increase the number of clusters ($k = 5$) in the algorithm configuration to produce more specific clusters. In general, GC students are still following the topics studied. Some tasks were not submitted, as shown in the submission rate shown in Table 12. One possibility is that students who need more help from the teacher are starting to give up on more difficult tasks. The teacher can rescue these specific students in the individual output presented in Table 13. Generally, reviewing the subjects studied and encouraging students to do more exercises to ensure learning is important.

8.5.2 Overview for the T2

We present the resulting midpoint for each T2 cluster in Table 18. A total of 26 students participated in T2 tasks submissions, corresponding to 63% of enrolled students. The other 15 enrolled students did not participate in the T2 tasks submission.

Table 18 – Midpoint obtained from the grouping in T2

Cluster*	# Stud.	NPt-s	NAdd-s	NPt-ds	NAdd-ds
GA	4	2.5 ±1.1%	2.5 ±1.1%	7.5 ±1.1%	9.5 ±0.5%
GB	1	5.0 ±0.0%	5.0 ±0.0%	3.0 ±0.0%	7.0 ±0.0%
GC	9	2.0 ±0.0%	2.0 ±0.0%	6.1 ±0.3%	5.4 ±1.4%
GD	12	1.8 ±0.4%	1.2 ±0.8%	3.0 ±1.6%	1.3 ±1.4%
Gold-Standard		3.0	3.0	6.0	6.0

* There were no applied (NA) in the grouping of 15 students who did not submit any source code solution for this learning topic.

Four students were labeled with GA. They used the number of simple type pointers and addresses closest to the reference value. However, GA also had the highest number of array pointers. Also, GA is probably repeating the call of some function and therefore needs to pass more array-type addresses. The students in the GA cluster do not represent a concern for the teacher.

On the other hand, GB had the highest number of simple pointers and their references. As for arrays, we observe that the number of pointers is considerably less

than the number of addresses. In this cluster, the call of repeated functions that receive array pointers was even more aggravating. Also, what could be happening is that GB compensated for array-type pointers by using simple-type pointers. The student could receive reinforcement regarding the use of array pointers.

GC grouped a significant portion of students (22%). They presented features slightly below the reference value. The only exception was for the number of array-type pointers. These students are probably developing the source codes with the least amount of features expected to solve the task. However, the results show that the GB follows the techniques and topics taught. It is important that the teacher encourages this cluster in the resolution of tasks and keeps attention to its results.

The most important point in this learning topic is the features related to pointers and their respective addresses. Note that all groups are declaring and using pointers. Overall, considering the resulting standard deviation, we observed that the T2 cluster had fewer variations within the clusters. Another highlight is that most groups present related features proportionally. Regardless of the resulting numbers, this consistency between the features indicates a good understanding of the source code being developed.

8.5.3 Overview for the T3

We present the resulting midpoint for each T3 cluster in Table 19. A total of 32 students participated in T3 tasks submissions, corresponding to 78% of enrolled students. The other 10 enrolled students did not participate in the T3 tasks submission.

Table 19 – Midpoint obtained from the grouping in T3

Cluster*	# Stud.	NStr	NStrM	NStrT	NStrI	NStrC
GA	1	5.0 ±0.0%	26.0 ±0.0%	5.0 ±0.0%	3.0 ±0.0%	38.0 ±0.0%
GB	20	4.4 ±0.9%	21.8 ±6.2%	4.3 ±1.0%	2.9 ±0.5%	17.7 ±2.5%
GC	8	5.0 ±0.0%	25.8 ±0.7%	1.0 ±1.5%	0.3 ±0.7%	1.0 ±2.6%
GD	2	2.5 ±0.5%	10.5 ±2.5%	0.0 ±0.0%	0.0 ±0.0%	0.0 ±0.0%
Gold-Standard		5.0	26.0	5.0	3.0	17.0

* There were no applied (NA) in the grouping of 10 students who did not submit any source code solution for this learning topic.

Only one student was labeled with GA. It presented results equal to the reference value. The exception was for the NStrC feature, which generated twice as many occurrences. What may have happened is that the student is being detailed in developing his source code. Once the student has scored well on the other features, a positive theory is valid. Therefore, it must print data in abundance to provide a better experience for those using the program.

The most expressive cluster was GB (48%). GB obtained results slightly below the reference values. Although the NStrC feature outperformed the benchmark and the NStrM feature was inferior, this should not be of concern to the teacher. We must consider that many students are involved and that the standard deviation for these two features was high (6.2% and 2.5%, respectively).

Although GC obtained regular results for NStr and NStrM, the cluster obtained a result much lower than expected for the last three features (NTstrT, NStrI, and NstrC). This cluster is likely bringing together students who have not fully understood the learning topic. They need tutoring to review learning and solve more exercises.

8.5.4 Overview for the T4

We present the resulting midpoint for each T4 cluster in Table 20. A total of 26 students participated in T4 tasks submissions, corresponding to 63% of enrolled students. The other 15 enrolled students did not participate in the T4 tasks submission.

Table 20 – Midpoint obtained from the grouping in T4

Cluster*	# Stud.	NFRec	NCRec	NIFPar	NRRec	NRNRec
GA	2	4.0 ±0.0%	12.0 ±0.0%	5.0 ±0.0%	3.0 ±1.0%	3.0 ±1.0%
GB	10	3.6 ±0.5%	7.7 ±1.3%	5.3 ±0.8%	3.5 ±0.9%	3.4 ±0.5%
GC	2	4.0 ±0.0%	9.0 ±0.0%	6.0 ±0.0%	0.0 ±0.0%	3.5 ±0.5%
GD	12	2.2 ±0.4%	4.7 ±0.8%	2.8 ±0.7%	1.6 ±1.0%	1.8 ±0.7%
Gold-Standard		4.0	10.0	8.0	6.0	4.0

* There were no applied (NA) in the grouping of 15 students who did not submit any source code solution for this learning topic.

Two students were labeled with GA. They used the same amount of recursive functions as the reference value. Calls to recursive functions were also close to expectations, indicating that recursive functions were reused throughout the source code. The other features were lower than expected. GB obtained similar results with the first cluster. The decisive feature in separating these two clusters was NCRec. GB made 30% fewer recursive calls than GA. However, GB also grouped a larger share of students (24%).

A point of attention was the number of recursive returns. NRRec indicates whether the student is implementing recursive functions that work and is an important feature for the learning topic. GA and GB obtained half of the expected recursive returns. However, this may not be a concern because students may not be anticipating more than one return situation in the same recursive function.

Two students were labeled with GC. The number of recursive returns that resulted in zero caught our attention in this cluster. That is, there is a high possibility that the functions developed are not recursive in the execution of the application. The students

in this cluster need the teacher's help solving this issue. The other features are balanced against the other clusters and the reference values.

Overall, considering the resulting standard deviation, we observed that the clustering of T4 had little variation within the clusters. The NIPar and NRNRec features generated similar behavior for GA, GB, and GC. All recursive functions must have at least one stop condition. In this sense, the NIFPar and NRNRec features complement each other. The GA, GB, and GC clusters met the minimum stopping conditions that agree with the number of recursive functions but obtained results below the reference value. It would be important for the teacher to review the subjects and demonstrate other source code examples to improve students' skills in controlling recursive functions.

8.5.5 Overview for the T5

We present the resulting midpoint for each T5 cluster in Table 21. A total of 28 students participated in T5 tasks submissions, corresponding to 68% of enrolled students. The other 13 enrolled students did not participate in the T5 tasks submission.

Table 21 – Midpoint obtained from the grouping in T5

Cluster*	# Stud.	NSizeof	NMalloc	NFree
GA	1	$5.0 \pm 0.0\%$	$3.0 \pm 0.0\%$	$3.0 \pm 0.0\%$
GB	13	$3.0 \pm 0.0\%$	$3.0 \pm 0.0\%$	$2.5 \pm 0.6\%$
GC	12	$2.4 \pm 0.5\%$	$2.4 \pm 0.5\%$	$1.1 \pm 0.6\%$
GD	2	$0.5 \pm 0.5\%$	$1.0 \pm 0.0\%$	$1.5 \pm 0.5\%$
Gold-Standard		3.0	3.0	3.0

* There were no applied (NA) in the grouping of 13 students who did not submit any source code solution for this learning topic.

Only one student was labeled with GA. This student exceeded the expected values in the reference for using the *sizeof* operators. However, its other features were equivalent to the reference values. The GA result does not negatively impact the structures used since the *Sizeof* operator is used as a precaution not to allocate more (or less) memory than necessary.

GB and GC formed large clusters at T5. Still, it is important to note that although it was the largest group, the standard deviation was low. GB met the expected values, except for the average of free functions, which was lower than expected. Given the number of students in the GC cluster, the Nfree feature does not represent a concern for the teacher. Regardless, the GC obtained lower results in relation to the reference values. The use of *sizeof* and *malloc* is balanced, representing a positive point. However, the students

are struggling to free up the memory that has been allocated. For this reason, GC students need help understanding the importance of the structures studied in T5.

8.6 Final Remarks

In this chapter, we present the analysis of the results obtained from our experiments. For this, we used a database with more than 650 real-world source code tasks that were developed in C language. Then, we extracted the 21 features defined in Chapter 7, grouped the students, and presented the results for the five learning topics. This thesis was financed in part by the Coordenação de Apereiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

In $\mathbf{RQ}_1$, we present the experiment to define the configuration of the machine learning technique. We verified which would be an adequate configuration to perform the grouping of students based on the extracted features. We tested the two categories of algorithms widely used in the literature: divisive and hierarchical. The clusters formed by the divisive algorithm were similar to those formed by the hierarchical algorithm. Other clustering algorithms could have been tested, finding the best clustering algorithm is beyond the scope of this research. Our focus was on testing the feature set we created. We used the agglomerative hierarchical clustering algorithm to cluster the students. We considered the freedom to define the number of clusters after clustering and the visual dendrogram system that can help debug cluster organization. Thus, we ran an experiment for each learning topic with different distance settings between students and between clusters. We apply an internal cluster validation to verify the best configuration. In addition, we performed a test to verify if the cluster separation was statistically significant. We found the Euclidean distance metric and the Average Linkage method as the setting that best represented the data.

In $\mathbf{RQ}_2$, $\mathbf{RQ}_3$, and $\mathbf{RQ}_4$, we present the experiment to verify the students' individual programming skills. These abilities were identified through the previously defined features and extracted from the source code tasks. First, we present the submission rate of the tasks to understand the context in which the results were obtained. A point of attention was the percentage of tasks submitted on each learning topic. The student may have chosen to do only some tasks instead of all, which can happen for several reasons. The student may not have had time to complete the tasks and decided to prioritize the quality of only a few tasks. Another possibility is that the student has only developed the first tasks, which are usually less complex and require less understanding of the learning topic. Our model allows us to measure development quality through the extracted features. We allow the teacher to input reference values that can be used as a parameter to verify the minimum features expected in a learning topic. However, our experiment considered all

tasks within the learning topic. This way of dealing with tasks prevents us from trying to find students who developed only a few source code tasks but the students who would be able to develop all of them. This granularity could be modified according to the teacher's interest and thus allow the tasks to be analyzed individually. In addition, we found that our features have statistical significance and present the correlation between the features. An interesting example was the simple-type structures that had a low correlation with the array-like structures in T2. This finding confirms the learning outcomes that were expected. On the other hand, we observed that NFree had a low correlation with the other features in T5. These results help us understand the strengths and weaknesses observed the source code solutions of each student.

In $\mathbf{RQ}_5$, we present the experiment performed to get an overview of programming skills by cluster. These results help the teacher find strengths and weaknesses within the cluster quickly. A weakness of the midpoint calculation is that clusters can have students at extreme limits, which can considerably differentiate students. Although we present the number of students grouped and the standard deviation obtained in the feature, the overview can cause a false impression. Therefore, we recommend that confirmation be made of the student's individual skills. Furthermore, we present the Gold-Standard concept that was used to define the expected values for each feature. Thus, it was possible to identify in a macro way which clusters were closer or farther from the expected learning outcomes and to direct specific help. The important point is that the Gold-Standard can also be used in the individual analysis of students.

In regards to the related work presented in Chapter 5, we present research related to students' grade that uses computational approaches to correct students' tasks automatically and presents a grade according to established criteria. Grading tools strictly check whether the source code is correct or incorrect. However, we are interested in finding out the strengths and weaknesses of students' programming skills. Our computational model provides a view of what is happening within the source code. In this way, the teacher has insights into what the student might be doing wrong without looking inside the source code. According to Ullah et al. (2018), one of the main limitations of the static approach is comparing students' code with a teacher-provided model solution since this comparison is usually made character by character. In our approach, this limitation is overcome since the teacher might optionally provide only one source code solution for each programming task, which is then used as a Gold-Standard. However, regardless of whether or not the Gold-Standard is present, the proposed method provides a visual and meaningful visualization of the student's programming skills. This visualization can be seen as either per student or group of students with similar programming skills.

We present also research related to students' performance. In particular, monitoring and evaluating students with machine learning techniques. Among the studies found, there is

special attention to predicting student performance to use this information for interventions in teaching-learning programming. These predictions are usually driven by high failure and drop out rates in CS1 courses. The use of supervised machine learning techniques has shown promising results. However, the most significant harm in teaching-learning is generating previous data to train the computational model. Often this data is generated in a classroom to be used in a new classroom, generating a new problem: the student profile. Regardless, our computational model uses the current classroom source codes. Also, the teacher can take advantage of the results from the first source codes developed by the students.

Overall, our model allows the teacher to analyze the situation of their students without opening any source code. The results allowed us to separate the students into groups. This separation does not necessarily define which student is better but defines which students have different skills. From this, teaching methodologies can be applied according to the teacher's strategies. We know that other features could be created, but we try to define features representing each learning topic's expected learning outcomes. Our features make it possible to identify whether students are acquiring enough knowledge to proceed in the course. The next chapter concludes this doctoral thesis. We rescued our goals and revisited our research questions. Also, we describe our contributions, discuss research limitations, and present future directions.

Chapter 9

CONCLUSIONS

Current professions have become dependent on computer software, and it is anticipated that future professions will be exercised by intelligent software. With this advance and the popularization of technology, the demand for CS1 courses increased significantly. However, the first contact with the programming languages is not easy; leading groups of students to fail and drop out of the courses.

Regardless, a great responsibility is attributed to teachers, who need to monitor student performance to promote appropriate teaching-learning for each group of students. However, evaluating source code solutions is time-consuming for the teacher and becomes even more challenging in large classes. We presented the efforts that have been made to develop computational approaches that support teaching-learning programming in the context of the problems presented in this research. The students' grade approaches can help professors grade large volumes of source code tasks. Regardless, students' performance approaches can help the teacher automatically identify students who are facing obstacles in the course. Finally, students' grouping tools can bring together large numbers of students and their source code tasks to provide a summary overview for the teacher.

Thus, the help of automated mechanisms that use machine learning techniques is greatly valued. In this context, our main research objective was to develop a computational model that uses machine learning techniques to automatically analyze source codes and group students with similar programming skills. Our thesis hypothesis was: "It is possible to group students with related programming skills from the automatic analysis of their source code tasks". To achieve our objective and answer our thesis hypothesis, we conducted a literature review to understand the teaching-learning area of programming (SILVA et al., 2019). Our main discoveries were the classifications of learning content, tools, and teaching-learning programming strategies. This gave us the insight to group students based on their source code. Thus, we delved into the fundamentals of unsupervised machine learning techniques focusing on clustering algorithms.

As an initial result, we developed a computational approach for clustering students from their source codes using an agglomerative hierarchical clustering algorithm (SILVA; SILLA, 2020). The extracted features were based on text using the keywords of the programming language. Our experiment was performed with source code tasks based on programming exercise lists for a complete academic period. We identified three clusters of students, one of whom needed more help in the learning process. Although this first experiment has some limitations, it was an important step toward building this thesis for several reasons. First, we performed all the steps involved in an unsupervised machine learning setting, from the database creation to the computational model evaluation. It is important to note that this thesis's author had no previous experience with machine learning research. Second, the first experiment's main limitation is that it considers all the source code solutions developed by students during the whole academic period at once. As discussed in Chapter 8, this prevents any possibility of teacher intervention to support the students who need help during the current academic period. Finally, the extracted features were generic and did not uniquely characterize each learning topic.

Based on the extensive investigations carried out and the limitations discovered, our research efforts led us to identify a significant research gap that constitutes the main focus of this doctoral thesis, highlighted in Chapter 7. The fundamental idea of this study was to provide valuable support to professors in tracking students' strengths and strengths in a programming course during the academic period. To achieve this overarching goal, the creation and adaptation of new source code features was considered essential. These designed features have been strategically tailored to capture and track students' academic progress as they traverse diverse and challenging learning topics. Consequently, this initiative not only facilitates more informed and personalized instructional approaches, but also enables personalized interventions to better address each student's individual learning needs.

In this research, we defined a total of 21 source code features for five learning topics. We present a computational model to cluster students from their source code tasks using an agglomerative hierarchical clustering algorithm. We carry out the practical application of our model in a database of more than 650 real-world source codes. Our results provided insights into the potentials and weaknesses obtained in developing learning topics. Our model allows the teacher to understand students without opening the source code. Furthermore, the teacher does not depend on a specific programming environment and can create his own source code tasks.

9.1 Research Limitations and Future Work Directions

After developing this doctoral thesis, several research gaps have come to light, presenting intriguing opportunities for future investigation. In the ensuing discourse, we aim to delve into the delineated limitations unveiled by our study, elucidating how these findings can serve as a foundation for further advancements in the field. By dissecting these constraints, we intend to offer insightful directions that can be leveraged to enhance and mitigate the identified issues, thereby fostering a more comprehensive understanding of the subject matter.

- **The diversity of programming languages:** our computational model is strongly linked to the C Language. This is because identifying and extracting features from source code tasks requires prior knowledge of language-specific behaviors. However, this limitation is only present in the extractor module, which works directly on the diversity of language structures to extract the features. A modification in the coding of the extractor module makes it possible to replicate our computational model to other input programming languages. We choose the C Language because it is widely used to teach programming, and it serves as a basis for learning other languages, such as Java or Python.
- **The number of clusters:** the nature of our database requires a process of discovering the ranking of students within the cluster. We define four clusters as a default to support the idea that we can have four groups of students: very skillful, skillful, medium, and unskilled. Furthermore, the rationale behind using four clusters in our experiments is that in our collective experience (considering the three authors of the research) in teaching programming for over 38 years in different universities, we normally came across students that struggle in the module, students that struggle a bit, students that do well in the module and a few number of students that excel at the module, hence the four clusters. Other authors in the literature defined a similar number of clusters that also took as a premise the qualification of students, like Anand et al. (2018) and Aottiwerch and Kokaew (2018). However, to guarantee the teacher's autonomy, the definition of the number of clusters in our model is parameterized. The teacher can modify this value according to the reality of the classroom and the teaching strategy. In addition, we chose not to group enrolled students who did not send a source code solution on a given topic. Regardless, the standard way our model was configured allows the teacher to carry out interactive interventions among students. Although our focus is not on methodologies, our model allows for cases in which the teacher wants to propose activities using active learning methodologies. These cases require the composition of groups of students with different abilities. Therefore, ranking techniques or student pairing benefit from

the results generated by our model. A future direction that would make our model more self-contained would be to add automatic adjustment of the number of clusters at runtime.

- **The temporality of the experiments:** our experiments were carried out with a class that had already been completed. In order to repeatedly test the created features and the developed implementations, it was necessary to have a complete database. Carrying out classroom experiments in real time would make our research unfeasible. To mitigate threats to validity and not impact our results, we use our database to simulate an ongoing academic period. Our experiments were run progressively across learning topics (T1 to T5). However, obtaining the results as the weeks of classes progressed would be interesting. One research possibility is to replicate our experiments in a production environment so that the teacher can follow each student's progress and interfere in the teaching-learning process to optimize the student's gains.

- **The learning topics:** our experiments cover specific learning topics that are not the most fundamental programming language commands. However, we understand that this stage involves complete and complex knowledge that requires a closer look from the teacher. An avenue for further investigation lies in the development of more elementary features aimed at catering to novices in the realm of programming education. As evidenced in our prior research (SILVA; SILLA, 2020), we introduced features suitable for more beginning topics encompassing areas like input and output, conditionals, diversion, and looping structures. Expanding the feature set to encompass foundational concepts can enhance the model's applicability across varying levels of programming proficiency.

- **The defined features:** our features were defined from the learning topics embedded within the course curriculum. However, the potential exists for the introduction of new features that align with criteria deemed significant by other teachers. To develop new features, it's crucial to ensure congruence with the established model's overarching goals and focus on capturing salient aspects of student skills. The alignment of these new features with the existing ones should be guided by the pedagogical context and the desired insights into students' programming skills. Rigorous testing and validation of these new features against the established benchmarks would be necessary to ascertain their effectiveness in enhancing the comprehensiveness of the model while preserving its foundational principles. In our endeavor, we made the strategic decision to reduce the dimensionality of our feature vector by retaining solely those features intricately associated with the learning topics and their corresponding learning outcomes. Furthermore, the exploration of different features combinations remains feasible. An interesting avenue for future investigations is to define weights

according to the importance of the learning topic. The delineation of feature relevance could facilitate the inclusion of additional features per topic while preserving the pedagogical essence of the learning process.

REFERENCES

ACM. *ACM Curricula Recommendations*. 2013. Available in: https://www.acm.org/education/curricula-recommendations. Cited on page 23.

AGGARWAL, Charu C.; REDDY, Chandan K. (Ed.). *Data Clustering: Algorithms and Applications*. Philadelphia, PA, USA: CRC Press, 2014. Cited 4 times on pages 47, 49, 50, and 51.

AHADI, Alireza; LISTER, Raymond; HAAPALA, Heikki; VIHAVAINEN, Arto. Exploring machine learning methods to automatically identify students in need of assistance. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. New York, NY, USA: ACM, 2015. (ICER '15), p. 121–130. Cited 2 times on pages 78 and 84.

AHADI, Alireza; LISTER, Raymond; LAL, Shahil; LEINONEN, Juho; HELLAS, Arto. Performance and consistency in learning to program. In: *Proceedings of the Nineteenth Australasian Computing Education Conference*. New York, NY, USA: ACM, 2017. (ACE '17), p. 11–16. Cited 4 times on pages 26, 79, 84, and 85.

ALFARO, Luca de; SHAVLOVSKY, Michael. Crowdgrader: A tool for crowdsourcing the evaluation of homework assignments. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 415–420. Cited 2 times on pages 25 and 40.

ALFIANI, Ardita Permata; WULANDARI, Febriana Ayu et al. Mapping student's performance based on data mining approach (a case study). *Agriculture and Agricultural Science Procedia*, Elsevier, v. 3, p. 173–177, 2015. Cited on page 79.

ALRASHEED, H.; MELTON, A. Understanding and measuring nesting. In: *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference*. Vasteras, Sweden: IEEE, 2014. p. 273–278. Cited on page 67.

ANAND, V. K.; RAHIMAN, S. K. Abdul; GEORGE, E. Ben; HUDA, A. S. Recursive clustering technique for students' performance evaluation in programming courses. In: *Proceedings of the 2018 Majan International Conference (MIC)*. Muscat, Oman: IEEE, 2018. p. 1–5. Cited 3 times on pages 79, 84, and 151.

ANDERSON, J. L. Using software tools and metrics to produce better quality test software. In: *Proceedings of the AUTOTESTCON 2004*. San Antonio, TX, USA: IEEE, 2004. p. 293–297. Cited 5 times on pages 15, 62, 64, 65, and 69.

ANDERSON, Michael R; ANTENUCCI, Dolan; BITTORF, Victor; BURGESS, Matthew; CAFARELLA, Michael J; KUMAR, Arun; NIU, Feng; PARK, Yongjoo; RÉ, Christopher; ZHANG, Ce. Brainwash: A data system for feature engineering. In: *Proceedings of the*

Conference on Innovative Data Systems Research. Asilomar, California, USA: CIDR Conference, 2013. Cited on page 57.

ANDERSON, Ruth E.; ERNST, Michael D.; nEZ, Robert Ordó PHAM, Paul; TRIBELHORN, Ben. A data programming cs1 course. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 150–155. Cited on page 39.

AOTTIWERCH, Naladtaporn; KOKAEW, Urachart. The analysis of matching learners in pair programming using k-means. In: *Proceedings of the 2018 5th International Conference on Industrial Engineering and Applications (ICIEA)*. Singapore, Singapore: IEEE, 2018. p. 362–366. Cited 4 times on pages 26, 79, 85, and 151.

BARR, Valerie; STEPHENSON, Chris. Bringing computational thinking to k-12: What is involved and what is the role of the computer science education community? *ACM Inroads*, Association for Computing Machinery, New York, NY, USA, v. 2, n. 1, p. 48–54, February 2011. Cited on page 23.

BASILI, V.R.; BRIAND, L.C.; MELO, W.L. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, v. 22, n. 10, p. 751–761, 1996. Cited on page 76.

BASSI, P. R. de; WANDERLEY, G. M. P.; BANALI, P. H.; PARAISO, E. C. Measuring developers' contribution in source code using quality metrics. In: *Proceedings of the 2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. Nanjing, China: IEEE, 2018. p. 39–44. Cited on page 85.

BATTESTILLI, Lina; AWASTHI, Apeksha; CAO, Yingjun. Two-stage programming projects: Individual work followed by peer collaboration. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 479–484. Cited on page 40.

BECK, Kent; ANDRES, Cynthia. *Extreme Programming Explained: Embrace Change*. 2nd. ed. Boston: Addison-Wesley Professional, 2004. Cited on page 38.

BENGFORT, Benjamin; BILBRO, Rebecca; OJEDA, Tony. *Applied Text Analysis with Python: Enabling Language-Aware Data Products with Machine Learning*. 1st. ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2018. Cited 2 times on pages 52 and 58.

BENGIO, Yoshua; COURVILLE, Aaron; VINCENT, Pascal. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, IEEE, v. 35, n. 8, p. 1798–1828, 2013. Cited on page 57.

BENNEDSEN, Jens; CASPERSEN, Michael E. Failure rates in introductory programming. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 39, n. 2, p. 32–36, June 2007. Cited on page 24.

BENOTTI, Luciana; ALOI, Federico; BULGARELLI, Franco; GOMEZ, Marcos J. The effect of a web-based coding tool with automatic feedback on students' performance and perceptions. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 2–7. Cited 3 times on pages 41, 78, and 84.

BHATIA, S.; MALHOTRA, J. A survey on impact of lines of code on software complexity. In: *Proceedings of the 2014 International Conference on Advances in Engineering Technology Research (ICAETR - 2014)*. Singapore, Singapore: IEEE, 2014. p. 1–4. Cited on page 62.

BONNER, Raymond E. On some clustering techniques. *IBM journal of research and development*, IBM, v. 8, n. 1, p. 22–32, 1964. Cited on page 54.

BOUGUETTAYA, Athman; YU, Qi; LIU, Xumin; ZHOU, Xiangmin; SONG, Andy. Efficient agglomerative hierarchical clustering. *Expert Systems with Applications*, Elsevier, v. 42, n. 5, p. 2785–2797, 2015. Cited on page 47.

BRUFF, D. *Teaching with Classroom Response Systems: Creating Active Learning Environments*. Hoboken, NJ, USA: Wiley, 2009. (Jossey-Bass higher and adult education series). Cited on page 37.

BRYANT, Sallyann; ROMERO, Pablo; BOULAY, Benedict du. Pair programming and the mysterious role of the navigator. *Int. J. Hum.-Comput. Stud.*, Academic Press, Inc., Duluth, MN, USA, v. 66, n. 7, p. 519–529, July 2008. Cited on page 38.

BURLINSON, David; MEHEDINT, Mihai; GRAFER, Chris; SUBRAMANIAN, Kalpathi; PAYTON, Jamie; GOOLKASIAN, Paula; YOUNGBLOOD, Michael; KOSARA, Robert. Bridges: A system to enable creation of engaging data structures assignments with real-world data and visualizations. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 18–23. Cited on page 42.

BUSE, R. P. L.; WEIMER, W. R. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, v. 36, n. 4, p. 546–558, 2010. Cited 2 times on pages 74 and 85.

BUTLER, Zack; BEZAKOVA, Ivona; FLUET, Kimberly. Pencil puzzles for introductory computer science: An experience- and gender-neutral context. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 93–98. Cited on page 40.

CACEFFO, Ricardo; GAMA, Guilherme; AZEVEDO, Rodolfo. Exploring active learning approaches to computer science classes. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 922–927. Cited 2 times on pages 37 and 38.

CAMP, Tracy; ADRION, William Richards; BIZOT, Betsy; DAVIDSON, Susan; HALL, Mary; HAMBRUSCH, Susanne; WALKER, Ellen; ZWEBEN, Stuart. Generation cs: The growth of computer science. *ACM Inroads*, ACM, New York, NY, USA, v. 8, n. 2, p. 44–50, May 2017. Cited on page 23.

CARDONA, Gabriel; MIR, Arnau; ROSSELLÓ, Francesc; ROTGER, Lucía; SÁNCHEZ, David. Cophenetic metrics for phylogenetic trees, after sokal and rohlf. *BMC Bioinformatics*, v. 14, n. 1, p. 3, January 2013. Cited 2 times on pages 55 and 98.

CARTER, Janet; WHITE, Su; FRASER, Karen; KURKOVSKY, Stanislav; MCCREESH, Colette; WIECK, Malcolm. Iticse 2010 working group report motivating our top students. In: *Proceedings of the 2010 ITiCSE Working Group Reports*. New York, NY, USA: ACM, 2010. (ITiCSE-WGR '10), p. 29–47. Cited on page 24.

CASTRO-WUNSCH, Karo; AHADI, Alireza; PETERSEN, Andrew. Evaluating neural networks as a method for identifying students in need of assistance. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 111–116. Cited 3 times on pages 40, 78, and 84.

CELEPKOLU, Mehmet; BOYER, Kristy Elizabeth. Thematic analysis of students' reflections on pair programming in cs1. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 771–776. Cited on page 38.

CHENG, Wei; WANG, Wei; BATISTA, Sandra. Grid-based clustering. In: *Data clustering*. Minneapolis, MN, USA: Chapman and Hall/CRC, 2018. p. 128–148. Cited on page 96.

COHEN, Jacob. *Statistical Power Analysis for the Behavioral Sciences*. New York, NY, USA: Lawrence Erlbaum Associates, 1988. Cited on page 100.

CORDER, Gregory W.; FOREMAN, Dale I. *Nonparametric Statistics: A Step-by-Step Approach*. Malden, MA, USA: John Wiley & Sons, 2014. Cited on page 94.

CORMEN, Thomas. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro, RJ, Brazil: ELSEVIER, 2012. 944 p. Cited on page 23.

COUNSELL, Steve; GATRELL, Matt; HIERONS, R; MURGIA, Alessandro; TONELLI, Roberto; MARCHESI, Michele; CONCAS, Giulio. Conditional-based refactorings and fault-proneness: an empirical study. In: *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. Luxembourg, Luxembourg: IEEE, 2013. p. 80–85. Cited 2 times on pages 72 and 73.

CROSS, James; HENDRIX, Dean; BAROWSKI, Larry; UMPHRESS, David. Dynamic program visualizations: An experience report. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 609–614. Cited on page 42.

CROUCH, Catherine H.; MAZUR, Eric. Peer instruction: Ten years of experience and results. *American Journal of Physics*, v. 69, n. 9, p. 970–977, 2001. Cited on page 37.

DEB, Debzani; FUAD, Mohammad Muztaba; KANAN, Mallek. Creating engaging exercises with mobile response system (mrs). In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 147–152. Cited on page 42.

DICHEVA, Darina; HODGE, Austin. Active learning through game play in a data structures course. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 834–839. Cited on page 39.

DOMINGUEZ, Adrián; NAVARRETE, Joseba Saenz de; MARCOS, Luis de; FERNáNDEZ-SANZ, Luis; PAGéS, Carmen; MARTíNEZ-HERRáIZ, José-Javier. Gamifying learning experiences: Practical implications and outcomes. *Computers & Education*, v. 63, p. 380 – 392, 2013. Cited on page 39.

DONG, Guozhu; LIU, Huan. *Feature engineering for machine learning and data analytics*. Boca Raton, FL, USA: CRC Press, 2018. Cited 2 times on pages 58 and 59.

DOUCE, Christopher; LIVINGSTONE, David; ORWELL, James. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, ACM New York, NY, USA, v. 5, n. 3, p. 4–es, 2005. Cited on page 75.

EDGCOMB, Alex; VAHID, Frank; LYSECKY, Roman; LYSECKY, Susan. Getting students to earnestly do reading, studying, and homework in an introductory programming class. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 171–176. Cited on page 42.

EDWARDS, Stephen H; PEREZ-QUINONES, Manuel A. Web-cat: automatically grading programming assignments. In: *Proceedings of the 13th annual conference on Innovation and technology in computer science education*. New York, NY, USA: Association for Computing Machinery, 2008. p. 328–328. Cited 2 times on pages 76 and 84.

EDWARDS, Stephen H.; TILDEN, Daniel S.; ALLEVATO, Anthony. Pythy: Improving the introductory python programming experience. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 641–646. Cited on page 41.

ERNST, Michael D.; COCKRELL, Jake; GRISWOLD, William G.; NOTKIN, David. Dynamically discovering likely program invariants to support program evolution. In: *Proceedings of the 21st International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 1999. (ICSE '99), p. 213–224. ISBN 1581130740. Cited on page 76.

ESTER, Martin. Density-based clustering. *Data Clustering*, Chapman and Hall/CRC, Minneapolis, MN, USA, p. 111–127, 2018. Cited on page 96.

ESTEY, Anthony; KEUNING, Hieke; COADY, Yvonne. Automatically classifying students in need of support by detecting changes in programming behaviour. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 189–194. Cited on page 40.

EVERITT, B.S.; LANDAU, S.; LEESE, M.; STAHL, D. *Cluster Analysis*. London, UK: Wiley, 2011. Cited 3 times on pages 49, 50, and 51.

FAGEN, Adam P; CROUCH, Catherine H; MAZUR, Eric. Peer instruction: Results from a range of classrooms. *The physics teacher*, American Association of Physics Teachers, v. 40, n. 4, p. 206–209, 2002. Cited on page 25.

FARGHALLY, Mohammed F.; KOH, Kyu Han; SHAHIN, Hossameldin; SHAFFER, Clifford A. Evaluating the effectiveness of algorithm analysis visualizations. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 201–206. Cited on page 42.

FäRNQVIST, Tommy; HEINTZ, Fredrik; LAMBRIX, Patrick; MANNILA, Linda; WANG, Chunyan. Supporting active learning by introducing an interactive teaching tool in a data structures and algorithms course. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 663–668. Cited on page 42.

FELDER, Richard M.; BRENT, Rebecca. Active learning: An introduction. *ASQ Higher Education Brief*, v. 2, n. 4, 2009. Cited on page 25.

FONSECA, Nuno Gil; MACEDO, Luís; MENDES, António José. Supporting differentiated instruction in programming courses through permanent progress monitoring. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 209–214. Cited 3 times on pages 25, 77, and 84.

FOWLER, Martin. *Refactoring: Improving the Design of Existing Code*. 2nd. ed. Boston, MA, USA: Addison-Wesley, 2018. Cited 5 times on pages 70, 71, 72, 73, and 104.

FRANGELLI, Ricardo R. Three hundred: Active and collaborative learning as an alternative to the problem of test anxiety. *Revista Eletrônica Gestão & Saúde*, v. 6, n. Supl 2, p. 860–72, 2015. Cited 2 times on pages 25 and 139.

FRANK-BOLTON, Pablo; SIMHA, Rahul. Docendo discimus: Students learn by teaching peers through video. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 473–478. Cited on page 40.

GALAN, Daniel; HERADIO, Ruben; VARGAS, Hector; ABAD, Ismael; CERRADA, Jose A. Automated assessment of computer programming practices: The 8-years uned experience. *IEEE Access*, v. 7, p. 130113–130119, 2019. Cited on page 77.

GALLIANO, G. *O metodo cientifico: teoria e pratica*. São Paulo: Mosaico, 1979. Cited on page 87.

GARRETA, Ral; MONCECCHI, Guillermo. *Learning Scikit-learn: Machine Learning in Python*. Birmingham, UK: Packt Publishing, 2013. Cited on page 85.

GAUDENCIO, Matheus; DANTAS, Ayla; GUERRERO, Dalton D.S. Can computers compare student code solutions as well as teachers? In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 21–26. Cited 2 times on pages 43 and 70.

GILL, A. C. *Como elaborar projetos de pesquisa*. 4. ed. São Paulo: Atlas, 2002. Cited on page 88.

GOYVAERTS, Jan; LEVITHAN, Steven. *Regular expressions cookbook*. Sebastopol, CA, USA: O'reilly, 2012. Cited on page 60.

GREEN, PD; LANE, Peter CR; RAINER, Austen; SCHOLZ, S. An introduction to slice-based cohesion and coupling metrics. University of Hertfordshire, 2009. Cited 2 times on pages 69 and 70.

HALKIDI, Maria; BATISTAKIS, Yannis; VAZIRGIANNIS, Michalis. On clustering validation techniques. *Journal of intelligent information systems*, Springer, v. 17, n. 2-3, p. 107–145, 2001. Cited 3 times on pages 54, 55, and 96.

HALSTEAD, Maurice H. *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977. Cited on page 68.

HAN, J.; PEI, J.; KAMBER, M. *Data Mining, Southeast Asia Edition*. Waltham, MA, USA: Elsevier Science, 2006. (The Morgan Kaufmann Series in Data Management Systems). Cited on page 88.

HANKS, Brian; FITZGERALD, Sue; MCCAULEY, Renée; MURPHY, Laurie; ZANDER, Carol. Pair programming in education: A literature review. *Computer Science Education*, Taylor & Francis, v. 21, n. 2, p. 135–173, 2011. Cited on page 25.

HANNAY, Jo E; DYBÅ, Tore; ARISHOLM, Erik; SJØBERG, Dag IK. The effectiveness of pair programming: A meta-analysis. *Information and software technology*, Elsevier, v. 51, n. 7, p. 1110–1122, 2009. Cited on page 139.

HAO, Qiang; BARNES, Bradley; WRIGHT, Ewan; KIM, Eunjung. Effects of active learning environments and instructional methods in computer science education. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 934–939. Cited on page 37.

HARRINGTON, Peter. *Machine Learning in Action*. USA: Manning Publications Co., 2012. Cited 3 times on pages 45, 47, and 96.

HARSLEY, Rachel; FOSSATI, Davide; EUGENIO, Barbara Di; GREEN, Nick. Interactions of individual and pair programmers with an intelligent tutoring system for computer science. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 285–290. Cited on page 38.

HEATON, Jeff. An empirical analysis of feature engineering for predictive modeling. In: *Proceedings of the 2016 IEEE SoutheastCon Conference*. Norfolk, VA, USA: IEEE, 2016. p. 1–6. Cited on page 57.

HEINONEN, Kenny; HIRVIKOSKI, Kasper; LUUKKAINEN, Matti; VIHAVAINEN, Arto. Using codebrowser to seek differences between novice programmers. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2014. (SIGCSE '14), p. 229–234. Cited on page 43.

HENDERSON-SELLERS, B. *Object-oriented Metrics: Measures of Complexity*. USA: Prentice Hall PTR, 1996. (Object-Oriented Series). Cited 3 times on pages 69, 103, and 104.

HERTZ, Matthew. What do "cs1" and "cs2" mean?: Investigating differences in the early courses. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2010. (SIGCSE '10), p. 199–203. Cited on page 23.

HINNEBURG ALEXANDER E KEIM, Daniel A. Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In: *Proceedings of the 25th International Conference on Very Large Data Bases*. Konstanz, Germany: KOPS, 1999. Cited on page 123.

HMELO-SILVER, Cindy E. Problem-based learning: What and how do students learn? *Educational psychology review*, Springer, v. 16, n. 3, p. 235–266, 2004. Cited on page 25.

IBANEZ, M.; DI-SERIO, A.; DELGADO-KLOOS, C. Gamification for engaging computer science students in learning activities: A case study. *IEEE Transactions on Learning Technologies*, v. 7, n. 3, p. 291–301, July 2014. Cited on page 39.

IEEE. *Curriculum and Accreditation Committee*. 2013. Available in: https://www.computer.org/web/peb/curricula. Cited on page 24.

IHANTOLA, Petri; VIHAVAINEN, Arto; AHADI, Alireza; BUTLER, Matthew; BöRSTLER, Jürgen; EDWARDS, Stephen H.; ISOHANNI, Essi; KORHONEN, Ari; PETERSEN, Andrew; RIVERS, Kelly; RUBIO, Miguel Ángel; SHEARD, Judy; SKUPAS, Bronius; SPACCO, Jaime; SZABO, Claudia; TOLL, Daniel. Educational data mining and learning analytics in programming: Literature review and case studies. In: *Proceedings of the 2015 ITiCSE on Working Group Reports*. New York, NY, USA: ACM, 2015. (ITICSE-WGR '15), p. 41–63. Cited on page 78.

ILIC, Milos; SPALEVIC, Petar; VEINOVIC, Mladen; ALATRESH, Wejdan Saed. Students' success prediction using weka tool. *Infoteh-Jahorina*, v. 15, p. 684–688, 2016. Cited on page 79.

JAIN, Anil K. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, v. 31, n. 8, p. 651 – 666, 2010. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR). Cited 3 times on pages 47, 49, and 93.

JAIN, Anil K; MURTY, M Narasimha; FLYNN, Patrick J. Data clustering: a review. *ACM computing surveys (CSUR)*, Acm New York, NY, USA, v. 31, n. 3, p. 264–323, 1999. Cited on page 96.

JAZAYERI, Mehdi. Combining mastery learning with project-based learning in a first programming course: An experience report. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 315–318. Cited on page 39.

JONES, Karen Sparck. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, MCB UP Ltd, v. 28, n. 1, p. 11–21, 1972. Cited on page 59.

JUNG, Carlos Fernando. *Metodologia para pesquisa e desenvolvimento: aplicada a novas tecnologias, produtos e processos*. Rio de Janeiro, RJ, BR: Axcel Books, 2004. Cited 2 times on pages 87 and 89.

KASTO, Nadia; WHALLEY, Jacqueline. Measuring the difficulty of code comprehension tasks using software metrics. In: *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*. AUS: Australian Computer Society, Inc., 2013. (ACE '13), p. 59–65. Cited on page 85.

KEUNING, Hieke; JEURING, Johan; HEEREN, Bastiaan. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*, Association for Computing Machinery, New York, NY, USA, v. 19, n. 1, sep 2018. Cited 3 times on pages 24, 25, and 75.

KHALIL, Mohammed K.; HAWKINS, H. Gregory; CRESPO, Lynn M.; BUGGY, James. The relationship between learning and study strategies inventory (lassi) and academic

performance in medical schools. *Medical Science Educator*, v. 27, n. 2, p. 315–320, June 2017. Cited on page 79.

KINNUNEN, Päivi; MALMI, Lauri. Why students drop out cs1 course? In: *Proceedings of the Second International Workshop on Computing Education Research*. New York, NY, USA: ACM, 2006. (ICER '06), p. 97–108. Cited on page 24.

KOKOTSAKI, Dimitra; MENZIES, Victoria; WIGGINS, Andy. Project-based learning: A review of the literature. *Improving schools*, SAGE Publications Sage UK: London, England, v. 19, n. 3, p. 267–277, 2016. Cited on page 25.

KöLLING, Michael; BARNES, David J. Enhancing apprentice-based learning of java. In: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2004. (SIGCSE '04), p. 286–290. Cited on page 38.

KOREN, Yehuda; HAREL, David. A two-way visualization method for clustered data. In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: ACM, 2003. (KDD '03), p. 589–594. Cited on page 51.

KOTHIYAL, Aditi; MAJUMDAR, Rwitajit; MURTHY, Sahana; IYER, Sridhar. Effect of think-pair-share in a large cs1 class: 83% sustained engagement. In: *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. New York, NY, USA: ACM, 2013. (ICER '13), p. 137–144. Cited on page 25.

KOZIK, R.; CHORAS, M.; PUCHALSKI, D.; RENK, R. Q-rapids framework for advanced data analysis to improve rapid software development. *Journal of Ambient Intelligence and Humanized Computing*, v. 10, p. 1927–1936, 2019. Cited on page 73.

LANCE, G. N.; WILLIAMS, W. T. Mixed-data classificatory programs i - agglomerative systems. *Australian Computer Journal*, v. 1, p. 15–20, 1967. Cited on page 47.

LATULIPE, Celine; LONG, N. Bruce; SEMINARIO, Carlos E. Structuring flipped classes with lightweight teams and gamification. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 392–397. Cited on page 35.

LEFFINGWELL, Dean. *Scaling software agility: best practices for large enterprises*. Boston, MA, USA: Pearson Education, 2007. Cited on page 70.

LEGÁNY, Csaba; JUHÁSZ, Sándor; BABOS, Attila. Cluster validity measurement techniques. In: *Proceedings of the 5th international conference on artificial intelligence, knowledge engineering and data bases*. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2006. p. 388–393. Cited on page 55.

LEVY, Steven. How google's algorithm rules the web. *Wired Magazine*, v. 18, n. 3, 2010. Cited on page 57.

LI, Yang. Reengineering a scientific software and lessons learned. In: *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*. New York, NY, USA: Association for Computing Machinery, 2011. p. 41–45. Cited on page 71.

LI, Zhen; KRAEMER, Eileen. Social effects of pair programming mitigate impact of bounded rationality. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 385–390. Cited on page 38.

LIEBIG, Jörg; JANKER, Andreas; GARBE, Florian; APEL, Sven; LENGAUER, Christian. Morpheus: Variability-aware refactoring in the wild. In: *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, 2015. v. 1, p. 380–391. Cited on page 71.

LIU, Xiao; WANG, Shuai; WANG, Pei; WU, Dinghao. Automatic grading of programming assignments: An approach based on formal semantics. In: *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. Montreal, QC, Canada: IEEE, 2019. p. 126–137. Cited 2 times on pages 76 and 84.

LIU, Yanchi; LI, Zhongmou; XIONG, Hui; GAO, Xuedong; WU, Junjie. Understanding of internal clustering validation measures. In: *Proceedings of the 2010 IEEE International Conference on Data Mining*. Sydney, NSW, Australia: IEEE, 2010. p. 911–916. Cited on page 98.

LLOYD, Stuart. Least squares quantization in pcm. *IEEE transactions on information theory*, IEEE, v. 28, n. 2, p. 129–137, 1982. Cited on page 47.

LÓPEZ, Félix; ROMERO, Víctor. *Mastering Python Regular Expressions*. Birmingham, UK: Packt Publishing Ltd, 2014. Cited on page 60.

LOVELLETTE, Ellie; MATTA, John; BOUVIER, Dennis; FRYE, Roger. Just the numbers: An investigation of contextualization of problems for novice programmers. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 393–398. Cited on page 39.

LUCAS, Joan M. Illustrating the interaction of algorithms and data structures using the matching problem. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 247–252. Cited on page 39.

LUXTON-REILLY, Andrew; AJANOVSKI, Vangel V; FOUH, Eric; GONSALVEZ, Christabel; LEINONEN, Juho; PARKINSON, Jack; POOLE, Matthew; THOTA, Neena. Pass rates in introductory programming and in other stem disciplines. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2019. p. 53–71. Cited on page 24.

MACQUEEN, James et al. Some methods for classification and analysis of multivariate observations. In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Oakland, CA, USA: Lucien M. Le Cam, Jerzy Neyman, 1967. v. 1, n. 14, p. 281–297. Cited on page 47.

MAIA, M. C.; SEREY, D.; FIGUEIREDO, J. Learning styles in programming education: A systematic mapping study. In: *Proceedings of the 47th Frontiers in Education Annual Conference*. Indianapolis, IN, USA: IEEE, 2017. (FIE '17, v. 00), p. 1–7. Cited on page 25.

MARTIN, R. Oo design quality metrics : an analysis of dependencies. *ROAD 1995*, v. 2, n. 3, 1995. Cited on page 69.

MCCABE, T. J. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2, n. 4, p. 308–320, 1976. Cited on page 64.

MCCHESNEY, Ian. Three years of student pair programming: Action research insights and outcomes. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 84–89. Cited on page 38.

MCKIGHT, Patrick E; NAJAB, Julius. Kruskal-wallis test. *The corsini encyclopedia of psychology*, Wiley Online Library, p. 1–1, 2010. Cited on page 99.

MCKNIGHT, Patrick E; NAJAB, Julius. Mann-whitney u test. *The Corsini encyclopedia of psychology*, Wiley Online Library, p. 1–1, 2010. Cited on page 99.

MCQUAIGUE, Matthew; BURLINSON, David; SUBRAMANIAN, Kalpathi; SAULE, Erik; PAYTON, Jamie. Visualization, assessment and analytics in data structures learning modules. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 864–869. Cited on page 42.

MEDEIROS, R. P.; RAMALHO, G. L.; FALCãO, T. P. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, v. 62, n. 2, p. 77–90, May 2019. Cited on page 25.

MICHIE, Michael G. Use of the bray-curtis similarity measure in cluster analysis of foraminiferal data. *Journal of the International Association for Mathematical Geology*, Springer, v. 14, n. 6, p. 661–667, 1982. Cited on page 46.

MOOERS, Calvin N. Zatocoding applied to mechanical organization of knowledge. *American Documentation*, v. 2, n. 1, p. 20–32, 1951. Cited on page 58.

MORESI, Marco; GóMEZ, Marcos J.; BENOTTI, Luciana. Predicting students' difficulties from a piece of code. *IEEE Transactions on Learning Technologies*, v. 14, n. 3, p. 386–399, 2021. Cited on page 84.

MORRISON, Briana B.; MARGULIEUX, Lauren E.; ERICSON, Barbara; GUZDIAL, Mark. Subgoals help students solve parsons problems. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 42–47. Cited on page 39.

MULLNER, Daniel. Modern hierarchical, agglomerative clustering algorithms. *CoRR*, abs/1109.2378, 2011. Cited 2 times on pages 49 and 52.

MUNSON, J. C.; KHOSHGOFTAAR, T. M. Measuring dynamic program complexity. *IEEE Software*, v. 9, n. 6, p. 48–55, 1992. Cited on page 64.

MUNSON, Jonathan P.; ZITOVSKY, Joshua P. Models for early identification of struggling novice programmers. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 699–704. Cited 2 times on pages 78 and 84.

NIDHRA, Srinivas; DONDETI, Jagruthi. Black box and white box testing techniques - a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, v. 2, p. 29–50, June 2012. Cited on page 77.

NURMINEN, Jukka. Using software complexity measures to analyze algorithms - an experiment with the shortest-paths algorithms. *Computers & OR*, v. 30, p. 1121–1134, July 2003. Cited on page 68.

OLAGUE, Hector M.; ETZKORN, Letha H.; COX, Glenn W. An entropy-based approach to assessing object-oriented software maintainability and degradation - a method and case study. In: *Proceedings of the Software Engineering Research and Practice*. Las Vegas, Nevada, USA: Citeseer, 2006. Cited on page 66.

OLIVEIRA, M. F. S.; REDIN, R. M.; CARRO, L.; LAMB, L. d. C.; WAGNER, F. R. Software quality metrics and their impact on embedded software. In: *Proceedings of the 2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*. Budapest, Hungary: IEEE, 2008. p. 68–77. Cited 7 times on pages 62, 63, 64, 65, 67, 103, and 104.

OTT, Linda M; THUSS, Jeffrey J. Slice based metrics for estimating cohesion. In: *Proceedings of the IEEE First International Software Metrics Symposium*. Baltimore, MD, USA: IEEE, 1993. p. 71–81. Cited on page 69.

OYELADE, OJ; OLADIPUPO, Olufunke O; OBAGBUWA, IC. Application of k means clustering algorithm for prediction of students academic performance. *arXiv preprint arXiv:1002.2425*, 2010. Cited on page 79.

OZTURK, Alisan; BONFERT-TAYLOR, Petra; FUGENSCHUH, Armin. Using data to improve programming instruction. In: KROMKER, Detlef; SCHROEDER, Ulrik (Ed.). *DeLFI 2018 - Die 16. E-Learning Fachtagung Informatik*. Bonn: Gesellschaft für Informatik e.V., 2018. p. 23–32. Cited 3 times on pages 77, 78, and 84.

PAI, GA Vijayalakshmi; MICHEL, Thierry. Evolutionary optimization of constrained k-means clustered assets for diversification in small portfolios. *IEEE Transactions on Evolutionary Computation*, IEEE, v. 13, n. 5, p. 1030–1053, 2009. Cited on page 45.

PARIHAR, Sagar; DADACHANJI, Ziyaan; SINGH, Praveen Kumar; DAS, Rajdeep; KARKARE, Amey; BHATTACHARYA, Arnab. 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*. New York, NY, USA: Association for Computing Machinery, 2017. (ITiCSE '17), p. 92–97. Cited 2 times on pages 77 and 84.

PEARS, Arnold; SEIDMAN, Stephen; MALMI, Lauri; MANNILA, Linda; ADAMS, Elizabeth; BENNEDSEN, Jens; DEVLIN, Marie; PATERSON, James. A survey of literature on the teaching of introductory programming. In: *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*. New York, NY, USA: ACM, 2007. (ITiCSE-WGR '07), p. 204–223. Cited on page 24.

PETERSEN, Andrew; CRAIG, Michelle; CAMPBELL, Jennifer; TAFLIOVICH, Anya. Revisiting why students drop cs1. In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. New York, NY, USA: ACM, 2016. (Koli Calling '16), p. 71–80. Cited on page 25.

PETERSEN, Kai; FELDT, Robert; MUJTABA, Shahid; MATTSSON, Michael. Systematic mapping studies in software engineering. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. Swindon, UK: BCS Learning & Development Ltd., 2008. (EASE'08), p. 68–77. Cited on page 33.

PETTIT, Raymond; HOMER, John; GEE, Roger; MENGEL, Susan; STARBUCK, Adam. An empirical study of iterative improvement in programming assignments. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 410–415. Cited on page 40.

POLITO, Giuseppina; TEMPERINI, Marco; STERBINI, Andrea. 2tsw: Automated assessment of computer programming assignments, in a gamified web based system. In: *Proceedings of the 2019 IEEE 18th International Conference on Information Technology Based Higher Education and Training (ITHET)*. Magdeburg, Germany: IEEE, 2019. p. 1–9. Cited 2 times on pages 76 and 84.

POLYZOU, Agoritsa; KARYPIS, George. Feature extraction for next-term prediction of poor student performance. *IEEE Transactions on Learning Technologies*, IEEE, v. 12, n. 2, p. 237–248, 2019. Cited on page 79.

PORFIRIO, Andres; PEREIRA, Roberto; MASCHIO, Eleandro. A-learn evid: A method for identifying evidence of computer programming skills through automatic source code assessment. *Revista Brasileira de Informática na Educação*, v. 29, n. 0, 2021. Cited 2 times on pages 77 and 84.

PORTER, Leo; BOUVIER, Dennis; CUTTS, Quintin; GRISSOM, Scott; LEE, Cynthia; MCCARTNEY, Robert; ZINGARO, Daniel; SIMON, Beth. A multi-institutional study of peer instruction in introductory computing. *ACM Inroads*, ACM, New York, NY, USA, v. 7, n. 2, p. 76–81, May 2016. Cited 2 times on pages 37 and 38.

POSNETT, Daryl; HINDLE, Abram; DEVANBU, Premkumar. A simpler model of software readability. In: *Proceedings of the 8th working conference on mining software repositories*. New York, NY, USA: Association for Computing Machinery, 2011. (MSR '11), p. 73–82. Cited on page 74.

PRETE, Kyle; RACHATASUMRIT, Napol; SUDAN, Nikita; KIM, Miryung. Template-based reconstruction of complex refactorings. In: *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. Timisoara, Romania: IEEE, 2010. p. 1–10. Cited on page 72.

PRICE, Thomas W.; DONG, Yihuan; LIPOVAC, Dragan. isnap: Towards intelligent tutoring in novice programming environments. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 483–488. Cited on page 42.

PURBA, Windania; TAMBA, Saut; SARAGIH, Jepronel. The effect of mining data k-means clustering toward students profile model drop out potential. In: *Journal of Physics: Conference Series*. Prima, Indonesia: IOP Publishing, 2018. v. 1007, n. 1, p. 012049. Cited on page 79.

RANA, Shiwani; GARG, Roopali. Application of hierarchical clustering algorithm to evaluate students performance of an institute. In: *Proceedings of the 2nd international*

conference on computational intelligence & communication technology (CICT). Ghaziabad, India: IEEE, 2016. p. 692–697. Cited on page 79.

RANI, U.; SAHU, S. Comparison of clustering techniques for measuring similarity in articles. In: *Proceedings of the 2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*. Ghaziabad, Uttar, India: IEEE, 2017. p. 1–7. Cited on page 97.

REIMANN, Jan; WILKE, Claas; DEMUTH, Birgit; MUCK, Michael; ASSMANN, Uwe. Tool supported ocl refactoring catalogue. In: *Proceedings of the 12th Workshop on OCL and Textual Modelling*. New York, NY, USA: Association for Computing Machinery, 2012. p. 7–12. Cited 2 times on pages 71 and 72.

REZAEI, Mohammad. Clustering validation. *University of Eastern Finland*, 2016. Cited on page 54.

ROBINS, Anthony; ROUNTREE, Janet; ROUNTREE, Nathan. Learning and teaching programming: A review and discussion. *Computer Science Education*, Routledge, v. 13, n. 2, p. 137–172, 2003. Cited on page 24.

RODRÍGUEZ, Fernando J.; PRICE, Kimberly Michelle; BOYER, Kristy Elizabeth. Exploring the pair programming process: Characteristics of effective collaboration. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 507–512. Cited 4 times on pages 26, 38, 42, and 79.

ROKACH, Lior; MAIMON, Oded. Clustering methods. In: *Data mining and knowledge discovery handbook*. New York, NY, USA: Springer, 2005. p. 321–352. Cited 4 times on pages 45, 48, 55, and 97.

SAHU, Lokesh; MOHAN, Biju R. An improved k-means algorithm using modified cosine distance measure for document clustering using mahout with hadoop. In: *Proceedings of the 2014 9th International Conference on Industrial and Information Systems (ICIIS)*. Gwalior, India: IEEE, 2014. p. 1–5. Cited on page 46.

SANTOS, Priscila; ARAUJO, Luis; BITTENCOURT, Roberto. A mapping study of computational thinking and programming in brazilian k-12 education. In: *Proceedings of the 48th Frontiers in Education Annual Conference*. San Jose, CA, USA: IEEE, 2018. (FIE '18), p. 1–8. Cited on page 25.

SARWAR, Mir Muhammd Suleman; SHAHZAD, Sara; AHMAD, Ibrar. Cyclomatic complexity: The nesting problem. In: *Proceedings of the Eighth International Conference on Digital Information Management (ICDIM 2013)*. Islamabad, Pakistan: IEEE, 2013. p. 274–279. Cited on page 64.

SCAICO, Pasqueline; SCAICO, Alexandre; QUEIROZ, Ruy José Guerra Barretto De. An initial analysis of the research on interest and introductory programming. In: *Proceedings of the 2018 IEEE Frontiers in Education Conference (FIE)*. San Jose, CA, USA: IEEE, 2018. (FIE '18), p. 1–9. Cited on page 25.

SCHREIBER, Benjamin J.; DOUGHERTY, John P. Assessment of introducing algorithms with video lectures and pseudocode rhymed to a melody. In: *Proceedings of the 2017 ACM*

SIGCSE Technical Symposium on Computer Science Education. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 519–524. Cited on page 42.

SCHWAB, Klaus. *The future of jobs report 2018*. Geneva, Switzerland; World Economic Forum, 2018. Cited on page 23.

SHARMA, Rashmi; SAHA, Anju. A systematic review of software testability measurement techniques. In: *Proceedings of the 2018 International Conference on Computing, Power and Communication Technologies (GUCON)*. Greater Noida, India: IEEE, 2018. p. 299–303. Cited on page 76.

SHETH, Swapneel; MURPHY, Christian; ROSS, Kenneth A.; SHASHA, Dennis. A course on programming and problem solving. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 323–328. Cited on page 39.

SIEGEL, Sidney. Nonparametric statistics for the behavioral sciences. McGraw-hill, 1956. Cited on page 100.

SILVA, Davi Bernardo; DECONTO, Diogo Steinke; AGUIAR, Rafael de Lima; SILLA, Carlos N. Recent studies about teaching algorithms (cs1) and data structures (cs2) for computer science students. In: *Proceedings of the 49th Frontiers in Education Annual Conference*. Cincinnati, OH, USA: IEEE, 2019. (FIE '19). Cited 4 times on pages 25, 31, 75, and 149.

SILVA, Davi Bernardo; SILLA, Carlos N. Evaluation of students programming skills on a computer programming course with a hierarchical clustering algorithm. In: *Proceedings of the 50th Frontiers in Education Annual Conference*. Uppsala, Sweden, UK: IEEE, 2020. (FIE '20). Cited 5 times on pages 80, 84, 122, 150, and 152.

SILVA, Matheus Camilo da; BASSI, Patricia Rucker de; WANDERLEY, Gregory Moro Puppi; TACLA, Cesar Augusto; PARAISO, Emerson Cabrera. A visual tool for supporting collaborative code quality. In: *Proceedings of the 2019 IEEE 23rd International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. Porto, Portugal: IEEE, 2019. p. 368–373. Cited on page 73.

SINWAR, Deepak; KAUSHIK, Rahul. Study of euclidean and manhattan distance metrics using simple k-means clustering. *International Journal for Research in Applied Science and Engineering Technology*, v. 2, n. 5, p. 270–274, 2014. Cited on page 46.

SOUZA, Fabio Rezende de; ZAMPIROLLI, Francisco de Assis; KOBAYASHI, Guiou. Convolutional neural network applied to code assignment grading. In: *Proceedings of the International Conference on Computer Supported Education*. Crete, Greece: SCITEPRESS, 2019. Cited 2 times on pages 77 and 84.

SOUZA, RMCR De; CARVALHO, FAT De. Dynamic clustering of interval data based on adaptive chebyshev distances. *Electronics Letters*, IET, v. 40, n. 11, p. 658–660, 2004. Cited on page 46.

SPACCO, Jaime; DENNY, Paul; RICHARDS, Brad; BABCOCK, David; HOVEMEYER, David; MOSCOLA, James; DUVALL, Robert. Analyzing student work patterns using programming exercise data. In: *Proceedings of the 46th ACM Technical Symposium on*

Computer Science Education. New York, NY, USA: Association for Computing Machinery, 2015. (SIGCSE '15), p. 18–23. Cited on page 24.

SZABO, Claudia; FALKNER, Nickolas. Silence, words, or grades: The effects of lecturer feedback in multi-revision assignments. In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. New York, NY, USA: ACM, 2017. (ITiCSE '17), p. 293–298. Cited on page 25.

TAN, Pang-Ning; STEINBACH, Michael; KUMAR, Vipin. *Introduction to data mining*. Florida, USA: Pearson Education India, 2016. Cited 3 times on pages 52, 53, and 54.

TANG, Terry; RIXNER, Scott; WARREN, Joe. An environment for learning interactive programming. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 671–676. Cited on page 41.

THEODORIDIS, S; KOUTROUBAS, K. Feature generation ii. *Pattern recognition*, v. 2, p. 269–320, 1999. Cited on page 54.

TIANYI, Shao; YULIN, Kuang; YIHONG, Huang; YUJUAN, Quan. Paaa: An implementation of programming assignments automatic assessing system. In: *Proceedings of the 2019 4th International Conference on Distance Education and Learning*. New York, NY, USA: Association for Computing Machinery, 2019. p. 68–72. Cited 2 times on pages 77 and 84.

ULLAH, Zahid; LAJIS, Adidah; JAMJOOM, Mona; ALTALHI, Abdulrahman; AL-GHAMDI, Abdullah; SALEEM, Farrukh. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education*, v. 26, n. 6, p. 2328–2341, 2018. Cited 2 times on pages 77 and 146.

VANDEGRIFT, Tammy. Pogil activities in data structures: What do students value? In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2017. (SIGCSE '17), p. 597–602. Cited on page 40.

VIHAVAINEN, Arto; AIRAKSINEN, Jonne; WATSON, Christopher. A systematic review of approaches for teaching introductory programming and their influence on success. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. New York, NY, USA: ACM, 2014. (ICER '14), p. 19–26. Cited on page 24.

WATSON, Christopher; LI, Frederick W.B. Failure rates in introductory programming revisited. In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. New York, NY, USA: ACM, 2014. (ITiCSE '14), p. 39–44. Cited on page 24.

WEISER, Mark. Program slicing. *IEEE Transactions on software engineering*, IEEE, n. 4, p. 352–357, 1984. Cited on page 69.

WHALLEY, Jacqueline; KASTO, Nadia. How difficult are novice code writing tasks? a software metrics approach. In: *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*. AUS: Australian Computer Society, Inc., 2014. (ACE '14), p. 105–112. Cited on page 85.

WOOD, Diana F. Problem based learning. *BMJ (Clinical research ed.)*, v. 336, p. 971, June 2008. Cited on page 38.

WOOD, Zoë; KEEN, Aaron. Building worlds: Bridging imperative-first and object-oriented programming in cs1-cs2. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 144–149. Cited on page 39.

XU, Rui; WUNSCH, Don. *Clustering*. Hoboken, NJ, USA: John Wiley & Sons, 2008. v. 10. (IEEE Press Series on Computational Intelligence, v. 10). Cited 6 times on pages 48, 50, 51, 52, 53, and 54.

YADAV, Ramjeet Singh; AHMED, P; SONI, AK; PAL, Saurabh. Academic performance evaluation using soft computing techniques. *Current Science*, JSTOR, p. 1505–1517, 2014. Cited on page 79.

YAN, Lisa; MCKEOWN, Nick; SAHAMI, Mehran; PIECH, Chris. Tmoss: Using intermediate assignment work to understand excessive collaboration in large classes. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 110–115. Cited 2 times on pages 25 and 42.

YOUNG, Jeffrey; WALKINGSHAW, Eric. A domain analysis of data structure and algorithm explanations in the wild. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 870–875. Cited on page 42.

YOURDON, Edward; CONSTANTINE, Larry L. *Structured design: fundamentals of a discipline of computer program and systems design*. Englewood Cliffs, NJ, USA: Prentice-Hall, Inc., 1979. Cited on page 69.

ZAINUDDIN, Zamzami; CHU, Samuel Kai Wah; SHUJAHAT, Muhammad; PERERA, Corinne Jacqueline. The impact of gamification on learning and instruction: A systematic review of empirical evidence. *Educational Research Review*, Elsevier, v. 30, p. 100326, 2020. Cited on page 25.

ZARB, Mark; HUGHES, Janet; RICHARDS, John. Evaluating industry-inspired pair programming communication guidelines with undergraduate students. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 361–366. Cited on page 38.

ZARB, Mark; HUGHES, Janet; RICHARDS, John. Further evaluations of industry-inspired pair programming communication guidelines with undergraduate students. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 314–319. Cited on page 38.

ZAVALA, Laura; MENDOZA, Benito. On the use of semantic-based aig to automatically generate programming exercises. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 14–19. Cited 2 times on pages 26 and 43.

ZHENG, Alice; CASARI, Amanda. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. 1st. ed. North, Sebastopol, CA: O'Reilly Media, Inc., 2018. Cited on page 58.

ZHU, Xiaojin; GOLDBERG, Andrew B. *Introduction to semi-supervised learning*. Oregon: Springer Nature, 2022. Cited 2 times on pages 45 and 93.

ZINGARO, Daniel. Peer instruction contributes to self-efficacy in cs1. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 373–378. Cited on page 37.