Journal of Applied Informatics and Computing (JAIC)
Vol.9, No.6, December 2025, pp. 3079~3086
e-ISSN: 2548-6861

Identification of Source Code Plagiarism Using a Natural Language Processing (NLP) Approach Based on Code Writing Style Analysis

Muhammad Ilham Akbar [1], Novita Kurnia Ningrum [2]
Teknik Informatika, Universitas Dian Nuswantoro
111202214109@mhs.dinus.ac.id [1], novita.kn@dsn.dinus.ac.id [2]

Article Info

Article history:

Received 2025-09-13
Revised 2025-11-11
Accepted 2025-11-15

Keyword:

Code Plagiarism,
CodeBert,
Siamese Network,
Deep Learning,
Source Code Plagiarism,
Identification.

ABSTRACT

Source code plagiarism identificatio requires a system capable of identifying semantic similarity rather than mere textual resemblance. This study utilized a dataset of 1,000 source code files, which after cleaning resulted in 996 individual code samples collected from GitHub repositories. The dataset included various programming languages (Python, Java, JavaScript, TypeScript, C++), divided into 697 training data, 149 validation data, and 149 testing data. The model employed was CodeBERT, configured with a hidden size of 768, 12 layers, and 12 attention heads. CodeBERT generated vector embeddings for each code sample, which were then projected by a Siamese Network to calculate cosine similarity between code pairs. Testing used a threshold of 0.80 to classify plagiarism. The identification results achieved an accuracy of 96.4%, precision of 95.2%, recall of 97.8%, F1-score of 96.4%, and an error rate of 4.6%. The system produced similarity scores and status labels of "plagiarism detected" or "not detected," demonstrating the effectiveness of the CodeBERT-based approach for adaptive and intelligent code similarity identificatio.

This is an open access article under the CC–BY-SA license.

I. Introduction

Source code plagiarism is a serious issue within academic environments, particularly in the fields of informatics and computer science. Various studies have reported that this unethical practice has become increasingly widespread in the digital era, driven by the ease of access to online repositories and the low awareness of academic integrity among students [5], [15], [21]. Its impact is multidimensional: for students, plagiarism hinders conceptual understanding and reduces authentic programming competence; while for institutions, the prevalence of such violations can undermine academic credibility and erode public trust in the quality of graduates. According to Dickey [23], "Plagiarism in CS education is unfortunately common. Surveys of students have repeatedly shown a significant majority admit to some form of code copying or undue collaboration. For instance, in a large lower-division CS course of 200–300 students, teaching staff typically discovered 20–40 blatant cases of code plagiarism each semester. These confirmed cases (around 10–15% of the class) represent a lower bound, as instructors often focus only on the most obvious instances and ignore cases with plausible deniability. In other words, many subtle or well-disguised code copying incidents go undetected or ignored under current practices. Other studies confirm this lower bound, with some as high as 75%. The true incidence of plagiarism is thus suspected to be higher, posing a serious threat to the fairness and educational validity of programming assessments and CS degrees."This quotation indicates that the level of plagiarism in computer science education is considerably high and often not fully detected, since most existing systems are limited to identifying textual similarities. Therefore, code plagiarism identificatio requires a more contextual and semantic approach rather than merely comparing character- or token-level similarities.

A number of methods have been developed to detect source code similarity; however, each has its own limitations. Traditional approaches such as Winnowing and TF-IDF with Cosine Similarity [1], [2], [22] are capable of identifying textual similarities but fail to capture the semantic and functional meaning of programs [9]. These systems operate only at the lexical level, making them easily deceived by syntactic modifications that do not alter the underlying program logic [12], [14].Recent studies have shown that deep learning–based approaches can overcome these limitations by understanding the functional context and semantic relationships between lines of code. Models such as the Siamese Network have proven effective in comparing code representations that convey similar meanings despite differences in structure or writing style [3], [18]. Meanwhile, pre-trained models such as CodeBERT are designed to comprehend semantic representations across programming languages, producing embeddings that are robust to variations in syntax and enabling more accurate identificatio of code similarity [4], [6], [11], [20].

This study identifies source code plagiarism using a deep learning–based approach with a focus on analyzing programming style (coding style). The dataset consists of 1,000 source code projects collected from GitHub repositories, covering five major programming languages: Python, Java, JavaScript, TypeScript, and C++. The data are divided into 700 training samples, 150 validation samples, and 150 testing samples to ensure result generalization.The system process comprises four main stages: (1) code collection and preprocessing, including the removal of comments, license headers, and structural normalization; (2) semantic representation using CodeBERT with a hidden size of 768 and 12 attention heads; (3) mapping of the two representations into a lower-dimensional vector space using a Siamese Network architecture with contrastive loss; and (4) calculation of similarity levels using cosine similarity values.The system produces outputs in the form of a similarity percentage between two code segments along with a status label "plagiarism detected" or "no plagiarism detected." Experimental results demonstrate that this approach can effectively measure semantic code similarity with high accuracy and low error rates, while also reinforcing the potential of programming style analysis as a unique digital identity to support academic integrity.

In general, the identification system operates by uploading two or more program code files, which then undergo a preprocessing stage to remove comments, excessive spaces, and irrelevant string literals, as well as to normalize variable and function names to reduce lexical bias. Subsequently, the cleaned code is tokenized using CodeBERT's built-in tokenizer. The feature extraction stage maps the tokens into high-dimensional vector representations that capture both syntactic and semantic information. The extracted vectors are then processed by a Siamese Network to compute the similarity level between code snippets, producing similarity scores ranging from 0 to 1. The final stage presents the results in the form of a similarity percentage along with a plagiarism status label.

The objective of this study is to develop an NLP-based source code plagiarism identification system that integrates CodeBERT and a Siamese Network to identify code similarity even when modifications are made to programming style aspects such as variable renaming, comment insertion, or syntactic reformatting. The system is designed to generate embeddings that capture both syntactic and semantic information, directly compare pairs of code snippets to obtain similarity scores, and evaluate the effectiveness of the proposed method on a multi-language programming dataset. Accordingly, this research is expected to assist lecturers and academic institutions in detecting indications of source code plagiarism and upholding academic integrity within higher education environments.

II. METHODS

This research is designed with a systematic workflow consisting of several main stages, starting from data collection and code preprocessing to feature extraction, model training, and evaluation. Each stage plays a crucial role in establishing a solid methodological foundation to ensure results that are accurate, measurable, and reproducible.

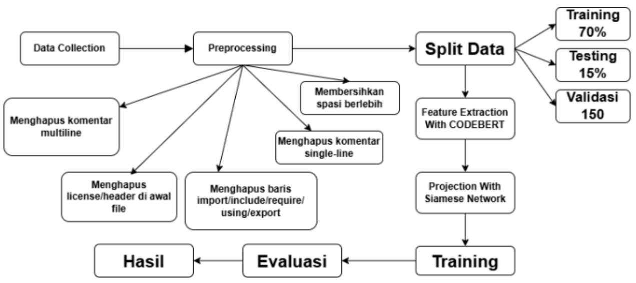
Data Collection
Preprocessing
Split Data
Training 70%
Testing 15%
Validasi 150
Membersihkan spasi berlebih
Menghapus komentar multiline
Menghapus komentar single-line
Menghapus license/header di awal file
Menghapus baris import/include/require/using/export
Feature Extraction With CODEBERT
Projection With Siamese Network
Training
Evaluasi
Hasil

Figure 1. Research workflow

To provide an overview of the research workflow, Figure 1 presents the flowchart of the proposed research methodology. The diagram visually illustrates the relationships between each stage, facilitating the reader's understanding of the entire process—from raw data collection to the evaluation of the plagiarism identification model.

A. Data Collection and Preparation

The dataset was obtained from public GitHub repositories using the GitHub API with token-based authentication. The search focused on simple algorithms such as Fibonacci, Greatest Common Divisor (GCD), Least Common Multiple (LCM), calculator, and sorting algorithms, implemented in five popular programming languages: Python, Java, JavaScript, TypeScript, and C++.

Each retrieved file was centrally stored in the directory /content/drive/MyDrive/Jurnal/dataseyrrts2. The total dataset initially contained 1,000 source code files, each exhibiting diverse structures and coding styles. However, after the data cleaning process, 996 files remained.

For experimental purposes, code pairs were divided into two categories:

- Plagiarized pairs — code pairs that share identical logic or algorithms but differ in writing style, such as variable renaming, indentation changes, or the addition of comments.

- Non-plagiarized pairs — code pairs with different functions and logical structures, derived from non-equivalent algorithm implementations.

The pairing process was conducted semi-manually, considering algorithmic similarity and program logic structure rather than mere character-level resemblance. This process involved analyzing the functional context and execution sequence of the programs to ensure that the plagiarized pairs authentically represented conceptual similarity, while the non-plagiarized pairs reflected clear semantic differences.

TABLE I
DATASET INFORMATION

No	Programming Language	Start Year	Updated Year	Total Files
1	Python	2020	2025	387
2	Typescript	2020	2025	239
3	Java	2020	2025	220
4	Javascript	2020	2025	143
5	C++	2022	2024	10
	Total Files			1000

B. Pra-Pemrosesan Data

The preprocessing stage aimed to ensure format consistency and reduce noise without removing distinctive stylistic features of the source code. A light cleaning approach was applied, consisting of the following steps.

1. Comment Removal

The first step was to remove comments from the code. Comments were categorized into two types:

- Multi-line comments, such as /* ... */, """...""", or '''...'''
- Single-line comments, such as // ..., # ..., or -- ...

Comments were removed because they serve only as decorative explanations that do not affect the core logic of the program. Moreover, comments can easily be manipulated to disguise plagiarism without changing the execution flow. By removing comments, the analysis focuses on executable code, leading to more objective identification results.

2. License Header Removal

The second step was to remove license headers, which typically appear at the beginning of source files—usually within the first ten lines. These headers often contain keywords such as *copyright*, *license*, *MIT*, *Apache*, or *GPL.*License information was excluded because it is irrelevant to a programmer's coding style and often identical across many projects, potentially creating false similarities.By removing license headers, the system can better focus on the unique stylistic aspects of each programmer's code.

TABLE II
BEFORE AND AFTER COMMENT REMOVAL

Before	After
#==============	#==============
# Copyright (c) 2024	# Copyright (c) 2024
# License: MIT	# License: MIT
import os	import os
import sys # system library	import sys # system
	library
'''	
Fungsi untuk menghitung	def factorial(n): #
faktorial	fungsi utama
dengan metode rekursif	if n == 0:
'''	return 1
	else:
def factorial(n): # fungsi utama	return n *
if n == 0:	factorial(n-1) # rekursi
return 1	
else:	print(factorial(5)) #
return n * factorial(n-1) #	test
rekursi	
print(factorial(5)) # test	

TABLE III
BEFORE AND AFTER LICENSE HEADER REMOVAL

Before	After
#==============	import os
# Copyright (c) 2024	import sys
# License: MIT	
	def factorial(n):
import os	if n == 0:
import sys # system library	return 1
	else:
'''	return n *
Fungsi untuk menghitung	factorial(n-1)
faktorial	
dengan metode rekursif	print(factorial(5))
'''	
def factorial(n): # fungsi utama	
if n == 0:	
return 1	
else:	
return n * factorial(n-1) #	
rekursi	
print(factorial(5)) # test	

3. Removal of Import/Include/Require/Using/Export Statements

The next step was to remove lines containing external library declarations, such as import, from, include, require, using, and export. These lines do not contribute to the program's logical writing style but merely indicate dependencies. Since import patterns tend to be highly uniform, they were considered noise in the stylistic analysis.By eliminating these statements, preprocessing became more effective in highlighting features that truly reflect the programmer's unique coding style.

TABLE IV
BEFORE AND AFTER IMPORT/INCLUDE/REQUIRE/USING/EXPORT REMOVAL

Before	After
#==============	def factorial(n):
# Copyright (c) 2024	if n == 0:
# License: MIT	return 1
	else:
import os	return n *
import sys # system library	factorial(n-1)
'''	print(factorial(5))
Fungsi untuk menghitung faktorial	
dengan metode rekursif	
'''	
def factorial(n): # fungsi utama	
if n == 0:	
return 1	
else:	
return n * factorial(n-1) # rekursi	
print(factorial(5)) # test	

4. Whitespace Normalization

The fourth step involved whitespace normalization. All excessive spaces, tabulations, and consecutive blank lines were reduced to a single space or a single blank line. The purpose of this step was to minimize formatting differences that are purely cosmetic, such as indentation variations, line spacing, or extra spaces.Through normalization, codes with different visual formats but identical logic were treated as equivalent, preventing superficial differences that do not actually reflect distinct programming styles.

TABLE V
BEFORE AFTER NORMALISASI SPASI

Before	After
#==============	def factorial(n):
# Copyright (c) 2024	if n == 0:
# License: MIT	return 1
	else:
import os	return n *
import sys # system library	factorial(n-1)
	print(factorial(5))
'''	
Fungsi untuk menghitung faktorial	
dengan metode rekursif	
'''	
def factorial(n): # fungsi utama	
if n == 0:	
return 1	
else:	
return n * factorial(n-1) # rekursi	
print(factorial(5)) # test	

5. Preservation of Core Writing Style

The final stage focused on preserving the authenticity of the programmer's writing style. Preprocessing intentionally retained variable and function names such as myVariable, temp, or calculateAverage because naming choices reflect an individual's stylistic tendencies.Moreover, the code structure, including the order of statements, looping patterns, conditional expressions, and operator styles (e.g., ++i vs i++), was preserved. Even unused library imports were not removed if they indicated habitual patterns.Thus, preprocessing only removed irrelevant elements comments, license headers, import statements, and redundant spaces while maintaining the original stylistic features of the source code.

TABLE VI
BEFORE AND AFTER PRESERVATION OF WRITING STYLE

Before	After
#==============	def factorial(n):
# Copyright (c) 2024	if n == 0:
# License: MIT	return 1
	else:
import os	return n *
import sys # system library	factorial(n-1)
	print(factorial(5))
'''	
Fungsi untuk menghitung faktorial	
dengan metode rekursif	
'''	
def factorial(n): # fungsi utama	
if n == 0:	
return 1	
else:	
return n * factorial(n-1) # rekursi	
print(factorial(5)) # test	

After the cleaning and normalization stages, the dataset was divided into three subsets using the train_test_split function from the *scikit-learn* library, with proportions of 70% training, 15% testing, and 15% validation, and the parameter random_state=42 to ensure reproducibility.

Data splitting was performed on the preprocessed source code level, rather than on the embedding representations, to preserve the distribution of coding styles and algorithmic complexity across all subsets.

This strategy aligns with the principles outlined by Joseph and Vakayil [7], who argue that there is no universal split ratio—the optimal proportion depends on dataset size and model complexity to minimize generalization error.

Furthermore, the CodeXGLUE benchmark framework [8] adopts similar approaches in various tasks, such as code clone identificatio and code summarization, ensuring fair and comparable model evaluation across studies.

Such data-splitting strategies have become a standard practice in deep learning experiments on source code [4], [6], [11], [18].

Therefore, the 70:15:15 split was chosen as it provides a balanced trade-off between training size and evaluation reliability, enabling the model to capture diverse writing styles while maintaining robust validation and testing performance.

C. Model Architecture and Algorithm

This study employed a deep learning approach based on CodeBERT + Siamese Network to detect source code similarity.

1. Feature Extraction Using CodeBERT

Each code snippet was tokenized using the built-in tokenizer from CodeBERT (microsoft/codebert-base). The model generated 768-dimensional embeddings with the following parameters:

- Hidden size: 768
- Layers: 12
- Attention heads: 12
- Max token length: 512
- Mode: partial fine-tuning — the final layer of CodeBERT is fine-tuned for task-specific adaptation while maintaining the semantic stability of the earlier frozen layers.

The [CLS] token embedding of each code snippet was used as the vector representation h, then normalized using L2 normalization:

$$e = \frac{h}{|h|_2}, e \in R^{768}$$

where h is the raw [CLS] token vector and $\|h\|_2$ is its L2 norm.This normalization step is standard in contrastive learning pipelines [7].

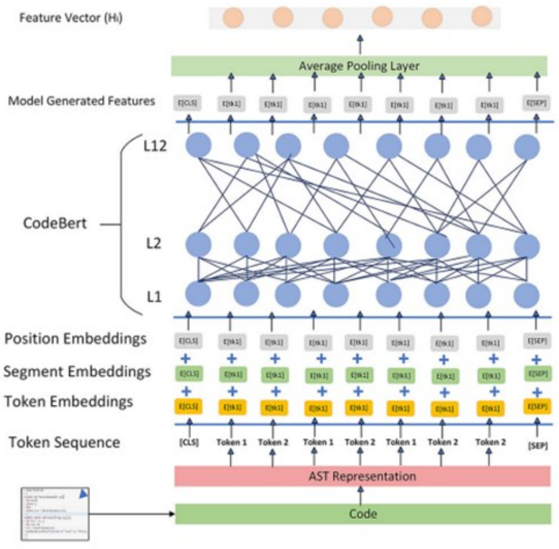

Figure 2. Workflow of vector embedding computation

2. Projection Using Siamese Network

Two embedding vectors (e_1 and e_2) were passed through two identical branches of a Siamese Network with the following architecture:

Linear (768→256) → ReLU → Dropout(0.3) →LinearLayer(256→128).

Both branches share weights (shared parameters). The output is a 128-dimensional latent vector representing the final semantic representation for each code snippet.

3. Training with CosineEmbedding Loss

The model was trained using CosineEmbeddingLoss, which encourages similar code vectors to be close together and dissimilar vectors to move farther apart. The loss function is defined as:

$$L(x_1, x_2, y) = \begin{cases} \frac{1 - S\,(z_1, z_2)}{\max(0, S(z_1, z_2) - m),} & \frac{y = 1\ (\text{mirip})}{y = -1\ (\text{tidak mirip})} \end{cases}$$

where z_1 and z_2 are the latent vectors of code pairs, and m is the margin for dissimilar pairs.

4. Cosine Similarity Evaluation and Thresholding

After training, the similarity between two code snippets was measured using cosine similarity:

$$S(z_1, z_2) = \frac{z_1 . z_2}{||z_1||z_2|}$$

A similarity score $S \geq 0.80$ was categorized as plagiarized, while lower scores were considered non-plagiarized.Model performance was evaluated using accuracy, precision, recall, and F1-score, along with error analysis to identify false positives and false negatives. To ensure evaluation robustness, the dataset was split into training (70%), validation (15%), and test (15%) sets, with the validation set used for threshold optimization and early stopping guidance.

III. Results and Discussion

A. CodeBERT Mechanism and Parameters

CodeBERT operates based on masked language modeling (MLM) and replaced token detection (RTD) to learn relationships between program tokens and their semantic contexts.In this study, the base model microsoft/codebert-base was used with the parameters shown below.

TABLE VII
MODEL PARAMETER

Parameter	Value	Description
Model Type	CodeBERT (base) + Siamese Network	Transformer with 12 layers, 768 hidden size, and Siamese head
Tokenizer	RobertaTokenizer	Tokenizer from microsoft/codebert-base

Max Sequence Length	512	Maximum token length per code snippet
Batch Size	8	Number of code pairs per iteration
Learning Rate	2e-5	Optimizer: AdamW
Epoch	5	Number of training cycles
Loss Function	CosineEmbeddingLoss	Distinguishes between plagiarized and non-plagiarized pairs
Hidden Dim (Siamese)	256	Dimension of the Siamese hidden layer
Output Dim (Siamese)	128	Dimension of the Siamese output embedding
Dropout Rate	0.3	Dropout applied to the Siamese head
Training Strategy	Partial Fine-tuning	The final layer of CodeBERT is fine-tuned for task adaptation
Similarity Threshold	0.80	Classification threshold

CodeBERT produced 768-dimensional embeddings that capture writing style, structure, and logical meaning of source code.These embeddings were then used as inputs for the Siamese Network to calculate pairwise distances.

B. Siamese Network Mechanism

The Siamese Network receives two embeddings (emb_1, emb_2) from CodeBERT and computes the cosine similarity to determine semantic similarity:

$$\text{similarity}(A, B) = \frac{A \cdot B}{\parallel A \parallel \parallel B \parallel}$$

Similarity scores range from 0 to 1:

- 0 → dissimilar
- 1 → highly similar

The model is trained to yield high similarity scores for plagiarized pairs and low scores for non-plagiarized ones. The final output is a similarity percentage, which is then labeled as "Plagiarized" or "Non-plagiarized" based on the chosen threshold.

C. Threshold Determination and Rationale

The threshold value of 0.80 was selected empirically after testing multiple thresholds from 0.4 to 0.9.

TABEL VIII
MULTI-THRESHOLD EVALUATION OF THE CODEBERT–SIAMESE MODEL

Threshold	Accuracy	Precision	Recall	F1-Score
0.20	0.946	0.899	0.998	0.946
0.25	0.946	0.905	0.998	0.949
0.30	0.946	0.905	0.998	0.949
0.35	0.946	0.905	0.998	0.949
0.40	0.945	0.904	0.996	0.948
0.45	0.949	0.910	0.996	0.951
0.50	0.951	0.914	0.996	0.953
0.55	0.952	0.916	0.996	0.954
0.60	0.954	0.919	0.996	0.956
0.65	0.954	0.926	0.987	0.956
0.70	0.955	0.932	0.982	0.956
0.75	0.960	0.940	0.982	0.961
0.80	0.964	0.952	0.978	0.964
0.85	0.963	0.964	0.962	0.963
0.90	0.965	0.981	0.948	0.964

Based on the evaluation results across multiple threshold values, it can be concluded that a threshold of 0.80 represents the optimal value for detecting source code plagiarism. This selection is grounded in achieving an ideal balance between precision and recall, which is crucial in academic contexts [1], [5]. A precision value of 95.2% at this threshold is particularly important for minimizing false positives, which could have serious implications for student integrity [5], [23]. Meanwhile, a recall value of 97.8% indicates the system's strong ability to detect the majority of actual plagiarism cases [3], [18].

An F1-score of 96.5% at the 0.80 threshold reflects a harmonious balance between precision and recall, following the principle of multi-objective optimization recommended in similarity detection studies [16]. Previous research suggests that too low a threshold often leads to high false-positive rates, while too high a threshold risks missing subtle plagiarism cases [1], [12]. The chosen threshold of 0.80 provides an optimal solution and aligns with related studies recommending thresholds within the 0.75–0.85 range for code plagiarism identificatio in educational settings [14], [15].

Therefore, the 0.80 threshold was selected as the optimal decision boundary, balancing accuracy and practicality in the implementation of a plagiarism identificatio system for source code, and adopted as the standard value in this study.

D. Comparison with AST- and TF-IDF-Based Methods

To highlight the research gap, three baseline approaches were compared:

Table IX
COMPARISON OF IDENTIFICATION RESULTS ACROSS METHODS

Method	**Accuracy**	**Precision**	**Recall**	**F1-Score**
winnowing	0.538	1.000	0.074	0.062
TF-IDF + Cosine	0.553	1.000	0.0187	0.122
AST Edit Distance	0.502	0.000	0.000	0.000
CodeBERT–Siamese	0.964	0.952	0.978	0.964

Analysis:

- The AST-based approach only captures syntactic structures and fails to detect cases where identical logic is written in a different order.
- The TF-IDF approach relies solely on token-level similarity and cannot understand semantic meaning.
- The CodeBERT–Siamese approach, on the other hand, captures both semantics and stylistic patterns, making it more adaptive to variations in coding styles.

E. Error Analysis

Based on test set results, the model produced an error rate of 4.6%, corresponding to 7 misclassifications out of 150 code pairs. These errors fall into two main categories:

1. False Positives (FP) — cases where two code snippets were classified as similar but were actually different. These errors often stem from structural similarities or shared templates in lab assignments [1], [2], where syntax-based similarity methods frequently misinterpret structural resemblance as semantic similarity.
2. False Negatives (FN) — cases where two genuinely similar code snippets were not detected as such. This typically occurs due to significant alterations in variable names, function orders, or comments, which modify the token representation and reduce semantic similarity [4], [12]. Such findings align with previous studies noting that surface-level structural changes can obscure underlying semantic equivalence.

The 0.80 threshold was determined through an empirical trade-off between precision (0.952) and recall (0.978). This value offers an optimal equilibrium between high recognition capability and low classification error, consistent with established guidelines for optimal threshold selection in Siamese networks [16].

Overall, the error rate below 5% demonstrates that the proposed model achieves high reliability in detecting semantic similarities between code fragments—consistent with recent studies on vector-based semantic plagiarism detection [6], [11], [20], [21].

IV. CONCLUSION

Based on the conducted experiments and analysis, it can be concluded that the CodeBERT–Siamese Network–based source code plagiarism identificatio system has been successfully developed, demonstrating high effectiveness in identifying semantic similarities between code fragments. The model achieved an accuracy of 96.4% with an error rate of 4.6% on a dataset comprising 996 pairs of multilingual source code.

The light cleaning approach applied during data preprocessing proved effective in maintaining essential coding-style characteristics while removing decorative or irrelevant elements such as comments, license headers, and import declarations. This strategy allowed the model to focus on logical structure and stylistic writing patterns, enhancing semantic representation accuracy.

Experimental results confirmed that the optimal threshold value of 0.80 provided the best balance between precision (95.2%) and recall (97.8%), minimizing false positives while maintaining sensitivity to subtle plagiarism cases. These findings highlight the superiority of the proposed method compared to traditional approaches like Winnowing and TF-IDF, which rely primarily on lexical similarity and fail to capture contextual semantics.

The integration of CodeBERT as a semantic feature extractor with a Siamese Network as a similarity learning model provides a significant advantage in handling syntactic variations and stylistic modifications. This combination makes the system more robust against plagiarism obfuscation techniques, such as variable renaming, comment insertion, or formatting changes.

In summary, this research contributes to the development of an adaptive and intelligent plagiarism identificatio system, with strong potential applications in academic integrity assurance, particularly within computer science and informatics education.

REFERENCES

[1] M. S. Ramli, S. Cokrowibowo, and M. F. Rustan, "Uji Plagiarism pada Tugas Mahasiswa Menggunakan Algoritma Winnowing," *J. Appl. Comput. Sci. Technol.*, vol. 2, no. 2, pp. 108–112, 2021, doi: 10.52158/jacost.v2i2.177.

[2] I. G. A. Eka Putra and I. W. Supriana, "Deteksi Plagiarisme Source Code Tugas Mahasiswa Menggunakan Algoritma Cosine Similarity Dan Pembobotan TF-IDF," *J. Nas. Teknol. Inf. dan Apl.*, vol. 1, no. 1, p. 575, 2022, [Online]. Available: https://ojs.unud.ac.id/index.php/jnatia/article/view/92871

[3] Di. K. Tankala, T. Venugopal, and B. Vikas, "Java Source Code Similarity Detection Using Siamese Networks," *J. Theor. Appl. Inf. Technol.*, vol. 100, no. 17, pp. 5507–5514, 2022.

[4] T. Sonnekalb, B. Gruner, C. A. Brust, and P. Mäder, "Generalizability of Code Clone Detection on CodeBERT," in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Sep. 2022. doi: 10.1145/3551349.3561165.

[5] M. A. Pratiwi and N. Aisya, "Fenomena plagiarisme akademik di era digital," *Publ. Lett.*, vol. 1, no. 2, pp. 16–33, 2021, doi: 10.48078/publetters.v1i2.23.

[6] S. Sahar, M. Younas, M. M. Khan, and M. U. Sarwar, "DP-CCL: A Supervised Contrastive Learning Approach Using CodeBERT Model in Software Defect Prediction," *IEEE Access*, vol. 12, no. January, pp. 22582–22594, 2024, doi: 10.1109/ACCESS.2024.3362896.

[7] V. R. Joseph and A. Vakayil, "SPlit: An Optimal Method for Data Splitting," *Technometrics*, vol. 64, no. 2, pp. 166–176, 2022, doi: 10.1080/00401706.2021.1921037.

[8] S. Lu *et al.*, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," *Adv. Neural Inf. Process. Syst.*, 2021.

[9] M. Sajid, M. Sanaullah, M. Fuzail, T. S. Malik, and S. M. Shuhidan, "Comparative analysis of text-based plagiarism detection techniques," *PLoS One*, vol. 20, no. 4 April, pp. 1–28, 2025, doi: 10.1371/journal.pone.0319551.

[10] V. R. Joseph, "Optimal ratio for data splitting," *Stat. Anal. Data Min.*, vol. 15, no. 4, pp. 531–538, 2022, doi: 10.1002/sam.11583.

[11] S. Arshad, S. Abid, and S. Shamail, "CodeBERT for Code Clone Detection: A Replication Study," *Proc. - 2022 IEEE 16th Int. Work. Softw. Clones, IWSC 2022*, pp. 39–45, 2022, doi: 10.1109/IWSC55060.2022.00015.

[12] F. Ebrahim and M. Joy, "Semantic Similarity Search for Source Code Plagiarism Detection: An Exploratory Study," *Annu. Conf. Innov. Technol. Comput. Sci. Educ. ITiCSE*, vol. 1, pp. 360–366, 2024, doi: 10.1145/3649217.3653622.

[13] A. Fedele, R. Guidotti, and D. Pedreschi, *Explaining Siamese networks in few-shot learning*, vol. 113, no. 10. Springer US, 2024. doi: 10.1007/s10994-024-06529-8.

[14] N. Gandhi, K. Gopalan, and P. Prasad, "A Support Vector Machine based approach for plagiarism detection in Python code submissions in undergraduate settings," *Front. Comput. Sci.*, vol. 6, 2024, doi: 10.3389/fcomp.2024.1393723.

[15] E. E. Htet *et al.*, “Code Plagiarism Checking Function and Its Application for Code Writing Problem in Java Programming Learning Assistant System †,” *Analytics*, vol. 3, no. 1, pp. 46–62, 2024, doi: 10.3390/analytics3010004.

[16] B. Kriuk and F. Kriuk, “Multi-Objective Optimal Threshold Selection for Similarity Functions in Siamese Networks for Semantic Textual Similarity Tasks,” 2024, doi: 10.20944/preprints202407.0020.v1.

[17] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, “GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT,” *J. Syst. Softw.*, vol. 214, no. August 2023, p. 112059, 2024, doi: 10.1016/j.jss.2024.112059.

[18] B. Wan, S. Dong, J. Zhou, and Y. Qian, “SJBCD: A Java Code Clone Detection Method Based on Bytecode Using Siamese Neural Network,” *Appl. Sci.*, vol. 13, no. 17, 2023, doi: 10.3390/app13179580.

[19] M. A. Yahya and D. K. Kim, “CLCD-I: Cross-Language Clone Detection by Using Deep Learning with InferCode,” *Computers*, vol. 12, no. 1, pp. 1–11, 2023, doi: 10.3390/computers12010012.

[20] M. Zubkov, E. Spirin, E. Bogomolov, and T. Bryksin, *Evaluation of Contrastive Learning with Various Code Representations for Code Clone Detection*, vol. 1, no. 1. Association for Computing Machinery, 2022. doi: 10.2139/ssrn.4159812.

[21] R. Maertens *et al.*, “Discovering and exploring cases of educational source code plagiarism with Dolos,” *SoftwareX*, vol. 26, no. May, p. 101755, 2024, doi: 10.1016/j.softx.2024.101755.

[22] A. Y. Bramantya, T. Hasanuddin, and F. Umar, “Analisis Metode Winnowing Dalam Pendeteksian Plagiarisme Judul,” *Bul. Sist. Inf. dan Teknol. Islam*, vol. 3, no. 4, pp. 268–273, 2022, doi: 10.33096/busiti.v3i4.1469.

[23] E. Dickey, “The Failure of Plagiarism Detection in Competitive Programming,” 2025, [Online]. Available: http://arxiv.org/abs/2505.08244

[24] W. Yang, “Identification and Prevention of Code Open Source Quotation and Plagiarism — Innovative Solutions to Enhance Code Plagiarism Detection Tools,” *Acad. J. Comput. Inf. Sci.*, vol. 7, no. 1, pp. 65–71, 2024, doi: 10.25236/ajcis.2024.070110.