

Original Article

A CNN-LSTM-based Hybrid Deep Learning Model Enhanced with PSO-GA Optimization for Accurate Code Clone Detection

Tanvi¹, Geetika²

¹Department of Computer Applications, Tilak Raj Chadha Institute of Management and Technology, Haryana, India.

²Department of Artificial Intelligence and Emerging Technologies, Lovely Professional University, Punjab, India.

²Corresponding Author : geetikachatley@gmail.com

Received: 21 August 2026

Revised: 15 September 2026

Accepted: 24 September 2026

Published: 28 October 2026

Abstract - Code clone analysis is needed when software repositories contain multiple implementations of the same or closely related functionality. Such duplication can make later maintenance harder because a change made in one fragment may not be reflected in another. This paper presents a hybrid CNN-LSTM framework in which PSO and GA are used to refine the feature representation before classification. The workflow consists of source-code preparation, feature construction, optimization, and clone classification. CNN is used to learn local patterns from the code representation, whereas LSTM models dependencies that extend across the sequence. The optimized representation is evaluated on BigCloneBench, Google Code Jam, and a Java repository. The reported experiments show that the combined model provides higher recall, accuracy, and precision than the comparison models. The findings indicate that optimization-assisted deep learning can provide a practical route toward more reliable clone analysis in software repositories.

Keywords - Clone detection accuracy, CNN-LSTM, Feature set optimization, Learning-based software engineering, PSO-GA.

1. Introduction

Software duplication is common in large repositories. Developers may intentionally reuse an existing implementation, or similar code may emerge independently as a project evolves. Although reuse can shorten development time, duplicated fragments also increase the number of locations that must be considered during debugging and maintenance. A correction applied to one occurrence may not automatically reach its related copies. For this reason, clone analysis is relevant to software quality, technical debt, and vulnerability management [4].

The difficulty of the detection task depends on how much the two fragments have changed. Textual and token-level comparison works well when the surface form is nearly identical [5, 10], but it becomes less dependable after identifiers, statements, or control structures have been modified. Structural techniques based on representations such as ASTs and PDGs provide more context [19, 23], although generating and processing those representations can be expensive for large collections. Conventional classifiers, including SVMs and Decision Trees, can learn from engineered representations, but their results remain tied to the quality of those features. Deep learning provides an alternative

because useful representations can be learned from the input rather than being specified entirely by hand. A CNN can emphasize local patterns, while an LSTM can retain information across a sequence. Combining them is therefore attractive for source-code analysis, where both local constructs and their ordering may carry useful information. Nevertheless, model tuning, class imbalance, and generalization across repositories remain practical concerns.

This work investigates a CNN-LSTM detector assisted by PSO-GA optimization. The optimization stage is used to refine the feature and parameter configuration before the final classification step. The framework is tested on BigCloneBench, Google Code Jam, and a Java repository, and the reported results are compared with alternative models. The objective is to examine whether the combined architecture can provide a more consistent detection performance across different clone categories. Clone-detection methods can broadly be viewed as lexical, structural, machine-learning, or deep-learning approaches. Lexical techniques compare tokens or other surface-level representations and are useful for straightforward duplication. Their limitations become apparent when two fragments implement similar behaviour after substantial syntactic changes. Such cases are commonly



associated with Type-3 and Type-4 clones. Structural methods attempt to retain information about the organization of a program. AST representations describe syntactic structure, while PDGs capture relationships associated with dependencies [19, 23]. These methods can recognize relationships that are missed by direct text comparison, but the additional representation-building step increases processing requirements. Hashing and locality-sensitive hashing can reduce the candidate search space, although faster matching does not necessarily provide the same semantic sensitivity. Machine-learning classifiers introduced a data-driven alternative [17]. Their performance, however, is influenced by the features supplied during training. Deep neural models address part of this limitation by learning higher-level representations from source-code inputs [16, 25].

CNNs are suited to recurring local patterns, whereas LSTMs can preserve information from earlier elements of a sequence. Their combination offers a way to use both forms of information. At the same time, deep models can be affected by overfitting, class imbalance, training instability, and limited interpretability.

Optimization-assisted learning has consequently become an active direction. PSO and GA can search for useful feature or parameter configurations rather than relying only on manual selection [34, 44]. The motivation for the present work is to combine this search capability with CNN–LSTM representation learning and then examine the resulting detector on heterogeneous benchmark data.

The combination of an optimizer and a learned representation is useful because the two components solve different problems. CNN and LSTM learn patterns from the source-code representation, while PSO and GA search for a configuration that gives favourable validation behaviour. PSO updates candidate solutions according to swarm movement, whereas GA introduces new candidates through evolutionary operations. Using both mechanisms broadens the search over the available feature and parameter space.

In the proposed framework, optimization is performed before the final clone classification. The intention is to remove or down-weight less useful information and to obtain a configuration that is less sensitive to manual parameter choices. The resulting model is evaluated on three datasets and is compared with individual CNN and LSTM variants as well as other reported methods. The experimental framework uses CNN–LSTM for the learning stage and PSO–GA for feature and parameter refinement. BigCloneBench, Google Code Jam, and the selected Java repository provide the evaluation data. The comparison considers accuracy, precision, recall, F1-score, and AUC, together with the results reported for alternative approaches. This design allows the effect of the hybrid learning architecture and the optimization stage to be

examined separately. The development of clone detection has gradually moved from direct textual comparison toward representations that preserve more information about program structure and behaviour.

Token-level matching remains useful for obvious copies, but it can miss related implementations after syntactic modification. AST- and PDG-oriented methods retain structural information and therefore offer greater descriptive power, although they also introduce additional processing overhead.

Learning-based approaches changed the problem from fixed similarity rules to data-driven classification. Early classifiers still depended on manually selected features. Deep networks reduce that dependence by learning representations during training. CNNs are particularly useful for local patterns, while LSTMs can retain sequence-level context. A combined CNN–LSTM architecture can consequently examine a code fragment from both perspectives.

AST and PDG representations provide information that is unavailable in simple token matching because they describe the organization and relationships within a program. The additional information comes with a cost: parsing and constructing these representations can become demanding as repository size increases. Moreover, small implementation changes may alter parts of the representation and affect the similarity calculation.

Candidate-reduction techniques such as hashing and locality-sensitive hashing address the computational side of the problem by avoiding unnecessary pairwise comparisons. Their main trade-off is between search efficiency and the amount of semantic information retained. This is one reason why structural and learned representations continue to be explored for more difficult clone categories.

Traditional classifiers such as SVMs and Decision Trees can be effective when the input features describe the relevant properties of the code. Their limitation is that missing or poorly designed features cannot be recovered by the classifier itself. The move toward deep neural networks allows the representation to be learned jointly with the classification task [16, 25]. Within this family, CNNs can detect repeated local configurations, while recurrent networks can preserve information across a sequence. Combining these capabilities is useful when code fragments have related functionality but differ in their local syntax. The approach still requires careful training because data imbalance, overfitting, and parameter selection can influence the final result. Another research direction couples deep models with metaheuristic search. Instead of fixing every feature or training parameter manually, PSO and GA can explore several candidate configurations. Their use is motivated by the need to find a useful compromise

between model quality and computational effort. The present study applies this idea to a CNN–LSTM detector and evaluates whether the optimized configuration generalizes across different code collections.

2. Literature Review

A clone detector attempts to identify two fragments that are identical, closely related, or functionally equivalent. The task matters because duplicated implementations can increase maintenance effort and can also allow the same defect to appear in several locations. Direct textual comparison is effective for obvious copies but becomes weaker when equivalent functionality is written differently [1, 2]. Machine-learning approaches provide a way to classify richer representations than simple text matches. SVMs and Decision Trees were among the earlier choices [4], but they depend on engineered features. Deep learning instead allows the model to learn representations from source-code input. CNNs can focus on local structures, and LSTMs can preserve information across a sequence. Their combination is therefore a natural candidate for clone classification, provided that the training configuration is properly selected. Recent deep-learning studies have explored learned source-code representations beyond direct token matching [6, 7]. CNN-based components can emphasize local patterns, while sequence models can retain ordering information; the combined CNN–LSTM architecture examined in related work uses both forms of evidence [8]. Its effectiveness is nevertheless sensitive to training choices and to the balance between clone and non-clone examples. Other studies have investigated graph representations, transformer-based models, Siamese networks, and language-independent settings. These approaches reflect a broader shift toward models that capture

relationships in source code rather than relying only on surface similarity. The present work follows this direction while adding PSO–GA optimization to the CNN–LSTM configuration. The recent literature also illustrates several ways of enriching clone representations. Some studies combine source-code structure with program graphs [9], while transformer-based architectures can provide broader contextual representations [33]. Cross-language embedding approaches have been examined for clone comparison [11], and code-alignment neural networks have also been proposed for clone detection [12].

The common challenge is not only representation quality but also reliable model configuration. A complex representation may still perform unevenly when it is transferred to another dataset. The optimization stage in this study is intended to reduce that dependence by searching for a suitable feature and parameter configuration before final classification.

The literature indicates that different clone categories favour different forms of evidence. Lexical approaches are efficient when the fragments remain close to one another, structural representations expose program relationships, and neural models can learn patterns that are difficult to encode manually. A useful framework, therefore, needs to balance representational richness with manageable computation and training. The proposed study combines CNN and LSTM for complementary local and sequential learning and uses PSO–GA to refine the associated feature and parameter space. Evaluation on several datasets is used to determine whether the resulting configuration maintains its advantage beyond a single code collection.

Table 1. Comparative analysis of recent clone detection methods

Author(s)	Year	Dataset Used	Techniques Used	Tools / Frameworks	Research Gap Identified
Wang et al. [9]	2020	BigCloneBench and related Java benchmarks	Graph neural network + flow-augmented AST	Deep learning / GNN	Captures graph semantics; no metaheuristic hyperparameter search
Hua et al. [22]	2021	BigCloneBench	Hybrid code representation + attention	Deep neural network	Strong representation learning; no PSO-GA feature/parameter optimization
Zhang et al. [12]	2021	Public clone benchmarks	Code alignment network	Deep neural network	Alignment-focused learning; optimization stage not explored
El Arnaoty and Servant [11]	2024	Cross-language clone datasets	Common embedding space	Deep representation learning	Addresses cross-language detection; not focused on PSO-GA tuning
Arshad et al. [51]	2022	BigCloneBench, SemanticCloneBench, Android	CodeBERT fine-tuning	Transformer	Strong semantic modeling; generalization varies on unseen functionality

Geetika et al. [8]	2025	BigCloneBench, Google Code Jam, Java dataset	CNN-LSTM + PSO-GA	Python / deep learning	Optimization-assisted hybrid model motivates further calibration and validation.
--------------------	------	--	-------------------	------------------------	--

3. Materials and Methods

The complete workflow is divided into four connected stages. First, the source fragments are normalized and prepared for analysis. Next, relevant features are constructed from the processed code.

PSO and GA are then used to refine the representation. Finally, the optimized features are supplied to the CNN-LSTM classifier, which produces the clone decision. Keeping these stages explicit makes the contribution of the optimization step easier to examine.

3.1. Dataset Description

The evaluation uses three sources of code: BigCloneBench (BCB) [45], the Google Code Jam dataset [21], and a Java dataset derived from Eclipse JDT-Core [20]. BCB contains annotated Java methods representing Type-1 through Type-4 clone relationships. GCJ provides programming-contest solutions with substantial variation in implementation style and logic. The Eclipse JDT-Core collection represents source files from a real software-development project. Training, validation, and testing subsets were formed from these collections, and the reported evaluation uses recall, accuracy, and precision.

Table 2. Benchmark datasets employed for training and evaluation of the proposed hybrid deep learning framework for code clone detection

Dataset	Description	Language	Source / Repository	Application Domain	Size / Scope
BigCloneBench (BCB)	Benchmark dataset with annotated Java methods across Type-1 to Type-4 clones	Java	BigCloneBench / BigCloneEval [42, 45]	Large-scale clone detection	> 25k methods, labeled
Google Code Jam (GCJ)	Programming-contest submissions with stylistic and logical variability	C, C++, Java, Python	Google Code Jam dataset repository [21]	Algorithmic problem solving	Thousands of code solutions
Java Dataset (Eclipse JDT-Core)	Source files extracted from the Eclipse IDE project	Java	Eclipse JDT-Core repository [20]	Software development projects	900+ Java classes

3.2. Initial Preparation

Before feature construction, the source fragments are converted into a more uniform form. Comments, redundant spacing, and unnecessary tokens are removed, while information needed for later semantic analysis is retained. The remaining code is tokenized and parsed so that subsequent processing operates on a structured representation. This prepared representation is then passed to the feature-generation stage.

3.3. Creation of Feature Sets

Feature construction extracts information that may help distinguish related and unrelated fragments. Operators, constants, identifiers, and control-flow elements are considered during this step. The extracted elements are weighted according to their role in the representation used for similarity analysis. Parser-based extraction is supplemented by manual checking to ensure that relevant programming constructs remain represented after preprocessing.

3.4. Optimization of Feature Sets

The feature-refinement stage combines GA with PSO. GA creates variation through selection, crossover, and mutation, while PSO moves candidate solutions through the

search space using swarm-based updates. The two mechanisms provide complementary exploration of possible feature configurations. The selected representation is subsequently passed to the deep-learning model, with the aim of limiting redundant information and reducing the influence of noisy features. The similarity index represents the amount of overlap between the feature sets generated for two code fragments. It is used as the basis for deciding whether the fragments should be treated as a clone pair. For two fragments C_1 and C_2 , the similarity measure is defined as:

$$S(C_1, C_2) = |F(C_1) \cap F(C_2)| / |F(C_1) \cup F(C_2)| \quad (1)$$

A decision threshold is applied to the resulting similarity value. Pairs whose score reaches the selected threshold are assigned to the clone class; pairs below that threshold are treated as non-clones.

$$\text{Clone}(C_1, C_2) = 1 \text{ if } S(C_1, C_2) \geq \theta; \\ 0 \text{ if } S(C_1, C_2) < \theta \quad (2)$$

At the CNN stage, local patterns in the feature representation are learned from the code fragments. Patterns involving constructs such as operators and loops can therefore

contribute to the representation used for comparison. The convolution operation is given by:

$$h_{ij} = f(\sum_m \sum_n (w_{mn} \cdot x_{i+m,j+n}) + b) \quad (3)$$

The LSTM component processes the sequence information retained in the representation. Its hidden state can preserve information about statement order and control flow, which may be useful when two fragments have similar functionality but differ in their local token patterns. The corresponding LSTM state equations are given below.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (4)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (5)$$

$$\tilde{C}_t = \tanh(WC \cdot [h_{t-1}, x_t] + bC) \quad (6)$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \quad (7)$$

$$h_t = o_t \cdot \tanh(C_t) \quad (8)$$

After the CNN and LSTM processing stages, the final layer estimates the probability that a pair of fragments belongs to the clone class. This probability is used to make the final clone or non-clone decision:

$$P(\text{clone} | C_1, C_2) = \sigma(W \cdot h + b) \quad (9)$$

Table 3. Preprocessing operations are applied to the source code prior to feature extraction and optimization

Preprocessing Task	Operational Details	Intended Outcome
Comment elimination	Removal of inline and block comments from code fragments	Reduces noise and improves clarity of feature sets
Whitespace normalization	Removal of redundant spaces, tabs, and blank lines	Ensures uniform code structure for analysis
Lexical tokenization	Splitting source code into identifiers, constants, and operators	Creates a structured representation of source code
Syntax parsing	Conversion of preprocessed code into an abstract feature representation	Enables systematic feature extraction
Code standardization	Normalizing code fragments into a consistent intermediate form	Enhances generalization across datasets

Table 4. Hybrid feature-set optimization strategy integrating PSO and GA

Technique	Role in Optimization	Advantages
Particle Swarm Optimization (PSO)	Swarm-based exploration of the search space	Fast convergence, global search ability
Genetic Algorithm (GA)	Selection, crossover, mutation for variation	Avoids local optima, introduces diversity
Hybridization (PSO + GA)	Combines velocity update with evolutionary steps	Balanced exploration–exploitation trade-off

Table 5. Deep learning architecture of the proposed CNN–LSTM model

Architectural Module	Operational Mechanism	Role in Clone Detection
Convolutional Layer (CNN)	Extracts localized spatial features from input feature vectors	Captures structural patterns in source code fragments
Pooling Layer	Down-samples feature maps while preserving dominant characteristics	Reduces dimensionality and mitigates overfitting
Fully Connected Layer	Integrates learned features into a unified representation	Enables higher-level abstraction of code semantics
Long Short-Term Memory (LSTM)	Models sequential dependencies using gated memory mechanisms	Detects temporal and semantic relationships in code
Output Layer	Computes similarity indices and final classification	Determines clone vs. non-clone instances

3.5. CNN-LSTM Clone Detection

The classifier receives the representation produced after the PSO–GA stage. CNN processing emphasizes local patterns in the code features, while LSTM processing preserves information that spans the sequence. The two outputs are combined before the final decision layer. This arrangement allows the detector to use both local structure and sequence-level evidence when distinguishing Type-1, Type-2, Type-3, and Type-4 relationships.

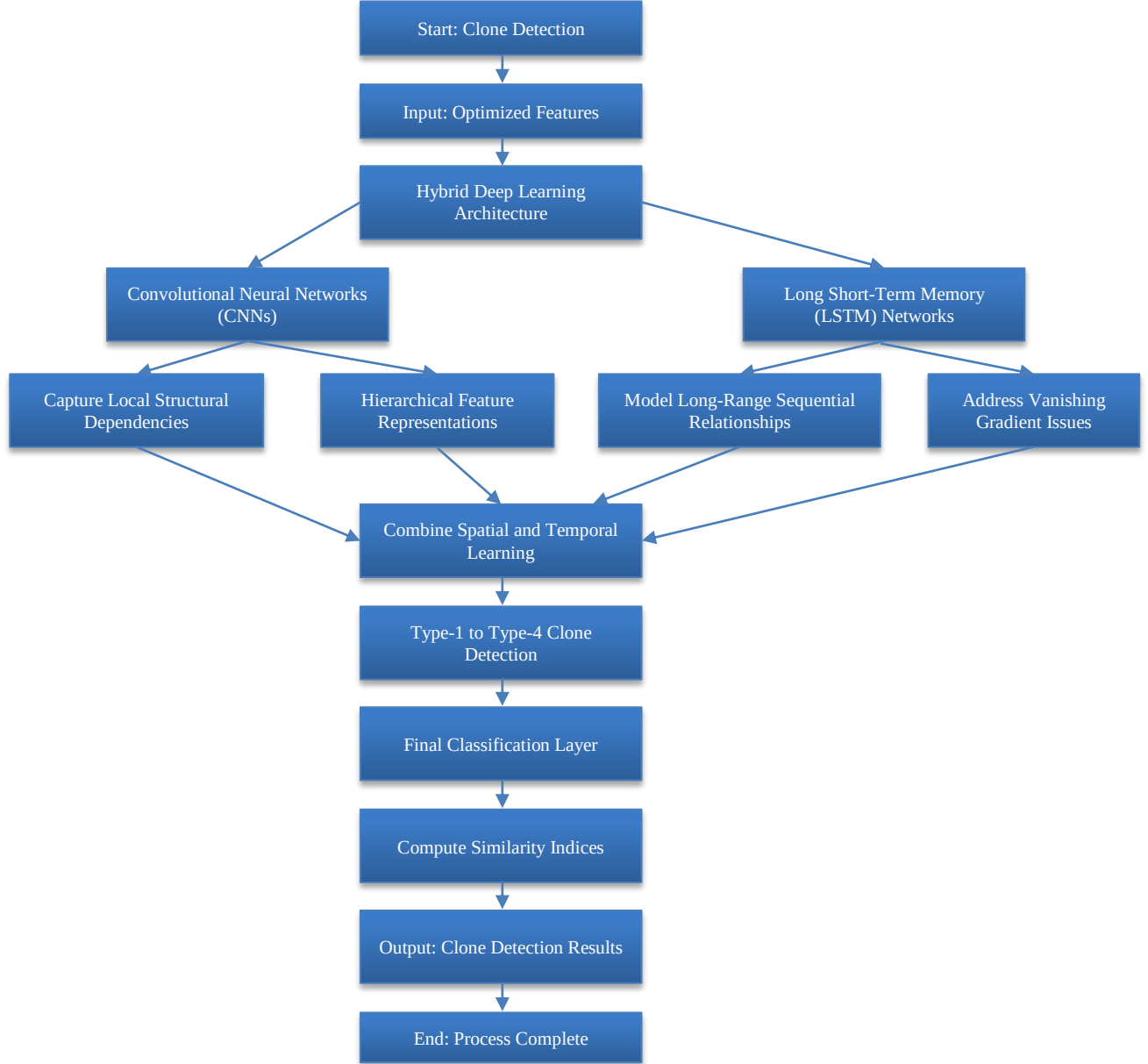


Fig. 1 CNN–LSTM hybrid framework for code clone detection using optimized feature inputs

3.6. Measures of Evaluation

The experimental evaluation uses accuracy, precision, and recall as the principal classification measures. Together, these quantities show how often the model is correct overall, how dependable its positive predictions are, and how many of the actual clone cases it can recover.

Accuracy is calculated as the proportion of all evaluated pairs that receive the correct class label.

$$\text{Accuracy} = (\text{TP} + \text{TN}) / (\text{TP} + \text{FP} + \text{TN} + \text{FN}) \quad (10)$$

Precision measures the fraction of predicted clone pairs that are genuine clone cases.

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP}) \quad (11)$$

Recall measures how much of the actual clone population is recovered by the detector.

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN}) \quad (12)$$

The three measures are calculated for the datasets used in the study and are considered alongside the comparison results.

This provides a consistent basis for judging detection quality and generalization.

4. Models Calibration and Validation

4.1. Calibration of Model Parameters

Several parameter settings were examined during model calibration. Batch sizes of 32, 64, and 128 were tested, and the learning rate was searched between 0.001 and 0.01. CNN filter counts from 64 to 256, and kernel sizes from 3×3 to 5×5 were considered. LSTM hidden units were varied from 64 to 128. For PSO, the inertia weight was varied from 0.4 to 0.9, and C1 and C2 from 1.0 to 2.0; GA crossover probability was explored between 0.6 and 0.9. The configuration reported in the paper was selected using cross-validation performance.

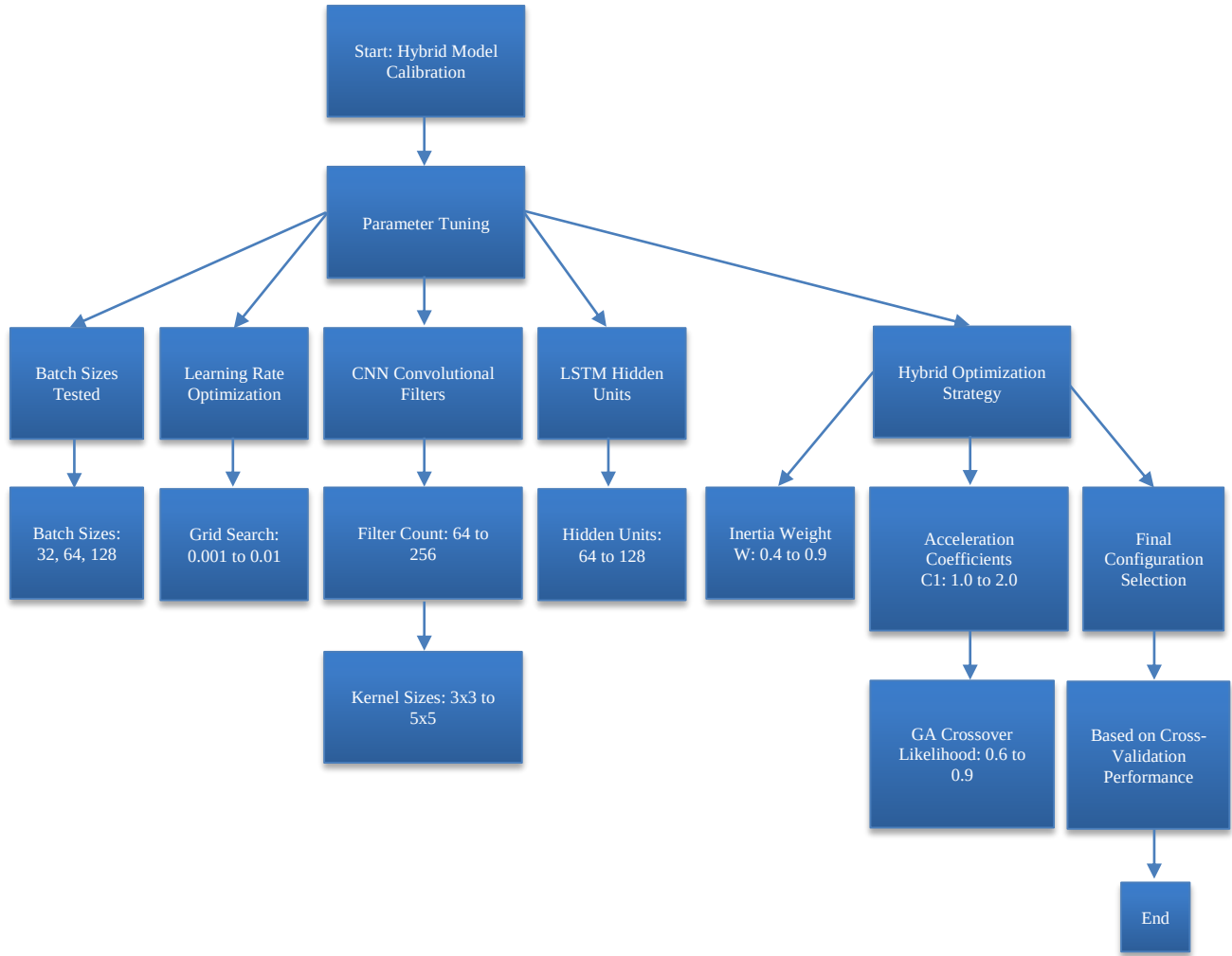


Fig. 2 Hybrid model calibration and parameter-tuning workflow

4.2. Validation Strategy

Stratified 10-fold cross-validation was used to keep the class proportions reasonably consistent across folds. The data were divided into training, validation, and testing portions of 70%, 15%, and 15%. Code originating from the same project was retained within the same split to limit information leakage between related fragments. Accuracy, precision, recall, and F1-score were used for model assessment. The evaluation

includes individual CNN and LSTM models and a traditional Vector Space Model (VSM), together with state-of-the-art results reported in the literature. These comparisons provide a reference for judging the additional contribution of the hybrid architecture and the PSO–GA refinement stage.

5. Results and Discussion

The calibration and validation results are summarized through the sequence of experiments shown in Figure 3. The

following analysis describes the parameter settings and the performance obtained from the trained models. Calibration was performed before the final model comparison. PSO-GA was used to search the configuration space associated with the learning rate, batch size, dropout, CNN filters, and LSTM hidden units.

The best learning rate was reached after 37 iterations at 0.0018. A dropout probability of 0.35 was retained, the CNN used 64 filters, and the LSTM used 128 hidden units. These settings were used for the subsequent experiments.

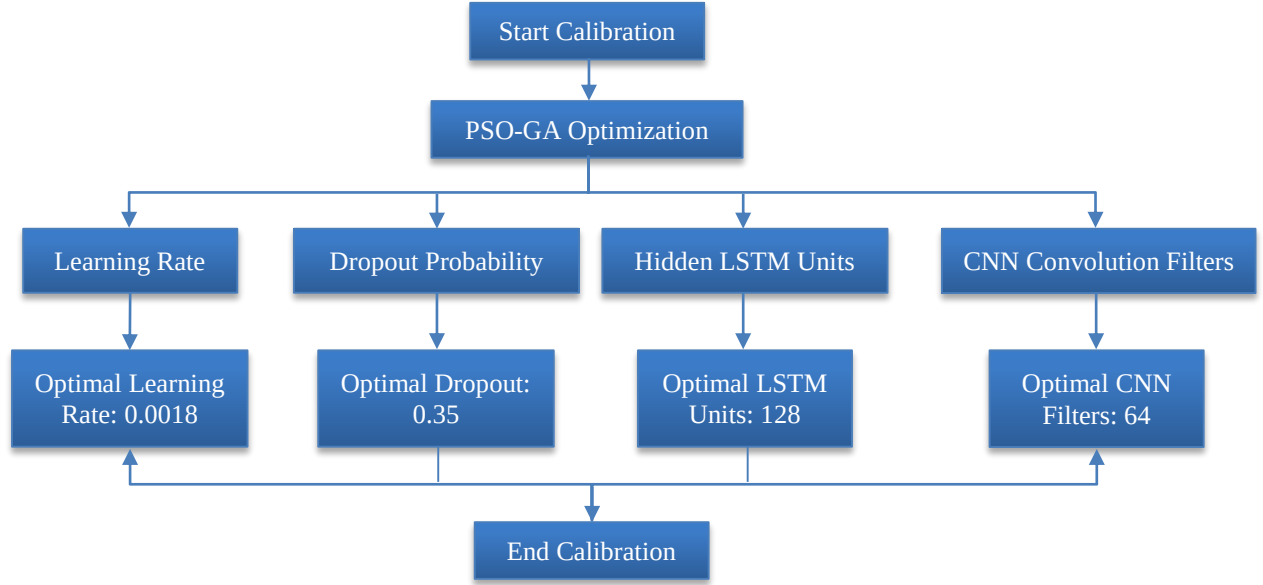


Fig. 3 Hybrid CNN-LSTM model calibration process

The CNN-LSTM and PSO-GA components were calibrated together so that the main learning parameters could be adjusted rather than fixed arbitrarily. The calibration considered the learning rate (η), dropout probability (p), number of LSTM hidden units (h), and number of CNN filters (f). The equations below describe the individual components used in this stage.

- CNN Feature Extraction

The convolutional layer learns local patterns from the sequence of code tokens:

$$z_{ij} = \sigma(\sum_m \sum_n w_{mn} \cdot x_{i+m, j+n} + b) \quad (13)$$

- LSTM Sequential Learning

The LSTM layer processes the source-code sequence and retains information from earlier tokens:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (14)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \quad (15)$$

$$C_t = f_t \square C_{t-1} + i_t \square \tilde{C}_t \quad (16)$$

$$h_t = o_t \odot \tanh(C_t), o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (17)$$

- Hybrid Representation

CNN and LSTM features are concatenated to form the snippet embedding:

$$u = h_snippet = [h_cnn; h_lstm] \in \mathbb{R}^{d_u} \quad (18)$$

- PSO-GA Optimization Objective

Let x represent a candidate set of hyperparameters, such as learning rate η , dropout p , and the number of hidden units h . The fitness function combines validation performance with training time and model size:

$$F(x) = \alpha \cdot F1val(x) + \beta \cdot AUCval(x) - \gamma \cdot Timetrain(x) - \delta \cdot Params(x) \quad (19)$$

The PSO and GA procedures search for the candidate configuration that gives the highest value of this fitness function:

$$x^* = \operatorname{argmax}_x F(x) \quad (20)$$

The PSO update of particle velocity and position is expressed as:

$$v_i(t+1) = \omega v_i(t) + c1 r1 (pbest_i - x_i(t)) + c2 r2 (gbest - x_i(t)) \quad (21)$$

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (22)$$

the CNN and LSTM baselines reached 92.1% and 90.7%, respectively.

5.1. Training Performance of the Proposed Model

Training was observed over 50 epochs. By epoch 40, the optimized CNN–LSTM model reached 96.8% training accuracy and increased to 97.1% by epoch 50. At epoch 40,

The reported loss behaviour also shows a steadier reduction for the optimized hybrid model, indicating faster convergence during the training run.

Table 6. Training accuracy across epochs

Epochs	CNN (%)	LSTM (%)	CNN–LSTM (%)	CNN–LSTM + PSO–GA (%)
10	78.5	74.3	82.1	86.4
20	85.2	81.6	88.7	92.5
30	88.9	85.4	91.8	95.1
40	92.1	90.7	94.6	96.8
50	92.3	90.9	94.8	97.1

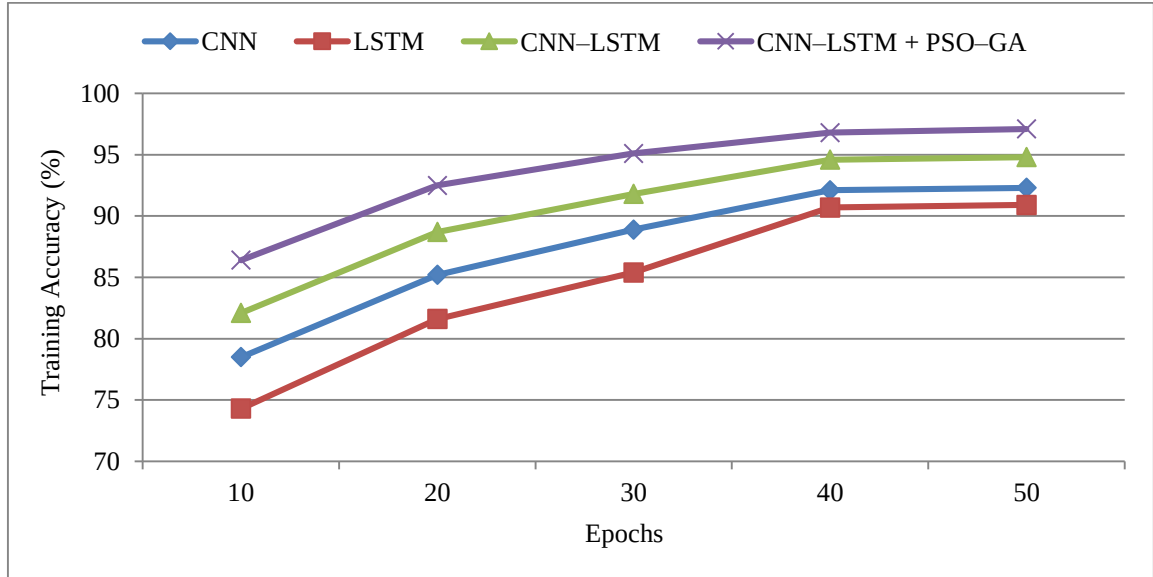


Fig. 4 Model convergence trends across epochs

5.2. Validation Metrics of the Proposed Model

A separate hold-out portion corresponding to 20% of the data was used for validation. Precision, recall, F1-score, and

AUC were considered to provide a broader view of classifier behaviour than accuracy alone.

Table 7. Validation performance comparison

Model	Precision	Recall	F1-score	AUC
CNN	0.89	0.87	0.88	0.91
LSTM	0.86	0.85	0.85	0.89
CNN–LSTM	0.91	0.90	0.90	0.94
CNN–LSTM + PSO–GA	0.95	0.94	0.94	0.97

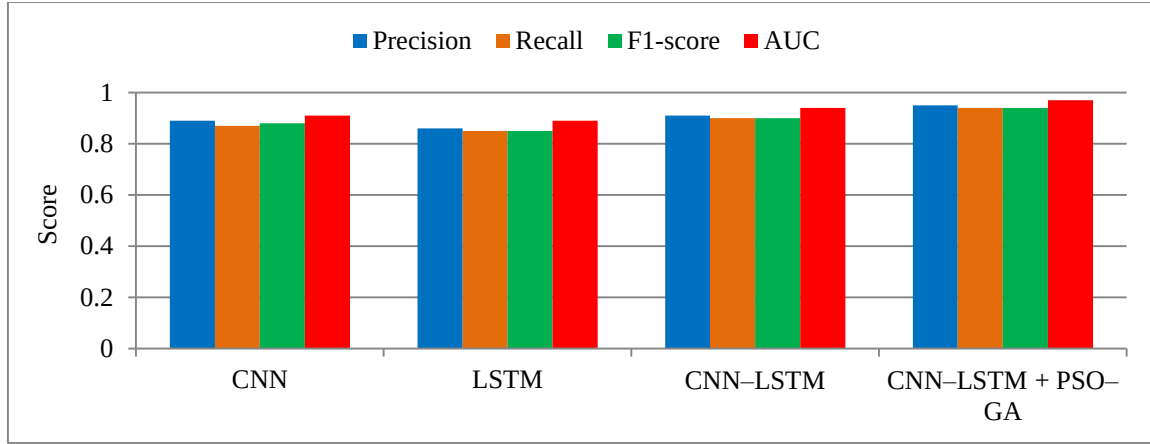


Fig. 5 Comparative validation performance of CNN, LSTM, and CNN-LSTM variants

The optimized configuration recorded an F1-score of 0.94 and an AUC of 0.97. The corresponding F1-scores for CNN and LSTM were 0.88 and 0.85, while their AUC values were 0.91 and 0.89. The comparison suggests that the optimized hybrid model produced a more balanced separation of clone and non-clone cases. Testing was additionally performed on projects that were not included in training. The purpose was to examine whether the observed advantage persisted when the code source changed.

- The CNN-LSTM with PSO-GA reached 95.7% testing accuracy, whereas CNN and LSTM reached 91.4% and 90.1%, respectively.
- The largest relative improvements occurred for Type-2 and Type-3 cases, where both structural patterns and sequence information contribute to the decision.
- The optimized configuration also produced fewer false-positive decisions than the corresponding non-optimized CNN-LSTM model.

Table 8. Testing results across clone types

Clone Type	CNN Accuracy (%)	LSTM Accuracy (%)	CNN-LSTM (%)	CNN-LSTM + PSO-GA (%)
Type-1	96.5	95.9	97.2	98.1
Type-2	89.8	88.7	92.3	95.0
Type-3	85.6	84.9	89.4	93.5
Type-4	79.2	77.8	82.5	87.1

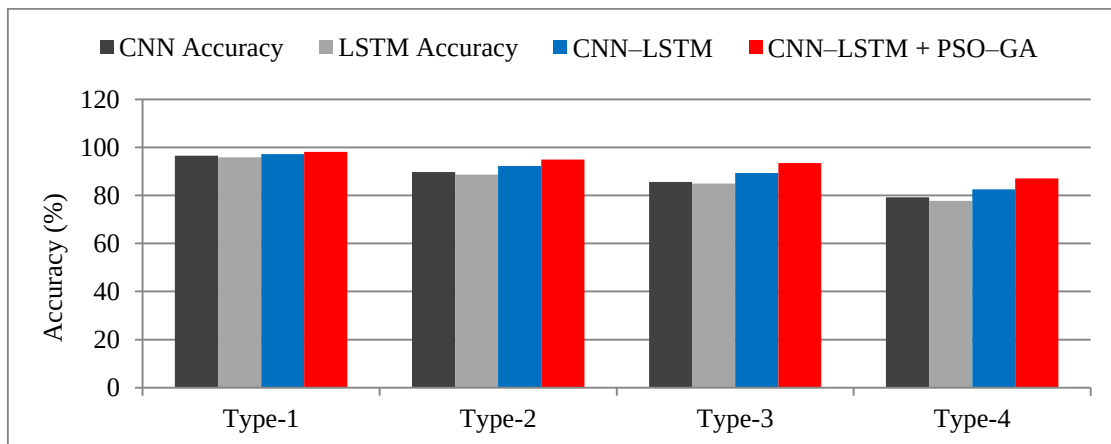


Fig. 6 Clone detection effectiveness across clone types

The experimental results favour the optimized CNN-LSTM configuration over the individual CNN and LSTM baselines. The two neural components contribute different information: CNN emphasizes local code patterns, while

LSTM carries information through the sequence. PSO-GA adds a search-based refinement step that, in the reported experiments, improved convergence and reduced false positives. The advantage is visible across the four clone

categories, although Type-4 remains the most difficult category, which is consistent with the greater semantic gap between such fragments.

6. Conclusion

This work presented a CNN–LSTM approach for code-clone detection with PSO–GA used to refine the feature and parameter configuration. CNN was used to capture local patterns, LSTM to model longer-range sequence information, and the optimization stage to search for a more suitable representation before classification. On the reported evaluation measures, the combined model performed better than the individual CNN and LSTM baselines. The testing results also show an advantage across Type-1 to Type-4 clone categories. The findings support the use of complementary local and sequential representations when clone detection extends beyond straightforward textual duplication.

The findings provide evidence that optimization-assisted deep learning can be useful for source-code analysis tasks in which several forms of structural information must be considered together. In this framework, the learning and optimization components play distinct roles: CNN and LSTM learn the representation, while PSO–GA searches for a configuration that supports the final classifier. A natural next step is evaluation on substantially larger industrial repositories, where the number of files and candidate pairs is

much greater than in the benchmark collections. Cross-language detection is another direction, particularly if language-independent representations or transfer learning can reduce the dependence on a single programming language. The method could also be incorporated into IDEs and CI/CD workflows. Such integration would make clone analysis part of routine development and could provide earlier indications of duplication that may warrant refactoring. Interpretability is another area for extension. A useful system for developers should ideally indicate which portions of a code fragment contributed to a clone decision. Security-oriented detection is also worth investigating because duplicated vulnerable code can spread the same weakness through a repository.

Finally, alternative optimization methods, including Ant Colony Optimization and the Grey Wolf Optimizer, could be evaluated against PSO–GA to determine whether further improvements are possible.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Funding Statement

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

- [1] B.S. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems,” *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE)*, Toronto, ON, Canada, pp. 86-95, 1995. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [2] Chanchal Kumar Roy, and James R. Cordy, “A Survey on Software Clone Detection Research,” Queen's School of Computing, Technical Report 2007-541, pp. 1-109, 2007. [[Google Scholar](#)]
- [3] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh, “Software Clone Detection: A Systematic Review,” *Information and Software Technology*, vol. 55, no. 7, pp. 1165-1199, 2013. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [4] Manpreet Kaur, and Dhavleesh Rattan, “A Systematic Literature Review on the Use of Machine Learning in Code Clone Research,” *Computer Science Review*, vol. 47, 2023. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [5] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A Multilingual Token-Based Code Clone Detection System for Large Scale Source Code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, 2002. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [6] Martin White et al., “Deep Learning Code Fragments for Code Clone Detection,” *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 87-98, 2016. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [7] Liuqing Li et al., “CCLearner: A Deep Learning-Based Clone Detection Approach,” *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai, China, pp. 249-260, 2017. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [8] Geetika, Navdeep Kaur, and Amandeep Kaur, “An Optimized Hybrid Deep Learning Model for Code Clone Detection,” *International Journal of Information Technology*, vol. 17, no. 3, pp. 1589-1595, 2025. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [9] Wenhan Wang et al., “Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree,” *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, London, ON, Canada, pp. 261-271, 2020. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [10] Chanchal K. Roy, James R. Cordy, and Rainer Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,” *Science of Computer Programming*, vol. 74, no. 7, pp. 470-495, 2009. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [11] Mohammed El Arnaoty, and Francisco Servant, “OneSpace: Detecting Cross-Language Clones by Learning a Common Embedding Space,” *Journal of Systems and Software*, vol. 208, pp. 1-31, 2024. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]

- [12] Aiping Zhang et al., "Learn to Align: A Code Alignment Network for Code Clone Detection," *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, Taipei, Taiwan, pp. 1-11, 2021. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [13] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar, "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1-38, 2020. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [14] Abdullah Sheneamer, Swarup Roy, and Jugal Kalita et al., "A Detection Framework for Semantic Code Clones and Obfuscated Code," *Expert Systems with Applications*, vol. 97, pp. 405-420, 2018. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [15] Stefan Bellon et al., "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577-591, 2007. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [16] Huihui Wei, and Ming Li, "Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code," *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 3034-3040, 2017. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [17] Maggie Lei et al., "Deep Learning Application on Code Clone Detection: A Review of Current Knowledge," *Journal of Systems and Software*, vol. 184, 2022. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [18] Qurat Ul Ain et al., "A Systematic Review on Code Clone Detection," *IEEE Access*, vol. 7, pp. 86121-86144, 2019. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [19] I.D. Baxter et al., "Clone Detection using Abstract Syntax Trees," *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, Bethesda, MD, USA, pp. 368-377, 1998. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [20] Eclipse JDT Core, Eclipse Foundation, 2026. [Online]. Available: https://download.eclipse.org/releases/2026-09/202609091000/buildInfo/archive/download.eclipse.org/releases/2026-09/202609091000/index/org.eclipse.jdt.core.source_3.47.0.v20260828-1141.html
- [21] Juraj Petrik, and Daniela Chuda, "The Effect of Time Drift in Source Code Authorship Attribution: Time Drifting in Source Code – Stylochronometry," *Proceedings of the 22nd International Conference on Computer Systems and Technologies (CompSysTech)*, Ruse Bulgaria, pp. 87-92, 2021. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [22] Wei Hua et al., "FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304-318, 2021. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [23] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, Stuttgart, Germany, pp. 301-309, 2001. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [24] Z. Li et al., "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176-192, 2006. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [25] Gang Zhao, and Jeff Huang, "DeepSim: Deep Learning Code Functional Similarity," *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lake Buena Vista FL USA, pp. 141-151, 2018. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [26] Chunrong Fang et al., "Functional Code Clone Detection with Syntax and Semantics Fusion Learning," *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Virtual Event USA, pp. 516-527, 2020. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [27] Dongjin Yu et al., "Graph-based Code Semantics Learning for Efficient Semantic Code Clone Detection," *Information and Software Technology*, vol. 156, pp. 1-13, 2023. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [28] J.I. Maletic, and A. Marcus, "Supporting Program Comprehension using Semantic and Structural Information," *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, Toronto, ON, Canada, pp. 103-112, 2001. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [29] A. Marcus, and J. I. Maletic, "Identification of High-Level Concept Clones in Source Code," *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, USA, pp. 107-114, 2001. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [30] Collin McMillan et al., "Portfolio: Finding Relevant Functions and Their Usage," *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Waikiki, Honolulu HI USA, pp. 111-120, 2011. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [31] Lili Mou et al., "Convolutional Neural Networks over Tree Structures for Programming Language Processing," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, pp. 1287-1293, 2016. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [32] Hoan Anh Nguyen et al., "Clone Management for Evolving Software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008-1026, 2012. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [33] Daya Guo et al., "GraphCodeBERT: Pre-Training Code Representations with Data Flow," *International Conference on Learning Representations (ICLR)*, pp. 1-18, 2021. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [34] J. Kennedy, and R. Eberhart, "Particle Swarm Optimization," *Proceedings of ICNN'95 - International Conference on Neural Networks*, Perth, WA, Australia, pp. 1942-1948, 1995. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]

- [35] Yoshiki Higo et al., "Incremental Code Clone Detection: A PDG-Based Approach," *2011 18th Working Conference on Reverse Engineering*, Limerick, Ireland, pp. 3-12, 2011. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [36] Jeffrey Svajlenko, and Chanchal K. Roy, "Fast and Flexible Large-Scale Clone Detection with CloneWorks," *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires, Argentina, pp. 27-30, 2017. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [37] Vaibhav Saini et al., *Oreo: Scaling Clone Detection Beyond Near-Miss Clones*, Code Clone Analysis: Research, Tools, and Practices, Springer, pp. 63-74, 2021. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [38] Jeremy R. Pate, Robert Tairas, and Nicholas A. Kraft, "Clone Evolution: A Systematic Review," *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 261-283, 2013. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [39] Chanchal K. Roy, and James R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, Amsterdam, Netherlands, pp. 172-181, 2008. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [40] Parham Moradi, and Mozghan Gholampour, "A Hybrid Particle Swarm Optimization for Feature Subset Selection by Integrating a Novel Local Search Strategy," *Applied Soft Computing*, vol. 43, pp. 117-130, 2016. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [41] Hitesh Sajjani et al., "SourcererCC: Scaling Code Clone Detection to Big-Code," *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin Texas, pp. 1157-1168, 2016. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [42] Jeffrey Svajlenko, and Chanchal K. Roy, "BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench," *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Raleigh, NC, USA, pp. 596-600, 2016. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [43] M. Nazir, A. Majid-Mirza, and S. Ali-Khan, "PSO-GA Based Optimized Feature Selection Using Facial and Clothing Information for Gender Classification," *Journal of Applied Research and Technology*, vol. 12, no. 1, pp. 145-152, 2014. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [44] Youming Wang, Jiali Han, and Tianqi Zhang, "A Relief-PGS Algorithm for Feature Selection and Data Classification," *Intelligent Data Analysis: An International Journal*, vol. 27, no. 2, pp. 399-415, 2023. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [45] Jeffrey Svajlenko, and Chanchal K. Roy, "Evaluating Clone Detection Tools with BigCloneBench," *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Bremen, Germany, pp. 131-140, 2015. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [46] Yusuke Oda et al., "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation," *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, pp. 574-584, 2015. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [47] Suresh Thummalapenta, and Tao Xie, "Mining Exception-Handling Rules as Sequence Association Rules," *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, Vancouver, BC, Canada, pp. 496-506, 2009. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [48] Morteza Zakeri-Nasrabadi et al., "A Systematic Literature Review on Source Code Similarity Measurement and Clone Detection: Techniques, Applications, and Challenges," *Journal of Systems and Software*, vol. 204, pp. 1-49, 2023. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [49] Asra Sulaiman Alshabib, Sajjad Mahmood, and Mohammad Alshayeb, "A Systematic Literature Review on Cross-Language Source Code Clone Detection," *Computer Science Review*, vol. 58, 2025. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [50] Jia Li et al., "ZC3: Zero-Shot Cross-Language Code Clone Detection," *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Luxembourg, Luxembourg, pp. 875-887, 2023. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [51] Saad Arshad, Shamsa Abid, and Shafay Shamail, "CodeBERT for Code Clone Detection: A Replication Study," *Proceedings of the IEEE 16th International Workshop on Software Clones (IWSC)*, Limassol, Cyprus, pp. 39-45, 2022. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [52] Ripon K. Saha et al., "Understanding the Evolution of Type-3 Clones: An Exploratory Study," *2013 10th Working Conference on Mining Software Repositories (MSR)*, San Francisco, CA, USA, pp. 139-148, 2013. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [53] Fahmi H. Quradaa, Sara Shahzad, and Rashad S. Almoqbily, "A Systematic Literature Review on the Applications of Recurrent Neural Networks in Code Clone Research," *PLOS ONE*, vol. 19, no. 2, pp. 1-40, 2024. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [54] Jeffrey Svajlenko, and Chanchal K. Roy, "Evaluating Modern Clone Detection Tools," *2014 IEEE International Conference on Software Maintenance and Evolution*, Victoria, BC, Canada, pp. 321-330, 2014. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [55] Miryung Kim et al., "An Empirical Study of Code Clone Genealogies," *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 187-196, 2005. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]