International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 13 Issue: 07 | Jul 2026 www.irjet.net p-ISSN: 2395-0072

AI-ASSISTED CODE CLONE DETECTION USING EMBEDDINGS AND SIMILARITY METRICS

Aruri Akhila[1], Dr. M. Chandra Mohan [2]

[1]M.Tech Scholar, Dept. of CSE, University College of Engineering, Science & Technology Hyderabad, JNTUH, Hyderabad, India

[2]Professor, Dept. of CSE, University College of Engineering, Science & Technology Hyderabad, JNTUH, Hyderabad, India

---

Abstract -*Code clones, which are duplicated or semantically similar code fragments, increase software maintenance effort and contribute to defect propagation in large software systems. Traditional clone detection techniques primarily rely on textual or syntactic comparison and often fail to identify semantically equivalent code that differs in structure, variable names, or control flow. This paper presents an AI assisted code clone detection framework that combines transformer based code embeddings with multiple similarity metrics to improve semantic clone identification. Source code is preprocessed to remove comments and formatting inconsistencies before generating high dimensional vector representations using the Sentence Transformer all MiniLM L6 v2 model, while CodeBERT embeddings provide programming language specific contextual information. Cosine similarity, together with lexical and structural similarity measures, is used to classify code pairs into Type 1 Exact, Type 2 Renamed, Type 3 Modified, Type 4 Semantic, or Non Clone categories. A Flask based web application enables users to compare code snippets or upload multiple source files and visualize the results through similarity gauges, comparison charts, clone network graphs, and embedding projections. Experimental evaluation on representative test cases demonstrates that the proposed framework accurately distinguishes exact, renamed, modified, and semantically equivalent code while assigning low similarity scores to unrelated programs. The proposed system provides an effective, extensible, and interpretable solution for software maintenance, plagiarism detection, and code quality assessment.**

Key Words: Code Clone Detection, Artificial Intelligence, Code Embeddings, Sentence Transformers, CodeBERT, Cosine Similarity, Semantic Similarity, Software Maintenance.

1. INTRODUCTION

Reusing, adapting, and repurposing existing source code is a routine part of software development, and as systems grow in size and complexity, code clones duplicated or highly similar code fragments tend to accumulate across files and modules. While code reuse accelerates development, excessive duplication increases maintenance effort, makes systems harder to reason about consistently, and can propagate defects: a bug fixed in one clone instance often persists, unnoticed, in its counterparts elsewhere in the codebase.

Traditional clone detection techniques are built primarily around textual, lexical, or syntactic comparison. These methods are effective at identifying exact or near-exact duplicates but are far less reliable at detecting semantically equivalent code that has been restructured, renamed, or reimplemented using different logic. As a result, a significant class of clone relationships those requiring an understanding of what code does rather than how it is written goes undetected by conventional tools. Manually identifying such clones does not scale: modern repositories are simply too large for exhaustive human review, and manual inspection is slow and error-prone.
Recent advances in natural language processing, particularly transformer-based embedding models, offer a way to represent source code numerically in a manner that captures semantic meaning rather than surface syntax.

1.1 Problem Statement

Existing clone detection systems predominantly rely on textual or structural comparison, which limits their ability to recognize code fragments that perform the same function through different implementations. This has three practical consequences: (i) semantically equivalent clones (Type-3 and Type-4) are frequently missed; (ii) developers lack an accessible, interactive tool that provides both a similarity score and an interpretable breakdown of why two fragments are considered similar; and (iii) most tools do not expose the underlying reasoning (lexical vs. structural vs. semantic contribution) needed to trust or audit a clone-detection decision. This work addresses these gaps by combining semantic code embeddings with multiple complementary similarity metrics inside an interactive, visualization-driven web application.

2. LITERATURE SURVEY

Recent work on AI-assisted programming increasingly treats source code as a learnable representation rather than plain text, with code embeddings and transformer architectures now widely applied to tasks such as code

completion, bug detection, and summarization. This establishes embeddings as a natural foundation for similarity-based clone detection, since they can capture both lexical and semantic information in compact vector form, although some embedding-based representations still struggle to capture deeper contextual or domain-specific intent, and a rigorous comparison of embedding-based methods across tasks remains limited in the literature.

3. PROPOSED METHODOLOGY AND SYSTEM ARCHITECTURE

The proposed system is an AI-Assisted Code Clone Detection System that identifies similar source code by jointly analyzing lexical, structural, and semantic similarity between code fragments. The system combines transformer-based embedding models CodeBERT and Sentence Transformer (all-MiniLM-L6-v2) with complementary similarity metrics to improve the accuracy and interpretability of clone detection compared with purely textual approaches.

Through a Flask-based web application, users may either paste two code snippets directly for comparison or upload one or more source files for batch analysis. Uploaded or entered code is preprocessed, converted into semantic embeddings, and evaluated using multiple similarity metrics; based on the resulting similarity scores, code pairs are classified as Type-1 (Exact Copy), Type-2 (Renamed), Type-3 (Modified), Type-4 (Semantic Clone), or Non-Clone.

3.1 Proposed System

1. Input Layer Accepts source code as two directly entered snippets, or one or more uploaded source files.

2. Preprocessing Layer Removes comments and docstrings; normalizes identifiers, spacing, and formatting; tokenizes the code; performs AST parsing to produce clean, structured code for analysis.

3. AI Analysis Engine Generates code embeddings using CodeBERT and Sentence Transformer models; computes lexical, structural, semantic, and cosine similarity, together with an aggregated confidence score.

4. Clone Classification Layer Classifies each code pair as Type-1 (Exact), Type-2 (Renamed), Type-3 (Modified), Type-4 (Semantic), or Non-Clone, based on the aggregated similarity score and a user-adjustable threshold.

5. Output & Visualization Layer Renders a similarity gauge, metrics-comparison chart, clone network graph, and PCA-based embedding visualization; displays the final classification to the user.

6. Logging & Storage Layer Records interaction details (input used, similarity metrics, clone type, timestamp) to a CSV log file for later review.

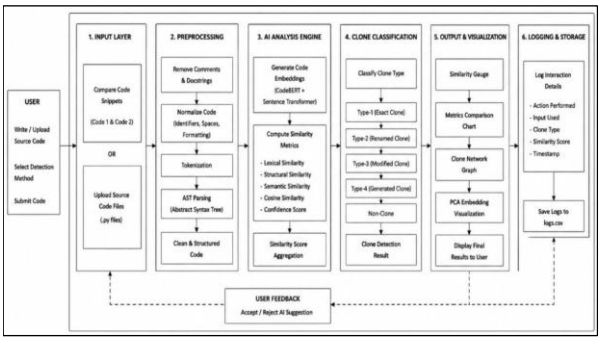

Fig -1: Architecture

3.2 System Workflow

The workflow begins when the user submits source code either as two snippets for direct comparison or as one or more uploaded files together with a chosen similarity threshold (adjustable from 0.1 to 1.0, default 0.80). The preprocessing layer strips comments, normalizes formatting, and tokenizes the code before the AI analysis engine generates embeddings and computes similarity metrics. The clone classification layer compares the aggregated similarity score against the selected threshold to assign a clone type, or to report a Non-Clone verdict when the score falls below it. Results including the similarity gauge, metrics-comparison chart, and, for uploads, a clone network graph and PCA embedding plot are then displayed to the user, and the interaction is logged for later reference.

4. PROPOSED ALGORITHM

The clone-detection algorithm proceeds through the following steps:

Step 1-Accept source code (two snippets, or one or more uploaded files) through the web interface.

Step 2-Preprocess each fragment: remove comments and docstrings, normalize identifiers and formatting, tokenize, and parse into an Abstract Syntax Tree (AST) representation.

Step 3-Generate a semantic embedding for each fragment using the Sentence Transformer (all-MiniLM-L6-v2) model, supplemented with CodeBERT embeddings to capture programming-language-specific context.

Step 4-Compute similarity metrics: lexical similarity (token/text-level overlap), structural similarity (AST/structure-level similarity), semantic similarity

(cosine similarity between embedding vectors), and an aggregated confidence score.

Step 5-Compare the aggregated similarity score against the user-selected threshold and classify the pair as Type-1, Type-2, Type-3, Type-4, or Non-Clone.

Step 6-Generate visual outputs: similarity gauge, metrics-comparison chart, and for multi-file uploads a clone network graph and a PCA-based 2D projection of the embeddings.

Step 7-Log the interaction (input used, computed metrics, clone type, timestamp) to a CSV file.

Step 8-Display the final classification, confidence score, and visualizations to the user.

5. EXPERIMENTAL SETUP

Existing clone detection systems predominantly rely on textual or structural comparison, which limits their ability to recognize code fragments that perform the same function through different implementations. This has three practical consequences: (i) semantically equivalent clones (Type-3 and Type-4) are frequently missed; (ii) developers lack an accessible, interactive tool that provides both a similarity score and an interpretable breakdown of why two fragments are considered similar; and (iii) most tools do not expose the underlying reasoning (lexical vs. structural vs. semantic contribution) needed to trust or audit a clone-detection decision. This work addresses these gaps by combining semantic code embeddings with multiple complementary similarity metrics inside an interactive, visualization-driven web application.

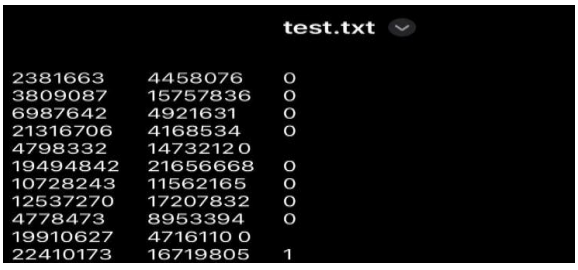

Fig -2: test.txt

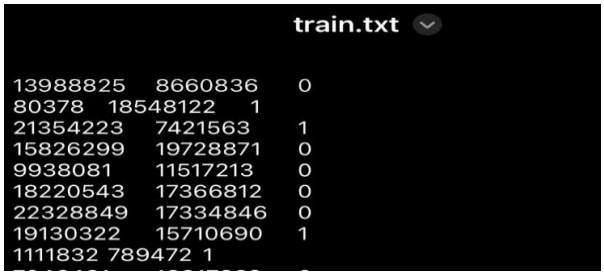

Fig -3: train.txt

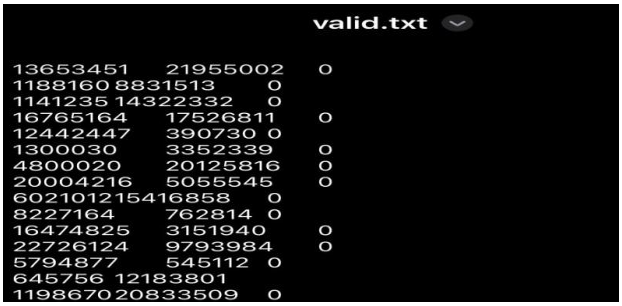

Fig -4: valid.txt

6. RESULTS AND DISCUSSION

Existing clone detection systems predominantly rely on textual or structural comparison, which limits their ability to recognize code fragments that perform the same function through different implementations. This has three practical consequences: (i) semantically equivalent clones (Type-3 and Type-4) are frequently missed; (ii) developers lack an accessible, interactive tool that provides both a similarity score and an interpretable breakdown of why two fragments are considered similar; and (iii) most tools do not expose the underlying reasoning (lexical vs. structural vs. semantic contribution) needed to trust or audit a clone-detection decision. This work addresses these gaps by combining semantic code embeddings with multiple complementary similarity metrics inside an interactive, visualization-driven web application.

Fig -5: Type 1- exact copy

Fig -6: Type 2 - Renamed

Fig -7: Type 3 - Modified

Fig -8: Type 4 - Semantic

7. COMPARITIVE ANALYSIS

A large-scale quantitative benchmark against external clone-detection systems was outside the scope of this work; the comparison below is therefore presented at the feature level, based on the capabilities reported for each category of approach in the literature reviewed in Section 2, and should be read as illustrative rather than as a head-to-head empirical evaluation.

Table -1: Comparitive Analysis of clones

Approach	Typical Strengths	Typical Limitations
Textual / lexical clone detectors	Fast; effective for Type-1 and Type-2 clones	Little or no support for semantically equivalent (Type-3/4) clones; no similarity breakdown
Graph- / structure-aware embedding detectors.	Strong performance on Type-3/Type-4 semantic clones	Higher training & computational complexity; not typically presented as an interactive end-user tool
Proposed system	Combines lexical, structural, and semantic (embedding-based) similarity in one	Evaluated on a small set of illustrative cases rather than a large labelled benchmark (see Section 8)

8. CONCLUSIONS

This paper presented an AI-Assisted Code Clone Detection System that identifies similar source code by combining artificial intelligence with multi-metric similarity analysis. The system preprocesses source code, generates semantic embeddings using CodeBERT and Sentence Transformer models, computes lexical, structural, and semantic similarity scores, and classifies code pairs into Type-1, Type-2, Type-3, Type-4,or Non-Clone categories. A Flask-based web application provides an interactive platform for comparing snippets, uploading files, and visualizing results through similarity gauges, metrics-comparison charts, clone network graphs, and PCA-based embedding plots. Test cases spanning all four clone types and a Non-Clone case demonstrate that the system correctly distinguishes exact copies, renamed variants, structurally modified code, and semantically equivalent implementations, while assigning low similarity to functionally unrelated code. The proposed approach offers a simple, interpretable, and extensible solution for embedding-based code clone detection, with clear directions identified for further quantitative validation and scaling.

REFERENCES

1) Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Proc. 2020 Conf. Empirical Methods in Natural Language Processing (EMNLP), Nov. 2020, pp. 1536–1547.M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

2) S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Tufano, M. Gong, M. Zhou, and N. Duan, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," arXiv:2102.04664, Feb. 2021..

3) C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," Queen's School of Computing, Tech. Rep. TR 541, pp. 1–115, Sept. 2007

4) J. Svajlenko and C. K. Roy, "BigCloneBench: A Big Data Benchmark of Code Clones," in Proc. 2015 IEEE Int. Conf. Software Maintenance and Evolution (ICSME), Sept. 2015, pp. 131–140.