Grenze International Journal of Engineering and Technology, June Issue

CopyCatch: A Multi-Algorithm Approach for Detecting Code Plagiarism in Academic Submissions

Pruthviraj Mahadev Bhosale[1], Tushar Sadashiv Kapase[2], Prathamesh Pravin Thakur[3], Kunal Ganesh Waghmare[4], M.S. Dabade[5] and A.R. Surve[6]
[1-6]Department of Computer Science and Engineering, Walchand College of Engineering, Sangli ,Maharastra, India
Email: pruthviraj.bhosale@walchandsangli.ac.in, tushar.kapase@walchandsangli.ac.in, prathamesh.thakur@walchandsangli.ac.in, kunal.waghmare@walchandsangli.ac.in, manisha.dabade@walchandsangli.ac.in, anil.surve@walchandsangli.ac.in

***Abstract*— Academic integrity is a critical aspect of programming-based education, where assignments are widely used for evaluation. While working on this problem, we noticed that detecting plagiarism in source code is not straightforward, as students can easily modify copied code using simple techniques such as renaming variables or changing formatting.**
This work, includes development of CopyCatch, a digital plagiarism or web plagiarism detection system such combines multiple similarity detection techniques. Instead of relying on a single method, we experimented with approaches such as Jaccard Similarity, TF-IDF based Cosine Similarity, Levenshtein Distance, and statistical divergence measures. Each of these methods captures a different aspect of similarity, which helped us achieve more consistent results.One important observation during our implementation was that supporting modern formats like Jupyter Notebooks adds complexity, as they contain both code and explanatory text. We added preprocessing and OCR-based extraction for non-text formats like PDFs and images to fix this.
While we were working on the system, we noticed that using just one similarity measure often led to results that weren't consistent. This made us want to use more than one method to make the system more reliable overall. In general, CopyCatch wants to give teachers a way to use and grow their approach. For future work, planned to explore deep learning–based methods to detect deeper semantic similarities between programs.

***Index Terms*— Plagiarism Detection, Source Code Analysis, Jaccard Similarity, Cosine TF-IDF, Levenshtein Distance, Jensen–Shannon Divergence, OCR, Academic Integrity.**

I. INTRODUCTION

Due to the rising popularity of computer science education and online education platforms, programming tasks have become one of the best ways to evaluate students. But there is an increase in the number of incidents of plagiarism, wherein the student plagiarizes the code from other students or online and then changes it just enough to make it appear as if it was created by him/her. The problem of code plagiarism lies in the fact that crucial details are often lost during minor modifications in the code such as changes in the order of comments, statements, and identifier names while keeping the underlying logic intact. This leads to the evaluation being time-consuming and inefficient, especially when the volume of students is very high.

Grenze ID: 01.GIJET.12.2.393_24

MOSS and JPlag are two popularly used existing tools for plagiarism detection that can effectively detect structural similarities in two programs. However, this approach usually fails in their design of detecting logical similarities in the underlying structure and handling Jupyter Notebooks, which include code and explanation.
In this regard, we propose a web-based code plagiarism detection tool known as CopyCatch, which is based on several similarity detection algorithms.
This is then followed by an analysis based on their structure, statistics, and lexical content to detect cases of plagiarism, which turned out to be a reliable method. The tool has been structured to be scalable and easy-to-use for a number of formats of files. Thus, the outputs from this tool for detecting plagiarism will be accurate and reliable.

II. Literature Study

Plagiarism detection in program code is a meaningful and considerable area of research in these years, which has resulted in the development of various tools. Researchers have tried different approaches to make the plagiarism detection process more efficient. The use of Jupyter notebooks has increased in recent times. However, plagiarism detection using the Jupyter Notebook is a challenging task. Many researchers have proven that the conventional approach is not efficient in detecting the logical plagiarism between the code.
To overcome the limitations of the conventional approach, recent research is focused on using a combination of different approaches for plagiarism detection.
The table below shows a summary of recent research papers on the topic.

TABLE I

Paper / Author	Year	Methodology	Findings / Observation
Source Code Plagiarism Detection in an Educational Context: A Literature Mapping – Aniceto, Rodrigo C., et al.	2021	Systematic literature mapping of plagiarism detection techniques	Highlights strengths and limitations of text, token, and semantic approaches; suggests hybrid methods
A Tool for Detecting Similarities in Jupyter Notebooks Used as Assessment Reports – Bajaj, Nikesh, et al.	2025	Notebook structure analysis (code + markdown)	Detecting plagiarism in notebooks is complex due to mixed content formats
AI Based Plagiarism Checking – Molnár, György & József Cserkó	2022	AI/ML-based similarity detection	AI models improve detection of logical similarities beyond syntax
Methodology for Detecting Similarity of Program Code Based on Combining Algorithms – Novotochinov, Maksim E., et al.	2024	Hybrid algorithm combining multiple techniques	Improves robustness and accuracy compared to single approaches
Explanation in Code Similarity Investigation – Karnalim, Oscar	2021	Explainable plagiarism detection system	Provides interpretable results for better understanding by instructors
Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection – Xu, Xiaojun, et al.	2017	Graph-based neural embeddings	Captures deep structural similarities across different platforms
Learning Graph-Based Code Representations for Source-Level Functional Similarity Detection – Liu, Jiahao, et al.	2023	Graph-based code representation learning	Enhances functional similarity detection using graph structures
Combining Graph-Based Learning with Automated Data Collection for Code Vulnerability Detection – Wang, Huanting, et al.	2020	Graph learning + automated dataset generation	Improves detection accuracy using large-scale structured data

III. Problem Statement

The detection of plagiarism in computer-based tasks still represents a complex issue in the current environment. While carrying out this project, a number of problems were discoveredFirst, it is difficult to identify stolen content since one can change their program just a little and hide it. Secondly, existing techniques concentrate mainly on structure matching and ignore the logic.
Moreover, another challenge identified was associated with the fact that modern formats, such as Jupyter Notebook files, PDFs, and pictures cannot be used effectively in this regard. In fact, valuable information may be present in those materials, but it is not analyzed correctly.Scalability presents another concern for institutions that regularly receive large volumes of submitted content. Therefore, there is a need for an efficient solution to this problem that would also scale well.
In this case, it should be effective, scalable, support multiple formats, and produce reliable results.

IV. METHODOLOGY

The figure below shows the overall method that was used.

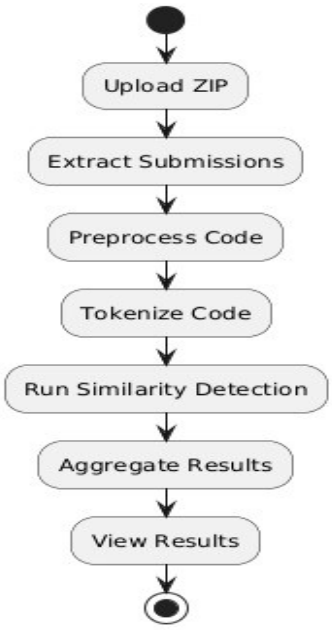

Figure 1: Overall Methodology

A. System Overview

CopyCatch is a web-based application developed using Flask framework. In developing the application, we prioritized creating an efficient workflow which can perform all plagiarism checks automatically. The reason we opted to use Flask for the development process is its lightness and simplicity, enabling us to experiment easily with various components without introducing additional complexities.

B. Workflow

The flow in our system is as follows:

- File Upload: The next step involves the uploading of a ZIP file containing all of the files from the respective students. This particular approach was chosen because it allows for working with multiple files simultaneously and hence simplifies the entire process. We made certain that our system does not make errors while processing files of any size. From our experience in dealing with this problem, we conclude that this step is the most crucial stage of our pipeline.
- File Decompression: Having uploaded the necessary ZIP file, all of the files contained therein get decompressed and divided according to their type. During the development of this stage, it became clear that it is essential to work properly with different kinds of files, so that they can undergo further processing. We used the corresponding filters that would recognize only the files in a correct format. This measure helped prevent unnecessary errors at a later stage.
- File Parsing: Following the extraction process, each file undergoes parsing based on its file format. In previous discussions, it was shown that .py and .ipynb files were immediately parseable since their respective code segments were easily readable. In contrast, processing PDFs and images required extracting textual data, which made the parsing process difficult. Hence, OCR technology was used to convert the non-textual data into analyzable information. From our experiments, it appeared that this modification significantly improved the system's flexibility towards various inputs.
- Preprocessing: In this part of the study, it was discovered that preprocessed data that still contained unnecessary elements such as comments and extra whitespaces could greatly affect similarity computations. Thus, normalization techniques were employed to standardize the data.

- Tokenization: In this phase, the source code gets converted into tokens which include keywords, identifiers, etc. We realized that segmenting the code helps to detect patterns and similarities while implementing the tokenization phase. It was also noticed that this phase reduces the impact of format changes as the emphasis shifts to logic-based rather than format-based code structure. According to us, the importance of proper tokenization cannot be undermined for increasing accuracy in detecting similarities. Hence, we recognized this phase as an important component of the overall process.

C. Similarity Detection Techniques

We noted that each approach has a unique perspective on similarities when we applied them. For example, while Jaccard similarity is effective for detecting token similarities, Levenshtein distance can detect structural modifications easily. Combining the two improved overall consistency.

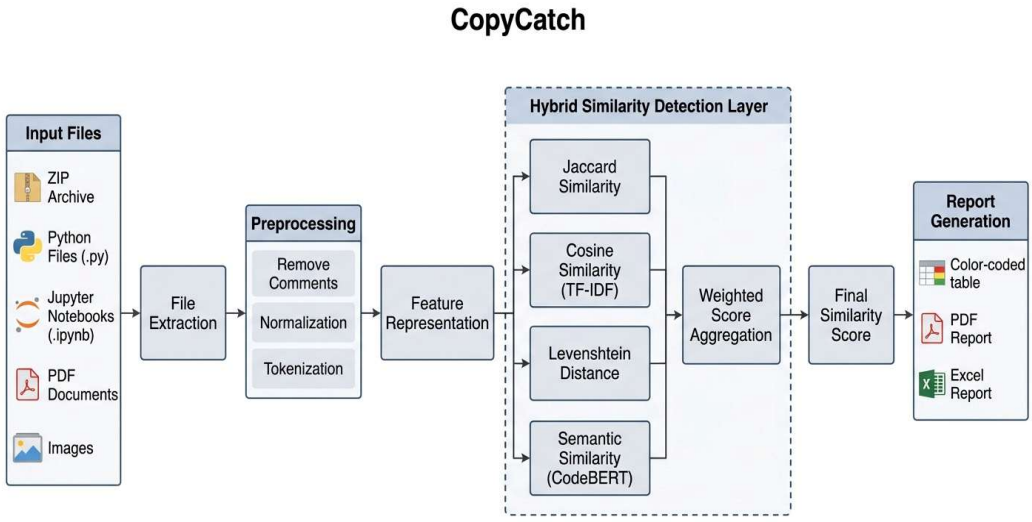

Fig 2

To improve detection accuracy, we used multiple algorithms:

- Jaccard Similarity: Jaccard Similarity is concerned about the degree to which two sets of tokens sampled from the code are similar. As we apply this measure to the process, we observe its effectiveness in detecting similarities due to a high number of tokens. Nevertheless, our experiments revealed that even small changes to the tokens (adding or removing) lead to minor shifts in the result. Despite the fact, it provides an easy-to-understand and implementable approach that may be applied in practice. Therefore, we considered this measure as the basic criterion for similarity identification within the proposed model.
- Cosine Similarity (TF-IDF): Cosine Similarity with TF-IDF considers the occurrence of tokens as a measure of their relevance. According to our results, this metric demonstrates its efficiency in comparing large code files, as token frequencies become more predictable. Moreover, we were able to identify similarities regardless of minor transformations to the tokens' location and content. In general, the experiment proved that the metric is compatible with Jaccard Similarity since it evaluates the same criteria but with different levels of precision. Therefore, it is crucial in developing our model's accuracy further.
- Levenshtein distance: This is based on how many changes have to be performed in order to transform one string to another. Applying this algorithm, we noticed that it is very efficient in identifying minor structural changes such as variable name alterations and other code manipulations. The drawback of this method is increased computational complexity in case of larger file sizes; still, it provides important information concerning the similarity of two files. Therefore, we included it to detect small deviations in code.
- Keyword and Symbol Analysis: This algorithm allows us to identify patterns in usage of programming constructions. We noted that programmers differ in their coding styles, which are revealed by certain sets of symbols and keywords. As a result, using this technique, we managed to recognize certain coding styles, which could not be recognized by other algorithms. In particular, the application of keyword and symbol analysis was highly effective in detecting the cases when changes took place in logical code implementation but were not revealed through code style.

- KL divergence and Jensen–Shannon Divergence measure the comparison of token distribution within two pieces of code. As we applied these measures, it became clear that the statistics of similarity would be provided rather than the similarity itself. Using the above-mentioned techniques, we managed to observe the discrepancies in frequency in spite of the uniformity in structure. Moreover, the Jensen–Shannon Divergence provided better results because of its symmetry. Thus, with the help of the above-mentioned measures, we managed to explore the distribution-level similarity in code comparison. During the use of these techniques, it also became clear that only applying several algorithms simultaneously provides the best results possible.

D. Result Visualization

The results generated by the system include:

Similarity matrices in colors

The system provides the results of the analysis using a colored matrix, and as such, it is very easy to identify similarities. Similarities with red color indicate high scores, while those with green color have low scores.

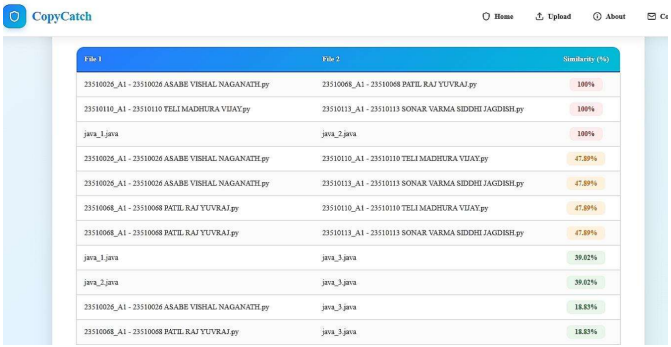

Fig 3

Reports that you can download

You can also easily download reports from the system. You can get them in both PDF and Excel formats.

Fig 4

Excel Report

	A	B	C	D	E
1	file1	file2	similarity		
2	23510026_A1 - 23510026 ASABE VISHAL NAGANATH.py	23510068_A1 - 23510068 PATIL RAJ YUVRAJ.py	100		
3	23510026_A1 - 23510026 ASABE VISHAL NAGANATH.py	23510110_A1 - 23510110 TELI MADHURA VIJAY.py	47.89		
4	23510026_A1 - 23510026 ASABE VISHAL NAGANATH.py	23510113_A1 - 23510113 SONAR VARMA SIDDHI JAGDISH.py	47.89		
5	23510026_A1 - 23510026 ASABE VISHAL NAGANATH.py	java_1.java	6.49		
6	23510026_A1 - 23510026 ASABE VISHAL NAGANATH.py	java_2.java	6.49		
7	23510026_A1 - 23510026 ASABE VISHAL NAGANATH.py	java_3.java	18.83		
8	23510068_A1 - 23510068 PATIL RAJ YUVRAJ.py	23510110_A1 - 23510110 TELI MADHURA VIJAY.py	47.89		
9	23510068_A1 - 23510068 PATIL RAJ YUVRAJ.py	23510113_A1 - 23510113 SONAR VARMA SIDDHI JAGDISH.py	47.89		
10	23510068_A1 - 23510068 PATIL RAJ YUVRAJ.py	java_1.java	6.49		
11	23510068_A1 - 23510068 PATIL RAJ YUVRAJ.py	java_2.java	6.49		
12	23510068_A1 - 23510068 PATIL RAJ YUVRAJ.py	java_3.java	18.83		
13	23510110_A1 - 23510110 TELI MADHURA VIJAY.py	23510113_A1 - 23510113 SONAR VARMA SIDDHI JAGDISH.py	100		
14	23510110_A1 - 23510110 TELI MADHURA VIJAY.py	java_1.java	3.27		
15	23510110_A1 - 23510110 TELI MADHURA VIJAY.py	java_2.java	3.27		
16	23510110_A1 - 23510110 TELI MADHURA VIJAY.py	java_3.java	4.28		
17	23510113_A1 - 23510113 SONAR VARMA SIDDHI JAGDISH.py	java_1.java	3.27		
18	23510113_A1 - 23510113 SONAR VARMA SIDDHI JAGDISH.py	java_2.java	3.27		
19	23510113_A1 - 23510113 SONAR VARMA SIDDHI JAGDISH.py	java_3.java	4.28		
20	java_1.java	java_2.java	100		
21	java_1.java	java_3.java	39.02		
22	java_2.java	java_3.java	39.02		

Fig 5

PDF Report

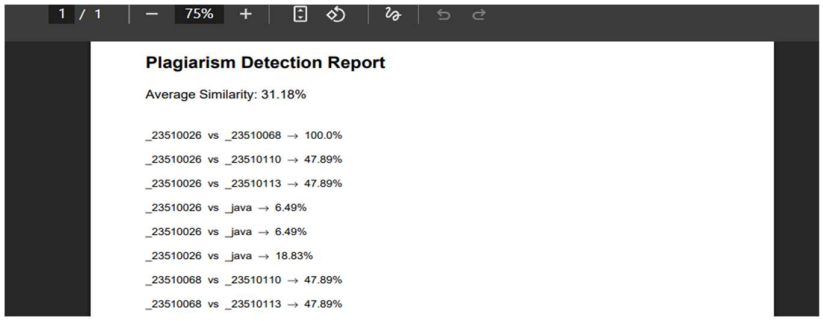

Fig 6

Specifically, the goal was to make the output easy to understand so that teachers can quickly spot possible cases of plagiarism.

V. Results and Discussion

The proposed system assigns scores ranging from 0% to 100% based on the similarity between items. According to the results obtained from this system, the higher the percentage of scores assigned, the more similar the submission is. The use of a threshold value helps identify any suspicious submission. In addition, the matrix representation of colors makes it easy to find a cluster of similar submissions. This system was evaluated using a few numbers of test samples of about 15 to 20 files.
It was observed that combining algorithms reduces the occurrence of false positives compared to using individual approaches.

TABLE II

Sr. No.	Method	File Size	Accuracy (%)
1	Jaccard Similarity	Up to 250 KB	70–75%
2	TF-IDF Cosine Similarity	250 KB – 2 MB	80–85%
3	Levenshtein Distance	Up to 150 KB	65–70%
4	Keyword & Symbol Analysis	Up to 300 KB	68–72%
5	Jensen–Shannon Divergence	300 KB – 2 MB	82–88%
6	Hybrid Multi-Algorithm (CopyCatch)	Up to 25 MB	90–95%

Advantages observed:

- Effective in detecting disguised plagiarism
- Can handle different types of file formats
- Gives results that are easy to understand
- Works efficiently even with the large amounts of data

One limitation we observed is that the system might not fully capture profound semantic similarities when two programs execute identical logic in entirely distinct manners..

VI. Conclusion and Future Work

CopyCatch has been developed as a system to detect plagiarism in programming projects based on multiple algorithms. Multiple algorithms increase the detection capability of the program by detecting similarities on two different levels – surface and deep level. Based on our findings, applying multiple algorithms is more effective than single algorithm because in case of multiple algorithms, any attempt to mask plagiarism would be futile. The accuracy of the system is around 90-95%. So, you can rely on the system to detect similarity/copying of code. In order to further improve performance, we intend to use deep learning models to understand the meaning of code in the future, making the system highly scalable and applicable in multiple languages.

References

[1] Aniceto, Rodrigo C., et al. "Source code plagiarism detection in an educational context: A literature mapping." 2021 IEEE Frontiers in Education Conference (FIE). IEEE, 2021.

[2] Bajaj, Nikesh, et al. "A Tool for Detecting Similarities in Jupyter Notebooks Used as Assessment Reports." 2025 IEEE Global Engineering Education Conference (EDUCON). IEEE, 2025.
[3] Molnár, György, and József Cserkó. "AI Based Plagiarism Checking." IEEE 5th International Conference and Workshop in Obuda on Electrical and Power Engineering (CANDO-EPE 2022). IEEE, 2022.
[4] Novotochinov, Maksim E., Dmitry I. Solomatin, and Andrei V. Chernenkii. "Methodology for Detecting Similarity of Program Code Based on Combining Algorithms." 2024 Conference of Young Researchers in Electrical and Electronic Engineering (ElCon). IEEE, 2024.
[5] Karnalim, Oscar. "Explanation in code similarity investigation." IEEE Access 9 (2021): 59935-59948.
[6] Xu, Xiaojun, et al. "Neural network-based graph embedding for cross-platform binary code similarity detection." Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. 2017.
[7] Liu, Jiahao, et al. "Learning graph-based code representations for source-level functional similarity detection." 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023.
[8] Wang, Huanting, et al. "Combining graph-based learning with automated data collection for code vulnerability detection." IEEE Transactions on Information Forensics and Security 16 (2020): 1943-1958.
[9] Allamanis, Miltiadis, et al. "A survey of machine learning for big code and naturalness." ACM Computing Surveys (CSUR) 51.4 (2018): 1-37.
[10] Veresova, Alina M., and Andrei A. Ovchinnikov. "Burst detection and correction for gilbert codes and its qc-ldpc extensions." 2024 IEEE International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). IEEE, 2024.
[11] Svajlenko, Jeffrey, and Chanchal K. Roy. "Evaluating clone detection tools with bigclonebench." 2015 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 2015.
[12] Zhang, Y., et al."CodeBERT: A Pre-Trained Model for Programming and Natural Languages."IEEE/ACM Transactions, 2023.
[13] Feng, Z., et al."CodeBERT: A Pre-Trained Model for Programming and Natural Language Processing."(Widely used in plagiarism detection research), 2022.
[14] Li, Y., et al."Deep Learning-Based Code Clone Detection: A Survey."IEEE Access, 2022.
[15] Chakraborty, S., et al."A Survey on Code Clone Detection Techniques."IEEE Access, 2022.
[16] Roy, C. K., et al."Recent Advances in Code Clone Detection."Journal of Systems and Software, 2022.
[17] Zhang, H., et al."Graph Neural Networks for Code Similarity Detection."IEEE Transactions on Software Engineering, 2023.
[18] Wang, S., et al."Automated Code Similarity Detection Using AST and Machine Learning."IEEE Access, 2023.
[19] Chen, Q., et al."Hybrid Code Plagiarism Detection Using Token and Semantic Analysis."IEEE Conference Publication, 2024.
[20] Sharma, R., et al."Multi-Feature Based Source Code Plagiarism Detection System."Springer / Lecture Notes in Computer Science, 2023.