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

G Sanjith, S Premkumaran , S Mukilesh , Sri Sivasubramaniya Nadar College of Engineering

Abstract—**In the digital age, where access to information is abundant and academic pressure is high, ensuring academic integrity has become a paramount concern for educational institutions worldwide. As colleges embrace Learning Management Systems (LMS) to facilitate teaching and learning, integrating robust plagiarism and error detection mechanisms within these platforms emerges as a crucial step towards upholding academic honesty. The implementation of a plagiarism detection tool within the LMS not only acts as a deterrent against academic dishonesty but also fosters a culture of originality and critical thinking among students. By scanning submitted assignments, projects, and essays, this tool identifies instances of content duplication, improper citation, or unauthorized collaboration, enabling educators to address such issues promptly and uphold the integrity of academic assessment. Our model is developed for our college's Learning Management System (LMS), enhancing the process of code analysis within student assignments. This tool will ease the extraction of code from uploaded assignments, followed by a tokenization and parsing process aligned with predefined grammar rules. The outcome of this process is the generation of metric scores, providing a comprehensive assessment of the quality and structure of the code. These metric scores, collected from all submitted assignments, are sent to our comparator where then compiled matric scores are stored into a matrix format. This matrix acts as a centralized repository that encapsulates the coding proficiency and stylistic variations among students. Through a systematic process, our tool facilitates the clustering of assignments based on similarities and discrepancies on factors that include identifying syntax errors, detecting instances of direct code replication, monitoring changes in variable names, and assessing the significance of alterations made to the codebase. This clustering mechanism not only simplifies the process of identifying common trends and patterns within student submissions but also enables educators to pinpoint areas requiring attention and intervention.**

I. INTRODUCTION

THe identification of code plagiarism is an essential practice with multiple primary motivations. Its primary goal is to detect instances of unauthorized code copying in order to protect the integrity of software development. Protecting the original developers' intellectual property is another goal, but it also makes sure that software is free from security flaws, maintains its quality, and conforms with the law. Code plagiarism is a serious problem that affects academics of all lines, including teachers and students. Students frequently give in to the temptation of duplicating code under the stressful environment of tests and assessments, compromising real learning in favour of quick benefits. However, the effects go well beyond the classroom. Employers value integrity and skill, thus students who use plagiarised code jeopardise not only their academic standing but also their future employment opportunities. Faculty members must simultaneously fight plagiarism and maintain academic integrity, which takes time and resources away from teaching and research. In order to effectively address this issue, academic institutions need to develop a culture of academic integrity and put in place strong detection procedures.

II. PRELIMINARY

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.

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.

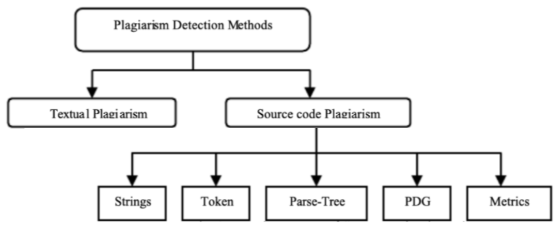

Fig. 1. Example of an Overall Plagiarism checker

III. PROBLEM DEFINITION

In this section, we define the problem our proposed defense framework aims to address and present the system model that underlies our approach.

A. Problem Definition

Designing and developing a comprehensive software solution aimed at detecting plagiarism and identifying errors in assignments submitted through the Learning Management System (LMS) of the college portal. The software generates finalized score indicative of similarity and error density within submitted assignments, facilitating faculty members in streamlining the evaluation process. The primary objective is to enhance the efficiency and accuracy of assignment evaluation while ensuring academic integrity within the educational institution.

B. Objectives

The main objectives of this 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.

IV. PROPOSED SYSTEM

This chapter delineates the distinct phases of our system, encompassing information retrieval, tokenization and parsing,

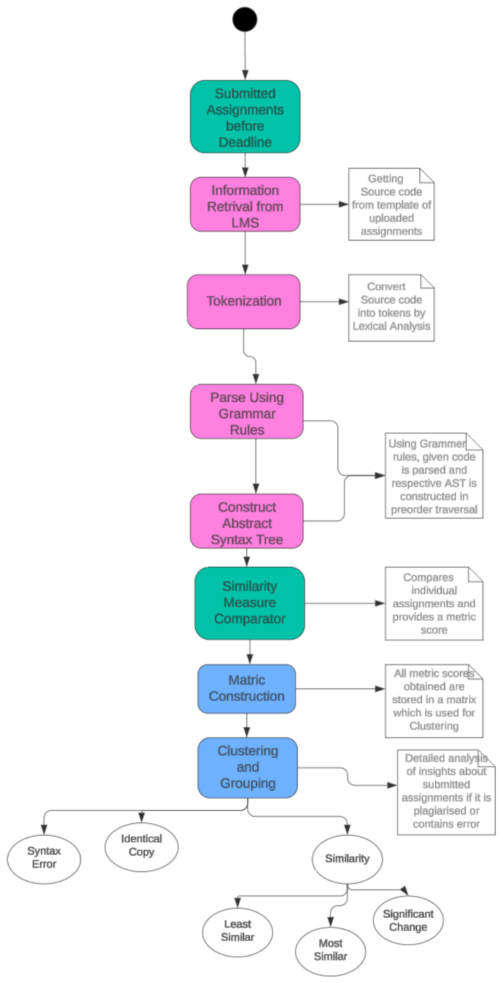
Submitted Assignments before Deadline
Information Retrival from LMS
Getting Source code from template of uploaded assignments
Tokenization
Convert Source code into tokens by Lexical Analysis
Parse Using Grammar Rules
Using Grammer rules, given code is parsed and respective AST is constructed in preorder traversal
Construct Abstract Syntax Tree
Similarity Measure Comparator
Compares individual assignments and provides a metric score
Matric Construction
All metric scores obtained are stored in a matrix which is used for Clustering
Clustering and Grouping
Detailed analysis of insights about submitted assignments if it is plagiarised or contains error
Syntax Error
Identical Copy
Similarity
Least Similar
Most Similar
Significant Change

Fig. 2. Example of an Overall Plagiarism checker

comparator function, and evaluation criteria. It elaborates on each component of the architecture 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.

A. 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 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.

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.

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 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.

B. Tokenozation 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 which is also used 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.

C. 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.

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.

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 range of matric score ranges from 0 to 1, where 1 is the highest possible value for plagiarism.

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:

ratio = 2 x (number of matching element)

(length of sequence A + length of sequence B)

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 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.

V. Related Works

From 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.

From 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 two 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.

From 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.

From 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.

From 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.

From 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.

From 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.

From 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

VI. EXPERIMENTAL RESULTS

A. 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.

B. 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.

VII. PERFORMANCE ANALYSIS

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

1. **Accurate Similarity Detection** Outcome: The tool accurately identified similarities between code submissions, enabling effective plagiarism detection. Example: A threshold range value of 0.8 indicated significant similarity between two code samples, suggesting a high likelihood of plagiarism.

2. **Error-Free Code Recognition** Outcome: Syntax errors were effectively detected and flagged by the tool, ensuring the integrity of code submissions. Example: Code snippet containing syntax errors was assigned a lower similarity score (e.g., 0.3), indicating discrepancies compared to error-free submissions.

3. **Optimal Threshold Determination** Outcome: 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.

4. **Consistent Performance Across Experiments** Outcome: The tool consistently performed well across different experiments, showcasing its reliability and robustness. Example: Despite variations in code complexity and structure, the tool consistently assigned appropriate similarity scores, maintaining consistency in performance.

5. **Enhanced Evaluation Process** Outcome: The tool facilitated a streamlined evaluation process for code submissions, enabling instructors to efficiently identify plagiarism and errors. Example: By automatically categorizing code submissions based on similarity scores, instructors could prioritize reviewing potentially plagiarized content, enhancing assessment efficiency.

6. **Improved Academic Integrity** Outcome: The implementation of the tool contributed to the promotion of academic integrity by discouraging plagiarism and fostering originality. Example: Students were more conscious of adhering to coding ethics and submitting original work, knowing that their submissions would undergo rigorous scrutiny.

VIII. Conclusion

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.

IX. Future Works

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 :-

A. 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.

B. Customizable evaluation criteria

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.

C. Real-time feedback

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

[2] cowir1996. E. Hopcroft and J. K. Wong. ,*Performanceevaluation of the VF graph matching algorithm.* , Proc. of 10th Int. Conf. on Image Analysis andProcessing, 1999

[3] F. Mittelbach and M. Goossens, *The LaTeXCompanion*, 2nd ed. Boston, MA, USA: Pearson, 2004.

[4] Mayank Agrawal, Dilip Kumar Sharma*A State of Art on Source Code Plagiarism Detection* ,mputer Engineering & Applications GLA University Mathura, India In Proceedings of ACL 2012.

[5] cowir1996Saul Schleimer ,Daniel S. Wilkerson, Alex Aiken ,*Winnowing: Local Algorithms for Document Fingerprinting* , Proceedings of the 2003 ACM SIGMOD

[6] H. Sira-Ramirez, "On the sliding mode control of nonlinear systems," *Syst. Control Lett.*, vol. 19, pp. 303–312, 1992.

[7] M. Fliess, C. Join, and H. Sira-Ramirez, "Non-linear estimation is easy," *Int. J. Model., Ident. Control*, vol. 4, no. 1, pp. 12–27, 2008.

[8] Nevin Heintze ,*calable document fingerprinting* , *1996 USENIX Workshop on Electronic Commerce*, November 1996.

[9] Ahadi, A., & Mathieson, L ,*A comparison of three popular source code similarity tools for detecting student plagiarism. 21st Australasian Computing Education Conference* , 112–117

[10] Cosma, G., & Joy, M. (2008) ,*Towards a definition of source-code plagiarism. IEEE Transactions on Education* , *51*(2), 195–200.

[11] Cosma, G., Joy, M., Sinclair, J., Andreou, M., Zhang, D., Cook, B., & Boyatt, R(2017),*comparison of source-code plagiarism within students from UK, China, and South Cyprus higher education institutions. ACM Transactions on Computing Education*, *51*(2), 195–200.

[12] Donaldson, J. L., Lancaster, A.-M., & Sposato, P. H. (1981). ,*A plagiarism detection system. 12th SIGCSE Technical Symposium on Computer Science Education* ,*3*(1), 21–2

[13] Oscar Karnalim, Setia Budi, Hapnes Toba, Mike Joy*Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation Informatics in Education, 2019, Vol. 18, No. 2, 321–344*