| Frontiers in Computer Science

TYPE Brief Research Report
PUBLISHED 13 June 2024
DOI 10.3389/fcomp.2024.1393723

Check for updates

OPEN ACCESS

EDITED BY
Simon K. S. Cheung,
Hong Kong Metropolitan University, China

REVIEWED BY
Paul Logasa Bogen,
Google, United States
James Harold Davenport,
University of Bath, United Kingdom
Hapnes Toba,
Maranatha Christian University, Indonesia

*CORRESPONDENCE
Kaushik Gopalan
✉ kaushik.gopalan@flame.edu.in

RECEIVED 29 February 2024
ACCEPTED 27 May 2024
PUBLISHED 13 June 2024

CITATION
Gandhi N, Gopalan K and Prasad P (2024) A Support Vector Machine based approach for plagiarism detection in Python code submissions in undergraduate settings.
Front. Comput. Sci. 6:1393723.
doi: 10.3389/fcomp.2024.1393723

COPYRIGHT
© 2024 Gandhi, Gopalan and Prasad. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

A Support Vector Machine based approach for plagiarism detection in Python code submissions in undergraduate settings

Nandini Gandhi, Kaushik Gopalan* and Prajish Prasad

School of Computing and Data Sciences, FLAME University, Pune, Maharashtra, India

Mechanisms for plagiarism detection play a crucial role in maintaining academic integrity, acting both to penalize wrongdoing while also serving as a preemptive deterrent for bad behavior. This manuscript proposes a customized plagiarism detection algorithm tailored to detect source code plagiarism in the Python programming language. Our approach combines textual and syntactic techniques, employing a support vector machine (SVM) to effectively combine various indicators of similarity and calculate the resulting similarity scores. The algorithm was trained and tested using a sample of code submissions of 4 coding problems each from 45 volunteers; 15 of these were original submissions while the other 30 were plagiarized samples. The submissions of two of the questions was used for training and the other two for testing-using the leave-p-out cross-validation strategy to avoid overfitting. We compare the performance of the proposed method with two widely used tools-MOSS and JPlag—and find that the proposed method results in a small but significant improvement in accuracy compared to JPlag, while significantly outperforming MOSS in flagging plagiarized samples.

KEYWORDS

source code plagiarism detection, Python programming, textual similarity, Abstract Syntax Trees, Support Vector Machine

1 Introduction

Code plagiarism has been an area of concern for teachers of programming courses for a long time now. Research has estimated between 50% to 80% of all undergraduate students have plagiarized a minimum of once during their academic years (Yeo, 2007). From an instructor's perspective, detecting plagiarism is difficult, particularly in large classrooms, where manual inspection of code is impractical and time-consuming. Moreover, manually detecting the similarity between code snippets is a difficult task. Hence, it is imperative that instructors employ automated means to detect plagiarism in source code, especially in large classrooms.

Source Code Plagiarism takes place when a student copies the source code of another student and submits it as their own. "A plagiarized program can be defined as a program which has been produced from another program with a small number of routine transformations and without a detailed understanding of the source code" (Parker and Hamblen, 1989). Plagiarized code is changed largely at a superficial level, but its behavior is kept very close to the original while avoiding detection in an attempt to make it look authentic. In this manuscript, we focus on source code plagiarism in the Python programming language. There has been a significant push in recent years to use the

Python language to teach introductory programming across universities globally (Jayal et al., 2011; Bogdanchikov et al., 2013; Shein, 2015; Wainer and Xavier, 2018), with Python's simplified syntax and ease of use often cited as a rationale for its choice as an introductory programming language.

Although there are techniques for code plagiarism detection in general, there is a lack of specific techniques for plagiarism detection in the Python programming language (Novak et al., 2019). Moreover, there are several challenges in detecting plagiarism, such as the presence of code transformations including code obfuscation techniques and code structure alterations (Alsmadi et al., 2014; Agrawal and Sharma, 2016).

In this manuscript, we propose a technique to combine elements of textual similarity with syntactic methods such as Abstract Syntax Trees to reliably detect source code plagiarism in Python. We use Support Vector Machine (SVM) to optimally weight different components of similarity between pairs of code files. Further details of our data collection methodology as well as specifics of our algorithm follow in subsequent sections.

2 Literature review

2.1 Plagiarism detection algorithms

There are two broad classes of plagiarism detection algorithms and tools: (1) Text-based and (2) Syntax-based analysis techniques. Text-based techniques use Python-specific features such as its indentation syntax, white space, and comments to detect plagiarism. Syntax-based techniques use Abstract Syntax Trees (ASTs) which are tree-like data structures used by compilers and interpreters to represent the syntactic structure of a program. These techniques parse the AST to detect instances of plagiarism. Sulistiani and Karnalim proposed a framework for detecting code plagiarism in Python code by using Python specific textual features and applying cosine similarity to filter out pairs with a high likelihood of substantial overlap for further scrutiny using the Running Karp-Rabin Greedy-String-Tiling (RKRGST) algorithm (Sulistiani and Karnalim, 2019). The findings show that the proposed technique improved time efficiency and enhanced sensitivity.

Syntax-based techniques employ Abstract Syntax Trees (ASTs), which are tree-like data structures used by compilers and interpreters to represent the syntactic structure of a program. Zhao et al. (2015) developed an AST-based plagiarism detection method (AST-CC) that enhances detection accuracy through hash value computations of syntax tree nodes. It is specifically tailored to arithmetic operations for error minimization, thus detecting sophisticated plagiarism patterns like block copying and renaming identifiers. Ping et al. explored an AST-based approach integrated with biological sequence matching algorithms, effectively identifying plagiarism by extracting and clustering AST feature vectors. This pinpoints "copy clusters" within programming languages like C and Java, suggesting further refinement in AST feature analysis and clustering methodologies (ping Zhang and sheng Liu, 2013). Wen et al. (2019) proposed a hybrid model combining code text and AST, where they first clean the code by removing non-essential elements. Word segmentation, word frequency statistics and weight calculation operations are carried out, after which the code fingerprint is obtained by applying Simhash and the Zhang-Shasha algorithm to compute the similarity between ASTs. Li et al. introduced an AST-based detection method capable of accurately identifying plagiarism even when plagiarists alter function declarations. This addressed the limitations of existing token-based and other syntactic plagiarism detection methods; and emphasized the need for enhanced lexical and parser grammars to accommodate new programming languages (Li and Zhong, 2010). Experimental results from all the above-mentioned studies show that these syntax-based methods are capable of detecting several common means of plagiarism in source code.

Combinations of text-based and syntax-based analysis have also been employed to detect similarities between code. Sharma et al. (2015) developed "Parikshak," using tokenization, N-Gram representation, and the Greedy String Tiling algorithm to detect plagiarism in source codes, catering to multiple programming languages and integrating text and syntax analysis. Donaldson et al. (1981) implemented algorithms that assess the sum of differences and count of similarity between assignments by offering a method to compare programming solutions beyond textual similarity by considering syntactic structure to aid in the detection process.

One aspect to consider while combining multiple features to infer code similarity is that weighting different features in a way that is effective can be a non-trivial problem. While there are several ways to solve this problem, Support Vector Machine (SVM) is quite commonly used in this context. In fact, SVM is widely used in text plagiarism detection and has also previously been used in code plagiarism detection. Awale et al. (2020) employed SVM and xgboost classifiers to assess C++ programming assignments for plagiarism, focusing primarily on string matching techniques, with particular focus on the location of braces and the commenting style on a variety of features including coding style and logic structure. Eppa and Murali (2022) compared SVM with other machine learning methods for detecting plagiarism in C programming, evaluating the efficacy of different features like syntax and code structure in identifying copied content. Huang et al. (2020) utilized multiple machine learning classifiers to analyze a series of student submissions to track code similarity between pairs of students over time, thus incorporating both code similarity and "student behavior" (patterns in student submissions over time) to provide improved estimates of likely plagiarism.

In this study we combine the use of textual or stylistic elements of the code similar to those used by Awale et al. (2020) along with syntax-based elements such as those used by Zhao et al. (2015) with the key difference that our technique utilizes customized Python-specific textual and stylistic similarity measures, whereas the studies cited previously have primarily focused on C, C++, and Java.

2.2 Plagiarism detection tools

MOSS (Measure Of Software Similarity) and JPLAG are two well-known plagiarism detection tools widely used in the field (Novak et al., 2019). MOSS is specifically designed for programming assignments and can detect similarities between code

submissions by comparing code structures, variable names, and comments (Aiken, 2023). JPLAG, on the other hand, is a tool that supports multiple programming languages and employs a tree-based algorithm to analyze code similarities (Prechelt et al., 2002).

Ahadi and Mathieson (2019) conducted a study to investigate the effectiveness of using MOSS and JPlag in detecting source code plagiarism in computer science courses. The results showed that both tools were effective in detecting plagiarism, with MOSS being slightly more accurate than JPlag. However, the study also identified several limitations of the tools, including their inability to detect plagiarism involving minor modifications to the original code. Additionally, a study compared nine plagiarism detection tools and concluded that MOSS came out on top using F-measures and precision-recall curves, while JPlag also had a high score (Heres and Hage, 2017). For our work, we use MOSS and JPlag as benchmarks; and adopt F1 score, precision and recall as our evaluation metrics to measure the performance of our proposed plagiarism detector against them.

2.3 Ethical considerations in using plagiarism detection tools

Noynaert (2005) discusses the challenges and limitations of plagiarism detection software, emphasizing the necessity of manual investigation to complement these tools. While software like Turnitin can help identify potential plagiarism, it often requires additional manual verification by instructors to confirm instances of plagiarism and avoid false positives. It highlights the need for a balanced approach to plagiarism detection, combining both automated and manual methods. Mozgovoy et al. (2010) explore the limitations of existing automatic plagiarism detection systems and their inability to detect sophisticated forms of plagiarism- such as extensive paraphrasing or the use of technical tricks to evade detection. They discuss the evolution of plagiarism due to the accessibility of electronic texts and the challenges faced by detection tools in adapting to new forms of plagiarism. The paper proposes enhancing future systems with natural language processing and information retrieval technologies to overcome these limitations. It calls for a balance between technical and ethical considerations in the deployment of plagiarism detection software.

Brinkman (2013) addresses the ethical concerns surrounding the use of plagiarism detection systems, focusing on student privacy rights. It critiques the permanent archiving of student works by services like TurnItIn, discussing how this practice could lead to potential harm to students in the future. The paper calls for educators to fully inform students about the workings of plagiarism detection services and understand the implications of archiving student works.

3 Data collection

For this study, we selected four distinct programming questions that are frequently used as introductory exercises in Python programming coursework. The questions were as follows:

Question A (Fizzbuzz): A program, when given a number, returns “Fizz” if the number is a multiple of 3, “Buzz” if the number is a multiple of 5, and “FizzBuzz” if the number is a multiple of both 3 and 5.

Question B (Rock Paper Scissors): A program that simulates the game of Rock, Paper, and Scissors between a player and the computer.

Question C (Mean value): A program that calculates the mean values of an array of integers, input by the user, until the user enters -99, indicating the end of the array.

Question D (First n prime numbers): A program that generates a list of the first n prime numbers.

These questions were selected due to their relative simplicity, making them representative of the types of programming exercises commonly found in an introductory course on Python.

The dataset for this study was collected from 45 students recruited through convenience sampling from multiple universities across India. All participants were informed about the purpose of the study; and were explicitly informed that they had the option to opt out of the study if they did not wish to participate. To ensure privacy, we assigned a unique identifier code to each participant, keeping their personal information independent from the data. Out of the 45 participants, 15 participants submitted four original programs (one for each question), resulting in a total of 15 original code samples per question, and 60 original code samples in total. After collecting the original code samples, we instructed two people to plagiarize each of these code samples, resulting in a total of 30 plagiarized code samples per question, and 120 plagiarized code samples in total.

During the process of providing a plagiarized sample, the participants were asked to modify samples in a manner that would resemble a typical approach to plagiarizing code for submission in an academic environment. They were asked to imagine a scenario where they were pressed for time before a submission deadline and were provided a sample original submission to imitate. We find that many used common tactics such as changing variable names, altering the order of statements, and adding or removing comments, among others. Other volunteers submitted code that was essentially identical to the original. By incorporating samples with various degrees of obfuscation, we aim to generate a realistic sample dataset, and one representative of a university course. This allows us to test the effectiveness of our proposed model under conditions that closely resemble real-world occurrences of Python source code plagiarism.

4 Methodology

The proposed algorithm is designed to compare pairs of Python files and identify similarities in code. As mentioned earlier, one way that plagiarism can manifest is as similarities in the text, such as similar comments, similar or identical variable names and similar strings. In this study, we use a combination of both textual and logical similarity to flag potential plagiarism. Figure 1 illustrates the plagiarism detection methodology. Each pair of files is passed to a model which is based on seven indicators of similarity- Comments, Loops, Program Level Syntactic Similarity (PLS Similarity), Variables, Functions, Strings, and Vertical Similarity,

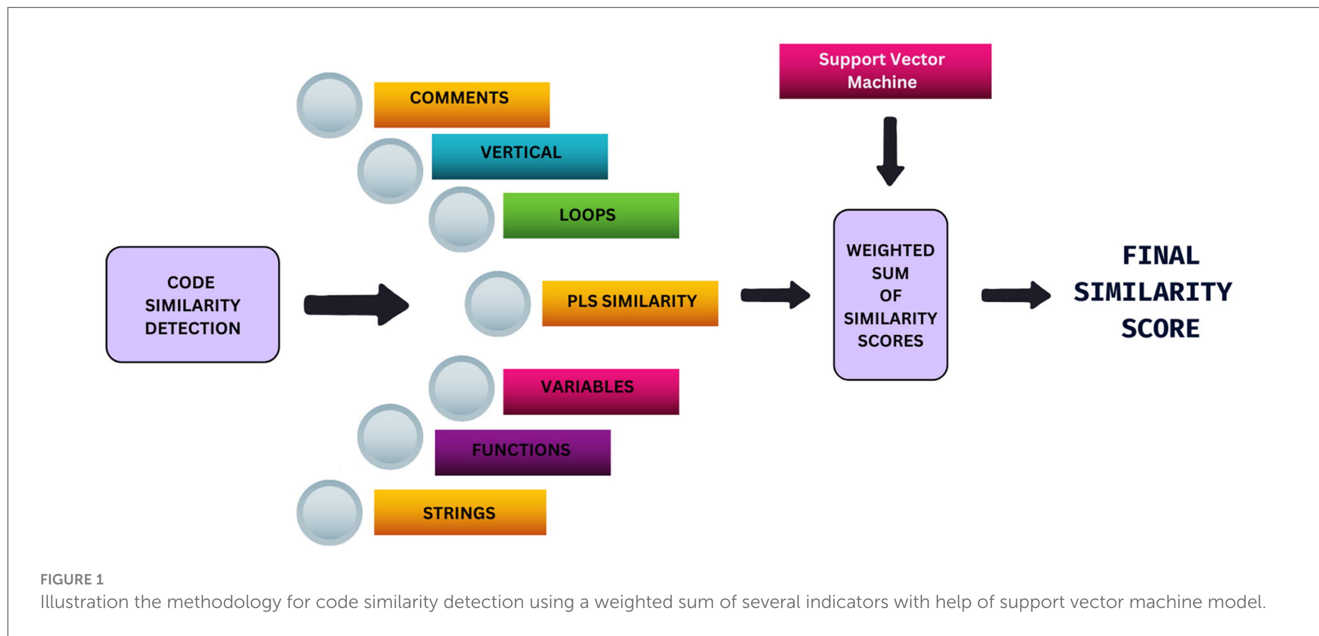

FIGURE 1
Illustration the methodology for code similarity detection using a weighted sum of several indicators with help of support vector machine model.

TABLE 1 Weights assigned to each component by SVM.

Component	Weight	Typical range for original	Typical range for plagiarized
Program level syntax	5.685	0.396–0.634	0.759–1
Vertical structure	1.182	0.445–0.719	0.626–0.996
Function	1.034	0–0.480	0.120–0.909
String	0.950	0–0.414	0.189–0.959
Variable	0.680	0–0.314	0–0.678
Comment	-0.624	0–0.079	0–0.539
Loop	0.376	0–0.460	0.007–0.874

that measure different characteristics of code similarity, covering the functional, textual and structural aspects. The similarity scores from each indicator are combined using their weighted sum, which is given to us by the SVM. This combined score, which can then be tweaked based on user preference, is our final similarity score. In this section, we explain each of the indicators used before we detail the procedure used to combine them optimally to provide a unified similarity score.

Each pair of the code files is parsed into a pair of abstract syntax tree (AST) objects. ASTs are hierarchical tree-like representations of the code that captures its structure and functionality. They are constructed by recursively breaking down the code into its constituent parts, such as statements, expressions, and function calls.

1. Vertical structure scoring: Vertical structure comparison is done using the Levenshtein distance algorithm on encoded vertical structures. We start by encoding the vertical structure of each file into a sequence of characters. An empty line is encoded as "0", a comment as "#", and all other executable lines as "1". The Levenshtein distance algorithm is applied to the two encoded strings. This algorithm calculates the minimum number of insertions, substitutions, and deletions that are required to transform one sequence into another. This distance, normalized by the length of the longer sequence, represents the similarity between the structures of the code files.
2. Comments scoring: The similarity between comments of plagiarized files is detected using a combination of techniques-comment extraction, preprocessing and similarity calculation. Comments are isolated from the code using regular expressions, extracting lines that start with # or are enclosed within three single or double quotes. These lines are compiled into separate lists for each file. The lines are then preprocessed, which involves converting all comments to lowercase, and removing non-alphanumeric characters, to get a more accurate comparison. Next, the similarity score calculation is carried out using the Term Frequency-Inverse Document Frequency (TF-IDF) approach to represent comments as numerical vectors. Each word in the comment is assigned a weight according to its frequency in that comment and its inverse frequency across all comments. Once we have the TF-IDF vectors for each comment, the cosine similarity metric can be used to measure the similarity. The maximum similarity value for a comparison between each file is recorded, and the most similar comment in the second code for each comment in the first code file is identified. Using this, we can calculate the average maximum comment similarity score between every pair of code files.
3. String similarity: The content of each file is scanned to identify strings, and regular expressions are used to extract them. Text, excluding the comments, that are enclosed in either single or double quotes are captured. The similarity between the two sets of strings is then calculated using a similarity metric, and the score is returned.
4. Variables scoring: The variables are extracted from the ASTs by iterating over the targets of the assignment nodes and identifying the variable names. The Jaccard similarity coefficient is then

calculated by dividing the intersection of the variable sets by their union. This coefficient gives us the overlap between the variables of the two code files as the variable similarity score.

5. Functions scoring: The functions of every code file are compared with the other, and the comparison is done on three different criteria: comparison of the number of functions, function names, function arguments and function calls. Once these are extracted from the function, they are compared pairwise using a similarity-matching algorithm to give each pair an overall similarity score.
6. Loops scoring: After getting the ASTs, the loop nodes from each code file for "FOR" and "WHILE" are extracted by filtering relevant nodes of the AST. For each loop node found in the first code file, a matching loop is searched for in the second code file. By analyzing the type of loop and other relevant characteristics, a similarity score is computed to quantify the resemblance between the loops of both files.
7. Program level syntactic similarity scoring:. The nodes from each AST representing different elements of the code like classes, names, values, statements, expressions, or functions, among others, are extracted. The nodes of the two code files are compared using the Levenshtein distance algorithm to generate a similarity score. This score is indicative of the structural and syntactic similarity between the files and captures commonality in the organization and composition of the code.

While there are several ways to combine the different indicators of similarity to generate a single similarity score that is required to flag potentially plagiarized pairs of submissions, it is fairly standard and intuitive to use a linear combination of the different indicator scores to derive a final score. However, the relative weighting of the different components plays a significant role in determining the efficacy of the final score in accurately designating likely code plagiarism. In this study, we use SVM to derive optimal coefficients for combining all the similarity indicators into a final similarity score. The SVM aims to find an optimal hyperplane that separates data points of different classes by maximizing the margin between them. As mentioned previously we designate the decision boundary between the two classes ("likely original" and "potentially plagiarized") to be linear and hence use the linear kernel for the SVM. Since SVM is a supervised learning method, we use the submissions for two of the four problems to train the SVM and the remaining two to test the efficacy of the proposed method. Further, to avoid model over-fitting, we employ a leave-p-out cross-validation strategy. Out of the 15 original submissions, in every iteration we choose combinations of five originals and samples plagiarized from them to form the validation set and the remaining samples to form the training sets. Thus given five originals and two plagiarized from each, there are $\binom{15}{2}$ samples per question in the test set, resulting in 210 (105 each for two questions) pair samples in the test set for every iteration. This results in 1,770 remaining pairs that form the training set. Since there are $\binom{15}{5} = 3{,}003$ ways of selecting five values from a set of 15, the model is trained 3,003 times in the cross-validation exercise. This design ensures that none of the participants whose submissions form the test set in any iteration have their samples—or samples plagiarized from them—to be a part of the training set. This prevents the model from over-fitting based on the style of any particular participant.

Direct plagiarism comparisons involve comparing a single plagiarized file with its corresponding original file, and indirect plagiarism comparisons involve comparing two plagiarized files which were plagiarized from the same source. We classified both of these as instances of plagiarism, based on the assumption that any form of replication, whether directly from the original file or indirectly by a shared origin, falls under the umbrella of "potential plagiarism". Thus, our training set in each iteration contains 60 plagiarized pairs out of 1,770 and the test set comprises of 30 plagiarized pairs out of 210. To address this imbalance in the different classes, we configure the SVM model to "balanced" mode; i.e., weights are adjusted with a factor that is inversely proportional to class frequencies in the input data (King and Zeng, 2001). This ensures that the minority class ("potentially plagiarized") receives greater consideration, thereby enhancing the overall predictive performance. We compare the classification results obtained from the proposed model with standard alternatives and detail the comparisons in the next section.

5 Results

In the previous section, we detailed a similarity detection model that incorporates both textual and syntactic similarity. Our algorithm was evaluated for its ability to differentiate between original and plagiarized Python code files. The Proposed model was trained to optimize the weights for seven components indicative of code similarity: Program Level Syntax, Vertical, Function, String, Variable, Comment, and Loop. The trained model assigned varying levels of importance to each component, which are summarized in Table 1. These weights are the average of the optimal weights obtained from training the model 3,003 times, plus or minus the standard deviation. We find that the SVM model assigned higher weights to those components which had a wider separation between original and plagiarized samples, which is as expected.

The weight assigned to Program Level Syntax, at 5.685, was the highest, indicating its strong influence on the detection of plagiarism. This suggests that structural and syntactic features play an important role in distinguishing between original and plagiarized code. The range of scores for original code in this component was between 0.396 and 0.634, whereas plagiarized code showed a higher range of 0.759–1, aligning with the expectation that plagiarized code would have more syntactic similarities than original work. Vertical structure comparison, which focuses on the layout of the code, also received a high weight of 1.182. The typical range for plagiarized code was notably higher than that for original code, showcasing the model's capability to discern the alteration of code structure as an indicator of potential plagiarism. Functions and strings within the code were also significant components, with weights of 1.034 and 0.950, respectively. The range of scores for functions in original code was 0–0.480, whereas in plagiarized code it was 0.120–0.909, demonstrating a broader overlap in the use of functions among plagiarized files. Variables and loops were given moderate importance with weights of 0.680 and 0.376, respectively. The overlap of variable names in plagiarized code was higher, with scores reaching up to 0.678, compared to 0.314 in original code.

The component representing comments was assigned a negative weight of -0.624. This finding indicates that, within the

scope of our dataset and model, a higher similarity in comments did not add significantly to the skill of the classification model and was inversely related to the classification of code as plagiarized. This is a consequence of the fact that the value of the "Comment scoring" index is 0 for 95% of the samples in the original samples and in 80% of the samples in the plagiarized samples. This large imbalance in the distribution of values is a challenge in incorporating this feature in the linear SVM-based classification model.

Continuing from the examination of the weights assigned to the components of our plagiarism detection system, the performance metrics of the proposed model were evaluated and compared with two established plagiarism detection systems, MOSS and JPlag. The results are presented in Figure 2, which outlines the precision and recall for both original and plagiarized code, as well as the overall accuracy of each algorithm.

The proposed algorithm demonstrated high precision in identifying original code (Precision_Originals) with a value of 0.978, with similar performance as MOSS and JPlag, which scored 0.958 and 0.945, respectively. The proposed algorithm along with JPLAG also had a high recall value for the "likely original" class (0.991 and 0.974 respectively), while the recall for MOSS is substantially lower at 0.790. This is also reflected in the Precision values for the "potentially plagiarized" class, where MOSS has a precision value of 0.387 whereas approximately 95.5% of the values tagged by the proposed algorithm as "potentially plagiarized" are done so correctly. JPLAG also exhibits high precision of 0.837 among the samples labeled as "potentially plagiarized". In absolute terms, the proposed model results in approximately 2 false positives and 4 false negatives out of 210 samples averaged over each training/validation split in the cross-validation exercise. JPLAG has the next-best performance with 5 false positives and 10 false negatives, followed by MOSS which has 38 false positives and 6 false negatives. Finally, we find that the proposed model achieved an accuracy of 0.973 (204 out of 210 classified correctly), while MOSS and JPlag recorded scores of 0.790 (166 out of 210) and 0.929 (195 out of 210) respectively.

To evaluate whether the differences observed in the performance of the proposed model relative to MOSS and JPLAG were statistically significant, a repeated measures Analysis of Variance (ANOVA) followed by Tukey's Honestly Significant Difference (HSD) *post-hoc* test was conducted as seen in Table 2. ANOVA helps in identifying whether there are any overall differences among groups; however, it does not specify where these differences lie. Therefore, following the ANOVA tests, we conducted Tukey's Honestly Significant Difference (HSD) tests. These tests are used for pairwise comparisons between groups, providing insights into which specific models differ from each other.

The results from the ANOVA were highly significant at <0.001 level across all metrics, signaling substantial differences in model performances. The F-statistics, a measure of the ratio of variance between the groups to the variance within the groups, were also high. For instance, the F-statistic for the Accuracy metric reached 58,712.6662, with a corresponding p-value effectively at zero. This indicates a strong rejection of the null hypothesis, which posits that all models perform equivalently across the tested metrics. With the ANOVA establishing significant differences, the subsequent Tukey's HSD tests allowed us to examine these differences on a model-to-model basis. This test also resulted in p-values of < 0.001 for the mean differences between the proposed model and both JPLAG and MOSS for each of the 5 parameters that were considered. While the improvements in the proposed model over JPLAG in particular are generally small (with the exception of the recall value for the plagiarized class), the statistical tests suggest that they are statistically significant. The improvements over MOSS are substantial for each of the parameters considered.

For the cases where the proposed algorithm misclassified the samples—i.e., the cases where the proposed algorithm either resulted in a false positive or a false negative- we performed a crude error analysis to understand the reason for these model failures. We calculated the mean values of each of the 7 model components separately for the false positives as well as false negatives, as shown in Table 3. For the false negatives—i.e. cases where plagiarized samples are classified as "likely original"—the components with low mean values are primarily responsible for the misclassification since they cause the overall similarity score to be low. We find that the "Comments scoring", "Functions scoring", "String similarity" and "Variable scoring" parameters are the most responsible for the false negatives. This likely reflects the cases where the plagiarized samples contain obfuscation techniques such as changing variable names, comments, etc. in order to escape detection. Such obfuscation techniques would affect the parameters specified above more than logic-based parameters such as "Program Level Syntax" or "Loop scoring". For the false positives—i.e. cases where original samples are classified as "potentially plagiarized"- the components with high mean values are primarily responsible for the misclassification since they cause the overall similarity score to be high. We find that "Program Level Syntax", "Function scoring", "Loop scoring", and "Vertical structure scoring" are most responsible for false positives. These likely reflect cases where multiple people arrive at highly similar programming logic to solve the given exercise without colluding with each other.

6 Discussion

The proposed Python plagiarism detection model analyzes code similarity using a variety of similarity indicators. The model leverages logical heuristics and an analysis of multiple code components, including comments, loops, AST similarity, variables, functions, strings, and vertical similarity. The results demonstrate that the model achieves a small, but significant improvement in performance over existing plagiarism detection tools such as JPlag and MOSS in terms of accuracy, effectiveness and ability to detect plagiarism. There was a notable improvement in the evaluation metrics, with the proposed model classifying 204 out of 210 samples (97%) correctly, compared to 195 (93%) for JPlag and 166 (79%) for MOSS.

The improved performance can be explained by the design of the proposed model, which is designed exclusively for Python, leveraging the characteristics and features of the language. MOSS and JPlag, on the other hand, are primarily built for other languages such as Java and C, which might limit their effectiveness when applied to Python code. However, it is important to identify the limitations of this study. One limitation is the size of the dataset

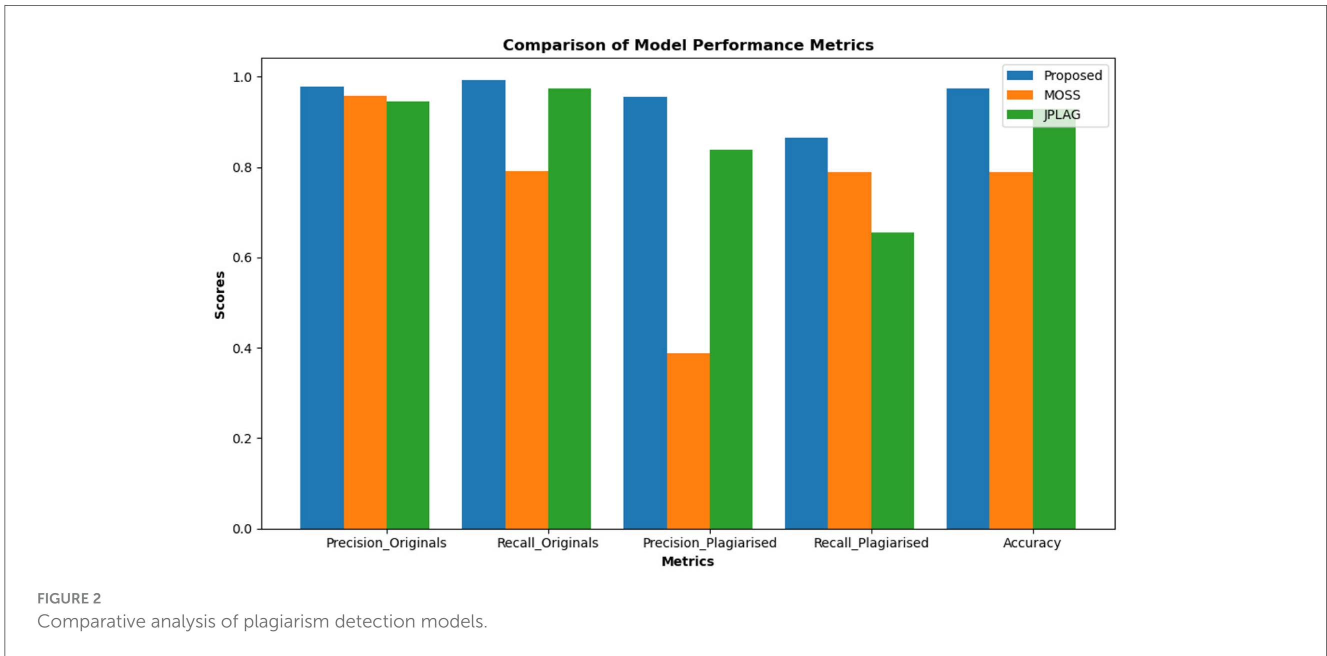

FIGURE 2
Comparative analysis of plagiarism detection models.

TABLE 2 ANOVA and Tukey's HSD test results.

Metric	ANOVA		Tukey's HSD test			Reject null
	F-statistic	*P*-value	Comparison	Mean diff.	*P*-value	
Precision_0	5,943.9049	<0.001	Proposed—JPlag	0.0333	<0.001	Yes
			Proposed - MOSS	0.0203	<0.001	Yes
Recall_0	56,440.8843	<0.001	Proposed—JPlag	0.0174	<0.001	Yes
			Proposed - MOSS	0.2015	<0.001	Yes
Precision_1	31,952.8752	<0.001	Proposed—JPlag	0.1178	<0.001	Yes
			Proposed - MOSS	0.5679	<0.001	Yes
Recall_1	5,856.8262	<0.001	Proposed—JPlag	0.2086	<0.001	Yes
			Proposed - MOSS	0.0753	<0.001	Yes
Accuracy	58,712.6662	<0.001	Proposed—JPlag	0.0447	<0.001	Yes
			Proposed - MOSS	0.1835	<0.001	Yes

TABLE 3 Mean values of each feature in erroneous estimates.

	Comment	Vertical structure	Function	Program level syntax	String	Variable	Loop
False negatives	0.08	0.67	0.17	0.65	0.15	0.06	0.45
False positives	0.02	0.58	0.64	0.71	0.20	0.30	0.67

used for training and evaluation. Validating and testing the model on a larger and more diverse dataset would be useful to assess its performance in different scenarios. Future research could include the integration of deep learning algorithms to enhance the model's capabilities and improve its performance.

As computer science instructors, we (the 2nd and 3rd authors) find that our instruction and assessment design are influenced by the threat of undetected plagiarism and its influence on academic integrity. Specifically, while autograders are widely available and convenient to use, these generally do not possess the ability to accurately detect plagiarism. On the other hand, manual grading is generally reliable in detecting flagrant instances of plagiarism but is extremely tedious and time-consuming for anything but the smallest class sizes. This additional burden on instructor time disincentivizes the allocation of take-home coding assignments in favor of shorter quizzes or in-class tests. Any improvement in source code plagiarism detection models can reduce the need for such tradeoffs.

However, there are also serious implications of falsely labeling original code as plagiarized. Hence, a more balanced approach is needed, where findings from plagiarism tools are complemented by instructors' personal knowledge of individual student contexts and careful investigation of the flagged submissions. Students should also be instructed on the academic integrity policies of the course beforehand, so that they are aware of the implications of their actions.

A recent study found that existing tools for plagiarism detection in programming assignments often fail to detect AI-generated code, raising concerns about their effectiveness in detecting plagiarism in the age of AI (Finnie-Ansley et al., 2022). Future work could aim to detect AI plagiarism by incorporating advanced technologies like Machine Learning and Natural Language Processing to effectively identify instances of code generated by AI models such as ChatGPT.

In conclusion, our proposed plagiarism detection model demonstrates a high level of precision and surpasses benchmark models in terms of accuracy and effectiveness, making it a noteworthy addition to efforts toward preventing code plagiarism in academic settings.

Data availability statement

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

Ethics statement

The studies involving humans were approved by FLAME University IRB Committee. The studies were conducted in accordance with the local legislation and institutional requirements. Written informed consent for participation was not required for this study in accordance with the national legislation and the institutional requirements.

Author contributions

NG: Methodology, Software, Validation, Writing – original draft, Writing – review & editing. KG: Conceptualization, Software, Supervision, Writing – original draft, Writing – review & editing. PP: Investigation, Software, Writing – original draft, Writing – review & editing.

Funding

The author(s) declare that no financial support was received for the research, authorship, and/or publication of this article.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

Agrawal, M., and Sharma, D. K. (2016). "A state of art on source code plagiarism detection," in *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)* (Dehradun: IEEE), 236–241. doi: 10.1109/NGCT.2016.7877421

Ahadi, A., and Mathieson, L. (2019). "A comparison of three popular source code similarity tools for detecting student plagiarism," in *Proceedings of the Twenty-First Australasian Computing Education Conference*, 112–117. doi: 10.1145/3286960.3286974

Aiken, A. (2023). *A System for Detecting Software Similarity*. Available online at: https://theory.stanford.edu/aiken/moss/ (accessed June 8, 2023).

Alsmadi, I. M., AlHami, I., and Kazakzeh, S. (2014). *Issues Related to the Detection of Source Code Plagiarism in Students Assignments*. San Antonio, TX: Texas A&M University.

Awale, N., Pandey, M., Dulal, A., and Timsina, B. (2020). Plagiarism detection in programming assignments using machine learning. *J. Artif. Intellig. Capsule Netw.* 2, 177–184. doi: 10.36548/jaicn.2020.3.005

Bogdanchikov, A., Zhaparov, M., and Suliyev, R. (2013). "Python to learn programming," in *Journal of Physics: Conference Series* (Bristol: IOP Publishing), 012027.

Brinkman, B. (2013). An analysis of student privacy rights in the use of plagiarism detection systems. *Sci. Eng. Ethics* 19, 1255–1266. doi: 10.1007/s11948-012-9370-y

Donaldson, J. L., Lancaster, A.-M., and Sposato, P. H. (1981). A plagiarism detection system. *SIGCSE Bull.* 13, 21–25. doi: 10.1145/953049.800955

Eppa, A., and Murali, A. (2022). "Source code plagiarism detection: a machine intelligence approach," in *2022 IEEE Fourth International Conference on Advances in Electronics, Computers and Communications (ICAECC)* (Bengaluru: IEEE), 1–7.

Finnie-Ansley, J., Denny, P., Becker, B. A., Luxton-Reilly, A., and Prather, J. (2022). "The robots are coming: Exploring the implications of openai codex on introductory programming," in *Australasian Computing Education Conference* (Association for Computing Machinery), 10–19.

Heres, D., and Hage, J. (2017). "A quantitative comparison of program plagiarism detection tools," in *Proceedings of the 6th Computer Science Education Research Conference*, 73–82.

Huang, Q., Song, X., and Fang, G. (2020). "Code plagiarism detection method based on code similarity and student behavior characteristics," in *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)* (Dalian: IEEE), 167–172.

Jayal, A., Lauria, S., Tucker, A., and Swift, S. (2011). Python for teaching introductory programming: a quantitative evaluation. *Innovat. Teach. Learn. Informat. Comp. Sci.* 10, 86–90. doi: 10.11120/ital.2011.10010086

King, G., and Zeng, L. (2001). Logistic regression in rare events data. *Polit. Analy.* 9, 137–163. doi: 10.1093/oxfordjournals.pan.a004868

Li, X., and Zhong, X. J. (2010). "The source code plagiarism detection using ast," in *2010 International Symposium on Intelligence Information Processing and Trusted Computing* (Chiang Mai: IEEE), 406–408.

Mozgovoy, M., Kakkonen, T., and Cosma, G. (2010). Automatic student plagiarism detection: future perspectives. *J. Educ. Comp. Res*. 43, 511–531. doi: 10.2190/EC.43.4.e

Novak, M., Joy, M., and Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Trans. Comp. Educ*. (*TOCE)* 19, 1–37. doi: 10.1145/3313290

Noynaert, J. E. (2005). "Plagiarism detection software," in *Midwest Instruction and Computing Symposium*.

Parker, A., and Hamblen, J. O. (1989). Computer algorithms for plagiarism detection. *IEEE Trans. Educ*. 32, 94–99. doi: 10.1109/13.28038

ping Zhang, L., and sheng Liu, D. (2013). "Ast-based multi-language plagiarism detection method," in *2013 IEEE 4th International Conference on Software Engineering and Service Science* (Beijing: IEEE), 738–742.

Prechelt, L., Malpohl, G., Philippsen, M. (2002). Finding plagiarisms among a set of programs with jplag. *J. Univers. Comput. Sci*. 8:1016. doi: 10.5445/IR/542000

Sharma, S., Sharma, C. S., and Tyagi, V. (2015). "Plagiarism detection tool "parikshak"," in *2015 International Conference on Communication, Information* & *Computing Technology (ICCICT)* (Mumbai: IEEE), 1–7.

Shein, E. (2015). Python for beginners. *Commun. ACM* 58, 19–21. doi: 10.1145/2716560

Sulistiani, L., and Karnalim, O. (2019). Es-plag: efficient and sensitive source code plagiarism detection tool for academic environment. *Comp. Appl. Eng. Educ*. 27, 166–182. doi: 10.1002/cae.22066

Wainer, J., and Xavier, E. C. (2018). A controlled experiment on python vs c for an introductory programming course: Students outcomes. *ACM Trans. Comp. Educ*. (*TOCE)* 18, 1–16. doi: 10.1145/3152894

Wen, W., Xue, X., Li, Y., Gu, P., and Xu, J. (2019). Code similarity detection using ast and textual information. *Int. J. Performab. Eng*. 15:2683. doi: 10.23940/ijpe.19.10.p14.26832691

Yeo, S. (2007). First-year university science and engineering students understanding of plagiarism. *High Educ.Res. Dev*. 26:199–216. doi: 10.1080/072943607013 10813

Zhao, J., Xia, K., Fu, Y., and Cui, B. (2015). "An ast-based code plagiarism detection algorithm," in *2015 10th International Conference on Broadband and Wir eless Computing, Communication and Applications (BWCCA)* (Krakow: IEEE), 178–182.