iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 11 **Issue:** XII **Month of publication:** December 2023

DOI: https://doi.org/10.22214/ijraset.2023.57722

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue XII Dec 2023- Available at www.ijraset.com

Transcompiler CNN-GMN Based Cross Language Clone Detection and Algorithm Recognition

Himanshu Singh[1], Teetas Bhutiya[2], Aniket Goyal[3], Kandikattu Sai Rachana[4], Umang Diyora[5]
[1]*Undergraduate Student, (Computer Science Engineering), ADGIPS, Delhi, India*
[2] *Software Development Engineer, Zigram, Gurugram, India*
[3]*Undergraduate Student, NMIMS, Indore, India*
[4]*Software Engineer, Samsung Research and Development, Delhi, India*
[5]*Undergraduate Student, Thadomal Shahani Engineering College, Mumbai, India*

Abstract: ***Software replication stemming from code reutilization introduces complexities in software maintenance. While deep learning-driven clone detection tools abound, they predominantly cater to singular programming languages. Concurrently, the task of identifying algorithms within program code remains challenging due to the absence of metadata, rendering the selection process intricate. This research seeks to establish synergies between these disparate domains by advancing innovative methodologies. The study delves into the unexplored domain of harnessing data augmentation, particularly leveraging transcompiler-based techniques, to enhance the discernment of code clones across different programming languages. Drawing upon the insights derived from transcompilers, this investigation applies data augmentation through source-to-source translation, exemplified by the Transcoder. This method extends the applicability of single-language models, such as the Graph Matching Network (GMN), to encompass cross-language detection. Concurrently, a program code classification model is introduced, leveraging Convolutional Neural Networks (CNNs) to discern algorithms based on structural features (SFs). These SFs are extracted from program codes and subsequently transformed into a one-hot binary matrix. The CNN model undergoes meticulous fine-tuning, optimizing its structural configurations and hyperparameters for superior algorithm classification. This research signifies an integration of disparate realms, intertwining cross-language clone detection and algorithm identification within a unified framework. The exploration of data augmentation for cross-language clone detection and the employment of CNN-GMNs for algorithm identification converge to furnish significant contributions to both domains. These insights, thus, hold promise for advancing the fields of software engineering and programming education.***

Keywords: Code Clone Detection, Cross-Language Clones, Data Augmentation, Deep Learning, Graph Matching Networks, Program Code Classification, Structural Features, Algorithm Identification, Convolutional Neural Network.

I. INTRODUCTION

In the domain of software engineering (SE), the implementation of algorithms is a fundamental facet embedded within the functional layers of code. The reuse of solution codes, manifesting in libraries, open-source projects, components, and application programming interfaces (APIs), is integral to expediting coding processes [1]. Code reuse, characterized by the utilization of existing code snippets to formulate new functions or code segments, necessitates a profound understanding of diverse codes and their underlying algorithms. The identification of algorithms holds paramount importance not only for developers engaged in code reuse but also for the functionality of development environments, encompassing integrated development environments (IDEs), editors, and related intelligent software tools. These tools often involve feedback mechanisms and support functions, necessitating services that facilitate various types of searches within a repository of program codes. The discernment of algorithms within code becomes particularly pertinent for advanced code analyses, spanning activities such as code cloning, refactoring, function prediction, debugging, code evaluation, and software metrics. Concurrently, in the realm of intelligent software tools, various machine learning (ML) models are intricately designed to perform tasks such as code generation, evaluation, modification, supplementation, and enhancement of source code. The accuracy and efficiency of these specialized ML models, catering to operations such as augmentation and retrieval tasks, are profoundly reliant on the precise identification of program code [2]. Consequently, the algorithmic composition inherent in the code emerges as a crucial feature, enhancing the utility and performance of ML models within the context of intelligent software tools.

The exponential growth in the volume of accumulated code poses a formidable challenge for manual code retrieval, where traditional methods involve searching through keywords, comments/documents, tags, names, and other metadata.

The inherent difficulties in this manual process are exacerbated by the unavailability, non-uniformity, and inadequacy of metadata, representing a significant impediment to efficient code retrieval. The non-uniformity arises from the diverse and freely defined nature of keywords by programmers, making them unsuitable for precise code classification. In the pursuit of locating similar codes for reference purposes, the reliance solely on metadata for finding identical codes with comparable algorithms proves insufficient. Artificial Intelligence (AI) emerges as a pivotal technology to address this challenge. Recent advancements in deep neural network (DNN) models, including recurrent neural networks (RNN), feed-forward neural networks (FNN), long short-term memory (LSTM) [3], bidirectional long short-term memory (BiLSTM) [4], and convolutional neural network (CNN) [5], have proven to be effective across diverse tasks such as computer vision [6], analysis of time series data in travel and Internet-of-Things applications [7], fault diagnosis in chemical data [8], and autonomous transportation systems [9]. In parallel, the application of DNN models has gained recognition as an effective methodology within the realm of programming activities. As a result, the integration of AI, particularly advanced DNN models, offers a promising avenue to overcome the challenges associated with code retrieval in the face of vast and varied code repositories.

Despite the notable achievements demonstrated by Deep Neural Network (DNN) models in various programming tasks, there exists a noteworthy gap in the literature about the comprehensive exploration of the structural or algorithmic features inherent in code. Understanding the algorithms embedded within program code holds significant implications for both educational and software development contexts, fostering a deeper comprehension of the codebase. Consequently, the classification of program code based on its structural features represents an unresolved challenge within the current research landscape. To bridge this gap, a novel Convolutional Neural Network (CNN)-based program code classification model is introduced, designed to contribute to both programming education and software development endeavors. The proposed model operates by discerning the algorithms encapsulated within program codes, thereby addressing the dearth of methodologies specifically tailored for classifying program codes based on their structural characteristics. Moreover, this study introduces an innovative data preprocessing approach for program codes, enhancing the robustness and efficacy of the proposed classification model. By addressing this research gap and proposing a comprehensive solution, this work aims to advance the understanding of program code structures, offering practical utility in educational settings and software development environments.

The contribution of the research work is as follows:

1) The proposed Convolutional Neural Network (CNN)- Graph Matching Networks(GMN) model demonstrates proficiency in identifying the algorithm employed in program code and subsequently classifying the code based on the discerned algorithm.

2) A novel program code processing strategy is introduced, wherein Structural Features (SFs) are extracted from program codes and transformed into a One-Hot Binary Matrix (OBM) for model training. This innovative approach enhances the model's ability to comprehend the algorithmic properties inherent in the codes.

3) To evaluate the efficacy of mutation-based data augmentation compared to the source-to-source translation-based approach, a research question is addressed. The study investigates the impact of inputting a deep-learning model with codes generated through random modification. Models trained using this approach are systematically compared with those trained with data augmented through the transcompiler-based methodology.

Detecting cross-language clones presents a formidable challenge owing to disparities in syntax and textual characteristics across source codes [23]. The inherent dissimilarities stemming from language-specific grammatical nuances render token-based and text-based traditional approaches less effective for cross-language clone detection [24]. Recent advancements in deep learning models have demonstrated commendable performance in detecting code clones within single-language contexts [25], [26], [27]. These models exhibit a notable ability to harness the underlying structure and semantics of code fragments, facilitating the identification of code clones. Notably, certain techniques for single-language clone detection are grounded in graph neural networks, which consider both syntactic and semantic features of code fragments. This approach arguably yields superior representations compared to other deep learning methods that exclusively account for syntactical features [28].

To investigate the potential utility of transcompilers in extending single-language clone detection models to address cross-language scenarios, the Graph Matching Network (GMN) was selected. GMN stands as a widely acknowledged technique for single-language clone detection, demonstrating notable proficiency on benchmark datasets. The underlying approach involves utilizing transcompilers to align the language of one code fragment with that of another, facilitating the subsequent application of a single-language clone detection tool to the transformed pair. It is essential to note that the output from transcompilers may not consistently be parseable, presenting a hindrance to direct integration with GMN. This challenge is addressed through the application of the srcML parser, enabling the construction of parseable XML representations suitable for GMN analysis.

Experimental findings indicate that while the performances of these extended GMN models may not consistently attain peak levels, they remain comparable to cutting-edge cross-language clone detection models, which typically necessitate substantial computing resources. This positions GMN as a viable alternative in resource-constrained environments. Additionally, these outcomes present avenues for further exploration, prompting consideration of refinements to the proposed framework for extending single-language clone detection models. The question of whether this framework can be enhanced or leveraged to construct more advanced cross-language clone detection models emerges as an intriguing prospect for subsequent research endeavors.

II. RELATED BACKGROUND

Codes exhibiting both syntactic and semantic resemblances are considered clones, wherein syntactic clones result from modified or transformed code fragments that retain identical functionalities. Distinctively, semantic clones involve code fragments manifesting significant structural distinctions while conveying equivalent meanings or semantics [29]. The taxonomy of clones is generally categorized into four types [30]:

1) Type-I: Identical code fragments with variations in comments and white spaces.
2) Type-II: Code fragments demonstrating syntactic equivalence with alterations in identifier names, literals, types, layout, and comments.
3) Type-III: Extending beyond Type-I and Type-II, this category encompasses additions, removals, and/or modifications of statements.
4) Type-IV: Code fragments characterized by the same functional behavior despite substantial syntactic divergence.

In the context of software systems maintained across diverse platforms, where development languages may differ, instances arise where code fragments, though authored in distinct languages, exhibit identical functionalities. Such instances are referred to as cross-language clones [31], fitting into the Type-IV classification due to their shared functional behavior despite disparate syntax.

Graph Matching Network (GMN) is a type of graph neural network (GNN) designed to assess the similarity between two homogeneous graph structures [32]. The fundamental objective of the GNN model is to generate embeddings for the nodes within a graph by assimilating information about the contextual structure and semantics. Code fragments inherently possess a structural essence that lends itself to representation as trees or graphs. In the realm of representation learning, GNN, and by extension, GMN, derive insights from their adjacent nodes to generate embeddings for each node. The efficacy of the GMN model in the domain of clone detection within single-language contexts is notable, particularly when applied judiciously with an appropriate embedding. Leveraging cross-graph attention, the GMN model ensures that comparable structures are proximate in the embedding space, while disparate structures are distanced, facilitating the creation of a comprehensive global embedding.

In recent times, machine learning (ML) has garnered significant attention as a methodology for the development of diverse software systems, spanning domains such as speech recognition, computer vision, natural language processing (NLP), robot control, and various other applications. The integration of ML capabilities into software systems manifests through various avenues, encompassing ML components, tools, libraries that encapsulate ML functionalities, and overarching frameworks. This trend has been marked by a notable characteristic: the rapid and cost-effective development and implementation of ML-enabled systems. However, a discernible challenge emerges in the form of long-term maintenance, which proves to be less economically viable [10]. The investigation conducted by Wan et al. delves into the distinctions in software development practices between ML and non-ML contexts [11]. Furthermore, insights into prevalent practices and workflows for constructing large-scale ML applications, systems, and platforms have been elucidated based on experiences at technology giants such as Microsoft, Amazon, and Google [12]. Proposals for testing and debugging tools tailored for ML-based applications and systems have also been advanced in the literature [13]. Despite these endeavors, the imperative for the standardization and operationalization of reliable ML systems remains evident. Drawing from real-world ML-enabled software development practices, a comprehensive set of approximately 11 challenges has been identified, spanning from data collection to model evolution, evaluation, and deployment. In this context, our proposed classification model assumes a supporting role, particularly in the construction of large-scale ML-based applications and systems dealing specifically with structural features (SFs).

In a seminal work [14], the introduction of a multi-modal attention network (MMAN) was proposed to adeptly capture the structural features (SFs) of source codes, thereby enhancing the interpretability of features influencing the results. The MMAN framework adeptly encapsulates both structured and non-structured features inherent in source codes, employing a tree-based Long Short-Term Memory (tree-LSTM) for the abstract syntax tree (AST) and a gated graph neural network (GNN) for the control flow graph. This comprehensive approach aims to provide a nuanced representation of the intricate features within source codes.

In a related endeavor [15], an LSTM-based model was conceived for the identification of source code errors in the context of C programming. The model intricately encodes characters, variables, keywords, tokens, numbers, functions, and classes with predefined identifiers, demonstrating a meticulous approach to error detection. Notably, the model exhibits high accuracy in discerning errors within faulty solution codes. Terada et al. [16] presented an intriguing model designed for predicting subsequent code sequences to facilitate code completion. Leveraging an LSTM network architecture, their model serves as a valuable resource for novice programmers grappling with the challenge of crafting complete code from scratch. The efficacy of the model is evident in its capability to predict accurate words for completing the code, thereby assisting in the learning process for individuals acquiring programming proficiency. Additionally, the application of LSTM neural networks has been extended to diverse tasks, including code evaluation, completion, and repair, across various stages of programming learning [17]. This broad spectrum of applications underscores the versatility and utility of LSTM models in addressing multifaceted challenges within the programming domain.

Ugurel et al. [18] conducted a dual-faceted classification study utilizing Support Vector Machines (SVM). Their approach involved the first classification task, wherein they discerned programming languages, and the second task, where they classified distinct categories of programs, such as databases, multimedia, and graphics. Tian et al. [19] employed a Latent Dirichlet Allocation (LDA) mapping method to ascertain the programming language associated with source code, grounded in the analysis of textual elements. Alreshedy et al. [20] introduced a machine-learning language model tailored for the classification of source code snippets based on programming language attributes. Utilizing a Multinomial Naive Bayes (MNB) classifier, they categorized source code snippets, with a specific focus on features such as comments, variables, and functions, in contrast to syntactic information. Reyes et al. [21] presented a model utilizing Long Short-Term Memory (LSTM) networks for the classification of source code based on written programming languages. This study focused on archived source codes and demonstrated that the LSTM model outperformed Naive Bayes and linguistic classifiers in empirical evaluations. Gilda employed a Convolutional Neural Network (CNN) model [22] to discern programming languages from source code snippets, showcasing the versatility of neural network architectures in language identification tasks.

III. RESEARCH METHODOLOGY

In the present investigation, our focus centers on the generation of data tailored for cross-language clone detection models, employing a source-to-source translation methodology. Specifically, we utilized the Transcoder, a pre-trained deep learning model, to facilitate the conversion of code snippets from one programming language to another within our dataset, which comprises Java and Python codes. This transposition involved the conversion of all Java codes to Python and vice versa. The efficacy of the Transcoder, owing to its pre-trained nature, lies in its capacity to generate fragments that maintain semantic equivalence with the original counterparts. Notably, the patterns discerned by the pre-trained model are not contingent upon manually crafted rules, rendering the generated codes diverse in nature. Details pertaining to the semantic, syntactic, and computational accuracy of the generated codes can be found in the accompanying literature [citation]. The schematic representation of our study's overarching approach, as illustrated in Figure 1, revolves around the examination of the influence of data augmentation on cross-language clone detection models. Initially, each code fragment from the original dataset undergoes conversion to obtain an alternative version in another programming language (e.g., Java to Python and vice versa). Subsequently, both the original code fragments and their transcompiled counterparts are archived in a database for subsequent assessment of the augmentation's impact. In the ensuing phase, the models undergo training through two distinct methodologies: one exclusively utilizing the original dataset and the other incorporating the augmented dataset. The ensuing computation of model accuracies enables a comparative analysis, elucidating the discernible impact of data augmentation on the efficacy of the cross-language clone detection models.

To illustrate the application of transcompilers in extending single-language clone detection tools to cross-language settings, we leverage a Graph Matching Network (GMN). GMN, recognized as a proficient single-language clone detection model, has demonstrated superior performance across all four types of clones, owing to its adept utilization of both the structural and semantic aspects of code fragments, enhancing its robustness in source code analysis. The intricate process of employing GMN in conjunction with Transcoder is delineated in Figure 1. GMN, designed primarily for matching homogeneous structures, bases its embedding space on the textual information of tokens for each node. Consequently, diverse structures and texts yield distinct embeddings within GMN. To mitigate this, we designate Java as the target language due to its capacity to accommodate varied structures. The process unfolds with the transmission of a Python code fragment from a given pair through Transcoder, yielding a converted code representation in Java. Subsequently, both fragments undergo processing through srcML to generate an Abstract Syntax Tree (AST) representation, serving as the basis for model training. Once the model is trained, any fragment can be subjected to the same conversion steps for testing, thereby validating the effectiveness of the cross-language clone detection model.

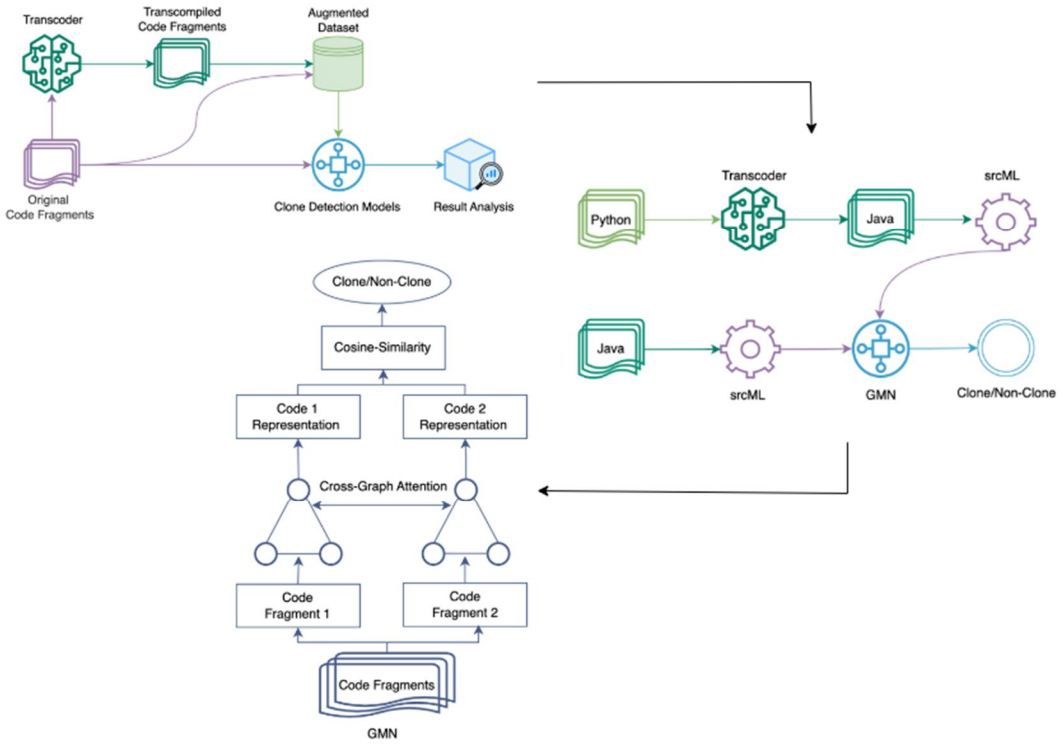


Fig. 1. CNN-GMN is combined using a transcompiler.

The program code transformation process involves the exclusive extraction of structural properties for tokenization. Standard program code constituents, including operators, operands, loops, branches, keywords, methods, and classes, constitute the primary focus of this extraction, representing key attributes inherent in program code structures. In contrast, elements of a user-defined nature, such as comments, variables, classes, and functions, which wield comparatively minimal impact, are omitted from consideration. A delineation of featured tokens (T) alongside their corresponding identifiers is presented in Table 3. Initiating the process, structural features (SFs) are systematically extracted from program codes following the procedural guidelines outlined in Algorithm 1. The ensuing program code preprocessing steps, delineated in subsequent subsections, embody the comprehensive methodology employed in this transformative process.

Steps:

1) *Deletion of Comments:* The initial preprocessing step involves the identification and removal of all comments within the program code. This process is executed through the implementation of the removeComments() function. Given the non-significant nature of comments in the program code, their deletion contributes to the streamlining of subsequent analyses and deletion of comments: All comments in the program code are identified and removed with the removeComments() function because the comments in the program code are not significant.
2) *Extraction of Feature Tokens:* After the removal of comments from the code, the identification and selection of feature tokens ensue. Feature tokens encompass elements such as conditional statements (e.g., if, else), loops, mathematical operators, bitwise operators, assignment operators, compound assignment operators, comparison operators, as well as symbols like braces, parentheses, and square brackets. In the context of C++ programming, parentheses conventionally serve purposes such as function calls and declarations, conditional statements (e.g., if, while, do), loops, and operator precedence. Similarly, braces are employed for encapsulating functions, classes, and structs, as well as conditional statements and loops, while square brackets facilitate array access. The extractSelectedFeatures() function is employed for the extraction of all relevant feature tokens within the program code, as illustrated in Figure 2. In parallel, the identification and subsequent removal of irrelevant tokens, such as user-defined functions and variables, are undertaken. Notably, these user-defined elements, encompassing variable and function names arbitrarily assigned by programmers, exhibit variability within codebases.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue XII Dec 2023- Available at www.ijraset.com

The elimination of these dynamic elements, guided by the principles of static typing in C++ and the varying nomenclature employed by programmers, enhances the contextual clarity of the code. This preprocessing step is essential to ensure that the Deep Neural Network (DNN) model comprehends the code context more effectively.

```
/*C++ code*/
#include<iostream>
#include<algorithm>
#include<vector>
#include<math>
using namespace std;
int main(){
    int k, most_recent[10];
    for(int i=0; i<10; i++){
       cin>> most_recent[i]; //taking 10 inputs
    }
    sort(most_recent, most_recent+10, greater<int>());
     for(int i=0; i<3; i++){
       cout<< most_recent[i] << endl;
     }
}
```

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<math>
using namespace std;
int main(){
    int k, most_recent[10];
    for(int i=0; i<10; i++){
       cin>> most_recent[i];
    }
    sort(most_recent, most_recent+10, greater<int>());
     for(int i=0; i<3; i++){
       cout<< most_recent[i] << endl;
     }
}
```

Fig. 2 Identification and removal of unnecessary comments from the code

3) *Tokenization of the features:* All feature tokens extracted from the code, as depicted in Figure 2, undergo the subsequent step of conversion into token IDs. This transformative process, referred to as tokenization or encoding, serves the purpose of representing each Structural Feature (SF) of the code as a distinct token. The tokenization/encoding methodology employed in this research entails associating each SF with a numeric identifier, facilitating the feeding of these tokens into Deep Neural Network (DNN) models. Within the framework of DNN model learning, a sequence of tokens is effectively translated into a sequence of numerical vectors, subsequently processed by the neural network. It is crucial to underscore that DNN models lack inherent knowledge of the SFs, such as {+ & = []}, or an understanding of the semantic or algorithmic features embedded in the code. Hence, the tokenization/encoding process assumes paramount significance in enabling DNN models to learn the intricacies of the neural network model from scratch. For instance, when features like {+ & = []} are extracted from the code, the tokenization/encoding process translates these features into numeric representations, exemplified by 13 1 2 0 15 16 14. Upon completion of the tokenization process, the code is effectively transformed into a sequence of numeric IDs, as illustrated in Figure 2.
4) *One-hot binary matrix conversion from token IDs:* Following the tokenization process, each feature token is assigned a unique identifier represented by token IDs. The resultant sequence of token IDs transforms a matrix structure denoted by P × Q, where P signifies the total number of token IDs, and Q is equal to the highest token ID value plus one. Given the defined range of token IDs (0 to 16), the maximum length of Q (columns of matrices) is determined to be 17.

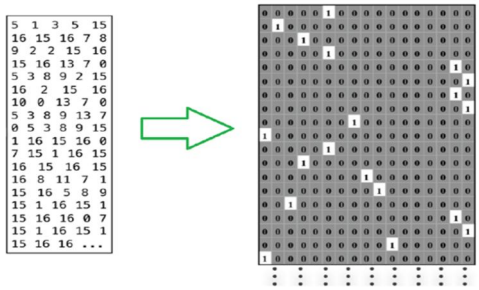


Fig. 3 OBM conversion process

IV. IMPLEMENTATION & RESULTS

In this investigation, Structural Features (SFs) are extracted from program codes, serving as the foundational elements for subsequent training of a Convolutional Neural Network (CNN) model designed for program code classification. The model's proficiency lies in its ability to categorize program codes based on algorithmic classifications, achieving a notable F1 score of approximately 91.7%. The elevated precision underscores the effectiveness of the proposed methodology, encompassing SFs extraction, One-Hot Binary Matrix (OBM) conversion, and the training and evaluation phases of the optimal CNN model using real-world program codes.

Table 1: Experimental results comparison with baseline models of different studies

Models	LoC	Language	Classification type	Accuracy (%)
Stacked Bi-LSTM	35,500	Multi-lingual	Multi-Class	89.34%
LSTM	10,400	C	Binary	78.00%
RF	3,340	Java	Binary	50.01%
DP-ARNN	3,400	Java	Binary	57.00%
CNN-GMN	62,000	C++	Multi-Class	95.87%

Furthermore, the experimental scope extends to program codes scripted in C++, a representative procedural programming language. The encompassing nature of the proposed model's applicability is demonstrated by its successful classification of program codes written in C++, suggesting its potential utility in analogous tasks involving procedural languages such as Python, Java, and C. This cross-applicability reinforces the versatility and generalizability of the proposed model beyond its initial context, enhancing its potential impact within the broader spectrum of procedural programming languages.

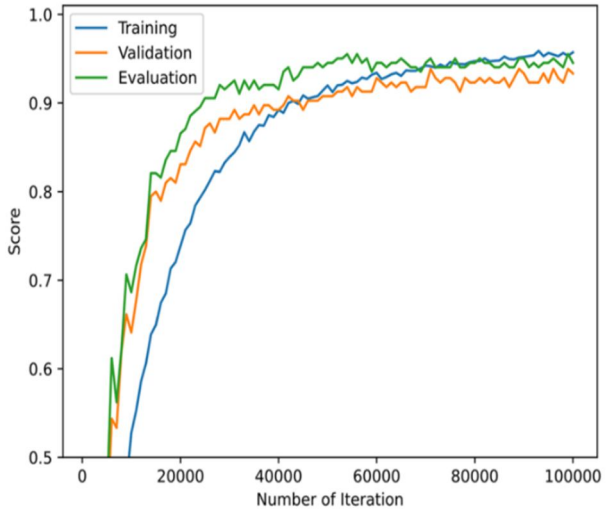


Fig 4: Training-Validation-Evaluation of CNN-GMN

This paper concentrates on the training of Deep Neural Network (DNN) models, emphasizing the utilization of algorithmic features inherent in the code rather than relying on meta-information. Structural Features (SFs), recognized as integral components delineating the algorithm within each solution code, are considered key factors in model training. A substantial corpus of practice-oriented solution codes is amassed and processed to facilitate the training and evaluation of the model. Comprehensive experiments are conducted, encompassing diverse Convolutional Neural Network (CNN) architectures and hyperparameter configurations.

Among the various DL models explored, the CNN-GMN model emerges as the most proficient, exhibiting superior training, validation, and evaluation accuracy when compared to its counterparts. Comparative analyses are extended to include CNN, Long Short-Term Memory (LSTM), and Bidirectional LSTM (BiLSTM) models, elucidating their respective classification performances. Experimental outcomes substantiate that DNN models adeptly recognize algorithms embedded within solution codes, achieving a commendable level of accuracy. This outcome underscores the model's exceptional accuracy in categorizing "program codes" in the absence of meta-information, thereby affirming its robust performance in algorithm recognition.

Table 2: Comparative Analysis of DL Models & Non-DL Models.

MODELS	Name of Model	Precision	F1
Deep Learning Models	GMN-CNN	0.90	0.91
Non-DL Models	CLCMiner	0.36	0.44

In summary, the experimental findings underscore the considerable efficacy of transcompiler-based data augmentation in significantly enhancing the performance of deep learning models for cross-language clone detection. Notably, the precision and F1-score metrics for the deep learning models yielded values of 0.90 and 0.91, respectively, underscoring the substantial improvements achieved through the proposed data augmentation technique.

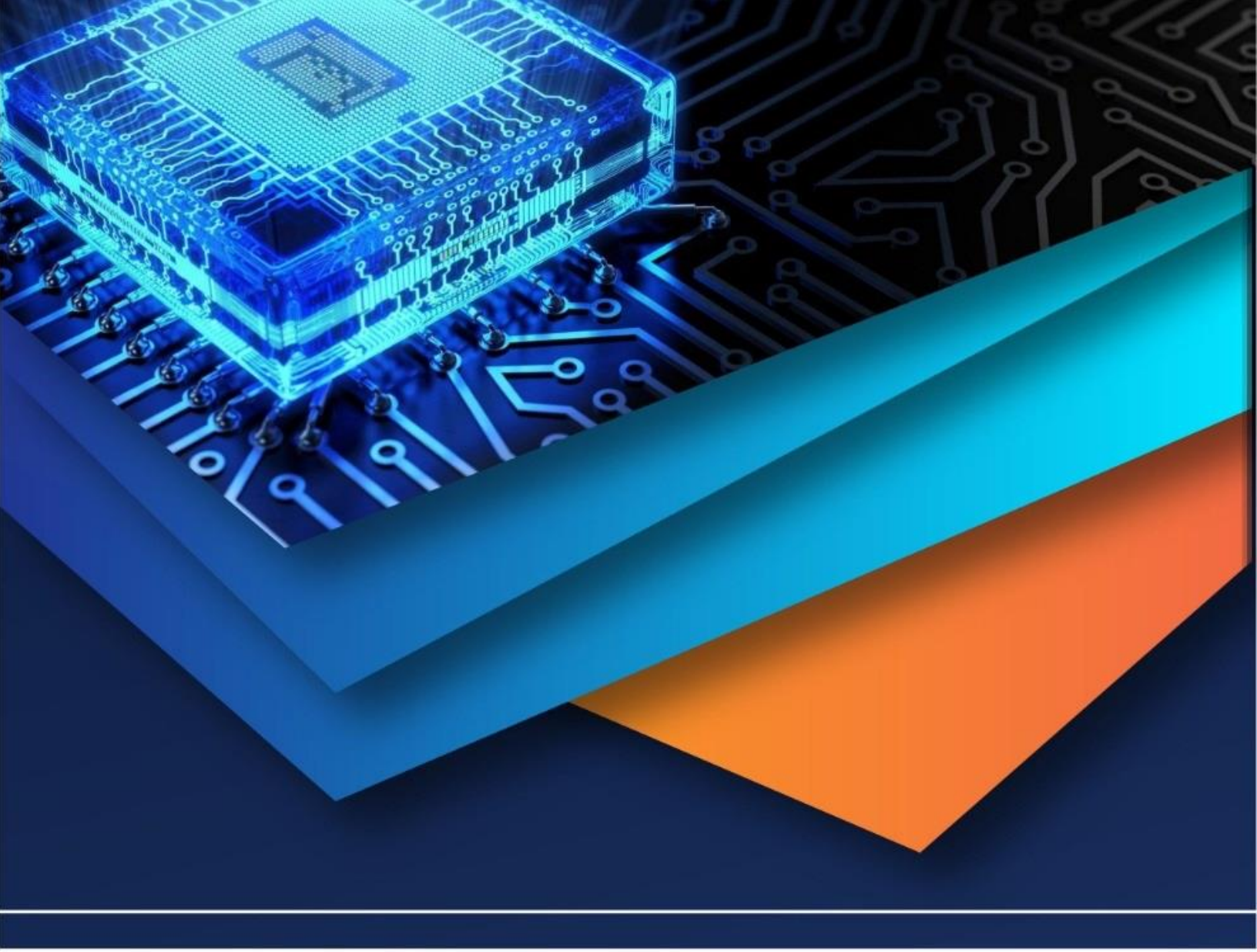
V. CONCLUSION

In this study, Convolutional Neural Network (CNN)-Graph Matching Network (GMN) models were devised for the classification of program codes based on identified algorithms. Structural features (SFs) extracted from the program codes served as the basis for CNN model learning, wherein they transformed a one-hot binary matrix and subsequent processing steps. Various combinations of hyperparameters, including CL, LR, AF, and BS, were systematically employed in the CNN models, with the top-performing models and their corresponding hyperparameters selected based on superior experimental results. A rigorous 10-fold cross-validation further determined the most suitable CNN model and hyperparameters for subsequent experiments, culminating in an optimized model achieving an accuracy of 95.87%. Furthermore, a comparative analysis against baseline models demonstrated the superior performance of the proposed CNN model in classifying program codes across diverse algorithms. Notably, the scalability of the proposed model was affirmed, indicating its potential applicability to program codes written in other procedural programming languages such as C, Java, Python, and C#. Future avenues for research include exploring code block sequences instead of structural features (SFs) to assess model performance, considering multi-label classification models for codes with multiple labels, and extending the model's evaluation to large-scale industrial program codes. In parallel, the research addressed the underexplored domain of cross-language clone detection through the introduction of a novel data augmentation technique utilizing a transcompiler. Leveraging pre-trained deep learning models for source-to-source translation, the proposed approach demonstrated enhanced performance in cross-language clone detection models. The experimental findings underscore the considerable efficacy of transcompiler-based data augmentation in significantly enhancing the performance of deep learning models for cross-language clone detection. Notably, the precision and F1-score metrics for the deep learning models yielded values of 0.90 and 0.91, respectively, underscoring the substantial improvements achieved through the proposed data augmentation technique. Additionally, the adaptation of a single-language model for cross-language clone detection exhibited a substantial margin of improvement over existing baselines, suggesting avenues for further optimization. Future research directions include the incorporation of explainable AI techniques to elucidate the efficacy of data augmentation and the development of strategies for judiciously selecting augmented data to augment the overall effectiveness of the proposed approach.

REFERENCES

[1] Taibi F (2013) Reusability of open-source program code: a conceptual model and empirical investigation. SIGSOFT Softw. Eng. Notes 38(4):1–5. https://doi.org/10.1145/2492248.2492276
[2] Wan Z, Xia X, Lo D, Murphy GC (2021) How does machine learning change software development practices? IEEE Trans Softw Eng 47(9):1857–1871. https://doi.org/10.1109/TSE.2019. 2937083
[3] Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780. https://doi.org/10.1162/neco. 1997.9.8.1735
[4] Schuster M, Paliwal K. K (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45(11):2673–2681. https://doi.org/10.1109/78.650093
[5] Krizhevsky A, Sutskever I, Hinton G. E (2017) Imagenet classification with deep convolutional neural networks. Commun ACM 60(6):84–90. https://doi.org/10.1145/3065386
[6] Gao H, Xiao J, Yin Y, Liu T, Shi J (2022) A mutually supervised graph attention network for few-shot segmentation: the perspective of fully utilizing limited samples. IEEE Trans Neural Netw Learn Syst :1–13
[7] Ran X, Shan Z, Fang Y, Lin C (2019) An lstm-based method with attention mechanism for travel time prediction. Sensors 19(4):861. https://doi.org/10.3390/s19040861
[8] Zhao H, Sun S, Jin B (2018) Sequential fault diagnosis based on lstm neural network. IEEE Access 6:12929–12939. https://doi.org/10.1109/ACCESS.2018.2794765
[9] Gao H, Huang W, Liu T, Yin Y, Li Y (2022) Ppo2: location privacy-oriented task offloading to edge computing using reinforcement learning for intelligent autonomous transport systems. In: IEEE Transactions on Intelligent Transportation Systems, pp 1–14
[10] Jordan MI, Mitchell TM (2015) Machine learning: trends, perspectives, and prospects. Science 349(6245):255–260. https://doi.org/10.1126/science.aaa8415
[11] Amershi S, Begel A, Bird C, DeLine R, Gall H, Kamar E, Nagappan N, Nushi B, Zimmermann T (2019) Software engineering for machine learning: a case study. In: 2019 IEEE/ACM 41st international conference on software engineering: software engineering in practice (ICSE-SEIP), pp 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042
[12] Salvaris M, Dean D, Tok WH (2018) Microsoft AI platform. Apress, pp 79–98. https://doi.org/10.1007/978-1-4842-3679-6 4
[13] Ma L, Juefei-Xu F, Zhang F, Sun J, Xue M, Li B, Chen C, Su T, Li L, Liu Y, Zhao J, Wang Y (2018) DeepGauge: multi-granularity testing criteria for deep learning systems. Association for Computing Machinery, New York, pp 120–131. https://doi.org/10.1145/3238147.3238202
[14] Wan Y, Shu J, Sui Y, Xu G, Zhao Z, Wu J, Yu P (2019) Multi-modal attention network learning for semantic source code retrieval. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE), pp 13–25. https://doi.org/10.1109/ASE.2019.00012

[15] Teshima Y, Watanobe Y (2018) Bug detection based on lstm networks and solution codes. In: 2018 IEEE international conference on systems, man, and cybernetics (SMC), pp 3541– 3546. https://doi.org/10.1109/SMC.2018.00599
[16] Terada K, Watanobe Y (2021) Code completion for programming education based on deep learning. Int J Comput Intell Stud 10(2- 3):78–98. https://doi.org/10.1504/IJCISTUDIES.2021.115424
[17] Rahman MM, Watanobe Y, Nakamura K (2020) A neural network based intelligent support model for program code completion. Sci Program 2020:18. https://doi.org/10.1155/2020/7426461
[18] Ugurel S, Krovetz R, Giles CL (2002) What's the code? automatic classification of source code archives. In: Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining. KDD '02, ACM, pp 632–638. https://doi.org/10.1145/775047.775141
[19] Tian K, Revelle M, Poshyvanyk D (2009) Using latent dirichlet allocation for automatic categorization of software. In: 2009 6th IEEE international working conference on mining software repositories, pp 163–166. https://doi.org/10.1109/MSR.2009.5069496
[20] Alreshedy K, Dharmaretnam D, German DM, Srinivasan V, Gulliver TA (2018) Scc: automatic classification of code snippets. In: 2018 IEEE 18th international working conference on source code analysis and manipulation (SCAM), pp 203–208. https://doi.org/10.1109/SCAM.2018.00031
[21] Reyes J, Ram´ırez D, Paciello J (2016) Automatic classification of source code archives by programming language: a deep learning approach. In: 2016 international conference on computational science and computational intelligence (CSCI), pp 514–519. https://doi.org/10.1109/CSCI.2016.0103
[22] Gilda S (2017) Source code classification using neural networks. In: 2017 14th international joint conference on computer science and software engineering (JCSSE), pp 1–6. https://doi.org/10.1109/JCSSE.2017.8025917
[23] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: Contrastive cross-language code clone detection," in 2022 30th IEEE/ACM International Conference on Program Comprehension (ICPC), pp. 413–424, 2022.
[24] L. Nichols, M. Emre, and B. Hardekopf, "Structural and nominal crosslanguage clone detection," in International Conference on Fundamental Approaches to Software Engineering, pp. 247–263, Springer, 2019.
[25] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 783–794, IEEE, 2019.
[26] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 261–271, IEEE, 2020.
[27] S. B. Ankali and L. Parthiban, "Detection and classification of cross-language code clone types by filtering the nodes of antlr-generated parse tree," International Journal of Intelligent Systems and Applications, vol. 13, no. 3, pp. 43–65, 2021.
[28] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in International Conference on Learning Representations, 2018.
[29] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's School of Computing TR, vol. 541, no. 115, pp. 64– 68, 2007.
[30] F. Al-Omari, C. K. Roy, and T. Chen, "Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge," in 2020 IEEE 14th International Workshop on Software Clones (IWSC), pp. 57–63, IEEE, 2020.
[31] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: cross language code clone detection using syntactical features and api documentation," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1026–1037, IEEE, 2019.
[32] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in International conference on machine learning, pp. 3835–3845, PMLR, 2019.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY