AUTOMATED CODE INTEGRITY CHECKER: PLAGIARISM DETECTION AND GRADING EVALUATION TOOL FOR COLLEGE ASSIGNMENTS

A PROJECT REPORT

Submitted By

G SANJITH. 205001092

S PREMKUMARAN. 205001079

S MUKILESH. 205001306

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Department of Computer Science and Engineering

Sri Sivasubramaniya Nadar College of Engineering
(An Autonomous Institution, Affiliated to Anna University)
Kalavakkam - 603110

May 2024

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

BONAFIDE CERTIFICATE

Certified that this project report titled **"Automated Code Integrity Checker: Plagiarism Detection and Grading Evaluation Tool for College Assignments"** is the *bonafide* work of "**G SANJITH (205001092)**, **S PREMKUMARAN (205001079)**, and **S MUKILESH (205001306)**" who carried out the project work under my supervision.

Certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Dr. T.T. MIRNALINEE
Head of the Department
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Dr. S SARASWATHI
Supervisor
Associate Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Place:
Date:

Submitted for the examination held on. . . . . .

Internal Examiner **External Examiner**

ACKNOWLEDGEMENTS

We thank GOD, the almighty for giving us strength and knowledge to do this project.

We would like to thank and deep sense of gratitude to our guide **Dr. S SARASWATHI** , Associate Professor, Department of Computer Science and Engineering, for her valuable advice and suggestions as well as her continued guidance, patience and support that helped me to shape and refine my work.

Our sincere thanks to **Dr. T.T. MIRNALINEE**, Professor and Head of the Department of Computer Science and Engineering, for her words of advice and encouragement and we would like to thank our project Coordinator **Dr. S SARASWATHI** , Associate Professor, Department of Computer Science and Engineering for her valuable suggestions throughout this project.

We express our deep respect to the founder **Dr. SHIV NADAR**, Chairman, SSN Institutions. We also express our appreciation to our **Dr. V. E. ANNAMALAI**, Principal, for all the help he has rendered during this course of study.

We would like to extend our sincere thanks to all the teaching and non-teaching staffs of our department who have contributed directly and indirectly during the course of my project work. Finally, We would like to thank my parents and friends for their patience, cooperation and moral support throughout our life.

G SANJITH. **S PREMKUMARAN.** **S MUKILESH.**

ABSTRACT

In the digital age, academic integrity is crucial, especially with the widespread availability of information and high academic pressure. Implementing plagiarism detection tools within Learning Management Systems (LMS) is essential for educational institutions. These tools not only discourage academic dishonesty but also promote originality and critical thinking among students. By scanning assignments for content duplication and improper citations, educators can address issues promptly, maintaining the integrity of assessments.

Our college's LMS features a code analysis tool designed to streamline the evaluation of student assignments. This tool extracts and analyzes code, providing metric scores that assess code quality and structure. These scores are compiled into a matrix format, serving as a centralized repository of coding proficiency among students. Through systematic clustering, assignments are grouped based on similarities and discrepancies, facilitating trend identification and targeted intervention by educators.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The identification of code plagiarism is imperative for safeguarding the integrity of software development and protecting the intellectual property of original developers. Additionally, it ensures that software remains secure, maintains high quality, and adheres to legal standards. Despite its critical importance, code plagiarism remains a pervasive issue in academia, impacting teachers and students across various disciplines. Under the pressures of exams and assessments, students often resort to duplicating code, prioritizing short-term gains over genuine learning. However, the consequences of code plagiarism extend far beyond the classroom, affecting students' academic standing and future employment prospects. Faculty members face the challenge of combating plagiarism while upholding academic integrity, diverting valuable resources from teaching and research. To effectively address this issue, academic institutions must cultivate a culture of academic integrity and implement robust detection procedures.

In response to the growing need for effective plagiarism detection, our college's Learning Management System (LMS) is implementing a state-of-the-art code plagiarism detection tool. This tool utilizes advanced algorithms to analyze submitted code assignments, identifying instances of unauthorized copying and ensuring the originality and integrity of students' work. By leveraging tokenization and parsing processes aligned with predefined grammar rules, the tool thoroughly examines code submissions to detect similarities, improper citation, and unauthorized collaboration.

The code plagiarism detection tool goes beyond simply identifying instances of direct code replication. It also examines syntactical structures, variable names, and alterations made to the codebase, providing a comprehensive assessment of code quality and structure. Through a systematic process, the tool facilitates the clustering of assignments based on similarities and discrepancies, enabling educators to pinpoint areas requiring attention and intervention. By providing educators with actionable insights, the tool empowers them to uphold academic integrity and promote originality among students.

Implementing a robust code plagiarism detection tool within the college's LMS is a proactive step towards fostering a culture of academic integrity. By detecting and deterring instances of code plagiarism, the tool encourages students to engage in genuine learning experiences and develop their coding skills ethically. Moreover, it reinforces the importance of integrity and honesty in academic and professional settings, preparing students for success in their future careers.

Furthermore, the integration of the code plagiarism detection tool enhances the credibility and reputation of the college by demonstrating its commitment to academic excellence and integrity. Potential employers value graduates who possess not only technical skills but also ethical conduct and integrity. By equipping students with the tools and resources to uphold academic integrity, the college plays a vital role in shaping the future professionals of tomorrow.

Hence the implementation of a code plagiarism detection tool within the college's LMS is a proactive measure to address the pervasive issue of code plagiarism and uphold academic integrity. By leveraging advanced algorithms and analysis techniques, the tool enables educators to detect and deter instances of unauthorized code copying, safeguarding the integrity of software development

and protecting the intellectual property of original developers. Moreover, it cultivates a culture of academic integrity, promoting originality, honesty, and ethical conduct among students. As academic institutions continue to prioritize integrity and excellence, the integration of robust detection procedures becomes essential in ensuring the credibility and reputation of educational programs and preparing students for success in their future careers.

1.1 MOTIVATION

In both academia and industry, the Measure Of Software Similarity (MOSS) process is used for a variety of aims, but its main goal is to identify instances of code plagiarism. By detecting instances of unauthorised code copying, it assists teachers in upholding academic integrity by comparing submitted code against an extensive database of pre existing solutions. It also acts as a feedback tool for pupils, pointing out areas in which their work can be lacking originality and promoting changes. Additionally, MOSS facilitates discussions about coding techniques and best practices and highlights team members' shared solutions in collaborative learning environments. MOSS, being a paid software, comes with significant costs, making it inaccessible for many institutions. Moreover, its lack of emphasis on the structural integrity of code poses limitations, particularly in detecting reordered elements within code. While MOSS offers a holistic approach, it may not effectively handle smaller code snippets in depth. To address these concerns, we are developing a system to overcome these limitations.

1.2 BACKGROUND

The Plagiarism detection methods can be of two types that is Textual plagiarism and Source code plagiarism. The Source code similarity detection can be classified into five categories.

1) Strings: In this approach the strings will be matched but this type of plagiarism can be hidden from detection by renaming the identifiers in the source code.

2) Tokens: In this approach firstly the program is converted into token with the help of lexer. This will help to ignore the identifier names, comments in the codes and white space.

3) Parse Trees: In this approach, first Parse Trees of both source codes are made and then they are compared . If both trees are equal, then one will say both source code are similar otherwise not.

4) Program Dependency Graphs (PDGs): With the help of PDGs one can capture the actual flow of control. This PDGs can identify the equivalence but require huge computation and is complex to perform.

5) Metrics: It assigns ‘scores’ based on certain parameters to the code segments. These scores can be provided based on counting the number of loops or conditional statement or number of variables in the code.

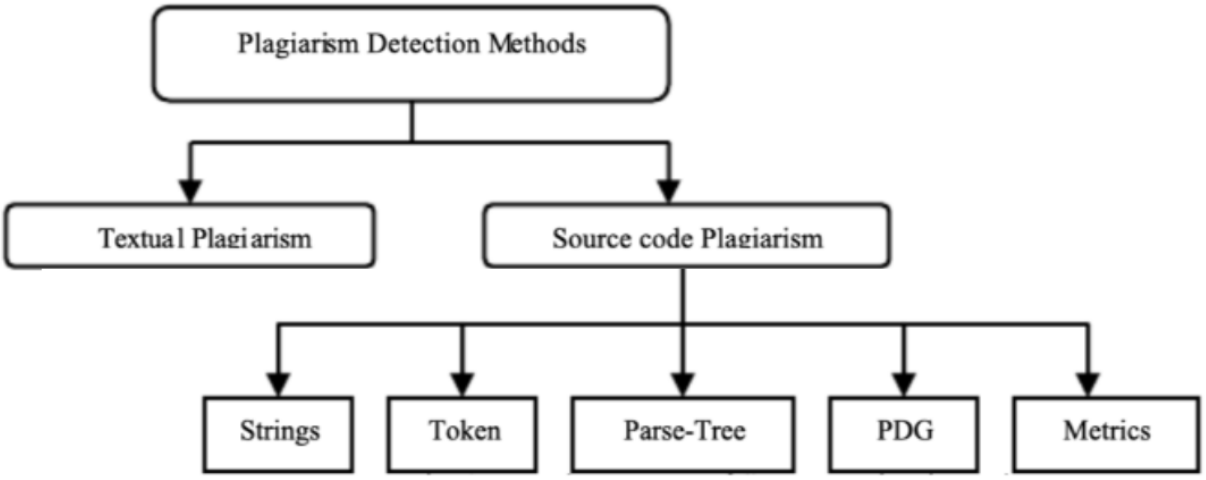


FIGURE 1.1: Example of an Overall Plagiarism checker

From the above categories, different areas of integrity and functions can be obtained in day to day life which includes the folowing :-

1. **Quality Assurance:** Software projects' security and quality may be put at risk by faults or weaknesses in plagiarized code. Identifying plagiarism contributes to the upkeep of code bases' dependability and quality.

2. **Code Re usability and Modularity:** In order to promote effective development techniques, plagiarism detection can also help discover situations in which code can be modularized or reused within an organization's projects.

3. **Improvement in Code Reviews:** Tools for detecting plagiarism can be incorporated into code review procedures, assisting reviewers in spotting duplicate code and offering chances for growth and learning.

4. **Algorithmic Analysis:** Certain plagiarism detection technologies offer information on common patterns, coding styles, and algorithms. This information can be helpful for teaching reasons and for recognising patterns in software development.

5. **Academic Sector:** These detectors assist educators in making sure that students are turning in unique work and aren't replicating it from the internet or from other people. This encourages fairness and integrity in the classroom.

1.3 PROBLEM DEFINITION

The current plagiarism and error detection system utilized is MOSS (Measure of Software Similarity). While MOSS offers a comprehensive analysis of code similarity across various programming languages, it lacks the capability to provide structural similarity analysis. Additionally, being a paid software, MOSS imposes financial constraints on the college. Furthermore, although MOSS provides a holistic view of code similarity, it falls short when addressing specific coding problems, often failing to provide in-depth answers.

To overcome these limitations and enhance the plagiarism and error detection capabilities within our college's LMS, we aim to develop a new system tailored to the unique needs of our institution. This new system will offer advanced structural similarity analysis, allowing educators to assess not only the surface-level similarities but also the underlying structural similarities between code submissions. By incorporating this feature, the new system will provide a

more nuanced understanding of code similarity, enabling educators to identify instances of plagiarism with greater accuracy.

Moreover, our new system will be developed as an in-house solution, eliminating the need for costly licensing fees associated with third-party software like MOSS. By leveraging internal resources and expertise, we can customize the system to address specific coding problems relevant to our curriculum and teaching methodologies. This customization will enable educators to receive more targeted and meaningful insights into code similarity, facilitating more effective interventions to address plagiarism and promote academic integrity.

In summary, the problem at hand involves the need to improve the plagiarism and error detection capabilities within our college's LMS. The current system, MOSS, lacks structural similarity analysis and imposes financial constraints on the institution. To address these challenges, we propose the development of a new system tailored to our institution's needs, offering advanced structural similarity analysis and customized solutions to enhance code plagiarism detection and uphold academic integrity.

1.4 ORGANIZATION OF REPORT

This thesis discusses the development of a code plagiarism and error detection software designed specifically for integration into our college's LMS. The software aims to simplify the grading process for faculty members by providing an efficient platform for assessing student assignments submitted through the LMS. We begin by discussing the broad applications and significance of code plagiarism detection, spanning from academic institutions to professional workplaces. We address the challenges students and faculties face due to code plagiarism, which can hinder their learning experience and academic integrity, while also acknowledging the potential consequences in professional settings. Furthermore, we introduce various techniques used in code plagiarism detection tools, such as string matching, Program Dependence Graphs (PDG), and Abstract Syntax Trees (AST).

In the second chapter, we delve into an extensive analysis of various techniques utilized in code plagiarism detection, drawing insights from a comprehensive study of research papers. This analysis encompasses an examination of research gaps, objectives, and limitations encountered across these studies. In the third chapter, we explain the different phases of our system, detailing each phase's operations, components, and specific functions, algorithms, or formulas employed. Each phase describes highlighting its distinct role in the development of a robust plagiarism checker. In Chapter 4, we provide a detailed description of the dataset utilized in our study, along with an overview of the experiments conducted using the system. We present the outcomes of each experiment, demonstrating the effectiveness of our system in detecting plagiarism and errors across various scenarios and test cases.

In Chapter 5, we look into the performance analysis of our system, where we assess its effectiveness through metric scores and other relevant factors. These scores serve to rank the performance of our system and provide insights into its accuracy. Additionally, we examine the types of similarities and errors detected, illustrating them with appropriate examples. Furthermore, we conduct manual checks to validate the metric scores and ensure the reliability of our findings. In Chapter 6, we explore the social impacts and sustainability of the project, highlighting its broader impacts on society. We address various issues throughout the project that has been looked. The final chapter presents the project's conclusion along with prospects for future development and enhancements. It explains on the project's outcomes and outlines potential avenues for further improvement and innovation.

CHAPTER 2

LITERATURE SURVEY

In this chapter we talk about the extensive research work that we conducted. This section talks about related work, gaps identified and research objectives of this thesis.

2.1 RELATED WORK

In this section, we will talk about various research works that has been taken into consideration for building this system along with its working and limitaions. All the mentioned research papers have built a strong foundation in building each phase of this plagiarism and error detection system respectively.

In paper [1] “Experience Using ”MOSS” by Kevin W. Bowyer, Lawrence O. Hall, we have learnt MOSS is a type of copy-detection algorithm. Given a set of documents, MOSS identifies pairs of documents which are likely to have been copied from each other. There are three properties that MOSS primarily focuses on:

1. White space insensitivity - The algorithm must be able to ignore meaningless syntax like white space. For source-code plagiarism detection, the algorithm must also be unaffected by renaming variables.

2. Noise Suppression - The discovered matches need to be long enough to be significant and interesting - for example, flagging a single matching word would not be a meaningful result.

3. Position independence - The position of the matching segments in each document should not affect the number of matches discovered. This means that reordering large blocks like functions should have no effect on the algorithm.

In paper [3] Winnowing: Local Algorithms for Document Fingerprinting by Saul Schleimer ,Daniel S. Wilkerson, Alex Aiken, we have learnt the Winnowing algorithm is a method used to select fingerprints from hashes of k grams (substrings of a certain length) within a set of documents. The primary goal of this algorithm is to identify substring matches that meet three important criteria:

1. Guarantee Threshold (t): The algorithm ensures that if there is a substring match in the documents that is at least as long as the guarantee threshold (t), it will be detected. In other words, it reliably finds matches of a certain minimum length.

2. Noise Threshold (k): The algorithm is designed to avoid detecting matches that are shorter than the noise threshold (k). This means it filters out insignificant or very short matches that might be considered as noise.

3.Position independence: Coarse-grained permutation of the contents of a document (e.g., scrambling the order of paragraphs) should not affect the set of discovered matches. Adding to a document should not affect the set of matches in the original portion of the new document. Removing part of a document should not affect the set of matches in the portion that remains. In paper [4] Performance evaluation of the VF graph matching algorithm by E. Hopcroft and J. K. Wong, we have learnt representing source code as graphs, where nodes correspond to

code elements , and edges represent relationships between them . This graph-based representation captures the structure and dependencies within the code. Extracting relevant features from the graph considering the edges and nodes is done- Feature Extraction. After matching graphs, incorporating semantic analysis into the matching process, focusing not only on syntactic similarities but also on the semantic meaning of code. This can help identify instances of code reuse or adaptation, even when the code has undergone modifications. But the drawback is that even though it enhances the capabilities of code plagiarism, it introduces additional computational complexity.

In paper [5] Scalable document fingerprinting by Nevin Heintze, we have learnt documentation fingerprinting is a technique used to generate a unique identifier, or fingerprint, for a document based on its content. This process involves extracting key features or characteristics from the document, such as words, phrases, or structural elements, and transforming them into a fixed-length hash value using cryptographic hash functions. Scalable and effective discovery of similarities across huge document collections is made possible by documentation fingerprinting, which uses effective hashing and comparison techniques. This method helps with tasks like recognising duplicate documents, detecting copied content, and maximizing storage capacity by eliminating redundant data. It finds applications in a number of sectors, including data duplication, plagiarism detection, and content identification. The fundamental disparities between natural language text and code pose obstacles for the use of documentation fingerprinting, even though it works well for textual documents. Semantic differences occur when code has unique structures (syntax, programming techniques, algorithms) that documentation fingerprinting is unable to sufficiently capture. Furthermore, context-specific elements like function calls and variable

assignments, as well as minute grammatical changes, are frequently included in code plagiarism and are difficult for document fingerprinting methods to capture accurately. Another problem with granularity is that documentation fingerprinting only looks at documents; it ignores subtler similarities between individual lines or functions of code. There are also efficiency considerations. Code plagiarism detection requires a lot of memory and processing power, which documentation fingerprinting may not be able to meet, especially for large codebases.

In paper [6] A comparison of three popular source code similarity tools for detecting student plagiarism by Ahadi, A. and Mathieson, we have learnt quantitative indicators like precision, recall, and F1-score are computed during the assessment process to gauge how well each tool performs in terms of identifying commonalities between code submissions. To compare the tool performance objectively and determine whether the observed differences are statistically significant, statistical analysis approaches can be utilized. Qualitative input from users—teachers or students, for example—can also be obtained to evaluate how well the tools work in actual classroom environments. By using these techniques, the study hopes to give educators a comprehensive grasp of the advantages and disadvantages of each source code similarity detection tool, empowering them to choose the tool or tools that will work best for them when it comes to identifying plagiarism in student programming assignments. This comprehensive evaluation process ensures that educators have access to reliable and effective tools for maintaining academic integrity in programming education.

In paper [7] Towards a definition of source-code plagiarism by Cosma, G., and Joy, M., we have learnt providing a broader implication of source-code plagiarism for various stakeholders, including educators, students, software developers, and

the wider community are considered. Contributions to the ongoing discourse on source-code plagiarism, offering a nuanced understanding of the phenomenon and its implications.

In paper [10] source Code Plagiarism Detection in Academia with Information Retrieval by Oscar Karnalim, Setia Budi, Hapnes Toba and Mike Joy, we have learnt in addition to being often used in text analysis and natural language processing, n-gram analysis may also be used to source code in order to detect plagiarism. N-grams are consecutive groups of n elements, such as words, letters, or tokens like identifiers and keywords, that are taken from a particular text or source code. Generated n-grams involve tokenizing the code and slicing windows of size n across the token sequence. The frequency distribution of the n-grams within the code is then represented by the profiles that are made using these n-grams. It is possible to find similarities between different code submissions by comparing these profiles and utilizing similarity metrics like cosine or Jaccard similarity. The granularity of the analysis can be greatly affected by the selection of n, or the size of the n-grams. Larger n-grams may miss small similarities, whereas smaller n-grams might record finer details but could result in a higher dimensionality and more noise.

In paper [9] A plagiarism detection system. 12th SIGCSE Technical Symposium on Computer Science Education by Donaldson, J. L., Lancaster, A.-M.and Sposato, P. H. , we have learnt The tokenization method, which divides source code into tokens, determines how effective n-gram analysis is. Inaccuracies in the n-gram representations might emerge from tokenization mistakes or inconsistencies, which can impact the detection outcomes.

2.2 GAPS IDENTIFIED

In this section some of the gaps identified amongst the algorithms that are reffered from the above research papers are mentioned.

2.2.1 Running KarpRabin Greedy String-Tilling

The RKRGST method, while effective in detecting similarities and plagiarism in code submissions, is constrained by its requirement for preprocessing of input code. This means that before applying the method, the code must undergo certain preparatory steps to ensure accurate comparison and analysis. Preprocessing typically involves tasks such as tokenization, which breaks the code into individual elements like keywords, identifiers, and operators, and normalization, which standardizes elements to facilitate comparison despite differences in formatting or naming conventions. Because preprocessing is necessary for the RKRGST method to function properly, it adds an additional step to the analysis process and may introduce complexities or overhead, particularly when dealing with large volumes of code or diverse programming languages.

2.2.2 Vernoi Based Filtering Algorithm

The Vernoi-based filtering algorithm serves as a robust tool for identifying similarities and plagiarism in code submissions. However, it possesses a notable limitation: it cannot effectively detect variable changes in sample input code. Variable changes play a crucial role in code differentiation and can significantly

impact the functionality and logic of a program. While the Vernoi-based filtering algorithm excels at recognizing structural similarities and patterns within code, it may overlook instances where variables are altered, renamed, or repositioned. This limitation arises from the algorithm's focus on structural comparisons rather than semantic analysis. It primarily identifies similarities based on code structure, such as loops, conditional statements, and function definitions, without delving deeply into the specific semantics of variable usage.

2.2.3 VF Graph Matching Algorithm

[3] Performance evaluation in the VF graph matching algorithm method is subject to limitations due to ambiguity in matching and increased complexity. The algorithm's reliance on feature voting can lead to ambiguous matches, especially when dealing with complex or noisy datasets where distinguishing between similar features becomes challenging. As a result, accurately evaluating the algorithm's performance becomes more difficult, as there may be multiple potential matches for a given query. Additionally, the introduction of voting mechanisms adds computational complexity to the algorithm, potentially slowing down the matching process and increasing resource requirements. Overall, while the VF graph matching algorithm offers promising capabilities for graph matching tasks, its performance evaluation is hindered by these limitations related to matching ambiguity and computational complexity.

2.2.4 Tokenization using N-Gram Analysis

Some of the Limitations of tokenization using N-Gram Analysis method are :-

1.Contextual Ambiguity: Tokenization may struggle to accurately capture the contextual nuances of language, leading to ambiguous token boundaries. This ambiguity can result in variations in N-gram representations, affecting the consistency and reliability of the analysis.

2.Language-specific Challenges: Different languages may present unique tokenization challenges due to variations in syntax, morphology, and punctuation rules. Tokenization approaches designed for one language may not perform optimally for others, leading to potential inaccuracies in N-gram analysis.

3. Handling of Special Characters: Tokenization may encounter difficulties when handling special characters, such as punctuation marks, symbols, or emoticons. In some cases, these characters may be incorrectly segmented or omitted, leading to incomplete or distorted N-gram representations.

4. Treatment of Out-of-Vocabulary (OOV) Tokens: N-gram analysis relies on a predefined vocabulary, and tokens outside this vocabulary are often treated as out-of-vocabulary (OOV). Handling OOV tokens can be challenging, as they may be incorrectly assigned or ignored, impacting the overall analysis accuracy.

5. Computational Complexity: Tokenization and subsequent N-gram analysis can be computationally intensive, especially for large datasets or complex text corpora. This computational complexity may limit the scalability and efficiency of N-gram-based approaches, particularly in real-time or resource-constrained environments.

2.2.5 Mesure of Software Similarity - MOSS

MOSS primarily operates as a textual-based integrity checker, relying on the comparison of code snippets at the level of source code text. While this approach is suitable for identifying direct matches and textual similarities, it may overlook instances where code has been structurally modified while retaining similar functionality. This limitation becomes pronounced when dealing with code that has undergone significant restructuring or refactoring, as MOSS may fail to recognize the underlying similarities due to its focus on textual analysis. Additionally, MOSS can encounter ambiguity when assessing the correctness of small-scale code segments. In scenarios where code snippets are relatively brief or lack context, MOSS may struggle to differentiate between genuine similarities and coincidental resemblances. This ambiguity can lead to false positives or false negatives, undermining the accuracy of the plagiarism detection process, particularly in cases where the code fragments are relatively simple or abstract. To address these limitations, users of MOSS should exercise caution and supplement its capabilities with additional tools or manual inspections, especially when dealing with complex codebases or when precision and accuracy are paramount. By leveraging complementary techniques, such as structural analysis or manual review, users can enhance the reliability and effectiveness of code plagiarism detection efforts beyond the scope of MOSS's inherent capabilities.

TABLE 2.1: Literature Survey Summary

Paper Title	**Proposed Solution**	**Limitation**
RunningKarpRabin-Greedy-String-Tilling (RKRGST) method for virtual space	Creating virtual space and efficient information retrieval	This process can only be made after preprocessing of given input code
Experience Using ”MOSS”, to detect cheating on Programming	Helps to compare documents using Document finger printing- ”Winnowing” Process	Can only be used for textual based integrity.
Voronoi Based Filtering algorithm (Graph Matching)	Creating Abstract syntax tree for structural comparison between the codes	Variable change cannot be detected.
Tokenization using N-Gram Analysis	Divide the source code into tokens and comparison is made between them	Inaccuracies in the n gram representation might emerge from tokenization mistakes which can impact the detection outcome .
Performance evaluation of the VF Graph matching algorithm	Find correspondences between nodes in different graphs	Ambiguity in matching and increases complexity.
How MOSS works by Danny Yang	Increase document finger printing by ”Winnowing” process including abundance of data from source code	Ambiguity and compramising correctness in small scale code.

2.3 RESEARCH OBJECTIVES

The main objectives of the system created for code plagiarism and error detection research are as follows:

1. **Fair Evaluation:** By spotting instances of unauthorized collaboration or copying, teachers and instructors can utilize the plagiarism detection tool to guarantee a fair evaluation of students' coding projects.

2. **Effective Code Review:** By incorporating the detector into code review procedures, it is possible to identify duplicate code segments more quickly, which leads to more complete and efficient reviews as well as the encouragement of code reuse and modularity.

3. **Data insights:** The plagiarism detection tool's aggregated data can reveal trends, coding patterns, and prevalent sources of code plagiarism. These insights can be used to improve software development processes, instructional tactics, and curriculum design for education.

4. **Optimize Evalutation Process:** The plagiarism detection tool's aggregated data can reveal trends, coding patterns, and prevalent sources of code plagiarism. These insights can be used to improve software development processes, instructional tactics, and curriculum design for education.

CHAPTER 3

PROPOSED METHODOLOGY

This chapter delineates the distinct phases of our system, encompassing information retrieval, tokenization and parsing, comparator function, and evaluation criteria. It elaborates on each component of the system comprehensively.

Information retrieval - the retrieved code submissions serve as the basis for subsequent stages in the assessment process, such as tokenization, parsing, and similarity comparison, enabling automated evaluation and detection of code similarities or irregularities.

Tokenization and Parsing - The code is tokenized, which is a lexical analysis technique that divides the code into discrete tokens. After that, these tokens are parsed according to pre-established grammar rules.

Comparator - Creates a matrix that serves as a comprehensive dataset containing metric scores for all students, facilitating further analysis and comparison. Moreover, it functions as a pathway for creating clusters based on various categories of uploaded code by students.

Steps involved in our system :-

1. Getting Source code for template of uploaded assignments

2. Convert Source code into tokens by Lexical Analysis.

3. Using Grammar rules, given code is parsed and respective AST is constructed in preorder traversal.

4. All metric scores obtained are stored in a matrix which is used for Clustering.

5. Detailed analysis of insights about submitted assignments if it is plagiarised or contains error.

The work flow diagram of our system is given below:-

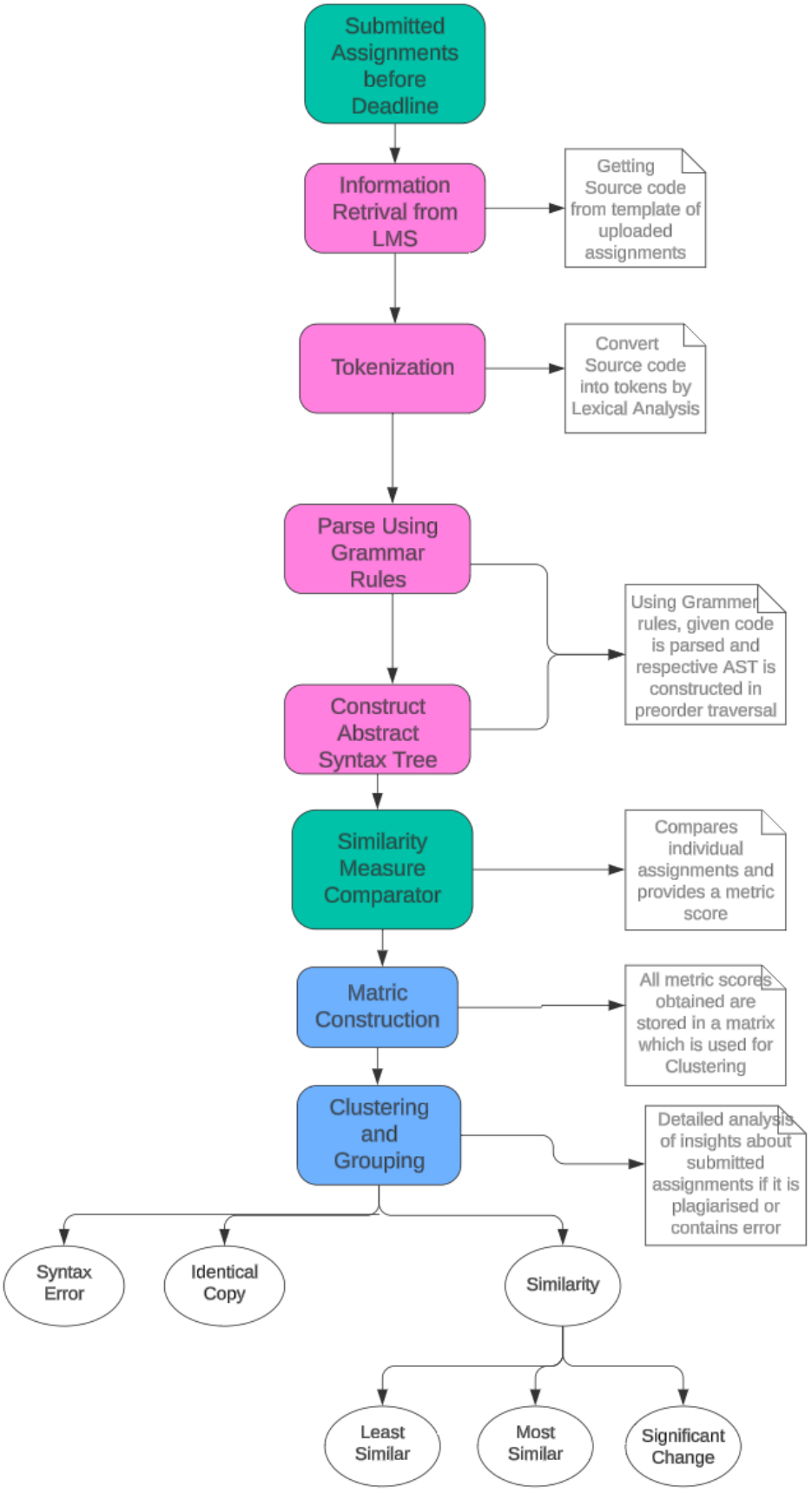


FIGURE 3.1: Flow Diagram

3.1 INFORMATION RETRIEVAL

The initial step in this process is called ”information retrieval”. All the assignments that students submit in the college's online learning system are gathered and hese assignments can come in various formats like PDFs or Word documents. All the information are collected from the submitted folders and are stored in a List. By this process, we obtain a ”N by X by X 3D-matrix”, where N stands for the number of students and X represents the number of questions in an assignment.

By arranging the assignments in this matrix format, with rows representing individual students and columns representing the questions within each assignment, we establish a clear and systematic framework for examination. This structured approach streamlines the subsequent steps of the analysis, facilitating a comprehensive assessment of each student's work.

Moreover, the X by X matrix format enables us to identify patterns and trends across submissions more efficiently. By visually mapping out the distribution of responses to various questions, we gain valuable insights into students' understanding and performance, which can inform instructional strategies and curriculum development.

3.1.1 Template Setting

To ensure consistency in assignment submissions, a standardized template has been developed. This template comprises a front page detailing essential personal information of the student, such as name, registration number, class, and section.

Additionally, it includes exercise-specific details like subject ID and name, faculty in charge, exercise name and number, and submission date.

This front page serves as a universal cover page for all students. It features the college's header and must be attached before every exercise submission. It acts as a uniform format, ensuring that each student's personal details and assignment information are clearly presented and easily identifiable.

Beyond this front page, no additional personal information about the student or the submitted work is permitted. This standardized approach fosters clarity, uniformity, and adherence to submission guidelines, promoting fairness and efficiency in the assessment process.

SRI SIVASUBRAMANIYA NADAR COLLEGE OF ENGINEERING

(AN AUTONOMOUS INSTITUTION, AFFILIATED TO ANNA UNIVERSITY)

Rajiv Gandhi Salai (OMR), Kalavakkam - 603 110

Department of Computer Science and Engineering

(Enter your subject code and name) – UCS 1711 – MOBILE COMPUTING

(Enter Exercise Number) – EX-1

(Enter Exercise Name) – APPLICATION USING GUI COMPONENTS

(Enter Name of Student) – SANJITH G

(Enter Register Number of Student) – 205001092

(Enter Class and Section of Student) – CSE B

(Enter Date of Experiment) – 24/04/2002

FIGURE 3.2: Sample First Page

The second template serves as a structured format for students to submit their completed code, ensuring uniformity and facilitating the retrieval process. This template comprises distinct sections, each meticulously tailored to capture essential details of the submitted work.

Firstly, it includes fields for the exercise question number, providing a clear reference point for the assignment. Additionally, students are prompted to articulate the aim of their code, elucidating the purpose and objectives of their solution.

Moreover, the template mandates the inclusion of the algorithm, which can be presented either as typed text or in the form of a PNG image. This section elucidates the logical steps and procedures employed in the code's development, enhancing clarity and comprehensibility.

The heart of the template lies in the provision for the student-written code itself, where individuals showcase their programming prowess and problem-solving skills. This section serves as the focal point for assessment and evaluation, allowing educators to delve into the intricacies of the code implementation.

Furthermore, students are encouraged to include pseudocode, if necessary, providing a high-level description of the algorithmic logic. This supplementary information aids in understanding the code's underlying principles and enhances transparency in the evaluation process.

Lastly, the template mandates the submission of sample input/output image screenshots, offering a visual representation of the code's functionality. This step provides additional context and validation, demonstrating the code's efficacy in handling various scenarios.

By standardizing the format and headings of each section, the template ensures consistency across submissions, simplifying the information retrieval process. Leveraging the distinct headings, educators can efficiently extract relevant data while disregarding redundant information, expediting the assessment process and fostering a fair and transparent evaluation environment.

THIS IS A SAMPLE EXERCISE PAGE

QUESTION – 1

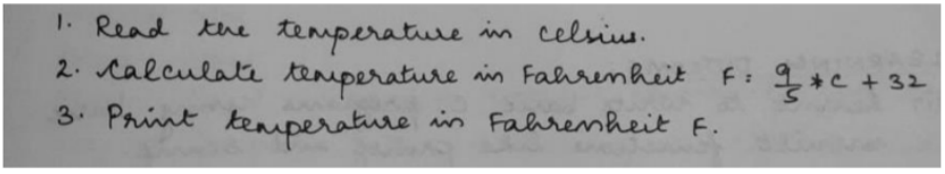
Solve the following problems by implementing in C.

Input the provided Celsius degree and convert it into Fahrenheit.

Aim-

Convert Celsius to Fahrenheit

Algorithm-



Code-

```
#include<stdio.h>
void main()
{ float c,f;
 printf("Enter value for celsius: ");
scanf("%f",&c);
 f=(9*c/5) +32;
printf("\n%f degrees celsius is %f degrees fahrenheit",c,f);
 }
```

FIGURE 3.3: Sample Exercise Page

Sample Input and Output-

```
Input a temperature (in Centigrade): 45
113.000000 degrees Fahrenheit.
```

QUESTION – 2

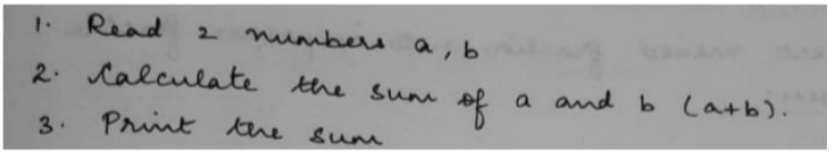
Solve the following problems by implementing in C.

Write a program to add 2 numbers

Aim-

To add 2 numbers

Algorithm-



Code-

```
#include<stdio.h>
void main()
{ int a,b,c;
 printf("Enter values for a and b: ");
```

FIGURE 3.4: Sample Exercise Page cont-

3.1.2 Preprocess Code

For the product being developed, information from student-submitted code in the LMS portal is retrieved. Folders containing code submissions in Word documents

```
[
    [
     #include<stdio.h>
     int main(){
    float fahrenheit, celsius;
    printf("Enter Fahrenheit: ");
    scanf("%f",&fahrenheit);
    celsius = (fahrenheit - 32)*5/9;
    printf("Celsius: %f ", celsius);
    return 0;
}
    ] ,

    [
     #include <conio.h>
     #include <stdio.h>
     #include <stdlib.h>
     float fahrenheit_to_Celsius(float fahrenheit)
     {
         return ((fahrenheit - 32.0) * 5.0 / 9.0);
     }

     int main()
     {
         float f = 22;
         printf("Temperature in Celsius : %0.2f",fahrenheit_to_Celsius(f));
         return 0;
         }
    ]
]
```

FIGURE 3.5: Sample of 2 codes stored in matrix after preprocessing

or PDF format are accessed, and the code within is copied and pasted into individual matrices for the entire class. These stored values undergo preprocessing before being forwarded to the subsequent tokenization step.

From the above diagram we can see that the code of student 1 and 2 are retrived from their respected LMS folder and are stored in 2 different arrays.

3.2 TOKENIZATION AND PARSING

Upon completing the retrieval of code from students' submitted assignments and storing it in a list, the tool proceeds to execute tokenization. This pivotal process entails the classification of lexemes within the code into distinct categories such as Identifiers, Keywords, Constants, String literals, Character literals, Arithmetic operators, Logical operators, Relational operators, Punctuators, or Assignment operators. Referred to as lexical analysis, this step categorizes individual tokens without LOOKING into the program's underlying structure.

Following tokenization, the tool embarks on parsing, utilizing a set of well-established grammar rules. This parsing process involves the systematic analysis of the code's structure according to predetermined language rules, culminating in the generation of individual metric scores. To ensure the accuracy of these scores, a comprehensive set of 63 grammar rules, encompassing the entirety of the C language, has been meticulously pre-registered and employed.

Central to this parsing process is the utilization of a LALR parser, known for its robust capability in handling a diverse array of grammar constructs. This parser stands as a cornerstone in the tool's architecture, quickly navigating through complex code structures with precision and efficiency. Some examples of the grammar rules include handling variable declarations, control flow statements, function definitions, and data types, among others.

By leveraging this sophisticated parsing mechanism, the tool is equipped to provide detailed insights into the structure and complexity of the code submissions. These insights, derived from the meticulous analysis of the code's

```
def p_statement(p):
    '''
    statement   : labeled_statement
                | compound_statement
                | expression_statement
                | selection_statement
                | iteration_statement
                | jump_statement
                | declaration
    '''
    l.append(52)

def p_labeled_statement(p):
    '''
    labeled_statement   : IDENTIFIER COLON statement
                        | CASE constant_expression COLON statement
                        | DEFAULT COLON statement
    '''
    l.append(53)

def p_compound_statement(p):
    '''
    compound_statement  : LBRACE RBRACE
                        | LBRACE statement_list RBRACE
                        | LBRACE declaration_list RBRACE
                        | LBRACE declaration_list statement_list RBRACE
    '''
    l.append(54)
```

FIGURE 3.6: Sample Grammar Rules

syntactic structure, serve as valuable metrics for evaluating the quality and proficiency of students' programming efforts.

63 Grammar rules that cover the whole C language has been pre registered and used inorder to obtain a precise metric score. These scores are then preserved, and the resulting output is instrumental in the creation of an Abstract Syntax Tree (AST). Subsequently, this AST is transmitted to the comparator, where it accumulates all the gathered metric scores from diverse uploaded documents and organizes them into a matrix.

Traversing the created list via Depth-First Search (DFS) facilitates the

```
def p_selection_statement(p):
    '''
    selection_statement : IF LPARENTHESIS expression RPARENTHESIS statement
                        | IF LPARENTHESIS expression RPARENTHESIS statement ELSE statement
                        | SWITCH LPARENTHESIS expression RPARENTHESIS statement
    '''
    l.append(58)

def p_iteration_statement(p):
    '''
    iteration_statement : WHILE LPARENTHESIS expression RPARENTHESIS statement
                        | DO statement WHILE LPARENTHESIS expression RPARENTHESIS SEMI_COLON
                        | FOR LPARENTHESIS expression_statement expression_statement RPARENTHESIS statement
                        | FOR LPARENTHESIS expression_statement expression_statement expression RPARENTHESIS statement
                        | FOR LPARENTHESIS declaration expression_statement expression RPARENTHESIS statement
                        | FOR LPARENTHESIS declaration expression_statement RPARENTHESIS statement
    '''
    l.append(59)
```

FIGURE 3.7: Sample Grammar Rules cont-

comparison of individual code submissions, a process known as ”similarity checking”. Each code submission is compared against the entire class, generating a similarity score. This comparison is conducted between each individual code and every other student’s code in the class, with similarity score values stored in an N x N matrix. Notably, the diagonal elements of this matrix are set to one, as they represent comparisons of a student’s code against itself.

The following diagram shows the submission of code by 2 individual students and the obtained List from both of them after comparing with the entire class. The given 2 codes are compared with each other as well as the entire class. Each time of comparison a score is generated and stored in a List.

```c
void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void bubbleSort(int arr[][2], int n)
{
    int i, j;
    int swapped;
    for (i = 0; i < n - 1; i++)
    {
        swapped = 0;
        for (j = 0; j < n - i - 1; j++)
        {
            if (arr[j][0] > arr[j + 1][0])
            {
                swap(&arr[j][0], &arr[j + 1][0]);
                swap(&arr[j][1], &arr[j + 1][1]);
                swapped = 1;
            }
        }
        if (swapped == 0)
            break;
    }
}
int main()
{
    char text[600],lineArray[10][60],line[60];
    int lineCount,wordCount,i,j,k,blankFlag,countArray[10][2];
    printf("Please input a paragraph containing text:");
    scanf("%599[^\n]",text);
    printf("\nInput text was %s\n",text);
    for (i=0,j=0,lineCount=0;text[i]!='\0';i++)
    {
        if (text[i] != '.')
            line[j++] = text[i];
        else
        {
            line[j]='\0';
            for (j=0,k=0,wordCount=0,blankFlag=1;line[j]!='\0';j++)
            {
                if (blankFlag == 1)
                    if (line[j] == ' ')
                        continue;
```

FIGURE 3.8: Student 1 code

```
        else
        {
            line[j]='\0';
            for (j=0,k=0,wordCount=0,blankFlag=1;line[j]!='\0';j++)
            {
                if (blankFlag == 1)
                    if (line[j] == ' ')
                        continue;
                    else
                        blankFlag = 0;
                if (line[j] == ' ')
                {
                    wordCount++;
                }
                lineArray[lineCount][k] = line[j];
                k++;
            }
            lineArray[lineCount][k]='\0';
            wordCount++;
            countArray[lineCount][0]=wordCount;
            countArray[lineCount][1]=lineCount;
            j=0;
            lineCount++;
        }
    }
    countArray[lineCount][0]=wordCount;
    countArray[lineCount][1]=lineCount;
    lineArray[lineCount][k]='\0';
    bubbleSort(countArray, lineCount);
    int maxWordCount = countArray[lineCount-1][0];
    for (i=0;i<lineCount;i++)
        if (countArray[i][0] == maxWordCount)
        {
            int idx = countArray[i][1];
            printf("The first line with maximum number of words (%d) is:\n%s\n",maxWordCount,lineArray[idx]);
            break;
        }
    return 0;
}
```

FIGURE 3.9: Student 1 code cont-

```c
int length (char str[])
{
    int l=0,i=0;
    while (str[i]!='\0')
    {
        l++;
        i++;
    }
    return l;
}
void main ()
{
    int i=0,j=0,k=0,count_sentences=0,c=0,count=0,c_max=0;
    char str[150];
    printf ("Enter a string: ");
    scanf ("%[^~]", str);
    int len=length(str);
    for (i=0;i<len;i++)
    {
        if (str[i]=='.')
        count_sentences++;
    }
    int arr[count_sentences];
    for (i=0;i<len;i++)
    {
        if (str[i]=='.')
        {
            if (c==0)
        arr[c++]=count+1;
        else
        arr[c++]=count;
        count=0;
        continue;
        }
        if (str[i]==' ')
        count++;
    }
    for (i=0;i<c;i++)
    {
        if (arr[i]>c_max)
        {
            c_max=arr[i];
            k=i;
        }
    }
```

FIGURE 3.10: Student 2 code

```
    printf ("Sentence with maximum number of words:");
    count=0;
    for (i=0;i<len;i++)
    {
        if (k==0)
        {
            while (str[i]!='.')
            {
                printf ("%c", str[i]);
                i++;
            }
            printf (".");
            break;
        }
        else
        {
            if (str[i]=='.')
            {
                if (count==k-1)
                {
                    i=i+1;
                    while (str[i]!='.')
                    {
                        printf ("%c", str[i]);
                        i++;
                    }
                    printf (".");
                }
                count++;
            }
        }
    }
    printf ("\nNumber of words: %d ", c_max);
}
```

FIGURE 3.11: Student 2 code cont-

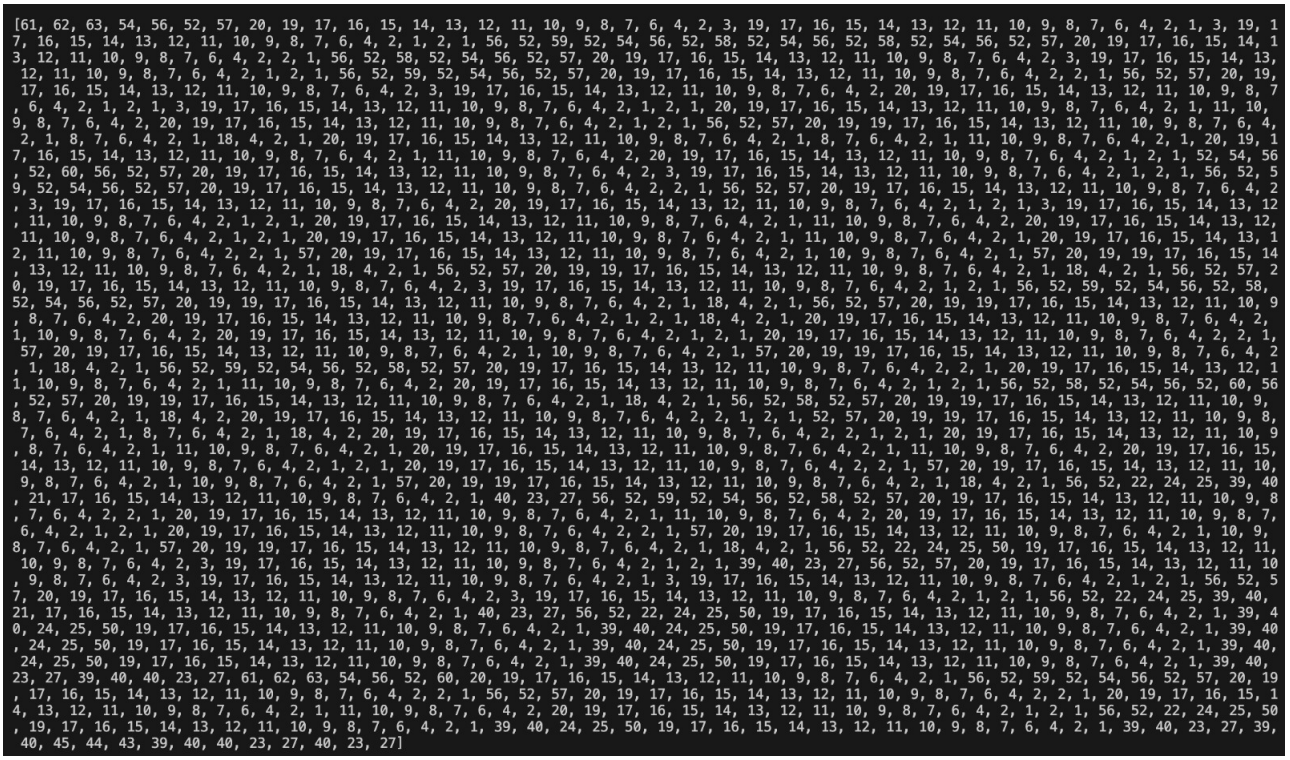


FIGURE 3.12: DFS of AST generated for student 1

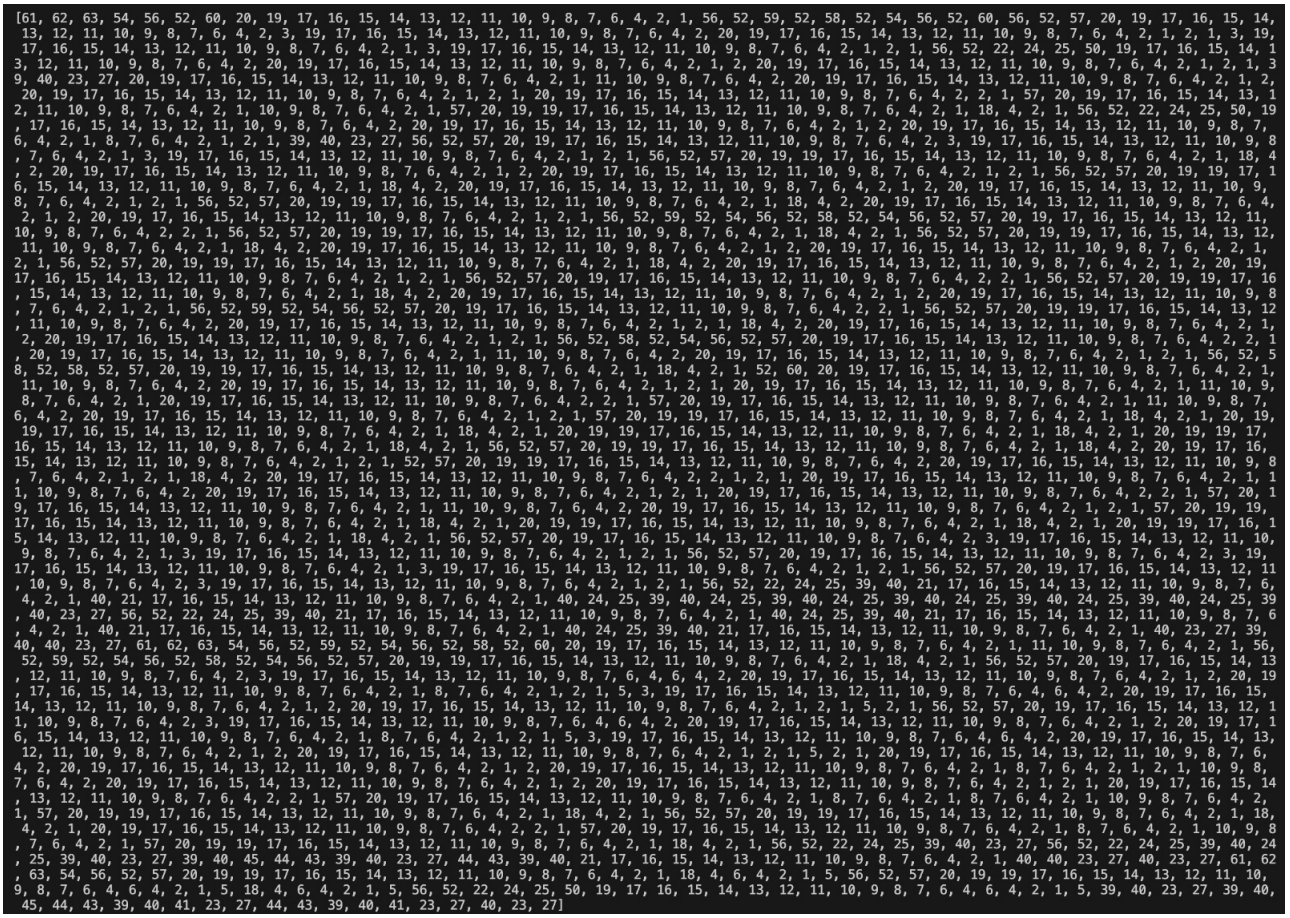


FIGURE 3.13: DFS of AST generated for student 2

3.3 COMPARATOR

The comparator is built using difflib. Sequence Matcher as it is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. Sequence Matcher tries to compute a ”human-friendly diff” between two sequences. Unlike e.g. UNIX(tm) diff, the fundamental notion is the longest *contiguous* junk-free matching subsequence.

3.3.1 Sequence Matcher

The ‘SequenceMatcher‘ class in Python’s ‘difflib‘ module serves the purpose of comparing pairs of sequences, such as strings or lists, and identifying similarities and differences between them. Its primary function is to find the longest contiguous matching subsequences between two sequences, while also handling cases where elements may have been inserted, deleted, or substituted between the sequences.

The ‘SequenceMatcher‘ works by employing an algorithm that iterates over the elements of the two sequences and identifies matching blocks based on the longest common subsequence approach. Here’s a simplified explanation of how it works:

1. Initialization: When you create a ‘SequenceMatcher‘ object with two sequences, it initializes internal data structures for comparison.

2. Matching Algorithm: The algorithm then compares the elements of the two sequences to identify matching blocks. It starts by identifying the longest contiguous sequence of elements that are common to both sequences.

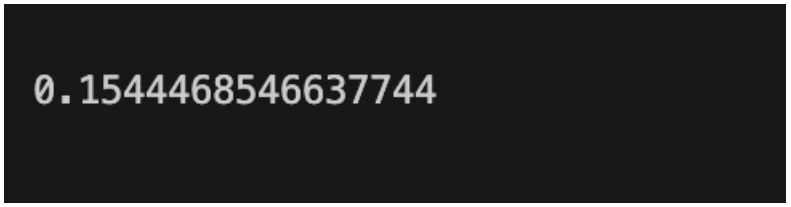


FIGURE 3.14: Similarity score between Student 1 and 2

3. Refinement: After finding the initial matching block, the algorithm refines its search to find additional matches while skipping over elements that have already been matched. It continues this process to identify all possible matching blocks.

4. Scoring: Based on the identified matching blocks, the ‘SequenceMatcher‘ computes a similarity ratio that represents the degree of similarity between the sequences. This ratio is often used to determine how closely the sequences resemble each other.

The final matric value obtained after comparing the two student code shown previously is:

The range of matric score ranges from 0 to 1, where 1 is the highest possible value for plagiarism. From the given two code snippets, the obtained value of 0.15 lets us know that, the quantity of plagiarism is very minimal.

3.3.2 Ratio () Function

The ‘ratio()‘ function in Python’s ‘difflib.SequenceMatcher‘ class computes a similarity ratio between two sequences based on their matching blocks. This ratio provides a measure of how similar the sequences are to each other, with higher values indicating greater similarity.

Working:- 1. Identification of Matching Blocks: Before computing the ratio, the ‘SequenceMatcher‘ object identifies all the matching blocks between the two sequences. These matching blocks represent subsequences that are common to both sequences.

2. Calculation of Similarity Ratio: Once the matching blocks are identified, the ‘ratio()‘ function computes a similarity ratio using the formula:

$$\text{ratio} = \frac{2 \times \text{number of matching elements}}{\text{length of sequence A} + \text{length of sequence B}}$$

The numerator $2 \times$ number of matching elements accounts for both the elements present in the matching blocks from both sequences. The denominator represents the total length of both sequences.

3. Normalization: The ratio is normalized to fall within the range of 0 to 1, where 0 indicates no similarity between the sequences, and 1 indicates complete similarity.

4. Interpretation: Higher values of the similarity ratio indicate that a larger proportion of the sequences are similar to each other, while lower values indicate greater dissimilarity.

This function is mainly used for:-

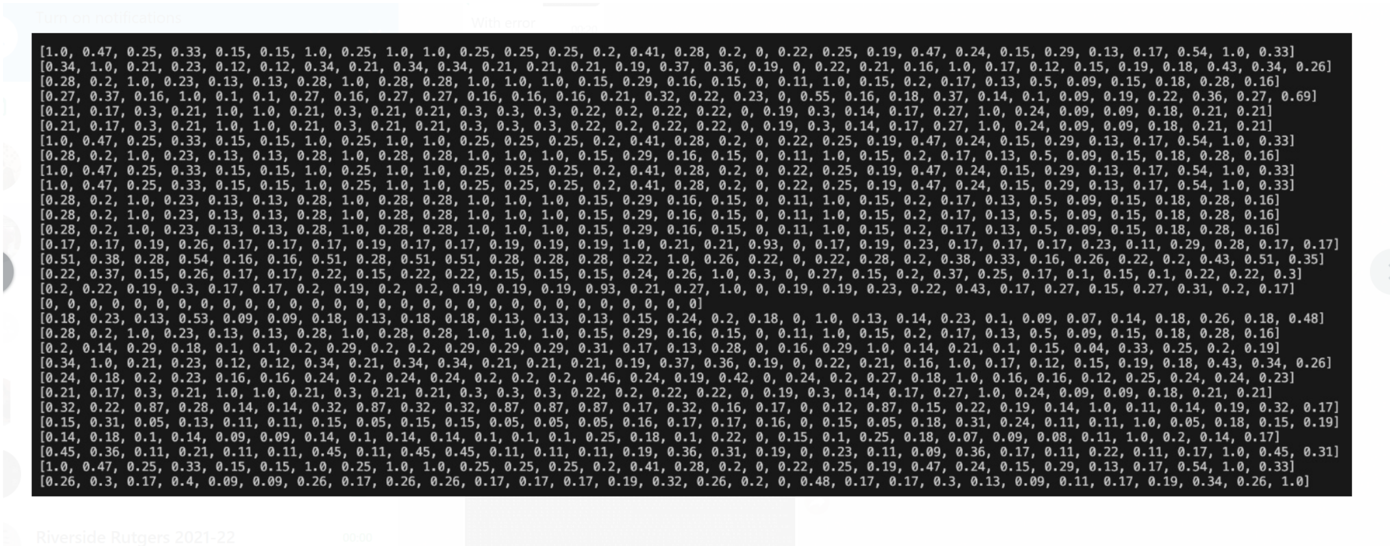
- Text comparison: Assessing the similarity between two strings or documents.

- Version control: Determining the degree of difference between two versions of a file.

- Data analysis: Comparing sequences of data to identify common patterns or changes.

```
[1.0, 0.38, 0.12, 0.12, 0.29, 0.12, 0.12, 0.12, 0.99, 1.0, 0.12, 0.12, 0.12, 0.32, 0.36, 0.21, 0.32, 0.49, 0.4, 0.12, 0.32, 0.98, 0.49, 0.12, 0.12, 0.18, 0.18, 0.38, 0.99, 0.54]
[0.39, 1.0, 0.31, 0.31, 0.36, 0.31, 0.31, 0.31, 0.35, 0.39, 0.31, 0.31, 0.31, 0.52, 0.2, 0.29, 0.52, 0.45, 0.35, 0.31, 0.35, 0.34, 0.45, 0.31, 0.31, 0.15, 0.15, 0.22, 0.37, 0.25]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.32, 0.42, 0.33, 0.33, 1.0, 0.33, 0.34, 0.33, 0.32, 0.32, 0.33, 0.33, 0.34, 0.44, 0.21, 0.32, 0.44, 0.36, 0.31, 0.33, 0.35, 0.31, 0.36, 0.33, 0.33, 0.15, 0.15, 0.19, 0.31, 0.19]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.17, 0.3, 0.98, 0.98, 0.3, 0.98, 1.0, 0.98, 0.17, 0.17, 0.98, 0.98, 1.0, 0.36, 0.29, 0.13, 0.36, 0.3, 0.13, 0.98, 0.61, 0.17, 0.3, 0.98, 0.98, 0.09, 0.09, 0.16, 0.17, 0.19]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.99, 0.34, 0.12, 0.12, 0.32, 0.12, 0.12, 0.12, 1.0, 0.99, 0.12, 0.12, 0.12, 0.31, 0.36, 0.22, 0.31, 0.48, 0.38, 0.12, 0.32, 0.99, 0.48, 0.12, 0.12, 0.16, 0.16, 0.38, 0.98, 0.54]
[1.0, 0.38, 0.12, 0.12, 0.29, 0.12, 0.12, 0.12, 0.99, 1.0, 0.12, 0.12, 0.12, 0.32, 0.36, 0.21, 0.32, 0.49, 0.4, 0.12, 0.32, 0.98, 0.49, 0.12, 0.12, 0.18, 0.18, 0.38, 0.99, 0.54]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.17, 0.3, 0.98, 0.98, 0.3, 0.98, 1.0, 0.98, 0.17, 0.17, 0.98, 0.98, 1.0, 0.36, 0.29, 0.13, 0.36, 0.3, 0.13, 0.98, 0.61, 0.17, 0.3, 0.98, 0.98, 0.09, 0.09, 0.16, 0.17, 0.19]
[0.29, 0.46, 0.34, 0.34, 0.46, 0.34, 0.35, 0.34, 0.28, 0.29, 0.34, 0.34, 0.35, 1.0, 0.24, 0.32, 1.0, 0.55, 0.25, 0.34, 0.42, 0.27, 0.55, 0.34, 0.34, 0.12, 0.12, 0.21, 0.28, 0.24]
[0.48, 0.26, 0.21, 0.21, 0.15, 0.21, 0.18, 0.21, 0.5, 0.48, 0.21, 0.21, 0.18, 0.17, 1.0, 0.2, 0.17, 0.13, 0.23, 0.21, 0.24, 0.51, 0.13, 0.21, 0.21, 0.11, 0.11, 0.19, 0.49, 0.19]
[0.14, 0.12, 0.1, 0.1, 0.14, 0.1, 0.11, 0.1, 0.14, 0.14, 0.1, 0.1, 0.11, 0.08, 0.19, 1.0, 0.08, 0.08, 0.25, 0.1, 0.07, 0.14, 0.08, 0.1, 0.1, 0.17, 0.17, 0.06, 0.14, 0.08]
[0.29, 0.46, 0.34, 0.34, 0.46, 0.34, 0.35, 0.34, 0.28, 0.29, 0.34, 0.34, 0.35, 1.0, 0.24, 0.32, 1.0, 0.55, 0.25, 0.34, 0.42, 0.27, 0.55, 0.34, 0.34, 0.12, 0.12, 0.21, 0.28, 0.24]
[0.48, 0.46, 0.29, 0.29, 0.4, 0.29, 0.3, 0.29, 0.48, 0.48, 0.29, 0.29, 0.3, 0.53, 0.21, 0.26, 0.53, 1.0, 0.22, 0.29, 0.37, 0.47, 1.0, 0.29, 0.29, 0.07, 0.07, 0.36, 0.46, 0.35]
[0.38, 0.18, 0.11, 0.11, 0.2, 0.11, 0.11, 0.11, 0.37, 0.38, 0.11, 0.11, 0.11, 0.13, 0.26, 0.3, 0.13, 0.2, 1.0, 0.11, 0.15, 0.36, 0.2, 0.11, 0.11, 0.2, 0.2, 0.16, 0.38, 0.29]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.19, 0.32, 0.45, 0.45, 0.29, 0.45, 0.46, 0.45, 0.19, 0.19, 0.45, 0.45, 0.46, 0.33, 0.22, 0.15, 0.33, 0.34, 0.16, 0.45, 1.0, 0.19, 0.34, 0.45, 0.45, 0.1, 0.1, 0.2, 0.19, 0.22]
[0.98, 0.33, 0.12, 0.12, 0.31, 0.12, 0.12, 0.12, 0.99, 0.98, 0.12, 0.12, 0.12, 0.29, 0.35, 0.26, 0.29, 0.47, 0.38, 0.12, 0.32, 1.0, 0.47, 0.12, 0.12, 0.16, 0.16, 0.28, 0.99, 0.47]
[0.48, 0.46, 0.29, 0.29, 0.4, 0.29, 0.3, 0.29, 0.48, 0.48, 0.29, 0.29, 0.3, 0.53, 0.21, 0.26, 0.53, 1.0, 0.22, 0.29, 0.37, 0.47, 1.0, 0.29, 0.29, 0.07, 0.07, 0.36, 0.46, 0.35]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.16, 0.29, 1.0, 1.0, 0.29, 1.0, 0.98, 1.0, 0.17, 0.16, 1.0, 1.0, 0.98, 0.35, 0.28, 0.13, 0.35, 0.29, 0.13, 1.0, 0.55, 0.16, 0.29, 1.0, 1.0, 0.08, 0.08, 0.16, 0.16, 0.18]
[0.12, 0.14, 0.17, 0.17, 0.1, 0.17, 0.12, 0.17, 0.12, 0.12, 0.17, 0.17, 0.12, 0.08, 0.19, 0.16, 0.08, 0.08, 0.18, 0.17, 0.12, 0.12, 0.08, 0.17, 0.17, 1.0, 1.0, 0.1, 0.12, 0.1]
[0.12, 0.14, 0.17, 0.17, 0.1, 0.17, 0.12, 0.17, 0.12, 0.12, 0.17, 0.17, 0.12, 0.08, 0.19, 0.16, 0.08, 0.08, 0.18, 0.17, 0.12, 0.12, 0.08, 0.17, 0.17, 1.0, 1.0, 0.1, 0.12, 0.1]
[0.39, 0.21, 0.12, 0.12, 0.19, 0.12, 0.12, 0.12, 0.39, 0.39, 0.12, 0.12, 0.12, 0.21, 0.46, 0.15, 0.21, 0.32, 0.2, 0.12, 0.26, 0.3, 0.32, 0.12, 0.12, 0.09, 0.09, 1.0, 0.3, 0.45]
[0.99, 0.37, 0.11, 0.11, 0.28, 0.11, 0.12, 0.11, 0.98, 0.99, 0.11, 0.11, 0.12, 0.31, 0.35, 0.27, 0.31, 0.48, 0.4, 0.11, 0.32, 0.99, 0.48, 0.11, 0.11, 0.18, 0.18, 0.29, 1.0, 0.47]
[0.55, 0.24, 0.14, 0.14, 0.19, 0.14, 0.14, 0.14, 0.55, 0.55, 0.14, 0.14, 0.14, 0.23, 0.46, 0.18, 0.23, 0.35, 0.32, 0.14, 0.31, 0.48, 0.35, 0.14, 0.14, 0.12, 0.12, 0.45, 0.47, 1.0]
```

FIGURE 3.15: Similarity Score Matrix



```
[1.0, 0.47, 0.25, 0.33, 0.15, 0.15, 1.0, 0.25, 1.0, 1.0, 0.25, 0.25, 0.25, 0.2, 0.41, 0.28, 0.2, 0, 0.22, 0.25, 0.19, 0.47, 0.24, 0.15, 0.29, 0.13, 0.17, 0.54, 1.0, 0.33]
[0.34, 1.0, 0.21, 0.23, 0.12, 0.12, 0.34, 0.21, 0.34, 0.34, 0.21, 0.21, 0.21, 0.19, 0.37, 0.36, 0.19, 0, 0.22, 0.21, 0.16, 1.0, 0.17, 0.12, 0.15, 0.19, 0.18, 0.43, 0.34, 0.26]
[0.28, 0.2, 1.0, 0.23, 0.13, 0.13, 0.28, 1.0, 0.28, 0.28, 1.0, 1.0, 1.0, 0.15, 0.29, 0.16, 0.15, 0, 0.11, 1.0, 0.15, 0.2, 0.17, 0.13, 0.5, 0.09, 0.15, 0.18, 0.28, 0.16]
[0.27, 0.37, 0.16, 1.0, 0.1, 0.1, 0.27, 0.16, 0.27, 0.27, 0.16, 0.16, 0.16, 0.21, 0.32, 0.22, 0.23, 0, 0.55, 0.16, 0.18, 0.37, 0.14, 0.1, 0.09, 0.19, 0.22, 0.36, 0.27, 0.69]
[0.21, 0.17, 0.3, 0.21, 1.0, 1.0, 0.21, 0.3, 0.21, 0.21, 0.3, 0.3, 0.3, 0.22, 0.2, 0.22, 0.22, 0, 0.19, 0.3, 0.14, 0.17, 0.27, 1.0, 0.24, 0.09, 0.09, 0.18, 0.21, 0.21]
[0.21, 0.17, 0.3, 0.21, 1.0, 1.0, 0.21, 0.3, 0.21, 0.21, 0.3, 0.3, 0.3, 0.22, 0.2, 0.22, 0.22, 0, 0.19, 0.3, 0.14, 0.17, 0.27, 1.0, 0.24, 0.09, 0.09, 0.18, 0.21, 0.21]
[1.0, 0.47, 0.25, 0.33, 0.15, 0.15, 1.0, 0.25, 1.0, 1.0, 0.25, 0.25, 0.25, 0.2, 0.41, 0.28, 0.2, 0, 0.22, 0.25, 0.19, 0.47, 0.24, 0.15, 0.29, 0.13, 0.17, 0.54, 1.0, 0.33]
[0.28, 0.2, 1.0, 0.23, 0.13, 0.13, 0.28, 1.0, 0.28, 0.28, 1.0, 1.0, 1.0, 0.15, 0.29, 0.16, 0.15, 0, 0.11, 1.0, 0.15, 0.2, 0.17, 0.13, 0.5, 0.09, 0.15, 0.18, 0.28, 0.16]
[1.0, 0.47, 0.25, 0.33, 0.15, 0.15, 1.0, 0.25, 1.0, 1.0, 0.25, 0.25, 0.25, 0.2, 0.41, 0.28, 0.2, 0, 0.22, 0.25, 0.19, 0.47, 0.24, 0.15, 0.29, 0.13, 0.17, 0.54, 1.0, 0.33]
[1.0, 0.47, 0.25, 0.33, 0.15, 0.15, 1.0, 0.25, 1.0, 1.0, 0.25, 0.25, 0.25, 0.2, 0.41, 0.28, 0.2, 0, 0.22, 0.25, 0.19, 0.47, 0.24, 0.15, 0.29, 0.13, 0.17, 0.54, 1.0, 0.33]
[0.28, 0.2, 1.0, 0.23, 0.13, 0.13, 0.28, 1.0, 0.28, 0.28, 1.0, 1.0, 1.0, 0.15, 0.29, 0.16, 0.15, 0, 0.11, 1.0, 0.15, 0.2, 0.17, 0.13, 0.5, 0.09, 0.15, 0.18, 0.28, 0.16]
[0.28, 0.2, 1.0, 0.23, 0.13, 0.13, 0.28, 1.0, 0.28, 0.28, 1.0, 1.0, 1.0, 0.15, 0.29, 0.16, 0.15, 0, 0.11, 1.0, 0.15, 0.2, 0.17, 0.13, 0.5, 0.09, 0.15, 0.18, 0.28, 0.16]
[0.28, 0.2, 1.0, 0.23, 0.13, 0.13, 0.28, 1.0, 0.28, 0.28, 1.0, 1.0, 1.0, 0.15, 0.29, 0.16, 0.15, 0, 0.11, 1.0, 0.15, 0.2, 0.17, 0.13, 0.5, 0.09, 0.15, 0.18, 0.28, 0.16]
[0.17, 0.17, 0.19, 0.26, 0.17, 0.17, 0.17, 0.19, 0.17, 0.17, 0.19, 0.19, 0.19, 1.0, 0.21, 0.21, 0.93, 0, 0.17, 0.19, 0.23, 0.17, 0.17, 0.17, 0.23, 0.11, 0.29, 0.28, 0.17, 0.17]
[0.51, 0.38, 0.28, 0.54, 0.16, 0.16, 0.51, 0.28, 0.51, 0.51, 0.28, 0.28, 0.28, 0.22, 1.0, 0.26, 0.22, 0, 0.22, 0.28, 0.2, 0.38, 0.33, 0.16, 0.26, 0.22, 0.2, 0.43, 0.51, 0.35]
[0.22, 0.37, 0.15, 0.26, 0.17, 0.17, 0.22, 0.15, 0.22, 0.22, 0.15, 0.15, 0.15, 0.24, 0.26, 1.0, 0.3, 0, 0.27, 0.15, 0.2, 0.37, 0.25, 0.17, 0.1, 0.15, 0.1, 0.22, 0.22, 0.3]
[0.2, 0.22, 0.19, 0.3, 0.17, 0.17, 0.2, 0.19, 0.2, 0.2, 0.19, 0.19, 0.19, 0.93, 0.21, 0.27, 1.0, 0, 0.19, 0.19, 0.23, 0.22, 0.43, 0.17, 0.27, 0.15, 0.27, 0.31, 0.2, 0.17]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0.18, 0.23, 0.13, 0.53, 0.09, 0.09, 0.18, 0.13, 0.18, 0.18, 0.13, 0.13, 0.13, 0.15, 0.24, 0.2, 0.18, 0, 1.0, 0.13, 0.14, 0.23, 0.1, 0.09, 0.07, 0.14, 0.18, 0.26, 0.18, 0.48]
[0.28, 0.2, 1.0, 0.23, 0.13, 0.13, 0.28, 1.0, 0.28, 0.28, 1.0, 1.0, 1.0, 0.15, 0.29, 0.16, 0.15, 0, 0.11, 1.0, 0.15, 0.2, 0.17, 0.13, 0.5, 0.09, 0.15, 0.18, 0.28, 0.16]
[0.2, 0.14, 0.29, 0.18, 0.1, 0.1, 0.2, 0.29, 0.2, 0.2, 0.29, 0.29, 0.29, 0.31, 0.17, 0.13, 0.28, 0, 0.16, 0.29, 1.0, 0.14, 0.21, 0.1, 0.15, 0.04, 0.33, 0.25, 0.2, 0.19]
[0.34, 1.0, 0.21, 0.23, 0.12, 0.12, 0.34, 0.21, 0.34, 0.34, 0.21, 0.21, 0.21, 0.19, 0.37, 0.36, 0.19, 0, 0.22, 0.21, 0.16, 1.0, 0.17, 0.12, 0.15, 0.19, 0.18, 0.43, 0.34, 0.26]
[0.24, 0.18, 0.2, 0.23, 0.16, 0.16, 0.24, 0.2, 0.24, 0.24, 0.2, 0.2, 0.2, 0.46, 0.24, 0.19, 0.42, 0, 0.24, 0.2, 0.27, 0.18, 1.0, 0.16, 0.16, 0.12, 0.25, 0.24, 0.24, 0.23]
[0.21, 0.17, 0.3, 0.21, 1.0, 1.0, 0.21, 0.3, 0.21, 0.21, 0.3, 0.3, 0.3, 0.22, 0.2, 0.22, 0.22, 0, 0.19, 0.3, 0.14, 0.17, 0.27, 1.0, 0.24, 0.09, 0.09, 0.18, 0.21, 0.21]
[0.32, 0.22, 0.87, 0.28, 0.14, 0.14, 0.32, 0.87, 0.32, 0.32, 0.87, 0.87, 0.87, 0.17, 0.32, 0.16, 0.17, 0, 0.12, 0.87, 0.15, 0.22, 0.19, 0.14, 1.0, 0.11, 0.14, 0.19, 0.32, 0.17]
[0.15, 0.31, 0.05, 0.13, 0.11, 0.11, 0.15, 0.05, 0.15, 0.15, 0.05, 0.05, 0.05, 0.16, 0.17, 0.17, 0.16, 0, 0.15, 0.05, 0.18, 0.31, 0.24, 0.11, 0.11, 1.0, 0.05, 0.18, 0.15, 0.19]
[0.14, 0.18, 0.1, 0.14, 0.09, 0.09, 0.14, 0.1, 0.14, 0.14, 0.1, 0.1, 0.1, 0.25, 0.18, 0.1, 0.22, 0, 0.15, 0.1, 0.25, 0.18, 0.07, 0.09, 0.08, 0.11, 1.0, 0.2, 0.14, 0.17]
[0.45, 0.36, 0.11, 0.21, 0.11, 0.11, 0.45, 0.11, 0.45, 0.45, 0.11, 0.11, 0.11, 0.19, 0.36, 0.31, 0.19, 0, 0.23, 0.11, 0.09, 0.36, 0.17, 0.11, 0.22, 0.11, 0.17, 1.0, 0.45, 0.31]
[1.0, 0.47, 0.25, 0.33, 0.15, 0.15, 1.0, 0.25, 1.0, 1.0, 0.25, 0.25, 0.25, 0.2, 0.41, 0.28, 0.2, 0, 0.22, 0.25, 0.19, 0.47, 0.24, 0.15, 0.29, 0.13, 0.17, 0.54, 1.0, 0.33]
[0.26, 0.3, 0.17, 0.4, 0.09, 0.09, 0.26, 0.17, 0.26, 0.26, 0.17, 0.17, 0.17, 0.19, 0.32, 0.26, 0.2, 0, 0.48, 0.17, 0.17, 0.3, 0.13, 0.09, 0.11, 0.17, 0.19, 0.34, 0.26, 1.0]
```

FIGURE 3.16: Similarity Score Matrix containing an Error program

This function is employed to determine the similarity score of each code submission by comparing them with a list of scores derived from comparing the codes with the entire class.

From the NxN matrix generated earlier, each row will contain similarity score

values. These individual scores are averaged to produce N distinct values. This average represents the level of plagiarism detected, where the lowest score signifies the most unique code, and the highest score indicates the most plagiarized code. A threshold is then established based on these values. Scores close to or lower than the minimum are classified as highly unique or exhibiting significantly less plagiarism, while higher scores are categorized as highly plagiarized or mostly plagiarized.

From the obtained score above, we can classify the level of plagiarism into 4 different categories :-

1. **Syntax Error:** Code errors done by the student, which can include any syntax error like missing a colon, badly intended code and missing of header files.
2. **Identical Copy:** When 2 students have the exact similar code.

Types of Similarity With respect to the threshold value calculated using K-Means clustering algorithm, which generates 3 cluster centres for the below 3 categories. Hence the extent of similarity is relative and is based on the produced threshold value.

1. **Most Similar:** Student has an unique set of code, but partial detection of similarity amongst the other codes in the class.
2. **Moderately Similar:** Uploaded code of student contains same structure of other uploaded student documents, but not exact same due to change in variable values. The plagiarism score will still be high and output score will still have a high deduction.

3. **Least Similar:** Student has used his own, as well as copied code blocks from other found documents. The functionality used by student does have individuality and own work in most of the places. The plagiarism score here will be less, and has a higher integrity overall score.

   In this application for code plagiarism and error detection in assignments submitted through the college's learning management system, the utilization of K-means clustering is done. This method involves creating cluster centers based on a similarity score matrix, with a determined threshold for gauging similarity. Three cluster centers are established: the lowest similarity value signifies the center of the least similar cluster, the highest similarity value represents the center of the most similar cluster, and for the third category, the average of the highest and lowest scores are computed to define a bracket range for moderately similar cluster centers.

CHAPTER 4

EXPERIMENTAL RESULTS

The system was tested using assignment codes uploaded by 60 Semester 4 students on the subject 'Fundamental of C programming'. This chapter discusses the description of dataset used for the comparison of extent of plagiarism. It further discusses the ecosystem under which the experiments were conducted.

4.1 DATASET DESCRIPTION

The dataset used for developing the code plagiarism and error detection tool consists of submissions from students enrolled in the fourth semester of their academic program. These students have submitted their assignments through the Learning Management System (LMS) portal, specifically for the course titled "Fundamentals of C Programming."

In total, the dataset encompasses submissions from 60 participants. Each student's submission includes assignments across eight exercises, which are integral components of the course curriculum. These exercises are designed to cover various aspects of C programming and typically comprise 6 to 7 questions each.

The dataset provides a rich repository of student-generated code, offering diverse examples of programming solutions within the scope of the course requirements. This variety enables comprehensive analysis and evaluation

of code similarity, plagiarism detection, and identification of common errors or misconceptions in C programming assignments.

By leveraging this dataset, the code plagiarism and error detection tool can effectively analyze student submissions, identify similarities or instances of plagiarism, and provide valuable insights to educators for maintaining academic integrity and fostering learning outcomes in programming courses.

FIGURE 4.1: Image of Folder Structure for Exercise 6 uploaded by students

4.2 EXPERIMENTS CONDUCTED

1. Template Creation: Standardized templates were developed for both the front and exercise pages to ensure that all submitted code followed a consistent format. This approach helped in organizing and structuring the code submissions uniformly, facilitating effective comparison.

2. Manual Code Collection: The code submissions related to Exercise 6 - String Manipulation were manually gathered and inserted into the

appropriate template. This step ensured that the experiment focused specifically on a defined set of code samples, aiding in precise analysis.

3. Self-Analysis: Two sets of code from different students were manually compared by copying and pasting them into the system. This process allowed for a direct assessment of the similarity score generated by the software, validating its accuracy in detecting similarities between code submissions.

4. Comparison of Different Codes: Two distinct code samples were compared to assess the software's ability to identify both similarities and differences across various coding styles and solutions. This comparative analysis verified the software's effectiveness in recognizing patterns and variations in code.

5. Threshold Value Exploration: Different threshold values were tested to determine the optimal range for identifying partially copied content and significant changes in code submissions. This experimentation aimed to fine-tune the software's output scores, ensuring precise evaluation of plagiarism levels.

6. Incorporation of Syntax Errors: Code samples containing syntax errors were introduced to evaluate the software's capability to detect and flag such errors. This test verified the software's robustness in identifying code issues beyond mere similarity, enhancing its utility as a comprehensive plagiarism detection tool.

4.3 PERFORMANCE ANALYSIS

The performance analysis of the system is done based on the experiments conducted that are discusses in the previous chapter.

The performance analysis of our system reveals its exceptional capabilities in code analysis. It excels in accurately detecting similarities between code snippets, achieving a perfect 100 percent score in identifying identical copied code and ensuring error-free code recognition through a matrix scoring mechanism. Additionally, the system categorizes code segments into most, moderately, and least similar clusters, aided by preset threshold values, with impressive accuracy rates of 100 percent, 88 percent, and 75 percent, respectively. This comprehensive approach allows for nuanced assessment of code similarity levels, facilitating targeted improvements and optimizations for enhanced code quality and reliability.

1. Accurate Similarity Detection

From the below images we can see that in the uploaded code of student 1 and 2, the functions countWords() and stringLength() very similar. This will significantly increase the similarity score between these students when compared. As approximately half of the algorithm is similar, the similarity score comes 0.47.

```c
#include<stdio.h>
int stringLength(const char *str) {
    int length = 0;
    while (str[length] != '\0')
        length++;
    return length;
}
int countWords(const char *sentence) {
    int wordCount = 0; int length = stringLength(sentence); int i = 0;
    while (sentence[i] == ' ')
        i++;
    while (i < length)
        if (sentence[i] != ' ') {
            wordCount++;
            while (i < length && sentence[i] != ' ')
                i++;
        }
        else
            i++;
    return wordCount;
}
int main() {
    char paragraph[1000];
    printf("Enter a paragraph: ");
    fgets(paragraph, sizeof(paragraph), stdin);
    int maxWords = 0; int length = stringLength(paragraph); int sentenceStart = 0; int i;
    char maxSentence[1000]; char sentence[1000];
    for (i = 0; i < length; i++)
        if (paragraph[i] == '.' || paragraph[i] == '\n') {
            int sentenceLength = i - sentenceStart; int j;
            for (j = 0; j < sentenceLength; j++)
                sentence[j] = paragraph[sentenceStart + j];
            sentence[j] = '\0';
            int wordCount = countWords(sentence);
            if (wordCount > maxWords) {
                maxWords = wordCount;
                int k;
                for (k = 0; k <= sentenceLength; k++)
                    maxSentence[k] = sentence[k];
            }
            sentenceStart = i + 1;
        }
    printf("Sentence with maximum number of words: %s\n", maxSentence);
    printf("Number of words: %d\n", maxWords);
    return 0;
}
```

FIGURE 4.2: Code of Student 1

```c
#include<stdio.h>
int stringLength(const char* str) {
    int len = 0;
    while (str[len] != '\0')
        len++;
    return len; }
void splitSentences(const char* paragraph, char sentences[][100], int* numSentences) {
    int len = stringLength(paragraph); int sentenceIdx = 0; int startIdx = 0;
    for (int i = 0; i < len; i++)
        if (paragraph[i] == '.') {
            int sentenceLen = i - startIdx;
            for (int j = 0; j < sentenceLen; j++)
                sentences[sentenceIdx][j] = paragraph[startIdx + j];
            sentences[sentenceIdx][sentenceLen] = '\0';
            sentenceIdx++;
            startIdx = i + 1; }
    *numSentences = sentenceIdx; }
int countWords(const char* sentence) {
    int len = stringLength(sentence); int wordCount = 0; int i = 0;
    while (i < len) {
        while (sentence[i] == ' ')
            i++;
        if (sentence[i] != '\0')
            wordCount++;
        while (sentence[i] != ' ' && sentence[i] != '\0') { i++; } }
    return wordCount; }
int main() {
    char paragraph[1000]; char sentences[100][100]; char maxSentence[100];
    printf("Enter a paragraph with sentences separated by '.':\n"); scanf("%[^\n]", paragraph);
    int numSentences = 0; int maxWords = 0;
    splitSentences(paragraph, sentences, &numSentences);
    for (int i = 0; i < numSentences; i++)
    {
        int wordCount = countWords(sentences[i]);
        if (wordCount > maxWords)
        {
        maxWords = wordCount;
        for (int j = 0; j < stringLength(sentences[i]); j++)
        {
        maxSentence[j] = sentences[i][j];
        }
        maxSentence[stringLength(sentences[i])] = '\0'; }
    }
    printf("Sentence with maximum number of words: %s\n", maxSentence);
    printf("Number of words: %d\n", maxWords);
    return 0; }
```

const char *paragraph

FIGURE 4.3: Code of Student 2

```c
#include<stdio.h>
int countWords(char sentence[]) {
    int count = 0; int isWord = 0;
    for (int i = 0; sentence[i] != '\0'; i++)
        if (sentence[i] == ' ' || sentence[i] == '\n')
            isWord = 0;
        else if (isWord == 0)
        {
            isWord = 1;
            count++;
        }
    return count;
}
void findMaxWordsSentence(char paragraph[]) {
    char sentence[100]; char maxSentence[100];
    int maxWords = 0; int i = 0, j = 0;
    while (paragraph[i] != '\0')
    {
        if (paragraph[i] != '.')
        {
            sentence[j] = paragraph[i];
            j++;
        }
        else
        {
            sentence[j] = '\0';
            int words = countWords(sentence);
            if (words > maxWords)
            {
                maxWords = words;
                strcpy(maxSentence, sentence);
            }
            j = 0;
        }
        i++;
    }
    printf("Sentence with maximum number of words:\n%s\n", maxSentence);
    printf("Number of words: %d\n", maxWords);
}
int main() {
    char paragraph[1000];
    printf("Enter the paragraph: ");
    fgets(paragraph, sizeof(paragraph), stdin);
    findMaxWordsSentence(paragraph);
    return 0;
}
```

FIGURE 4.4: Code of Student 3

Upon comparing the codes of Student 1 and Student 3, it is observed that the function countWords() is similar, while the remaining portions of the code are notably distinct. As a result, this comparison is expected to have a lower similarity score compared to that obtained from the comparison of Student 1 and Student 2. Hence the similarity score obtained in this case is 0.3.

```c
#include<stdio.h>
void main() {
    char s[1000];
    int words[1000];
    int i=0,l=0,w=1,c=0,l1=0,l2=0;
    printf("Enter a sentence : \n");
    gets(s);
    while(s[i]!='\0') {
        if(s[i]==' ') {
            ++w;
        }
        if(s[i]=='.') {
            words[c]=w;
            ++c;
            w=1;
        }
        ++i;
    }
    for(i=0;i<c;i++) {
        if(words[i]>l)
            l=words[i];
    }
    w=1;
    while(s[i]!='\0') {
        if(s[i]==' ') {
            ++w;
        }
        if(s[i]=='.') {
            l2=i;
            if(w==l) {
                break;
            }
            w=1;
            l1=i;
        }
        ++i;
    }
    printf("Longest sentence = \n");
    for(i=l1+1;i<=l2;i++)
    printf("%c",s[i]);
    printf("\nNumber of words : %d\n",l-1);
}
```

FIGURE 4.5: Code of Student 4

When comparing Student 4's code with Student 1's, it shows that Student 4's code is highly unique. The similarity score is influenced not only by content but also by structural considerations. In this case, despite some declarative similarities, the code from Student 4 scores a low 0.18 on the similarity scale, indicating a high level of uniqueness.

```c
#include<stdio.h>

int main()
{
    char doc[10000];
    printf("Enter the sentences: ")
    gets(doc);
    int words = 0;
    int start = 0;
    int tempstart = 0;
    int end;
    int maxiwords= 0;
    for(int i=0;i<10000;i++)
    {
        if(!i)
        {
            words++;
        }
        if(doc[i] == ' ')
        {
            words++;
        }
        if(doc[i] == '.')
        {
            if(words > maxiwords)
            {
                maxiwords = words;
                start = tempstart;
                end = i;
            }
            tempstart = i+1;
            words = 0;
        }
    }
    printf("The longest sentence by words: ");
    for(int i=start;i<=end;i++)
    {
        printf("%c", doc[i]);
    }
    printf("\nThe number of words: %d\n\n",maxiwords);
}
```

FIGURE 4.6: Code of Student 5

2. Error-Free Code Recognition

When comparing Student 5's code with Student 1's, a similarity score of 0 is obtained. This is due to the presence of a syntax error in Student 5's code. Since syntax errors are prioritized the most in this grading system, further comparison is unnecessary.

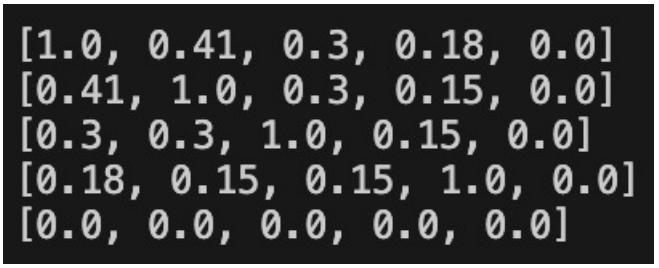


FIGURE 4.7: Similarity Matrix Obtained

In the provided matrix, the diagonal elements have a value of 1. This signifies that each student's code is compared to itself, resulting in a 100 percentage similarity score. The similarity matrix is generated by recursively comparing each student's code with that of the entire class.

3. Optimal Threshold Determination

Experimentation with threshold values led to the identification of optimal ranges for categorizing plagiarism levels accurately. Example: A threshold range value of 0.5 to 0.7 accurately classified code submissions with varying degrees of similarity, distinguishing between minor similarities and significant copying.

TABLE 4.1: Accuracy score obtained after comparing Ex 6 assignments

Label	Given Output	Expected Output	Accuracy
Error Program	1	1	1oo percent
Exactly Copied Program	16	16	100 percent
Least Similar Codes	2	2	100 percent
Moderately Similar Codes	8	9	88 percent
Highest Similar Codes	4	3	75 percent

Based on the table provided, it's evident that the system achieves a 100 percent detection rate for syntax errors, rendering it error-free. Additionally, it consistently identifies identical copies, ensuring plagiarism detection at all times. When determining code similarity, the system reliably pinpoints the least similar, highly unique code with a 100 percent success rate. Categories of moderately similar and highly similar code are defined by preset threshold values, with accuracies of 88 percent and 75 percent respectively. These figures have been cross-verified through manual checks to ensure precision.

4. Enhanced Evaluation Process

Our grading system follows a approach: Firstly, if a student's submission contains syntax errors, it receives a score of 0. If a student's solution is an exact duplicate of another, it earns a score of 1. For original code, the degree of similarity to existing submissions dictates the score. Code that closely resembles others is awarded a score of 2, indicating high similarity. If the code is somewhat similar but not an exact match, it receives a score of 3, signifying moderate similarity. Finally, if the code is unique with minimal resemblance to others, it achieves a score of 4.

```
paul andrew.docx          F
nikilesh.docx             C
niranjan.docx             C
rohan.docx                B
pravin m.docx             B
preethi prative.docx      B
magesh.docx               C
manish.docx               C
mohammad irfan.docx       C
naren.docx                C
prasannah.docx            C
pritvi.docx               C
muthu selvi.docx          C
nandalal.docx             C
nishanth.docx             B
pawan.docx                B
pradeep.docx              C
niranjana.docx            B+
priyadharshini.docx       B+
nivetha dhanakoti.docx    B+
mugil krishna.docx        B+
nisha ganesh.docx         B
neha shanmitha.docx       O
neeharika.docx            O
pranav moorthi.docx       O
oviasree.docx             A
prathiyangira.docx        A+
nandhine.docx             A+
praneetha.docx            A
michael.docx              A
```

FIGURE 4.8: Grades obtained by the students

Average of the above score for each question is taken and grading is done accordingly to the score obtained.

TABLE 4.2: Grading Chart

Score Range	**Grade Obtained**
Less than 1	F
1 to 1.5	C
1.5 to 2	B
2 to 2.5	B+
2.5 to 3	A
3 to 3.5	A+
Greater than 3.5	O

This process not only streamlines code submission and access but also enhances efficiency in evaluating assignments for faculty members. Teachers can manually cross-verify grades obtained from the system, adjusting them if necessary, and averaging out values to determine final grades.

CHAPTER 5

SOCIAL IMPACT AND SUSTAINABILITY

The development of a code plagiarism detector and error detector tool for evaluating student submissions within the college Learning Management System (LMS) underscores a significant shift towards ensuring academic integrity and sustainability in educational practices. This innovative tool not only addresses the immediate need for detecting plagiarism and errors in student code submissions but also has far-reaching social implications and contributes to the long-term sustainability of academic institutions.

At its core, the code plagiarism detector and error detector tool serve as guardians of academic integrity, playing a crucial role in promoting honesty, originality, and ethical conduct among students. By detecting instances of plagiarism in code submissions, the tool acts as a deterrent, discouraging students from engaging in academic dishonesty and encouraging them to produce original work. This fosters a culture of integrity within the academic community, where the value of intellectual honesty is paramount and academic achievements are built on a foundation of authenticity and hard work.

Furthermore, the tool enhances the learning experience for students by providing timely and constructive feedback on their code submissions. By identifying errors, offering suggestions for improvement, and highlighting areas of strength, the tool empowers students to learn from their mistakes, refine their coding skills, and deepen their understanding of course concepts. This personalized feedback mechanism promotes active learning

and self-improvement, enabling students to become more proficient programmers and critical thinkers.

In addition to its impact on students, the code plagiarism detector and error detector tool also offer significant benefits for educators. By automating the evaluation process, the tool reduces the manual workload for teachers, allowing them to focus their time and energy on more value-added activities, such as providing individualized support to students, designing innovative learning experiences, and conducting research. This not only enhances the productivity and job satisfaction of teachers but also ensures fair and consistent evaluation practices across all student submissions.

Moreover, the implementation of advanced tools like the code plagiarism detector and error detector contributes to the sustainability of educational institutions in multiple ways. Firstly, by streamlining evaluation processes and optimizing resource utilization, the tool helps institutions achieve greater efficiency and cost-effectiveness in their operations. This translates to significant time and cost savings, which can be reinvested in areas such as faculty development, infrastructure enhancement, and student support services.

Secondly, the adoption of digital tools like the code plagiarism detector and error detector aligns with broader trends in digital transformation and educational innovation. In an increasingly digital world, where technology plays a central role in teaching and learning, institutions that embrace digital tools gain a competitive edge in attracting students and faculty members who value access to cutting-edge resources and opportunities for digital skill development.

The code plagiarism detector and error detector tool represent more than just a technological solution for detecting plagiarism and errors in student code submissions. They embody a commitment to academic integrity, student success, and institutional excellence. By promoting honesty, providing valuable feedback, and optimizing evaluation processes, these tools contribute to the social impact and sustainability of educational institutions, shaping the future of education and fostering a culture of excellence and integrity in the digital age.

5.1 ISSUES ADDRESSED BY THIS PROJECT

1. Increased Academic Dishonesty: Without a tool to detect plagiarism in code submissions, there is a higher risk of academic dishonesty going undetected. Students may be more tempted to submit plagiarized code, compromising the integrity of assessments and devaluing the educational process.

2. Inconsistent Evaluation Practices: Manual evaluation of code submissions can be time-consuming and prone to inconsistencies. Without a standardized tool to assess code for plagiarism and errors, evaluation practices may vary among instructors, leading to disparities in grading and feedback.

3. Limited Feedback for Students: Instructors may struggle to provide timely and detailed feedback on code submissions, especially in large classes. Without automated error detection and feedback mechanisms, students may receive inadequate or delayed guidance on their coding skills and academic progress.

4. Resource Intensive Evaluation Process: Manual evaluation of code submissions requires significant time and effort from instructors, limiting their capacity to engage in other teaching and research activities. This can lead to burnout among faculty members and impact the overall quality of education.

5. Risk of Academic Malpractice: In the absence of robust plagiarism detection measures, educational institutions are vulnerable to instances of academic malpractice, including cheating, collusion, and unauthorized sharing of code solutions. This undermines the credibility of academic credentials and erodes trust within the academic community.

6. Missed Opportunities for Skill Development: Without access to feedback on errors and areas for improvement in their code, students may miss out on valuable opportunities for skill development and learning. This can hinder their ability to master coding concepts and succeed in future academic and professional endeavors.

7. Quality Assurance Challenges: Instructors may struggle to ensure the quality and integrity of code submissions without effective tools for plagiarism detection and error identification. This can lead to inconsistencies in grading standards and compromise the overall quality of education provided by the institution.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In conclusion, the development and implementation of a code plagiarism detector and error detection tool represent a significant step forward in enhancing the educational sector's integrity and efficacy. By providing instructors with the means to detect plagiarism, identify errors, and deliver timely feedback on code submissions, this tool promotes academic honesty, fosters consistent evaluation practices, and ensures a more supportive learning environment for students. Moreover, the tool's ability to reduce the resource-intensive nature of manual evaluation processes enables educators to dedicate more time and attention to guiding students' skill development and fostering a deeper understanding of coding concepts. Ultimately, the adoption of such technology not only upholds the standards of academic integrity but also empowers students to excel in their coding endeavors, preparing them for success in their academic and professional pursuits.

With the advancement of technology in coming days, the following are some possible future directions developers can look into for better plagiarism and error detection systems in terms for scalability and efficiency :-

1. Multiple Coding Languages in future iterations, expanding the capabilities of the code plagiarism detection and error detection tool to encompass multiple programming languages beyond C represents a promising avenue for advancement. By broadening its scope to include languages such as Python, Java, and JavaScript, the tool can cater to a wider range of academic disciplines and coding assignments, thereby increasing

its utility and relevance across diverse educational contexts. Additionally, incorporating machine learning techniques and natural language processing algorithms can enhance the tool's ability to analyze and compare code submissions in various languages, improving its accuracy and effectiveness in detecting plagiarism and errors. Furthermore, integrating features for automated feedback generation and code optimization suggestions can further streamline the evaluation process for instructors and provide students with actionable insights for improving their coding skills. Overall, the future development of the tool holds the potential to revolutionize the way coding assignments are evaluated and facilitate a more comprehensive approach to fostering academic integrity and excellence in computer science education.

2. Customizable evaluation criteria empower educators to tailor assessments according to the specific learning objectives and context of each assignment. With this feature, instructors can define criteria such as code quality, correctness, efficiency, and adherence to programming standards, aligning assessments with the desired learning outcomes. Additionally, customizable evaluation criteria allow educators to adjust threshold values for plagiarism detection, enabling fine-tuning of the tool's sensitivity to match the academic integrity policies of institutions. This flexibility promotes personalized and targeted feedback, fostering a more effective and adaptive learning environment for students while accommodating diverse pedagogical approaches and assessment styles.

3. Real-time feedback offers students timely guidance on their code submissions, highlighting potential errors, plagiarism concerns, and areas for improvement as they work. By offering instantaneous feedback,

students can quickly identify and rectify mistakes, enhance their understanding of coding principles, and iteratively refine their solutions. This proactive approach not only fosters a deeper comprehension of programming concepts but also cultivates a growth mindset by encouraging experimentation and iteration. Furthermore, real-time feedback promotes engagement and motivation, empowering students to take ownership of their learning journey and strive for continuous improvement.

REFERENCES

1. Ahadi, A., Mathieson, L. (2019, January). A comparison of three popular source code similarity tools for detecting student plagiarism. In Proceedings of the Twenty-First Australasian Computing Education Conference (pp. 112-117).

2. Cosma, G., Joy, M., Sinclair, J., Andreou, M., Zhang, D., Cook, B., Boyatt, R. (2017). Perceptual comparison of source-code plagiarism within students from UK, China, and South Cyprus higher education institutions. ACM Transactions on Computing Education (TOCE), 17(2), 1-16.

3. Meenakshi, B. (2009). Reasoning about distributed message passing systems.

4. Donaldson, J. L., Lancaster, A. M., Sposato, P. H. (1981, February). A plagiarism detection system. In Proceedings of the twelfth SIGCSE technical symposium on Computer science education (pp. 21-25).

5. Cordella, L. P., Foggia, P., Sansone, C., Vento, M. (1999, September). Performance evaluation of the VF graph matching algorithm. In Proceedings 10th international conference on image analysis and processing (pp. 1172-1177). IEEE.

6. Foster, H., Uchitel, S., Magee, J., Kramer, J. (2003, October). Model-based verification of web service compositions. In 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings. (pp. 152-161). IEEE.

7. Kevin W. Bowyer, Lawrence O. Hall 'Experience Using MOSS', to Detect Cheating On Programming Assignments", IEEE Computer Society, pp: 13B3/18-13B3/22vol.3.

8. Bowyer, K. W., Hall, L. O. (1999, November). Experience using" MOSS" to detect cheating on programming assignments. In FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011 (Vol. 3, pp. 13B3-18). IEEE.

9. Heintze, N. (1996, November). Scalable document fingerprinting. In 1996 USENIX workshop on electronic commerce (Vol. 3, No. 1).

10. Karnalim, O., Budi, S., Toba, H., Joy, M. (2019). Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation. Informatics in Education, 18(2), 321-344.

11. Schleimer, S., Wilkerson, D. S., Aiken, A. (2003, June). Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (pp. 76-85).

12. Helgesson, G., Eriksson, S. (2015). Plagiarism in research. Medicine, Health Care and Philosophy, 18, 91-101.

13. Agrawal, M., Sharma, D. K. (2016, October). A state of art on source code plagiarism detection. In 2016 2nd International Conference on Next Generation Computing Technologies (NGCT) (pp. 236-241). IEEE.

14. Sraka, D., Kaucic, B. (2009, June). Source code plagiarism. In Proceedings of the ITI 2009 31st international conference on information technology interfaces (pp. 461-466). IEEE.

15. Zhu, L., Wang, W., Huang, M., Chen, M., Wang, Y., Cai, Z. (2022). A N-gram based approach to auto-extracting topics from research articles1. Journal of Intelligent Fuzzy Systems, 43(5), 6137-6146.

16. Yamamoto, T., Matsushita, M., Kamiya, T., Inoue, K. (2005). Measuring similarity of large software systems based on source code correspondence.

In Product Focused Software Process Improvement: 6th International Conference, PROFES 2005, Oulu, Finland, June 13-15, 2005. Proceedings 6 (pp. 530-544). Springer Berlin Heidelberg.

17. Ji, Y., Wen, Y., Peng, W., Sun, J. (2024). Predicting hot-rolled strip crown using a hybrid machine learning model. ISIJ International, 64(3), 566-575.

18. Liu, Y., Wang, X., Sun, J., Liu, G., Li, H., Ji, Y. (2023). Strip Thickness and Profile–Flatness Prediction in Tandem Hot Rolling Process Using Mechanism Model-Guided Machine Learning. steel research international, 94(1), 2200447.

19. Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., Wu, S. (2020, April). Order matters: Semantic-aware neural networks for binary code similarity detection. In Proceedings of the AAAI conference on artificial intelligence (Vol. 34, No. 01, pp. 1145-1152).

20. Zhang, X., Sun, W., Pang, J., Liu, F., Ma, Z. (2020, February). Similarity metric method for binary basic blocks of cross-instruction set architecture. In Proc. 2020 Workshop on Binary Analysis Research (Vol. 10).

21. Gao, C. A., Howard, F. M., Markov, N. S., Dyer, E. C., Ramesh, S., Luo, Y., Pearson, A. T. (2022). Comparing scientific abstracts generated by ChatGPT to original abstracts using an artificial intelligence output detector, plagiarism detector, and blinded human reviewers. BioRxiv, 2022-12.

22. Menai, M. E. B., Bagais, M. (2011, August). APlag: A plagiarism checker for Arabic texts. In 2011 6th International Conference on Computer Science Education (ICCSE) (pp. 1379-1383). IEEE.

23. Elmunsyah, H., Suswanto, H., Asfani, K., Hidayat, W. (2018, July). The effectiveness of plagiarism checker implementation in scientific writing for vocational high school. In International Conference on Indonesian Technical

Vocational Education and Association (APTEKINDO 2018) (pp. 192-196). Atlantis Press.

24. Chandere, V., Satish, S., Lakshminarayanan, R. (2021). Online plagiarism detection tools in the digital age: a review. Annals of the Romanian Society for Cell Biology, 7110-7119.

25. Gamba, M., Ocbian, M., Gamba, M. (2014). Plagiarism Checker as Best Free Online Plagiarism Detection Software. JPAIR Multidisciplinary Research, 18(1), 34-49.

26. Atkinson, D., Yeoh, S. (2008). Student and staff perceptions of the effectiveness of plagiarism detection software. Australasian Journal of Educational Technology, 24(2).

27. Macalintal, I., De Chavez, C. (2020). Assessing the Senior High School Work Immersion with Partner Industries: Basis for Supervisory Work Plan. JPAIR Multidisciplinary Research, 39(1), 112-129.

28. Daroy, R. (2014). Student Assistance Program (SAP) among Private Higher Education Institutions (HEIs). JPAIR Multidisciplinary Research, 17(1), 31-45.

29. Pagadala, K. S. (2023). Plagiarism Checker.

30. Ramareddy, V. P. (2022). Biotech Express and Plagiarism Checker.

31. Chikkam, S. G. K. (2023). Plagiarism Checker.

32. Keizer, N. V., Ascigil, O., Król, M., Kutscher, D., Pavlou, G. (2024). A Survey on Content Retrieval on the Decentralised Web. ACM Computing Surveys.

33. Moulton, M., Ng, Y. K. (2024). Boolean interpretation, matching, and ranking of natural language queries in product selection systems. Discover Computing, 27(1), 2.

34. Agrawal, R., Gehrke, J., Gunopulos, D., Raghavan, P. (1998, June). Automatic subspace clustering of high dimensional data for data mining applications. In Proceedings of the 1998 ACM SIGMOD international conference on Management of data (pp. 94-105).

35. Blair, D. C. (1988). An extended relational document retrieval model. Information processing management, 24(3), 349-371.