Applied Computer Science 2026, vol. 22, no. 2, pp. 208–224
https://doi.org/10.35784/acs_8915

Submitted: 2025-12-27 | Revised: 2026-02-28 | Accepted: 2026-03-07

CC-BY 4.0

Keywords: plagiarism, programming, machine learning, transformers, academic integrity

Oscar KARNALIM [1*], *Yehezkiel David SETIAWAN* [1], *Maresha Caroline WIJANTO* [1], *Rossevine Artha NATHASYA* [1]

[1] Maranatha Christian University, Indonesia, oscar.karnalim@it.maranatha.edu, 2479011@maranatha.ac.id, maresha.cw@it.maranatha.edu, rossevine.an@it.maranatha.edu

[*] Corresponding author: oscar.karnalim@it.maranatha.edu

Machine learning approach to detect GAI-disguised academic programming plagiarism

Abstract

Plagiarism is a common issue in programming education, and it is exacerbated by the emergence of Generative Artificial Intelligence (GAI). Plagiarism acts can be disguised with GAI, resulting in pervasive, consistent changes across the entire program. We present a programming plagiarism detector dedicated to GAI disguises. It not only relies on program similarities but also on GAI characteristics. GAI has its own way of writing programs. Our plagiarism detector employs 23 features. Five of them are related to structure (program similarities) while the rest are associated with GAI characteristics (e.g., the use of list comprehension and recursion). It features seven machine learning models: Logistic Regression, Random Forest, XGBoost, LightGBM, CatBoost, Voting Classifier, and Stacking Classifier. According to our evaluation of 6344 program-pair instances from the machine intelligence course, the Stacking Classifier achieves the highest performance, with 89.17% accuracy, 88.94% precision, 89.17% recall, and 88.77% F-score. It outperforms the baseline (similarity-based plagiarism detectors) by a factor of 2 across most metrics. All structural features (program similarities) are considered important by our machine learning models, accompanied by several GAI-characteristic features. The prominent GAI characteristics are the use of list comprehension, recursion, and branching condition statements without parentheses.

1. INTRODUCTION

In programming education, plagiarism and collusion are quite common (Albluwi, 2019). They are both about reusing someone else's programs and claiming them as one's own (Novak et al., 2019). The only major difference is that, for plagiarism, the owners of the reused programs are not aware of the academic breach (Parthasarathy et al., 2024). A number of mitigation strategies have been developed to address three sides of the fraud triangle (Cendrowski & Martin, 2015). Students can engage in plagiarism and collusion through misrepresentation of concepts, pressure to complete assignments, and opportunities to do so.

Among the three sides of the fraud triangle, lecturers often focus on reducing the opportunities to cheat. Some strategies are straightforward and practical. For instance, lecturers can personalize the assignments by adding minor variations to each student's task (Fowler et al., 2024). Students may not be able to plagiarize their colleague's program since they are addressing different tasks. Lecturers can also continuously update the assignments to prevent cross-semester plagiarism (Surahman & Wang, 2022). Finally, a plagiarism detector can be employed to identify potential cases of plagiarism (Maertens et al., 2024). While high similarity is not always a result of plagiarism or collusion (Bubenkova et al., 2025), it is still substantial evidence.

There are a number of plagiarism detectors (Blanchard et al., 2022). Some of those are publicly available, including MOSS (Sheahen & Joyner, 2016), JPlag (Sağlam et al., 2024), Dolos (Maertens et al., 2024), and SSTRANGE (Karnalim, 2023). Generally, they define similarities by preprocessing the programs and by comparing them via string-matching algorithms, such as Cosine similarity (Foltýnek et al., 2020) and the Karp-Rabin greedy string tiling (RKRGST) (Sağlam et al., 2024). Although these approaches are somewhat effective, they cannot capture the dynamic characteristics of plagiarized programs. Students' concealment of plagiarism can vary across assignments, depending on assignment design, programming proficiency, courses, institutions, or even countries. For example, in simple assignments, attempts at disguising might focus more on code layout or superficial appearance, since the solutions are obvious. In advanced assignments, they might

focus more on code structure and semantics as diverse variations are expected. Plagiarism detectors focusing on semantic similarity might treat all programs in simple assignments as similar. Plagiarism detectors that focus on surface similarity are ineffective for advanced assignments.

Several machine-learning-based plagiarism detectors have been developed to accommodate variations in attempts to disguise plagiarism (Zhang & Saber, 2025). Machine learning algorithms can analyze students' programs and identify plagiarized work. These plagiarism detectors convert programs into features based on the programs' structures (Fokam & Ajoodha, 2021) or characteristics (e.g., data reuse and loop patterns) (Yasaswi et al., 2017). Further, they identify plagiarized programs through machine learning algorithms such as naïve Bayes (Bandara & Wijayarathna, 2011) and neural networks (Yasaswi et al., 2017).

Features extracted from the programs' structures might be helpful in cases of plagiarism where the perpetrators disguise only some parts of the programs. Features extracted from the programs' characteristics might be helpful when some unique characteristics of the original programs are retained, or when perpetrators employ uncommon writing patterns. Since lecturers often lack knowledge of what is retained in their students' plagiarism cases, there is a need for a machine-learning-based plagiarism detector that extracts features from both programs' structures and characteristics. They can dynamically handle disguising attempts.

Generative Artificial Intelligence (GAI) is a form of AI that focuses on generating structure (Jovanovic & Campbell, 2022), and its misuse substantially complicates academic integrity (Xie et al., 2023). With GAI, students can complete assignments without achieving the expected competencies (Kosmyna et al., 2025). Furthermore, they can disguise their plagiarism using GAI (Pudasaini et al., 2024). Several GAI assistance detectors have been developed. They rely on either GAI writing patterns (Hoq et al., 2024), structural differences (Karnalim, 2025), and anomaly characteristics (Karnalim et al., 2024). While these can detect unauthorized use of GAI, they cannot identify the sources of GAI-disguised programs. Their output is a list of programs that are likely to be GAI-assisted. Another issue is that GAI-disguised plagiarism is more complex to deal with than the human-disguised counterpart. The disguises tend to be more pervasive and consistent. The disguised programs might not be considered similar to their original counterparts by similarity-based plagiarism detectors.

In response to the aforementioned gap, this study presents a detector for identifying GAI-disguised plagiarism that uses 23 features extracted from both programs' structures and characteristics. Programs' structures are extracted into five features: RKRGST, Term-Frequency-Inverse-Document-Frequency (TF-IDF) Cosine, CodeBERT Cosine, GraphCodeBERT Cosine, and CodeT5 Cosine similarity algorithms. Programs' characteristics are extracted as the sum and the difference of eight code-anomaly aspects (e.g., use of a dictionary and an empty return). The overall sum and difference of the programs' characteristic anomalies are also included. Our detector offers a range of machine learning algorithms to detect GAI-disguised programs. The algorithms include Logistic Regression, Random Forest, XGBoost, LightGBM, CatBoost, voting classifier, and stacking classifier. To the best of our knowledge, ours is the first automated detector that: 1) is intended for GAI-disguised plagiarism in programming; 2) combines features from both programs' structures and characteristics; 3) employs machine learning algorithms to learn from students' programs dynamically.

Our study has four research questions:

- RQ1: What disguises do GAI often employ to cover programming plagiarism?
- RQ2: Which machine learning model is the most effective for identifying GAI-disguised plagiarism?
- RQ3: Which features are the most important for identifying GAI-disguised plagiarism?
- RQ4: Is our plagiarism detector more effective than baseline plagiarism detectors in identifying GAI-disguised plagiarism?

2. LITERTURE REVIEW

2.1. Strategies to deal with programming plagiarism and collusion

Several strategies have been developed to deal with programming plagiarism and collusion. They can be grouped by the side of the fraud triangle they primarily address (Albluwi, 2019). Students can engage in plagiarism and collusion due to misinterpretation of the concepts, external pressure, and availability of opportunities.

To address misinterpretations of the concepts of plagiarism and collusion, lecturers can set out their expectations at the beginning of the course or right before issuing an assignment (Simon et al., 2018). Students can get reminded of such expectations through a standalone educational application (Tsang et al., 2018) or formative feedback upon completing an assignment (Karnalim et al., 2023). A clear explanation of lecturers' expectations on plagiarism and collusion can reduce the number of plagiarism and collusion cases (Mason et al., 2019).

External pressures to commit plagiarism or collusion typically stem from poor time management, task difficulty, and boring assignments. To deal with poor time management, lecturers can encourage students to submit their programs early (Spacco et al., 2013). They are less likely to have enough time to complete the assignments. To address task difficulty, lecturers can break down assignments (Allen et al., 2018). Smaller assignments tend to be easier to complete. Lecturers can also allow students to work in groups (Aivaloglou & Meulen, 2021) or pairs (Hawlitschek et al., 2023). Students can help one another. Although it remains uncommon, students may also be allowed to use GAI to assist with completing assignments (Li et al., 2025). If used correctly, GAI can help students complete assignments faster (Toba et al., 2023).

Opportunities for engaging in plagiarism or collusion can arise in many ways. For instance, assignments reused multiple times might allow students to copy their seniors' programs and engage in collusion (Simon, 2017). Lecturers are recommended to update their assignments periodically. To address students colluding with one another, lecturers can issue several variations of the assignments at once (Fowler & Zilles, 2021). Students cannot blindly copy from their colleagues' programs if the versions of the assignment differ. It is also possible to allow students to have their own case studies (Bradley, 2020), which are expected to differ from one student to another.

Another way to reduce opportunities for plagiarism and collusion is to monitor their program creation as they complete assignments. Although this might raise privacy concerns, the approach can be quite effective. There are several integrated development environments (IDEs) that can add a watermark to each program (Ryman et al., 2022), monitor change logs (Schneider et al., 2018), or capture keystroke information (Ljubovic & Pajic, 2020).

Many plagiarism detectors have been developed to detect plagiarism and collusion (Blanchard et al., 2022). Given a set of students' submitted programs for a particular assignment, the detectors will identify similar programs by performing pairwise comparisons (i.e., comparing each program to every other). Plagiarism detectors commonly rely on programs' structures (structures) and/or characteristics. Program structures are typically represented as token strings that have been preprocessed in various ways to address attempts at disguising (Novak et al., 2019). Comments and white space are usually removed. Further, identifiers are generalized to their own type. Other representations for the plagiarism detectors include syntax trees (Sharma et al., 2024) and program dependency graphs (Cheers et al., 2021). The comparison employs standard matching algorithms for strings, trees, and graphs.

Program characteristics are derived from token occurrences (Yasaswi et al., 2017), syntactic constructs (Viuginov et al., 2020), and programming styles (Karnalim & Kurniawati, 2020). Plagiarism detectors often treat these characteristics as features and measure their similarity using information retrieval (Duracik et al., 2020), machine learning (Viuginov et al., 2020), or clustering methods (Yasaswi et al., 2017).

Some plagiarism detectors employ machine learning algorithms so that their identification aligns well with students' own programs (Zhang & Saber, 2025). Bandara & Wijayarathna (2011) implemented Naïve Bayes, K-Nearest Neighbor, and AdaBoost to identify similar programs based on nine program characteristics, including the number of characters and words per line. Yasaswi et al. (2017) developed a clustering algorithm (t-Distributed Stochastic Neighbor Embedding) to identify similar programs based on 55 program characteristics extracted using Milepost GCC. Most of these features count the occurrences of syntax blocks or statements. Yasaswi et al. (2017) used a support vector machine to identify similarities from students' programs by treating their characters as features.

Ullah et al. (2018) filtered tokens from students' programs using principal component analysis and identified similarities using multinomial logistic regression. Karnalim & Kurniawati (2020) used stylistic characteristics from 83 programs to identify similarities, including comments, identifier names, and program layout. The identification itself was aided by a naïve Bayes algorithm. Fokam & Ajoodha (2021) decomposed abstract syntax trees into features and employed Siamese neural networks for identification.

Ullah et al. (2021) used a random forest classifier to classify similar programs as part of their syntax-tree-based clustering algorithm. Eppa & Murali (2021) implemented the DBSCAN algorithm to group similar

programs by function. They also applied multiple machine learning algorithms to define program similarities based on the tokens (Eppa & Murali, 2022).

2.2. Strategies to deal with GAI misuse in programming

Generative Artificial Intelligence (GAI) can be misused to complete programming assignments without expected competencies (Kosmyna et al., 2025). GAI can generate solutions to the given assignments based on natural-language instructions. Several GAI assistance detectors have been developed to address this. Hoq et al. (2024) employed SVM, XGBoost, ASTNN, and SANN to identify GAI-generated programs based on their code2vec representations. Karnalim et al. (2024) identified 34 anomaly metrics to identify GAI assistance in Python programs. GAI misuse was determined using simple statistics: more anomaly features indicate a stronger tendency toward GAI misuse. The metrics mainly covered syntax constructs. They were then employed as features for machine learning models such as random forests, logistic regression, and multilayer perceptrons (Toba & Karnalim, 2025).

Xu and Sheng (2024) identified GAI-generated programs using a targeted masking perturbation and a unified scoring system that combines perplexity, variation, and burstiness. Nguyen et al. (2024) introduced GPTSniffer, a CodeBERT-based method, to identify GAI-generated programs by their structure. CodeBERT was also employed by EX-CODE (Bulla et al., 2024), but in combination with logistic regression for identification. Pham et al. (2024) employed the CodeT5+ pretrained model to identify GAI-generated programs based on the likelihood of the structures. Xu et al. (2025) encoded students' programs with UniXcoder and identified GAI-generated programs via contrastive learning. Karnalim (2025) used parallel KNN-based weight outlier detection to identify GAI assistance in students' programs using token strings.

Another form of GAI misuse is using it to disguise plagiarism (Pudasaini et al., 2024). Students can easily ask GAI to disguise their plagiarized programs. To the best of our knowledge, there are no automated detectors specifically designed to detect this kind of disguise. Existing GAI detectors cannot identify the original programs behind GAI-disguised plagiarism, making them less helpful for determining such plagiarism.

3. METHODS

Existing plagiarism detectors might be ineffective at detecting GAI disguises, as these disguises reduce program similarity due to their pervasiveness and consistency. Many of them are not adaptive as they do not learn the plagiarism patterns from students' programs. Even though some of them are adaptive, their effectiveness is limited because they rely solely on the programs' structures or characteristics.

Our detector is designed explicitly to detect GAI-disguised plagiarism, with adjustable patterns based on comprehensive feature sets. It employs machine learning algorithms that combine features from both programs' structures and characteristics. For our case, we focus on Python programs.

Our automated detector has three components: a feature extractor, a model builder, and a plagiarism classifier. The feature extractor accepts students' programs and extracts features from them. It is used in both the model builder and the plagiarism classifier. The model builder accepts students' programs and develops a machine learning classification model from them. The plagiarism classifier accepts students' programs and determines their likelihood of GAI-disguised plagiarism.

3.1. Feature extractor

Our feature extractor accepts student program pairs and converts each pair into features. The pairs are obtained by comparing students' programs to one another. For instance, if there are four student programs (P01, P02, P03, and P04), then there will be six program pairs (P01-P02, P01-P03, P01-P04, P02-P03, P02-P04, and P03-P04).

For each pair, twenty-three features are extracted. Five of them (S01-S05) are based on the programs' structures, and they are derived from similarity measurements with resulting values between 0 and 1 inclusive (0 indicates completely different programs, and 1 indicates identical programs):

1. S01: Running Karp-Rabin Greedy String Tiling (RKRGST) similarity, which is a common algorithm for plagiarism detectors to detect programming plagiarism (Novak et al., 2019). The feature is generated by tokenizing the programs with ANTLR (Parr, 2013), comparing them with RKRGST, and measuring the similarity as defined in Equation (1). Pa and Pb are students' programs part of the pair. match(Pa,

Pb) denotes the number of matched tokens between the two programs. len(Pa) and len(Pb) denote the sizes of the programs, respectively. RKRGST's minimum matching length is set to 40, assuming 8 program statements in Python (the programming language used in our datasets). It refers to eight program statements, with one statement being about five tokens (Karnalim et al., 2023).

$$sim(Pa, Pb) = \frac{match(Pa,Pb)}{len(Pa)+len(Pb)} \quad (1)$$

2. S02: Term-Frequency-Inverse-Document-Frequency (TF-IDF) Cosine similarity. Cosine is widely used to compare vector representations of the programs (Ebrahim & Joy, 2024). It measures how two similar vectors are based on the angles between them. TF-IDF itself prioritizes tokens that occur frequently in specific programs but are rare across students' programs. Vectorization is aided by TfidfVectorizer from Python's sklearn library.
3. S03: CodeBERT Cosine similarity. CodeBERT is a transformer encoder derived from the RoBERTa style that is trained on both human and programming languages (Feng et al., 2020). It can convert students' programs into semantic embeddings, which can be useful to determine program similarities. Cosine is then employed to measure similarities between the resulting embeddings. The implementation is based on Hugging Face's transformer library.
4. S04: GraphCodeBERT Cosine similarity. GraphCodeBERT is an extension of CodeBERT that incorporates programming graph structure (Guo et al., 2020). Its semantic embedding tends to be more comprehensive. The feature is measured in a manner similar to CodeBERT's Cosine similarity.
5. S05: CodeT5 Cosine similarity. CodeT5 is a text-to-text transformer that can be useful for program understanding and generation (Wang et al., 2021). Its semantic embeddings can serve as an alternative for measuring program similarity. Again, the feature is calculated in a manner similar to CodeBERT's Cosine similarity.

Eighteen features (C01-C18) are based on counting the occurrences of programs' characteristics that are identified via abstract syntax trees with ANTLR. They are then normalized to the maximum number of occurrences for each feature, where 1 refers to the highest possible value, and 0 refers to the lowest one. These features are often employed by GAI but not by students. Students prefer simple and readable syntax constructs. Our eighteen features are:

1. C01: The sum of occurrences of shorthand branching conditions before the statements (e.g., "if(c < 0): c = c + 2") from both programs. It is determined by counting the branching statements where the first and last child nodes in the abstract syntax tree appear on the same line.
2. C02: Similar to C01, except the sum is replaced with the differences in occurrences. The shorthand branching conditions before the statements might only occur on the GAI-disguised program, not the original program.
3. C03: The sum of occurrences of shorthand branching conditions after the statements (e.g., "c = c + 2 if(c < 0) else c = c - 2") from both programs. It is determined by counting simple or expression statements containing at least one "if" keyword.
4. C04: Similar to C03, except the sum is replaced with the differences in occurrences.
5. C05: The sum of occurrences of branching condition statements without parentheses (e.g., "if a == 10:") from both programs. It is determined by counting these statements from the abstract syntax trees.
6. C06: Similar to C05, except the sum is replaced with the differences in occurrences.
7. C07: The sum of occurrences of while looping condition statements without parentheses (e.g., "while a < 10:") from both programs. It is determined by counting these statements from the abstract syntax trees.
8. C08: Similar to C07, except the sum is replaced with the differences in occurrences.
9. C09: The sum of occurrences of the "return" keyword without returned values from both programs. It is determined by counting keywords without accompanying values in the abstract syntax trees.
10. C10: Similar to C09, except the sum is replaced with the differences in occurrences.
11. C11: The sum of recursion occurrences from both programs. It is determined by counting functions that call themselves in their body.
12. C12: Similar to C11, except the sum is replaced with the differences in occurrences.
13. C13: The sum of occurrences of list comprehension statements from both programs. It is determined by counting assignment statements whose right side starts with '[' and then followed by 'for', 'in', and ']' sequentially.

14. C14: Similar to C13, except the sum is replaced with the differences in occurrences.
15. C15: The sum of occurrences of dictionaries from both programs. It is determined by counting assignment statements of dictionaries.
16. C16: Similar to C15, except the sum is replaced with the differences in occurrences.
17. C17: The sum of all summative characteristic features: C01, C03, C05, C07, C09, C11, C13, and C15. Combining these features might amplify the characteristics of GAI programs.
18. C18: The differences in occurrences of all summative characteristic features, assuming these features can exclusively occur on GAI disguised programs, not the original ones.

3.2. Model builder

Our model builder accepts students' program pairs, obtained by combinatorially pairing the programs as described in the previous section. Each program pair is treated as a single instance, preprocessed into feature vectors with the feature extractor, and its target class is labeled with either GAI-disguised plagiarism or not.

The automated detector has several machine learning models (Zhou, 2021) to choose from:

1. Logistic regression: a linear machine learning model that determines the probability of a target class by applying the logistic function to a weighted sum of input features.
2. Random forest: An ensemble machine learning model that consists of many weak decision trees on random subsets of data and features. The prediction is determined by aggregating their outputs.
3. XGBoost: A gradient boosting machine learning model that builds trees sequentially to correct previous errors.
4. LightGBM: A fast gradient boosting machine learning model that grows trees leaf-wise and uses histogram-based splits for efficiency.
5. CatBoost: A gradient boosting machine learning model designed to handle categorical features using ordered boosting and target statistics.
6. Voting classifier: An ensemble machine learning model that combines predictions from multiple models by averaging probabilities. For our context, it employs the first five machine learning models, from logistic regression to CatBoost.
7. Stacking Classifier: An ensemble machine learning model that incorporates a meta-model to combine the predictions of several base models for improved performance. The base models are the first five machine learning models described here.

3.3. Plagiarism classifier

The plagiarism classifier accepts program pairs, preprocesses them to feature vectors via the feature extractor, and for each program pair, classifies whether the pair is likely to have GAI-disguised plagiarism. The similarity classifier requires a model created by the model builder based on students' programs.

4. EVALUATION

The four research questions described at the end of the introduction were addressed here using four effectiveness metrics. Accuracy referred to the proportion of correctly predicted instances, both (GAI-disguised) plagiarized and non-plagiarized instances (true positive and true negative) among all predicted instances. Precision was defined as the proportion of plagiarized instances (true positives) among suspected instances (true and false positives). Recall is the proportion of suspected instances (true positives) among all plagiarized instances (true positives and false negatives). F-score was the harmonic mean between precision and recall.

Our dataset consisted of one semester's worth of weekly individual assignments from a machine intelligence course. The course covered basic machine learning concepts in Python for third-year Informatics Engineering undergraduates with 13 weekly assignments. The course was offered to 53 students in two classes. A total of 530 student programs were collected for the whole semester. For each assignment, students were expected to complete template programs in Python to implement and evaluate a machine learning model.

GAI-disguised plagiarized programs were created in three steps by the second author and one graduate student. There would be three plagiarized programs for each assignment per class. First, three student programs were selected randomly. Second, each program was then uploaded to ChatGPT with all identifiable

information removed. The prompt was “You are an informatics engineering student who wants to plagiarize the program below. Disguise the program so that it can pass the plagiarism detector”. Third, for each class assignment, all students' programs, including the GAI-disguised one, were paired with one another. They were then considered instances. For any programs paired with the GAI-disguised one, if deemed similar, they would be regarded as plagiarized program pairs as well. This was done to prevent class imbalance. By default, the number of plagiarized programs was far smaller than that of non-plagiarized programs. Our dataset contained 6344 instances, of which 1583 were identified to have GAI-disguised plagiarism.

This study and all its experimental protocols have received the appropriate ethics approval from the institution.

4.1. Addressing RQ1: GAI disguises for covering programming plagiarism

RQ1 asked what disguises GAI often employs to cover programming plagiarism. This was addressed during the dataset creation. For each GAI-disguised plagiarized program, GAI would be asked to list the changes that it made. All changes were then summarized, with their occurrence frequencies counted.

Tab. 1 shows the reported GAI disguises, including their occurrence frequencies, sorted in descending order. The most common GAI disguise was changing variable names (76). Superficial modifications were the preferred and easiest strategy to make code look different without altering its core logic. Renaming function names (48) and restructuring code flow or refactoring (44) were also frequently used, suggesting that many attempts went beyond simple renaming and involved moderate structural changes to reduce direct similarity.

Tab. 1. GAI disguises and their occurrence frequencies

Disguises	Frequency
Change variable names	76
Change function names	48
Change code structure (flow, order, and refactoring)	44
Change comments	40
Change values of variables for model parameters and experiments (e.g., random_state and test_size)	36
Change loop mechanism (e.g., introducing list comprehension or enumerate)	25
Change the output formatting	24
Change ways of calling library functions	22
Change used data structure (e.g., list to dict)	21
Change visualisation style	20
Change validation and logging	16
Change preprocessing style	13
Change evaluation metrics	11
Change programming paradigm (OOP to procedural or vice-versa)	5

A second group of GAI disguises focused on altering how the programs behaved or appeared during execution. Changes to comments (40) and adjustments to experimental or model parameters such as random_state and test_size (36) were common, as they helped produce different outputs while preserving the same underlying methodology. Modifying loop mechanisms (25), output formatting (24), library function calls (22), and data structures (21) further helped disguise similarities by leveraging alternative yet functionally equivalent implementations.

Less frequently used disguises tend to involve deeper design and analytical decisions. Changes in visualization style (20), validation and logging (16), preprocessing methods (13), and evaluation metrics (11) required more domain understanding and effort, which GAI might not consistently achieve. The least common disguise was changing the overall programming paradigm, such as switching between object-oriented and procedural styles (5), because this required substantial rewriting and could risk producing incorrect programs.

4.2. Addressing RQ2: Most effective machine learning model

RQ2 asked which machine learning model was the most effective for identifying GAI-disguised plagiarism. It was addressed by comparing the effectiveness metrics across all machine learning models covered by our plagiarism detector: Logistic Regression, Random Forest, XGBoost, LightGBM, CatBoost, Voting Classifier,

and Stacking Classifier. For non-ensemble models, feature selection was performed before hyperparameter tuning. Only features with positive importance would be considered. The employed features for each machine learning model are shown in Tab. 2, sorted by importance. XGBoost used the most features (19), while Logistic Regression used the fewest (8).

Tab. 2. Important features per machine learning model

Model	Features	Total Features
Logistic Regression	C14, C17, S04, C11, C18, C12, C01, C02	8
Random Forest	S01, S02, S03, S05, S04, C17, C15, C18, C13, C16, C14, C03, C05, C04, C06	15
XGBoost	S01, C13, C17, S03, C05, C15, C09, C05, C06, S03, C14, S04, C10, C18, C16, C12, C04, C03, C08	19
LightGBM	S04, S03, S05, S02, S01, C17, C15, C18, C13, C16, C05, C14, C03, C09, C10, C04, C06	17
CatBoost	S01, S02, S04, S03, S05, C17, C15, C13, C18, C16, C14, C05, C03, C10, C06, C09, C04, C11	18

For each model except the ensemble models (the last two), hyperparameter tuning was conducted using the search space shown in Tab. 3, with a stratified 5-fold cross-validation. The best parameters for each model were shown in Tab. 4.

Tab. 3. Hyperparameter search space

Model	Parameter	Values
Logistic Regression	penalty	11, 12
	C	1, 10, 100, 1000
Random Forest	n_estimators	50, 100, 200
	max_depth	10, 20, 50, None
	min_samples_split	1, 2, 5, 10
	min_samples_leaf	1, 2, 5, 10
	max_features	sqrt, log2
	bootstrap	True, False
XGBoost	n_estimators	50, 100, 200
	max_depth	3, 5, 7
	learning_rate	0.01, 0.1, 0.5, 1.0
	subsample	0.5, 0.75, 1.0
LightGBM	n_estimators	50, 100, 200
	max_depth	5, 10, 15
	learning_rate	0.01, 0.1, 0.5, 1.0
	num_leaves	10, 25, 50, 100
CatBoost	iterations	100, 200, 500, 1000
	depth	4, 6, 8, 10
	learning_rate	0.01, 0.1, 0.5, 1.0
	l2_leaf_reg	1, 3, 5, 7
	border_count	16, 32, 64

Each model was evaluated on the dataset using a stratified 5-fold cross-validation. The resulting accuracies for all models are shown in Fig. 1. The results indicate clear differences between linear and non-linear machine learning models. Logistic Regression achieved the lowest accuracy (72.18%), as expected, because it is a linear model that assumes a simple relationship between features and the target variable. The underlying dataset appeared to exhibit nonlinear patterns and complex feature interactions that are difficult to capture with Logistic Regression. This was supported by the fact that Linear Regression used only eight features. The patterns and feature interactions, however, could be captured by tree-based models such as Random Forest, XGBoost, LightGBM, and CatBoost. They were designed to handle nonlinearity and interactions, which explains their substantially higher accuracy (more than 85%).

Tab. 4. Best parameters per model

Model	Parameter	Values
Logistic Regression	penalty	12
	C	10
Random Forest	n_estimators	200
	max_depth	50
	min_samples_split	2
	min_samples_leaf	1
	max_features	sqrt
	bootstrap	True
XGBoost	n_estimators	200
	max_depth	7
	learning_rate	0.1
	subsample	0.75
LightGBM	n_estimators	200
	max_depth	10
	learning_rate	0.5
	num_leaves	50
CatBoost	iterations	200
	depth	4
	learning_rate	0.5
	l2_leaf_reg	7
	border_count	32

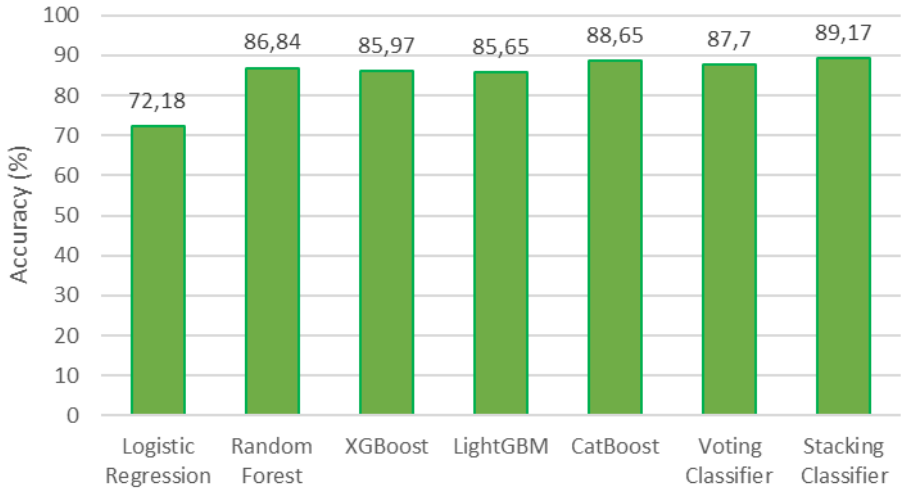

Fig. 1. Accuracy of the machine learning models

Among the tree-based models, CatBoost achieved the highest accuracy (88.65%) because it was designed to effectively handle categorical features, reduce overfitting through ordered boosting, and manage complex feature interactions without extensive preprocessing. Random Forest also performed well (86.84%) by averaging multiple decision trees, thereby reducing variance and improving generalization. Gradient boosting methods (XGBoost and LightGBM) delivered competitive results by sequentially correcting previous errors. Slight differences in accuracy might stem from how each model handles tree growth, regularization, and data sampling.

The ensemble models (voting and stacking classifiers) achieved the best overall accuracy, with stacking achieving the highest (89.17%). The stacking classifier combined predictions from five models (Logistic Regression, Random Forest, XGBoost, LightGBM, and CatBoost) and learned to optimally weight them using a meta-learner. This promoted the strengths of each base model while mitigating its weaknesses. The voting classifier also benefited from model diversity but used a more straightforward aggregation strategy, resulting in slightly lower accuracy than stacking.

In terms of precision, Fig. 2 depicts that the results followed a similar pattern to the accuracy metrics, with Logistic Regression producing the lowest precision (75.59%). The linear decision boundary of Logistic

Regression is less effective at minimizing the number of non-plagiarized instances misclassified as plagiarized (false positives) in a dataset with nonlinear features. Tree-based models again demonstrated strong performance, with Random Forest and XGBoost both achieving 86.49% precision, and LightGBM slightly lower at 85.54%. These models improved precision by learning complex decision rules that reduce the false-positive rate. CatBoost achieved the highest precision (88.97%), due to its robust handling of categorical variables and its ordered boosting mechanism. Both yielded more true positives (suspected and plagiarized instances) with fewer false positives.

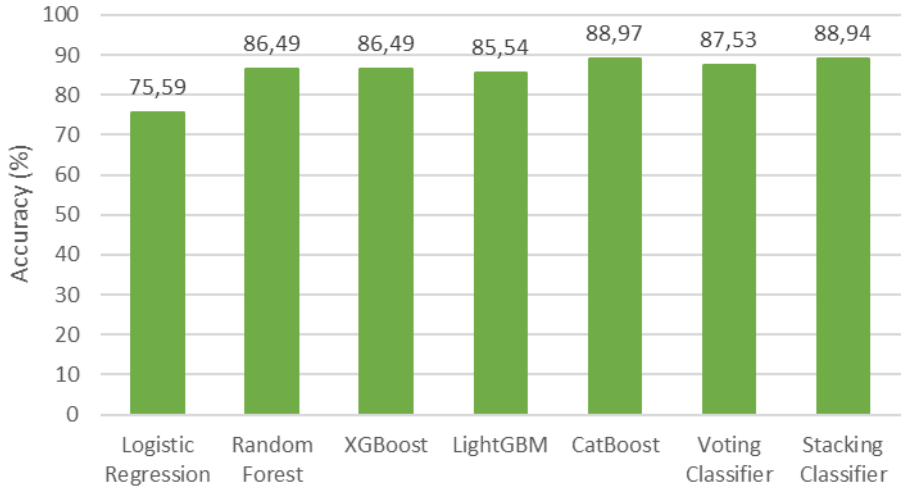

Fig. 2. Precision of the machine learning models

The ensemble models (voting and stacking classifiers) maintained high precision values (87.53% and 88.94%, respectively), closely matching CatBoost's performance. As with accuracy, stacking performed slightly better due to its meta-learning approach. The approach optimally combined base model predictions to reduce false positives (plagiarized but not suspected).

In terms of recall and F-score, the results were somewhat comparable. The patterns applied not only promoted more plagiarized instances among the suspected instances (precision) but also promoted more suspected plagiarized instances (recall). Linear Regression yielded the lowest performance (72.18% recall and 73.37% F-score). Voting and stacking classifiers maintained the highest performance. The voting classifier achieved 87.7% recall and 87.6% F-score. The stacked classifier achieved 89.17% recall and 88.77% F-score. The stacking classifier was the most effective, achieving high scores across all effectiveness metrics.

4.3. Addressing RQ3: Most important machine learning features

RQ3 asked which machine learning model was the most important for identifying GAI-disguised plagiarism. It was addressed based on the feature importance analysis in Tab. 2. Each feature was assigned an importance score as defined in Equation (2), considering only the top five most important features for each machine learning model. While other features might be necessary, their impact was less substantial. importance_score (f) refers to the overall score of feature f. It averaged the result of subtracting six from the feature's rank across all machine learning models. Six was subtracted from the rank so that a higher rank corresponded to a higher importance score, with scores between 1 and 5 inclusive. Six was chosen because we considered only the top five features for each model. total_models refers to the total number of models used to measure feature importance. In our case, it was assigned five as we employed five machine learning models: Logistic Regression, Random Forest, XGBoost, LightGBM, and CatBoost.

$$importance\_score(f) = \frac{\sum_{i=1}^{total\_models} 6 - rank(f,i)}{total\_models} \quad (2)$$

Tab. 5 showed 11 important features, sorted in descending order by importance score. S01 (RKRGST similarity) was the most important one with a 3.2 score. This confirmed the usability of RKRGST in detecting program similarity and justified its popularity for identifying plagiarism. S04 (GraphCodeBERT Cosine

similarity) was the second-most important feature, with a score of 2.4. GraphCodeBERT was another similarity-based feature, but it focused more on program structure, leading many students' programs to be considered similar. The assignments were given weekly, and the expected variations were quite limited. S03 (CodeBERT Cosine similarity) was also considered important since it was the base model of GraphCodeBERT. S02 (TF-IDF Cosine similarity) ranked as the fourth most important feature. This justified the popularity of TF-IDF Cosine similarity for identifying identical structures.

Tab. 5. Important features sorted based on their importance score

Feature	Importance score
S01	3.2
S04	2.4
S03	2.2
S02	2
C17	1.4
S05	1.2
C14	1
C13	0.8
C11	0.4
C05	0.2
C18	0.2

C17 had an importance score of 1.4, the highest among the characteristic features. Since it was the sum of all summative characteristic features, it confirmed our hypothesis stating that both structural and characteristic features were important in identifying GAI-disguised plagiarism. GAI often employed uncommon syntactic constructs that were easily identifiable by characteristic features. However, students sometimes used these constructs as well. C18, which was the difference in occurrences of all summative characteristic features, resulted in a low importance score (0.2), and it was only considered important by Logistic Regression.

S05 (CodeT5 Cosine similarity) resulted in 1.2 importance score. This confirmed that all structural features were important in identifying GAI-disguised plagiarism. However, they should be accompanied by characteristic features. Even S01, the most important structural feature, was not considered important in Logistic Regression. Other structural features were considered important only in certain machine learning models.

C14 (the differences in occurrences of list comprehension statements from both programs) was important, with a score of 1, since GAI often employed such syntax constructs. However, students seldom used such constructs because they had limited programming experience, and the syntax was quite advanced. The importance of C14 supported the importance of its counterpart, C13 (the sum of occurrences of list comprehension statements from both programs).

Other weakly important features were C11 (the sum of occurrences of recursion from both programs), C05 (the sum of occurrences of branching condition statements without parentheses), and C18 (the difference in occurrences of all summative characteristic features). The importance of C11 and C05 showed that GAI could employ recursion and branching condition statements without parentheses to disguise similarities. Students could also use the syntax constructs, though the use was less frequent. This was supported by the contrasting importance of C17 and C18. The former was substantially more important than the latter.

To sum up, structural features were all-important and could be used to identify GAI-disguised plagiarism. However, it should be supported by characteristic features. GAI might employ uncommon syntax constructs such as list comprehension, recursion, and branching condition statements without parentheses. They were seldom used by students, though some still might. Their occurrence can compensate for the reduced similarity introduced by GAI when detecting GAI-disguised plagiarism.

4.4. Addressing RQ4: Effectiveness compared to baseline detectors

RQ4 asked whether our plagiarism detector is more effective than the baseline detectors. In our case, we employed all structural features as baseline detectors: RKRGST, TF-IDF Cosine, CodeBERT Cosine, GraphCodeBERT Cosine, and CodeT5 Cosine. For each feature, two similarity thresholds were considered: 50% and 75%. Per threshold, program pairs with a similarity degree higher than the threshold were suspected of GAI-disguised plagiarism. The baseline detectors would be compared against two variants of our plagiarism

detector: one using the worst-performing machine learning model (Linear Regression) and another using the best-performing model (Stacking Classifier).

Fig. 3 compares the performance of the baseline detectors with a 50% similarity threshold against our plagiarism detector, which uses Logistic Regression and the Stacking Classifier. In terms of accuracy, RKRGST (40.91%) and TF-IDF (35.03%) performed modestly, while CodeBERT, GraphCodeBERT, and CodeT5 showed low accuracy (around 25%). RKRGST and TF-IDF were more favored in terms of accuracy because they focus on surface similarities, which can be substantially affected by GAI disguises. Both would consider many instances non-plagiarized, and there were more non-plagiarized instances than plagiarized ones. However, even RKRGST and TF-IDF still could not outperform the two variants of our plagiarism detector: one with Logistic Regression (72.18%) and another with Stacking Classifier (89.17%). This showed that program similarity alone was insufficient to identify GAI-disguised plagiarism. GAI disguises might negatively affect the resulting similarities.

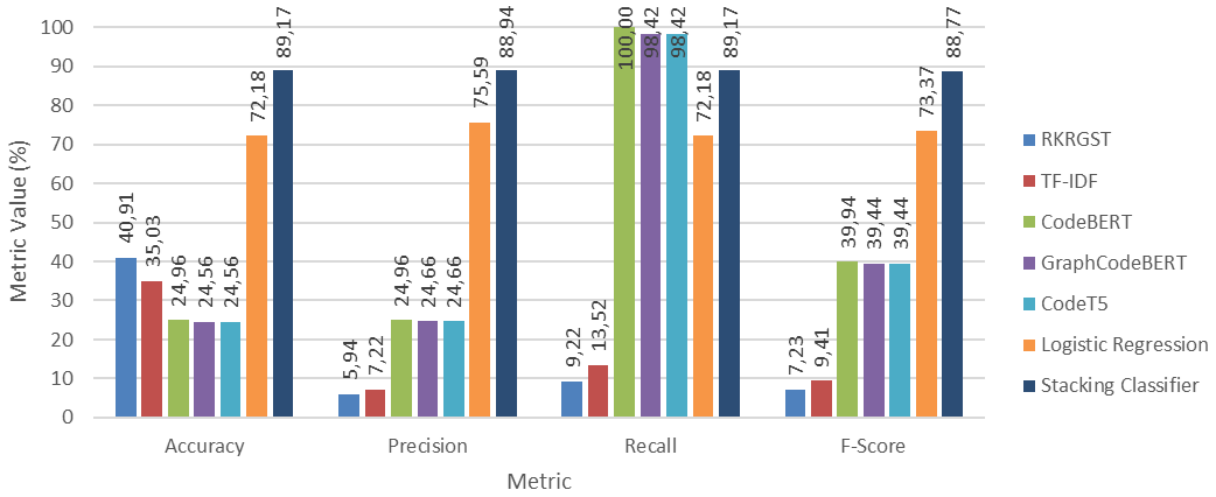

Fig. 3. Performance of baseline detectors with 50% similarity threshold compared to logistic regression and a stacking classifier

In terms of precision, RKRGST (5.94%) and TF-IDF (7.22%) performed worse than transformer-based similarities (CodeBERT with a 24.96% precision, GraphCodeBERT with 24.66% precision, and CodeT5 with a 24.66% precision). This was expected, since transformer-based similarities focused on program semantics, making them resistant to certain changes, including those arising from GAI disguises. Nevertheless, their precision was unmatched by the variants of our plagiarism detector. Ours with Logistic Regression entailed a 75.59% precision, while ours with Stacking Classifier entailed an 88.94% precision.

Regarding recall, RKRGST and TF-IDF performed poorly, with 9.22% and 13.52%, respectively. They tended to classify instances as non-plagiarized because the resulting similarity scores were low. They could be easily affected by any changes and disguises. Transformer-based similarities outperformed our plagiarism detector, achieving nearly 100% recall. They considered many program pairs to be plagiarized, since students' programs submitted for the same assignment tended to have similar program semantics. This was supported by their low precision and F-score.

All baseline detectors entailed lower F-Scores than our plagiarism detector variants. This was expected based on our findings on precision and recall. The F-score was the harmonic mean between those two.

When the similarity threshold was increased from 50% to 75% (see Fig. 4), the baseline detectors exhibited similar patterns, except that, for both RKRGST and TF-IDF, accuracy and precision improved at the expense of recall and F-score. Changes in accuracy were substantial, whereas changes in other metrics were minor. Increasing the similarity threshold prioritized more non-plagiarized instances as not suspected (true negatives). Nevertheless, both variants of our plagiarism detector still outperformed the baseline models. Transformer-based similarities did not change much, since on the dataset, their resulting similarity degrees were already above 75%. Programs submitted to an assignment were expected to share some program semantics.

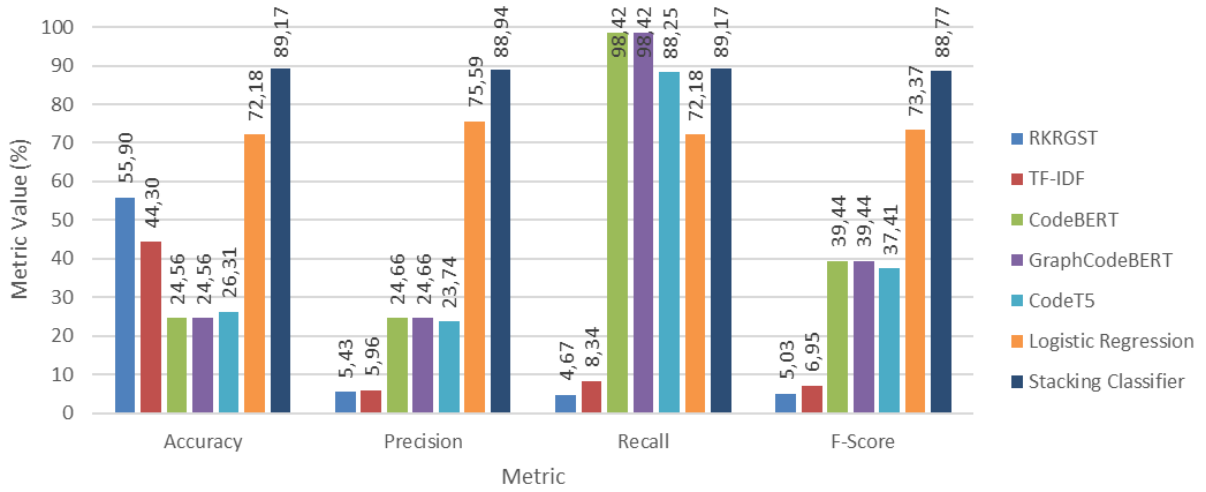

Fig. 4. Performance of baseline detectors with 75% similarity threshold compared to logistic regression and a stacking classifier

In general, while our plagiarism detector might not have the best recall, it outperformed baseline detectors in terms of accuracy, precision, and F-score. This indicated that structural similarities alone were unreliable for classifying GAI-disguised plagiarism at a particular similarity threshold. They should be considered as features for machine learning models. As program similarities might be substantially reduced due to GAI disguises, characteristic features were needed to complement them.

4.5. Discussion

Our study posed four research questions regarding the GAI-disguised plagiarism detector. RQ1 listed the disguises GAI commonly employed. Based on our observations, GAI behaved somewhat similarly to humans. They would focus on superficial disguises, such as renaming identifiers (variables and functions), changing comments, constants (variable values), and output formatting. However, unlike humans, the disguises were more pervasive and consistent.

GAI sometimes changed program structure, including flow, order, refactoring, and function calls. They could also modify the program semantics, though it was rare. The targeted components included data structure, visualization style, validation, and logging.

Our GAI-disguised plagiarism detector employed seven variants of machine learning models: Logistic Regression, Random Forest, XGBoost, LightGBM, CatBoost, Voting Classifier, and Stacking Classifier. According to the findings for RQ2, Logistic Regression was the least effective. The relationships between the features and the classes were not linear. Stacking Classifier, by contrast, achieved the best performance. This was expected since the Stacking Classifier combined the benefits of all base models to reduce false positives.

The GAI-disguised plagiarism detector employed 23 features. Based on our RQ3's observation, five features related to program similarities (structure) were considered important. While GAI disguises were pervasive and consistent, they did not change the whole program. Some parts were left as they were. Nevertheless, these features should be accompanied by features related to GAI characteristics.

Compared with the baseline detectors, as shown in RQ4's findings, our GAI-disguised plagiarism detector was twice as effective in terms of accuracy, precision, and F-score. However, it was less effective than transformer-based similarity methods in terms of recall.

4.6. Limitations

Our study has several limitations. First, the evaluation was conducted on students' programs for a particular course (machine intelligence) that had weekly assignments. Different findings may be obtained across courses or institutions, especially those with varying weekly assignment designs. Second, the detection was focused on programs written in Python. The findings could not be generalized to programs written in different programming languages, especially those with different programming styles. Program characteristic features need to be adjusted. Third, as GAI models evolve rapidly, detection performance might change over time. The research itself was conducted in the middle of 2025. Fourth, our study focuses on effectiveness, as that is the

core metric for measuring the usefulness of a GAI-disguised plagiarism detector. Another study focusing on computational cost on a larger dataset might be useful to complement the findings. Fifth, our detector is compared with the baseline detectors with structural features. Including other detectors with characteristic features can be useful for expanding the comparison.

5. CONCLUSIONS

This study presents a plagiarism detector to identify GAI-disguised plagiarism. GAI disguises tend to be more pervasive, negatively affecting the program's similarities. The detector employs five structural and 18 characteristic features. Structural features are derived from program similarities, while characteristic features are derived from syntax constructs commonly used by GAI. According to our evaluation, GAI often disguises programming plagiarism by altering program structure in addition to variable and function names. This complicates detection, as program similarities alone are not sufficient. GAI characteristics are needed to complement program similarities. Program similarities (structural features) focus on parts that remain unchanged or undergo only superficial changes, while GAI characteristics focus on GAI-disguised parts. Our plagiarism detector, which considers not only program similarities but also GAI characteristics, outperforms baseline detectors. A variant with the Stacking Classifier achieved an accuracy of 89.17%, a precision of 88.94%, a recall of 89.17%, and an F-Score of 88.77%.

For future work, we plan to test our plagiarism detector on more students' programs from different courses, preferably from other study programs and institutions. We are interested in considering programs resulting from case studies or group projects. Second, we are interested in expanding the programming language coverage of our plagiarism detector to enable broader impact. We might focus on Java or any programming language with different programming styles. Third, replicating the study several years later is preferable to account for the rapid evolution of GAI models. Fourth, an analysis of the detector's computational cost might be useful for scalability. Fifth, the detector can be compared with existing detectors based on their characteristic features.

Funding

This research was funded by the Institute of Research and Community Service, Maranatha Christian University, Indonesia.

Acknowledgments

To Ronaldo Khan Kashali for collecting part of the data set. To the Institute of Research and Community Services, Maranatha Christian University, Indonesia, for its support of the research.

Conflicts of Interest

The authors declare no conflict of interest.

REFERENCES

Aivaloglou, E., & Meulen, A. van der. (2021). An Empirical Study of Students' Perceptions on the Setup and Grading of Group Programming Assignments. *ACM Transactions on Computing Education (TOCE)*, *21*(3), 1–22. https://doi.org/10.1145/3440994

Albluwi, I. (2019). Plagiarism in programming assessments: a systematic review. *ACM Transactions on Computing Education*, *20*(1), 6:1-6:28. https://doi.org/10.1145/3371156

Allen, J. M., Vahid, F., Downey, K., & Edgcomb, A. D. (2018). Weekly programs in a CS1 class: experiences with auto-graded many-small programs (MSP). *ASEE Annual Conference \& Exposition*, 1–13. https://doi.org/10.18260/1-2--31231

Bandara, U., & Wijayarathna, G. (2011). A machine learning based tool for source code plagiarism detection. *International Journal of Machine Learning and Computing*, *1*(4), 337–343. https://doi.org/10.7763/IJMLC.2011.V1.50

Blanchard, J., Hott, J. R., Berry, V., Carroll, R., Edmison, B., Glassey, R., Karnalim, O., Plancher, B., & Russell, S. (2022). Stop reinventing the wheel! Promoting community software in computing education. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (pp. 261–292). Association for Computing Machinery. https://doi.org/10.1145/3571785.3574125

Bradley, S. (2020). Creative assessment in programming: Diversity and divergence. In *Proceedings of the Fourth Conference on Computing Education Practice* (Article 13). Association for Computing Machinery. https://doi.org/10.1145/3372310.3372325

Bubenkova, L., Pietrikova, E., & Horvath, M. (2025). Code reuse and good clones in programming education. In *2025 IEEE 23rd International Symposium on Applied Machine Intelligence and Informatics (SAMI)* (pp. 401–406). IEEE. https://doi.org/10.1109/SAMI63904.2025.10883291

Bulla, L., Midolo, A., Mongiovì, M., & Tramontana, E. (2024). EX-CODE: A robust and explainable model to detect AI-generated code. *Information*, *15*(12), Article 819. https://doi.org/10.3390/info15120819

Cendrowski, H., & Martin, J. (2015). The fraud triangle. In H. Cendrowski & J. Martin (Eds.), *The handbook of fraud deterrence* (pp. 41–46). John Wiley & Sons. https://doi.org/10.1002/9781119202165.ch5

Cheers, H., Lin, Y., & Smith, S. P. (2021). Academic source code plagiarism detection by measuring program behavioral similarity. *IEEE Access*, *9*, 50391–50412. https://doi.org/10.1109/ACCESS.2021.3069367

Duracik, M., Hrkut, P., Krsak, E., & Toth, S. (2020). Abstract syntax tree based source code antiplagiarism system for large projects set. *IEEE Access*, *8*, 175347–175359. https://doi.org/10.1109/ACCESS.2020.3026422

Ebrahim, F., & Joy, M. (2024). Semantic similarity search for source code plagiarism detection: An exploratory study. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education (ITiCSE)* (Vol. 1, pp. 360–366). Association for Computing Machinery. https://doi.org/10.1145/3649217.3653622

Eppa, A., & Murali, A. H. (2021). Machine learning techniques for multisource plagiarism detection. In *2021 5th International Conference on Computational Systems and Information Technology for Sustainable Solutions (CSITSS)*. IEEE. https://doi.org/10.1109/CSITSS54238.2021.9683752

Eppa, A., & Murali, A. (2022). Source code plagiarism detection: A machine intelligence approach. In *2022 4th International Conference on Advances in Electronics, Computers and Communications (ICAECC)*. IEEE. https://doi.org/10.1109/ICAECC54045.2022.9716671

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547). Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Fokam, M. A., & Ajoodha, R. (2021). Influence of contrastive learning on source code plagiarism detection through recursive neural networks. In *2021 3rd International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*. IEEE. https://doi.org/10.1109/IMITEC52926.2021.9714688

Foltýnek, T., Všianský, R., Meuschke, N., Dlabolová, D., & Gipp, B. (2020). Cross-language source code plagiarism detection using explicit semantic analysis and scored greedy string tilling. In *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020* (pp. 523–524). Association for Computing Machinery. https://doi.org/10.1145/3383583.3398594

Fowler, M., & Zilles, C. (2021). Superficial code-guise: Investigating the impact of surface feature changes on students' programming question scores. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (pp. 3–9). Association for Computing Machinery. https://doi.org/10.1145/3408877.3432420

Fowler, M., Smith, D. H., & Zilles, C. (2024). Quickly producing 'isomorphic' exercises: Quantifying the impact of programming question permutations. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education (ITiCSE)* (Vol. 1, pp. 178–184). Association for Computing Machinery. https://doi.org/10.1145/3649217.3653617

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2020). *GraphCodeBERT: Pre-training code representations with data flow*. *ArXiv, abs/2009.08366*. https://arxiv.org/abs/2009.08366

Hawlitschek, A., Berndt, S., & Schulz, S. (2023). Empirical research on pair programming in higher education: A literature review. *Computer Science Education*, *33*(3), 400–428. https://doi.org/10.1080/08993408.2022.2039504

Hoq, M., Shi, Y., Leinonen, J., Babalola, D., Lynch, C., Price, T., & Akram, B. (2024). Detecting ChatGPT-generated code submissions in a CS1 course using machine learning models. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education* (Vol. 1, pp. 526–532). Association for Computing Machinery. https://doi.org/10.1145/3626252.3630800

Jovanovic, M., & Campbell, M. (2022). Generative artificial intelligence: Trends and prospects. *Computer*, *55*(10), 107–112. https://doi.org/10.1109/MC.2022.3192720

Karnalim, O., & Kurniawati, G. (2020). Programming style on source code plagiarism and collusion detection. *International Journal of Computing*, *19*(1), 27–38. https://doi.org/10.47839/ijc.19.1.1691

Karnalim, O. (2023). Maintaining academic integrity in programming: Locality-sensitive hashing and recommendations. *Education Sciences*, *13*(1), Article 54. https://doi.org/10.3390/educsci13010054

Karnalim, O., Simon, & Chivers, W. (2023). Gamification to help inform students about programming plagiarism and collusion. *IEEE Transactions on Learning Technologies*, *16*(5), 1–14. https://doi.org/10.1109/TLT.2023.3243893

Karnalim, O., Toba, H., & Johan, M. C. (2024). Detecting AI assisted submissions in introductory programming via code anomaly. *Education and Information Technologies*, *29*(13), 16841–16866. https://doi.org/10.1007/s10639-024-12520-6

Karnalim, O. (2025). Identifying AI generated code with parallel KNN weight outlier detection. In *Lecture Notes in Networks and Systems* (Vol. 1140, pp. 459–470). Springer. https://doi.org/10.1007/978-3-031-71530-3_29

Kosmyna, N., Hauptmann, E., Yuan, Y. T., Situ, J., Liao, X.-H., Beresnitzky, A. V., Braunstein, I., & Maes, P. (2025). *Your brain on ChatGPT: Accumulation of cognitive debt when using an AI assistant for essay writing task*. *ArXiv, abs/2506.08872*. https://doi.org/10.48550/arXiv.2506.08872

Li, S., Liu, J., & Dong, Q. (2025). Generative artificial intelligence-supported programming education: Effects on learning performance, self-efficacy and processes. *Australasian Journal of Educational Technology*, *41*(3), 1–25. https://doi.org/10.14742/ajet.9932

Ljubovic, V., & Pajic, E. (2020). Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories. *IEEE Access*, *8*, 96505–96514. https://doi.org/10.1109/ACCESS.2020.3000523

Maertens, R., Van Neyghem, M., Geldhof, M., Van Petegem, C., Strijbol, N., Dawyndt, P., & Mesuere, B. (2024). Discovering and exploring cases of educational source code plagiarism with Dolos. *SoftwareX*, *26*, Article 101755. https://doi.org/10.1016/j.softx.2024.101755

Mason, T., Gavrilovska, A., & Joyner, D. A. (2019). Collaboration versus cheating: Reducing code plagiarism in an online MS computer science program. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 1004–1010). Association for Computing Machinery. https://doi.org/10.1145/3287324.3287443

Nguyen, P. T., Di Rocco, J., Di Sipio, C., Rubei, R., Di Ruscio, D., & Di Penta, M. (2024). GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT. *Journal of Systems and Software*, *214*, Article 112059. https://doi.org/10.1016/j.jss.2024.112059

Novak, M., Joy, M., & Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Transactions on Computing Education*, *19*(3), 1–37. https://doi.org/10.1145/3313290

Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

Parthasarathy, P. D., Kapoor, I., Joshi, S., & Thomas, S. (2024). Influence of personality traits on plagiarism through collusion in programming assignments. In *Proceedings of the 2024 ACM Conference on International Computing Education Research* (Vol. 1, pp. 143–153). Association for Computing Machinery. https://doi.org/10.1145/3632620.3671121

Pham, H., Ha, H., Tong, V., Hoang, D., Tran, D., & Le, T. N. (2024). MAGECODE: Machine-generated code detection method using large language models. *IEEE Access*, *12*, 190186–190202. https://doi.org/10.1109/ACCESS.2024.3509987

Pudasaini, S., Miralles-Pechuán, L., Lillis, D., & Llorens Salvador, M. (2024). Survey on AI-generated plagiarism detection: The impact of large language models on academic integrity. *Journal of Academic Ethics*, *23*(3), 1137–1170. https://doi.org/10.1007/s10805-024-09576-x

Ryman, D., Imbrie, P. K., & Kastner, J. (2022). Enhancement of plagiarism detection techniques via watermarking. In *2022 IEEE Frontiers in Education Conference (FIE)*. IEEE. https://doi.org/10.1109/FIE56618.2022.9962396

Sağlam, T., Hahner, S., Schmid, L., & Burger, E. (2024). Obfuscation-resilient software plagiarism detection with JPlag. In *Proceedings of the 2024 International Conference on Software Engineering* (pp. 264–265). Association for Computing Machinery. https://doi.org/10.1145/3639478.3643074

Schneider, J., Bernstein, A., vom Brocke, J., Damevski, K., & Shepherd, D. C. (2018). Detecting plagiarism based on the creation process. *IEEE Transactions on Learning Technologies*, *11*(3), 348–361. https://doi.org/10.1109/TLT.2017.2705056

Sharma, N., Shinde, S., Bhosale, S., & Patil, S. (2024). SourcePlag: Source code plagiarism detection based on abstract syntax trees. In *2024 IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS)*. IEEE. https://doi.org/10.1109/ICBDS61829.2024.10837209

Sheahen, D., & Joyner, D. (2016). TAPS: A MOSS extension for detecting software plagiarism at scale. In *Proceedings of the Third (2016) ACM Conference on Learning @ Scale* (pp. 285–288). Association for Computing Machinery. https://doi.org/10.1145/2876034.2893435

Simon. (2017). Designing programming assignments to reduce the likelihood of cheating. In *Proceedings of the 19th Australasian Computing Education Conference* (pp. 42–47). Association for Computing Machinery. https://doi.org/10.1145/3013499.3013506

Simon, Sheard, J., Morgan, M., Petersen, A., Settle, A., & Sinclair, J. (2018). Informing students about academic integrity in programming. In *Proceedings of the 20th Australasian Computing Education Conference* (pp. 113–122). Association for Computing Machinery. https://doi.org/10.1145/3160489.3160502

Spacco, J., Fossati, D., Stamper, J., & Rivers, K. (2013). Towards improving programming habits to create better computer science course outcomes. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (pp. 243–248). Association for Computing Machinery. https://doi.org/10.1145/2462476.2462483

Surahman, E., & Wang, T. H. (2022). Academic dishonesty and trustworthy assessment in online learning: A systematic literature review. *Journal of Computer Assisted Learning*, *38*(6), 1535–1553. https://doi.org/10.1111/jcal.12708

Toba, H., Karnalim, O., Johan, M. C., Tada, T., Djajalaksana, Y. M., & Vivaldy, T. (2023). Inappropriate benefits and identification of ChatGPT misuse in programming tests: A controlled experiment. In *Proceedings of the International Conference on Interactive Collaborative Learning* (pp. 520–531). Springer. https://doi.org/10.1007/978-3-031-52667-1_50

Toba, H., & Karnalim, O. (2025). Machine learning models to detect AI-assisted code anomaly in introductory programming course. In *Lecture Notes in Networks and Systems* (Vol. 1140, pp. 163–181). Springer. https://doi.org/10.1007/978-3-031-71530-3_11

Tsang, H. H., Hanbidge, A. S., & Tin, T. (2018). Experiential learning through inter-university collaboration research project in academic integrity. In *Proceedings of the 23rd Western Canadian Conference on Computing Education*. Association for Computing Machinery. https://doi.org/10.1145/3209635.3209641

Ullah, F., Wang, J., Farhan, M., Habib, M., & Khalid, S. (2018). Software plagiarism detection in multiprogramming languages using machine learning approach. *Concurrency and Computation: Practice and Experience*, *30*(21), e5000. https://doi.org/10.1002/cpe.5000

Ullah, F., Jabbar, S., & Mostarda, L. (2021). An intelligent decision support system for software plagiarism detection in academia. *International Journal of Intelligent Systems*, *36*(6), 2730–2752. https://doi.org/10.1002/int.22399

Viuginov, N., Grachev, P., & Filchenkov, A. (2020). A machine learning based plagiarism detection in source code. In *Proceedings of the 3rd International Conference on Algorithms, Computing and Artificial Intelligence* (pp. 1–6). Association for Computing Machinery. https://doi.org/10.1145/3446132.3446420

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing* (pp. 8696–8708). Association for Computational Linguistics. https://doi.org/10.18653/v1/2021.emnlp-main.685

Xie, Y., Wu, S., & Chakravarty, S. (2023). AI meets AI: Artificial intelligence and academic integrity - A survey on mitigating AI-assisted cheating in computing education. In *Proceedings of the 24th Annual Conference on Information Technology Education* (pp. 79–83). Association for Computing Machinery. https://doi.org/10.1145/3585059.3611449

Xu, Z., & Sheng, V. S. (2024). Detecting AI-generated code assignments using perplexity of large language models. *AAAI Conference on Artificial Intelligence*, *38*(21), 23155–23162. https://doi.org/10.1609/aaai.v38i21.30361

Xu, X., Ni, C., Guo, X., Liu, S., Wang, X., Liu, K., & Yang, X. (2025). Distinguishing LLM-generated from human-written code by contrastive learning. *ACM Transactions on Software Engineering and Methodology*, *34*(4), Article 100. https://doi.org/10.1145/3705300

Yasaswi, J., Kailash, S., Chilupuri, A., Purini, S., & Jawahar, C. V. (2017). Unsupervised learning based approach for plagiarism detection in programming assignments. In *Proceedings of the 10th Innovations in Software Engineering Conference* (pp. 117–121). Association for Computing Machinery. https://doi.org/10.1145/3021460.3021477

Yasaswi, J., Purini, S., & Jawahar, C. V. (2017). Plagiarism detection in programming assignments using deep features. In *2017 4th IAPR Asian Conference on Pattern Recognition (ACPR)* (pp. 652–657). IEEE. https://doi.org/10.1109/ACPR.2017.146

Zhang, Z., & Saber, T. (2025). Machine learning approaches to code similarity measurement: A systematic review. *IEEE Access*, *13*, 51729–51764. https://doi.org/10.1109/ACCESS.2025.3553392

Zhou, Z.-H. (2021). *Machine learning*. Springer. https://doi.org/10.1007/978-981-15-1967-3