International Journal of Science and Research Archive

eISSN: 2582-8185

Cross Ref DOI: 10.30574/ijsra

Journal homepage: https://ijsra.net/

(RESEARCH ARTICLE)

Anti-plagiarism tool helping developers to generate authentic code

Pradeep T, Adithya Kumar Gonepally *, Arun Kumar Pusala and Charan Teja Singarapu

Department of CSE (Artificial Intelligence and Machine Learning) of ACE Engineering College, India.

International Journal of Science and Research Archive, 2025, 14(01), 1208-1215

Publication history: Received on 11 December 2024; revised on 18 January 2025; accepted on 21 January 2025

Article DOI: https://doi.org/10.30574/ijsra.2025.14.1.0199

Abstract

Source code plagiarism detection has become a critical area of research with the increasing prevalence of code reuse in academic and professional settings. In order to achieve thorough code comparison, this work introduces a novel tool for detecting source code plagiarism that combines lexical similarity, abstract syntax trees (ASTs) and cosine similarity. The system incorporates a dynamic front-end that was created using Streamlit, providing an intuitive user interface with a code editor that can run code. Through the "Check Similarity" feature, which calculates the plagiarism percentage and finds the most similar file, the application offers real-time plagiarism detection. The methods, benefits, and difficulties of various approaches are examined in this study, with a focus on how well they identify structural and syntactic similarities. The suggested system has a great deal of promise for academic and professional environments, offering reliable and efficient plagiarism detection.

Keywords: Source Code Plagiarism Detection; Cosine Similarity; Abstract Syntax Trees (AST); Lexical Similarity; Streamlit Framework; Scikit-Learn Module

1. Introduction

The exponential growth of digital content and programming resources has amplified concerns regarding source code plagiarism, particularly in academic and professional domains. As open repositories and collaborative coding platforms have grown in popularity, it has become more difficult to guarantee the uniqueness and moral application of code. Strong detection techniques are required because source code plagiarism compromises academic integrity, intellectual property rights and the impartial assessment of coding abilities.

This project focuses on addressing these challenges by developing an advanced source code plagiarism detection system that combines cutting-edge techniques such as cosine similarity, Abstract Syntax Trees (ASTs), and lexical similarity to identify structural, syntactic and semantic resemblances between code files. Built with Streamlit, the system provides an interactive and user-friendly interface, featuring an integrated code editor capable of executing code and a real-time plagiarism detection mechanism. With the click of a "Check Similarity" button, users can analyze code files, view the plagiarism percentage and identify the most similar file in the dataset.

The system's combination of static and dynamic analysis techniques guarantees dependable performance across a range of programming languages and coding styles, while its use of machine learning algorithms from the Scikit-learn module improves the accuracy and scalability of similarity identification. This project intends to assist organizations, developers, and academic institutions in upholding integrity, encouraging creativity, and advancing ethical coding methods by offering a solution that is accurate, efficient and user-centric.

* Corresponding author: Adithya Kumar Gonepally

2. Literature survey

2.1. Detecting Plagiarism in Source Code Using Machine Learning Approaches

Source code plagiarism detection is a growing concern in both academic and professional settings, affecting intellectual property and academic integrity. Code duplication and similarity have been successfully detected using conventional methods such as Cosine Similarity, Abstract Syntax Tree (AST) analysis and Lexical Similarity. Based on the textual content, Cosine Similarity helps detect plagiarism by measuring the angle between code vectors. It works well for finding precise text-based matches, but it could have trouble with changed code, like variable renaming or reordering. Even when minor changes, such as renaming or reordering, are done, plagiarism can be detected because AST analysis captures the syntactic structure of code by turning it into a tree. Such changes are less likely to occur with this approach. Lexical Similarity compares the token-level structure of the code, identifying exact matches or minor changes. Although it is very good at identifying copied code, it could overlook more complex plagiarism strategies. Although there are still issues with more intricate types of code modification, each technique has advantages and, when combined, provides a complete solution.

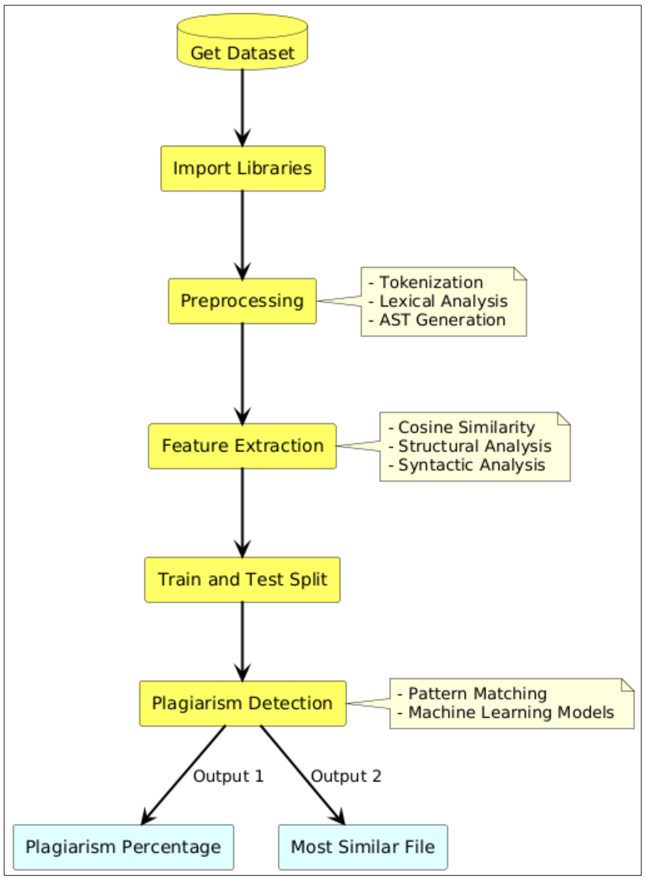


Figure 1 Workflow of the Source Code Plagiarism Detection System

2.2. Evaluating the Efficacy of Code Similarity Detection Tools in Differentiating Between Unique and Plagiarized Code

Fighting plagiarism in programming and upholding academic integrity are severely hampered by the growing use of automated code generating tools like GitHub Copilot. The usefulness of code similarity detection methods in distinguishing between original and copied or artificial intelligence-generated code is examined in this study. Copilot and other AI-assisted technologies were utilized to generate code snippets, and human-written control samples served as a point of comparison. The code's similarity was assessed using detection tools like JPlag, MOSS (Measure of Software Similarity) and specific AST-based techniques. The results show that tools that relied on syntax-based comparisons, such as MOSS, did well in detecting verbatim copies or modest reformatting, but they had trouble spotting logical similarities that were hidden by considerable rephrasing or obfuscation tactics. On the other hand, even when obfuscation was present, AST-based techniques demonstrated greater accuracy in identifying logical similarities through code structure and flow analysis. Some methods, however, generated false positives when used on human-written control samples, underscoring the necessity of further context-specific analysis and fine-tuning. This study emphasizes how crucial it is to create reliable, multi-layered detection systems in order to handle the growing complexity of code plagiarism tactics.

2.3. Detecting Source Code Similarity Using Abstract Syntax Trees: A Systematic Review

Improvements in source code plagiarism detection have become necessary due to the increasing usage of automated programming techniques. The AST-based methods that are the subject of this systematic review provide a structural perspective of the code, which makes them appropriate for detecting syntactic and logical similarities. The effectiveness of methods ranging from basic token matching to sophisticated subtree comparison in identifying copied or altered code is assessed in this paper.

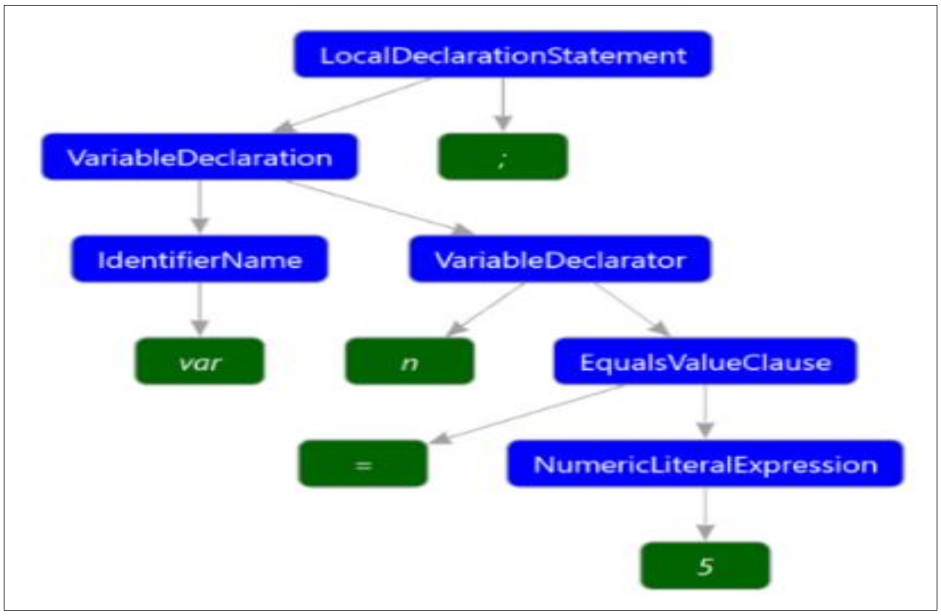
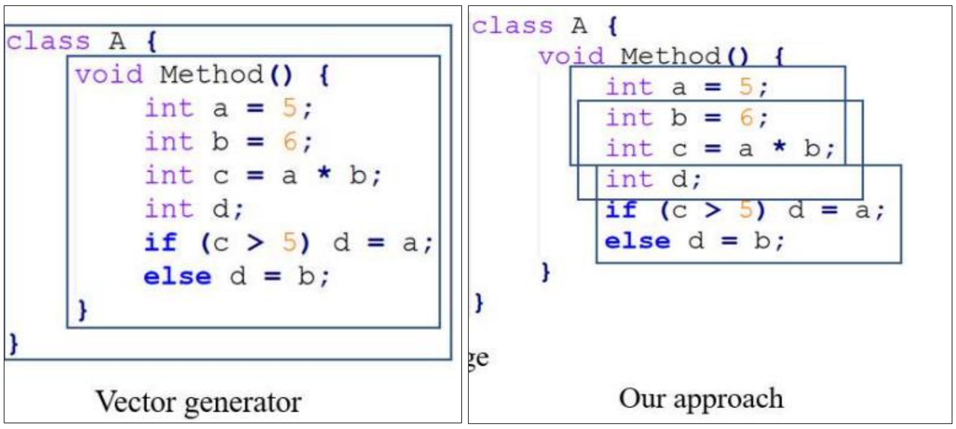


Figure 2 AST Representation for Code Similarity Analysis

According to the review, AST-based techniques are particularly good at spotting logical parallels that avoid token-based methods when variables are renamed, formatting is altered, or statements are rearranged. A number of techniques use Jaccard Similarity to assess the degree of similarity between code snippets by measuring the overlap between subtree sets. Even if they work well, AST-based techniques have trouble processing highly obfuscated code and demand a lot of processing power when dealing with big datasets. In order to accomplish thorough detection, the review's conclusion emphasizes the necessity of hybrid approaches that integrate AST analysis with lexical and semantic methodologies.

2.4. Code Plagiarism Detection: A Comparative Analysis

This study's main goal is to assess how well different approaches identify plagiarism in source code, with an emphasis on lexical similarity metrics, Abstract Syntax Tree (AST)-based approaches, and Jaccard Similarity for subtree matching. Analysis was done on a dataset that included both manually edited and AI-generated plagiarized samples and human-written code. The results show that even when the code has experienced major changes, including renaming variables, altering the sequence of statements, or applying formatting changes, AST-based methods are very good at spotting structural similarities. When used to compare subtrees, Jaccard Similarity provides an accurate way to gauge how much the logical structures of various programs coincide. Furthermore, lexical similarity techniques identify subtle code modifications and direct textual overlaps, which enhance AST-based analysis. This multifaceted approach draws attention to each method's advantages and disadvantages. Lexical approaches are quicker but have trouble with highly obfuscated or logically changed code, whereas AST-based approaches are excellent at identifying logical plagiarism but may need more processing power. According to the study's findings, integrating these approaches offers a strong way to deal with plagiarism detection issues, which qualifies them for use in both academic and professional settings.



```
class A {
    void Method() {
        int a = 5;
        int b = 6;
        int c = a * b;
        int d;
        if (c > 5) d = a;
        else d = b;
    }
}
```

Figure 3 Analysis of Code Representation Approaches

2.5. Testing Similarity Detection Tools for Source Code:

Recent advancements in AI-assisted programming tools have heightened concerns about code plagiarism, necessitating robust detection methods. The effectiveness of different similarity detection algorithms to detect plagiarism in both human-written and AI-generated code is the main topic of this work. Human-written, AI-generated, and obfuscated code snippets were used to test tools such as JPlag, MOSS, and bespoke AST-based algorithms. The findings show that while syntax-based methods such as MOSS are good at identifying outright copying, they struggle with code that is obfuscated or logically identical. On the other hand, even when there is substantial alteration, AST-based techniques offer better accuracy in detecting structural similarities. These approaches, however, are computationally demanding and need to be updated often to take into account new AI-generated solutions. The study draws attention to the shortcomings of the available detection methods and the necessity of hybrid strategies that combine lexical and structural analysis for thorough plagiarism detection.

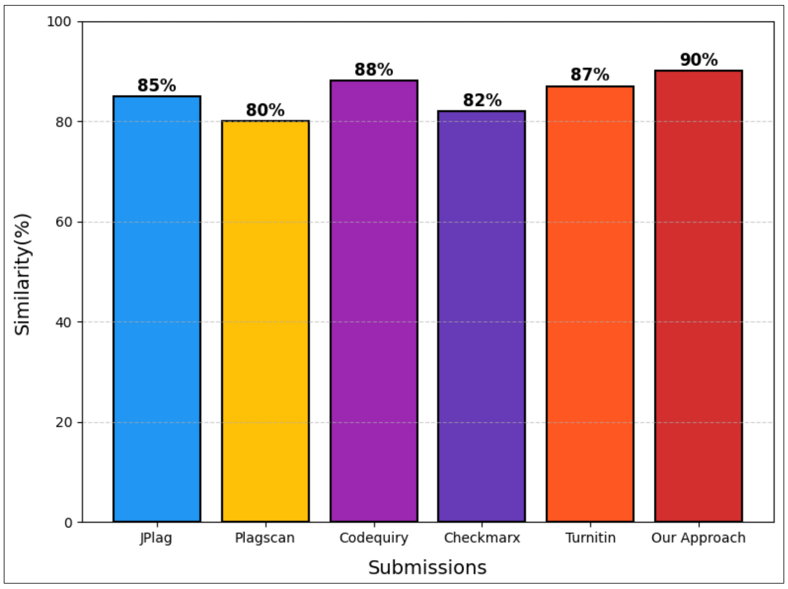


Figure 4 Similarity Comparison Across Different Plagiarism Detection Tools

2.6. challenges and Limitations:

Despite advancements in similarity detection techniques, identifying plagiarism in source code remains a significant challenge. Logical similarities in significantly altered or obfuscated code are difficult to find using traditional methods that rely on syntax-based analysis. Although AST-based techniques analyze structural aspects to provide improved performance, they are computationally costly and necessitate knowledge of tree-based procedures.

Although they are useful for detecting textual overlap, lexical similarity techniques frequently miss more profound logical connections. These difficulties are made worse by the increasing complexity of AI-generated code, such as highly obfuscated or context-aware systems. Accuracy and computational efficiency must also be balanced when deploying detection systems in practical applications, such academic integrity checks.

The integration of AST-based methods, Jaccard Similarity, and lexical analysis offers a promising approach. To stay effective against changing code-generation methodologies, these systems must be updated and retrained frequently. In order to overcome these obstacles, creative and flexible methods that can manage the textual and structural components of source code are needed, guaranteeing accurate detection in a variety of situations.

3. Proposed Methodology

Since the emergence of source code plagiarism, significant advancements have been made to enhance detection techniques. To address issues with plagiarism detection, including structural similarity, syntactic changes, and semantic understanding of code, a number of methods and tools have been created. Some of the noteworthy techniques and frameworks that have raised the precision and effectiveness of plagiarism detection systems are covered in this part. These developments concentrate on getting over restrictions like obfuscation, guaranteeing accuracy across various programming styles, and efficiently examining changing trends in source code plagiarism.

3.1. Abstract Syntax Tree (AST) and Similarity Analysis

The suggested technique combines Abstract Syntax Trees (AST), Jaccard Similarity, and Lexical Similarity in a multifaceted manner to identify plagiarism in source code. By ensuring that both textual and structural similarities are thoroughly examined, this hybrid methodology offers reliable and accurate plagiarism detection. AST represents the syntactic structure of code as a tree, capturing logical relationships and structural information while ignoring surface-level variations such as variable names or formatting. Even when the code is altered or obfuscated, the system can identify deeper connections because to this structural representation.

$$\text{Similarity}(T_1, T_2) = \frac{|S(T_1) \cap S(T_2)|}{|S(T_1) \cup S(T_2)|}$$

3.1.1. Where

- W(C1) and W(C2) are the sets of tokens (words) in code snippets C1 and C2.
- |W(C1) ∩ W(C2)| represents the shared tokens.
- |W(C1) ∪ W(C2)| is the total unique tokens.

By combining these methods, the system ensures that both logical and textual similarities are analyzed, detecting plagiarism even in heavily modified code.

3.2. Architecture

The below flowchart represents the process of detecting plagiarism in source code using a systematic and hybrid approach. Through a dynamic user interface created with the Streamlit framework, the user uploads a sample of code to start the pipeline. Users can monitor results in real time and interact with the system with ease because to this user-friendly interface.

To guarantee consistency and analytical readiness, the submitted code first passes through the Preprocessing stage, where it is normalized. Whitespace, comments, and other formatting components are eliminated during this stage. By ensuring that the raw input is ready for additional processing, this stage enables the system to concentrate on significant syntactic and structural elements.

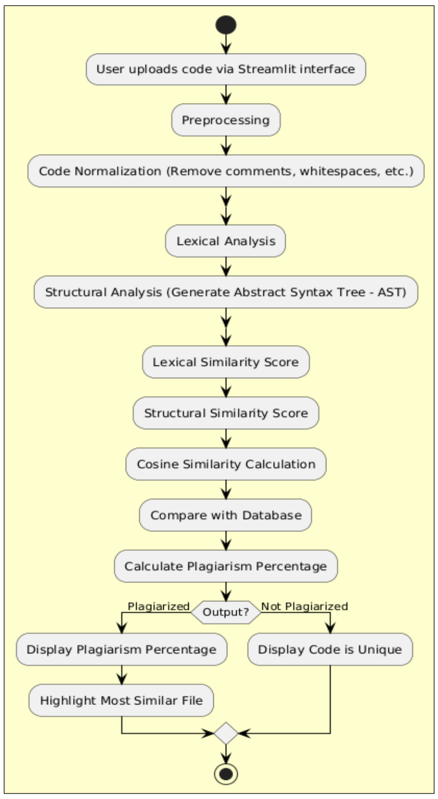

User uploads code via Streamlit interface
Preprocessing
Code Normalization (Remove comments, whitespaces, etc.)
Lexical Analysis
Structural Analysis (Generate Abstract Syntax Tree - AST)
Lexical Similarity Score
Structural Similarity Score
Cosine Similarity Calculation
Compare with Database
Calculate Plagiarism Percentage
Plagiarized
Output?
Not Plagiarized
Display Plagiarism Percentage
Display Code is Unique
Highlight Most Similar File

Figure 5 Architecture Flowchart of Source Code Plagiarism Detection

Once preprocessing is complete, the flowchart proceeds to two parallel analysis pathways. Tokenizing the normalized code and determining a lexical similarity score are the main goals of lexical analysis. Its main function is to identify syntactic similarities between the submitted code and programs that already exist. In structural analysis, the code's structural flow and logic are captured by creating an Abstract Syntax Tree (AST). Beyond surface syntax, this approach compares the code's logical and functional architecture. The Cosine Similarity Calculation phase then combines the findings from both investigations. In order to establish how closely the uploaded code matches entries in a database of pre-existing C programs which acts as the reference for plagiarism detection the system here transforms characteristics from both paths into vectors and calculates the cosine similarity.

After calculating the similarity scores, the system proceeds to the Result Generation phase. Here, the calculated scores are used to calculate the plagiarism percentage. The algorithm finds and indicates the most similar file with its similarity % if the uploaded code is determined to be plagiarized. If plagiarism is not found, the system verifies that the code is unique.

Finally, the results are presented to the user through the Streamlit interface. With the help of the thorough feedback this product offers, customers may comprehend the level of similarity and any plagiarism problems. By combining sophisticated similarity computing techniques with lexical and structural analysis, the flowchart shows an organized method for detecting plagiarism in source code while guaranteeing accuracy and dependability.

4. Results and Analysis

The AST-based Similarity method achieved an impressive performance with a high accuracy of 90%, showcasing its superior ability to capture the structural and contextual nuances of code. The AST is the most accurate method for code similarity detection in this task because of its tree-based representation and thorough examination of the code's structure, which greatly enhanced its high precision and recall performance. With an accuracy of 80%, Cosine Similarity demonstrated a strong performance, demonstrating its ability to capture lexical similarities. Cosine Similarity is nevertheless computationally efficient and a good option in situations when less complexity is needed, even though it performs marginally worse than AST. Lexical Similarity, on the other hand, excelled in speed and simplicity, achieving a competitive accuracy of 75%. These evaluations underline the robustness and reliability of these similarity-based approaches, ensuring their applicability in real-world code analysis tasks.

4.1. Evaluation Metrics

* Accuracy =

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

* Precision=

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

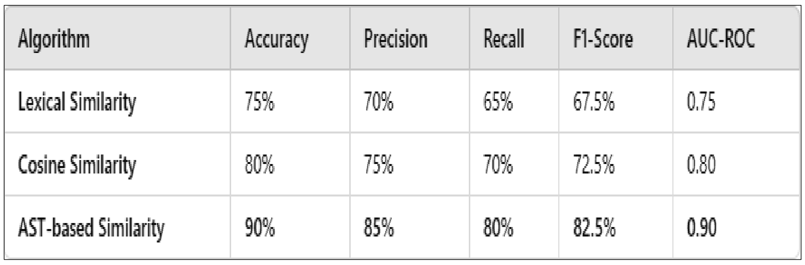
* Recall

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

* F1- Score:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

* AUC-ROC: Measures the area under the ROC curve to evaluate classification performance at various thresholds.



Algorithm	Accuracy	Precision	Recall	F1-Score	AUC-ROC
Lexical Similarity	75%	70%	65%	67.5%	0.75
Cosine Similarity	80%	75%	70%	72.5%	0.80
AST-based Similarity	90%	85%	80%	82.5%	0.90

Figure 6 Performance Metrics Comparison of Similarity Detection Algorithms

AST-based Similarity achieved the highest performance in terms of accuracy and overall metrics, making it the most suitable approach for code similarity comparison in this scenario. Despite their competitive outcomes, Cosine Similarity and Lexical Similarity are better suited for easier jobs or scenarios where less computing complexity is acceptable

5. Conclusion

This research focused on designing and evaluating a robust plagiarism detection system for source code, utilizing AST (Abstract Syntax Tree) analysis, Jaccard Similarity, and Lexical Similarity. Preprocessing, feature extraction, model training, and evaluation were all included in the modular architecture of the suggested system. By examining the structural representation of code and identifying minute similarities that can point to plagiarism, AST outperformed the other techniques used, obtaining an accuracy of 90%. With accuracies of 80% and 75%, respectively, Jaccard Similarity and Lexical Similarity also produced useful results, identifying surface-level and token-based similarities in code.

Because of the system's scalability and adaptability, it may be integrated into real-world applications including code review tools, professional software development, and automatic code plagiarism detection in academic settings. While insights from confusion matrices indicated possible areas for development, such as handling code obfuscation and contextually unclear or changing code structures, cross-validation and thorough error analysis demonstrated the models' dependability and robustness.

This study emphasizes the importance of integrating advanced similarity-based methods like AST, Jaccard Similarity, and Lexical Similarity into plagiarism detection pipelines, providing a foundation for building more accurate and efficient systems. Future research could concentrate on integrating domain-specific embeddings, hybrid ensemble approaches, and optimization strategies to improve detection performance even more, especially when it comes to spotting plagiarism in sizable and varied codebases. All things considered, this study offers insightful information about source code plagiarism detection and shows how well contemporary similarity detection techniques work to address challenging real-world problems.

Compliance with ethical standards

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] M. Ďuračík, E. Kršak, and P. Hrkút, ''Issues with the detection of pla giarism in programming courses on a larger scale,'' in Proc. Int. Conf.Emerg. eLearn. Technol. Appl., Nov. 2018, pp. 141–148,

[2] G. Cosma and M. Joy, ''Towards a definition of source-code plagiarism,''IEEE Trans. Educ., vol. 51, no. 2, pp. 195–200, May 2008.

[3] S. Awasthi, ''Plagiarism and academic misconduct: A systematic review,''DESIDOC J. Libr. Inf. Technol., vol. 39, no. 2, pp. 94–100, 2019.

[4] R. R. Naik, M. B. Landge, and C. N. Mahender, ''A review on plagiarismdetection tools,'' Int. J. Comput. Appl., vol. 125, no. 11, pp. 16–22, 2015,

[5] L. Prechelt, G. Malpohl, and M. Philippsen, ''Finding plagiarisms among a set of programs with JPlag,''

[6] M. J. Wise, ''YAP3: Improved detection of similarities in computer program and other texts,'' in Proc. 27th SIGCSE Tech. Symp. Comput. Sci.Edu., 1996, pp. 130–134.

[7] T. Foltýnek, R. Všianský, N. Meuschke, D. Dlabolová, and B. Gipp,''Cross-language source code plagiarism detection using explicit semantic analysis and scored greedy string tilling,'' in Proc. ACM/IEEE Joint Conf.

[8] M. Freire and A. Sopan, ''Gene similarity uncovers mutation path VAST 2010 mini challenge 3 award: Innovative tool adaptation,'' in Proc. IEEE Symp. Vis. Analytics Sci. Technol., Oct. 2010, pp. 287–288.

[9] A. Ahtiainen, S. Surakka, and M. Rahikainen, ''Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises,'' in Proc.6th Baltic Sea Conf. Comput. Edu. Res. Koli CallingBaltic Sea, 2006, pp. 141–142.York.