Decoding the DNA of Code: An AI-Infused Approach to Detect Code Cloning in Software Systems

By

Nikita Mehrotra
PhD18013

Under the supervision of Rahul Purandare

Computer Science and Engineering

Indraprastha Institute of Information Technology Delhi

New Delhi– 110020

June, 2024

DECODING THE DNA OF CODE: AN AI-INFUSED APPROACH TO DETECT CODE CLONING IN SOFTWARE SYSTEMS

BY

NIKITA MEHROTRA
PhD18013

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

COMPUTER SCIENCE AND ENGINEERING

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI

NEW DELHI– 110020

JUNE, 2024

Certificate

This is to certify that the thesis titled *Decoding the DNA of Code: An AI-Infused Approach to Detect Code Cloning in Software Systems* being submitted by *Nikita Mehrotra* to the Indraprastha Institute of Information Technology Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by her under my supervision. In my opinion, the thesis has reached the standard fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree or diploma.

Dr. Rahul Purandare

21 June, 2024

Department of Computer Science & Engineering, IIIT-Delhi

Abstract

Code clones, duplicate code fragments sharing similar syntax or semantics, have become increasingly prevalent due to the success of software management tools like GitHub and advancements in Open Source Software (OSS). Previous research has shown that an astonishing 70% of the code hosted on GitHub consists of clones derived from previously existing files. Furthermore, research has also found that between 9% and 31% of software projects on Github contain a substantial portion, sometimes up to 80%, of files that have identical counterparts elsewhere. While clones facilitate code reuse and refactoring, they simultaneously complicate software evolution, necessitating effective clone detection techniques.

Historically, substantial amount of research has been conducted on code clone detection, most traditional approaches focus on syntactic clones by leveraging lexical and syntactic information. However, only a few of them target semantic clones. Furthermore, the evolution of software engineering has led to the development of modern multilingual software from traditional mono-language systems, where functionality replication across multiple programming languages is common. This results in clones having similar functionality but belonging to different languages. Since such code snippets are syntactically unrelated, traditional single-language clone detection approaches are not feasible for their detection.

Motivated by the success of deep learning models in various domains, researchers have explored deep learning techniques for code clone detection. These techniques leverage the power of machine learning to learn the underlying patterns and features of code to measure code similarity. However, the majority of these techniques rely on supervised learning, which necessitates a substantial volume of labeled data to achieve optimal performance. The acquisition and creation of such labeled datasets present considerable challenges, as they involve not only the scarcity of accurately labeled examples but also the laborious and time-consuming process of manual annotation.

In the face of inadequate labeled data, the supervised techniques often encounter significant limitations when applied to new benchmarks or datasets, as they struggle to adapt to the issue of domain shift. This limits the generalizability of supervised techniques, impeding their practical applicability and effectiveness in diverse and evolving software systems.

To address the challenges and enhance semantic code clone detection in modern software systems, this thesis presents the following contributions through the investigation of three innovative approaches: 1) We propose a novel method to model semantic similarity between code snippets by utilizing customized graph neural networks for code combined with program dependency graphs. This approach effectively leverages the structured syntactic and semantic information present within the code snippets. We have developed a prototype tool, called "HOLMES", based on this approach and rigorously evaluated its performance on popular code

clone benchmarks. 2) To model cross-language code similarity, we introduce a semi-supervised deep learning tool, "RUBHUS", which leverages control and data flow-enriched abstract syntax trees (ASTs). We demonstrate the effectiveness of "RUBHUS" through experiments conducted on datasets consisting of Java, C, and Python programs, showcasing its ability to detect cross-language clones compared to other state-of-the-art cross-language and single-language clone detection tools. 3) We propose an adversarial unsupervised domain adaptation approach and tool, "CLODIA", which employs multiple latent spaces for domain adaptation. This approach enhances the performance and generalization capabilities of learning-based clone detection techniques on unseen domains, reducing the need for human annotation. We conduct extensive evaluations of "CLODIA" on various datasets, including programming competition and open-source datasets, as well as a handcrafted dataset of clones curated from real-world software systems.

In summary, our research addresses a critical gap in the realm of semantic and cross-language code clone detection, offering innovative solutions and prototype tools as proof-of-concept. Rigorously tested against popular code clone benchmarks, these tools showcase their effectiveness by outperforming state-of-the-art counterparts. The pivotal findings and insights gleaned from this thesis not only advance our comprehension of clone detection in contemporary software systems but also tackle the challenges stemming from labeled data scarcity and semantic clone detection, paving the way for future research in this field.

To my Father,
for his unwavering faith in me and always being the steadfast pillar of support in my life.

Acknowledgements

"A journey of a thousand miles begins with a single step." - Lao Tzu

With this quote in mind, I would like to express my sincerest gratitude and appreciation to all those who have supported me throughout the journey of completing this thesis.

First and foremost, I wish to convey my deepest gratitude and sincere appreciation to my thesis advisor, Dr. Rahul Purandare, for his steadfast support, mentorship, and guidance throughout my research journey. Rahul's extensive knowledge, expertise, and enthusiasm for the subject have consistently inspired me to excel and develop both academically and personally. I consider myself incredibly fortunate to have such a friend, philosopher, and guide in my Ph.D. advisor, who encouraged me to maintain a positive attitude towards my work and life. Rahul has not only been an exceptional mentor but also a firm believer in my abilities, consistently trusting and enabling me to explore new horizons and push the boundaries of my research. His pursuit of knowledge, dedication to high-quality work, and strong work ethics will continue to inspire me. Some crucial lessons I learned during my time with him include critiquing work effectively and embracing criticism as an opportunity for improvement. The invaluable advice, constructive feedback, and timely encouragement I received from Rahul have been crucial in molding my thought process, refining my skills, and nurturing my resilience in overcoming challenges. Moreover, I am profoundly thankful for the freedom and autonomy Rahul granted me while conducting my research, which fostered an atmosphere that encouraged creativity, curiosity, and intellectual growth. This degree of trust and support has been critical in my development as a researcher and has enabled me to truly thrive in my field. I wholeheartedly acknowledge the transformative impact Rahul has had on my life and career. I will be forever indebted to him for his unwavering faith in me, his unwavering dedication to my success, and the countless lessons he has imparted. I am what I am today due to his guidance and the opportunities he has provided me, for which I am eternally grateful.

I would also like to express my heartfelt appreciation and gratitude to my collaborators, Dr. David Lo and Dr. Saket Anand, who have played a vital role in my Ph.D. journey. Their guidance, insights, and invaluable feedback have significantly contributed to the success of my dissertation. Rahul has been instrumental in sharpening my program analysis skills and providing feedback in that area. Saket's expertise in deep learning has greatly benefited my work. His guidance in this realm has broadened my understanding and allowed me to delve deeper into this captivating field. I am truly thankful for his contributions to my research and the knowledge he has shared with me. David has also been an exceptional part of this journey, from whom I have acquired a wide range of skills. His guidance in defining the problem statement, transforming research ideas into prototypes, and effectively presenting the work has been invaluable. I am immensely grateful for his mentorship, which has shaped my approach to research and has left

an enduring impact on my academic development.

This dissertation has received support through the Prime Minister's Fellowship Scheme for Doctoral Research, a collaborative initiative between the Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India, and the Confederation of Indian Industry (CII). I extend my gratitude to the Apex Council and everyone involved in this initiative. I appreciate the efficient management of the fellowship by Ms. Neha Gupta and Mr. Ravi Hira, who were always available to address my concerns. The fellowship has also been jointly supported by Nucleus Software, as the industry partner. I am grateful to Mr. Praveen Jain, my industry mentor, and Nucleus Software for generously providing fellowship support and other opportunities.

Engaging in conversations with the members of my research committee, Dr. David Lo, Dr. Saket Anand, and Dr. Vivek Kumar, has significantly contributed to the development of my Ph.D. journey. Their involvement went beyond merely reviewing my Ph.D. progress; they were always available to provide advice and guidance whenever I needed it. The insights and perspectives they shared have greatly influenced the shaping of my research, and I am truly grateful for their invaluable support throughout this journey.

I would like to convey my appreciation to the administrative staff, the IT team, and the support staff at IIIT Delhi. Their diligent efforts and assistance have enabled me to concentrate on my research activities without any major distractions.

I would like to take this opportunity to express my gratitude to all the members of my PAG Lab for fostering a research-friendly environment. Their presence made my research experience at IIIT-Delhi more welcoming and fruitful. I am particularly grateful to my colleagues Dhriti, Devika, Ridhi, and Khushboo for the inspiring and productive collaborations we shared. Over time, we also formed close friendships that made my Ph.D. journey more enjoyable. Their unwavering support, camaraderie, and friendship have truly enriched my graduate life, creating cherished memories that will last a lifetime.

I would like to extend my gratitude to my co-authors of the papers published throughout my Ph.D. journey. Collaborating with each one of you has been a truly enriching experience. I am profoundly appreciative of your contributions and the shared efforts in our research endeavors.

I would also like to extend my heartfelt gratitude to my partner Abhinav Sharma, whose unwavering support and encouragement have been the bedrock of my thesis journey. He not only provided emotional support but also went above and beyond by making a tangible difference in my daily life. During hectic deadlines, he selflessly fetched coffee, keeping me fueled and focused when the pressure was at its peak. We stayed awake together during those late nights, turning what could have been lonely hours into shared moments of perseverance. Moreover, he also played an integral role in shaping the quality of my work while generously dedicating time to review and providing thoughtful feedback on my writings, offering a fresh perspective that significantly enriched the final output. These actions were more than gestures; they were the embodiment of true partnership. Thank you, Abhinav, for being my anchor, confidante, and cheerleader. Your exceptional support, especially during those crucial moments, has left an indelible mark on my thesis journey, and I am profoundly grateful for your unwavering presence in my life.

Finally, I would like to express my deepest gratitude to my family – my mother, father,

and brother – for their unwavering love, support, and encouragement throughout my academic journey. Their constant presence and belief in me have been the pillars of strength that I needed in pursuing my dreams. In particular, I am incredibly grateful to my father for his countless sacrifices and tireless dedication to ensuring that I have the best opportunities in life. From providing emotional support during challenging times to celebrating my accomplishments, he has been there every step of the way. Throughout the tough times, his unwavering presence has been invaluable, and no words of gratitude will ever be sufficient to express my appreciation and my love for you. His dream of seeing the "Dr." prefix in front of my name will soon come true, and it is all thanks to his steadfast belief in me and my abilities. I am truly blessed to have such an inspiring and loving father, and I wholeheartedly dedicate this thesis to him.

nikita

Research Papers From This Work

1. **N. Mehrotra**, A. Sharma, and R. Purandare, (2024). "Improving Semantic Code Clones Detection with Adversarial Domain Learning" [Under review at TSE'24].
2. **N. Mehrotra**, A. Sharma, and R. Purandare, (2023). "Improving Cross-Language Code Clone Detection via Code Representation Learning and Graph Neural Networks". 2023 IEEE Transactions on Software Engineering as a Journal First paper (**(Presented at ICSE 2024)**.
3. **N. Mehrotra**, N. Agarwal, P. Gupta, S. Anand, D. Lo and R. Purandare, (2021). "Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks". 2021 IEEE Transactions on Software Engineering as a Journal First paper (**(Presented at ICSE 2022)**.
4. P. Gupta, **N. Mehrotra**, and R. Purandare, (2020). "JCoffee: Using Compiler Feedback to Make Partial Code Snippets Compilable". 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Tool Track, 810-813.

Contents

Abstract **i**

Dedication **iii**

Acknowledgements **iv**

Publications **vii**

List of Tables **xiii**

List of Figures **xv**

1 Introduction **1**

1.1 The Imperative for Code Clone Detection 2

1.2 A Comprehensive Look at Code Clone Detection 3

1.3 Identified Challenges and Proposed Solutions 5

1.4 Thesis Statement 11

1.5 Contribution 12

1.6 Outline of Dissertation 14

2 Background **15**

2.1 Code Clones 15

2.2 Program Representations 18

2.2.1 Abstract Syntax Trees 18

2.2.2 Program Dependence Graphs 20

2.3 Deep Learning 21

2.3.1 Artificial neural networks 22
2.3.2 Graph Neural Networks 22
2.3.3 Attention Neural Networks 26
2.3.4 Siamese Networks 26
2.3.5 Domain Adaptation 27
2.3.6 Neural Code Representation Learning 29
2.4 Learning-based Code Clone Detection Process 30

3 Related Work 32
3.1 Traditional Approaches to Code Clone Detection 32
3.2 Learning-based Techniques to Code Clone Detection 35
3.3 Representation Learning for Source Code 40

4 Graph-Based Siamese Networks for Assessing Functional Similarity in Source Code 42
4.1 Introduction 42
4.2 Motivation 45
4.2.1 Motivating Example. 45
4.2.2 Key Ideas. 47
4.3 Approach Overview 48
4.3.1 Attention-based Global Context Learning 48
4.3.2 LSTM-based Local Context Learning. 52
4.3.3 PDG Representation Learning with Jumping Knowledge Networks and Soft Attention 53
4.3.4 Edge-Attributed PDG Representation Learning 54
4.3.5 Implementation and Comparative Evaluation. 55
4.4 Experimental Design 56
4.4.1 Datasets 57
4.4.2 Experimental Procedure and Analysis. 59
4.5 Results 60
4.5.1 RQ 4.1: How Effective is HOLMES Compared to Other State-of-the-Art Approaches? 60

4.5.2 RQ 4.2: How Well Does HOLMES Generalize on Unseen Projects and Datasets? . . . 65
4.6 Discussion . . . 66
4.6.1 Why HOLMES Outperforms Other State-of-the-art clone detectors. . . . 66
4.6.2 Representation learning using Graph Attention networks (GATs). . . . 66
4.6.3 Qualitative Analysis of the Features Learned by EA-HOLMES. . . . 67
4.6.4 Ablation Study. . . . 68
4.6.5 Limitations of HOLMES. . . . 70
4.7 Chapter Summary . . . 70

5 Improving Cross-Language Code Clone Detection via Code Representation Learning and Graph Neural Networks 72
5.1 Introduction . . . 72
5.2 Motivation . . . 75
5.2.1 Motivating Example . . . 75
5.2.2 Key Ideas . . . 78
5.3 Approach Overview . . . 78
5.3.1 Flow-Enriched ASTs (FE-ASTs) Generation and Vector Formulation . 79
5.3.2 FE-AST Representation Learning Module . . . 80
5.3.3 Clone Detection Module . . . 84
5.3.4 Implementation Details . . . 84
5.4 Experimental Design . . . 86
5.4.1 Datasets . . . 86
5.4.2 Research Questions . . . 89
5.4.3 Baseline Evaluation . . . 91
5.5 Results . . . 94
5.5.1 RQ 5.1: How Does the Performance of RUBHUS Compare in the Cross-Language Context Between Statically and Dynamically Typed Languages, and How Well Does It Generalize on Unseen Projects? . . . 95
5.5.2 RQ 5.2: How Do State-of-the-Art Learning-Based Single-Language Code Clone Detection Tools Perform on Cross-Language Code Clone Detection Datasets? . . . 101

5.5.3 RQ 5.3: How Effectively Can RUBHUS Detect Clones in Statically Typed Languages? . . . 103
5.5.4 RQ 5.4: How Effectively Can RUBHUS Detect Clones in Dynamically Typed Languages? . . . 106
5.5.5 Performance Analysis of RUBHUS: Model Size, Dataset Processing, and Training/Evaluation Time. . . . 107
5.6 Discussion . . . 108
5.6.1 FE-ASTs for Representation Learning and Similarity Prediction . . . 109
5.6.2 Role of GNNs and Semi-Supervised Learning . . . 109
5.6.3 Qualitative Analysis of the Learned Feature Space . . . 110
5.6.4 Ablation Study . . . 111
5.6.5 Limitations . . . 112
5.7 Chapter Summary . . . 112

6 Improving Semantic Code Clones Detection with Adversarial Domain Learning 114

6.1 Introduction . . . 114
6.2 Approach Overview . . . 118
6.3 Experiment Design . . . 125
6.3.1 Datasets . . . 125
6.3.2 Baseline Comparisons . . . 129
6.3.3 Research Questions . . . 132
6.3.4 Implementation Details . . . 134
6.4 Results . . . 136
6.4.1 RQ 6.1: How does CLODIA impacts the performance of supervised clone detection tools on unseen targets domains? . . . 136
6.4.2 RQ 6.2: How does CLODIA compares with other domain adaptation techniques for improving the performance of supervised code clone detection techniques on unseen targets domains? . . . 140
6.4.3 Qualitative analysis of the learned feature space . . . 143
6.4.4 Statistical Tests . . . 144
6.5 Chapter Summary . . . 144

7 Conclusion and Future Work 146

7.1 Conclusion 146
7.2 Future Work 149
7.2.1 Development of Automated Tools for Code Clone Management and Refactoring 149
7.2.2 Fine-Grained Clone Detection 150
7.2.3 Exploring the Potential of Large Language Models for Detecting Semantic Clones 151
References **154**

List of Tables

4.1 Dataset Statistics. 57

4.2 Percentage of clone-types in BigCloneBench. 57

4.3 Comparative evaluation with GGNN and TBCCD variants. 61

4.4 F1 value comparison w.r.t various clones types in BigCloneBench dataset. . . . 61

4.5 Performance on GCJ* and SeSaMe dataset. 65

4.6 F1 value comparison w.r.t various clones types in BCB*. 65

5.1 Cross-language code clones datasets. 86

5.2 Single-language semantic clones datasets. 88

5.3 % clone-types in BigCloneBench. 88

5.4 Hyperparameter configurations for deep learning-based tools used in the experiments. 92

5.5 Hyperparameter settings for traditional code clone detection tools used in the experiments. 93

5.6 Classification performance of RUBHUS and other baselines (in %) in the cross-language setting. Empty cells in the tables indicate that the tools do not provide support for C language. 95

5.7 Classification performance of single-language code clone detection tools on cross-language code clones datasets. 102

5.8 Classification performance of RUBHUS and other baselines on statically-typed languages in single-language setting. Empty cells in the table indicate that the tools do not provide support for C language. 103

5.9 Classification performance of RUBHUS and other baselines on unseen statically-typed single-language code clones datasets. Empty cells in the tables indicate that the tools do not provide support for C language. 105

5.10 Classification performance of RUBHUS and other baselines on dynamically-typed language in single-language setting. 107

6.1 Dataset Statistics & Percentage of clone-types in BCB. 125

6.2 Performance evaluation of code clone detection baselines in supervised setting. 135

6.3 Performance evaluation of code clone detection baselines on GCJ, BCB and AC datasets without domain adaptation. 136

6.4 Robustness analysis of domain adaptation techniques on GCJ, BCB and AC datasets for code clone detection tools. 136

List of Figures

1.1 An example of semantic code clones that sort an array of natural numbers by randomly shuffling the array n times. 2

1.2 Annual Growth in Code Clone-Related Research. 3

1.3 Evolution of code clone detection techniques from text-based beginnings to AI-powered techniques. 4

1.4 An example of cross language code clone snippets that implements fibonacci in Java and JavaScript. 6

1.5 Java code clone pair computing the SHA-256 hash of a string using built-in "MessageDigest" class in Listing 1.5 and using the "Apache Commons Codec" library in Listing 1.6. 8

1.6 Java code clone pair from the BCB dataset computing the MD5 hash of a string. 9

1.7 Summary of Thesis Contributions.First, we enhance semantic clone detection using PDGs and GNNs in HOLMES. This overcomes the limitations associated with traditional syntactic feature-based tools. Second, we introduce RUBHUS, a cross-language clone detection tool, tailored to meet the demands of modern software systems. Third, we propose CLODIA, based on domain adaptation to enhance the performance and generalization capabilities of learning-based code clone detection tools, particularly when working with unseen datasets. 10

2.1 Different clone types of gcd. 16

2.2 Simplified AST of Listing 2.6; blue rounded box are non-terminal AST syntax nodes; red rectangular boxes are terminal AST leaf nodes representing variables, literals, etc. 18

2.3 Simplified PDG of Listing 2.6; blue rounded box are PDG nodes; black solid lines denotes control dependence edges; red dashed edges denotes data dependence edges. 18

2.4 Simplified Flow-enriched AST for Listing 2.6 with black edges denoting ast edges; with orange edges denoting control flow; with purple dashed edges denoting last read relation; with green dashed edges denoting last write relation; with yellow dashed edges denoting compute from relation. 19

2.5 A general framework of 3-layer spectral graph convolutional neural network. . 24

2.6 Visual illustration of the graph neural network framework. (a) Illustration of the tth layer of the graph neural network. The feature vectors h_b^{t-1}, h_d^{t-1} from the neighboring nodes of A are *aggregated* and *combined* with h_a^{t-1}, the features of node A from the $t-1^{th}$ layer. This constitutes the representation of node A at the tth layer. (b) Illustration showing multiple rounds of propagation in a graph neural network. At the nth propagation round, a node receives information from each of its neighbors that are n hops away. For example, node A at propagation round 1 receives messages from its one-hop neighbors D and B. At propagation round two, it receives information from its two-hop neighbor, i.e., node C, and so on. 25

2.7 Domain adaptation aims to overcome the challenge of domain gap, where the source and target domains have different feature distributions. This can lead to poor performance of machine learning models when they are applied to the target domain. Domain adaptation techniques, such as feature alignment and adversarial training, can help align the feature distributions of the source and target domains, making it possible to transfer knowledge from the source to the target domain. This image illustrates the concept of domain adaptation, where the model learns to generalize across domains by aligning the feature distributions of the source and target domains, thus improving the model's performance in the target domain. 27

4.1 The proposed Siamese deep neural network consists of two identical sub-networks. The input to the sub-network is the pair of Java methods represented as PDGs. Each sub-network incorporates an attention-based graph neural network model to learn PDGs node features, which are then aggregated using soft attention to constitute a graph-level representation of the given input method. The proposed network is trained to learn the similarity between two feature vectors h1 and h2. Horizontal blue arrows denotes weight shared sub-networks. 43

4.2 A semantic code clone example detected by HOLMES, which was reported as false negative by TBCCD. The code in Listings 4.1 and 4.2 sort an array of natural numbers by randomly shuffling the array n times. 45

4.3 AST for Listing 4.1. 45

4.4 AST for Listing 4.2. 45

4.5 PDG for Listing 4.1. 46

4.6 PDG for Listing 4.2. 46

4.7 The architecture of one branch of the Siamese neural network is shown in Figure 4.1. (a) Our model first parses the given Java methods in the datasets to build PDGs. Node feature matrix and graph adjacency matrix are extracted from the source code. HOLMES then passes this as input to a multi-head masked linear attention module, which learns the importance of different sized neighborhood for a node. (b)The attention module outputs the set of learned node features that are then passed through an LSTM, which extracts and filters the features aggregated from different hop neighbors. (c)The learned node features are then passed to a graph pooling module. Graph pooling employs a soft attention mechanism to downsample the nodes and to generate a coarsened graph representation. 48

4.8 A synthetic example showing explored receptive path (area covered by the dotted red line) for the target node. The edge thickness denotes the received attention scores while learning features for the target node. Control and data-dependent edges are shown through solid and dotted edges, respectively. 52

4.9 Visualization of the 4-layer architecture of JK networks in HOLMES. In each propagation round t, the feature vector of node v is combined with its tth order neighbors. During the final layer (4th propagation round), all hidden feature vectors from previous rounds are concatenated to form the ultimate hidden representation for node v. This concatenation preserves features learned from n^{th} hop neighbors across various propagation rounds, ensuring their presence and impact in the final hidden representation of node v. 53

4.10 A WT3/T4 clone example from BCB dataset. The code snippets are implementing the functionality for copying the directory and its content. Although the snippets are reported under WT3/T4 clone category they are syntactically similar with some differences in the sequence of invoked methods and API calls. 62

4.11 ROC curve of TBCCD, EU-HOLMES, EA-HOLMES on GCJ dataset (AUC values are rounded up to 2 decimal places). 63

4.12 ROC curve of TBCCD, EU-HOLMES, EA-HOLMES on BCB dataset (AUC values are rounded up to 2 decimal places). 63

4.13 Time performance analysis on the GCJ dataset. 64

4.14 t-SNE plot of graph embeddings of clone and non-clone pairs of GCJ dataset generated by EA-HOLMES. 67

4.15 Qualitative analysis of the features learned by EA-HOLMES. 68

4.16 The effect on the performance of EA-HOLMES after varying the number of attention heads in both the attention blocks and after removing the LSTM layer. AB in the legend stands for Attention Block, for instance, "AB1, 1 Head" corresponds to "Attention Block 1 and 1 attention head". 69

4.17 The effect on the performance of EA-HOLMES after removing Jumping Knowledge (JK) nets and soft attention mechanism from the graph readout layer. . . . 69

5.1 A cross-language Java-Python clone example detected by RUBHUS, which is reported as false negative by GRAPHCODEBERT. 75
5.2 AST for lines $8 - 10$ of Listing 5.1. 76
5.3 AST for lines $6 - 9$ of Listing 5.2. 76
5.4 FE-AST for lines $8 - 10$ of Listing 5.1. 76
5.5 FE-AST for lines $6 - 9$ of Listing 5.2. 76
5.6 RUBHUS overview. 78
5.7 FE-AST for Lines $9 - 10$ in Listing 5.1. 79
5.8 Illustration of unsupervised graph embedding learning in RUBHUS. 82
5.9 A true clone pair reported true by RUBHUS but false by PCCD, C4, and GRAPHCODEBERT. 97
5.10 A true clone pair reported false by RUBHUS but true by XCODE and C4. . . . 98
5.11 A true clone pair reported false by RUBHUS and all other baselines. 99
5.12 ROC curve of RUBHUS and other baselines on AtCoder dataset. (AUC values are rounded up to 2 decimal places.) 100
5.13 ROC curve of RUBHUS and other baselines on CodeChef dataset. (AUC values are rounded up to 2 decimal places.) 100
5.14 A semantically similar code clone example from the BCB dataset implementing recursive file/directory deletion. 104
5.15 Training time analysis on the AtCoder and CodeChef datasets. 108
5.16 Evaluation time analysis on the AtCoder and CodeChef datasets. 108
5.17 Cliff's delta (within) & 95% Confidence Interval (2-tailed) 109
5.18 t-SNE plot of graph embeddings of clone and non-clone pairs of RUBHUS on the AtCoder dataset. 111

6.1 A semantic clone pair example from Apache Dubbo detected by applying CLODIA on HOLMES and TBCCD which otherwise was detected as false negative by both the tools. 115
6.2 A semantic clone pair example from BCB dataset detected as true positive by HOLMES and TBCCD. 115
6.3 The training and testing phase of CLODIA. 119
6.4 A true positive clone pair from AC dataset detected as true positive by TBCCD, FA-AST, and HOLMES after applying CLODIA. 139
6.5 A true negative clone pair from AC dataset detected as false positive by FA-AST and TBCCD which otherwise was detected as true negative by HOLMES. 141

6.6 t-SNE visualisations of samples from GCJ_{sn} and GCJ_{un} by different DA methods. 143

7.1 A true negative clone pair detected as true positive GPT-3.5. 151

Chapter 1

Introduction

In the grand symphony of software development, each line of code plays a critical role in harmonizing functionality and design. Yet, within this intricate composition, a subtle but pervasive phenomenon often goes unnoticed—the emergence of code clones. These are not the doppelgängers of science fiction; rather, they are repeated patterns of code that resonate throughout our software, a natural byproduct of developers seeking efficiency and familiarity in their craft.

The existence of code clones is a testament to the complexity and creativity of human problem-solving—where once a solution is found, it is replicated, adapted, and embedded within the fabric of our digital constructs. Yet, this replication carries with it a silent weight: the burden of maintenance, the specter of technical debt, and the potential for errors that can ripple out with unforeseen consequences. Understanding the role of code clones is not just an academic exercise; it is a crucial aspect of building sustainable and resilient software systems that stand the test of time and usage.

In this chapter, we motivate the reader towards the need for detection of code clones. We provide a concise yet comprehensive overview of the existing techniques in code clone detection, shedding light on the existing challenges within this domain. It is within this context that we introduce our proposed contributions, which seek to address these challenges and provide solutions tailored to the demands of modern software systems.

```java
public static void main(String[] args){
  Scanner in = new Scanner(System.in) ;
  int T = in.nextInt() ;
  int[] a = new int[T] ;
  for (int j = 0 ; j < T ; j ++)
    a[j] = in.nextInt() ;
  int c = 0 ;
  for (int j = 0 ; j < T ; j ++)
    if (a[j] == j+1)
      c ++ ;
  System.out.println("Case
  ↪  #"+i+":"+((double)T-(double)c));
}
```

Listing 1.1: Program1.java

```java
public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  int n = in.nextInt(),t=0;
  float count = 0.0f;
  while(n>0){
    if(++t! = in.nextInt())
      count++;
    n--;
  }
  System.out.printf("Case#%d:%.6f%n", i,
  ↪  count);
}
```

Listing 1.2: Program2.java

Figure 1.1: An example of semantic code clones that sort an array of natural numbers by randomly shuffling the array n times.

1.1 The Imperative for Code Clone Detection

Code clones are code fragments that are similar according to some definitions of similarity [1]. There are two types of similarity defined between code snippets: 1) **Syntactic similarity**: Two code snippets are considered syntactically similar if their program syntax or structure is alike, and 2) **Semantic similarity**: Two code snippets are deemed semantically similar if they possess similar functionality or if one code snippet encompasses the functionality of another [2]. Figure 1.1 shows an example of semantic clones that sorts an array of natural numbers. However, identifying them as clones can be challenging due to variations in variable names, coding style, and code structure.

The rise of open-source repository platforms, such as GitHub and BitBucket, has significantly transformed the way source code is developed, maintained, and shared. These platforms have enabled the effortless exchange and reuse of code, accelerating the software development process through extensive "copy-paste" practices [3]. Consequently, code cloning has become a widespread phenomenon in software projects. Numerous research studies on software cloning have underscored the substantial occurrence of clones in software systems [4, 3, 5, 6, 7, 8, 9, 10]. Although code cloning can expedite development and improve code comprehension [11, 12, 13, 14, 15, 16, 17, 18, 19], its influence on software maintenance is intricate and may hinder software evolution [20, 21, 22, 23, 24, 1, 25, 26, 27, 15]. Therefore, developing effective and efficient techniques for detecting both syntactic and semantic code clones is essential for ensuring the

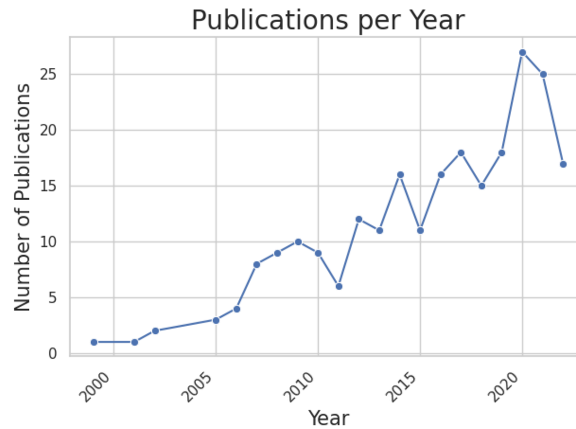


Figure 1.2: Annual Growth in Code Clone-Related Research.

integrity and maintainability of software systems.

The title of this thesis "DECODING THE DNA OF CODE: AN AI-INFUSED APPROACH TO DETECT CODE CLONING IN SOFTWARE SYSTEMS" metaphorically refers to examining the fundamental components and structures that form program ingredients, similar to DNA, which forms the essential ingredients of living organisms and determines their characteristics. In this context, "DNA of code" signifies the elements, patterns, and structures within a program that define its behavior and functionality.

This thesis proposes to use AI-infused techniques to understand these fundamental aspects of code and identify instances of code cloning. Thus, by "decoding the DNA" of code, the approach aims to uncover and analyze the intricate details of code to detect when and where such cloning occurs, thereby aiding in the maintenance and quality assurance of software systems.

1.2 A Comprehensive Look at Code Clone Detection

The field of code clone detection has seen considerable growth and innovation, as depicted in Figure 1.2, which shows the yearly increase in related academic research.[1] This figure underscores the field's dynamism and its significance in tackling code clone challenges in

[1]The data for Figure 1.2 is collected from DBLP between 1999 and 2023 using specific keywords such as "code clone detection," "code similarity detection," and related terms. The search results were filtered to eliminate duplicates before generating the graph.

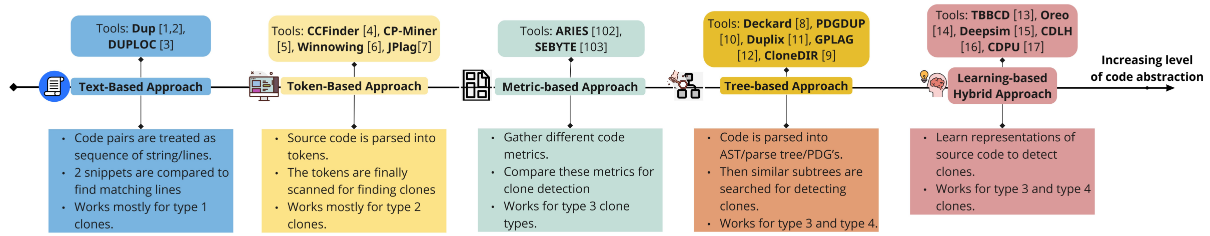


Figure 1.3: Evolution of code clone detection techniques from text-based beginnings to AI-powered techniques.

modern software systems. Figure 1.3 further illustrates the shift from traditional text-based methodologies to advanced AI-powered approaches, demonstrating the field's adaptability and innovation. In this section, we will explore these evolving techniques, their foundations, and their advancements over time.

Early clone detection strategies mainly targeted syntactic clones using a blend of program, lexical, and token analysis techniques [1, 28, 29]. These primarily focused on detecting Type 1-3 code clones, using various program representations [29, 30, 22, 31, 32]. They depended on manually crafted lexical and syntactic features to identify similar code pairs.

Early efforts were also made to measure semantic similarity using program analysis. For instance, [33, 34] proposed slicing and subgraph isomorphism over program dependency graphs (PDGs) to identify Type-4 clones. Some techniques also compared programs' runtime behavior [35] or program memory states [28] to identify code clones. There was also a growing interest in using data mining approaches to learn code similarity, as exemplified by techniques like those using latent semantic indexing [36].

Metric-based clone detection techniques [37, 38, 39, 40, 41, 42] have also contributed to early clone detection research, quantifying code aspects using predefined metrics and comparing these values to identify potential clones. While useful and scalable, these methods have limitations including sensitivity to metric and threshold choice, reliance on human-defined heuristics, and potential incompleteness in capturing all syntactic and semantic properties.

The growing interest in applying deep learning to software engineering led to its use in code clone detection [43, 44, 45, 46, 47, 48, 49, 50, 51]. These techniques transformed code into an intermediate representation like token sequences or syntax trees, which deep learning models

processed to learn high-level program features. These features were compared using similarity metrics, greatly improving the performance of semantic and syntactic clone detection.

As software development evolved, so did the challenges in code clone detection. Cross-language clone detection emerged to adapt to modern codebases, using common intermediate languages or deep learning techniques to measure code similarity across different languages. Binary clone detection, crucial for tasks like reverse engineering and software security analysis, focuses on detecting code clones in compiled code, often requiring the disassembly of binary code to create a lower-level representation for comparison.

1.3 Identified Challenges and Proposed Solutions

While clone detection has been a vibrant research area in software engineering for over a decade, the majority of techniques have centered on detecting syntactic clones within a single programming language. These techniques encompass a broad spectrum, ranging from traditional program analysis-based methods to more recent machine learning-based approaches. However, while syntactic clones are indispensable, capturing program semantics holds equal significance for measuring code functional similarity. Despite the importance of semantic clones, a notable gap persists in semantic code clone detection. Hence, the primary objective of this thesis is to address this gap by advancing semantic code clone detection research.

The existing learning-based approaches for semantic code clone detection make use of syntactic program representation like code tokens, ASTs, etc. Notwithstanding this, we argue that *these program representations do not capture program semantics even though it might be crucial for measuring code functional similarity*. Thus, a more sophisticated program representation is required to learn the functional behaviors of the source code. Furthermore, the limited number of techniques that attempt to incorporate program semantics, such as those based on program dependence graphs (PDGs) and graph isomorphism, face challenges related to precision and scalability. The inherent complexity of graph isomorphism and the approximations made when mapping PDG sub-graphs to ASTs, as observed in [52], pose practical limitations.

```java
public static int[] Fibonacci(int n) {
  int[] fib = new int[n];
  fib[0] = 0;
  fib[1] = 1;
  for (int i = 2; i < n; i++)
    fib[i] = fib[i - 1] + fib[i - 2];
  return fib;
}
```

Listing 1.3: Program3.java

```javascript
function generateFibonacci(n) {
  let fib = [0, 1];
  for (let i = 2; i < n; i++) {
  fib[i] = fib[i - 1] + fib[i - 2];
  }
  return fib;
}
```

Listing 1.4: Program4.js

Figure 1.4: An example of cross language code clone snippets that implements fibonacci in Java and JavaScript.

Recognizing these challenges as a foundational step in our research, we introduce a new tool called HOLMES, designed to address these issues and identify semantic code clones. HOLMES is built on two key insights. First, **feature learning plays a significant role in measuring code similarity** [46]. Thus, learned features should contain semantic information, specifically control and data dependence information, rather than structural information, such as lexical elements and high-level program constructs captured by ASTs. Code similarity based solely on syntactic features is too restrictive and rigid in its expression compared to its more powerful and expressive notion based on program semantics or functionality. Hence, HOLMES uses the control and data dependence information from PDGs as a basis of similarity metrics. Second, **to capture program semantics efficiently, one must capitalize on PDGs graphical structure.** Therefore, HOLMES employs a graph-based deep neural network to learn program representation.

The software development landscape is changing as modern software systems often integrate multiple programming languages, with each language encoding specific aspects of the overall system. Developers are now required to work with and code in multiple languages, and for the same, they adopt a *cross-language code reuse learning* strategy [53, 54]. Moreover, with the increasing popularity of cross-platform applications, similar functions and ever-evolving APIs are needed to be replicated in different languages. Propagating evolving code across different platforms can be automated efficiently with the detection of cross-language clones, thereby saving time and increasing efficiency. Moreover, the ability to look for similar code snippets across different programming languages can also be leveraged by code migration tools and transpilers.

Cross-language code snippets often lack significant similarity, which poses challenges for

traditional single-language code clone detection tools. Consider the examples in Listings 1.3 and 1.4, which constitute a cross-language clone pair for generating Fibonacci sequences. Although both Java and JavaScript code snippets employ the same algorithm, their stark differences in syntax, language-specific features, and execution environments are immediately evident. For instance, Java's static typing requires variable type declarations, whereas JavaScript allows the use of "let" or "const".

Adapting single-language code clone detection tools techniques for cross-language code clone detection tools is theoretically possible by modifying parsers or scripts for the target languages. However, this approach is inherently limited due to the unique challenges posed by cross-language code clone detection tools. These challenges stem from differences in syntactic and semantic features, as well as variations in programming paradigms across different languages. Consequently, single-language code clone detection tools techniques are ought to be less effective in detecting cross-language code clones.

Hence, to meet the evolving needs of contemporary software systems, such as bug detection [55] and program refactoring [56], in multi-language environments, there is a need to enhance traditional clone detection capabilities to encompass multiple languages. Cross-language code clone detection techniques must strike a delicate balance between syntax, semantics, and program structure to effectively identify similarities.

There have been some techniques that detect cross-language clones while leveraging some syntactic information or transforming the code snippets to some common intermediate language like .NET framework to measure similarity [57]. However, as the number of programming languages without a common intermediate representation has increased, these techniques have faced limitations. Historical approaches have included language-agnostic methods using revision histories [58] or software quality metrics (e.g., cyclomatic complexity, number of loops) [59], but these methods do not make use of both semantic and syntactic information when measuring similarity. Additionally, learning-based techniques like those employing conventional neural networks such as LSTMs often overlook structured program information in their assessments of code similarity.

```java
public static String sha256(String in){
  MessageDigest dig =
  ↪ MessageDigest.getInstance("SHA-256");
  byte[] Bytes =
  ↪ in.getBytes(StandardCharsets.UTF_8);
  byte[] hash = dig.digest(Bytes);
  StringBuilder hex = new
  ↪ StringBuilder(2*hash.length);
  for (byte b:hash){
    String hex = Integer.toHexString(0xff
    ↪ & b);
    if (hex.length() == 1)
      hex.append('0');
    hex.append(hex);
  }
  return hex.toString();
}
```

Listing 1.5: *Program5.java*

```java
import org.apache.commons.codec.digest.*;
import DigestUtils;
public class SHA256Example {
  public static void main(String[] args) {
    String input = "Hello, SHA-256!";
    String sha256Hash =
    ↪ DigestUtils.sha256Hex(input);
    System.out.println("Input: " + input);
    System.out.println("SHA-256 Hash: " +
    ↪ sha256Hash);
  }
}
```

Listing 1.6: *Program6.java*

Figure 1.5: Java code clone pair computing the SHA-256 hash of a string using built-in "MessageDigest" class in Listing 1.5 and using the "Apache Commons Codec" library in Listing 1.6.

Thus, as our second contribution in this thesis, we identified these challenges, and we propose a semi-supervised learning-based cross-language code clone detection tool, RUBHUS. This tool utilizes Abstract Syntax Trees (ASTs) enriched with semantic data extracted from the source code, and employs GNNs to model code similarity. RUBHUS extracts features from code ASTs and employs GNNs to learn high-level program representations. In particular, to enhance learned local AST node representations and capture various global latent structures and sub-structures, we employ unsupervised learning based on Mutual Information (MI) maximization. This approach enables us to encode different data aspects present in various AST sub-structures, which might be overlooked otherwise. Importantly, RUBHUS is designed to work effectively with both statically and dynamically typed programming languages.

While learning-based techniques hold promise for improved performance, they also introduce several challenges. One of the primary obstacles is the issue of domain shift, which arises from the disparity between the data distribution in the training set and the real-world deployment environment. This mismatch degrades the performance of these models in real-world datasets, making them practically unusable.

For example, consider the code listing pairs in Figures 1.5 and 1.6, which implement the SHA-256 and MD5 hashing algorithms. While both code pairs achieve the same goal of hashing data, they employ different hashing techniques. However, when using clone detection tools, a

```java
private static final String ALGO = "MD5";
public static void main(String[] args) {
  MessageDigest md = null;
  try{
    md = MessageDigest.getInstance(ALGO);
  }catch (NoSuchAlgorithmException e) {
    System.out.println(e);
    e.printStackTrace();
  }
  md.update((byte) 1);
  md.update((byte) 2);
  md.update((byte) 3);
  byte[] hash = sha.digest();
  System.out.println("Length of digest: " +
  ↪  hash.length);
}
```

Listing 1.7: Program7.java

```java
import java.security.*;
public static byte[]
↪  digestPassword(String pwd) {
byte[] b = pwd.getBytes();
try {
  MessageDigest digest;
  digest =
  ↪  MessageDigest.getInstance("MD5");
  synchronized (digest){
    digest.reset();
    b = digest.digest(pwd.getBytes());
}
} catch (NoSuchAlgorithmException nsa)
↪  {}
return b;
}
```

Listing 1.8: Program8.java

Figure 1.6: Java code clone pair from the BCB dataset computing the MD5 hash of a string.

potential issue may arise during testing. Suppose the tool has been trained on numerous hashing algorithm implementations including MD5 but not on SHA. In that case, it might mistakenly label code pairs in Listings 1.5 and 1.6 as false negatives.

The reason for this misclassification is that, even though both code pairs belong to the same class of "clones", they exhibit substantial differences. These differences include variations in coding styles, syntax, semantics, data distributions, and potentially different datasets. This diversity between training and testing data can lead to reduced performance on unseen datasets.

The decrease in performance of supervised techniques on new and unseen data can be primarily attributed to the constraints of the training dataset. While the training dataset may be substantial in size, it often fails to capture the full range of coding styles, libraries, and frameworks present in real-world projects.

For instance, consider the BigCloneBench (BCB) dataset, which contains around 5,000 pairs of code snippets, both clones and non-clones, implementing secure hash functions like MD5. Despite the dataset's size and scope, when a model trained on the BCB dataset encounters a code pair like the one shown in Figure 1.6, it may erroneously classify them as non-clones. This misclassification occurs because Listings 1.5 and 1.6 exhibit significant syntactic, and coding style differences from the Listings shown in 1.7 and 1.8.

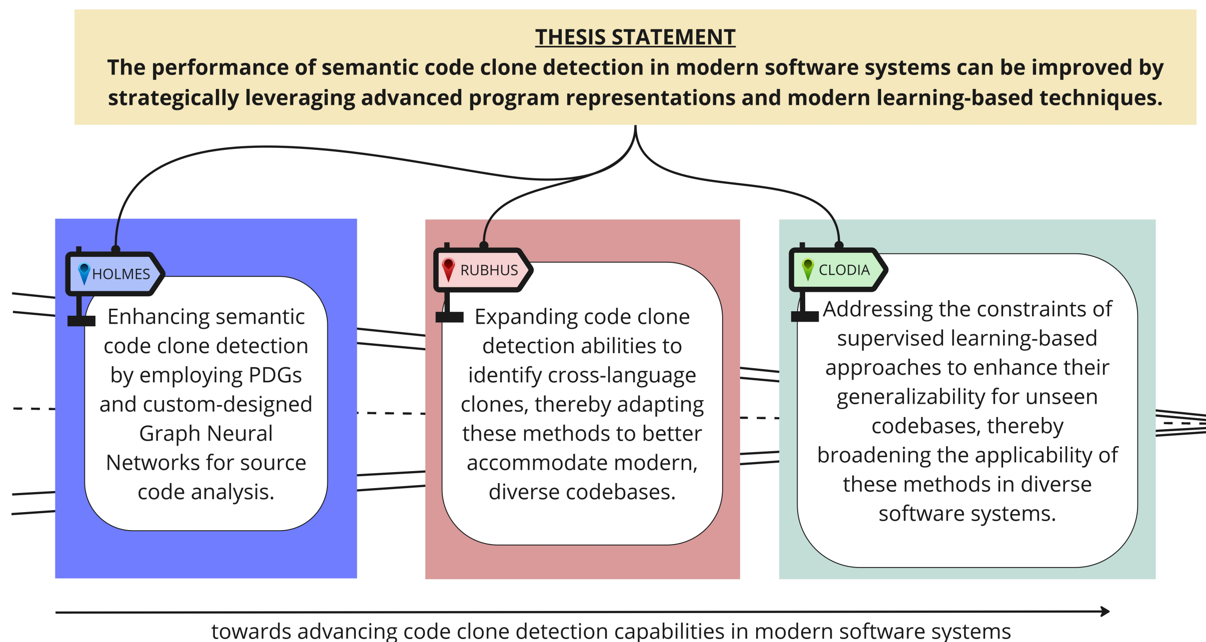


Figure 1.7: Summary of Thesis Contributions.First, we enhance semantic clone detection using PDGs and GNNs in HOLMES. This overcomes the limitations associated with traditional syntactic feature-based tools. Second, we introduce RUBHUS, a cross-language clone detection tool, tailored to meet the demands of modern software systems. Third, we propose CLODIA, based on domain adaptation to enhance the performance and generalization capabilities of learning-based code clone detection tools, particularly when working with unseen datasets.

Moreover, when dealing with new and previously unseen data, there is a possibility that this data introduces novel patterns or features that were not adequately covered in the original training set. To adapt these techniques for handling such unseen data effectively, re-training the models becomes a necessary step.

However, creating labeled datasets for code clones in every new domain can be a formidable challenge, as it is a labor-intensive and subjective task. These constraints restrict the practicality of supervised learning techniques in real-world scenarios.

Thus, as our third contribution to this thesis, we introduce an innovative approach and tool, CLODIA, focused on enhancing the performance and generalization capabilities of supervised clone detection tools when faced with unseen datasets. Adversarial Unsupervised Domain Adaptation (UDA) [60] serves as the foundation for this technique. UDA is a machine learning method that empowers pre-trained neural models to adapt seamlessly to new repositories without the requirement of re-training the model on the labeled target domain data. UDA techniques commonly incorporate an adversarial discriminator, tasked with distinguishing between source and

target domain data based on their feature representations. This approach serves a dual purpose: first, it aligns these feature representations by identifying common patterns amid their diversity, and second, it fosters uncertainty regarding the origin of a given feature vector—whether it originates from the source or target domain.

However, applying existing UDA methods in clone detection poses a significant challenge due to the inherent disparities between source and target datasets. In response to these issues, CLODIA proposes a novel approach focused on learning multiple latent representation spaces. These spaces retain discriminative power for both the (labeled) source and (unlabeled) target domains, while also ensuring that representations for the two domains remain well-separated. Figure 1.7 summarizes the contributions of this thesis.

1.4 Thesis Statement

This, dissertation confirms the following thesis statement:

Thesis Statement

The performance of semantic code clone detection in modern software systems can be improved by strategically leveraging advanced program representations and modern learning-based techniques.

More precisely,

TS-1 The adept utilization of advanced program representations and cutting-edge learning-based techniques can enhance the performance of semantic code clone detection in contemporary software systems. In this thesis, we emphasize the importance of feature learning and the selection of neural network architectures that effectively exploit code semantics and structural information. Through these means, we aim to augment our ability to measure code similarity and uncover deeper insights into code clone relationships within the software landscape.

TS-2 Transfer learning techniques, with a specific emphasis on domain adaptation, can enhance the performance and improve the generalizability of supervised clone detection techniques.

1.5 Contribution

The contributions of this dissertation are:

1. First, we propose an approach to improve semantic clone detection techniques using program dependence graphs and graph neural networks. The major contributions are the following:
    - A new code representation for semantic code clone detection. To the best of our knowledge, our work is the first to learn code representation for code clone detection in two different manners: i) Using control and data dependence relations between the program statements to model code functional dependency ii) Treating control and data dependence edges differently to give respective importance to syntactic and semantic information while learning code representation for a code snippet.
    - We proposed a new deep learning architecture for graph similarity learning. Our approach jointly learns the graph representation and graph matching function for computing graph similarity. In particular, we have used an attention-based Siamese graph neural network to detect semantic clones. Our approach uses the control-dependent and data-dependent edges of PDGs to model the program's semantic and syntactic features. We have used attention to give higher weights to semantically relevant paths. The learned latent features are then used to measure code functional similarity.
    - We created a large and representative dataset of clones containing approximately 1M code pairs of clones and non-clones. We developed a prototype tool HOLMES and evaluated it on our dataset and other popular benchmarks for clones against state-of-the-art-tool TBCCD [61]. Through a series of empirical evaluations, our results show that HOLMES outperforms TBCCD by a significant margin of $\sim 24\%$.
2. Second, to adapt clone detection techniques, to contemporary code bases with multiple languages, we propose a cross-language clone detection tool. The major contributions are the following:

- We propose a new unsupervised deep learning framework for learning high-level program features from control and data flow-enriched ASTs. Our approach learns structured semantic and syntactic information available with the flow-enriched ASTs using GNNs and the principle of mutual information maximization in an unsupervised way.
- We propose a new supervised deep learning architecture for modeling similarity between code snippets. Our work jointly learns the high-level program features and similarities between snippets using a learning-based technique.
- We create a dataset of cross-language clones containing around 40K C files and 15K Java files. The dataset also has annotation information on clones and non-clones pairs.
- We develop a prototype tool, RUBHUS, and evaluate it under a number of different experimental settings. The results of our experiments demonstrate that RUBHUS outperforms the state-of-the-art cross-language code clone detection tools and other baseline techniques.
- We also present the first study to assess the performance of single-language code clone detection tools in the context of cross-language code clone detection.

3. Third, to improve the generalizing capabilities of supervised clone detection techniques on unseen repositories and datasets, we propose an adversarial domain adaptation technique to reduce the domain shift and adapt the trained model from seen datasets to unseen settings. The major contributions are the following:

   - We propose a novel technique, CLODIA, which uses multiple latent spaces for domain adaptation for improving the generalizing capabilities and performance of supervised clone detection techniques in unseen domains and datasets.
   - Extensive evaluation of CLODIA on various datasets, including programming competition and open source datasets and also on a handcrafted dataset of clones curated from real-world software systems. We also compare and assess the effectiveness of various state-of-the-art UDA techniques against CLODIA in enhancing the generalization capabilities of state-of-the-art neural clone detection tools, utilizing standard evaluation metrics.

1.6 Outline of Dissertation

The rest of the dissertation is organized as follows. We give necessary background and related work on code clone detection in Chapter 2 and Chapter 3. Chapter 4 explains our proposed approach HOLMES to improve semantic code clone detection. While HOLMES proves effective in various scenarios, it encounters challenges when dealing with code repositories spanning multiple programming languages. Given the prevalent trend of utilizing multiple languages to build software systems, Chapter 5 introduces our second approach, RUBHUS, designed to address these challenges and extend code clone detection capabilities across languages. These learning-based techniques for clone detection have shown to outperform the traditional techniques by a significant margin, but their performance degrades when these techniques are applied on unseen data. Thus to improve the generalizing capabilities of these supervised learning-based clone detection techniques, Chapter 6 introduces an innovative adversarial domain adaptation technique, CLODIA. Finally, we conclude with Chapter 7, which synthesizes the findings and lays the groundwork for future research endeavors.

Chapter 2

Background

The world of software engineering is marked by an ever-increasing demand for innovative and efficient solutions. In this fast-paced landscape, code reusability and the identification of semantic code clones have emerged as pivotal concerns. Code clones, whether intentional or accidental, hold a dual role, providing opportunities for code reuse and posing intricate challenges for software evolution. Indeed, they are the two-edged sword of modern software development, offering both advantages and complications.

This chapter, lays the foundation for understanding the intricate landscape of semantic code clone detection in contemporary software systems. Before we delve into the methodologies and techniques that underpin our thesis's core research, we explain the essential preliminaries related to code clones, deep learning, and domain adaptation. Thus, it is not merely a prologue but an indispensable primer that equips you with the knowledge required to navigate the intricate intricacies and challenges we are about to address.

2.1 Code Clones

This dissertation follows the well-accepted definition and terminologies from [2]:

Code Fragment: A continuous segment of a code fragment is denoted by a triplet $\langle c, s, e \rangle$, where s and e are start and end lines respectively, and c is the code fragment.

```java
static int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
```

Listing 2.1: Original code snippet

```java
static int gcd1(int a, int b) {
    if (b == 0){
        return a;
    }
    return gcd1(b, a % b);
}
```

Listing 2.2: Type 1 clone

```java
public static int gcd2(int no1, int no2) {
    if (no2 == 0) {
        return 1;
    }
    return gcd2(no2, no1 % no2);
}
```

Listing 2.3: Type 2 clone

```java
public static int gcd3(int m, int n) {
    if (0 == n) {
        return m;
    } else {
        return gcd3(n, m % n);
    }
}
```

Listing 2.4: Type 3 clone

```java
static int gcd4(int a, int b) {
    while (b != 0) {
        int t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

Listing 2.5: Type 4 clone

Figure 2.1: Different clone types of gcd.

Code clones are pairs of similar code snippets existing in a source file or a software system. Researchers have broadly classified clones into following categories stretching from syntactic to semantic similarity:

- **Type-1 clones (textual similarity)**: Duplicate code snippets, except for variations in white space, comments, and layout.
- **Type-2 clones (lexical similarity)**: Syntactically identical code snippets, except for variations in the variable name, literal values, white space, formatting, and comments.
- **Type-3 clones (syntactic similarity)**: Syntactically similar code snippets that differ at the statement level. Code snippets have statements added, modified, or deleted with respect to each other.
- **Type-4 clones (semantic similarity)**: Syntactically different code snippets implementing the same functionality.
    1. **Cross-language clones**: cross-language clones are a subset of type-4 clones, denoting instances where identical or similar functionality found in code written in different programming languages, serving the same purpose but often requiring special handling due to language differences.

Figure 2.1 enumerates different clone types of original code snippet implementing greatest common divisor (GCD) functionality. The original code snippet (Listing 2.1) computes the GCD of two numbers. The Type-1 clone (Listing 2.2) of the original code snippet is identical except for the formatting variation. The Type-2 clone (Listing 2.3) has different identifier names (no1 and no2). Type-3 clone (Listing 2.4) of the original code snippet is syntactically similar but differs at the statement level. Finally, the Type-4 (Listing 2.5) clone of the original code snippet computes GCD using a completely different algorithm. There exists no syntactical similarity between the original snippet and its Type-4 clone.

Since there was no consensus on minimum similarity for Type-3 clones and it was difficult to separate Type 3 and Type-4 clones, the authors in [62] categorized Type-3 and Type-4 clones based on their syntactic similarity: Strongly Type-3, Moderately Type-3, and Weakly Type-3+4

```java
while (counter != 0){
  sum += counter;
  counter--;
}
```

Listing 2.6: Program.Java

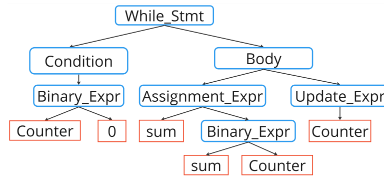


Figure 2.2: Simplified AST of Listing 2.6; blue rounded box are non-terminal AST syntax nodes; red rectangular boxes are terminal AST leaf nodes representing variables, literals, etc.

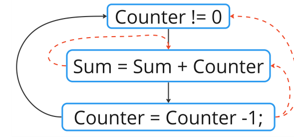


Figure 2.3: Simplified PDG of Listing 2.6; blue rounded box are PDG nodes; black solid lines denotes control dependence edges; red dashed edges denotes data dependence edges.

(Type-4) clones. Thus, Strongly Type-3 clones have at least 70% similarity at the statement level. These clone pairs are very similar and contain some statement-level differences. The clone pairs in the Moderately Type-3 category share at least half of their syntax but contain a significant amount of the statement-level differences. The Weakly Type-3+4 code clone category contains pairs that share less than 50% of their syntax.

Since the cross-language clones are syntactically different, they are not placed in Type-1 and Type-2 clone categories. However, cross-language clones may belong to the Moderately Type-3 category for syntactically similar languages like Java and C# where the differences may be at statement level only. Still, in many other cross-language setups like Java and Python, the cross-language clones belong to the Weakly Type-3+4 category where the clones are syntactically very different (also at the statement level) and have the same functionality [63].

2.2 Program Representations

2.2.1 Abstract Syntax Trees

An Abstract Syntax Tree (AST) is a hierarchical and tree-like data structure designed for representing the syntactic structure of source code. The nodes in the AST correspond to language constructs such as statements, expressions, and declarations and the tree structure reflects the syntactic relationships between different elements in the code. Figure 2.2 shows a simplified AST of Listing 2.6.

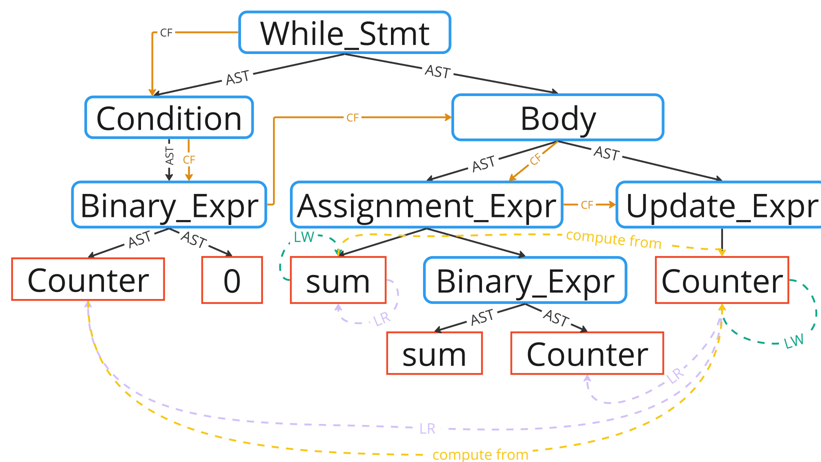


Figure 2.4: Simplified Flow-enriched AST for Listing 2.6 with black edges denoting ast edges; with orange edges denoting control flow; with purple dashed edges denoting last read relation; with green dashed edges denoting last write relation; with yellow dashed edges denoting compute from relation.

ASTs are language-dependent, uniquely tailored to the grammar and syntactic rules of a given programming language. AST captures the scope and nesting of constructs. They also provide a clean and concise representation of the code, discarding extraneous details like whitespace and comments, thus making them suitable for understanding the program's block structure and scoping rules.

AST includes information about data types, variables, and expressions and serves as a structural blueprint of code. They primarily represents the program's static structure and doesn't capture dynamic aspects like execution flow or data dependencies.

2.2.1.1 Flow-enriched ASTs

Flow-enriched ASTs (FE-AST) are advanced representations that go beyond traditional ASTs by including comprehensive information about both control flow and data flow within a program. An example of FE-AST is shown in Figure 2.4.

To comprehensively capture both control and data flow within a program represented by an AST, we enhance the AST by incorporating edges from the program's control flow graph. For data flow, we introduce additional edges that establish various connections between distinct instances of AST nodes.

For each token in the AST, we identify the set of nodes where a variable could have been last used. This set may encompass multiple nodes, such as instances where a variable is utilized after a conditional statement with branches, and may extend to syntax tokens that follow in the program, as observed in loops. This relationship forms the basis for the *last read* edge. Similarly, we determine nodes where the variable could have been last written, forming the *last write* edge.

Moreover, when encountering an assignment expression, we establish a *ComputeFrom* edge, connecting the variables present in the expression to the left-hand side (LHS) variable token. This relation clarifies the dependencies in the assignment expression.

FE-ASTs thus captures how the program's logic unfolds through control structures like conditional statements, loops, etc. offering a detailed insight into the order of execution and branching conditions. Moreover, they also encompass data flow information, highlighting how data variables and values propagate through the program. This includes variable assignments, data dependencies, and the flow of values between different parts of the code.

2.2.2 Program Dependence Graphs

Program Dependence Graphs (PDGs) are directed attributed graph that explicitly encode the program's control, and data dependence information [64]. Figure 2.3 shows a simplified PDG of Java code shown in Listing 2.6.

PDGs approximate program semantics. A node in a PDG represents a program statement such as an assignment statement, a method invocation statement, etc., and the edges denote control or data dependence relation between program statements.

A control dependence edge from statement s_1 to statement s_2 represents that s_2's execution depends upon s_1. While a data dependence edge between two statements s_1 and s_2 denotes that some component which is assigned at s_1 will be used in the execution of s_2. Control and data dependence relations in PDGs are computed using control flow and data flow analysis. Formally, control dependence can be stated as:

Given a control flow graph G for a program P, statements $s_1, s_2 \in G$ are control dependent iff:

- there exists a directed path ρ from s_1 to s_2 with any node S in P post-dominated ($S \neq [s_1,s_2]$) by s_2. Post-dominance refers to a situation where, for a given node, all paths from that node to the end of the control flow lead through another specific node.
- s_1 is not post-dominated by s_2

Data dependence can be formally defined as:

Two statements s_1 and s_2 are data dependent in G if there exists a variable v such that,

- v is assigned at statement s_1.
- s_2 uses the value of v.
- There exists a path between s_1 and s_2 along which there is no assignment made to v.

PDGs connect the computationally related parts of the program statements without enforcing the control sequence present in the control flow graphs [64]. Hence, they are not affected by syntactical changes like statement reordering and variable renaming. [64]. We propose that these properties make PDGs a suitable representation to detect semantic clones. Horwitz [65] and Podgurski and Clarke [66] also showed that program dependence graphs provide a good representation to measure code semantic similarity.

2.3 Deep Learning

Deep Learning covers a set of algorithms that extracts high-level representations from the input data. Deep learning models use artificial neural networks with several layers of neurons stacked together. Each layer learns to transform the previous layer's output into a slightly more abstract representation of the input data. Deep neural networks can readily model the linear and complex, non-linear relationships between input data and the desired output prediction. Many variants of Deep Neural Networks (DNN) exist, such as recurrent neural networks [67], convolutional networks [68], and graph-based neural networks [69]. In this thesis, we make use of graph-based neural networks.

2.3.1 Artificial neural networks

ANNs [70] or *connectionist systems* are machine learning models that are inspired by the human brain. ANNs consist of several artificial neurons stacked together across several layers trained to discover patterns present in the input data.

2.3.2 Graph Neural Networks

Traditional neural networks such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have shown unparalleled performance on tasks such as image recognition [71] and neural machine translation [72]. The ability of these learning models to extract complex features relevant to the downstream tasks from the input data directly is the primary reason for their matchless performance. For instance, CNN leverages image data's statistical properties (stationarity and compositionality) through local statistics. In images, stationarity is due to shift-invariance, the locality is due to the local connectivity, and compositionality stems from the multi-resolution structure of the grid. These properties are exploited by CNNs, which are built of alternating convolutional and pooling layers. The use of convolutions has a two-fold effect. First, it allows extracting local features shared across the image domain and greatly reduces the number of parameters in the network without sacrificing the expressive capacity of the network. Second, the convolutional architecture itself imposes some priors about the data, which appear very suitable, especially for natural images.

Despite the unprecedented performance of traditional neural networks on Euclidean data (in $1-$dimensional and $2-$dimensional domains), these models have restricted performance in non-Euclidean space data such as graph data. The complexity of the graph data arising due to the need to model node information and structural information simultaneously hampers the ability of traditional deep learning models to gain true insights into the underlying data. Moreover, graphs can be irregular, may have a variable size of unordered nodes, or nodes may have a different number of neighbors. Thus, these properties of graphs make some important operations, such as convolutions and filterings easy to compute in the image domain and difficult to apply to the graph domain [73, 74, 69]. Therefore, techniques from the spatial and spectral domains

have been adopted to learn representation from graph structures. Spectral graph theory studies properties of graph Laplacian or adjacency matrix using algebraic methods, e.g., studying the relationship between the eigenvalues of the Laplacian matrix and graph connectivity. Spectral and spatial graph neural networks (GNNs) are two common approaches for dealing with data represented in the form of graphs.

2.3.2.1 Spectral GNNs

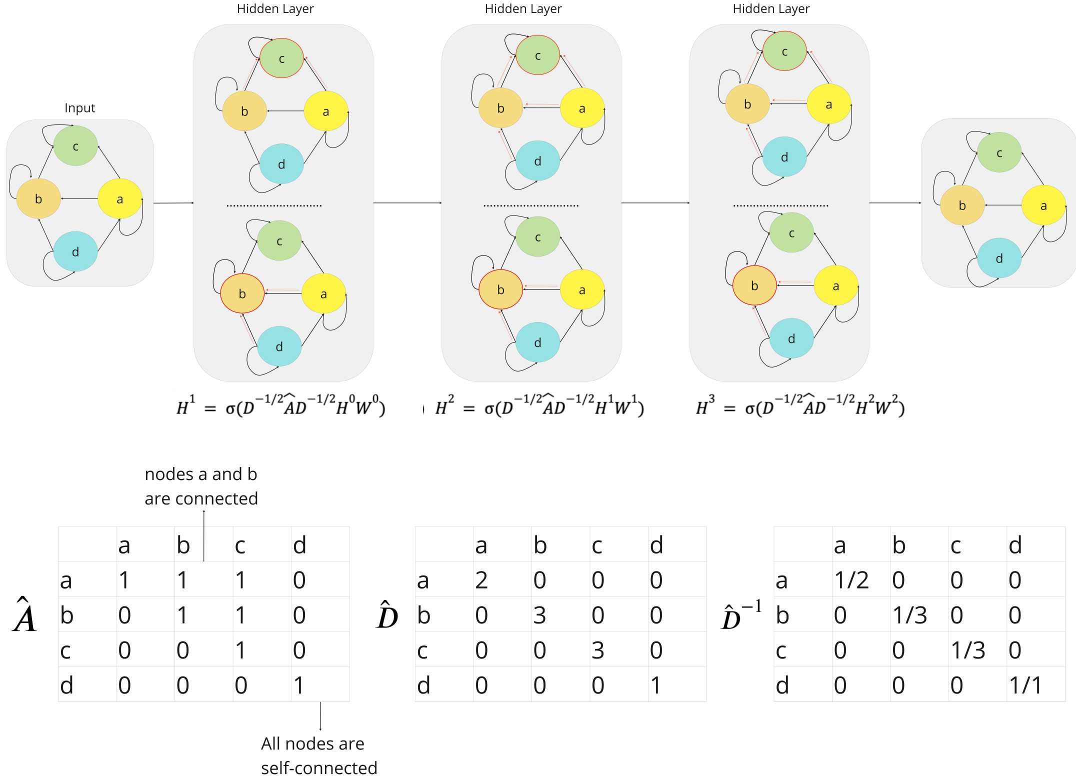
Spectral Graph Neural Networks (GNNs) are rooted in spectral graph theory, leveraging the Laplacian matrix to characterize a graph's structure. This matrix comes in two forms: the unnormalized Laplacian $L = D - A$ and the normalized Laplacian $L = I - D^{-0.5}AD^{-0.5}$, where D is the degree matrix and A is the adjacency matrix.

A distinctive feature of spectral GNNs is their reliance on the eigenvalues and eigenvectors of the Laplacian matrix. These eigenvalues represent various "frequencies" of the graph, enabling spectral GNNs to perform convolution-like operations in the spectral domain. Filters are designed as functions of these eigenvalues, facilitating the network's operation on graph signals in the spectral space.

In practice, graph signals, representing node features, are transformed into the spectral domain by projecting them onto the eigenvectors. Filters defined in this domain aggregate information from neighboring nodes, enabling effective capture of global graph properties. Common filters include graph convolution filters and ChebNet filters.

Figure 2.5 provides a high-level overview of a 3-layer Spectral Graph Convolutional Neural Network (GCN), a variant of convolutional networks is specifically designed for direct operation on graphs.

GCN operates based on a message-passing framework, leveraging a node's spatial connections to exchange information with its neighbors. The number of layers in the GCN indicates the n-hop neighborhood of the graph, allowing information exchange among neighbors. For instance, in a 3-layer GCN, each node incorporates information from its 3-hop neighbors.



$\hat{A}$

	a	b	c	d
a	1	1	1	0
b	0	1	1	0
c	0	0	1	0
d	0	0	0	1

$\hat{D}$

	a	b	c	d
a	2	0	0	0
b	0	3	0	0
c	0	0	3	0
d	0	0	0	1

$\hat{D}^{-1}$

	a	b	c	d
a	1/2	0	0	0
b	0	1/3	0	0
c	0	0	1/3	0
d	0	0	0	1/1

Figure 2.5: A general framework of 3-layer spectral graph convolutional neural network.

For each layer of the GCN, the processing involves passing $\hat{D}^{-1}\hat{A}H^iW^i$, where W^i is the weight matrix. Here, $\hat{A} = A + I$, where A is the graph adjacency matrix, and I is the identity matrix. $\hat{A}$ signifies that the output of a node in a hidden layer depends on itself and its neighboring nodes. H^i represents the node feature matrix, and $\hat{A}H^i$ indicates features on each node with its neighbors, $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$, where diagonal entries represent the degree of each node in the graph, and $\hat{D}^{-1}$ is the inverse diagonal node degree matrix.

To normalize the output of hidden layers and prevent issues like diminishing or exploding gradients, GCN multiplies $\hat{A}H^i$ with $\hat{D}^{-1}$, effectively averaging the feature vectors with its neighbors instead of summing them. The authors of the original paper [75] recommend symmetric normalization for more interesting dynamics. Therefore, for each GCN layer, we pass $\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^iW^i$.

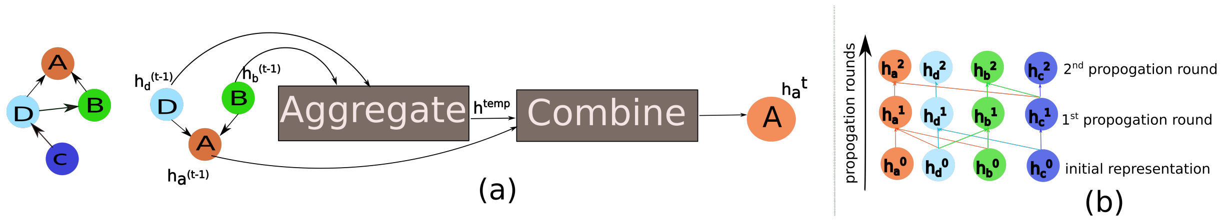


Figure 2.6: Visual illustration of the graph neural network framework. (a) Illustration of the tth layer of the graph neural network. The feature vectors h_b^{t-1}, h_d^{t-1} from the neighboring nodes of A are *aggregated* and *combined* with h_a^{t-1}, the features of node A from the $t-1^{th}$ layer. This constitutes the representation of node A at the tth layer. (b) Illustration showing multiple rounds of propagation in a graph neural network. At the nth propagation round, a node receives information from each of its neighbors that are n hops away. For example, node A at propagation round 1 receives messages from its one-hop neighbors D and B. At propagation round two, it receives information from its two-hop neighbor, i.e., node C, and so on.

2.3.2.2 Spatial GNNs

Spatial graph convolutions are grounded in the concept of message passing and define convolution operations based on a node's spatial connections. In each layer of the network, nodes engage in the exchange of messages, represented as feature vectors, with their immediate neighbors. The graph convolutional operator, denoted by the function f, is designed to generate the representation of a node v_i by aggregating its own features h_i and the features of its neighbors h_j. This aggregation function can take various forms, including simple summation or weighted combinations, and in some GNN variants, more sophisticated aggregation functions like LSTM-based methods are employed.

Spatial GNNs are typically structured into multiple layers or iterations, where each layer performs a neighborhood aggregation step. The process involves conducting multiple iterations of graph convolution to explore the depth and breadth of a node's influence. In each iteration, the node representation learned from the previous step is utilized to determine the representation for the current one. For example, in the initial iteration, information flow is restricted to first-order neighbors; in subsequent iterations, nodes accumulate information from second-order neighbors, i.e., the neighbors' neighbors. Following this iterative traversal, each node's final hidden representation incorporates information from an increasingly distant neighborhood. Figure 2.6 illustrates the general framework for spatial graph convolutions.

2.3.3 Attention Neural Networks

An Attention Neural Network, also known as an attention mechanism, is a specialized component within the neural networks designed to enhance the modeling and understanding of dependencies and relationships within data sequences. It accomplishes this by dynamically allocating different levels of focus, or "attention," to various elements of the input data during the network's processing.

This dynamic allocation is facilitated through a learnable mechanism that computes attention weights, allowing the network to weigh the importance of different elements in the input sequence based on the specific context of the task. This technique leverages the inherent structural and relational information within code to capture dependencies and interactions between code elements. Attention mechanisms, allow the model to selectively focus on different parts of the graph. By computing attention scores, the model can weigh the importance of neighboring nodes and decide which nodes to pay more attention to during the aggregation process.

Using attentions in code representation learning module, can help in learning meaningful and context-aware representation of code that can capture structural and semantic information of the code. GNNs coupled with attention mechanism enhance the ability of the model to capture nuanced relationships within code, making it more effective in understanding code semantics and structure. This approach is particularly valuable for improving the quality and efficiency of code analysis and code-related applications.

2.3.4 Siamese Networks

Siamese neural network or twin network [76, 77] is an artificial neural network for similarity learning that contains two or more identical sub-networks sharing the same set of weights and parameters. The Siamese neural networks are trained to learn the similarity between input data. They try to learn a mapping function such that the distance measured between the learned latent features in the target space represents the semantic similarity in the input space.

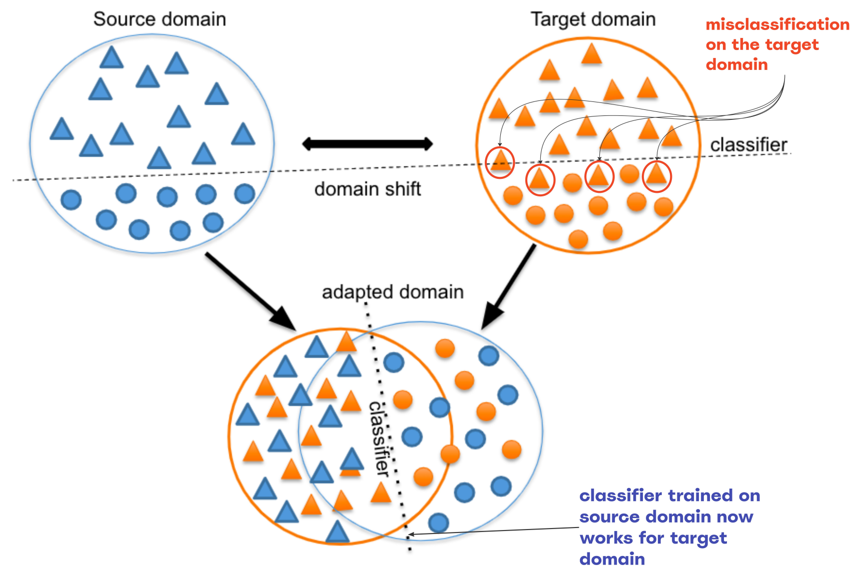


Figure 2.7: Domain adaptation aims to overcome the challenge of domain gap, where the source and target domains have different feature distributions. This can lead to poor performance of machine learning models when they are applied to the target domain. Domain adaptation techniques, such as feature alignment and adversarial training, can help align the feature distributions of the source and target domains, making it possible to transfer knowledge from the source to the target domain. This image illustrates the concept of domain adaptation, where the model learns to generalize across domains by aligning the feature distributions of the source and target domains, thus improving the model's performance in the target domain.

2.3.5 Domain Adaptation

Domain Adaptation (DA), is a technique used in machine learning to address the problem of training a model on a source domain, but applying it to a different target domain where the data distribution may be different. The idea is to learn a mapping function between the two domains, so that the model can generalize well to the target domain.

Unsupervised Domain Adaptation (UDA) is a machine learning technique that addresses the challenge of transferring knowledge learned from a source domain to a target domain, without the need for labeled data in the target domain. This is particularly useful when the distribution of the target data is different from the source data, which can lead to a domain gap and result in poor performance of the model on the target domain. UDA techniques aim to align the distributions of the source and target domains by mapping their features to a common latent space. This is typically achieved by minimizing the discrepancy between the source and target domains, either by minimizing the distance between their feature distributions or by training a domain classifier that predicts the domain labels of the data.

Adversarial unsupervised domain adaptation (UDA) [60, 78, 79, 80] is a type of DA technique used to adapt a model trained on one dataset (known as the source domain) to a different but related dataset (known as the target domain) by using adversarial training. The basic idea is to use a neural network architecture with two components: a **feature extractor** that maps input data to a feature representation, and a **domain classifier** that takes the feature representation and predicts the domain label (i.e. source or target). A high-level overview of DA is shown in Figure 2.7.

During training, the feature extractor is optimized to minimize the classification error on the source domain, while the domain classifier is trained to correctly predict the domain label. However, an adversarial loss is added to the feature extractor's objective function, which encourages the feature representations of the source and target domains to be similar, so that the classifier can't distinguish between them. By training the feature extractor to produce domain-invariant feature representations, the model can generalize better to the target domain.

Adversarial UDA can be applied to a wide range of problems, such as computer vision [60, 78], natural language processing [81], etc. where the goal is to adapt a model trained on one dataset to perform well on a different but related dataset.

Adversarial UDA is a variant of adversarial domain adaptation that adapts a model from a source domain to a target domain, where only the source domain has labeled data and the target domain has unlabelled data. Adversarial UDA for code clones is a technique that can be used to adapt a clone detection model trained on one dataset (the source domain) to perform well on a different dataset (the target domain) without any labeled data from the target domain. This can be achieved by adding an adversarial loss term to the model's training objective, which encourages the model to produce representations that are indistinguishable from those of a model trained on the target domain. A discriminator network is trained to distinguish between the source and target domains, while the main model is trained to fool the discriminator. The main model and the discriminator are trained together, in an adversarial fashion, until the main model is able to produce representations that are indistinguishable from those of a model trained on the target domain. These representations can then be used as inputs to algorithms such as clustering, classification, or deep learning models to detect code clones.

2.3.6 Neural Code Representation Learning

The practice of viewing programs as data entities and learning representations that carry syntactic and semantic significance has garnered substantial interest, as evidenced by several studies [82, 83, 84]. Given the success of deep neural networks in fields like natural language processing and computer vision, they have recently been applied to learning tasks on source code data. Areas such as program synthesis [85, 86], program repair [87, 88, 89, 90], bug localization [91, 84], and source code summarization [92] have seen extensive exploration.

The fundamental concept is to harness the knowledge embedded within existing code repositories to facilitate a broad spectrum of program analysis and maintenance tasks. The crucial initial step is transforming the source code into a format that the neural network can comprehend. This transformation, often referred to as tokenization or parsing, involves transforming the code into tokens or some intermediate representations (IR) such as abstract syntax trees, control flow graph, etc. The crux of the process is to establish a precise and semantically meaningful program representation that neural networks can utilize in a variety of downstream tasks. Most existing methodologies draw upon two types of program representations derived from static and dynamic program analysis techniques. These representations can be further divided into syntactic and semantic program representations. Following this, the IR representation is then translated into numerical vectors through methods like one-hot encoding or word embedding, a procedure known as vectorization. This process effectively morphs the source code into a mathematical entity that the neural network can process.

The vectorized code is subsequently used to train the neural network. Throughout this training phase, the network adapts to recognize patterns within the code, adjusting its internal weights and biases based on the prediction errors it incurs, typically through a method known as backpropagation. Once the network is adequately trained, it can extract features from new code samples. This extraction is accomplished by feeding the code through the network and noting the activations of the neurons within the hidden layers, which serve as the representative features of the code.

Graph Neural Networks (GNNs) for Code Representation Learning: GNNs have proven

to be highly effective for learning code representations, mainly due to their capacity to encapsulate the structural information of the code. Initially, the source code is parsed into a graph structure, such as an Abstract Syntax Tree (AST) or a Control Flow Graph (CFG), with each node representing a token or statement in the source code and the edges symbolizing their relationships. Each node in the graph is subsequently embedded into a high-dimensional vector using an embedding technique, thus creating the initial features of the nodes. The GNN then applies a propagation rule that updates the features of a node based on the features of its neighboring nodes. This process is iterated until the features converge. After propagation, a readout function aggregates the features of all the nodes in the graph to generate a single vector representation for the entire graph, essentially representing the source code. These vector representations are then used to train a machine learning model for specific tasks like code completion, bug detection, or code search. Once trained, the model can be employed to make predictions on new source code pieces.

2.4 Learning-based Code Clone Detection Process

Code clone detection process begins with the pre-processing of the source code. This step involves cleaning the code, removing unnecessary details, and converting it into a suitable format for further analysis. The next step is feature extraction, which involves transforming the source code into tokens or using more complex methods to extract syntactic and semantic features from the code. This step is crucial in representing the code in a way that can be processed by machine learning models.

After the features are extracted, a representation learning method is applied. This could be a traditional method like Bag of Words (BoW), or more advanced methods like Word Embedding or even deep neural networks like convolutional networks can be used. This step converts the extracted features into a numerical form that can capture the semantic meaning of the code. The numerical representations of the code are then used to train a machine learning model. Depending on the type of code clone detection problem, different models could be used. For example, binary classification models can be used for detecting if two code fragments are clones

or not, while clustering models can be used for finding groups of similar code fragments. The model is trained using a dataset of known clones.

To measure code similarity, the vectors can be then compared directly using a distance-based metric with some pre-determined threshold value or is passed to a feed-forward deep neural network to learn similarities between code snippets.

To learn program features, these techniques make use of different program representations. For instance, White et al. [93] represented source code as a stream of identifiers and literals and used recursive neural networks to learn program features. Perez and Chiba [63] proposed a semi-supervised learning-based approach to detect clones across different programming languages. The authors proposed to learn program representation using AST token vectors, and skip-gram model [94]. The learned representations are then passed to LSTMs to learn similarities between them. Wazed et al. [59] used a deep learning-based word vector learning method to learn semantic relations among API documentation to detect clones across different languages.

Chapter 3

Related Work

In the vast landscape of software engineering and code analysis, improving clone detection techniques has been a topic of great deal and interest. This chapter delves through the annals of literature and research, tracing the evolution of clone detection techniques that have laid the groundwork for this thesis.

3.1 Traditional Approaches to Code Clone Detection

The investigation of clones and their detection mechanisms is a multifaceted endeavor that spans a trajectory from traditional syntactic approaches to the modern learning-based detection techniques. Most traditional code clone detection techniques target Type 1-3 code clones. These techniques measure code similarity using program representations such as program text [31, 95, 32, 96], lexical tokens [30, 22, 97, 98], trees [29], and metrics [37, 38, 39, 40, 41, 42].

Text-based code clone detection techniques involve comparing the textual content of source code to identify clones. These techniques typically consider code as sequences of text and use string matching or other text comparison algorithms to detect similar code fragments. Dup [31, 95] matches sequence of lines to detects clones. During this process, Dup eliminates tabs, whitespaces, and comments and substitutes identifiers for functions, variables, and types with a unique parameter. It then merges all lines under analysis into a single text line, generates a hash

for each line for comparison purposes, and uses a suffix tree algorithm to extract a set of pairs that exhibit the longest matches.

Token-based detection approaches, parses source code into sequence of tokens, which represents the basic building blocks of a source code. The token sequences are then scanned for finding duplicated sub sequences of tokens, and the original code segments corresponding to these duplicates are returned as clones. Typically, token-based approaches prove more resilient to code alterations, such as formatting and spacing, compared to text-based methods. CCFinder [30] first converts the source code into tokens, concatenate into a single token sequence, and transform the token sequence based on the transformation rules aiming at regularization of identifiers and identification of structures. A suffix-tree based sub-string matching algorithm is then used to find the similar sub-sequences on the transformed token sequence where the similar sub-sequence pairs are returned as clone pairs/clone classes. CP-Miner [22] uses a frequent sub sequence mining technique for identifying a similar sequence of tokenized statements. SourcererCC [99] compares token sub sequences to identify program similarity. It targets three clone types, and exploits optimized inverted-index to achieve scalability to large inter-project repositories using a standard workstation. Filtering heuristics based on token ordering are also used to reduce the size of the index, the number of code-block comparisons needed to detect the clones, as well as the number of required token-comparisons needed to judge a potential clone.

Token-based techniques have also been used for **plagiarism detection** [98]. JPlag tokenizes the source code into series of language-independent tokens disregarding comments, variable names, etc. The sequence of transformed tokens are then compared pairwise to identify the longest common sub sequences of tokens that appear sequentially in both the programs.

Tree-based techniques, transform the program into parse trees or abstract syntax tress (ASTs) with a parser. Similar subtress are then searched in the tree with some tree matching algorithm to detect clones. CloneDR [100] uses a compiler to generate annotated parse tree, and compare its subtrees by characterization metrics based on a hash function through tree matching. Bauhaus's [101] clone detection tool named ccdiml is a variant of CloneDR with some differences like the avoidance of the similarity metric, the handling of sequences and the hashing. Deckard [29] is another popular tree-based code clone detection technique, which computes characteristic

vectors for AST nodes of a given program. It then applies Locality Sensitive Hashing (LSH) to find similar code pairs.

There are also **graph-based** techniques [33, 34, 102] that use program dependency graphs (PDGs) to identify clones. PDG-based techniques go a step further in abstraction and obtain semantic information from the source code by considering control and data dependencies. Once the PDGs are obtained from code snippets, isomorphic subgraph matching algorithms are applied for finding similar subgraphs which are then returned as clones. PDG-DUP [33] first converts the given program to PDG and then uses program slicing and subgraph isomorphism to identify clone pairs. DUPLIX [34] also uses program slicing and graph isomorphism to identify similar code pairs. Krinke [103] uses PDG representation of code snippets and modeled clone detection problem as *maximal similar subgraph construction* problem. Gabel et al. [52] compares PDGs of code pairs to detect clones. They reduce the graph isomorphism problem to a tree matching problem by mapping PDG representation to ASTs. However, the techniques are imprecise and are not scalable in practice due to the inherent complexity of graph isomorphism and the approximations made while mapping PDG's subgraphs to ASTs [52].

In addition to these, some techniques also exist that uses **dynamic program analysis** to detect clones. Su et al. [35] proposed the notion of code relatives, which are code snippet with similar execution behavior and present DYCLINK to detect code relatives within and across codebases. DYCLINK uses instruction level execution traces to search for code relatives. It converts these execution traces into dynamic dependence graph and employs a specialized subgraph matching algorithm to compare traces and detect code relatives. MeCC [28] proposes a clone detection technique, that compares the abstract memory states of programs, which are computed by a path-sensitive static analyzer.

Apart from these techniques which primarily focus on semantics, syntax and structure of the code, there exists techniques that exploits code characteristics, such as lines of code, cyclomatic complexity, depth of inheritance, number of children, etc., to identify potential clones. They do not directly compare the code but instead compare these extracted metrics. The idea is that if two code segments have very similar metrics, they might be clones. Mayrand et al. [40] calculated several metrics such as number of lines of code, number of function calls, number of edges in

control flow graph, etc for each function of a program. Functions with similar metrics were identified as clones. Kontogiannis et al. [37, 38] proposes two ways of detecting clones. One approach is the direct comparison of the metrics values that classify a code block in a fragment with the assumption that two code fragments are similar if their corresponding metrics values are proximate. The second approach uses a dynamic programming technique for comparing block at a statement-by-statement basis. A more recent technique, Oreo [104], proposed to use a combination of machine learning, information retrieval, and software metrics to detect clones in the twilight zone. The authors proposed a twilight zone category of clones, that are very hard to detect because they have weak syntactic similarity but strong semantic similarity.

3.2 Learning-based Techniques to Code Clone Detection

Learning from data to identify code clones has also been a great deal of interest from the past. There have been data mining-based techniques to learn code similarity [36, 105]. Marcus and Maletic [36] propose to use latent semantic indexing to detect semantic clones. Their approach examines source code text (comments and identifiers) and identifies the implementation of similar high-level concepts such as abstract data types.

With the increasing prevalence of learning-based techniques for understanding and reasoning over source code and the availability of large OSS projects, learning-based techniques to detect clones has recently garnered researchers' attention [106, 107, 108, 61, 109, 110]. The modus operandi of these techniques is to learn and compare the continuous vector-based representation of code using a distance metric (such as cosine similarity) or neural network. To achieve this goal, the program features defining the source code's functionality are learned. Diverse program representations comprising of tokens, Abstract Syntax Trees (ASTs), Control Flow Graph (CFGs), Data Flow Graphs (DFGs) are used to learn program features.

White et al. [46] uses a recursive neural network (RNNs) to learn program representations. They represent source code as a stream of identifiers and literals and use it as an input to their deep learning model. Zhao et al. [44] use feature vectors extracted from the data flow graph of a program to learn program representation using deep neural networks. Tufano et al. [50] using a

similar encoding approach as [46] encode four different program representations - identifiers, Abstract Syntax Trees, Control Flow Graphs, and Bytecode. A deep learning model is then used to measure code similarity based on their multiple learned representations. Wei et al. [47] use AST and tree-based LSTM to learn program representation. Yu et al. [43] use tree-based convolutions over ASTs to learn program representation. Saini et al. [111] propose to detect clones using software metrics and machine learning in the twilight zone.

While learning-based techniques for identifying semantic code clones are on the rise, a significant portion of these methods still heavily relies on syntactic features, often neglecting the crucial aspect of program semantics. We contend that comprehending program semantics is of utmost importance for measuring code's semantic similarity. Consequently, a more advanced program representation is imperative to capture the intricate functional behaviors interwoven within the source code.

Furthermore, the choice of neural network architecture for program representation learning falls short in capturing the structured syntactic and semantic information present in code snippets. For instance, RNNs, as employed by [46, 50], are well-suited for handling sequential data. However, source code, while possessing a degree of sequential structure, is not strictly sequential. It often involves complex dependencies and hierarchical relationships that go beyond the capabilities of RNNs, which primarily excel in capturing syntactic patterns. Furthermore, code representation learning necessitates capturing not just the syntax but also the semantics of the code. RNNs, by design, lean more towards capturing syntactic patterns and may not inherently capture higher-level semantic information.

In a similar vein, the tree-convolutional network used by [43] is also not ideally suited for code representation learning. While code does exhibit a hierarchical structure, it doesn't always conform to a strict tree format. Various code elements, such as control flow and function calls, introduce cyclic and complex relationships that don't neatly fit into a tree structure. This disconnect between the structure of code and the tree-based convolution approach can pose challenges in accurately representing and analyzing code. Additionally, source code often involves variable-length sequences and nested structures, making it less amenable to tree-based convolutions designed for fixed or shallow tree structures. These tree-based methods may struggle

to capture high-level semantic information as they are primarily geared towards capturing local syntactic patterns.

Thus, addressing the above issues, as a first contribution towards this thesis, we propose a new tool **HOLMES**, for measuring code functional similarity. HOLMES uses program dependency graphs and attention-based Siamese graph neural networks for program representation learning and code similarity detection.

Despite the progress in single-language code clone detection research, a significant research gap still exists in detecting cross-language clones. Initial attempts in this field were made by Kraft et al., using Microsoft .Net framework-based programming languages like C#, Visual Basic, and ASP.Net, which have a common intermediate representation, CodeDom, leveraged for detecting clones [112]. Their model, however, cannot detect clones across all programming languages, such as Java and Python or Java and C#, where the languages do not share any common intermediate framework.

Vislavski et al. proposed a tool named LICCA [113], which uses SSQSA [114] platform to generate common intermediate representations (eCST) for various languages and then applies a variant of longest common subsequence algorithm to detect clones. However, LICCA suffers from some restrictions such as equal source code length and similar control and data flow, limiting its applicability on real-world software systems. Cheng et al. proposed CLCMiner [115] to detect cross-language clones by mining revision histories of software systems. The authors use a word-token matching strategy to track code changes across different languages and rely on the assumption regarding alignment between cross-language code snippets as long as the changes in different languages use similar lexical features. Although, CLCMiner can detect clones across different programming languages, it fails to detect functionally similar clones with substantial syntactic differences.

Recently, Perez and Cheba proposed a deep learning-based approach that can learn vector representations from ASTs using LSTM [63]. Their model is quite capable of detecting cross-language clones between Java and Python, but the model suffers from low precision. The reason can be attributed to the use of ASTs and LSTMs. Since ASTs only represent programs' syntactic

structure with limited semantics, it might be possible that two dissimilar code snippets can share substantial syntax (in terms of API calls, token sequences, etc.) and are reported as clones by their tool. Moreover, LSTMs do not exploit the available structured information from source code which again limits the capability of their tool. Another recent work by Nafi et al. proposes a tool, CLCDSA [59], which can detect cross-language clones for Java, Python, and C#. It detects clones on the fly by comparing the similarity of nine features. It also proposes a deep neural network-based learning model to learn program representation using these handcrafted features and then measure code similarity using a binary classifier. CLCDSA makes use of handcrafted features for learning code similarity while overlooking the available structured semantics of source code. This limits CLCDSAs' performance on real-world projects.

Mathew et al. proposed a tool, SLACC [116], which uses a dynamic analysis approach to detect behavioral cross-language code clones. It identifies clones by comparing the IO relationship of segmented snippets of code from a target repository where these inputs are generated using multi-modal gray-box fuzzing. Nevertheless, SLACC can detect behavioral cross-language code snippets. Still, it relies on dynamic analysis, which is an expensive technique in itself and only supports primitive data types and simple data structures. These limitations make it difficult to use SLACC on real-world software systems. COSAL, a cross-language code search tool proposed by Mathew and Stolee [117], can also be utilized for cross-language clone detection by choosing its top-ranked results. To identify similar code snippets, COSAL employs both static and dynamic analysis techniques. However, like SLACC, COSAL also employs input-output based search and thus shares its limitations. This could pose challenges when dealing with code snippets that are hard to execute or fuzz.

Tao et al. [118] introduced C4 that employs a pre-trained language model named CodeBERT to translate programs from various languages into high-dimensional vector representations. This model is then fine-tuned using a contrastive learning objective that can differentiate between clone pairs and non-clone pairs. Guo et al. [119] developed a graph-based pre-trained language model for code called GraphCodeBert. This model takes into account the inherent structure of the code and uses data flow information in conjunction with code sequences to learn code representations. These representations encapsulate both the syntactic and semantic aspects of

the code. The authors demonstrated that GraphCodeBert can also be applied for cross-language code clone detection. XCode [120], on the other hand, proposes a pre-training method to learn code representations that can comprehend the cross-language semantics of the code. XCode utilizes a multi-task learning framework to learn code representations from a collection of code snippets in various programming languages and employs a Transformer-based encoder-decoder architecture for code representation learning.

However, these techniques do not fully reveal the semantic and syntactic information from the source code, which is essential for detecting similarity across languages. For instance, tools like XCode and C4 primarily rely on code tokens and ASTs, whereas GraphCodeBert suggests using a specific data flow relation known as "where the value comes from." These techniques fall short in exposing the complete semantics and syntax of a program, which is crucial for learning similarity across cross-language clones.

Thus, to overcome the aforementioned challenges in cross-language code clone detection, as a second contribution to this thesis, we propose a semi-supervised learning-based cross-language code clone detection tool, **RUBHUS**. RUBHUS uses ASTs enriched with semantic information from source code and GNNs to model similarity between code snippets. To improve the learned local AST node representations while capturing various global latent structures and sub-structures, we use unsupervised learning based on the principle of Mutual Information (MI) maximization. This approach allows us to encode different aspects of the data present at various AST sub-structures, which would have been missed otherwise.

There have also been some efforts in detecting clones in binaries [121, 122], including Graph Matching Network (GMN) proposed by Li et al. [123]. This becomes particularly relevant when the source code is inaccessible due to licensing constraints or security considerations. GMN used cross-graph attention-based matching to calculate similarity scores, while Gemini [124] used Structure2vec to identify binary function features from control flow graphs and compare them using cosine distance. Tracelet [125] proposed to measure binary similarity by calculating edit distance between instruction sequences. Genius [126], DeepBinDiff [127] fed code binaries into deep neural networks to learn their representation for code similarity detection. TREX [128] used a hierarchical transformer to encode execution semantics.

Despite these advancements, clone detection techniques still grapple with certain limitations. These include the reliance on labeled data for effective training, which can be challenging to acquire, and the inherent inability to identify novel types of clones not present in the training data.

In response to this challenge, Liu et al. [129] introduced a human-in-the-loop mechanism that combines transfer learning and active learning to adapt neural clone detectors to new code repositories that may contain numerous unseen functionalities. Although Liu et al.'s mechanism used for adapting neural clone detectors to new code repositories is effective in reducing the amount of annotations required. However, it still relies on human annotation, which can be time-consuming and resource-intensive, especially for larger repositories and cannot be autonomously adapted to different codebases and domains.

Adversarial Unsupervised Domain Adaptation (UDA) [60] is a machine learning technique that enables the autonomous adaptation of pre-trained neural models to unseen repositories, all without the need for labeled data in the target domain. Thus, UDA offers an appealing alternative for adapting a trained clone detection tool to a new dataset, eliminating the necessity for re-training or fine-tuning the model on the new unlabeled dataset.

Thus, as a third contribution to this thesis, we introduce a domain adaptation framework, CLODIA that eliminates the need for extensive re-training of models for each new dataset. **CLODIA** autonomously transfers the knowledge learned from seen source domains to unknown target domains. Thus, it enables the model to adapt and apply its previously acquired knowledge to new and unseen datasets without the need for time-consuming re-training.

3.3 Representation Learning for Source Code

In the realm of program analysis, there has been a shift towards utilizing deep learning models to obtain program embeddings, aiming to create precise representations of source code to address various software engineering problems. Initially, methods from Natural Language Processing (NLP) were employed, representing programs as sequences of lexical tokens [130, 131]. However,

it has become increasingly evident [132, 133] that capturing the structured nature of programs is crucial. Consequently, researchers have explored both syntactic (tree-based) and semantic (graph-based) representations [134, 135].

Abstract Syntax Trees (ASTs) have served as a cornerstone in program representation research. For instance, Dam et al. [136] annotated AST nodes with type information and utilized Tree-Based LSTMs [137] for defect prediction. Similarly, Raychev et al. [133] and Alon et al. [132, 83] leveraged path-based abstractions of ASTs as program representations. Furthermore, Allamanis et al. [84] augmented ASTs with additional typed edges and employed Graph Gated Neural Networks (GGNNs) [138] for tasks related to variable naming.

Another avenue of research involves modeling binary similarity using Control-Flow Graphs (CFGs), with adaptations like Graph Matching Networks [123]. Additionally, historical techniques like Program Dependence Graphs (PDGs) [64] have been pivotal for optimization, simplifying the graph representation of programs. Contemporary approaches, such as ConteXtual Flow Graphs (XFGs) [139], blend control flow with data flow to learn unsupervised embeddings of LLVM-IR statements. Similarly, IR2Vec [140] defines an LLVM-IR-specific statement representation, albeit requiring access to data flow analyses.

Recent advancements in deep learning have propelled the pursuit of precise program embeddings to address various software engineering challenges. For example, Gupta et al. [141] leverage sequence-to-sequence neural networks with attention mechanisms for rectifying programming errors. Similarly, Wang et al. [142] employ program execution traces for learning program embeddings to predict method names, while Ben-Nun et al. [143] utilize recurrent neural networks with LLVM-generated intermediate representations.

Graph Neural Networks (GNNs) have emerged as a versatile tool for software engineering tasks. Allamanis et al. [84] utilize GNNs for tasks akin to linters, aiding in variable name predictions and misuse detection. LeClair et al. [144] and Lu et al. [145] employ GNNs for program classification, facilitating code mining and automatic completion. Furthermore, GNNs are instrumental in resource management tasks, as demonstrated by Cummins et al. [146], who utilize them for compiler analysis to optimize compilation and execution times.

Chapter 4

Graph-Based Siamese Networks for Assessing Functional Similarity in Source Code

4.1 Introduction

Semantic or functional clones refer to code segments that, while differing in syntax or structure, perform similar operations or functions.[1] These clones may diverge in terms of code structure and syntactic elements, thereby posing a challenge for detection. In contrast, syntactic clones, which usually involve minimal to no changes from the copied source code, are much easier to identify. Despite their syntactic variations, semantic clones share operational semantics, meaning they execute similar tasks or functions. This makes understanding and managing them crucial in the field of software development and maintenance.

A substantial amount of research effort has been put in to detect semantic clones. For instance, Yu et al. [43] propose TBCCD that uses tree-based convolutions that exploit structural and lexical information from the ASTs of the code fragments to measure code similarity. Deepsim [108] propose to use deep learning to encode both the control flow and data flow of source code into a

[1] This chapter is a reproduction of a paper published at IEEE Transactions of Software Engineering, 2021. [147]

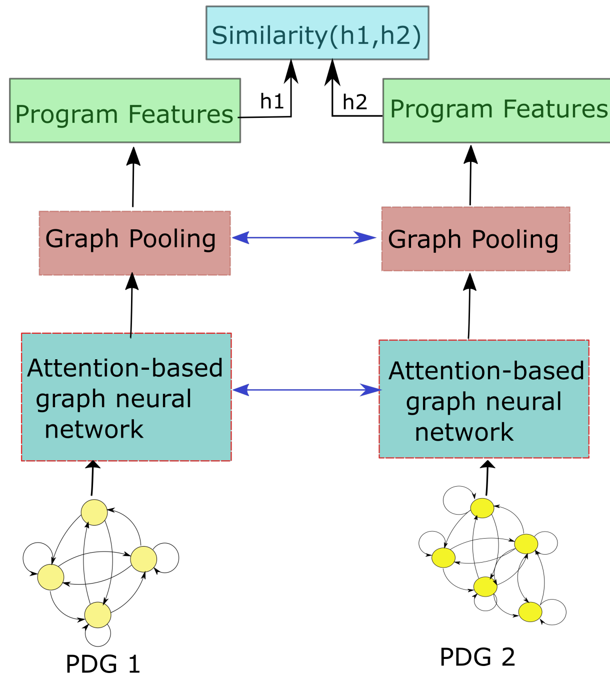


Figure 4.1: The proposed Siamese deep neural network consists of two identical sub-networks. The input to the sub-network is the pair of Java methods represented as PDGs. Each sub-network incorporates an attention-based graph neural network model to learn PDGs node features, which are then aggregated using soft attention to constitute a graph-level representation of the given input method. The proposed network is trained to learn the similarity between two feature vectors h1 and h2. Horizontal blue arrows denotes weight shared sub-networks.

semantic matrix, and then compares the similarity of the matrices to identify functional clones.

Notwithstanding this, we argue that these program representations may not adequately reveal the semantic information of a program to the neural network, a factor that could be essential in determining functional code similarity. For instance, ASTs, as utilized by TBCCD, serve merely as blueprints of the source code, capturing solely its syntactic structure. Similarly, the matrix encoding method proposed by DeepSim falls short in recognizing long-range dependencies or intricate relationships brought about by different code constructs. Thus, a more sophisticated program representation is required to learn the functional behaviors of the source code.

Thus, addressing the above issues, we propose a new tool HOLMES, for measuring code functional similarity. HOLMES is based on two key insights. First, feature learning plays a significant role in measuring code similarity [46]. Thus, learned features should contain semantic information, specifically control and data dependence information, rather than structural information, such as lexical elements and high-level program constructs captured by ASTs. Code similarity based solely on syntactic features is too restrictive and rigid in its expression compared

to its more powerful and expressive notion based on program semantics or functionality. Hence, HOLMES uses the control and data dependence information from PDGs as a basis of similarity metrics. Second, to capture program semantics efficiently, one must capitalize on PDGs graphical structure. Therefore, HOLMES employs a graph-based deep neural network to learn program representation. Figure 4.1 shows the overall architecture of the HOLMES.

We have implemented HOLMES in Java using the Soot optimization framework [148] and PyTorch Geometric [149] deep learning library. We evaluate HOLMES on programming competition datasets and real-world datasets. Our empirical results show that HOLMES outperforms another state-of-the-art tool, TBCCD [61], and generalizes better on unseen code pairs. We make the following contributions:

1. **A new code representation for semantic code clone detection**. To the best of our knowledge, our work is the first to learn code representation for code clone detection in two different manners: i) Using control and data dependence relations between the program statements to model code functional dependency ii) Treating control and data dependence edges differently to give respective importance to syntactic and semantic information while learning code representation for a code snippet.

2. **A new code clone detection approach**. We propose a new deep learning architecture for graph similarity learning. Our approach jointly learns the graph representation and graph matching function for computing graph similarity. In particular, we have used an attention-based Siamese graph neural network to detect semantic clones. Our approach uses the control-dependent and data-dependent edges of PDGs to model the program's semantic and syntactic features. We have used attention to give higher weights to semantically relevant paths. The learned latent features are then used to measure code functional similarity.

3. **A comprehensive comparative evaluation**. We developed a prototype tool HOLMES and evaluated it on popular benchmarks for code clones against state-of-the-art-tool TBCCD. Through a series of empirical evaluations, our results show that HOLMES outperforms TBCCD.

```java
public static void main(String[] args){
  Scanner in = new Scanner(System.in) ;
  int T = in.nextInt() ;
  int[] a = new int[T] ;
  for (int j = 0 ; j < T ; j ++)
    a[j] = in.nextInt() ;
  int c = 0 ;
  for (int j = 0 ; j < T ; j ++)
    if (a[j] == j+1)
      c ++ ;
  System.out.println("Case
  ↪  #"+i+":"+((double)T-(double)c));
}
```

Listing 4.1: Sort1.java

```java
public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  int n = in.nextInt(),t=0;
  float count = 0.0f;
  while(n>0){
    if(++t! = in.nextInt())
      count++;
    n--;
  }
  System.out.printf("Case#%d:%.6f%n", i,
  ↪  count);
}
```

Listing 4.2: Sort2.java

Figure 4.2: A semantic code clone example detected by HOLMES, which was reported as false negative by TBCCD. The code in Listings 4.1 and 4.2 sort an array of natural numbers by randomly shuffling the array n times.

4.2 Motivation

In this section, we present an example and our observations to motivate our approach.

4.2.1 Motivating Example.

Listings 4.1 and 4.2 show two solutions submitted for the *GoogleCodeJam* problem *Goro Sort*. The problem involves an interesting method of sorting an array of natural numbers in which the array is shuffled n times randomly to get it sorted. The users have to report the minimum number of times shuffling is required to sort the array.

Listing 4.1 implements the above functionality by first initializing an array of size T with

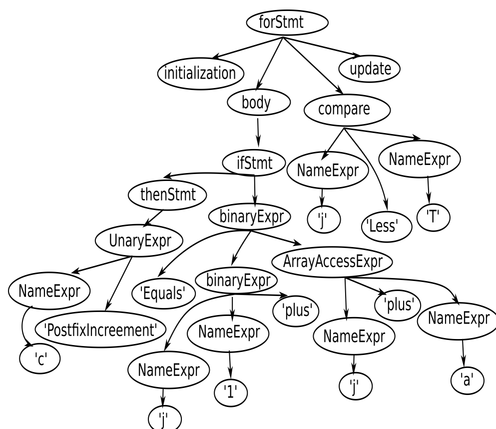


Figure 4.3: AST for Listing 4.1.

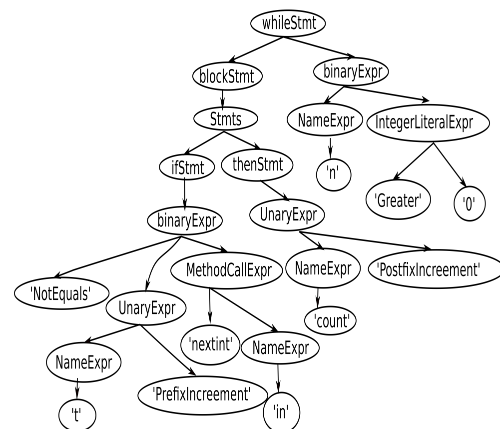


Figure 4.4: AST for Listing 4.2.

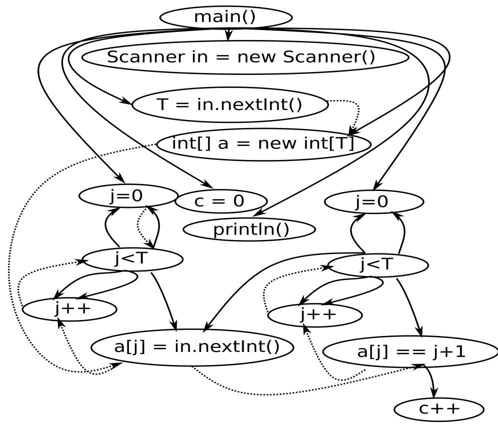


Figure 4.5: PDG for Listing 4.1.

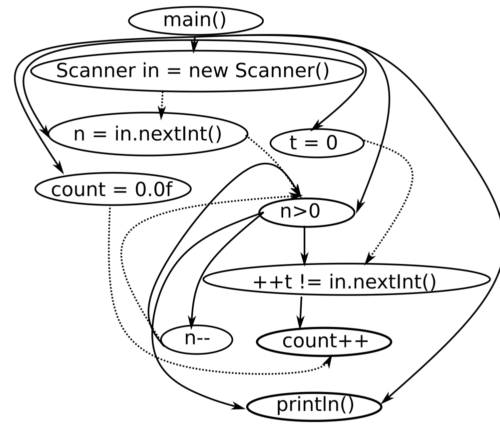


Figure 4.6: PDG for Listing 4.2.

random numbers. It then checks if the current index element is equal to the index of the next element. The minimum number of times shuffling is required to sort the array was given by the size of the array (T) minus the number of times the element at the current index equals the next index.

Listing 4.2 implements the same functionality while taking input from the user at run time. It keeps the counter; if the current value is equal to counter$+1$, it reduces the minimum number of times shuffling is required by -1. Syntactically, Listings 4.1 and 4.2 are quite different. However, semantically, they are similar and will be classified as type 4 clones according to the taxonomy proposed in [2].

The existing ML-based code clone detection approaches [47, 45, 43] use syntactic and(or) lexical information to learn program features. For instance, TBCCD [43] uses tree-based convolution over abstract syntax trees (ASTs) to learn program representation. If we look at the ASTs of Listings 4.1 (for line $8-11$) and 4.2 (for line $5-8$) shown in Figures 4.3 and 4.4, it is hard to infer that the two ASTs correspond to similar programs. We executed TBCCD, trained on *GoogleCodeJam* problems($2010-2017$), on this example and TBCCD caused a false negative by reporting Listings 4.1 and 4.2 as a non-clone pair. This led us to our first **Observation (O1):** *To achieve accurate detection of semantic clones, we need to incorporate more semantic information while learning program representation.*

We then computed PDGs for Listings 4.1 and 4.2. The PDGs are shown in Figures 4.5 and 4.6. From Figures 4.5 and 4.6, we observe that the flow of data and control between the two

PDGs are similar, as PDGs approximate the semantic dependencies between the statements. However, PDGs suffer from scalability problems. The size of PDGs can be considerably large. For a program with 40-50 lines of code, we can have around 100 vertices and 100 edges. This led us to our second **Observation (O2):** *To learn important semantic features from the source code, a model should not weigh all paths equally. It should learn to give higher weights to semantically relevant paths.*

Source code is a complex web of interacting components such as classes, routines, and program statements. Understanding source code amounts to understanding the interactions between different components. Previous studies such as [150] have shown that graphical representation of source code is better suited to study and analyze these complex relationships between different components. Yet, the recent code clone detection approaches [45, 44, 46] do not make use of these well defined graphical structures while learning program representation. These approaches use deep learning models that do not take advantage of the available structured input, for example, capturing induced long-range variable dependency between program statements. This led us to our third **Observation (O3):** *To capitalize on the source code's structured semantic features, one might have to expose these semantics explicitly as a structured input to the neural network model.*

4.2.2 Key Ideas.

Based on the above observations, we have created our approach with the following key ideas:
$a)$ From observation 1, we learned program features from the PDG representation of source code to capture the program semantics. Such graphs enable us to capture the data and control dependence between the program statements.
$b)$ From observation 2, we designed an attention-based deep neural network to model the relationship between the important nodes in the PDG. The attention-based model emphasizes learning the semantically relevant paths in the PDG necessary to measure code similarity.
$c)$ From observation 3, we used a graph-based neural network model to learn the structured semantic features of the source code. We have encoded the source code's semantics and syntax

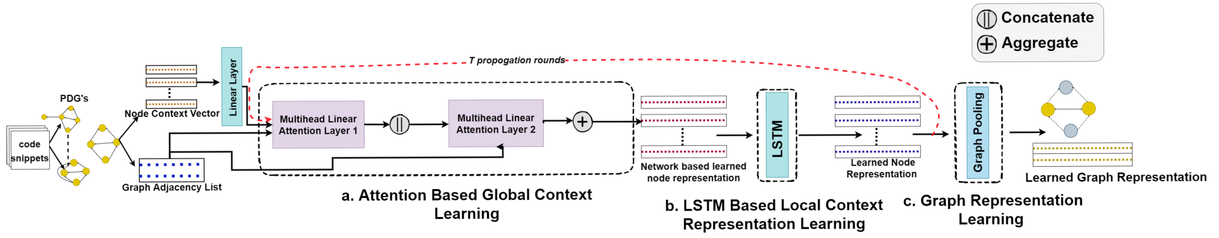


Figure 4.7: The architecture of one branch of the Siamese neural network is shown in Figure 4.1. (a) Our model first parses the given Java methods in the datasets to build PDGs. Node feature matrix and graph adjacency matrix are extracted from the source code. HOLMES then passes this as input to a multi-head masked linear attention module, which learns the importance of different sized neighborhood for a node. (b)The attention module outputs the set of learned node features that are then passed through an LSTM, which extracts and filters the features aggregated from different hop neighbors. (c)The learned node features are then passed to a graph pooling module. Graph pooling employs a soft attention mechanism to downsample the nodes and to generate a coarsened graph representation.

into a graph-based structure and used a graph-based deep learning model to learn latent program features.

4.3 Approach Overview

This section discusses the details of our graph neural network architecture that is used to learn the high-level program features from PDGs. Figure 4.7 shows an overview of one branch of the Siamese neural network shown in Figure 4.1. The following subsections give details of the main steps of the proposed approach.

4.3.1 Attention-based Global Context Learning

Our work builds on Graph Attention Networks (GAT) [151], and we summarize them here. Given a program dependence graph $G = (V, E, A, X)$, we have a set of V vertices representing program statements, and a list of directed control and data-dependent edges $E = (E_1, E_2)$. A denotes the adjacency matrix of G, where $A \in \mathbb{R}^{|V|\times|V|}$ with A_{ij} = 1 if $e_{ij} \in E$, and A_{ij} = 0 if $e_{ij} \notin E$. The feature embedding matrix is represented by $X \in \mathbb{R}^{|V|\times d}$, where $x_v \in \mathbb{R}^d$ denotes the feature vector of vertex v.

For every node $v \in V$, we associate a feature vector x_v, representing the type of statement it belongs to. We considered the following 18 types: `Identity, Assignment, Abstract,`

`Abstract Definition, Breakpoint, Enter Monitor, Exit Monitor, Goto, If, Invoke, Lookup, Switch, Nop, Return, Return void, Throw, JTableSwitch` corresponding to the types of the statements used by Soot's internal representation. We encode this statement type information into an 18-dimensional one-hot encoded feature vector. For example, the statement $x = y + z$ is of type `Assignment statement` and will be represented as **[0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]**. In the first place, to obtain initial node vectors, we pass node features through a linear transformation layer:

$$H^0 = X \times W + b \tag{4.1}$$

Where W and b are the learnable weight matrix and bias vector of the linear layer. $H^0 \in \mathbb{R}^{|V| \times d'}$ denotes the initial node embedding matrix where h_i^0 represents embedding vector for a single node $i \in V$. Line 16 in Algorithm 1 inside function $ComputeGFeatures$ denotes the above action.

Next, to obtain a high-level representation for the nodes of the PDG, we learn an adaptive function $\varphi(A, H; \phi)$ parameterized by ϕ similar to GAT [151]. The input to the function is the set of node features $\{h_1^0, h_2^0, \cdots, h_{|V|}^0\}$ obtained from Equation (4.1). The function φ then outputs the set of new node features, $\{h_1^{'}, h_2^{'}, \cdots, h_{|V|}^{'}\}$ which denotes the first attention block's output. It computes self-attention on nodes based on the graph structural information i.e., a node v_i attends to its one-hop neighboring node v_j, if $(v_i, v_j) \in E$. The attention mechanism $a : \mathbb{R}^{d'} \times \mathbb{R}^{d'} \to \mathbb{R}$ computes attention coefficients between nodes v_i and v_j.

$$e_{ij} = a(Wh_i, Wh_j) \tag{4.2}$$

Equation (4.2) denotes the importance of node $j's$ features for node i. The scores are then normalized using the softmax function.

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{j \in N(i)} exp(e_{ij})} \tag{4.3}$$

The attention scores computed in Equation (4.3) are then used to output a linear combination of features of node v_j, $\forall j \in N(i)$ that will be used as the final output features of node v_i. Also,

Algorithm 1: Code Similarity Detection

```
   Input: T rounds of propagation, train_Data
   Output: code_similarity
 1 Define ComputeNodeFeatures(H^0, A, T):
 2     H_F = [H^0]
       // C^0 is the initial cell state of LSTM
 3     Initialize Array C^0
 4     for t ← 1 to T do
           // Attention block 1 (Eqn. (4.4))
 5         H' = Attn_1(A, H^(t-1); φ_1)
           // Attention block 2 (Eqn. (4.5))
 6         H''^(t-1) = Attn_2(A, H'; φ_2)
           // t^th propagation round
 7         H^(t), C^(t) = LSTM(H''^(t-1), C^(t-1))
           // Final node features (Eqn. (4.6))
 8         H_F = CONCAT(H^(t))
 9     return H_F
10
11 Define GraphPooling(H_node):
12     H_G = a(MLP(H_node)) ⊙ MLP(H_node)
13     return H_G
14
15 Define ComputeGFeatures(X, A_c, A_d, T):
16     H^0 = X × W + b
17     H_d = ComputeNodeFeatures(H^0, A_d, T)
18     H_c = ComputeNodeFeatures(H^0, A_c, T)
19     H_final = H_d + H_c
20     G_final = GraphPooling(H_final)
21     return G_final
22
23 Define Train(train_data, T):
24     while not_converged do
25         while data in train_data do
26             X_1 = data.X_1
27             X_2 = data.X_2
28             A_c1 = data.A_control1
29             A_d1 = data.A_data1
30             A_c2 = data.A_control2
31             A_d2 = data.A_data2
32             Y = data.Y
33             G_1 = ComputeGFeatures(X_1, A_c1, A_d1, T)
34             G_2 = ComputeGFeatures(X_2, A_c2, A_d2, T)
35             featureRep = CONCAT(G_1, G_2)
36             featureRep = a(MLP(featureRep))
37             similarity = a(MLP(featureRep))
38             loss = LOSS(similarity, Y)
39             Update_Optimizer(loss)
```

here it should be noted that the attention is not computed for each path. We are only computing attention scores for neighboring nodes. Since the node representation keeps getting updated based on the graph topology, the path information also gets embedded in the learned node representation.

We have used two attention modules to learn node representation. The output of attention module 1 with eight different attention heads is shown by Equation 4.4.

$$h_i^{'} = \|_{k=1}^{8} \sigma \left(\sum_{j \in N(i)} \alpha_{ij}^{k} {W'}^{k} h_j^{0} \right) \tag{4.4}$$

Where $h_i^{'}$ denotes the intermediate representation after the first attention block, α denotes the corresponding attention scores, σ is the sigmoid activation function, W' are the weight parameters in the first attention block, and $\|$ represents the concatenation of the attention coefficients from eight different heads. Equation 4.5 represents the output of attention module 2. Here, we have aggregated the output from different attention nodes.

$$h_i^{''} = \sigma \left(\sum_{k=1}^{6} \sum_{j \in N(i)} \beta_{ij}^{k} {W''}^{k} h_j^{'} \right) \tag{4.5}$$

Here, β represents the attention scores computed for attention module 2, W'' are the weight parameters in the second attention block, and $h_i^{''}$ represents the learned node features at the output of the second attention block.

This learned node representation $h_i^{''}$ is then passed as an input to the Long Short Term Memory (LSTM) module similar to [152]. The output of the LSTM module is again fed back to the attention block as input as shown in Figure 4.7. The whole process described above is repeated T times to update node representations with the information from its Tth hop neighbors.

The T propagation rounds helps us to capture the semantic context through the graph's structural information and the learnable attention-based weights. Lines 5-7 of Function $ComputeNodeFeatures$ defined in Algorithm 1 presents the above exposition. The parameter sets ϕ_1 and ϕ_2 comprise all the first and second attention blocks' parameters, respectively.

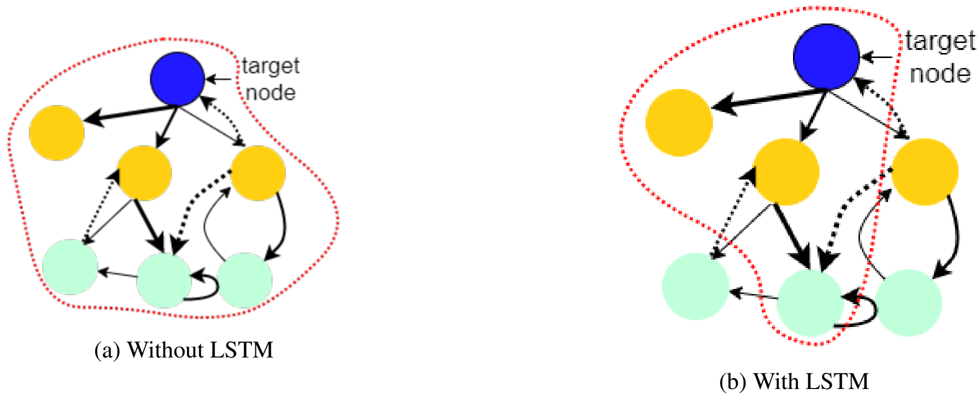


Figure 4.8: A synthetic example showing explored receptive path (area covered by the dotted red line) for the target node. The edge thickness denotes the received attention scores while learning features for the target node. Control and data-dependent edges are shown through solid and dotted edges, respectively.

4.3.2 LSTM-based Local Context Learning.

The attention module described above covers different hop neighborhoods of a PDG, thus capturing node's broader semantic context. This context (i.e., the receptive field) of each node in the graph is equivalent to a subgraph that consists of nodes along paths to the target node, as shown in Figure 4.8a. But as we have seen earlier, not all paths are important and meaningful while learning node representation. Thus, we employed an LSTM module to automatically choose the receptive field for a node while adaptively exploring the depth and breadth of the subgraph.

The LSTM module filters nodes from multiple propagation rounds. So, in a way, attention explores the graph's depth, while LSTMs explore the breadth. For example, in Figure 4.8b, we can see that the explored receptive path has been reduced while learning the target node's representation. This happens due to the forget gate of the LSTM, which filters out the information received from the different hop neighbors over multiple propagation rounds.

We randomly initialized the hidden and memory state of the LSTM module for each graph node (Note: This was a one-time initialization). Now during each propagation round, once we get the learned representation from the attention module, we pass it as an input with hidden states and memory states to the LSTM module. The LSTM module updates the cell state and hidden state using the input, forget, and output gates. Thus, when we receive information from the different hop neighbors at each propagation round, this whole process is repeated. In this

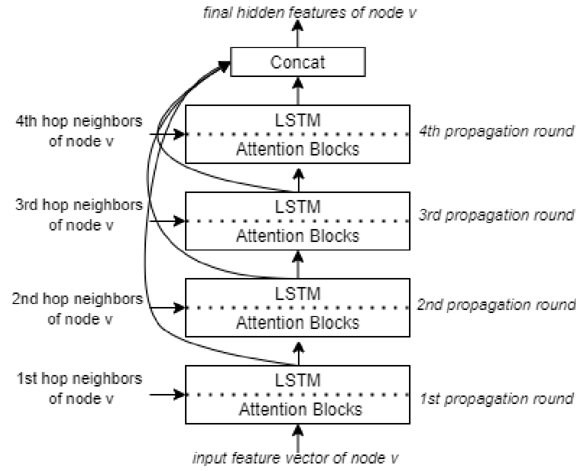


Figure 4.9: Visualization of the 4-layer architecture of JK networks in HOLMES. In each propagation round t, the feature vector of node v is combined with its tth order neighbors. During the final layer (4th propagation round), all hidden feature vectors from previous rounds are concatenated to form the ultimate hidden representation for node v. This concatenation preserves features learned from n^{th} hop neighbors across various propagation rounds, ensuring their presence and impact in the final hidden representation of node v.

way, LSTM filters information from multiple hop neighbors exploring the graph's depth.

4.3.3 PDG Representation Learning with Jumping Knowledge Networks and Soft Attention

To learn high-level program features from the PDGs, our GNN model iteratively aggregates the node features from different n^{th} hop neighbors via a message-passing scheme, described in Sections 4.3.1 and 4.3.2.

In order to learn the diverse and locally varying graph structure and the relations between program statements effectively, a broader context is needed, i.e., the GNN model should explore the deeper neighborhood. We observed that, while aggregating features from the different n^{th} hop neighbors (going till depth 4 – i.e. $n \in 1, 2, 3, 4$), our GNN model's performance degrades (also shown in Figure 4.17).

Thus, to stabilize the training and to learn the diverse local neighborhood for each node, we employed Jumping Knowledge Networks (JK) [153], shown in Figure 4.9. JK combines (concatenates; denoted by $\|$ operator below) the learned node embedding matrix ($H^{(t)}$ defined

on Line 7 from different propagation rounds as:

$$H_F = \left[H^0 \| H^{(1)} \| \cdots \| H^{(T)}\right] \tag{4.6}$$

Line 8 of Function $ComputeNodeFeatures$ defined in Algorithm 1 conveys the above description.

For obtaining graph level representation from the learned node embedding matrix, we have employed a soft attention mechanism proposed by Li et al. [138]:

$$H_G = \left(\sum_{i \in V} \sigma\big(MLP(h_i^{(T)})\big) \odot MLP(h_i^{(T)}) \right) \tag{4.7}$$

where T denotes the T rounds of propagation and $\sigma\left(MLP(h_i^{(T)})\right)$ computes the attention scores. The attention scores act as a filtering mechanism that helps to pull out irrelevant information. The Function $GraphPooling$ defined in Algorithm 1 shows the above exposition.

4.3.4 Edge-Attributed PDG Representation Learning

PDGs use data dependence and control dependence edges to capture the syntactic and semantic relationships between different program statements. Control dependence edges encode program structure while data dependence edges encode the semantics of the program.

Hence, to leverage the available syntactic and semantic information more effectively, we propose to learn program representations corresponding to each edge type. Therefore, given a program dependence graph G, we will learn two separate node feature matrix H_{data} and $H_{control}$. H_{data} represents the learned node feature matrix corresponding to the subgraph of G induced by data dependence edges, and $H_{control}$ represents the learned node feature matrices corresponding to the subgraph induced by control dependence edges in G. Next, to obtain the

final representation for the nodes in G, we do the vertex-wise addition of H_{data} and $H_{control}$.

$$\left.\begin{aligned} H_{data} &= ComputeNodeFeatures(H_0, A_d, T) \\ H_{control} &= ComputeNodeFeatures(H_0, A_c, T) \\ H_F &= H_{data} + H_{control} \end{aligned}\right\} \tag{4.8}$$

Thereafter, to obtain the final graph representation G_{final} we applied graph pooling defined in Section 4.3.3 on the learned node feature matrix H_{final}.

$$G_{final} = GraphPooling(H_F) \tag{4.9}$$

4.3.5 Implementation and Comparative Evaluation.

We have used the Soot optimization framework [154] to build the PDGs. To compute the control dependence graph, we first build a control flow graph. Then Cytron's method [155] is used to compute control dependence. For computing data dependence graph, reaching definition [156] and upward exposed analysis [157] is used.

We have used the PyTorch geometric [158] deep learning library to implement HOLMES. All LSTMs have a single LSTM layer with 100 hidden units. We have used the LeakyReLU [159] as the non-linear activation function with a negative slope of 0.02 and a sigmoid layer at the output for classification. The network is initialized using the Kaiming Uniform method [160]. The Siamese network is trained using Adam [161] optimizer with a learning rate set to 0.0002 and batch size to 50 to minimize binary cross-entropy loss given in Equation 5.6.

$$BCELoss = -\frac{1}{N}\sum_{i=1}^{N} y_i \times log(p(y_i)) + (1 - y_i) \times log(1 - p(y_i)) \tag{4.10}$$

Where y_i denotes the true binary label, and $p(y_i)$ denotes predicted probability (similarity score). N is the number of samples in the dataset. The output of Algorithm 1 is the similarity score. To determine the decision threshold (ϵ), we employed a threshold moving approach. We first predicted the probability for each sample on the validation set and then converted the probabilities

into the class label by varying ϵ from $[0.2 - 0.8]$ with the step size of 0.1. We evaluated the class labels on each threshold value in the range and selected ϵ on which we got the maximum F1-score on the validation set. This threshold was then used to evaluate the samples in the test set and is also used in the experiments defined in 4.4.2.2

For the BigCloneBench (BCB) dataset since it does not provide the input files' dependency information, we used JCoffee [162] to infer missing dependencies to generate PDGs. JCoffee infers missing dependencies based on the compiler's feedback in an iterative process. With JCoffee, we successfully compiled 90% of the snippets from the BCB dataset. In our experiments, we compared HOLMES with the state-of-the-art code clone detection tool TBCCD [43] as other recent machine-learning-based code clone detection tools namely, CDLH [45] and CDPU [47] do not have their implementations available open-source. DeepSim [44] does not provide implementation details of the semantic feature matrix construction. Thus, we could not replicate their experimental settings and hence do not perform a comparative evaluation with these approaches. Moreover, we did not compare HOLMES with Deckard[29], RtvNN [46] and Sourcerer [99] as TBCCD significantly outperformed these approaches. Therefore, TBCCD became our natural choice for comparative evaluation.

Baseline Model. We also compared our proposed approach with a baseline model GGNN [138]. GGNN is another variant of GNN that updates the node's representation simply by adding the neighboring nodes' information. This comparison's main objective is to understand the effectiveness of exploring the depth and breadth of the graph while learning node representations compared to the simple addition of information from the neighborhood.

4.4 Experimental Design

This section details the comprehensive evaluation of HOLMES. Specifically, we aim to answer the following research questions (RQs):

RQ 4.1: How effective is HOLMES as compared to other state-of-the-art approaches?

RQ 4.2: How well does HOLMES generalizes on unseen projects and data sets?

Table 4.1: Dataset Statistics.

Dataset	Project Files	Clone Pairs	Non-clone Pairs	No. of Methods	LOC
GCJ	$9.4k$	$440k$	$500k$	$58k$	$909k$
SeSaMe	11 projects	93	n.a.	419	$180k$
BCB	$9.1k$	$650k$	$650k$	$9.1k$	$267k$

Table 4.2: Percentage of clone-types in BigCloneBench.

Clone Type	T1	T2	ST3	MT3	WT3/T4
Percentage(%)	0.005	0.001	0.002	0.010	0.982

4.4.1 Datasets

Our experiments make use of the following datasets to evaluate the effectiveness of HOLMES:

1) Programming Competition Dataset: We followed the recent work [44] and used code submissions from GoogleCodeJam (GCJ). GCJ is an annual programming competition hosted by Google. GCJ provides several programming problems that participants solve. The participants then submit their solutions to Google for testing. The solutions that pass all the test cases are published online. Each competition consists of several rounds.

However, unlike the recent work [44] that used 12 different functionalities in their experiments, we collected 9436 solutions from 100 different functionalities from GCJ. [2] Thus, building a large and representative dataset for evaluation. Detailed statistics are reported in Table 6.1a. Programmers implement solutions to each problem, and Google verifies the correctness of each submitted solution. All 100 problems are different, and solutions for the same problems are functionally similar (i.e., belonging to Type 3 and Type 4 clone category) while for different problems, they are dissimilar.

2) Open Source Projects: We experimented with several open-source real-world projects to show the effectiveness of HOLMES's learned representations.

[2] https://code.google.com/codejam/past-contests

a) SeSaMe dataset. SeSaMe [163] dataset consists of semantically similar method pairs mined from 11 open-source Java repositories. The authors applied text similarity measures on Java doc comments mined from these open source projects. The results were then manually inspected and evaluated. This dataset reports 857 manually classified pairs validated by eight judges. The pairs were distributed in a way that three judges evaluated each pair. The authors have reported semantic similarity between pairs on three scales: `goals`, `operations`, and `effects`. The judges had the option to choose whether they `agree`, `conditionally agree`, or `disagree` with confidence levels `high`, `medium`, and `low`.

b) BigCloneBench dataset. BigCloneBench (BCB) [164] dataset, released by Svajlenko et al., was developed from the IJAdataset-2.0[3]. IJAdataset contains $25K$ open-source Java projects and $365M$ lines of code. The authors have built the BCB dataset from IJaDataset by mining frequently used functionalities, such as bubble sort. The initial release of the BCB dataset covers ten functionalities, including $6M$ clone pairs and $260K$ non-clone pairs. The current release of the BCB dataset has about $8M$ clone pairs covering 43 functionalities. Some recent code clone detection tools TBCCD [43], CDLH [45] has used the initial version of the BCB covering ten functionalities for their experiments. Hence, to present a fair comparison with TBCCD, we have also used the same version.

BCB dataset has categorized clone types into five categories: Type-1, Type-2, Strongly Type-3, Moderately Type-3, and Weakly Type-3+4 (Type-4) clones. Since there was no consensus on minimum similarity for Type-3 clones and it was difficult to separate Type-3 and Type-4 clones, the BCB creators categorized Type-3 and Type-4 clones based on their syntactic similarity. Thus, Strongly Type-3 clones have at least 70% similarity at the statement level. These clone pairs are very similar and contain some statement-level differences. The clone pairs in the Moderately Type-3 category share at least half of their syntax but contain a significant amount of statement-level differences. The Weakly Type-3+4 code clone category contains pairs that share less than 50% of their syntax. Tables 6.1a and 4.2 summarize the data distribution of the BCB

[3]https://sites.google.com/site/asegsecold/projects/seclone

dataset.

4.4.2 Experimental Procedure and Analysis.

4.4.2.1 RQ 4.1: How effective is HOLMES as compared to other state-of-the-art approaches?

To answer this RQ, we compared two variants of HOLMES with TBCCD [43], a state-of-the-art clone detector that uses ASTs and tree-based convolutions to measure code similarity. We followed similar experimental settings as used by Yu et al. in TBCCD. We used datasets from GCJ and BCB. We reserved 30% of the dataset for testing, and the rest we used for training and validation. For the BCB dataset, we used the same code fragments from the related work [43, 45]. We used around $700K$ code pairs for training. For validation and testing, we used $300K$ code clone pairs each. For the GCJ dataset, we had $440K$ clone pairs and $44M$ non-clone pairs. Due to the combinative nature of clones and non-clones, non-clone pairs rapidly outnumber the clone pairs. To deal with this imbalance in clone classes, we performed downsampling for non-clone pairs using a reservoir sampling approach. This gives us $500K$ non-clone pairs and $440K$ clone pairs. We evaluated the following variants of HOLMES against TBCCD:

1) Edge-Unified HOLMES (EU-HOLMES): We did not differentiate between the control and data-dependent edges in this variant to learn the program features.

2) Edge-Attributed HOLMES (EA-HOLMES): PDGs model control and data flow explicitly. Hence, it is logical to leverage this information as well while learning node representations. To model edge attributes, we have learned different program representations for data-dependent edges (G_{data}) and control-dependent edges ($G_{control}$) and aggregated them to obtain the final graph representation (G_{final}).

4.4.2.2 RQ 4.2: How well does HOLMES generalizes on unseen projects and data sets?

To evaluate the robustness and generalizability of EU-HOLMES and EA-HOLMES, we performed the following experiments on unseen projects and datasets:

1) Experiments on unseen GoogleCodeJam (GCJ*) projects: For this experiment, we reserved four projects from the GCJ dataset for testing, where each project had approx 100 submissions and the rest 96 projects we used for training. From the reserved projects, we got $17k$ clone pairs and $52k$ non-clone pairs for testing. We followed the similar training strategy defined in RQ 4.1 for this experiment.

2) Experiments on unseen BigCloneBench (BCB*) functionalities: BCB dataset contains clones and non-clones pairs from different functionalities. Since most of the pairs reported in the dataset are from functionality 4 (approx 80%), we reserved the pairs from this functionality for training purposes and the rest for evaluation.

3) Experiments on SeSaMe Dataset: This dataset [163] has reported 857 semantically similar clone pairs from 11 open-source Java projects. However, of the 11 projects, we were able to compile only eight projects, which gave us 93 clone pairs for evaluation.

4.5 Results

The results of our comprehensive evaluation are summarized in this section. The color coding indicates that the darker the color's hue better the results.

4.5.1 RQ 4.1: How Effective is HOLMES Compared to Other State-of-the-Art Approaches?

To answer this RQ, we compared our proposed approach variants EU-HOLMES and EA-HOLMES with two variants of TBCCD- (1) TBCCD (-token), and (2) TBCCD, as reported in [43]. The variant TBCCD (-token) uses randomly initialized AST node embeddings in place of source

Table 4.3: Comparative evaluation with GGNN and TBCCD variants.

Tool	No. of Params	BCB			GCJ		
		P	R	F1	P	R	F1
TBCCD (-token)	210K	77	73	74	77	80	80
TBCCD	170k	96	96	96	79	85	82
GGNN	280k	72	89	79	72	87	79
EU-HOLMES	1.7M	72	97	83	84	92	88
EA-HOLMES	6.6M	**97**	**98**	**98**	**91**	**93**	**92**

Table 4.4: F1 value comparison w.r.t various clones types in BigCloneBench dataset.

Tool	T1	T2	ST3	MT3	WT3/T4
TBCCD (-token)	**100**	90	80	65	60
TBCCD	**100**	**100**	98	96	96
GGNN	**100**	91	79	70	60
EU-HOLMES	**100**	**100**	86	80	80
EA-HOLMES	**100**	**100**	**99**	**99**	**99**

code tokens, which are fine-tuned during training. The second variant, TBCCD, uses the token-enhanced AST and PACE embedding technique. The token-enhanced AST contains source code tokens such as constants, identifiers, strings, and special symbols. Table 4.3 shows the comparative evaluation of EU-HOLMES and EA-HOLMES with TBCCD (-token) and TBCCD on the BCB and GCJ datasets.

On the BCB dataset from Table 4.3, we can see that both TBCCD and EA-HOLMES perform equally well, while the performance of TBCCD (-token) drops significantly. The BCB dataset categorizes clones into five categories: Type-1 clones, Type-2 clones, Strongly Type-3 clones, Moderately Type-3 clones, and Weakly Type-3+4 (Type-4) clones. Since there was no consensus on minimum similarity for Type-3 clones, and it was difficult to separate Type-3 and Type-4 clones, the BCB creators categorized Type-3 and Type-4 clones based on their syntactic similarity. Thus, Strongly Type-3 clones have at least 70% similarity at the statement level. These clone pairs are very similar and contain some statement-level differences. The clone pairs in the Moderately Type-3 category share at least half of their syntax but contain a significant amount of statement-level differences. The Weakly Type-3+4 code clone category contains pairs that share less than 50% of their syntax. Table 4.4 further shows the performance of TBCCD (-token), TBCCD, GGNN, EA-HOLMES, and EU-HOLMES on different code clone types in the

```java
public void copyDirectory(File srcDir,
↪  File dstDir){
  if (srcDir.isDirectory()){
    if (!dstDir.exists())
      dstDir.mkdir();
    String[] children = srcDir.list();
    for (int i = 0; i < children.length;
    ↪  i++) {
      copyDirectory(new File(srcDir,
      ↪  children[i]),
      new File(dstDir, children[i]));
    }}else{
      copyFile(srcDir, dstDir);
    }
}
```

```java
public static void copy(File src, File
↪  dst){
  if (src.isDirectory()) {
  String[] srcChildren = src.list();
  for (int i = 0; i <
  ↪  srcChildren.length; ++i) {
    File srcChild = new File(src,
    ↪  srcChildren[i]);
    File dstChild = new File(dst,
    ↪  srcChildren[i]);
    copy(srcChild, dstChild);
    }
  }else
    transferData(src, dst);
}
```

Figure 4.10: A WT3/T4 clone example from BCB dataset. The code snippets are implementing the functionality for copying the directory and its content. Although the snippets are reported under WT3/T4 clone category they are syntactically similar with some differences in the sequence of invoked methods and API calls.

BCB dataset. All the approaches achieve good performance on Type-1 and Type-2 code clone categories, as these code clone types are easier to detect. While on the hard-to-detect code clone categories, viz., Moderately Type-3 and Weakly Type-3+4, TBCCD (-token) and GGNN perform poorly compared to the TBCCD, we also see an improvement of ~3% in EA-HOLMES as compared to TBCCD.

On the other hand, the performance of both the TBCCD's variants, i.e., TBCCD (-token) and TBCCD, as well as GGNN, drops significantly on the GCJ dataset. We can see an improvement of ~10% in the F1-score on the GCJ dataset in EA-HOLMES compared to TBCCD.

The varying performance of TBCCD on the BCB and GCJ datasets is attributed to the following reasons: (1) There is a high syntactic similarity between the code snippets of the BCB dataset. For instance, Figure 4.10 shows a WT3/T4 code snippet from the BCB dataset having variations in the sequence of invoked methods and API calls only. This syntactic similarity existing in the form of identifiers, tokens, etc., is exploited by TBCCD while learning for code similarity. (2) As opposed to the BCB dataset, where there is some syntactic similarity between the code pairs, the GCJ code clone pairs have substantial differences in structure and algorithm [44, 165]. These differences are not unexpected because the submissions are made by independent programmers implementing the solutions from scratch. Consequently, without modeling semantics (TBCCD uses ASTs), the GCJ dataset's clones are harder to detect compared to BCB. (3) In the BCB dataset, there are ten functionalities and a large number of examples

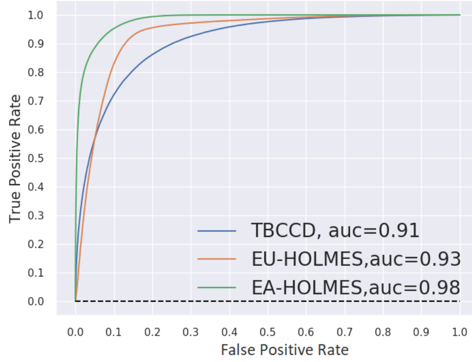


Figure 4.11: ROC curve of TBCCD, EU-HOLMES, EA-HOLMES on GCJ dataset (AUC values are rounded up to 2 decimal places).

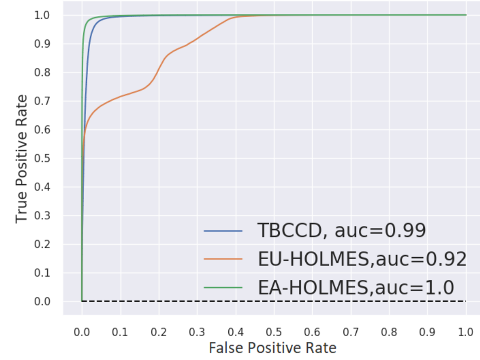


Figure 4.12: ROC curve of TBCCD, EU-HOLMES, EA-HOLMES on BCB dataset (AUC values are rounded up to 2 decimal places).

corresponding to each of the ten functionalities, thus making it easier for the deep learning models to learn similarities and differences. On the other hand, the GCJ dataset has 100 functionalities, each with 100 examples only, which makes it a much more challenging dataset than BCB. The GCJ dataset, in fact, represents a more realistic scenario as it is difficult to obtain a large number of examples corresponding to each category without inducing minor syntactic variations (as shown in Listing Figure 4.10).

Also, from Tables 4.3 and 4.4, we observe that EA-HOLMES performs better than EU-HOLMES on GCJ and BCB datasets in every evaluation metric. This performance difference demonstrates the importance of structured semantics of source code while learning code functional similarity. To further assert the confidence in the achieved performance of EA-HOLMES as compared to TBCCD, we performed Bootstrap hypothesis testing [166]. We sampled $300K$ code pairs from the test set of the BCB dataset and $200K$ pairs from the test set of the GCJ dataset using random sampling with replacement strategy. We used $100K$ bootstrap iterations to compute the Achieved Significance Level (ASL) value. We achieved ASL < 0.01 on both the datasets, which shows with a strong evidence that there exists a significant statistical difference between TBCCD and EA-HOLMES.

Additionally, to analyze the diagnostic ability of TBCCD, EU-HOLMES, and EA-HOLMES, we plotted the Receiver Operating Characteristics (ROC) curve by varying the classification threshold. Figures 4.11 and 4.12 show the ROC curve and corresponding Area Under Curve

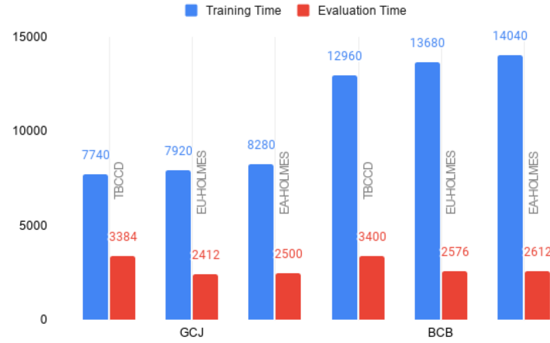


Figure 4.13: Time performance analysis on the GCJ dataset.

(AUC) values for the GCJ and BCB datasets. We have plotted the ROC curve of the TBCCD variant only, as it has outperformed the TBCCD (-token) variant and GGNN on the GCJ and BCB datasets. For all other experiments also, we have used the TBCCD variant only.

ROC curve is a graphical plot, visualizing trade-off between True Positive Rate (TPR) plotted on the y-axis and False Positive Rate (FPR) plotted on the x-axis. AUC metric measures the degree of separability. Generally, an excellent classifier has a high AUC, denoting the model is better at predicting clone pairs as clones and non-clone pairs as non-clones. We can see from Figures 4.11 and 4.12 that EA-HOLMES has the best AUC value on both the datasets.

Time Performance. We evaluated the time performance of TBCCD, EU-HOLMES, and EA-HOLMES on two parameters $-(1)$ time taken to build ASTs vs. time taken to build PDGs, and (2) total training and evaluation time. We run each of these tools with the same parameter settings reported in Section 4.3.5 on the full GCJ and BCB dataset on a Workstation with an Intel Xeon(R) processor having 24 CPU cores. We have used a GeForce RTX 2080Ti GPU with $11GB$ of GPU memory.

The total time taken to build the ASTs for 9436 project files was 30 minutes, while building the PDGs took 60 minutes. Figure 4.13 shows the training and evaluation time analysis of TBCCD, EU-HOLMES, and EA-HOLMES on GCJ and BCB datasets. On the BCB dataset, EA-HOLMES, EU-HOLMES, and TBCCD took more time as it has many clone/non-clone pairs as compared to the GCJ dataset.

Also, EA-HOLMES learns separate representations for the control and data dependence

Table 4.5: Performance on GCJ* and SeSaMe dataset.

Tool	GCJ*			SeSaMe		
	P	R	F1	P	R	F1
TBCCD	72	75	73	**100**	48	64
EU-Holmes	79	80	79	**100**	52	68
EA-Holmes	**84**	**85**	**85**	**100**	**85**	**92**

Table 4.6: F1 value comparison w.r.t various clones types in BCB*.

Tools	T1	T2	ST3	MT3	WT3/T4
TBCCD	**99**	**99**	96	95	95
EU-Holmes	**99**	**99**	83	79	79
EA-Holmes	**100**	**100**	**99**	**99**	**99**

graphs; it takes more training time than EU-HOLMES and TBCCD. Even though the total time taken to build PDGs is greater than building ASTs and EA-HOLMES takes longer training time, these are one-time offline processes. Once a model is trained, it can be reused to detect code clones.

4.5.2 RQ 4.2: How Well Does HOLMES Generalize on Unseen Projects and Datasets?

Experiment on GCJ* and SeSaMe dataset: Table 4.5 shows the performance of TBCCD, EA-HOLMES and EU-HOLMES on unseen projects from GCJ* dataset and SeSaMe dataset. From Table 4.5, we can see that EA-HOLMES outperforms TBCCD on both datasets, and the performance is consistent with the results reported in Table 4.3.

Experiment on BCB* dataset: Table 4.6 shows the performance of TBCCD, EU-HOLMES, and EA-HOLMES on the BCB* dataset. The results are consistent with the results reported in Tables 4.3 and 4.4. These results affirm that EA-HOLMES generalizes on unseen datasets also.

4.6 Discussion

4.6.1 Why HOLMES Outperforms Other State-of-the-art clone detectors.

Our approach uses PDGs for representation learning. PDGs represent a program's semantics through data dependence and control dependence edges. Our approach models the relations between the program statements in the PDGs using GNNs. We use an attention-based GNN and an LSTM module to filter and aggregate relevant paths in PDGs that enable us to learn semantically meaningful program representations. Attention-based GNNs draw importance to different direct (one-hop) neighbors, while LSTMs capture broader context and long-range dependencies between nodes of PDGs.

Thus, representing source code as graphs and modeling them through GNNs and LSTMs help us to leverage the program's structured semantics, contrary to using ASTs and token sequences for learning program features. Additionally, to give respective importance to control and data dependence relations between different statements, we learn two different representations corresponding to each edge type. This information helps us differentiate and prioritize the available semantic and syntactic relations between different program statements.

4.6.2 Representation learning using Graph Attention networks (GATs).

Though there are many graph feature learning layers in the literature, such as Graph Convolutional Networks (GCN), Gated Graph Neural Network (GGNN), our work uses the GAT layer to learn program dependency graph nodes' features. This is because the GAT layer can learn an adaptive receptive path for a node in a graph, i.e., assigning different importance to different nodes in the same neighborhood. On the other hand, the GCN layer has fixed receptive paths, which might not work well in our case as all paths in the PDG are not equally important.

Figure 4.14: t-SNE plot of graph embeddings of clone and non-clone pairs of GCJ dataset generated by EA-HOLMES.

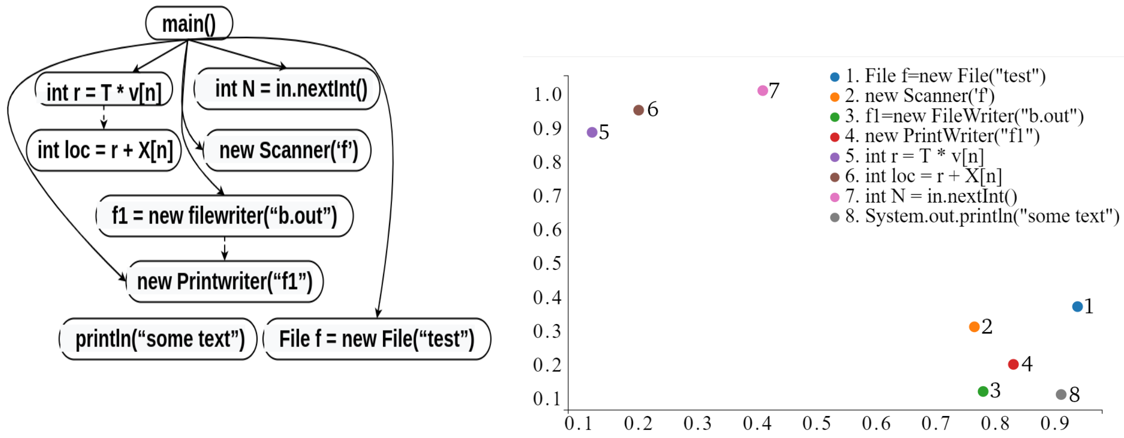
4.6.3 Qualitative Analysis of the Features Learned by EA-HOLMES.

We select ten random problems from the GCJ dataset and around 1000 code pairs from these problems to visualize the learned latent feature space of EA-HOLMES. We pass the 1000 sampled code pairs through the final layer of EA-HOLMES to obtain the latent representation and then visualize them using the t-SNE [167].

t-SNE visualizes high dimensional data in 2D or 3D space. It is a non-linear technique that tries to convert similarities between data points to joint probability and minimizes KL divergence distance between the joint probabilities of the low dimensional embedding and high dimensional data. Figure 4.14 shows the learned latent features by HOLMES. Here, the blue color points represent the non-clone class, and the other colors represent the ten randomly selected GCJ problems or clone pairs.

From the figure, we can see that the clone and non-clone classes are fairly separable, which implies that EA-HOLMES can differentiate between clone and non-clone pairs. A further interesting observation is the cluster formation of clone pairs. The clone pairs form class clusters, which shows that EA-HOLMES can implicitly learn separate representations for different classes without extra input. This indicates that we can also use EA-HOLMES for code classification tasks.

Further, to get more insights into the learned feature space of EA-HOLMES, we processed and extracted the node features and adjacency matrix from the PDG shown in Figure 4.15a. We



(a) PDG for a java source code. Solid line edges denote control dependence. Dashed line edges denote data dependence.

(b) t-SNE plot of input feature vectors generated by EA-HOLMES.

Figure 4.15: Qualitative analysis of the features learned by EA-HOLMES.

then pass them as an input to the first hidden layer of EA-HOLMES. Figure 4.15b shows the t-SNE visualization of the generated vector embedding.

From the figure, we can see that the statements that share similar semantics are plotted very closely. In contrast, the statements that are not similar are not close in the embedding space. For instance, the statements $int\ r = T * v[n]$ and $int\ loc = r + x[n]$ are plotted nearby, as the latter uses the former's result, and both are performing some numerical computation. Similarly, statements 1,2,3,4, and 8 are similar, thus plotted nearby in embedding space. These insights suggest that our approach models graph topology and node distribution simultaneously for learning the graph representation.

4.6.4 Ablation Study.

To understand the contribution of each component of our model, we perform an ablation study, and Figures 4.16 and 4.17 show the results. In the first set of experiments, we varied the number of attention heads of both the attention blocks while keeping the rest of the architecture the same. The left part of Figure 4.16 shows the results of varying the attention heads. From Figure 4.16, we can see that the performance of EA-HOLMES degrades on removing the second attention block. The F1-score also degrades further when we reduce the number of attention heads.

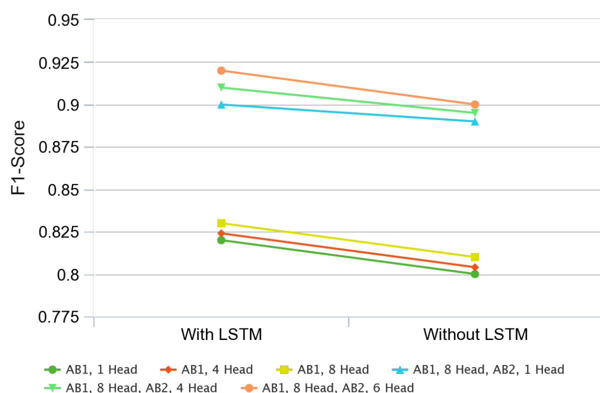


Figure 4.16: The effect on the performance of EA-HOLMES after varying the number of attention heads in both the attention blocks and after removing the LSTM layer. AB in the legend stands for Attention Block, for instance, "AB1, 1 Head" corresponds to "Attention Block 1 and 1 attention head".

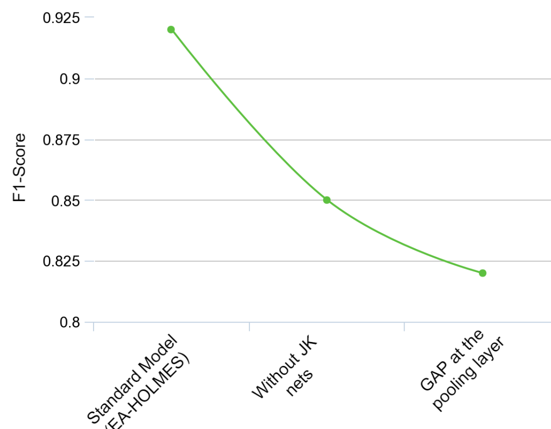


Figure 4.17: The effect on the performance of EA-HOLMES after removing Jumping Knowledge (JK) nets and soft attention mechanism from the graph readout layer.

Next, to examine the influence of LSTM on the model's performance, we remove the LSTM module from the EA-HOLMES architecture and vary the attention heads of both the attention blocks. The right part of Figure 4.16 shows the results. We can see that after removing the LSTM module from EA-HOLMES architecture, the performance degrades. This shows the importance of using LSTMs in aggregating the local neighborhood for learning node representation.

We then remove the Jumping Knowledge (JK) networks from the EA-HOLMES architecture in the next set of experiments. The results in Figure 4.17 show that the F1-score reduces drastically after removing JK nets. Thus, we can say that the JK nets help to stabilize the model and also improve the performance of HOLMES. In the end, we replace the soft attention mechanism with the Global Average Pooling (GAP) layer to learn graph representation. GAP layer simply averages all the learned node representations to make up the final hidden graph representation. From Figure 4.17, we can observe that the performance of EA-HOLMES degrades when we employ the GAP layer in place of the soft attention mechanism. This shows the importance of the soft attention mechanism at the graph readout layer.

4.6.5 Limitations of HOLMES.

We use PDG representation of the source code to learn program features. Static analysis is required to generate PDGs, and it only works for compilable code snippets. Therefore, we cannot directly apply our technique to incomplete programs. For this reason, we use JCoffee [162] to infer the missing dependencies in the BCB dataset. Besides, we use a supervised learning approach to learn code similarity, which is expensive in terms of labeled dataset procurement. However, as shown in the results, our model can learn a generalized representation and perform satisfactorily on unseen datasets. In our future work, we plan to extend the applicability of our model on unseen and unlabeled datasets by using techniques such as domain adaptation and transfer learning.

4.7 Chapter Summary

There has been a significant interest in detecting duplicated code fragments due to their pertinent role in software maintenance and evolution. Multitudinous approaches have been proposed to detect code clones. However, only a few of them can detect semantic clones. The proposed approaches use syntactic and lexical features to measure code functional similarity. They do not fully capitalize on the available structured semantics of the source code to measure code similarity. In this chapter, we have proposed a new tool HOLMES for detecting semantic clones by leveraging the semantic and syntactic information available with the program dependence graphs (PDGs). Our approach uses a graph-based neural network to learn program structure and semantics. We have proposed to learn different representations corresponding to each edge type in PDGs.

We have evaluated both variants of HOLMES on two large datasets of functionally similar code snippets and with the recent state-of-the-art clone detection tool TBCCD [43]. Our comprehensive evaluation shows that HOLMES can accurately detect semantic clones, and it significantly outperforms TBCCD, a state-of-the-art code clone detection tool. Our results show that HOLMES significantly outperforms TBCCD showing its effectiveness and generalizing capabilities on

unseen datasets. In the future, we would like to explore the combination of PDG with other program structures like token sequences for learning program representation. We would also like to examine the feasibility of the proposed approach in cross-language clone detection.

Chapter 5

Improving Cross-Language Code Clone Detection via Code Representation Learning and Graph Neural Networks

5.1 Introduction

Cross-language code clones refer to similar or identical segments of code that exist in different programming languages.[1] This can occur when code is manually translated from one language to another, or when a common algorithm or function is implemented in multiple languages.

Past efforts focused on developing single-language clone detection techniques, which relies on a program's syntactic structure or semantics to measure code similarity. These techniques, tailored to a specific language, utilize its unique syntax and semantics, making them effective for single-language detection due to shared syntactic and semantic characteristics.

However, adapting these techniques to cross-language clone detection is challenging. This adaptation would require parsers or scripts capable of analyzing syntax from various languages. The challenge arises from the different syntactic and semantic features and programming paradigms supported by different languages. For example, Deepsim [108] proposed encod-

[1] This chapter is a reproduction of a paper published at IEEE Transactions of Software Engineering, 2023. [168]

ing variable features into a 19-dimensional feature vector, but not all languages share the same data types or modifiers, making it difficult to identify clones across languages.

While single-language techniques excel within one language, their effectiveness may decline when detecting clones across languages due to their inability to handle the complexities of multiple languages. Thus, there's a growing need for techniques that can detect clones across multiple languages, essential for tasks like bug detection and program refactoring in multi-language software systems. Therefore, the techniques should strive to achieve a balance between syntax, semantics, and structure of programs to effectively detect similarities.

Techniques that use syntactic information from a common intermediate language like the .NET framework have been proposed for cross-language clone detection. However, these methods falter with the rise of languages not sharing a common intermediate representation. Language-agnostic techniques, which utilize revision histories or software quality metrics, and learning-based techniques using traditional neural networks like LSTMs have also been proposed. But these methods typically do not factor in semantic and syntactic information, or overlook structured program information when measuring code similarity.

Thus, to overcome the aforementioned challenges, we propose a semi-supervised learning-based cross-language code clone detection tool, RUBHUS. RUBHUS uses ASTs enriched with semantic information from source code and GNNs to model similarity between code snippets. RUBHUS extracts feature from code ASTs and learn a high-level program representation using GNNs. Specifically, to improve the learned local AST node representations while capturing various global latent structures and sub-structures, we use unsupervised learning based on the principle of Mutual Information (MI) maximization, as described in [169]. This approach allows us to encode different aspects of the data present at various AST sub-structures, which would have been missed otherwise. RUBHUS works for both statically and dynamically typed languages, as shown in Section 6.4. We validate the findings of our approach through experiments that assess the following:

1) Performance of RUBHUS in cross-language context using pairs from a mix of statically and dynamically typed languages – C & Java and Java & Python.

2) Generalizability of RUBHUS on unseen projects.

3) Performance of single language code clone detection tools in cross-language context.

4) Performance of RUBHUS in single-language context using statically and dynamically typed languages.

In the aforementioned settings, RUBHUS detects more clone and non-clone pairs with high F1 scores and outperforms the state-of-the-art clone detection tools. In summary, we make the following contributions:

1) We propose a new unsupervised deep learning framework for learning high-level program features from control and data flow-enriched ASTs (FE-ASTs). Our approach learns structured semantic and syntactic information available with the FE-ASTs using GNNs and the principle of mutual information maximization in an unsupervised way.

2) We propose a new supervised deep learning architecture for modeling similarity between code snippets. Our work jointly learns the high-level program features and similarities between snippets using a learning-based technique.

3) We develop a prototype tool, RUBHUS, and evaluate it under a number of different experimental settings, as reported in Section 5.4. The results of our experiments demonstrate that RUBHUS outperforms the state-of-the-art cross-language code clone detection tools and other baseline techniques.

4) We create a dataset of cross-language clones containing around 40K C files and 15K Java files. The dataset also has annotation information on clones and non-clones pairs.

5) An open source implementation of our tool RUBHUS.

6) We also present the first study to assess the performance of single-language code clone detection tools in the context of cross-language code clone detection tool.

```java
void main(String[] args){
    int N=scan.nextInt();
    int K=scan.nextInt();
    ArrayList ratings=new ArrayList();
    double rate=0;
    for(int i=0;i<N;i++)
        ratings.add(scan.nextInt());
    Collections.sort(ratings);
    for(int i=N-K;i<N;i++)
        rate=(rate+ratings.get(i))/2;
    System.out.println(rate);
}
```

Listing 5.1: solution.java

```python
def main():
    _input=input().split(' ')
    N=int(_input[0])
    K=int(_input[1])
    R=list(map(int,input().split()))
    R=sorted(R)
    ans=0
    for i in range(N-K,N):
        ans=(ans+R[i])/2
    print(ans)
```

Listing 5.2: solution.py

Figure 5.1: A cross-language Java-Python clone example detected by RUBHUS, which is reported as false negative by GRAPHCODEBERT.

5.2 Motivation

In this section, we present an example and our observations to motivate our approach.

5.2.1 Motivating Example

Listings 5.1 and 5.2 show Java and Python implementations of an Atcoder programming competition problem `Takahashi sort`. AtCoder website contains N video lectures, and Takahashi, a novice programmer, is allowed to watch K videos only. Each video at AtCoder has a rating that gets added to Takahashi's ability if he watches the video. Takahashi's initial ability is 0, and if he watches a video with a rating T, his ability is increased to $(Current\ Rating + T)/2$, provided T is higher than the current rating. The goal of the problem statement is to find the maximum ability that Takahashi can achieve by watching k videos from the pool of N videos.

Listings 5.1 and 5.2 implement `Takahashi sort` functionality, thereby being clones of each other. A quick manual inspection of the snippets is insufficient to infer that they perform similar functionality despite having some structural and semantic similarity etched between them. For example, both implementations follow a similar approach of sorting the rating array first and then selecting the K with the maximum rating. Moreover, both of them possess nodes corresponding to variable initialization, for loop, and an API call to sort an array. Further, some similar syntactic key node relationships are also preserved. This structural similarity is more

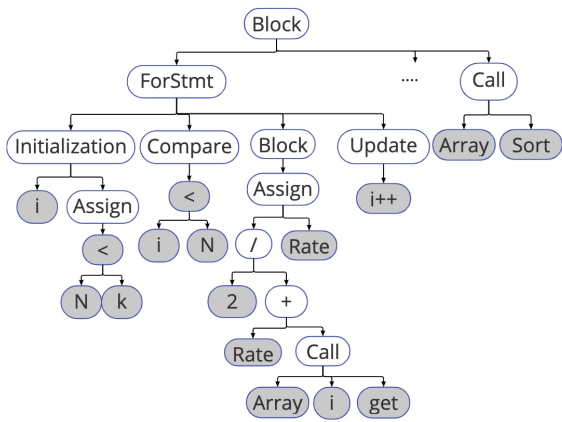


Figure 5.2: AST for lines $8-10$ of Listing 5.1.

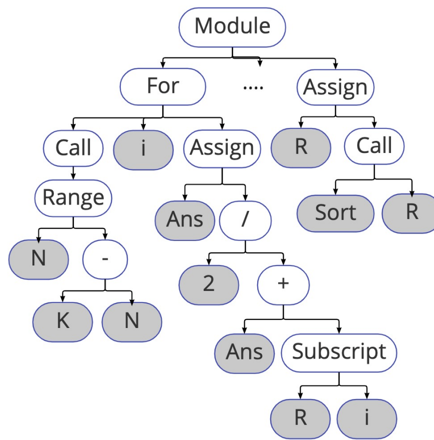


Figure 5.3: AST for lines $6-9$ of Listing 5.2.

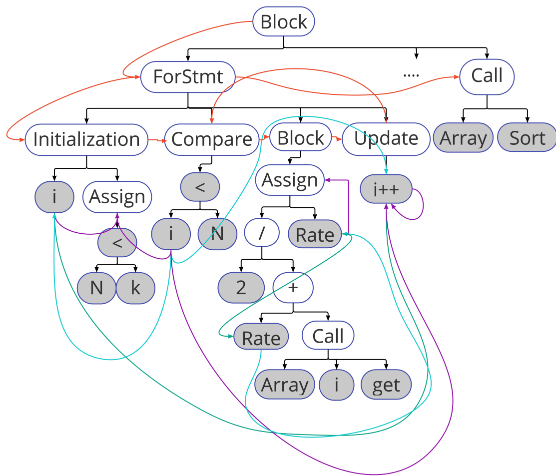


Figure 5.4: FE-AST for lines $8-10$ of Listing 5.1.

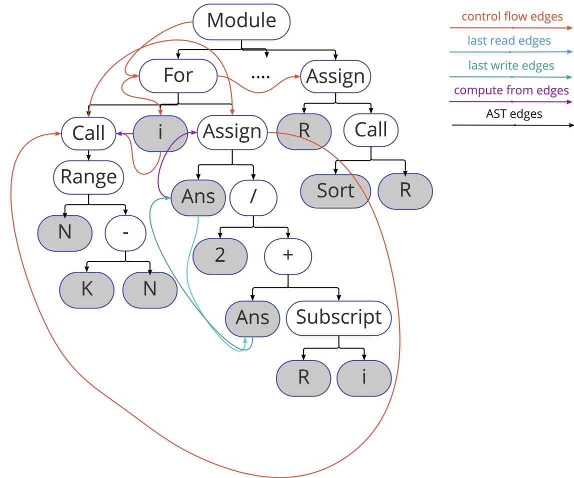


Figure 5.5: FE-AST for lines $6-9$ of Listing 5.2.

conspicuous if we look at the FE-AST representations of Listings 5.1 and 5.2, as shown in Figures 5.4 and 5.5 respectively. For instance, the FE-AST subtree of the second `for` loop has similar node dependencies across the two snippets.

The current single-language code clone detection techniques focus on capturing a specific type of information only for measuring similarity across code snippets. For instance, functional code clone detection tools make use of program dependence graphs (PDGs) to model program semantics, ignoring program structure and syntax [109] while some techniques [61] make use of syntactic constructs only like ASTs, code tokens, etc. These techniques can still be effective in detecting similarities across the same language due to shared language grammar, structure, and syntax. But cross-language code similarity detection poses many different and unique challenges, which makes it hard for these techniques to detect clones. For example, PDGs are felicitous

for learning semantic similarity between same-language code snippets. Still since they capture low-level fine granularity details, they might not be best suited for cross-language similarity detection. As can be observed in cross-language clones Listings 5.1 and 5.2, the contrasting control and data dependence for the two listings may make it hard for PDG-based tools to detect similarity between the two code snippets. Besides the mediocre textual similarity or AST-level similarity (shown in Figures 5.2 and 5.3) between the snippets will also make it difficult for the token-based or tree-based code clone detection tools to detect similarity between these two snippets. Thus, for cross-language code similarity detection, we need to find a golden mean between high-level and low-level program details that can expertly capture program syntactic, semantic, and structural information.

On the other hand, the state-of-the-art cross-language clone detection tools do not capture program semantic and syntactic information, which is crucial to detect similarity across cross-language clones. For example, the tools XCODE and C4 use only code tokens and ASTs while GRAPHCODEBERT proposes to use a specific kind of data flow relation "`where the value comes from`". These techniques do not expose the full semantics and syntax of a program, which is crucial for learning similarity across cross-language clones. If we refer to the Figures 5.2 and 5.3, which show the vanilla ASTs of Listings 5.1 and 5.2, the ASTs do not appear to be very similar. On the other hand, the FE-ASTs shown in Figures 5.4 and 5.5 share some similarities. The enriched AST nodes have similar control flow edges as well as similar last read and write edges making it easier to establish correspondence between nodes such as "Rate"(Listing 5.1) and "Ans" (Listing 5.2). Thus, FE-ASTs expose programs' structural, semantic, and syntactic information at a high-level, which makes it easier to model similarity across cross-language code snippets. Moreover, the current cross-language code clone detection techniques do not capture the long-range dependencies that exist between code statements. For example, the graph-guided masked attention proposed in GRAPHCODEBERT only considers direct edge relations. It does not incorporate information from its n-hop neighborhood or information that exists at various structures and sub-structures of AST. Thus, we infer that:

1) *To accurately measure cross-language similarity, syntactic and structural information should also be incorporated into the learned program features instead of relying only on shallow*

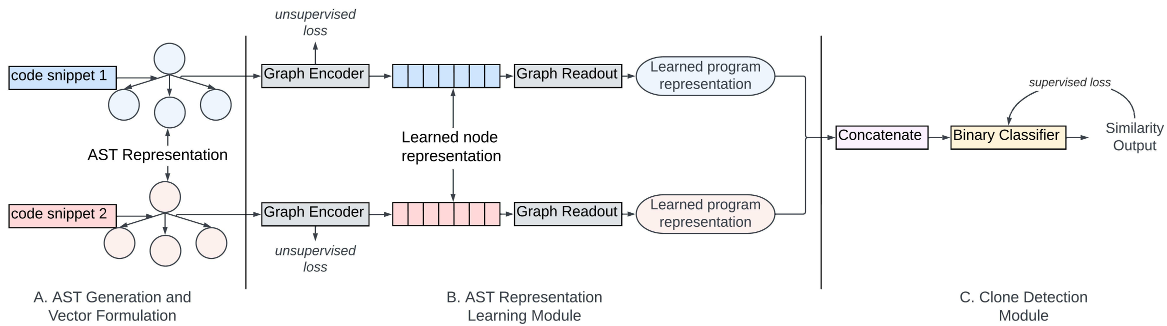


Figure 5.6: RUBHUS overview.

syntactic, lexical, and token-based similarity.

2) *Source code is a complex web of interacting code constructs. Understanding source code implies understanding the (syntactic and semantic) interactions between different code constructs. Thus, it necessitates exposing these complex interactions explicitly as a structured input to the neural network.*

5.2.2 Key Ideas

In consonance with the above observations, we have built RUBHUS, with the following key ideas:

1) To measure cross-language code similarity, we use AST program representation. ASTs are enriched with control and data flow edges (5.3.1) to capture semantic information from source code.

2) An inspection of ASTs in Figures 5.2 and 5.3 reveals similarities at structural, syntactic, and semantic-level. Traditional networks may overlook these patterns while learning for similarity, yet they might be essential in ascertaining code similarity. Thus, we employ GNNs to capture structural patterns while learning high-level program representation.

5.3 Approach Overview

In this section, we present the architectural design of RUBHUS. Figure 5.6 provides an overview of the pipeline, which comprises automated feature learning (semi-supervised using GNNs) and

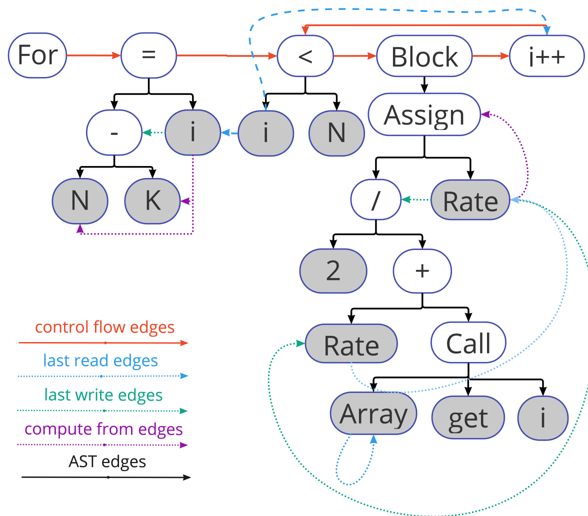


Figure 5.7: FE-AST for Lines 9 − 10 in Listing 5.1.

a binary classifier (supervised) for similarity prediction. The following subsections elaborate on these components in detail.

5.3.1 Flow-Enriched ASTs (FE-ASTs) Generation and Vector Formulation

Our work makes use of AST program representation. ASTs can be helpful in detecting syntactically similar programs. Nevertheless, programs having semantic similarities or significant structural differences cannot be detected by simply matching ASTs. Therefore, to enrich ASTs with more semantic information, we add additional *control and data flow* edges between AST nodes. We extend the graph with control flow edges to enforce the source code's execution order. To enrich ASTs with data-flow information, we add *LastRead* and *LastWrite* edges for all variables in the ASTs. We also add *ComputeFrom* edges whenever we observe an assignment statement. Finally, we introduce backward edges for all edge types to propagate information faster among the nodes. An example of this is shown in Figure 5.7.

FE-ASTs closely resemble the graphs proposed by Allamanis et al. [84], with some notable distinctions. Unlike the graphs proposed by Allamanis, FE-ASTs incorporate control flow edges, a feature absent in the former. In contrast, Allamanis et al. introduced NextToken edges connecting each AST token node to its successor to capture the flow of control through a program. Additionally, their graph includes additional edges like LastLexicalUse, ReturnsTo, and

FormalArgName, absent in FE-ASTs. The primary objective behind developing FE-ASTs was to strike a balance between granularity and abstraction. By avoiding overly detailed relationships, as proposed by Allamanis et al., FE-ASTs aim to establish a middle ground. This approach facilitates the detection of similarities across programming languages characterized by diverse structural, syntactic, and grammatical nuances.

Using ASTs represented by $G = (V, E, A, X)$, where V represents AST nodes and E, the list of edges, we extract adjacency matrix of G denoted by A, where $A \in \mathbb{R}^{|V| \times |V|}$ with $A_{ij} = 1$ if $e_{ij} \in E$, and $A_{ij} = 0$ if $e_{ij} \notin E$ and node feature matrix represented by X of dimension $\mathbb{R}^{|V| \times d}$, where $x_v \in \mathbb{R}^d$ denotes the feature vector of node v. The node feature vector x_v is a one-hot encoded vector mapping an AST node to its unique type. For instance, Java AST node vector will be a 71-d one hot encoded vector.

5.3.2 FE-AST Representation Learning Module

We design a semi-supervised GNN architecture for learning the high-level AST node features by exploiting structural and syntactic information. The model learns representation from the ASTs of both languages independently and concurrently. GNN model components are explained below.

5.3.2.1 Bi-GNN Architecture

Our architectural design, inspired by the Siamese neural network [76], has two parallel sub-networks for AST representation learning from the two code snippets. Contrary to Siamese networks, our architecture is not weight-shared due to the differences in the domains of the sub-networks we are dealing with (since the snippets are from different languages). To obtain initial feature vectors, we pass node feature matrix (X) through a linear transformation layer:

$$H^0 = X \times W + b, \tag{5.1}$$

where W and b are the learnable weight and bias vectors. $H^0 \in \mathbb{R}^{|V| \times d'}$ denotes the initial node embedding matrix where h_v^0 represents embedding vector for a single node $v \in V$. The initial linear layer is added to obtain sufficient expressive power to transform the input feature vector into high-level feature vector.

5.3.2.2 AST Encoder Module

Next, to obtain AST node representation, we use a graph encoder module to encode the AST node feature vector (h_v^0) into high-level feature representation $(h_v^{'})$. The graph encoder module learns an adaptive function $\varphi(A, H; \phi)$ parameterized by ϕ similar to GCN[75]. The module learns the node representation by *repeatedly aggregating* the node's local neighborhood features for K iterations. Formally, the kth layer of the encoder module is:

$$H^{(k)} = \sigma\big(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^{(k-1)}W^{(k-1)}\big), \tag{5.2}$$

where σ is the activation function, $\hat{A} = A + I$, I is the identity matrix, $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$, k denotes the kth layer of the GCN, and W is the learnable weight matrix.

The GCN module described above learns the node representation by adaptively exploring the depth of the AST. However, at a given depth, not all neighboring nodes are important, and thus, to adaptively explore the breadth of the AST, we employ an LSTM module at the end of GCN, similar to HOLMES[109]. The forget gate of the LSTM module filters the received information from different hop neighbors over multiple propagation rounds.

To achieve this, we initially randomly initialized the hidden and cell state of the LSTM. Now, after each propagation round of the GCN, once we get the learned node representation accumulating the information from the neighborhood, we pass it as an input to the LSTM. The LSTM module updates its hidden and cell state using the input, forget and output gates. The input gate of the LSTM extracts new information from the learned node representation after each GCN iteration and adds it to its memory. Forget gate of the LSTM learns to filter out information gathered at each propagation round of the GCN. Lastly, the output gate and the updated memory

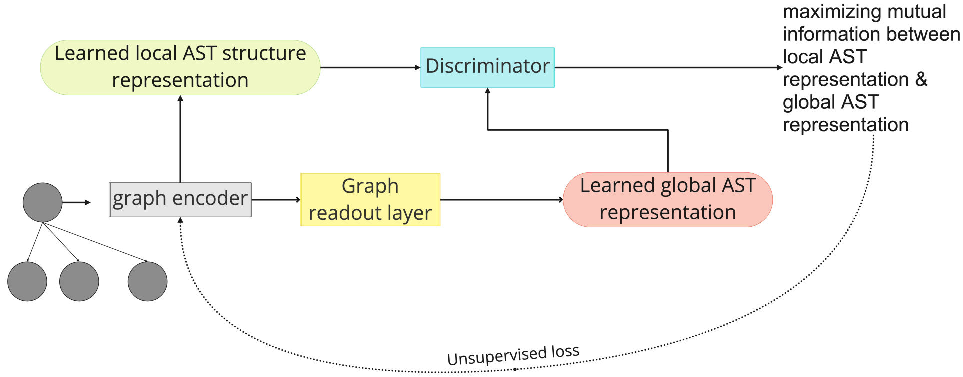


Figure 5.8: Illustration of unsupervised graph embedding learning in RUBHUS.

of the LSTM are used for constructing the final hidden node representation for the AST nodes.

5.3.2.3 Unsupervised AST Embedding Learning

In our work, we use unsupervised learning to learn a high-level program representation that captures the global information content of ASTs and information from different sub-structures of AST in the latent embedding. To achieve this, we use the principle of Mutual Information (MI) maximization, as described in [169]. We use MI to improve the representation learned at the local-level by incorporating local structures and sub-structures of AST into the learned node representation.

Specifically, we use a graph convolutional network (GCN) encoder (discussed in Section 5.3.2.2) to learn the node feature matrix H^K for a given code snippet after K iterations. H^K maintains information from its K hop neighbors. Next, we combine the information learned at each iteration of the GCN encoder to obtain the summarized node features H_α using a concatenation function.

$$H_\alpha = CONCAT\left(\{H^k\}_{k=1}^{K}\right), \tag{5.3}$$

Since GCN encodes a particular substructure of AST at each iteration, H_α has information from all different scales of AST. For instance, at depth level 1, all AST subgraphs with one-hop neighbors are encoded adaptively. Similarly, at depth level 2, AST subgraphs with two-hop

neighbors are encoded, and so on.

Next, to obtain the global representation ($H_\alpha(G)$), we apply a graph readout layer, as shown in Section 5.3.2.4. We then calculate the MI between the global AST representation and the local AST structures over the dataset of N samples $\mathbf{G} = \{\mathcal{G}_l \in \mathbb{G}\}_{l=1}^{N}$ as:

$$\hat{\phi}, \hat{\theta} = \underset{\hat{\phi},\hat{\theta}}{argmax} \sum_{\mathcal{G}\in\mathbf{G}} \frac{1}{|\mathcal{G}|} \sum_{i\in\mathcal{G}} \underset{\hat{\phi},\hat{\theta}}{I}(H_\alpha^i; H_\alpha(\mathcal{G})), \tag{5.4}$$

where, $\hat{\phi}$ and $\hat{\theta}$ are the parameters, and $\underset{\hat{\phi},\hat{\theta}}{I}$ is the MI calculated between local AST structures (H_α^i) and the global AST representation ($H_\alpha(G)$). The $argmax$ function maximizes the MI between $H_\alpha(G)$ and H_α^i, thereby encoding the global information of AST and information from the different sub-structures of AST in the latent embedding. Figure 5.8 shows how unsupervised learning is used for the training of RUBHUS. We learn this representation simultaneously while learning for code similarity. Thus, unsupervised loss improves the learned local AST node representations while capturing various global latent structures and sub-structures, and the supervised loss (described in Section 5.3.3) enables the clone detection module to learn program similarity.

Though, our unsupervised module closely resembles [169] in its objective of improving graph-level representation using MI. We make several design differences due to the different problems we are focusing on. First, we focus on learning local neighborhood information that captures global information content. Thus, we use random patches from other alternative graphs in the batch. Second, we use an adaptive AST encoder (described in Section 5.3.2.2) to encode AST representations in place of GIN. Third, our graph readout function is more expressive and captures information from the AST nodes while preserving the syntax, structure, and semantic information present in the AST.

5.3.2.4 AST Pooling Module

To obtain program representation G from the learned node embeddings H^K after K iterations of GCN, we apply set2set graph pooling layer [170].

The pooling layer is an iterative content-based attention layer that preserves AST node ordering information using LSTM and employs attention to give selective importance to each AST node. The pooling layer thus computes a coarsened graph representation considering both learned node features and topology, thereby preserving the syntax, structure, and semantic information present in the AST. To obtain global program representation, set2set computes:

$$\begin{aligned} q_t &= LSTM(q^*_{t-1)}) \\ \alpha_{i,t} &= softmax(h_i \cdot q_t) \\ r_t &= \sum_{i=1}^{N} \alpha_{i,t}, h_i \\ q^*_t &= [q_t || r_t] \end{aligned} \tag{5.5}$$

Where LSTM is an LSTM which computes recurrent state and evolves q^*_t by concatenating attention readout r_t with q_t, i index through each node of the AST tree, and q_t is the query vector which reads attention vector r_t.

5.3.3 Clone Detection Module

The AST embeddings for input code snippets obtained from the GNN encoder (5.3.2) are concatenated and passed through the binary classifier, which is a feed-forward neural network with a sigmoid output layer to model code similarity.

5.3.4 Implementation Details

Detecting clones across different programming languages poses a challenge due to disparities in syntax, typing, and paradigms. However, RUBHUS effectively addresses this challenge by leveraging an AST-based representation. This representation captures the structural and hierarchical information of code, enabling clone detection at various levels of granularity. The versatility of RUBHUS becomes apparent as it is not constrained by a specific granularity level

for clone detection. As long as the code blocks are compilable, our tool can identify clones at any desired level, providing flexibility in adapting to different scenarios and programming languages. In our current datasets, the files primarily consist of functions, and thus, we used function-level granularity in our experiments.

The power of RUBHUS lies in its ability to uncover similar patterns and relationships by analyzing the AST representations of code snippets. This can be observed in Figures 5.2 and 5.3, which demonstrate the presence of common structures in Listings 5.1 and 5.2, respectively. To further capitalize on these syntactic and semantic relationships, we have enriched the AST with control and data flow information. In our experiments, we employ two variants of our tool: **RUBHUS***, which utilizes the vanilla AST, and **RUBHUS**, which incorporates the FE-AST, to maximize the exploitation of these relationships.

For constructing ASTs, we employed the Java parser [171] for Java, Clang [172] for C, and the Python AST module [173] for Python. The implementation of RUBHUS is based on the PyTorch geometric [174] deep learning library, with the ReLU [175] function serving as the non-linear activation function. We initialized the network using the Xavier Uniform method [176] and trained it using the Adam [177] optimizer, with a learning rate of 0.0001 and a batch size of 20. The ReduceLROnPlateau Learning Rate Scheduler was employed. To learn program representation, we used the unsupervised loss as described in Section 5.3.2.3. For program similarity learning, we employed the BCEWithLogits Loss given by Equation 5.6.

$$BCELoss = -\frac{1}{N}\sum_{i=1}^{N} y_i \times log(p(y_i)) + (1 - y_i) \times log(1 - p(y_i)) \tag{5.6}$$

In the equation, y_i represents the true binary label, and $p(y_i)$ represents the predicted probability (similarity score). N denotes the number of samples in the dataset. The output of our model is the similarity score. To classify code snippets as clones or non-clones, we adopted a decision threshold of 0.5, in line with other code clone detection tools [63, 59]. If the similarity score exceeds the threshold, the code snippets are considered clones; otherwise, they are classified as non-clones.

	AtCoder		CodeChef	
	Java	Python	Java	C
Problems	576	576	179	179
Avg. Solutions	36	41	226	84
File Count	20k	23k	40k	15k
Avg. lines/files	46	13	42	56
Avg. tokens/files	324	76	105	306
#Clones	150k		100k	
#Non-clones	100k		150k	

Table 5.1: Cross-language code clones datasets.

5.4 Experimental Design

This section details the comprehensive evaluation of RUBHUS. The evaluation aims to fulfill the following objectives:

1) Objective 1 (O1): To perform a comparative evaluation of RUBHUS in the cross-language context with statically and dynamically typed languages.

2) Objective 2 (O2): To assess the generalizing capability of RUBHUS on unseen projects.

3) Objective 3 (O3): To assess architectural efficacy of single language code clone detection tools in cross-language context.

4) Objective 4 (O4): To assess RUBHUS's performance in the single-language context with statically and dynamically typed languages.

5.4.1 Datasets

A labeled dataset is required to train our semi-supervised model to measure code similarity. Often large amounts of training data are required for deep learning techniques to avoid overfitting and poor approximation. There are no openly available datasets of cross-language clones and manually validating a large number of code pairs to create a dataset is a challenging task. Thus, we use data from coding contest websites. Coding contests have several problems, with each problem attempted by independent programmers in different programming languages. Programming competitions have several problems, and each problem is attempted by different programmers independently and without language restrictions. The submissions across languages

are tested on the same input and expected output to validate them, which ensures the functional correctness of the submitted solutions and also the functional similarity existing between the different solutions [112]. The submissions are independent and are assumed to have sufficient diversity in their algorithms, data structures, and programming constructs [108, 63]. These properties make code submissions a reasonable choice for training and evaluating the clone detection tools. Multiple submissions implementing the same problem are considered clones, and the submissions made for different problems are considered non-clones. We use the following datasets in our experiments:

1. **AtCoder Dataset:** To evaluate RUBHUS, we use the dataset collected and used by [63]. This dataset consists of cross-language code pairs of Java and Python from 576 different problems from the AtCoder coding competition website [178]. The dataset contains correct and accepted solutions implemented in Java (a statically typed language) and Python (a dynamically typed language). The Java and Python solutions submitted for the same problem form clones, while the solutions submitted for different problem constitutes non-clones. We use 576 problems in total, and each problem has an average of 36 solutions for Java and 41 solutions for Python. The average code size is 46 lines for Java and 13 lines for Python. Detailed statistics are reported in Table 5.1.

2. **CodeChef Dataset:** To further assess the performance of RUBHUS, we create a dataset of cross-language code pairs consisting of C and Java problems from the CodeChef programming contests [179]. We use 244 different problems available at [180]. In Kaggle, each problem has an average of 226 submissions for C, and each submission has an average of 42 lines of code. For Java, the average number of submissions is 84, and each submission, on average, has 56 lines of code. To form clones, submissions corresponding to the same problem are paired; for non-clones, the submissions from the different problems are paired. Due to the combinative process of forming non-clones, they rapidly outnumber clones, leading to dataset imbalance. Therefore, we use reservoir down sampling prior to training to balance between clone and non-clone pairs. The detailed statistics for the CodeChef dataset are reported in Table 5.1.

For all experiments, we divide the datasets into training, testing, and validation set. We

Table 5.2: Single-language semantic clones datasets.

	Project Files	**# Methods**	**LOC**	**# clones**	**# non-clones**
BCB	9.1K	58K	267K	650K	650K
SeSaMe	11 projects	9.1k	180K	93	–

Table 5.3: % clone-types in BigCloneBench.

Clone Type	T1	T2	ST3	MT3	WT3/T4
Percentage(%)	0.005	0.001	0.002	0.01	0.982

reserve 60% of the entire dataset as the training set, 20% for validation and finetuning, and the remaining 20% as the test set for the final model evaluation over the metrics of precision, recall, and F1-score.

5.4.1.1 Generalizing capability on unseen datasets

To further fulfill objective **O2**, we restructure the process of clone and non-clone pair formation from the AtCoder and CodeChef datasets to have a substantial semantic difference in the training and the test set. This setting ensures that all pairs in the training and the test set are semantically different, unlike the 6:2:2 experimental setting used above, where semantically similar pairs from the training set can exist in the test set. To accomplish this, we create the datasets: AtCoder* and CodeChef*, where we use 460 and 143 problems from AtCoder and CodeChef, respectively, for creating pairs for the training set and validation set, and the remaining problems for testing the model.

5.4.1.2 Evaluation on single-language code clone benchmarks

To fulfill objective **O4**, we evaluate RUBHUS on the following single-language semantic clone benchmarks:

1. **SeSaMe Dataset:** SeSaMe [181] dataset contains semantically similar clone pairs mined from 11 open-source Java repositories. The authors applied text similarity measures on Java doc's comments on these open source-projects and then manually inspected and evaluated the results.

This dataset reports 857 manually classified clone pairs validated by eight judges. The pairs were distributed in a way that three judges evaluated each pair. The semantic similarity between pairs was reported on three scales: `goals, operations, and effects`. The judges had the option to choose whether they `agree, conditionally agree, or disagree` with confidence levels `high, medium, and low`.

2. **BigCloneBench Dataset:** BigCloneBench (BCB) [62] dataset, released by Svajlenko et al., was developed from the IJAdataset$-2.0.3$. IJAdataset contains $25K$ open-source Java projects and $365M$ lines of code. The authors have built the dataset by mining frequently used functionalities, such as bubble sort. The initial release of the BCB dataset covers ten functionalities, including six million clone pairs and $260K$ non-clone pairs. The current release of the BCB dataset has about eight million clone pairs covering 43 functionalities. Some recent code clone detection tools, such as, TBCCD [61] have used the initial version of the BCB covering ten functionalities for their experiments. Hence, to present a fair comparison, we have also used the same version.

BCB dataset has categorized clone types into five categories: Type-1, Type-2, Strongly Type-3, Moderately Type-3, and Weakly Type-3+4 (Type-4) clones. Since there was no consensus on minimum similarity for Type-3 clones and it was difficult to separate Type-3 and Type-4 clones, the BCB creators categorized Type-3 and Type-4 clones based on their syntactic similarity. Thus, Strongly Type-3 clones have at least 70% similarity at the statement level. These clone pairs are very similar and contain some statement-level differences. The clone pairs in the Moderately Type-3 category share at least half of their syntax but contain a significant amount of statement-level differences. The Weakly Type-3+4 code clone category contains pairs that share less than 50% of their syntax. Tables 5.2 and 5.3 summarize the data distribution of the BCB dataset.

5.4.2 Research Questions

To determine RUBHUS's effectiveness in detecting clones across different programming languages, we evaluate RUBHUS on two datasets containing code pairs from Java-Python (AtCoder

dataset) and Java-C (CodeChef dataset). Further, to assess the generalizing capability of RUBHUS, we create datasets AtCoder* and CodeChef* in which we exclusively reserved 460 and 143 problems and their submissions, respectively, for training and remaining problems and submissions are used for testing RUBHUS and other baseline models. This led us to our first research question:

Research Question 5.1

How does the performance of RUBHUS compare in the cross-language context between statically and dynamically typed languages, and how well does it generalize on unseen projects?

Thus, RQ 5.1 aims to fulfill objective **O1** and **O2**.

A lot of research has focused on developing single-language learning-based code clone detection techniques. However, none of these techniques have ever been applied to detect cross-language clones. We argue that these single-language clone detection techniques are comparable with cross-language clone detection techniques, as the key component of these techniques is to capture high-level representative features from code snippets, and any algorithm designed specifically for capturing these key syntactic and semantic relationships should perform well on both cross-language and single-language code clone detection techniques. Thus, to test our hypothesis and to develop a deeper understanding of the impact of these tools' high-level code representative features on cross-language code clone detection tasks, we evaluate state-of-the-art single-language clone detection techniques on cross-language datasets. This led us to our next research question:

Research Question 5.2

How do state-of-the-art single-language code clone detection tools perform on cross-language code clone detection datasets?

Thus, to fulfill **O3** and to assess the effectiveness of RUBHUS in the single-language context, we evaluate it on the datasets curated from single statically typed languages. We use two different programming paradigms: the procedural programming paradigm (C) and the object-oriented programming paradigm (Java). To create a benchmark of Java pairs, we use submissions curated

from the AtCoder dataset. For C language, we use submissions from the CodeChef dataset.

Further, to assess the efficacy of RUBHUS on open-source java projects, we use BCB and SeSaMe datasets consisting of semantically similar clone pairs. This led us to our third research question:

Research Question 5.3

How effectively can RUBHUS detect clones in statically typed languages?

Next, to assess the effectiveness of RUBHUS in dynamic-typed languages, we evaluate RUBHUS on a dataset curated using Python code files from the AtCoder dataset. This led us to our fourth research question:

Research Question 5.4

How effectively can RUBHUS detect clones in dynamically typed languages?

Thus, RQ 5.3 and RQ 5.4 are designed to accomplish the objective **O4**.

5.4.3 Baseline Evaluation

To answer our research questions, we use the following baseline (best-performing variants are selected) techniques to demonstrate and comparatively assess the performance of RUBHUS.

5.4.3.1 Comparison with cross-language state-of-the-art code clone detectors

To carry out a more reliable and comprehensive evaluation of RUBHUS, we compare it with state-of-the-art cross-language clone detection tools: C4 [118], GRAPHCODEBERT [119], XCODE [120], PCCD[2] [63], and CLCDSA [59]. Tao et al. propose a cross-language code clone detection tool C4, which takes a pre-trained code model CODEBERT to learn program representation. The authors use contrastive learning to finetune the model. Guo et al. propose a transformer-based technique, GRAPHCODEBERT, that exploits code data flow information " where-the-value-comes-from" to learn code representation. Lin et al. propose a cross-language

[2] For brevity, we refer to the tool as PCCD since the authors in the paper did not name the tool

Table 5.4: Hyperparameter configurations for deep learning-based tools used in the experiments.

Tools	**Input dimension**	**Encoder layer**	**Classifier**	**Optimizer**	**Epochs**	**Classification Threshold**
PCCD	100	Bidirectional LSTM stacked with 2 layers	Single hidden layer, 64 units	RMSprop	50	0.5
TBCCD	79	Number of convolutional kernel: 600, Depth of sliding window: 2	Single hidden layer, 50 units	SGD with LR of 0.0001	10	Threshold moving approach
CLCDSA	9	2 fully connected hidden layers of 100 units each, 3 fully connected hidden layer with 100, 50, 10 units respectively in the comparator network	Single hidden layer, 10 units	SGD with LR of 0.0001	55	0.5
FA-AST+GMN	100	Dimension for GNN: 100, GNN iterations: 4 steps	Single hidden layer with 100 units	Adam with LR of 0.001	10	Threshold moving approach
FA-AST+GGNN	100	Dimension for GNN: 100, GNN iterations: 4 steps, Global attention for graph pooling	Single hidden layer with 100 units	Adam with LR of 0.001	10	Threshold moving approach
XCODE	100	BiLSTM encoder hidden size: 100, Decoder LSTM hidden size: 100	3 layer feed forward neural network with 100, 100, 50 units respectively	Adam with LR of 0.001	50	0.8
C4	768	Roberta Model	6 Transfer Decoder Layer	AdamW with LR of 0.00005	10	Threshold moving approach
GRAPHCODEBERT	768	Roberta Model	two layer feed-forward network with 1536 and 768 units respectively	Adam with LR 0.00002	5	0.5

Table 5.5: Hyperparameter settings for traditional code clone detection tools used in the experiments.

Tools	Parameters	C	Java	Python
Nicad	Min. Line	5	5	5
	Max. Line	600	600	300
	Threshold	1	2	4
CCFinder	Min. token	15	15	15

code representation learning framework XCODE which utilizes language models pre-trained by AST and ELMo-enhanced variational autoencoders and SED architecture that uses the multi-teacher single-student method. Perez and Chiba propose a cross-language code clone detection technique PCCD that uses AST and code tokens to learn code representation vectors using the SkipGram algorithm [94]. Nafi et al. propose a cross-language clone detection tool CLCDSA using syntactical features from the ASTs and API documentation. The authors propose to detect clones on the fly using a deep neural network.

In literature, there are other cross-language clone detection tools such as SLACC [116], LICCA [113], and CLCMiner [115]. SLACC is based on dynamic analysis techniques and requires input generation and the execution of code files, which was not possible on our datasets. CLCDSA significantly outperforms LICCA and CLCMiner in their experimental evaluation. Hence, we do not compare RUBHUS with SLACC, LICCA, and CLCMiner in our experiments. This leads to C4, XCODE, GRAPHCODEBERT, PCCD, and CLCDSA being natural choices for comparative evaluation with RUBHUS. For detailed hyperparameter configurations of these cross-language code clone detection baselines, please refer to Table 5.4.

5.4.3.2 Comparison with single-language state-of-the-art code clone detectors

To assess the efficacy of RUBHUS in the single-language context, we use both traditional (NICAD [32], and CCFINDER [30]) and learning-based (TBCCD [61], and FA-AST [107]) state-of-the-art single-language code clone detection tools. TBCCD applies tree-based convolutions over ASTs of the code snippet to measure similarity. FA-AST employs a GNN-based technique, which parses code snippets into flow-augmented ASTs and then applies gated graph neural networks (FA-AST+GGNN) and graph matching networks (FA-AST+GMN) to learn program

similarity.

Cordy and Roy propose a scalable and flexible code clone detection tool NICAD to detect near-miss intentional clones. The tool parses the input code snippet, applies normalization, and abstractions and then compares the fragments using optimized LCS for similarity detection. CCFINDER is a token-based single-language code clone detection technique that transforms the input source code text and then performs token-by-token comparison for code similarity detection.

Note, for **O2** i.e., to assess the architectural efficacy of single language code clone detection tools in cross-language context, we only compare learning-based tools, i.e., TBCCD, FA-AST+GGNN, and FA-AST+GMN with RUBHUS on cross-language datasets. The traditional single language code clone detection tools (NICAD and CCFINDER) require significant modification in their implementation settings which were beyond the scope of this work.

For detailed hyperparameter configurations of these single-language code clone detection baselines, please refer to Table 5.4 and 5.5.

5.4.3.3 Comparison with a baseline GNN model

We also compare RUBHUS against a baseline GNN model - CHEBCONV [182] on MNIST and citation network datasets. This GNN architecture learns node feature representation using spectral graph convolutions, and to generate the graph representation, we aggregate all the learned node features. This simple baseline model comparison aims to study the effectiveness of learning more profound and informative graph representations through our proposed technique - RUBHUS.

5.5 Results

The results of our experiments demonstrate that RUBHUS can identify clones and non-clones with high precision and recall between languages and generalizes well on unseen datasets as compared

Table 5.6: Classification performance of RUBHUS and other baselines (in %) in the cross-language setting. Empty cells in the tables indicate that the tools do not provide support for C language.

(a) Performance on AtCoder (Java-Python) and CodeChef (Java-C) Dataset

Tool	AtCoder Dataset			CodeChef Dataset		
	Precision	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**
XCODE	89	92	90	-	-	-
C4	92	89	91	-	-	-
GRAPHCODEBERT	70	71	71	-	-	-
PCCD	55	81	65	-	-	-
CLCDSA	65	90	75	-	-	-
CHEBCONV	59	78	67	60	69	64
RUBHUS*	90	95	92	89	92	90
RUBHUS	**96**	**98**	**97**	**92**	**96**	**94**

(b) Performance on AtCoder (Java-Python)* and CodeChef (Java-C)* Dataset.

Tool	AtCoder* Dataset			CodeChef* Dataset		
	Precision	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**
XCODE	81	83	82	-	-	-
C4	84	83	83	-	-	-
GRAPHCODEBERT	60	61	61	-	-	-
PCCD	29	55	40	-	-	-
CLCDSA	45	60	51	-	-	-
CHEBCONV	44	65	52	40	49	44
RUBHUS*	79	86	82	69	74	71
RUBHUS	**90**	**91**	**90**	**84**	**89**	**86**

to other clone detection techniques (RQ 5.1). The results also suggest that the performance of single-language code clone detection tools abates when applied to cross-language code clone datasets (RQ 5.2). In addition, the results also demonstrate that RUBHUS can detect clones with high precision and recall in single-language settings (RQ 5.3 and RQ 5.4).

5.5.1 RQ 5.1: How Does the Performance of RUBHUS Compare in the Cross-Language Context Between Statically and Dynamically Typed Languages, and How Well Does It Generalize on Unseen Projects?

In this RQ, we discuss the efficacy of RUBHUS in detecting clones from different languages supporting different programming paradigms and typing. We use two datasets, AtCoder and CodeChef, to assess the efficacy of RUBHUS in detecting program similarity across statically and dynamically typed languages and across procedural and OOP paradigms as presented in Table 5.6a. We use precision (P), recall (R), and F1-score (F1) values to measure the efficacy of

RUBHUS against other baseline techniques. The color coding indicates that the darker the color's hue better the results.

From the results, we observe that the code clone detection tools have unbalanced precision and recall values. For C4, the precision is high because C4 uses contrastive loss to minimize the distance between similar pairs, which reduces false positives. However, C4 has a low recall value as it uses code tokens to learn program representation and, cross-language code snippets do not have much token-level similarity, which in turn, results in more false negatives. GRAPHCODEBERT only uses code tokens and a specific kind of data flow relation for program representation learning. Since the cross-language code snippets do not share much syntactic similarity, it misses out on several clone pairs or incorrectly identify non-clones as clones. XCODE achieves decent precision and recall values over other baselines because of its large-scale cross-language code representation framework using the multi-teacher single-student method to transfer knowledge method. CLCDSA also achieves very high recall values as it uses various syntactic features such as the number of variables declared and number of loops. These syntactic features reduce the occurrences of false negatives, but at the same time, the percentage of false positives also increases, which results in a low precision value.

We also observe that RUBHUS outperforms other baseline techniques on AtCoder and CodeChef datasets on every evaluation metric. On the AtCoder dataset, we observe an improvement of $\geq 33\%$ in the F1-score. The other cross-language code clone detection baselines, except CHEBCONV, only support Java and Python. To make them work for other languages, significant changes need to be made, which are beyond the scope of this work. Thus, for the CodeChef dataset, we only compare RUBHUS with CHEBCONV GNN model. On the CodeChef dataset, we see a substantial improvement in the F1 score in comparison to the CHEBCONV model.

Figure 5.9 shows the true clone pair predicted true by RUBHUS but false by PCCD, C4, and GRAPHCODEBERT. The two listings look very different; even the human developers will take some time to determine whether the code pairs are implementing similar functionality. However, RUBHUS correctly identifies the two pairs as clones through FE-ASTs, which expose similarities via induced control and data flow edges. This information is used by our semi-supervised representation learning framework, which incorporates global AST information of

```java
public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  double[] angle = new double[16];
  for(int i = 0; i < 16; i++)
    angle[i] = 11.25 + 22.5 * (double)i;
  String[] orientation = {"NNE", "NE", "ENE", "E", "ESE", "SE", "SSE", "S", "SSW", "SW",
  ↪  "WSW", "W", "WNW", "NW", "NNW"};
  int deg = sc.nextInt();
  double orient = (double)deg / (double)10;
  String dir = "";
  for(int i = 0; i < 15; i++) {
    if(orient >= angle[i] && orient < angle[i + 1]) {
    dir = orientation[i];
    break;
    }
  }
  if(dir.equals("")) dir = "N";
  double[] speed = {0.0, 0.3, 1.6, 3.4, 5.5, 8.0, 10.8, 13.9, 17.2, 20.8, 24.5, 28.5,
  ↪  32.7};
  int W = 0;
  int dis = sc.nextInt();
  double sp = (double)dis / (double)60;
  sp += 0.001;
  BigDecimal sp2 = new BigDecimal(sp);
  sp2 = sp2.setScale(1, BigDecimal.ROUND_HALF_UP);
  sp = sp2.doubleValue();
  for(int i = 12; i >= 0; i--) {
    if(sp >= speed[i]) {
      W = i;
      break;
    }
  }
  if(W == 0) dir = "C";
  System.out.println(dir + " " + W);
}
```

Listing 5.3: solution.java

```python
deg, dis = map(int, input().split())
deg = deg * 10
jdegs = [1125 + j * (36000 // 16)  for j in range(17)]
jdcts = ["N", "NNE", "NE", "ENE", "E", "ESE", "SE", "SSE", "S", "SSW", "SW", "WSW", "W",
↪  "WNW", "NW", "NNW", "N"]
for jdeg, jdct in zip(jdegs, jdcts):
  if deg < jdeg:
    dct = jdct
    break
jpows = [0.2, 1.5, 3.3, 5.4, 7.9, 10.7, 13.8, 17.1, 20.7, 24.4, 28.4, 32.6]
def round(x, d=0):
  p = 10**d
  return (x * p * 2 + 1) // 2 / p
spd = round(dis / 60, 1)
power = 12
for i, jpow in enumerate(jpows):
  if spd <= jpow:
    power = i
    break
if power == 0:
  dct = "C"
print(dct, power)
```

Listing 5.4: solution.py

Figure 5.9: A true clone pair reported true by RUBHUS but false by PCCD, C4, and GRAPHCODEBERT.

```java
public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  int a = sc.nextInt();
  boolean i;
  if(a%400==0){
    i = true;
  }else if(a%100==0){
    i = false;
  }else if(a%4==0){
    i = true;
  }else{i=false;}
  if(i)
    System.out.println("YES");
  else
    System.out.println("NO");
}
```

Listing 5.5: solution.java

```python
y=int(input())
print(['NO','YES'][y%4==0 and y%100!=0 or y%400==0])
```

Listing 5.6: solution.py

Figure 5.10: A true clone pair reported false by RUBHUS but true by XCODE and C4.

various structures and substructures into its learned local node representations. On the other hand, tools such as PCCD, C4, and GRAPHCODEBERT have difficulty in capturing this similarity because they do not incorporate the programs' structural, semantic and syntactic information into their representation learning framework and rely on a specific kind of information for measuring code similarity.

Thus, the reason for the improved performance of RUBHUS against the baselines can be attributed to the following reasons: (1) **Capturing better program semantics and syntactic information through control and data flow augmented ASTs as compared to the other tools**. XCODE and C4 use only code tokens and ASTs, while GRAPHCODEBERT proposes to use dataflow relation "where the value comes from". These techniques do not expose the full semantics and syntax of the program. (2) **Exploiting the long-range dependencies existing in source code and capturing AST structures and substructures through the proposed semi-supervised GNN architecture**. For example, RUBHUS can learn a node representation by incorporating information from the node's n-hop neighborhood through message passing but the graph-guided masked attention proposed in GRAPHCODEBERT only considers direct edge relations.

```java
int a = scanner.nextInt();
int b = 0;
for (int i = 0; i < a; i++)
  b += 10000 * (i + 1) / a;
System.out.println(b);
```

Listing 5.7: solution.java

```python
n = int(input())
ans = sum(list(range(n+1))) * 10000 / n
print(ans)
```

Listing 5.8: solution.py

Figure 5.11: A true clone pair reported false by RUBHUS and all other baselines.

Figure 5.10 shows the true clone pair identified incorrectly by RUBHUS but correctly by XCODE and C4. Listings 5.5 and 5.8 implement the same functionality but uses different algorithms to achieve the same. Java listing uses “if-else” to implement the functionality, while Python uses “list comprehension” to achieve the same. RUBHUS fails to detect the similarity of this kind, but since XCODE and C4 use code tokens for similarity learning, they predict Listings 5.5 and 5.8 as true clone pairs. This indicates that even though RUBHUS shows significant improvement in understanding the semantics of cross-language code snippets, it is still hard for deep learning-based techniques to fully understand the functionality in cases where we have significant differences in the algorithms.

Likewise, Figure 5.11 shows a true clone pair implementing the same functionality but using a different algorithm. This example has been reported as false negative by all the baselines as it is very hard to infer similarity of any kind between Listings 5.7 and 5.8.

Furthermore, we also observe improvements in the metrics when RUBHUS is compared against RUBHUS*, indicating that enriching ASTs with control and data flow information helps in capturing better semantic and syntactic relationships. This asserts that RUBHUS can learn enough similarities between the code snippets to classify them as clones and non-clones irrespective of programming paradigms and language typing.

To test the generalization capabilities of RUBHUS and RUBHUS* in the presence of substantial semantic differences between train and test datasets, we created two datasets: AtCoder* and CodeChef*, where code pairs in the test set are semantically different from those in the train set. Table 5.6b present the performance of various clone detectors, including RUBHUS and RUBHUS*, on these datasets.

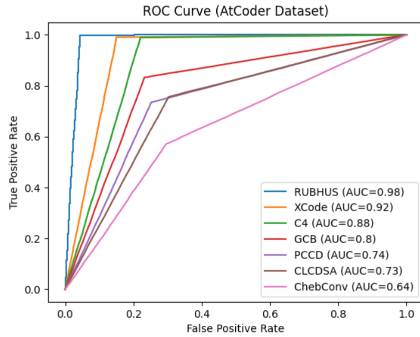


Figure 5.12: ROC curve of RUBHUS and other baselines on AtCoder dataset. (AUC values are rounded up to 2 decimal places.)

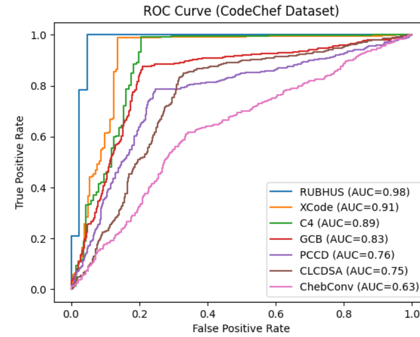


Figure 5.13: ROC curve of RUBHUS and other baselines on CodeChef dataset. (AUC values are rounded up to 2 decimal places.)

The tables reveal a significant drop in performance for CLCDSA and PCCD, which solely rely on syntactic features for program similarity learning. Additionally, there is a performance drop for C4, XCODE, and GRAPHCODEBERT, with C4 and XCODE outperforming GRAPHCODEBERT. The improved performance of C4 and XCODE can be attributed to the contrastive learning and multi-teacher single-student objectives they employ, respectively. Contrastive learning helps C4 identify subtle differences between similar samples while ignoring irrelevant differences between dissimilar samples, thereby promoting better generalization to new, unseen data. Similarly, the multi-teacher single-student objective training used by XCODE enables the student model to learn from multiple teacher models, reducing the impact of individual teacher biases or errors and promoting generalization to new, unseen data.

Interestingly, RUBHUS performs considerably better than the other baseline models on the unseen code files. This performance improvement is attributed to its representation learning framework (section 5.3.2), which focuses on learning generalized program features consisting of syntactic and semantic information while abstracting program-specific details. GNNs with unsupervised training loss can learn information from different scales of AST substructures, facilitating program similarity measurement.

Further, to assess the ability of RUBHUS to identify clone pairs and to compare its performance with other existing methods, we employed the Receiver Operating Characteristics (ROC) curve. The ROC curve is a graphical representation of the trade-off between the True Positive Rate (TPR) and False Positive Rate (FPR) for different classification thresholds. TPR is the ratio of

correctly classified clone pairs to the total number of actual clone pairs, and FPR is the ratio of non-clone pairs that are incorrectly classified as clone pairs to the total number of non-clone pairs. By varying the classification threshold, we plotted the ROC curve to visualize the diagnostic ability of RUBHUS and other baseline methods. The ROC curve is a useful tool that can help us determine the optimal threshold that balances the trade-off between TPR and FPR. The closer the ROC curve is to the top-left corner of the graph, the better the diagnostic ability of the method.

To compare the performance of RUBHUS and other methods, we presented the resulting ROC curves and corresponding Area Under Curve (AUC) values for the AtCoder and CodeChef datasets in Figures 5.12 and 5.13. The AUC is a metric that measures the degree of separability between clone and non-clone pairs. A high AUC value indicates a superior classifier that excels at correctly identifying clone pairs as clones and non-clone pairs as non-clones. From the figures, we can see that RUBHUS outperforms other baseline methods, with the highest AUC values on both the AtCoder and CodeChef datasets.

> **RQ 5.1:** RUBHUS *can successfully identify code clones across different programming languages with high precision and recall and also generalizes well on AtCoder* and CodeChef* datasets.*

5.5.2 RQ 5.2: How Do State-of-the-Art Learning-Based Single-Language Code Clone Detection Tools Perform on Cross-Language Code Clone Detection Datasets?

Though these tools are designed for single-language code clone detection, we believe these tools are comparable in cross-language settings also with little or no change in their original implementation. Hence, to develop a deeper understanding of the working of these tools in the cross-language context, in this RQ, we compare tools: TBCCD, FA-AST+GGNN, and FA-AST+GMN on cross-language code clone detection datasets- AtCoder, and CodeChef.

Table 5.7a present the results of comparative evaluation of TBCCD, FA-AST+GGNN, and FA-AST+GMN with RUBHUS and RUBHUS*. From the table 5.7a, we observe a significant drop in the performance of TBCCD, FA-AST+GGNN, and FA-AST+GMN. The reason for

Table 5.7: Classification performance of single-language code clone detection tools on cross-language code clones datasets.

(a) Performance on AtCoder (Java-Python) and CodeChef (Java-C) Dataset.

Tool	AtCoder Dataset			CodeChef Dataset		
	Precision	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**
TBCCD	66	3	6	15	0	0
FA-AST+GGNN	71	98	82	75	95	84
FA-AST+GMN	66	65	65	60	66	63
RUBHUS*	95	95	92	89	92	90
RUBHUS	**96**	**98**	**97**	**92**	**96**	**94**

(b) Performance on AtCoder (Java-Python)* and CodeChef (Java-C)* Dataset.

Tool	AtCoder* Dataset			CodeChef* Dataset		
	Precision	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**
TBCCD	71	0	0	33	0	0
FA-AST+GGNN	66	85	74	70	88	78
FA-AST+GMN	42	45	43	41	41	41
RUBHUS*	79	86	82	69	74	71
RUBHUS	**90**	**91**	**90**	**84**	**89**	**86**

the performance drop of these tools on the AtCoder and CodeChef datasets can be attributed to their representation learning framework and the choice of input program features. For instance, TBCCD uses tree-based convolutions over token-enriched AST to learn structural and syntactical program features, thereby not capturing program semantic information. Similarly, FA-AST use graph neural networks over flow-augmented ASTs. The extra control and data-flow edges (`next use edge`) added by FA-AST do not represent precise and complete program semantics. Moreover, the performance difference can also be observed between the variants of FA-AST: FA-AST+GGNN and FA-AST+GMN. FA-AST+GMN uses graph matching networks (GMNs) which try to find correspondence between graphs for predicting similarity. In the case of cross-language code clone detection, graph matching becomes challenging and unpredictable due to the difference in ASTs caused by differences in language, paradigms, structures, etc. This explains the low performance of FA-AST+GMN when compared with FA-AST+GGNN.

Hence, these techniques show impressive performance on datasets where there exists a syntactic or lexical similarity between the code snippets, but cross-language code snippets share the merest syntactic or token-level similarity, which explains the performance decline.

Table 5.8: Classification performance of RUBHUS and other baselines on statically-typed languages in single-language setting. Empty cells in the table indicate that the tools do not provide support for C language.

Tool	AtCoder (Java-Java) Dataset			CodeChef (Java-Java) Dataset			BCB Dataset (F1-Score)				
	Precision	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**	**T1**	**T2**	**ST3**	**MT3**	**WT3/T4**
NICAD	55	25	34	51	30	38	85	83	60	35	0
CCFINDER	45	5	9	44	5	9	82	81	55	20	0
PCCD	59	75	66	-	-	-	100	91	80	76	75
CLCDSA	70	92	80	-	-	-	100	95	90	89	89
TBCCD	80	86	83	73	81	77	100	100	98	96	96
FA-AST+GGNN	82	88	85	80	84	82	100	90	79	75	75
FA-AST+GMN	85	89	87	87	90	88	100	100	95	95	95
CHEBCONV	74	73	74	75	79	77	100	99	95	90	90
XCODE	85	89	87	-	-	-	100	100	99	98	96
C4	91	90	91	-	-	-	100	99	96	95	95
GRAPHCODEBERT	83	84	83	-	-	-	100	100	98	98	98
RUBHUS*	89	90	89	90	90	90	100	100	95	95	94
RUBHUS	92	93	93	91	94	93	100	100	99	99	98

Further Table 5.7b presents the performance of TBCCD, FA-AST+GGNN and FA-AST+GMN on the unseen AtCoder* and CodeChef* datasets. We can observe that RUBHUS outperforms the other baselines techniques, and the performance is consistent with the results reported in Table 5.7a.

RQ 5.2: *In cross-language context, where there exists merest syntactic or token-level similarity the performance of single-language code clone detection tools deteriorate.*

5.5.3 RQ 5.3: How Effectively Can RUBHUS Detect Clones in Statically Typed Languages?

In this RQ, we discuss the efficacy of RUBHUS in detecting single-language clones in statically typed languages: C & Java. To address the RQ, we evaluate cross-language and single-language clone detectors and a baseline GNN model on AtCoder(Java-Java), CodeChef (C-C), and BCB clones and non-clones datasets.

Table 5.8 presents the classification performance of RUBHUS and other clone detectors on Java and C code pairs. The results show that RUBHUS substantially outperforms the other tools and the baseline GNN model, CHEBCONV. RUBHUS performs better than RUBHUS* asserting the fact that explicitly enriching ASTs with control and data-flow information helps in learning the program's syntactic and semantic information.

```java
public class FileHelper {
  public static void
  ↪  recursiveDeleteDirectory(File
  ↪  cacheDir) {
  if (cacheDir.isDirectory()) {
    File[] files = cacheDir.listFiles();
    for (int i = 0; i < files.length;
    ↪  i++) {
      File file = files[i];
      recursiveDeleteDirectory(file);
    }
    } else {
      cacheDir.delete();
    }
    }
}
```

Listing 5.9: File1.java

```java
public static final boolean
↪  deleteAll(File what) throws
↪  FileNotFoundException, IOException
↪  {
  if (what == null) {
    return false;
  }
  if (what.isDirectory()) {
    File[] files = what.listFiles();
    for (int i = 0; i < files.length;
    ↪  i++) {
      deleteAll(files[i]);
    }
  }
  return what.delete();
}
```

Listing 5.10: File2.java

Figure 5.14: A semantically similar code clone example from the BCB dataset implementing recursive file/directory deletion.

We have observed that the performance of single-language code clone detection tools is considerably better on single-language code clone datasets. This is because, unlike cross-language code pairs, there is substantial similarity between snippets from the same language.

On the BCB dataset (Table 5.8), we observe that almost all the code clone detection tools are quite effective in recognizing the Type 1 and Type 2 similarity between code snippets. While for other similarity types in BCB, the performance of code clone detection tools degrades. NICAD and CCFINDER have an F1-score of 0 on WT3/T4 category. However, we can also observe that the performance depreciation of learning-based tools is not that severe. Most of the tools perform better for the WT3/T4 category of BCB than AtCoder and CodeChef datasets.

The reason for improved performance of these tools on the BCB dataset can be attributed to two reasons: 1) Even the hard-to-detect code clone categories (ST3−WT3/T4) in BCB have a high syntactic similarity. This syntactic similarity is being utilized by learning-based tools for code similarity detection. For example, Figure 5.14 shows an example of a WT3/T4 code clone pair, which has high similarity and varies in terms of lexical tokens and the sequence of invoked method calls and APIs. 2) Further, as opposed to the BCB dataset, the code pairs in AtCoder and CodeChef datasets have a significant difference in structure and syntax as they have been written by independent programmers from scratch. Hence, without learning program semantics, it isn't easy to measure similarity in AtCoder and CodeChef datasets.

Table 5.9: Classification performance of RUBHUS and other baselines on unseen statically-typed single-language code clones datasets. Empty cells in the tables indicate that the tools do not provide support for C language.

Tool	AtCoder (Java-Java)* Dataset			CodeChef (C-C)* Dataset		
	Precision	Recall	F1-score	Precision	Recall	F1-score
NICAD	-	-	-	-	-	-
CCFINDER	-	-	-	-	-	-
PCCD	28	50	36	-	-	-
CLCDSA	68	90	77	-	-	-
TBCCD	65	76	70	70	81	75
FA-AST+GGNN	60	62	61	55	61	57
FA-AST+GMN	70	76	72	71	75	73
CHEBCONV	50	52	51	51	51	51
XCODE	80	86	83	-	-	-
C4	87	88	88	-	-	-
GRAPHCODEBERT	79	79	79	-	-	-
RUBHUS*	79	80	79	75	78	76
RUBHUS	89	89	89	88	85	86

(a) Performance on AtCoder (Java-Java)* and CodeChef(C-C)* Datasets.

Tool	SeSaMe Dataset			BCB Dataset				
	Precision	Recall	F1-score	T1	T2	ST3	MT3	WT3/T4
NICAD	53	20	30	-	-	-	-	-
CCFINDER	44	5	9	-	-	-	-	-
PCCD	25	49	33	91	85	69	70	70
CLCDSA	60	80	68	95	91	86	85	85
TBCCD	60	65	62	95	95	96	92	92
FA-AST+GGNN	50	55	52	92	85	73	70	70
FA-AST+GMN	61	67	64	96	96	95	92	92
CHEBCONV	44	45	44	95	95	90	86	86
XCODE	78	85	81	95	95	94	94	94
C4	84	86	85	95	95	94	93	93
GRAPHCODEBERT	75	72	73	97	96	96	96	95
RUBHUS*	75	78	76	96	96	95	94	94
RUBHUS	85	88	86	99	99	99	98	95

(b) Performance on SeSaMe and BCB datasets.

Further, to assess the performance of these tools on unseen code files, following a similar methodology described in section 5.4.1.1, we create datasets AtCoder(Java-Java)* and CodeChef(C-C)*. In these datasets, we reserve 460 and 63 number of problems from Java and C, respectively, for training and validation. The remaining problems are used for testing the model. This setting ensures that the training and test set for the clone detection tools have semantically unrelated and unseen files. On the BCB dataset, we reserve a single functionality (`copy a file`) from the BCB dataset for training, and the rest we use for evaluation. Furthermore, we evaluate RUBHUS and other baselines (pre-trained on AtCoder(Java-Java) dataset) on the SeSaMe dataset. The SeSaMe dataset consists of semantically similar method pairs mined from 11 open-source Java repositories.

The general trend of results on AtCoder(Java-Java)* and CodeChef(C-C)* (Table 5.9a) datasets follow a similar pattern of AtCoder(Java-Java) and CodeChef(C-C) dataset. RUBHUS outperforms the baseline model and the other tools by a considerable margin. However, some performance drop in the classification metrics of all the tools is also observed, which can be attributed to the change in domain shift of the training and testing data. For the SeSaMe and BCB datasets, the results reported in Table 5.9b are consistent with the results obtained from other experiments, which further ascertain the generalizability of RUBHUS on unseen code snippets.

> **RQ 5.3:** RUBHUS *can successfully identify code clones in statically typed languages having different programming paradigms with high precision and recall when compared to other single-language and cross-language clone detection tools.*

5.5.4 RQ 5.4: How Effectively Can RUBHUS Detect Clones in Dynamically Typed Languages?

In this RQ, we demonstrate the efficacy of RUBHUS in detecting clones in dynamically typed languages. Table 5.10 shows the classification performance of RUBHUS and other tools on the AtCoder (Python-Python) dataset of clones and non-clones pairs. RUBHUS outperforms both single-language and cross-language clone detection tools and CHEBCONV on the AtCoder (Python-Python) dataset by a significant margin. This shows that RUBHUS can learn similarities

Table 5.10: Classification performance of RUBHUS and other baselines on dynamically-typed language in single-language setting.

Tool	AtCoder (Python-Python)			AtCoder (Python-Python)*		
	Precision	Recall	F1-score	Precision	Recall	F1-score
PCCD	50	100	70	30	55	39
CLCDSA	72	92	81	64	85	73
TBCCD	75	89	82	69	89	78
FA-AST+GGNN	90	88	89	72	75	73
FA-AST+GMN	90	100	98	80	88	84
CHEBCONV	78	75	76	50	52	51
XCODE	90	92	91	82	80	81
C4	93	94	93	81	86	83
GRAPHCODEBERT	91	91	91	82	81	81
RUBHUS*	91	90	90	78	79	78
RUBHUS	94	97	95	90	91	90

between code snippets irrespective of the typing of the languages.

To evaluate the generalizability of RUBHUS in dynamically typed languages, following a methodology similar to that defined in Section 5.4.1.1, we create another dataset, AtCoder (Python-Python)*. In this dataset, the train and test files have no overlap, highlighting substantial semantic differences between the two sets. The results of this experiment, as shown in Table 5.10, demonstrate that RUBHUS outperforms other tools in terms of generalization capabilities.

RQ 5.4: RUBHUS *succeeds in identifying clones in dynamic typed language with high precision and recall when compared to other single-language and cross-language clone detection tools.*

5.5.5 Performance Analysis of RUBHUS: Model Size, Dataset Processing, and Training/Evaluation Time.

We conduct performance analysis of RUBHUS in terms of model size, dataset processing time, and training/evaluation time. The RUBHUS model has 275K parameters with model size of 1.1MB, and all experiments are run on a Linux machine equipped with an Intel Xeon(R) processor, 24 CPU cores, and 32GB memory. To create FE-ASTs for the entire AtCoder and CodeChef dataset, we process the dataset as a whole, which took around 90 minutes. To optimize GPU utilization,

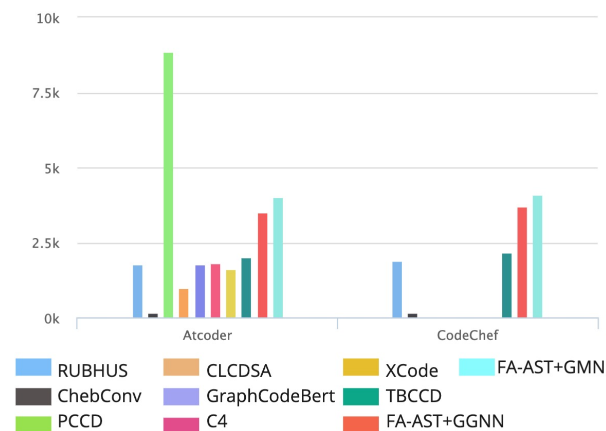


Figure 5.15: Training time analysis on the AtCoder and CodeChef datasets.

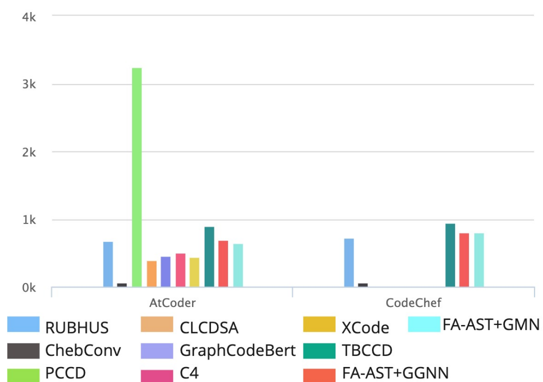


Figure 5.16: Evaluation time analysis on the AtCoder and CodeChef datasets.

we use an adjacency list instead of an adjacency matrix, and we connect sparse graphs into a single disconnected graph while preserving node identities, enabling us to batch process 32 pairs of graphs simultaneously. In terms of training and evaluation time, we compare RUBHUS with several baseline models, as shown in Figures 5.15 and 5.16. Our evaluation shows that CHEBCONV, the simplest technique, took the least time for training and evaluation on both datasets. In comparison, XCODE, C4, and GRAPHCODEBERT, which were pre-trained on large-scale code bases, required only fine-tuning for code clone detection, resulting in less time spent on training. However, the training time of RUBHUS was comparable to these techniques and significantly less than that of PCCD, TBCCD, FA-AST+GGNN, and FA-AST+GMN. Furthermore, all models require a comparable amount of time for evaluation. Overall, the results of our code clone detection experiments and time-performance analysis confirm that RUBHUS is capable of detecting more clones and is more time-efficient.

5.6 Discussion

Our results indicate that RUBHUS can successfully identify clones under diverse experimental settings. It also outperforms state-of-the-art tools in terms of the evaluation metrics used.

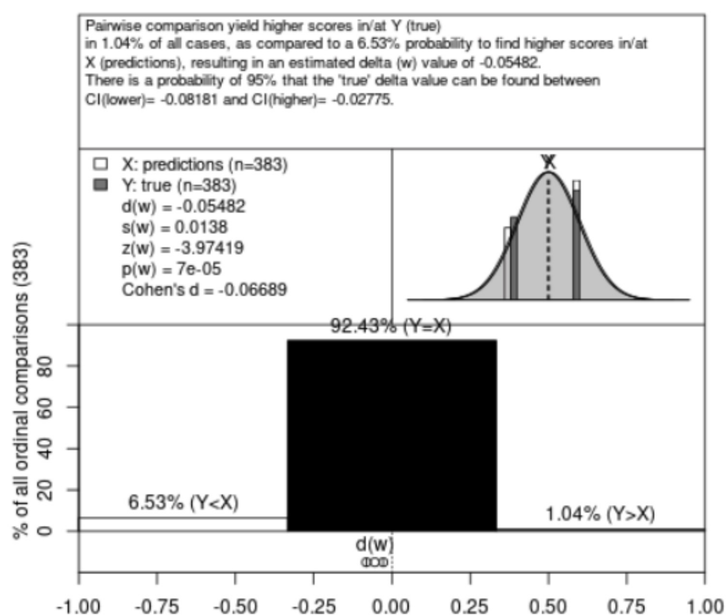


Figure 5.17: Cliff's delta (within) & 95% Confidence Interval (2-tailed)

5.6.1 FE-ASTs for Representation Learning and Similarity Prediction

Different languages vary substantially in syntax. Thus, it becomes essential to understand semantics to learn similarities across languages. Contrary to the tools that use shallow and textual features like program tokens, we use ASTs representing syntactic structures and abstract program-specific details. We also enrich ASTs with control and data flow edges to make the program's semantic information explicitly available within the ASTs. This source code representation enables us to capture well-defined semantics and rich additional information that is otherwise not visible using lexical and token sequences. ASTs also eliminate the need to have a common intermediate representation such as the .NET framework, making RUBHUS capable of detecting clones across diverse programming languages.

5.6.2 Role of GNNs and Semi-Supervised Learning

Exploiting the structured semantics of a program helps us to learn the program's functionality which is essential for similarity detection. The majority of learning-based techniques overlook the structured semantics of a program while learning high-level features. Therefore, we represent the program's semantic and syntactic structure through FE-ASTs and learn program features

with a semi-supervised GNN framework. The framework jointly learns AST representation (in an unsupervised manner using the principle of MI maximization) and trains a binary classifier (in a supervised manner to predict code similarity). This unsupervised learning of AST's node representations helps us capture rich semantics available at various substructures of AST, including nodes and edges, so that the global AST representation encodes all aspects of semantics and syntactic information present at various levels of AST. This helps the binary classifier learn similarity across code snippets. Our key insight is that exposing these semantics explicitly as a structured input to a GNN model allows us to learn similarities across code snippets and lessens the need for an extensive training dataset and deep neural networks.

To assess performance gain by RUBHUS in detecting cross-language clones, we use cliff's delta statistic [183]. It is a non-parametric effect size measure that quantifies the amount of difference between two groups of observations. We use a statistically significant sample size of 383 code pairs from the AtCoder dataset with a confidence level of 95 and confidence interval of 5 to compute the cliff delta score. As shown in Figure 5.17 the cliff delta score is -0.05 with Cohen's score of 0.06 when the distributions are normal with a continuous scale, which means that 92% of the time the value of predicted label matches with the actual label (i.e., y=x).

5.6.3 Qualitative Analysis of the Learned Feature Space

To visualize high dimensional latent feature space learned by RUBHUS, we use the t-SNE [184] dimensionality reduction technique. We select ten random problems from the AtCoder dataset and generate around 10K cross-language pairs. We feed these pairs as input to the model to obtain embeddings from the final layer. The embeddings are then visualized using the t-SNE technique. Figure 5.18 shows the latent feature space of RUBHUS, indicating that both the clone and non-clone classes are fairly separable and distant. This implies that RUBHUS is capable of learning semantic information to differentiate between clone and non-clone pairs.

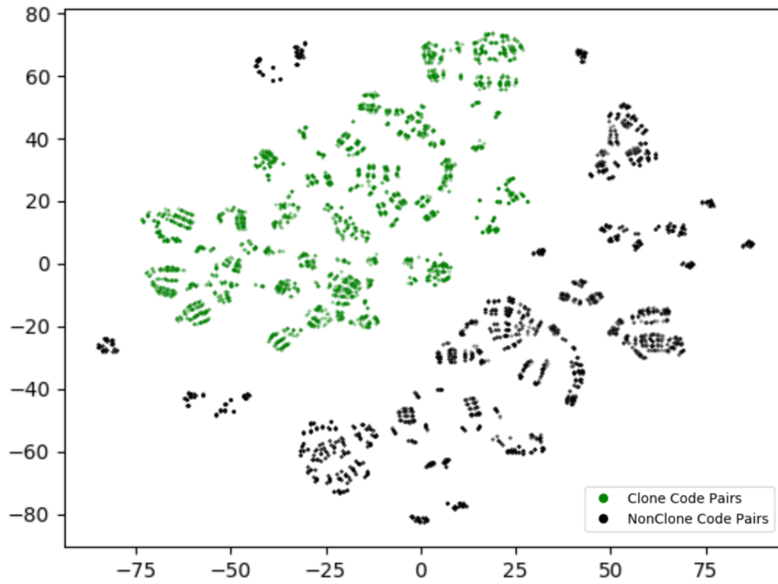


Figure 5.18: t-SNE plot of graph embeddings of clone and non-clone pairs of RUBHUS on the AtCoder dataset.

5.6.4 Ablation Study

We perform an ablation study to understand the effectiveness of each component of RUBHUS. To examine the impact of unsupervised training on RUBHUS, we train our main model in supervised way only on the AtCoder dataset. After removing the unsupervised loss, we observe a **4%** degradation in the F1 score (**94%**) of RUBHUS affirming that unsupervised training, which aims to learn different aspects of information across substructures of ASTs, helps to learn better program representation.

In the next experiment, we check the importance of the Set2set pooling by replacing it with Global Average Pooling (GAP), which simply averages node representations to form the high-level program representation. With this experimental setting, we achieve an F1-score of **93%**, depreciating RUBHUS's performance by **5%**. This implies that Set2Set pooling at the graph readout layer learns a coarsened graph representation that preserves much of the syntactic and structural information, improving program similarity prediction.

5.6.5 Limitations

We developed a semi-supervised tool RUBHUS to predict code similarity, which is expensive in terms of labeled dataset procurement. We plan to overcome this limitation by using transfer learning techniques in the future. To learn program features, we use AST representations, which are syntactic representations and capture limited program semantics. However, we enrich ASTs with control and data flow edges and use these diverse edge types to model program semantics. We also acknowledge that a stronger semantic program representation, such as a program dependence graph (PDG) might improve the results. But, PDGs are complex and computationally intensive. Therefore, we use flow-enriched AST representation in RUBHUS to maintain the balance between performance and complexity.

5.7 Chapter Summary

The detection of code clones has attracted considerable attention due to its applications in software maintenance. However, only a few approaches have been proposed to detect clones across programming languages with varying programming styles and paradigms. These approaches do not exploit the source code's rich syntactic and semantic information to predict similarity. In this chapter, we proposed a tool, RUBHUS, to detect language-agnostic code clones. RUBHUS uses AST program representation and GNNs to learn high-level program features, which are later used by a binary classifier for similarity prediction.

We assessed the performance of RUBHUS with respect to its ability to predict code similarity in three contexts: i) statically typed languages, ii) dynamically typed languages, and iii) both statically and dynamically typed languages. To support the evaluation, we created a dataset of cross-language code clones consisting of C and Java code pairs. We also compared RUBHUS with other cross and single-language code clone detection tools and a baseline GNN model CHEBCONV. Our results show that RUBHUS outperforms state-of-the-art tools in all phases by a substantial margin.

In the future, we would like to explore a combination of program representations for cross-

language code clone detection. We also aim to evaluate RUBHUS on real-world codebases to understand the extent to which it can assist in software maintenance.

Chapter 6

Improving Semantic Code Clones Detection with Adversarial Domain Learning

6.1 Introduction

Clone detection is a crucial task in software engineering that aims to identify similar code fragments within or across multiple codebases [185, 186] . Clones can be either benign [187, 188, 189, 190] or problematic [191, 192, 193, 194, 195, 26, 196], depending on their usage context. Clone detection facilitates functionality reuse, consistency in implementation, bug fixing and improved code performance. However, excessive cloning can lead to problems such as increased complexity, difficulty in maintaining code, bugs, security vulnerabilities, and difficulty in tracking down the source of bugs. Therefore, it is important to use clone detection tools to manage clones in a software system and to consider the benefits of code reuse before introducing new clones in a codebase.

Various techniques ranging from traditional [197, 198, 199, 200, 201, 202, 185] to learning-based [108, 106, 93, 61, 107, 47, 50, 51] approaches have been proposed for efficient detection of clones. Traditional clone detection tools use lexical and syntactical analysis, which may

```java
void addModule(ModuleModel m, boolean
↪ isInternal) {
  synchronized(instLock) {
  if (!this.moduleModels.contains(m)) {
    checkDestroyed();
    this.moduleModels.add(m);
    m.setInternalId (buildInternalId
    ↪ (getInternalId(),
    ↪ moduleIndex.getAndIncrement()));
    if (!isInternal)
      pubModuleModels.add(m);
    }
  }
}
```

Listing 6.1: Method to add a module with synchronization.

```java
void addApplication(ApplicationModel
↪ a){
  checkDestroyed();
  synchronized(instLock) {
  if (!this.applicationModels.contains
  ↪ (a)){
    a.setInternalId (buildInternalId
    ↪ (getInternalId(),
    ↪ appIndex.getAndIncrement()));
    this.applicationModels.add(a);
    if (!a.isInternal())
      this.pubApplicationModels.add(a);
    }}
}
```

Listing 6.2: Method to add an application with synchronization.

Figure 6.1: A semantic clone pair example from Apache Dubbo detected by applying CLODIA on HOLMES and TBCCD which otherwise was detected as false negative by both the tools.

```java
public static final boolean
↪ deleteAll(File what) throws
↪ FileNotFoundException, IOException
↪ {
  if (what == null)
    return false;
  if (what.isDirectory()) {
    File[] files = what.listFiles();
    for (int i = 0; i < files.length;
    ↪ i++)
      deleteAll(files[i]);
  }
  return what.delete();
}
```

Listing 6.3: Delete all files and subdirectories in a directory.

```java
public static void
↪ recursiveDeleteDirectory(File
↪ cacheDir) {
  if (cacheDir.isDirectory()) {
    File[] files =
    ↪ cacheDir.listFiles();
    for (int i = 0; i < files.length;
    ↪ i++) {
      File file = files[i];
      recursiveDeleteDirectory(file);
      }
  }
  else {cacheDir.delete();}
}
```

Listing 6.4: Recursively delete a directory and its contents.

Figure 6.2: A semantic clone pair example from BCB dataset detected as true positive by HOLMES and TBCCD.

have high false negative rates and low recall. Hence, learning-based clone detection tools were proposed as an alternative approach. These techniques learn high-level patterns and features from a labeled dataset of clones. They then classify unseen code fragments as either clones or non-clones based on their learned representations, identifying structural and semantic similarities. Unlike traditional techniques, learning based techniques can capture complex structural and contextual information from code and hence are effective in detecting semantic similarities between code snippets.

Despite the potential advantages that learning-based techniques offer, they come with several challenges. One of the primary obstacles is the issue of domain shift, which arises from

the disparity between the data distribution in the training set and the real-world deployment environment. This mismatch degrades the performance of these models in real-world datasets, making them practically unusable. For example, even state-of-the-art semantic clone detection tool like HOLMES [147], trained on the BigCloneBench (BCB) dataset [62], struggle to identify clone pair in Listings 6.1 and 6.2, despite its effectiveness in recognizing Type-4 clones within the BCB dataset. The reason for the performance degradation of these supervised techniques on new unseen data is attributed to the fact that, while the training dataset is extensive, it may not fully encompass the diversity of coding styles, libraries, and frameworks found in real-world projects. Additionally, new data may have new patterns or features not well-represented in the training set. Adapting these techniques for unseen data often necessitates re-training the models. However, labeled dataset creation for clones in every new domain is a significant challenge as it's a labor-intensive subjective task. These restrictions limit the applicability of these supervised learning techniques in real-world scenarios.

In response to this challenge, Liu et al. [129] introduced a human-in-the-loop mechanism that combines transfer learning and active learning to adapt neural clone detectors to new code repositories that may contain numerous unseen functionalities. Although Liu et al.'s mechanism used for adapting neural clone detectors to new code repositories is effective in reducing the amount of annotations required. However, it still relies on human annotation, which can be time-consuming and resource-intensive, especially for larger repositories and cannot be autonomously adapted to different codebases and domains.

Adversarial Unsupervised Domain Adaptation (UDA) [60] is a machine learning technique that enables the autonomous adaptation of pre-trained neural models to unseen repositories, all without the need for labeled data in the target domain. Thus, UDA offers an appealing alternative for adapting a trained clone detection tool to a new dataset, eliminating the necessity for re-training or fine-tuning the model on the new unlabeled dataset.

UDA techniques typically introduce a domain classifier, often referred to as an adversarial discriminator. This discriminator's purpose is to discern between source and target domain data based on their feature representations. Its goal is twofold: to align these feature representations by recognizing common patterns amidst the diversity and to become uncertain about the origin

of a given feature vector, whether it comes from the source or target domain.

However, applying existing UDA methods in clone detection poses a significant challenge due to the intrinsic dissimilarities between source and target datasets. The challenge lies in the fact that feature representations of a clone pair in the target domain can vastly differ from the learned feature representation of a seemingly similar clone in the source domain. These differences stem from variations in functionality, size, complexity, and coding style.

For example, consider the comparison between a clone pair from Apache Dubbo (Figure 6.1) and one from the BCB dataset (Figure 6.2). Although both pairs fall under the category of semantic clones, they exhibit significant dissimilarity. The Apache Dubbo code pair involves adding a module to a collection and verifying its existence before addition, whereas the BCB dataset's code pair deals with recursive directory deletion. These tasks fundamentally differ in nature, highlighting substantial disparities in functionality, intricacy, and size.

Consequently, these inherent variations in feature representations among code pairs pose a challenge for the domain classifier to effectively align the feature space of these two code snippets. The domain classifier assumes that pairs from the same class will have similar feature representations, i.e., the two clone pairs will be very close to each other in the feature space, especially when compared to non-clones. However, as demonstrated this assumption does not hold true. Thus, simply minimizing the distance between feature representations does not effectively mitigate domain shift or align pairs in a way that a binary classifier trained on the labeled source dataset works accurately for the unseen target dataset. The complexity arises from the need to account for these significant differences in feature representations while aligning the domains to enable a successful application of UDA techniques in clone detection.

To further validate this claim, we applied various existing domain adaptation approaches to state-of-the-art clone detection tools. The results show that majority of these techniques do not offer significant improvements when applied to the unseen target domain. To address this issue, we propose in this chapter, an adversarial UDA approach and tool, CLODIA, for clone detection by learning a representation space that retains discriminative power on both the (labeled) source and (unlabeled) target domains while keeping representations for the two domains well-separated.

CLODIA improves the model's performance and reduces the need for manual annotation. Our results show that, with CLODIA we get upto ∼60% maximum performance improvement in the F1-score when adapting a model trained on competition programming dataset (GCJ) to real-world dataset curated from Apache projects. Overall, we achieve a mean performance improvement of ∼44% when using CLODIA compared to other DA baselines methods. We make the following contributions in this research:

1) Proposal of a novel technique, CLODIA, which uses multiple latent spaces for domain adaptation for improving the generalizing capabilities and performance of supervised clone detection techniques in unseen domains and datasets.

2) Extensive evaluation of CLODIA on various datasets, including programming competition and open source datasets and also on a handcrafted dataset of clones curated from real-world software systems. We also compare and assess the effectiveness of various state-of-the-art UDA techniques against CLODIA in enhancing the generalization capabilities of state-of-the-art neural clone detection tools, utilizing standard evaluation metrics.

6.2 Approach Overview

We present CLODIA, an adversarial multispace domain adaptation (DA) technique designed to enhance the performance and adaptability of supervised clone detection methods on previously unseen datasets. CLODIA aligns multiple domains within a single joint learning framework for semantic code similarity detection by learning multiple mappings from each domain to a common latent space that captures shared features across domains. To evaluate CLODIA's effectiveness, we applied it to three state-of-the-art clone detection tools: HOLMES, TBCCD, and FA-AST.

In this section, we will demonstrate the functionality of CLODIA using HOLMES as a reference architecture. As shown in Figure 6.3, CLODIA is composed of several modules, starting with the Feature Learning Module (FLM). The primary purpose of the FLM is to automatically learn and extract relevant features from input code snippets, thereby facilitating code similarity detection.

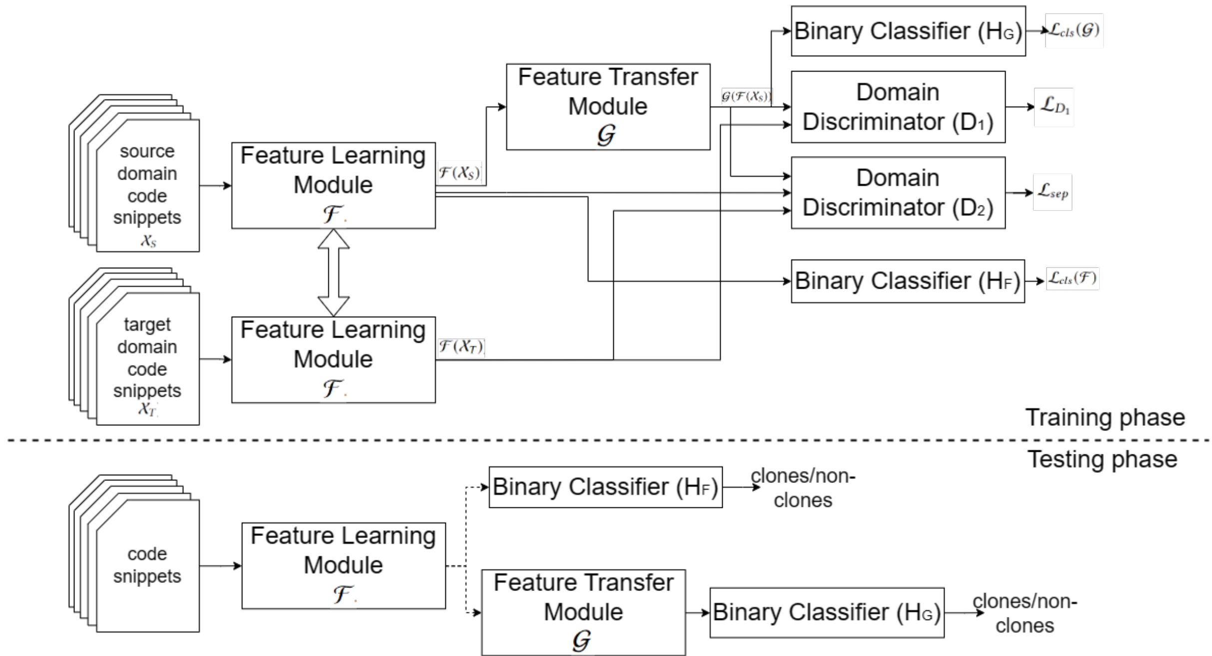


Figure 6.3: The training and testing phase of CLODIA.

FLM is denoted as a function, symbolized by $\mathcal{F}$. This function maps the source domain code snippets (represented by $\mathcal{X}_S$) to their corresponding representation space (indicated by $\mathcal{F}(\mathcal{X}_S)$) and the target domain code snippets (represented by $\mathcal{X}_T$) to their respective representation space (indicated by $\mathcal{F}(\mathcal{X}_T)$). The FLM is designed to learn discriminative features thereby allowing binary classifier to effectively differentiate between clones and non-clones pairs.

FLM is based on existing clone detection techniques and accepts a pair of code snippets as input, outputting a learned high-level program representation. For instance, in CLODIA (using HOLMES), input code snippets are initially converted into program dependence graphs. Subsequently, node feature matrices and edge feature matrices are extracted from these graphs. These matrices are then passed through multi-head attention-based graph neural networks to learn node feature representations. Following this, a graph pooling module processes the node feature representations to obtain high-level learned program representation. This entire procedure is carried out for both pairs of input code snippets, and the resulting high-level program representations are concatenated to create the learned feature representation for the given input code pair.

After acquiring the learned representation of a code pair from the source domain, we pass

it through the Feature Transfer Module (FTM), denoted as $\mathcal{G} : \mathbb{Z} \rightarrow \mathbb{Z}$. It is important to note that, only source domain data, $\mathcal{F}(\mathcal{X}_S)$, passes through the FTM, as we train FLM and Binary Classifier solely on labeled source data and not on unlabeled target domain data.

The Feature Learning Module (FLM) is designed to learn domain-specific features from the source data. However, to ensure the domain discriminator functions effectively and aligns the source and target distributions, domain-invariant features are necessary. This is because domain-specific features can differ significantly between the source and target domains, making it challenging to align these representations effectively. For instance, code pairs from different data sources may have differences in functionality and coding style, as evident in Figure 6.1 and Figure 6.2.

Thus, due to the inherent diversity between the source and target domain code pairs, it is difficult to minimize the distance and also to align them adversarially in such a way that classifier trained on source domain can be applied on target domain. To this end, we introduce FTM that separates target features from source features while simultaneously aligning them with an auxiliary domain of the transformed source features.

FTM transforms learned feature representation of code pair from source domain ($\mathcal{F}(\mathcal{X}_S)$) into $\mathcal{G}(\mathcal{F}(\mathcal{X}_S))$, so that the transformed feature distribution of source domain code pair ($\mathcal{G}(\mathcal{F}(\mathcal{X}_S))$) can be aligned with the feature distribution of target domain code pair ($\mathcal{F}(\mathcal{X}_T)$) using a *domain adversarial objective* which is defined as:

$$\mathcal{L}_{D_1} = \mathbb{E}x \in \mathcal{X}_S \log D_1(\mathcal{G}(\mathcal{F}(\mathcal{X}_S))) + \mathbb{E}x \in \mathcal{X}_T \log(1 - D_1(\mathcal{F}(\mathcal{X}_T)))] \tag{6.1}$$

$$\mathcal{L}_{adv} = \mathbb{E}_{x \in \mathcal{X}_T} \log D_1(\mathcal{F}) \tag{6.2}$$

Where $D_1 : \mathbb{Z} \rightarrow (0, 1)$, is a domain discriminator applied to discriminate between source and target domain, $\mathbb{E}x \in \mathcal{X}_S \log D_1(\mathcal{G}(\mathcal{F}(\mathcal{X}_S)))$ calculates the probability of the discriminator D_1 classifying the transformed source code pair coming from the source domain, and $\mathbb{E}x \in \mathcal{X}_T \log(1 - D_1(\mathcal{F}(\mathcal{X}_T)))$ calculates the probability of the discriminator D_1 classifying the

target samples as coming from the target domain. The domain adversarial objective, $\mathcal{L}_{D_1}$, is designed to minimize the discrepancy between the source and target domain feature distributions, so that binary classifier model trained on source domain performs well on target domain also. $\mathcal{L}_{adv}$ measures the adversarial loss for the target domain, which measures how well the FLM is able to "fool" a domain discriminator to minimize the discrepancy between the source and the target domain.

FTM is applied to improve feature alignment and transferability by providing an auxiliary space. It explicitly models the domain shift and integrates domain-invariant constraints, and transfers knowledge between domains, ensuring that features are more suitable for both domains. By minimizing the difference between source and target domain feature distribution, the FTM enables the FLM to extract shared features and use the classifier trained on the labeled source dataset to generalize effectively on the unlabeled target dataset. Thus, *domain adversarial objective* trains the FLM to fool the domain discriminator by generating domain-invariant features. This enhances its ability to extract meaningful features for the primary task of clone detection in the unseen target domain.

The code pairs within the same class may exhibit inherent diversity in syntax, coding style, and functionality. For instance, clone pairs implementing factorial (recursively and iteratively) and search algorithms (binary and linear) are categorically similar but substantially different in their implementations. Thus their representations would also be very far apart and if they belong to different domains it would be difficult to align them just by reducing the domain distance. To address this challenge, a discriminator (D2) and a domain separation objective are introduced. Although, in domain adaptation, the goal is indeed to minimize the distance between source and target domains. The domain separation objective aims to serve this purpose by encouraging a more distinguishable representation space for the FLM module. This is done to improve the FLM's ability to adapt and generalize to the target domain. This helps the FLM to better learn the differences between the source ($\mathcal{F}(\mathcal{X}_S)$) and target domains ($\mathcal{F}(\mathcal{X}_T)$), making it more effective at transferring knowledge between them. We formulate this objective using a separate Discriminator $D_2 : \mathbb{Z} \rightarrow (0, 1)$ which is defined as:

$$\mathcal{L}_{sep} = \mathbb{E}x \in \mathcal{X}_S \log D_2(\mathcal{F}) + \frac{1}{2}\mathbb{E}x \in \mathcal{X}_S[\log(1 - D_2(\mathcal{G}))] + \mathbb{E}_{x\in\mathcal{X}_T}[\log(1 - D_2(\mathcal{F}))] \quad (6.3)$$

The objective function $\mathcal{L}_{sep}$ ensures that not only is the source space $\mathcal{F}(\mathcal{X}_S)$ pushed apart from the target space $\mathcal{F}(\mathcal{X}_T)$, but it is also pushed apart from the augmented source space $\mathcal{G}(F(\mathcal{X}_S))$. D_2 acts as another domain classifier, which is trained to learn to distinguish between the source and target domains. The primary goal of this discriminator is to differentiate between the two domains, while the FLM's goal is to minimize this distinction. Thus, the FLM and D_2 work together through an adversarial process, where FLM tries to create representations that are indistinguishable between the source and target domains, while D_2 tries to identify which domain the representation belongs to.

This objective aims to increase separation between source and target domains. Thus, by training on loss from D2, FLM is trained to enable D2 to discern the domain label of input. This strategy compels FLM to learn more domain invariant features, facilitating robust domain alignment.

Thus, throughout training, a minmax game occurs between D1 (bringing domains closer) and D2 (pushing representations apart). Alternatively, one can view D2 introduces an extra validation for FLM, prompting it to acquire domain-invariant features. This encourages a more nuanced alignment of domains, particularly those with substantial differences. By compelling FE to prioritize domain-invariant features over domain-specific ones, it effectively enhances the system's adaptation and generalization capabilities on target domain.

Finally, we employ binary classifier (denoted as H) on a pair of code snippet to measure code similarity. H is again taken from the existing clone detection technique, and for a given pair of code snippet it outputs 1, if the input pair is clone and 0 otherwise. H is trained on labeled source domain data using a *supervised binary classification objective*. Note that, we train two classifiers, both on the representation produced by FLM ($\mathcal{F}(\mathcal{X}_S)$) and FTM ($\mathcal{G}(\mathcal{F}(\mathcal{X}_S))$) using binary classifiers H_F and H_G. At the time of inference we use the classifier from FLM (H_F). The FTM transforms the source domain code representations in an auxiliary domain with which

the target domain code representations are aligned. Now, to have a better alignment of the target domain with the above mentioned auxiliary domain, the auxiliary domain should have well separated vectors for clones & non-clone pairs, thereby necessitating a classifier after the FTM. The objective is defined as follows:

$$\mathcal{L}_{cls}(\mathcal{F}) = E_{(x_1,x_2)\in\mathcal{X}_S\times\mathcal{X}_S}\left[y\log H_F(f_1,f_2) + (1-y)\log(1-H_F(f_1,f_2))\right] \tag{6.4}$$

$$\mathcal{L}_{cls}(\mathcal{G}) = E_{(x_1,x_2)\in\mathcal{X}_S\times\mathcal{X}_S}\left[y\log H_G(g_1,g_2) + (1-y)\log(1-H_G(g_1,g_2))\right] \tag{6.5}$$

Where $g_i = G(F(x_i))$, $f_i = F(x_i)$ and $y = 1$ if x_1 and x_2 are clones and 0 otherwise. Note that, we aim to learn FLM ($\mathcal{F}$) and FTM($\mathcal{G}$) in such a way, that their embedding can be directly used to compute similarity between the code snippets, x_1 and x_2.

Thus, CLODIA aims to effectively transfer knowledge between a source and target domain despite their differences. The FLM is trained to extract essential features from the source domain, and the FTM then transforms these features into a suitable representation for minimizing the distance from the target domain code snippets. This process is enhanced by the domain adversarial and domain separation objectives, which work together to encourage the model to learn domain-invariant code features, adapt to domain shifts, and generalize effectively to the target domain. Through this end-to-end approach, the model focuses on learning shared features between domains and improves its adaptability and generalization capabilities, ultimately making the binary classifier, trained on the labeled source dataset, effective on the unseen target domain as well. Thus, we aim to achieve the following capabilities through these objectives:

1) If $x_1, x_2 \in \mathcal{X}_S$, then $\mathcal{F}(x_1)$ and $\mathcal{F}(x_2)$ will be close if they are clones and far away otherwise, due to the discriminative power acquired from optimizing H_F.

2) If $x_1, x_2 \in \mathcal{X}_T$,then $\mathcal{F}(x_1)$ and $\mathcal{F}(x_2)$ will be close if they are clones and far otherwise, due to the discriminative power acquired by optimizing H_G with domain adversarial training.

Finally, our training objective for CLODIA is defined as follows:

$$\mathcal{L}_{\mathcal{F}} = \frac{1}{2}\big[\mathcal{L}_{cls}(\mathcal{G}) + \mathcal{L}_{cls}(\mathcal{F})\big] + \lambda_1 \mathcal{L}_{adv} + \lambda_2 \mathcal{L}_{sep} \tag{6.6}$$

$$\mathcal{L}_{\mathcal{G}} = \mathcal{L}_{cls}(\mathcal{G}) + \lambda_2 \mathbb{E}_{\mathcal{X}_S} \log(1 - D_2(\mathcal{G})) \tag{6.7}$$

In clone detection, adapting a model from a source domain to a target domain is challenging due to the wide diversity in coding styles, languages, and functionalities across domains. This diversity often causes the adapted model to focus on a few dominant patterns, effectively collapsing into these modes. This is because the model tends to prioritize easily recognizable features, neglecting the full range of code variations in the target domain. Moreover, since the target domain lacks labeled data, the model relies heavily on source domain knowledge, which can further limit its diversity. Consequently, the model may assume that the target domain resembles the source domain, leading to a narrow set of predictions that may not accurately capture the nuances of the target domain. Thus, to prevent mode collapse, we add an additional regularization loss similar to [203]. Assuming that the representation of the source domain is already optimal, we aim to regularize the features of source examples to be similar to those from the reference network $\mathcal{F}_{ref} : \mathcal{X} \rightarrow \mathbb{Z}$. This reference network is pre-trained on labeled source data and fixed during the training of $\mathcal{F}$. Additionally, we introduce a similar regularization to target examples, which avoids collapsing and allows more room for target features to diverge from the original representations. Hence, the feature reconstruction loss is defined as:

$$\mathcal{L}_{\text{recon}} = -\lambda_3 \frac{1}{2} \sum_{x \in \mathcal{X}_S} |\mathcal{F}(x) - \mathcal{F}_{\text{ref}}(x)|_2^2 + \lambda_4 \frac{1}{2} \sum_{x \in \mathcal{X}_T} |\mathcal{F}(x) - \mathcal{F}_{\text{ref}}(x)|_2^2 \tag{6.8}$$

Finally, the training objective for $\mathcal{F}$ becomes, while it remains the same for $\mathcal{G}$:

$$\mathcal{L}_{\mathcal{F}} = \frac{1}{2}(\mathcal{L}_{cls}(\mathcal{G}) + \mathcal{L}_{cls}(\mathcal{F})) + \lambda_1 \mathcal{L}_{adv} + \lambda_2 \mathcal{L}_{sep} + \mathcal{L}_{\text{recon}} \tag{6.9}$$

We train the entire pipeline end to end with losses mentioned in equations 6.9 and 6.7 while

Table 6.1: Dataset Statistics & Percentage of clone-types in BCB.

(a) Dataset Statistics

Dataset	**Language**	**Files**	**Clone**	**Non-clones**	**Methods**	**LOC**
GCJ	Java	$9.4k$	$440k$	$500k$	$58k$	$909k$
BCB	Java	$9.1k$	$650k$	$650k$	$9.1k$	$267k$
AC	Java	$6k$	145	300	450	$24k$

(b) % of Clones in BCB Dataset

Type	T1	T2	ST3	MT3	WT3/T4
%**clones**	0.005	0.001	0.002	0.010	0.982

alternating update between (F,G,D2) and D1.

6.3 Experiment Design

In this study, we focus on examining the impact of CLODIA on the performance of supervised clone detection tools, with an emphasis on understanding the influence of source and target dataset selection. The experiment's objectives are twofold:
1) Objective 1 (O1): To compare and assess the performance of CLODIA against different domain adaptation techniques to understand how these techniques affect the performance of code clone detection models'.
2) Objective 2 (O2): To determine the impact of selecting source and target domains with varying characteristics, such as size, complexity, and coding style, on the performance of code clone detection and domain adaptation techniques.

6.3.1 Datasets

We use following datasets in our experiments:

1. **Programming competition dataset**: This dataset is taken from existing work [147]. It is obtained from GoogleCodeJam (GCJ), an annual programming competition hosted by Google. In this competition, participants solve various programming problems and submit their solutions to Google for testing. Only those solutions that pass all test cases are published online. Each

competition comprises several rounds. The solutions for the same problems were functionally similar (clones), belonging to Type 3 and Type 4 clone categories, whereas those for different problems were dissimilar (non-clones). GCJ dataset contains 9436 solutions from 100 different functionalities of GCJ, thereby building a substantial and representative dataset for evaluation. Detailed statistics can be found in Table 6.1a.

2. **Open-source projects:** We use two datasets curated from open-source projects:

1) ***BigCloneBench (BCB) dataset:*** We also use BigCloneBench (BCB) [62] open source projects dataset. Svajlenko et al. developed the dataset from the IJAdataset-2.0[1], which contains $25K$ open-source Java projects and $365M$ lines of code. The authors mined frequently used functionalities from the IJaDataset to build the BCB dataset. The dataset includes ten functionalities, which covered $6M$ clone pairs and $260K$ non-clone pairs. The authors categorizes clone types into five categories: Type-1, Type-2, Strongly Type-3 (ST3), Moderately Type-3 (MT3) and Weakly Type-3+4 (Type-4) (WT3/T4) clones. Since there was no consensus on minimum similarity for Type-3 clones, Type-3 and Type-4 clones are categorized based on their syntactic similarity. Thus, ST3 clones have at least 70% similarity at the statement level and contain some statement-level differences. The clone pairs in the MT3 category share at least half of their syntax but have a significant amount of statement-level differences. The WT3/4 code clone category contains pairs that share less than 50% of their syntax. Tables 6.1a and 6.1b summarize the data distribution of the BCB dataset.

2) ***Apache Clones (AC) dataset:*** In the pursuit of assessing CLODIA's efficacy in boosting the performance of clone detection techniques in real-world context, we curated a code clones dataset, from Apache Software Foundation projects[2]. Our initial step involved the selection of a spectrum of Apache Java projects, revered for their widespread adoption across diverse domains. These projects exhibited discernible functional resemblances or addressed cognate problem domains. This selection criterion ensured that our dataset faithfully represents real-world scenarios, encompassing an extensive repertoire of functionalities and coding paradigms from an array of projects. Specifically, we elected following middleware and software infrastructure projects: commons-collections, dubbo, thrift, zookeeper, kafka, pulsar and servicecomb-pack.

[1] https://sites.google.com/site/asegsecold/projects/seclone
[2] https://github.com/apache

These projects are interconnected by their roles in facilitating communication, data sharing, and common utilities across diverse software systems, making them suitable representatives for our study.

To ensure the code pairs were of suitable size for clone analysis, we opted for a code length range between 60 to 80 lines of code (LOC), subsequent to the removal of comments. The choice of this range was a deliberate effort to strike a balance between the incorporation of substantively representative code functionality and the avoidance of code snippets that were excessively terse or prolix. Code segments within this chosen range were deemed sufficiently comprehensive to encompass substantial logical constructs while remaining amenable to manual evaluation also. After applying these filtering criteria, we got around 450 Java methods and the pairwise combination of these methods resulted in around $\sim$ 100K code pairs. Out of these 100K code pairs, we sampled 1K pairs with a 99% confidence level and a 5% margin of error, for manual labeling.

The authors manually validated the selected code pairs for marking them as clones and non-clones. Initially, a pilot study was conducted, during which the authors individually assessed and labeled 20 randomly selected code pairs from the pool of 1K code pairs. Subsequent to the pilot study's conclusion, the authors deliberated their findings to dispel any potential ambiguities and to establish a unanimous consensus regarding the classification criteria distinguishing clones from non-clones. Following this phase, the authors equally divided the remaining code pairs among themselves for manual validation, expending an aggregate of approximately 60-70 hours in finishing this manual labeling exercise.

Out of the 1K code pairs, 145 were identified as clone pairs, while the remaining pairs were categorized as non-clones. Finally, we included all code pairs marked as clones and sampled approximately 300 non-clone pairs to create the final dataset. Detailed statistics of this dataset is reported in Table 6.1a.

The selected datasets exhibit a wide range of characteristics, including size, functionality, complexity, and diversity. This broad spectrum allows for a comprehensive evaluation of how dataset attributes impact domain adaptation and the performance of various domain adaptation techniques when integrated with different supervised clone detection methods.

We perform experiments with different combinations of the datasets as source and target domains, aiming to assess how the choice of source and target domains affects the domain adaptation performance. We consider the following combinations of these datasets as source and target domains: *1) GCJ as both source and target (*$GCJ_{sn} \rightarrow GCJ_{un}$*), 2) GCJ as source and BCB as target (GCJ* $\rightarrow$ *BCB), 3) BCB as source and GCJ as target (BCB* $\rightarrow$ *GCJ), 4) GCJ as source and AC as target (GCJ* $\rightarrow$ *AC)*, and *5) BCB as source and AC as target (BCB* $\rightarrow$ *AC).*

The above combinations allow us to explore the impact of various factors on domain adaptation effectiveness. For instance, using GCJ for both source and target ($\text{GCJ}_{sn} \rightarrow \text{GCJ}_{un}$), with different functionalities reserved for each can help to isolate the effect of domain adaptation techniques from the dataset characteristics such as varying code complexity, code size, etc. By keeping one dataset constant and only changing the functionality reserved for the source and target domains, we assess how the choice of source and target domains affects the effectiveness of domain adaptation techniques for clone detection.

On the other hand, GCJ $\rightarrow$ BCB or GCJ $\rightarrow$ AC, enable us to evaluate the effectiveness of adapting a model trained on a competition dataset to an open-source dataset. Furthermore, we can also assess the impact of dataset size and complexity on domain adaptation effectiveness while using programming competition and open-source dataset. For instance, BCB dataset, consists of thousands of short code snippets, with limited set of functionalities (10) and many snippets have variations only in the sequences of API call and methods invoked. In contrast, GCJ is much larger, having 100 different functionalities and substantial differences in the program structure, syntax and semantics, as the submissions are made by independent programmers implementing the solutions from scratch. Moreover, code snippets in BCB or AC are typically concise and optimized for performance, while code fragments in GCJ may be more complex and diverse in terms of application domains, implementation style, etc. These differences may affect the adaptability of a clone detection model and it is important to evaluate the impact of such factors systematically.

6.3.2 Baseline Comparisons

In order to evaluate the performance of CLODIA objectively, we conducted a comparative analysis with the following baseline models:

6.3.2.1 Code Clone Detection Models

Acknowledging the substantial potential of large language models (LLMs) for clone detection, this study primarily delves into the exploration of domain adaptation techniques aimed at augmenting the generalization of smaller models. Recent studies, such as [204], have raised concerns about LLMs memorizing training data, prompting questions about their performance on new, unseen datasets. Consequently, fine-tuning LLMs may become imperative for specific use cases. Additionally, the widespread adoption of LLMs is not without challenges, encompassing limitations in rate limits, pricing, and concerns regarding data and code privacy.

In contrast, smaller models present a more practical and addressable solution for these concerns. However, their application is impeded by the scarcity of labeled datasets and the considerable curation efforts required. To address these challenges and to promote the adoption of supervised models, particularly in industrial contexts, we introduce CLODIA. It is crucial to highlight that even in scenarios where future training of large models becomes feasible, the necessity for a labeled target dataset for fine-tuning persists. CLODIA, with potential adjustments, can serve as a bridging solution. Furthermore, practical considerations such as cost, rate limiting, and policies leading to usage constraints guided our decision to refrain from a direct comparison with GPT models.

Thus, to assess the efficacy of CLODIA, we employ it in conjunction with distinct state-of-the-art supervised learning-based code clone detection tools: **HOLMES**[147], **TBCCD**[61], and **FA-AST**[107]. These tools were selected for their open-source availability and their state-of-the-art architecture and superior performance compared to numerous other tools in detecting clones. The selected code clone detection techniques represents a diverse array of methodologies for clone detection, enabling a thorough evaluation of how CLODIA can improve their performance

in unfamiliar datasets and domains. **Holmes**, leverages program dependence graphs, in tandem with Siamese graph neural networks, to measure code functional similarity between different code snippets. **Tbccd**, employs a token-enriched abstract syntax tree (AST) along with a tree-based convolutional neural network for clone detection. This approach focuses on the syntactic structure of the code and the token information, enabling it to identify similar code patterns effectively. **FA-AST** utilizes flow-augmented ASTs and incorporates two distinct graph neural networks, Gated Graph Neural Network (GGNN)[138] and Graph Matching Network (GMN)[123], for identifying clones. FA-AST enhances the traditional AST representation by incorporating data flow information, thereby improving its ability to detect semantic clones.

6.3.2.2 Domain Adaptation Techniques

To compare the effectiveness of Clodia in improving the performance of code clone detection tools on unseen domain, we compare and assess Clodia with following domain adaptation techniques:

1. **CORAL**[205]: CORAL (Covariance Matrix Adaptation in Deep Neural Networks) mitigates domain shift between source and target domains by aligning their second-order statistics. In deep neural networks, the second-order statistics of activations across layers contain crucial information about input data. CORAL focuses on aligning the covariance matrix of activations in a specific layer between source and target domains, encouraging the learning of domain-invariant features. To implement CORAL, activations for a given layer are extracted from both domains, and covariance matrices are computed to capture pairwise correlations and represent the second-order statistics of the layer.

2. **DANN**[78]: DANN (Domain Adversarial Neural Networks) seeks to learn a domain-invariant representation of input data by aligning feature representations across diverse domains. It modifies a standard neural network architecture with an additional domain classifier, taking intermediate feature representations as input and predicting the domain label. The DANN objective includes a classification loss, domain adversarial loss, and a gradient reversal layer. The domain adversarial loss frames a binary classification problem,

with the domain classifier predicting the feature representation's domain label while the feature extractor aims for domain-invariant features. The gradient reversal layer ensures reverse gradient flow during backpropagation for the domain adversarial loss.

3. **ADDA**[206]: ADDA (Adversarial Domain Adaptation) enhances model performance in a target domain by leveraging knowledge from a source domain. It achieves this by training a feature extractor and domain classifier concurrently. The feature extractor aims for domain-invariant representations, while the domain classifier distinguishes between source and target domains. Adversarial training aligns feature distributions, aiding the model's generalization to the target domain. Unlike DANN, ADDA often employs a two-module approach with a separate domain classifier, offering flexibility for adapting pre-trained source domain models without extensive modifications.

4. **UAN**[207]: Existing domain adaptation methods like DANN and ADDA often rely on prior knowledge about the relationship between label sets in the source and target domains, limiting their practicality in real-world applications. Universal Domain Adaptation (UDA) represents a more formidable challenge, as it operates in scenarios where no prior information about the label sets in either domain is available. Within UDA, the task is for a model to proficiently classify a target sample, correctly assigning it to a shared label set if applicable to both domains or marking it as "unknown" if it falls outside this common label set. The Universal Adaptation Network (UAN) plays a pivotal role in addressing the intricacies of UDA. UAN quantifies sample-level transferability, enabling the discovery of both the common label set shared across domains and the label sets unique to each domain. By promoting adaptation within the automatically identified common label set, UAN effectively discerns and appropriately labels "unknown" samples.

5. **DADA**[208]: DADA (Discriminative Adversarial Domain Adaptation) aims to align the joint distributions of feature and category across domains for unsupervised domain adaptation. To overcome mode collapse induced by separate task and domain classifiers in tradiaitonal DA techniques like DANN, ADDA, DADA uses an integrated category and domain classifier with a novel adversarial objective that promotes a mutually inhibitory relation between category and domain predictions for input instances. This is achieved

through a minimax game that aligns joint distributions of domain labels and category labels.

6.3.3 Research Questions

To determine the impact of CLODIA in improving the generalizability and performance of code clone detection tools in unseen domains, we employ established code clone detection tools outlined in 6.3.2.1. These tools are combined with various domain adaptation techniques detailed in 6.3.2.2. This led to our first research question:

Research Question 1

How does CLODIA impacts the performance of supervised clone detection tools on unseen targets domains?

Our primary goal is to thoroughly assess the effectiveness of CLODIA in enhancing the performance of clone detection tools, particularly when applied to unseen datasets and domains. It is crucial to note that our evaluation intentionally avoids a direct comparison between traditional clone detection techniques and CLODIA applied to supervised learning-based methods. This decision is justified by two key considerations.

Firstly, traditional clone detection techniques are not vulnerable to the domain shift issue, which is the primary focus of our investigation. These techniques rely on syntactic, structural, and semantic patterns to identify code clones, without learning from the data. As a result, they are less affected by variations in data distribution across different domains. Comparing them with CLODIA, designed to address domain shift challenges, would not yield meaningful insights. Secondly, existing learning-based techniques have significantly outperformed traditional methods. These supervised techniques, relying on labeled data for training, have elevated clone detection accuracy and efficiency. Therefore, it is pertinent to evaluate CLODIA's effectiveness in further improving the performance of supervised learning-based methodologies on unseen data. CLODIA's core objective is to enhance the generalization of such techniques, making it essential to assess its impact in this specific context.

To answer this research question, we compare CLODIA with a diverse range of well-established code clone detection techniques, representing various approaches and algorithms. These techniques are evaluated with CLODIA across different datasets, encompassing a variety of programming languages, application domains, and code structures. Additionally, we conduct a baseline evaluation of the selected clone detection tools without integrating CLODIA to establish their performance on the chosen dataset without external influence. Popular metrics such as precision, recall, and F1-score are employed to assess the performance of these code clone detection tools when integrated with CLODIA.

Next, we conduct a comprehensive comparative analysis of CLODIA with various existing DA techniques. The primary objective of this experiment is to ascertain whether other well-established DA techniques could be readily applied to enhance the capabilities of clone detection tools. This led to our second research question:

Research Question 2

How does CLODIA compares with other domain adaptation techniques for improving the performance of supervised code clone detection techniques on unseen targets domains?

This choice of selecting DA techniques for comparison was underpinned by the following key considerations:

1. **Diverse range of techniques:** We systematically compare CLODIA with a diverse array of DA techniques, encompassing both traditional statistics-based approaches (CORAL [205]) and state-of-the-art adversarial methods. This was to ensure a holistic evaluation, allowing us to explore a wide spectrum of potential solutions for improving clone detection.

2. **Incorporating domain-invariant features:** Traditional statistics-based techniques, such as CORAL [205], while straightforward, may overlook higher-order information within the data and struggle when faced with complex distribution shifts. In contrast, adversarial methods like DANN[78] and ADDA[206] introduce a domain discriminator and employ a gradient reversal layer to promote the acquisition of domain-invariant feature representations. This choice was motivated by the desire to evaluate the efficacy of adversarial techniques in mitigating domain shift challenges in clone detection.

3. **Weight-sharing implications:** Building upon the principles of DANN, ADDA introduces a DA network designed to address larger domain shift using GAN-based loss. DANN imposes tied weights on model, implying that the feature extractor weights were shared between both source and target domain. While weight sharing can facilitate knowledge transfer from the source to the target domain, its benefit is most pronounced when underlying patterns are consistent across domains. However, in cases of larger domain shifts, this approach may restrict the model's flexibility to adapt to domain-specific characteristics, potentially leading to suboptimal performance on the target domain.

4. **Exploring contemporary approaches:** Previous adversarial techniques were designed for simple domain adaptation tasks, when there is not much difference in the source and target domain. These approaches assumes a shared label space between the source and target domains and exploits prior knowledge about the label set relationships. However, these assumptions significantly limit their applicability in the wild. Moreover, the previous techniques were prone to mode collapse. In clone detection, due to differences in coding style, functionality etc. the source and target domains does not share much in common and due to high data discrepancy, the previous DA techniques are also susceptible to mode collapse. To mitigate these issues, we sought to experiment with new DA approaches like UAN [207] and DADA[208], specifically designed for domain adaptation in diverse, real-world settings. These contemporary approaches propose explicit solutions to combat mode collapse and effectively adapt to significant domain discrepancies, aligning well with the challenges posed by clone detection scenarios.

6.3.4 Implementation Details

We implement all the domain adaptation techniques in Python using the PyTorch deep learning framework. We train these models on the source dataset using 60% of the data for training, 20% for validation and remaining 20% for testing. We also divide the target dataset in train and test set in 60:40 ratio, where we use 60% of the for domain adaptation and alignment and the rest 40% is used for evaluating the performance on unseen target dataset. For the experimental setting where we use GCJ as both seen source domain and unseen target domain dataset, we divide the

Table 6.2: Performance evaluation of code clone detection baselines in supervised setting.

Methods	GCJ			BCB		
	P	R	F1	P	R	F1
FA-AST + GGNN	75	80	77	85	90	87
FA-AST + GMN	77	83	80	**96**	**94**	**95**
TBCCD	79	85	82	**96**	**96**	**96**
HOLMES	**91**	**93**	**92**	**97**	**98**	**97**

data on the basis of functionality. We have a total 100 functionalities in the GCJ, from which we randomly reserve 50 functionalities for seen source domain dataset (GCJ_{sn}) and rest 50 for unseen target domain dataset (GCJ_{un}). We ensure that both domains have an equal number of problems and the problems in the source domain cover a diverse range of functionalities to ensure that the domain adaptation techniques are effectively tested. We did not conduct the same experiments using BCB as it covers only ten functionalities, which were not sufficient to divide the dataset into seen source and unseen target domains. We use available implementations of HOLMES, TBCCD, and FA-AST with the reported settings for the hyper parameters. To refer to a particular clone detection technique applied in conjunction with a specific DA technique, we follow a common notion, clone detection tool (DA technique). For instance, "TBCCD (DANN)" represents that TBCCD is applied in conjunction with DANN. We evaluate the performance of the model in terms of precision, recall and F1-score. We report the findings in terms of the relative performance of the different combinations of source and target domains, highlighting the factors that affect the domain adaptation effectiveness. We also compare our results with no domain adaptation setting i.e., applying the clone detection tools directly on unseen datasets, to demonstrate the effectiveness of domain adaptation techniques in improving the performance on unseen target domains. We determine statistical significance of our results using a two-tailed paired t-tests and Wilcoxon signed-rank tests with a significance level of 0.05. All experiments are conducted on a machine with a NVIDIA GeForce RTX 2080 Ti and NVIDIA V100 GPU. Overall, our methodology provides a systematic and rigorous way to evaluate the impact of dataset characteristics on domain adaptation effectiveness for code clone detection.

Table 6.3: Performance evaluation of code clone detection baselines on GCJ, BCB and AC datasets without domain adaptation.

Methods	$GCJ_{sn} \rightarrow GCJ_{un}$			GCJ → BCB			BCB → GCJ			GCJ → AC			BCB → AC		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
HOLMES	38.5	41.9	40.1	40.3	43.5	41.8	34.5	36	35.2	22.3	28.8	25.1	20.6	29.1	24.1
TBCCD	29.4	33.7	31.4	32.5	37.1	34.6	26.2	29.4	27.7	19.8	21.5	20.6	16.8	22.4	19.2
FA-AST + GGNN	22.5	26.4	24.2	23.6	26.7	25.0	21.5	24.6	22.9	11.4	15.6	13.1	10.8	12.5	11.5
FA-AST + GMN	27.8	31.4	29.4	29.7	35.7	32.4	20.2	25.1	22.3	13.1	16.3	14.5	12.3	16.5	14.0

Table 6.4: Robustness analysis of domain adaptation techniques on GCJ, BCB and AC datasets for code clone detection tools.

(a) Performance of FA-AST + GGNN clone detection tool.

Methods	$GCJ_{sn} \rightarrow GCJ_{un}$			GCJ → BCB			BCB → GCJ			GCJ → AC			BCB → AC		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
FA-AST + GGNN (CORAL)	30.2	31.8	30.9	33.4	35.8	34.5	26.1	28.7	27.3	29.5	31.6	30.5	24.5	27.9	26.0
FA-AST + GGNN (DANN)	37.6	40.7	39.0	39.2	41.2	40.1	31.2	34.8	32.9	35.6	38.9	37.1	30.8	33.5	32.0
FA-AST + GGNN (ADDA)	42.6	45.3	43.9	45.6	46.7	46.1	39.7	41.3	40.4	40.3	41.5	40.8	38.1	41.4	39.6
FA-AST + GGNN (DADA)	51.6	54.7	53.1	53.7	57.8	55.6	44.7	48.1	46.3	47.8	50.3	49.0	43.8	45.9	44.8
FA-AST + GGNN (UAN)	56.4	59.7	58.0	58.9	63.2	60.9	51.2	54.1	52.6	53.8	55.1	54.4	50.9	53.1	51.9
FA-AST + GGNN (CLODIA)	**64.8**	**69.8**	**67.2**	**67.9**	**70.3**	**69.0**	**61.2**	**63.5**	**62.3**	**63.6**	**66.4**	**64.9**	**60.2**	**61.5**	**60.8**

6.4 Results

6.4.1 RQ 6.1: How does CLODIA impacts the performance of supervised clone detection tools on unseen targets domains?

To evaluate the effectiveness of CLODIA in improving the performance of supervised clone detection tools when applied to unseen target domains, we conduct a comprehensive set of experiments. These experiments are designed to measure the performance of various baseline clone detection tools, namely HOLMES, TBCCD, FA-AST + GGNN, and FA-AST + GMN.

Initially, we establish a performance baseline for each clone detection tool by training and testing them within the same dataset. We use two primary datasets for this evaluation: GCJ and BCB. The AC dataset is not included due to its relatively small size, which might not be sufficient for effective training. The results, presented in Table 6.2, show that HOLMES achieves the highest F1-score of 92% and 97% on GCJ and BCB datasets respectively.

Next, we examine the generalization ability of these baseline models by training them on

(b) Performance of FA-AST + GMN clone detection tool.

Methods	$GCJ_{sn} \rightarrow GCJ_{un}$			GCJ → BCB			BCB → GCJ			GCJ → AC			BCB → AC		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
FA-AST + GMN (CORAL)	31.5	33.1	32.2	33.6	36.5	34.9	27.5	30.5	28.9	30.1	32.9	31.4	26.3	28.5	27.3
FA-AST + GMN (DANN)	40.3	42.5	41.3	41.6	44.7	43.1	33.1	36.3	34.6	36.1	39.2	37.5	31.5	35.6	33.4
FA-AST + GMN (ADDA)	44.2	47.6	45.8	45.9	47.1	46.4	40.6	43.8	42.1	41.5	42.1	41.8	40.2	43.1	41.6
FA-AST + GMN (DADA)	52.9	55.2	54.0	55.6	58.9	57.2	46.4	49.8	48.0	55.1	52.6	53.8	45.8	48.9	47.3
FA-AST + GMN (UAN)	58.8	61.7	60.2	60.7	64.8	62.6	54.6	58.2	56.3	54	55.8	54.8	52.7	56.3	54.4
FA-AST + GMN (CLODIA)	**66.7**	**70.1**	**68.3**	**69.1**	**73.5**	**71.2**	**63.2**	**65.8**	**64.4**	**64.2**	**66.1**	**65.1**	**62.1**	**65.3**	**63.6**

(c) Performance of TBCCD clone detection tool.

Methods	$GCJ_{sn} \rightarrow GCJ_{un}$			GCJ → BCB			BCB → GCJ			GCJ → AC			BCB → AC		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
TBCCD (CORAL)	34.8	38.6	36.6	37.3	40.1	38.6	29.1	31.5	30.2	31.6	33.4	32.4	28.5	30.4	29.4
TBCCD (DANN)	45.6	48.6	47.0	48.6	50.1	49.3	41.8	42.8	42.2	41.4	43.8	42.5	39.8	41.6	40.6
TBCCD (ADDA)	53.4	54.6	53.9	55.8	58.8	57.2	50.1	52.5	51.2	50.6	52.1	51.3	48.9	50.1	49.4
TBCCD (DADA)	59.6	62.5	61.0	64.6	65.8	65.1	58.2	61.2	59.6	58.9	61.1	59.9	55.6	58.7	57.1
TBCCD (UAN)	62.7	65.8	64.2	66.3	68.8	67.5	59.2	61.3	60.2	61.4	63.8	62.5	59.8	60.6	60.2
TBCCD (CLODIA)	**70.2**	**73.1**	**71.6**	**73.4**	**76.4**	**74.8**	**68.3**	**70.1**	**69.1**	**67.7**	**69.9**	**68.7**	**64.9**	**68.1**	**66.4**

one dataset (either GCJ or BCB) and then testing them on a different dataset (which could be any of GCJ, BCB, or AC). This step is crucial to understand how the models cope with the domain shift—when they are exposed to code from projects with different characteristics and functionalities than those seen during training.

Our experimental results, presented in Table 6.3, indicate a marked decline in the performance of HOLMES, TBCCD, and FA-AST when tested on target datasets that were not seen during training. For example, when HOLMES is trained exclusively on a specific segment of the GCJ dataset, referred to as GCJ_{sn}, and then assessed on an alternate segment, GCJ_{un}, we observe a drastic reduction in the F1 score, which nosedives from an impressive 92% to a mere 40.1%. It's important to highlight that both subsets are part of the same comprehensive GCJ dataset. The primary difference lies in the exclusion of certain problem sets and their solutions from the training set. This stark deterioration in model performance brings to light the difficulties inherent in supervised clone detection models when they encounter novel and varied code scenarios, thereby emphasizing the significant influence of domain shift. Furthermore, when these baseline models are evaluated across domains with more pronounced differences, such as transitioning from GCJ to AC, the F1 scores dip even further, accentuating the substantial challenges posed by

(d) Performance of HOLMES clone detection tool.

Methods	$GCJ_{sn} \rightarrow GCJ_{un}$			GCJ → BCB			BCB → GCJ			GCJ → AC			BCB → AC		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
HOLMES (CORAL)	40.6	42.9	41.7	41.4	44.9	43.0	32.3	35.6	33.8	30.2	34.5	32.2	30.8	33.5	32.0
HOLMES (DANN)	56.8	59.7	58.2	58.9	61.2	60.0	48.7	52.1	50.3	52.6	56.5	54.4	43.2	44.5	43.8
HOLMES (ADDA)	60.8	63.0	61.8	62.0	64.8	63.3	53.8	58.5	56.0	57.8	60.7	59.2	51.1	53.6	52.3
HOLMES (DADA)	72.4	75.1	73.7	74.1	76.7	75.3	62.8	65.1	63.9	73.5	75.6	74.5	61.8	64.8	63.2
HOLMES (UAN)	70.6	72.9	71.7	72.6	75.6	74.0	61.6	66.1	63.7	70.1	74.5	72.2	60.8	63.9	62.3
HOLMES (CLODIA)	**85.6**	**87.1**	**86.3**	**87.5**	**88.4**	**87.9**	**70.1**	**73.7**	**71.8**	**83.9**	**85.9**	**84.8**	**70.1**	**72.9**	**71.4**

domain shifts in the field of clone detection.

Table 6.4 presents the enhanced performance of code clone detection baselines when integrated with CLODIA and tested on previously unseen datasets. An examination of the table reveals a uniform enhancement in the baselines' performance, with F1-scores experiencing a notable increase across different domain pairs in comparison to the scores obtained without domain adaptation, as shown in Table 6.3.

Figure 6.4 illustrates a successfully identified true positive clone pair from the AC dataset by HOLMES, TBCCD, and FA-AST clone detection tools trained on the BCB dataset after applying CLODIA. The code snippets shown in Figure 6.4 performs string translation through the substitution of specific characters. Each method processes a source string, implements a character mapping algorithm, and outputs the translated string. Despite the absence of such functionality in the BCB dataset, the application of CLODIA enables the clone detection models to accurately classify this pair as clones. This can be ascribed to the domain adaptation facilitated by CLODIA, which helps these models to learn feature representations that are indicative of cloning, regardless of features that are specific to the source domain but not helpful for clone detection in the target domain.

The efficacy of domain adaptation is markedly influenced by the chosen source and target domains. Our analysis indicates that dataset characteristics, including size, complexity, programming language, and coding style, significantly affect model performance. The GCJ dataset is characterized by greater complexity and diversity, encompassing a broader range of variations compared to the BCB dataset [108]. The functionality of BCB clones is limited and the code

```java
public final void translate(final
    CharSequence in, final Writer writer)
    throws IOException {
  Objects.requireNonNull(writer, "writer");
  if (in == null) {return; }
  int pos = 0;
  final int len = in.length();
  while (pos < len) {
    final int consumed = translate(in, pos,
      writer);
    if (consumed == 0) {
      final char c1 = in.charAt(pos);
      writer.write(c1);
      pos++;
      if (Character.isHighSurrogate(c1) &&
        pos < len) {
        final char c2 = in.charAt(pos);
        if (Character.isLowSurrogate(c2)) {
          writer.write(c2);
          pos++;
          }
        }
      continue;
      }
    for (int pt = 0; pt < consumed; pt++)
      pos += Character.charCount(
        Character.codePointAt (in, pos));
    }
}
```

```java
public static String translate(String
    src, String from, String to) {
  if (isEmpty(src))
    return src;
  StringBuilder sb = null;
  int ix;
  char c;
  for (int i = 0, len = src.length();
    i < len; i++) {
    c = src.charAt(i);
    ix = from.indexOf(c);
    if (ix == -1){
      if (sb != null){
        sb.append(c);
        }
    }
    else{
      if (sb == null) {
        sb = new StringBuilder(len);
        sb.append(src, 0, i);
      }
      if (ix < to.length()) {
        sb.append(to.charAt(ix));
        }
      }
    }
  return sb == null ? src :
    sb.toString();
}
```

Figure 6.4: A true positive clone pair from AC dataset detected as true positive by TBCCD, FA-AST, and HOLMES after applying CLODIA.

snippets share significant syntactic similarity, even though they were reported under WT3/T4 category. Consequently, training a model on the comprehensive GCJ dataset and adapting it with CLODIA for the BCB dataset results in a average F1 score improvement of 43% . Conversely, on average training on the BCB dataset and adapting for the GCJ dataset yields an enhancement of around 35%. These findings suggest that models trained on larger and more diverse datasets can be more effectively adapted to simpler datasets resulting in improved performance.

RQ 6.1: CLODIA *can improve the performance of code clone detection models on unseen datasets. However, the choice of source and target domains, as well as the characteristics of the datasets, can also significantly impact the effectiveness of* CLODIA *in adapting to unseen environments.*

6.4.2 RQ 6.2: How does CLODIA compares with other domain adaptation techniques for improving the performance of supervised code clone detection techniques on unseen targets domains?

To address RQ2, we assess the effectiveness of various domain adaptation techniques on different clone detection models. These techniques include the traditional method CORAL and adversarial methods DANN, ADDA, DADA and UAN. Our findings as shown in Table 6.4 reveal that all domain adaptation techniques improve the performance of the baseline tool when tested on an unfamiliar code clone dataset.

CORAL shows moderate enhancement compared to the "No Adaptation" scenario (Table 6.3) across all instances. For instance, with TBCCD, it elevates the F1 score from 31.4 to 36.6 when transitioning from $\text{GCJ}_{sn} \rightarrow \text{GCJ}_{un}$, reflecting similar improvements across other models as well. This indicates that while statistical domain adaptation techniques like CORAL can offer some level of performance boost in facing domain shifts, their effectiveness is limited. They do not deliver substantial performance gains on unseen datasets.

Adversarial techniques DANN and ADDA further improves upon CORAL and over the "No Adaptation" scenario. For example, with TBCCD, ADDA boosts the F1 score to 53.9 in the $\text{GCJ}_{sn} \rightarrow \text{GCJ}_{un}$ scenario. Thus, these adversarial techniques exhibits strong performance, suggesting its effectiveness in more accurately adapting models to target domains.

Notably, DADA and UAN consistently outperform other techniques across most domain pairs and in many cases, begins to approach the performance levels of CLODIA. For example, DADA achieves an F1 score of 61.0 while UAN achieves an F1 score of 64.2 with TBCCD in the $\text{GCJ}_{sn} \rightarrow \text{GCJ}_{un}$ shift, indicating a robust adaptation capability. This results suggests that these new domain adaptation approaches, designed for domain adaptation in the wild, provide explicit solutions to domain shift challenges in code clone detection.

We also observe that the choice of the underlying clone detection tool significantly impacts the model's performance in the target domain. To illustrate this point, consider Figure 6.5, which showcases a code clone pair from the AC dataset. Interestingly, this pair was identified as a false

```java
...
String uri =
↪ url.toIdentityString();
ConcurrentMap<String, RpcStatus>
↪ map =
↪ METHOD_STATISTICS.get(uri);
if (map != null)
  map.remove(methodName);
...
```

```java
...
String uri = url.toIdentityString();
ConcurrentMap<String, RpcStatus> map =
↪ ConcurrentHashMapUtils.computeIfAbsent(
↪ METHOD_STATS, uri, k -> new
↪ ConcurrentHashMap<>());
...
return
↪ ConcurrentHashMapUtils.computeIfAbsent
↪ (map, methodName, k -> new
↪ RpcStatus());
```

Figure 6.5: A true negative clone pair from AC dataset detected as false positive by FA-AST and TBCCD which otherwise was detected as true negative by HOLMES.

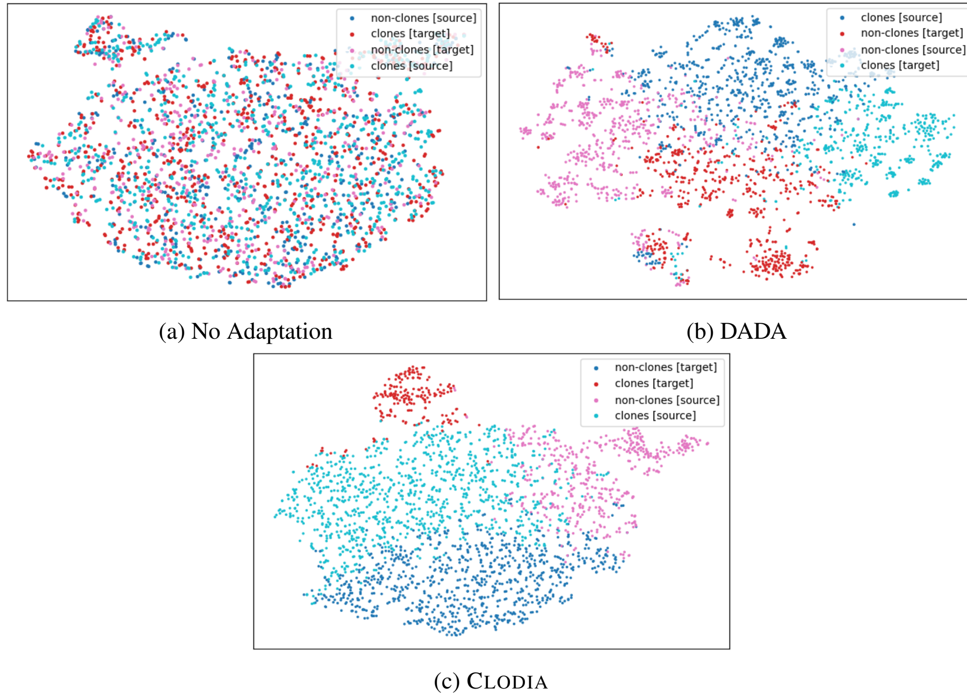
positive by TBCCD and FA-AST but correctly labeled as a true positive by HOLMES. The key to this distinction lies in how these tools approach clone detection. Both TBCCD and FA-AST rely on code tokens for feature learning. In cases where there is a high degree of token overlap between two code snippets, these tools may erroneously flag them as clones. However, HOLMES takes a different approach by modeling the relationships between different node types using a PDG. As a result, the PDG representations of the two snippets differ significantly, allowing HOLMES to accurately classify them as non-clones. This discussion also underscores the role of DA in enhancing the performance of models in previously unseen environments. While clone detection tools are essential in this context, they are not the sole determinants of a model's success in a different domain. DA can indeed help these models perform well in unseen environments, but it cannot improve their performance beyond the upper limit set by these models in supervised setting (as reported in Table 6.2).

Furthermore, across a range of scenarios, we observe that HOLMES outperforms both TBCCD and FA-AST. However, a deeper analysis of the performance of HOLMES (UAN) in comparison to HOLMES (DADA) reveals a slight decrease in performance, when compared with TBCCD and FA-AST. UAN employs an additional discriminator designed to measure the distance of input feature (regardless of the domain) to the source domain. This discriminator was introduced to enhance the alignment of the target domain with the source domain. Moreover, HOLMES, in its FLM module, attempts to learn relationships between different node types through PDG. This feature may be domain-specific, resulting in a lack of proper alignment with the additional discriminator introduced by UAN. Consequently, the performance of HOLMES is slightly affected. However, TBCCD and FA-AST rely on program tokens that are not domain-specific and can

be common across domains, especially when dealing with the same programming language. This commonality allows them to potentially train the additional discriminator of UAN more effectively.

However, CLODIA consistently achieves the highest F1-scores across all domain pairs. For instance, in the TBCCD model for the $\text{GCJ}_{sn} \rightarrow \text{GCJ}_{un}$, CLODIA achieves an F1 score of 71.6, surpassing UAN's 64.2, which is the next best performance. CLODIA demonstrates not just incremental but significant improvements over the other techniques. This is particularly evident in more challenging domain shifts, such as from BCB to AC, where CLODIA significantly raises the bar for performance. The reason CLODIA outperforms other domain adaptation techniques can be attributed to its ability to learn a representation space that balances discriminative power while maintaining a clear separation of the different domains representations. When dealing with clones, different datasets can vary significantly in terms of functionality, coding conventions, size and complexity (as shown in Figure 6.4), which must be considered when reducing the distance between the domains through adversarial training. The proposed *feature learning module* in CLODIA is designed to map examples from different domains to distinct representations while the *feature transfer module* separates the target features from the source features, aligning them with an auxiliary domain of transformed source features. This alignment ensures that the discriminative power of the augmented source representation space is transferred to the target representation space.

It is also interesting to see the performance of clone detection baselines on AC dataset. This is the most challenging domain adaptation dataset due to the substantial differences in coding style and functionality of the AC dataset from both GCJ and BCB datasets. However, from BCB $\rightarrow$ AC, we can still see most techniques show improved performance as compared to GCJ $\rightarrow$ AC. This might be because both source and target domains are more related in terms of coding style, small code snippets, etc. Thus, the effectiveness of the adaptation technique is highly dependent on the characteristics of the datasets used, highlighting the importance of carefully choosing the source and target domains for domain adaptation in clone detection models.



(a) No Adaptation (b) DADA

(c) CLODIA

Figure 6.6: t-SNE visualisations of samples from GCJ$_{sn}$ and GCJ$_{un}$ by different DA methods.

> **RQ 6.2:** CLODIA *outperforms other techniques by effectively learning distinct representations for different domains and aligning them through its feature learning and transfer modules. Additionally, the choice of clone detection tool and its underlying approach to feature learning plays a crucial role in determining the effectiveness of domain adaptation in overcoming domain shift challenges.*

6.4.3 Qualitative analysis of the learned feature space

We utilize t-SNE plots to visualize the HOLMES's learned feature space of the GCJ$_{sn}$ and GCJ$_{un}$ datasets. In the absence of any adaptation, the t-SNE plot shows (Figure 6.6a) that the features of clones and non-clones from the two different domains are heavily mixed, indicating that the datasets are quite distinct due to differences in coding style and functionalities. This suggests that simply reducing the distance between the two domains would not suffice in improving performance on the unseen target domains. We then visualize the generated feature space applying DADA DA technique on GCJ$_{sn}$ and GCJ$_{un}$ datasets. Although there is better separation between the source and target domains in the t-SNE plot (Figure 6.6b), there is still significant

overlap between clones and non-clones classes of the two datasets. In contrast, the feature space of CLODIA, shown in Figure 6.6c, demonstrates better alignment of clones and non-clones classes across domains. This indicates that CLODIA not only better separates the two classes within domain but also differentiates them across domains.

6.4.4 Statistical Tests

To verify the significance of our results, we perform paired t-tests and Wilcoxon signed-rank tests on the performance metrics of our domain adaptation techniques. The paired t-test is used to determine if there is a statistically significant difference between two paired samples, while the Wilcoxon signed-rank test is a non-parametric test used when the assumptions of the t-test are not met.

For the paired t-test, we compare the performance of each domain adaptation technique with the baseline clone detection model trained only on the source domain. Specifically, we compared the F1 scores of each technique on the target domain before and after applying domain adaptation. The results of the paired t-tests showed a statistically significant improvement in performance for all domain adaptation techniques ($p < 0.05$) when compared to the baseline model. Similarly, for the Wilcoxon signed-rank test, we compare the performance of each domain adaptation technique with the baseline model trained only on the source domain. We tested for a significant difference in the rank-sums of the F1 scores before and after applying domain adaptation. The results of the Wilcoxon signed-rank test also show a statistically significant improvement in performance for all domain adaptation techniques ($p < 0.05$) when compared to the baseline model.

6.5 Chapter Summary

Code clones are prevalent in software systems and pose significant problems in software engineering, resulting in increased maintenance costs and decreased system reliability. Therefore, their automatic detection becomes a key to their elimination. While supervised techniques

for clone detection work well on labeled data, they fail to generalize to new datasets, limiting their practicality. Unsupervised Domain Adaptation (UDA) emerges as a promising solution, allowing models to fine-tune autonomously on unlabeled target domains after initial training on labeled source domains, thereby enhancing their performance. This chapter explores the effectiveness of adversarial domain learning for semantic code clone detection. We introduce a novel multispace domain adaptation framework, CLODIA. This framework learns multiple mappings from each domain to a shared latent space, effectively capturing common features across domains. This approach aligns feature distributions between source and target domains while preserving the intrinsic data structure. We also conduct extensive experiments with various UDA techniques to assess their performance in improving clone detection tools. Thus, our study highlights the importance of domain adaptation techniques for semantic code clone detection and demonstrates UDA's potential in addressing the limitations of supervised techniques. CLODIA shows promising results in adapting clone detection models to unseen datasets with an average performance improvement of 44.4%. Our findings offer insights into the potential of adversarial domain learning for code clone detection and provide a basis for future research.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Clone detection is an important task in software engineering and maintenance that aims to identify similar or identical code fragments within or across multiple codebases. Clones can be either benign or problematic, depending on their usage context. Clone detection facilitates functionality reuse, consistency in implementation, bug fixing and improved code performance.

Traditional code clone detection techniques involve comparing pairs of code fragments and identifying similar or identical code. This can be done using a variety of methods such as *textual analysis, lexical analysis, and syntax-based analysis*. However, these techniques have several limitations and challenges. One major challenge is that they cannot detect code clones that are semantically similar but have different syntax or structure. Another challenge is that traditional techniques often produce a large number of false positives, where code fragments that are not true clones are identified as clones. This can be due to the presence of common patterns or idioms in code, which may not necessarily indicate a true code clone.

To overcome the limitations of traditional code clone detection techniques researchers have explored the use of machine intelligence in code clone detection. These techniques use a variety of features to represent code, such as token sequences, Control Flow Graphs (CFGs), and abstract syntax trees (ASTs). The learning algorithms then use these features to learn patterns in the code

and identify clones based on these patterns. These techniques have shown promising results in detecting clones that traditional techniques may miss.

Although ML-based code clone detection approaches have shown promising results in detecting clones, they still have certain limitations. The existing techniques primarily rely on syntactic and/or lexical information like abstract syntax trees to learn program features. Moreover, these techniques also do not use well-defined graphical structures when learning program representation. These approaches use deep learning models that do not take advantage of available structured input, such as capturing induced long-range variable dependency between program statements. This limitation may impact the accuracy of clone detection, as deep learning models may not effectively capture the nuances of program structures that traditional techniques can capture.

Thus, to address the limitations of learning-based code clone detection techniques, we propose a new tool HOLMES (chapter 4) as a first contribution of this thesis. HOLMES makes use of graph neural networks in conjunction with program dependence graphs (PDGs), and is based on two key insights. First, *Learned features should include semantic information, such as control and data dependence, rather than just structural information.* Syntactic features alone are too rigid and limiting compared to more expressive notions based on program semantics. Second, *to capture program semantics effectively, we took advantage of the graphical structure of PDGs, using a graph-based deep neural network to learn program representation.* To enhance HOLMES, we made use of attention-based graph neural networks and added tailor-made customizations to improve the representation learning framework. This includes adaptively exploring the depth and breadth of the PDG, adding skip connections to retain information learned at initial layers, and learning separate program representations for each edge type in PDG, as each edge type encodes different information.

As a **second** contribution of this thesis, we tackle the problem of detecting code clones that exist across different programming languages, which is known as cross-language code clone detection. This issue is crucial because software systems are frequently developed using multiple programming languages, and cross-language code clones can make it difficult to maintain and comprehend the system, resulting in issues such as code redundancy, inconsistency, and

maintainability. Traditional single-language code clone detection techniques rely on syntactic or lexical information specific to the programming language in which the code is written, making it challenging to detect code clones across different languages. Furthermore, since different programming languages have unique sets of features, constructs, and idioms, code clones may appear differently across various languages.

To address this challenge, we propose a cross-tool language code clone detection approach named RUBHUS. The main idea behind RUBHUS is to *incorporate semantic and structural information to accurately measure cross-language similarity*, in addition to shallow syntactic, lexical, and token-based similarity. This approach improves the learned program representation, making it easier to detect code clones. The second key insight behind RUBHUS is that *source code is made up of many interrelated code constructs*. Therefore, it is important to understand the interactions between different code constructs, both in terms of syntax and meaning, by presenting them as structured input to the neural network. RUBHUS use control and data flow enriched abstract syntax trees (ASTs) to detect similarity. Furthermore, mutual information maximization is employed to learn latent structures, substructures, and syntactic and semantic information present at different granularities of an AST. As a result,RUBHUS is capable of detecting clones across different programming languages, which single-language clone detection techniques fail to do effectively.

As a third contribution of this thesis, we propose a **multispace domain adaptation technique** called CLODIA to improve the generalisibility of supervised clone detection techniques on unseen datasets and address the challenge of manually annotating code snippets for training the model everytime on new dataset. Manual annotation can be a daunting task, and the availability of labeled datasets is often limited. The key idea behind CLODIA is that *different domains may have distinct underlying feature spaces, which can make it challenging to learn a common representation space for all the domains*. Instead, the approach seeks to learn multiple mappings from each domain to a common latent space that can capture the shared features across the domains. This technique aligns the feature distributions of seen source and unseen target domains while preserving the underlying structure of the data. To improve the accuracy of the neural code clone detection tools, several objectives were introduced, such as *domain separation* and *feature*

transfer modules, which align the feature spaces of source and target domains and prevent mode collapse during adversarial training. In addition, a *regularization objective* was proposed to prevent overfitting and improve generalization. The **domain separation module** is responsible for separating the shared and domain-specific features during the mapping process, while the **feature transfer module** ensures that the shared features are transferred across domains. The **regularization** objective promotes a smooth transition between domains and encourages the learned mappings to be consistent with the underlying data distribution. This approach enables the neural network to generalize well to unseen domains by learning to transfer knowledge from the source to the target domains. It can significantly reduce the annotation cost for supervised approaches and improve the accuracy of code clone detection in different domains.

Thus, the outcomes and findings of our research support the thesis statement that leveraging advanced learning-based techniques and program representations strategically can enhance the performance of semantic code clone detection tools.

7.2 Future Work

The work presented in this thesis has opened up new possibilities for future research. As software systems grow increasingly complex, the tools and techniques to manage this complexity must evolve accordingly. In the sections that follow, we will explore ways to build on the methods proposed in this thesis. We will look into promising research areas such as enhancing automated tools for managing code clones, detecting code clones with more precision, and using advanced language models to improve code clone detection capabilities.

7.2.1 Development of Automated Tools for Code Clone Management and Refactoring

Code clones, while facilitating code reuse, also pose challenges in terms of maintenance, as changes made to one instance of a clone need to be consistently applied to all its duplicates. Automated refactoring tools have the potential to significantly enhance code clone management by intelligently identifying and refactoring code clones to improve maintainability and reduce

redundancy.

The future research can delve into the development of novel techniques or enhancements for existing automated refactoring tools to recommend refactoring options or carry out automated refactoring to eradicate clones. This may include the integration of advanced machine learning algorithms to better understand the context and semantics of code, enabling more accurate and context-aware clone management. Additionally, exploring ways to incorporate user feedback and preferences into automated refactoring processes can contribute to tools that align more closely with developers' needs and preferences.

7.2.2 Fine-Grained Clone Detection

Current clone detection techniques predominantly operate at the method level, but there's a burgeoning opportunity to delve deeper, identifying clones at a more granular, such as statement or block level. This fine-grained clone detection approach promises a detailed and nuanced understanding of code duplication, enabling the precise identification and targeted refactoring of cloned segments. However, focusing intensely on fine-grained, syntactic details introduces the challenge of potentially overlooking the semantic relationships between code fragments, particularly when identifying semantic clones.

Despite this challenge, the practice of pinpointing fine-grained, statement/block-level clones within a codebase is invaluable. It unlocks a myriad of benefits and critical insights for improving software quality. This includes enabling precise code refactoring, enhancing code maintainability, conducting thorough security analyses, and optimizing resource use. Each benefit is instrumental in not only refining the software's quality but also in fortifying its robustness, security, and operational efficiency.

Therefore, by exploring this approach, we can contribute to the development of more sophisticated automated refactoring tools that can effectively manage and optimize code clones at a finer level of granularity. Thus, as future work for this thesis, exploring fine-grained clone detection involves diving into advanced algorithms and methodologies capable of operating at the statement or block level. This exploration may require adapting existing clone detection

```
for (i, row) in lst.iter().enumerate(){
    for (j, &value) in
↪   row.iter().enumerate(){
        if value == x{
            out.push(vec![i as i32, j as
↪   i32]);
        }
    }
}
```

```
for (int i = 0; i < lst.size(); i++){
    for (int j = lst[i].size() - 1; j >=
↪   0; j--){
        if (lst[i][j] == x){
            out.push_back({i, j});
        }
    }
}
```

Figure 7.1: A true negative clone pair detected as true positive GPT-3.5.

techniques to support finer granularity and developing new metrics to accurately quantify the similarity between smaller code fragments. Through this focused research, we aim to contribute significantly to the field, creating tools and methodologies that address the unique demands of managing code duplication at a finer level of detail.

7.2.3 Exploring the Potential of Large Language Models for Detecting Semantic Clones

Large Language Models (LLMs) demonstrate remarkable capabilities in understanding the syntax and structure of popular programming languages. They also exhibit a good grasp of the semantics of programs, as evidenced by research leveraging LLMs for generating summaries and comments for code snippets. However, LLMs sometimes falter when it comes to a deep understanding of programming constructs and complex reasoning.

Consider the paired C++ and Rust code snippets depicted in Figure 7.1, which both employ nested loops to traverse a 2-dimensional vector. The C++ snippet utilizes an outer loop for row iteration, with the inner loop processing each row's elements in reverse order. The conditional statement checks whether the element at 'lst[i][j]' matches a predefined integer 'x'. If the condition is met, the indices '(i, j)' are stored in the output vector 'out'. In contrast, the Rust snippet's inner loop iterates in the conventional left-to-right sequence, differing from the C++ snippet's reverse iteration. While these differences are accurately recognized, LLM erroneously categorize the snippets as clones, under the presumption that the sequence in which positions are added to the 'out' vector is not important. This assumption leads LLMs to classify the snippets as clones, on the basis that both are designed to locate a particular value 'x' in a 2-dimensional structure 'lst' and to log the value's coordinates. In the following box we show LLMs output:

Key Points

1. *Languages: The differences in the way the loops and conditions are written are due to the inherent differences between C++ and Rust syntax and features.*

2. *Iteration Direction: One notable difference is the iteration direction of the inner loop:*

   (a) *C++ iterates backwards (from the end of the row to the beginning).*

   (b) *Rust iterates forwards (from the beginning of the row to the end).*

Despite syntactic and iteration direction differences, the two snippets are indeed clones of each other in terms of their functional purpose: finding and storing positions of elements equal to x in a 2D list.

An average developer when faced with such a situation would likely invest time in understanding the functionality of the code snippets, possibly iterating on the implementation to better comprehend the functionality before deciding whether the two snippets are clones or non-clones. This process might involve backtracking and making edits, rather than a rigid, line-by-line comparison. In contrast, LLMs lack the ability to backtrack and make iterative edits; they typically operate on a next-word prediction model.

To explore the extent to which LLMs, when provided with ample information, can autonomously detect clones and non-clones, we prompted the LLM in a two-step process. In the first step, we asked the LLM to describe the functionality of the input code snippets. In the next step, we used the LLM's output as input in our prompt, along with the two code implementations. Surprisingly, this time the LLM was able to correctly identify that the two code snippets use different approaches to iterate over the 2-d array and are not similar in functionality.

Thus, it is noteworthy that LLMs have inherent limitations in reaching such optimal solutions directly. The autoregressive nature of these models compels them to address problems sequentially, which hampers their ability to generate a holistic solution or a "Eureka" idea.

This limitation becomes evident in tasks that involve a two-step process, like understanding the functionality of the code snippets and then deciding whether the two snippets are clones or non-clones. The LLM's incapacity to plan ahead and anticipate the complete solution hinders its effectiveness in handling discontinuous tasks or tasks that require complex reasoning.

In many instances, we also noticed that LLMs exhibit confusion about what constitutes clones and non-clones. For example, when prompted with implementations of bubble sort and merge sort, the LLM initially identified them as clones of each other. However, when prompted again with the same input, it clarified that the two implementations are very different both syntactically and algorithmically, and thus, not clones. This inconsistency can be attributed to the ambiguous and opaque definitions of Type 3 and Type 4 clones in the literature. Since the LLM is trained on such data, it naturally reflects this confusion in its responses.

Furthermore, the inherent randomness in the LLM's output occasionally led to hallucinated responses, even when explicit instructions were given to avoid this.

Despite these limitations, LLMs remain valuable for code clone detection. While they may face occasional challenges, LLMs possess a comprehensive understanding of various programming language ecosystems. Thus, with sufficient information and sophisticated prompting techniques that provide more semantic and syntactic context about the code snippets, along with leveraging more advanced models like GPT-4 that excel at following commands, the precision and efficacy of LLMs can be improved to identify clones in alignment with standard guidelines.

Moreover, delving deeper into how these models interpret and identify code similarities, especially in evolving software systems, holds the potential for substantial advancements in software maintenance practices. Additionally, investigating the impact of various training datasets and fine-tuning strategies on the performance of LLMs in clone detection presents a valuable trajectory for further exploration.

References

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, pp. 368–377.

[2] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *School of Computing TR 2007-541, Queen's University*, vol. 115, 2007.

[3] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: A map of code duplicates on github," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133908

[4] Y. Zhao, R. Mo, Y. Zhang, S. Zhang, and P. Xiong, "Exploring and understanding cross-service code clones in microservice projects," *IEEE International Conference on Program Comprehension*, vol. 2022-March, pp. 449–459, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3524610.3527925

[5] M. R. H. Misu and A. Satter, "An exploratory study of analyzing javascript online code clones," *IEEE International Conference on Program Comprehension*, vol. 2022-March, pp. 94–98, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3524610.3528390

[6] E. Wyss, L. D. Carli, and D. Davidson, "What the fork? finding hidden code clones in npm," *Proceedings - International Conference on Software Engineering*, vol. 2022-May, pp. 2415–2426, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510168

[7] Z. Li, T. H. Chen, J. Yang, and W. Shang, “Studying duplicate logging statements and their relationships with code clones,” *IEEE Transactions on Software Engineering*, vol. 48, pp. 2476–2494, 7 2022.

[8] H. Jebnoun, M. S. Rahman, F. Khomh, and B. A. Muse, “Clones in deep learning code: what, where, and why?” *Empirical Software Engineering*, vol. 27, pp. 1–75, 7 2022. [Online]. Available: https://link.springer.com/article/10.1007/s10664-021-10099-x

[9] A. Blasi and A. Gorla, “Replicomment: Identifying clones in code comments,” *Proceedings - International Conference on Software Engineering*, pp. 320–323, 5 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3196321.3196360

[10] M. Pyl, B. V. Bladel, and S. Demeyer, “An empirical study on accidental cross-project code clones,” *IWSC 2020 - Proceedings of the 2020 IEEE 14th International Workshop on Software Clones*, pp. 33–37, 2 2020.

[11] C. Kapser and M. W. Godfrey, “"cloning considered harmful" considered harmful,” in *2006 13th Working Conference on Reverse Engineering*, 2006, pp. 19–28.

[12] I. Keivanloo, J. Rilling, and Y. Zou, “Spotting working code examples,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, p. 664–675.

[13] M. F. Zibran and C. K. Roy, “Towards flexible code clone detection, management, and refactoring in ide,” in *Proceedings of the 5th International Workshop on Software Clones*, ser. IWSC ’11, p. 75–76.

[14] L. Jiang, Z. Su, and E. Chiu, “Context-based detection of clone-related bugs,” in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE ’07, p. 55–64.

[15] M. Mondal, C. K. Roy, and K. A. Schneider, “A fine-grained analysis on the inconsistent changes in code clones,” *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pp. 220–231, 9 2020.

[16] M. Hammad, Önder Babur, H. A. Basit, and M. van den Brand, "Deepclone: Modeling clones to generate code predictions," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12541 LNCS, pp. 135–151, 2020. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-64694-3_9

[17] S. Gholamian, "Leveraging code clones and natural language processing for log statement prediction," *Proceedings - 2021 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*, pp. 1043–1047, 2021.

[18] O. Ehsan, F. Khomh, Y. Zou, and D. Qiu, "Ranking code clones to support maintenance activities," *Empirical Software Engineering*, vol. 28, p. 70, 5 2023.

[19] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–28, 10 2019.

[20] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, pp. 1–34, 02 2010.

[21] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?" in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, 2017, pp. 1–7.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 176–192, 2006. [Online]. Available: https://doi.org/10.1109/TSE.2006.28

[23] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 157–166.

[24] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proceedings Eighth IEEE Symposium on Software Metrics*, 2002, pp. 87–94.

[25] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, H. Masuhara and T. Petricek, Eds. ACM, 2019, pp. 143–153. [Online]. Available: https://doi.org/10.1145/3359591.3359735

[26] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider, "Investigating context adaptation bugs in code clones," *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pp. 157–168, 9 2019.

[27] J. Islam, M. Mondal, C. Roy, and K. Schneider, "Comparing bug replication in regular and micro code clones," *IEEE International Conference on Program Comprehension*, vol. 2019-May, pp. 81–92, 5 2019.

[28] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 301–310.

[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105.

[30] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[31] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, 1992.

[32] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 219–220.

[33] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS ’01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 40–56.

[34] J. Krinke, “Identifying similar code with program dependence graphs,” in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 301–309.

[35] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, “Code relatives: Detecting similarly behaving software,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 702–714.

[36] A. Marcus and J. I. Maletic, “Identification of high-level concept clones in source code,” in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001, pp. 107–114.

[37] E. Buss, R. De Mori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. A. Muller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster, and K. Wong, “Investigating reverse engineering technologies for the cas program understanding project,” *IBM Systems Journal*, vol. 33, no. 3, pp. 477–500, 1994.

[38] M. Dagenais, E. Merlo, B. Laguë, and D. Proulx, “Clones occurence in large object oriented software packages,” in *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON ’98. IBM Press, 1998, p. 10.

[39] G. Di Lucca, M. Di Penta, and A. Fasolino, “An approach to identify duplicated web pages,” in *Proceedings 26th Annual International Computer Software and Applications*, 2002, pp. 481–486.

[40] Mayrand, Leblanc, and Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” in *1996 Proceedings of International Conference on Software Maintenance*, pp. 244–253.

[41] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “Aries: Refactoring support tool for code clone,” in *Proceedings of the Third Workshop on Software Quality*, ser. 3-WoSQ. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–4. [Online]. Available: https://doi.org/10.1145/1083292.1083306

[42] I. Keivanloo, C. K. Roy, and J. Rilling, “Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning,” in *2012 6th International Workshop on Software Clones (IWSC)*, 2012, pp. 36–42.

[43] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, “Neural detection of semantic code clones via tree-based convolution,” in *Proceedings of the 27th International Conference on Program Comprehension*, ser. ICPC ’19. IEEE Press, p. 70–80.

[44] G. Zhao and J. Huang, “Deepsim: Deep learning code functional similarity,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, p. 141–151.

[45] H.-H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI’17. AAAI Press, p. 3034–3040.

[46] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016.

[47] H.-H. Wei and M. Li, “Positive and unlabeled learning for detecting software functional clones with adversarial training,” in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 7 2018, pp. 2840–2846.

[48] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, “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)*, pp. 261–271, 2020.

[49] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[50] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 542–553.

[51] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 249–260.

[52] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, p. 321–330.

[53] N. Shrestha, T. Barik, and C. Parnin, "It's like python but: Towards supporting transfer of programming language knowledge," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2018, pp. 177–185.

[54] Q. Wu and J. R. Anderson, "Problem-solving transfer among programming languages," *Technical Report, Carnegie Mellon University*, 1990.

[55] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 367–377.

[56] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 115–126. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00021

[57] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, "Detecting clones across microsoft .net programming languages," in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 405–414.

[58] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu, and J. Zhao, "Mining revision histories to detect cross-language clones without intermediates," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 696–701.

[59] 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 *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 1026–1037.

[60] Y. Ganin and V. S. Lempitsky, "Unsupervised domain adaptation by backpropagation," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 1180–1189. [Online]. Available: http://proceedings.mlr.press/v37/ganin15.html

[61] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *Proceedings of the 27th International Conference on Program Comprehension*, ser. ICPC '19. IEEE Press, 2019, p. 70–80.

[62] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[63] D. Perez and S. Chiba, "Cross-language clone detection by learning over abstract syntax trees," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. IEEE Press, 2019, p. 518–528.

[64] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, jul 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[65] S. Horwitz, “Identifying the semantic and textual differences between two versions of a program,” in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI ’90, p. 234–245.

[66] A. Podgurski and L. Clarke, “The implications of program dependencies for software testing, debugging, and maintenance,” in *Proceedings of the ACM SIGSOFT ’89 Third Symposium on Software Testing, Analysis, and Verification*, ser. TAV3.

[67] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[68] Y. LeCun and Y. Bengio, *Convolutional Networks for Images, Speech, and Time Series*. Cambridge, MA, USA: MIT Press, 1998, p. 255–258.

[69] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, “Geometric deep learning: Going beyond euclidean data,” *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[70] G. E. Hinton, “Connectionist learning procedures,” *Artif. Intell.*, vol. 40, no. 1–3, p. 185–234, 1989.

[71] J. Hu, L. Shen, and G. Sun, “Squeeze-and-excitation networks,” pp. 7132–7141, 2018.

[72] G. Lample, M. Ott, A. Conneau, L. Denoyer, and M. Ranzato, “Phrase-based & neural unsupervised machine translation,” pp. 5039–5049, Oct.-Nov. 2018. [Online]. Available: https://aclanthology.org/D18-1549

[73] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, “Graph convolutional networks: a comprehensive review,” *Computational Social Networks*, vol. 6, 12 2019.

[74] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, “A comprehensive survey on graph neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

[75] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[76] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in neural information processing systems*, vol. 6, 1993.

[77] P. Baldi and Y. Chauvin, "Neural networks for fingerprint recognition," *Neural Comput.*, vol. 5, no. 3, p. 402–418.

[78] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, "Domain-adversarial training of neural networks," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 2096–2030, 2016.

[79] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell, "Adversarial discriminative domain adaptation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7167–7176.

[80] Y.-H. Tsai, W.-C. Hung, S. Schulter, K. Sohn, M.-H. Yang, and M. Chandraker, "Learning to adapt structured output space for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7472–7481.

[81] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," pp. 4487–4496, 2019. [Online]. Available: https://doi.org/10.18653/v1/p19-1441

[82] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[83] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019.

[84] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[85] X. Chen, C. Liang, A. W. Yu, D. Zhou, D. Song, and Q. V. Le, "Neural symbolic reader: Scalable integration of distributed and symbolic representations for reading comprehension," in *International Conference on Learning Representations*, 2020.

[86] R. Shin, I. Polosukhin, and D. Song, "Improving neural program synthesis with inferred execution traces," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 8931–8940.

[87] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embeddings for program repair," 2018. [Online]. Available: https://openreview.net/forum?id=BJuWrGW0Z

[88] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, p. 1–1, 2021.

[89] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 479–490.

[90] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, p. 602–614.

[91] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA.

[92] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 2091–2100. [Online]. Available: http://proceedings.mlr.press/v48/allamanis16.html

[93] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 87–98.

[94] C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, Eds., *Distributed Representations of Words and Phrases and their Compositionality*, 2013. [Online]. Available: https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html

[95] B. S. Baker, "Parameterized diff," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '99. USA: Society for Industrial and Applied Mathematics, 1999, p. 854–855.

[96] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, 1999, pp. 109–118.

[97] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 76–85. [Online]. Available: https://doi.org/10.1145/872757.872770

[98] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *J. Univers. Comput. Sci.*, vol. 8, no. 11, p. 1016, 2002. [Online]. Available: https://doi.org/10.3217/jucs-008-11-1016

[99] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1157–1168.

[100] I. Baxter, A. Yahin, L. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees." vol. 368-377, 01 1998, pp. 368–377.

[101] A. Raza, G. Vogel, and E. Plödereder, "Bauhaus - a tool suite for program analysis and reverse engineering," in *International Conference on Reliable Software Technologies*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:2499889

[102] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: Detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 872–881. [Online]. Available: https://doi.org/10.1145/1150402.1150522

[103] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 301–309.

[104] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: detection of clones in the twilight zone," in *ESEC/FSE 2018*, 2018.

[105] S. K. Abd-El-Hafiz, "A metrics-based data mining approach for software clone detection," in *2012 IEEE 36th Annual Computer Software and Applications Conference*, 2012, pp. 35–41.

[106] H.-H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, p. 3034–3040.

[107] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "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)*, pp. 261–271, 2020.

[108] G. Zhao and J. Huang, "Deepsim: Deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 141–151.

[109] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling functional similarity in source code with graph-based siamese networks," *IEEE Trans. Software Eng.*, vol. 48, no. 10, pp. 3771–3789, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3105556

[110] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 70–80.

[111] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: detection of clones in the twilight zone," in *ESEC/FSE 2018*, 2018.

[112] N. A. Kraft, B. W. Bonds, and R. K. Smith, "Cross-language clone detection," in *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008*. Knowledge Systems Institute Graduate School, 2008, pp. 54–59.

[113] T. Vislavski, G. Rakić, N. Cardozo, and Z. Budimac, "Licca: A tool for cross-language clone detection," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 512–516.

[114] Z. Budimac, G. Rakić, and M. Savić, "Ssqsa architecture," in *Proceedings of the Fifth Balkan Conference in Informatics*, ser. BCI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 287–290.

[115] X. CHENG, Z. PENG, L. Jiang, H. Zhong, H. Yu, and J. Zhao, "Clcminer: Detecting cross-language clones without intermediates," *IEICE Transactions on Information and Systems*, vol. E100.D, pp. 273–284, 02 2017.

[116] G. Mathew, C. Parnin, and K. T. Stolee, "SLACC: simion-based language agnostic code clones," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 210–221. [Online]. Available: https://doi.org/10.1145/3377811.3380407

[117] G. Mathew and K. T. Stolee, "Cross-language code search using static and dynamic analyses," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 205–217. [Online]. Available: https://doi.org/10.1145/3468264.3468538

[118] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: contrastive cross-language code clone detection," pp. 413–424, 2022. [Online]. Available: https://doi.org/10.1145/3524610.3527911

[119] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[120] Z. Lin, G. Li, J. Zhang, Y. Deng, X. Zeng, Y. Zhang, and Y. Wan, "Xcode: Towards cross-language code representation with large-scale pre-training," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–44, 2022.

[121] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, G. Scanniello, D. Lo, and A. Serebrenik, Eds. IEEE Computer Society, 2017, pp. 88–98.

[122] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu, "Binmatch: A semantics-based hybrid approach on binary code clone analysis," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 104–114.

[123] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," pp. 3835–3845, 2019. [Online]. Available: http://proceedings.mlr.press/v97/li19d.html

[124] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," *CoRR*, vol. abs/1708.06525, 2017.

[125] Y. David and E. Yahav, "Tracelet-based code search in executables," *SIGPLAN Not.*, vol. 49, no. 6, p. 349–360, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594343

[126] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'16. New York, NY, USA: Association for Computing Machinery, 2016, p. 480–491. [Online]. Available: https://doi.org/10.1145/2976749.2978370

[127] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/deepbindiff-learning-program-wide-code-representations-for-binary-diffing/

[128] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *CoRR*, vol. abs/2012.08680, 2020. [Online]. Available: https://arxiv.org/abs/2012.08680

[129] C. Liu, Z. Lin, J.-G. Lou, L. Wen, and D. Zhang, "Can neural clone detection generalize to unseen functionalities*f*," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 617–629.

[130] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 207–216.

[131] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 219–232.

[132] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.

[133] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from" big code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015.

[134] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[135] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020, pp. 201–211.

[136] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," *arXiv preprint arXiv:1802.00921*, 2018.

[137] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[138] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," in *Proceedings of ICLR'16*, April 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/gated-graph-sequence-neural-networks/

[139] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Advances in neural information processing systems*, vol. 31, 2018.

[140] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, dec 2020. [Online]. Available: https://doi.org/10.1145/3418463

[141] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17, p. 1345–1351.

[142] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embeddings for program repair," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=BJuWrGW0Z

[143] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Curran Associates Inc., p. 3589–3601.

[144] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," pp. 184–195, 2020. [Online]. Available: https://doi.org/10.1145/3387904.3389268

[145] M. Lu, D. Tan, N. Xiong, Z. Chen, and H. Li, "Program classification using gated graph attention neural network for online programming service," *CoRR*, vol. abs/1903.03804, 2019. [Online]. Available: http://arxiv.org/abs/1903.03804

[146] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," *CoRR*, vol. abs/2003.10536, 2020. [Online]. Available: https://arxiv.org/abs/2003.10536

[147] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling functional similarity in source code with graph-based siamese networks," *IEEE Trans. Software Eng.*, vol. 48, no. 10, pp. 3771–3789, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3105556

[148] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, no. 35, 2011.

[149] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[150] D. Binkley, "Source code analysis: A road map," in *IN FUTURE OF SOFTWARE ENGINEERING*, 2007, pp. 104–119.

[151] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[152] Z. Liu, C. Chen, L. Li, J. Zhou, X. Li, L. Song, and Y. Qi, "Geniepath: Graph neural networks with adaptive receptive paths," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, p. 4424–4431, Jul 2019.

[153] K. Xu, C. Li, Y. Tian, T. Sonobe, K. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 5449–5458. [Online]. Available: http://proceedings.mlr.press/v80/xu18c.html

[154] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," 10 2011.

[155] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490.

[156] D. Grunwald and H. Srinivasan, "Data flow equations for explicitly parallel programs," *SIGPLAN Not.*, p. 159–168.

[157] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, p. 137.

[158] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[159] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[160] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," pp. 1026–1034, 2015. [Online]. Available: https://doi.org/10.1109/ICCV.2015.123

[161] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[162] P. Gupta, N. Mehrotra, and R. Purandare, "Jcoffee: Using compiler feedback to make partial code snippets compilable," pp. 810–813, 2020. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00099

[163] M. Kamp, P. Kreutzer, and M. Philippsen, "Sesame: A data set of semantically similar java methods," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19, 2019, p. 529–533.

[164] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep., pp. 476–480.

[165] S. Wagner, A. Abdulkhaleq, I. Bogicevic, J.-P. Ostberg, and J. Ramadani, "How are functionally similar code clones syntactically different? an empirical study and a benchmark," *PeerJ PrePrints*, vol. 4, p. e1516, 2016.

[166] B. Efron and R. Tibshirani, *An Introduction to the Bootstrap*. Macmillan Publishers Limited. All rights reserved, 1993.

[167] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," 2008.

[168] N. Mehrotra, A. Sharma, A. Jindal, and R. Purandare, "Improving cross-language code clone detection via code representation learning and graph neural networks," *IEEE Trans. Software Eng.*, vol. 49, no. 11, pp. 4846–4868, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3311796

[169] F. Sun, J. Hoffmann, V. Verma, and J. Tang, "Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=r1lfF2NYvH

[170] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.06391

[171] D. van Bruggen, F. Tomassetti, R. Howell, M. Langkabel, N. Smith, A. Bosch, M. Skoruppa, C. Maximilien, ThLeu, Panayiotis, S. K. (@skirsch79), Simon, J. Beleites, W. Tibackx, jean pierre L, A. Rouél, edefazio, D. Schipper, Mathiponds, W. you want to know, R. Beckett, ptitjes, kotari4u, M. Wyrich, R. Morais, M. Coene, bresai, Implex1v, and B. Haumacher, "javaparser/javaparser: Release javaparser- parent-3.16.1," May 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3842713

[172] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[173] Python Core Team, *Python: A dynamic, open source programming language*, Python Software Foundation, 2019. [Online]. Available: https://www.python.org/

[174] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[175] W. Shang, K. Sohn, D. Almeida, and H. Lee, "Understanding and improving convolutional neural networks via concatenated rectified linear units," vol. 48, pp. 2217–2225, 2016. [Online]. Available: http://proceedings.mlr.press/v48/shang16.html

[176] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial*

Intelligence and Statistics, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256.

[177] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[178] "Atcoder," 2012. [Online]. Available: https://atcoder.jp/

[179] "Codechef," 2009. [Online]. Available: https://www.codechef.com/

[180] "Kaggle," 2010. [Online]. Available: https://www.kaggle.com/arjoonn/codechef-competitive-programming

[181] M. Kamp, P. Kreutzer, and M. Philippsen, "Sesame: A data set of semantically similar java methods," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 529–533.

[182] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, Eds., 2016, pp. 3837–3845. [Online]. Available: https://proceedings.neurips.cc/paper/2016/hash/04df4d434d481c5bb723be1b6df1ee65-Abstract.html

[183] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[184] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1–12.

[185] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272).* IEEE, 1998, pp. 368–377.

[186] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE).* IEEE, 2014, pp. 18–33.

[187] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *14th working conference on reverse engineering (WCRE 2007).* IEEE, 2007, pp. 170–178.

[188] N. Göde and J. Harder, "Clone stability," in *2011 15th European Conference on Software Maintenance and Reengineering.* IEEE, 2011, pp. 65–74.

[189] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is duplicate code more frequently modified than non-duplicate code in software evolution? an empirical study on open source software," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, 2010, pp. 73–82.

[190] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[191] L. Barbour, F. Khomh, and Y. Zou, "An empirical study of faults in late propagation clone genealogies," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1139–1165, 2013.

[192] P. Jablonski and D. Hou, "Cren: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, 2007, pp. 16–20.

[193] J. Li and M. D. Ernst, "Cbcd: Cloned buggy code detector," in *2012 34th International Conference on Software Engineering (ICSE).* IEEE, 2012, pp. 310–320.

[194] A. Lozano and M. Wermelinger, “Assessing the effect of clones on changeability,” in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 227–236.

[195] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider, “An empirical study on bug propagation through code cloning,” *Journal of Systems and Software*, vol. 158, p. 110407, 2019.

[196] M. Mondal, C. K. Roy, and K. A. Schneider, “An empirical study on clone stability,” *ACM SIGAPP Applied Computing Review*, vol. 12, no. 3, pp. 20–36, 2012.

[197] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: A multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[198] C. K. Roy and J. R. Cordy, “Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in *2008 16th iEEE international conference on program comprehension*. IEEE, 2008, pp. 172–181.

[199] H. Li and S. Thompson, “Incremental clone detection and elimination for erlang programs,” in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2011, pp. 356–370.

[200] C. Brown and S. Thompson, “Clone detection and elimination for haskell,” in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, 2010, pp. 111–120.

[201] N. Göde and R. Koschke, “Incremental clone detection,” in *2009 13th European conference on software maintenance and reengineering*. IEEE, 2009, pp. 219–228.

[202] H. Li and S. Thompson, “Clone detection and removal for erlang/otp within a refactoring environment,” in *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, 2009, pp. 169–178.

[203] K. Sohn, S. Liu, G. Zhong, X. Yu, M.-H. Yang, and M. Chandraker, “Unsupervised domain adaptation for face recognition in unlabeled videos,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[204] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, “What do code models memorize? an empirical study on large language models of code,” 2023.

[205] B. Sun, J. Feng, and K. Saenko, “Correlation alignment for unsupervised domain adaptation,” in *Domain Adaptation in Computer Vision Applications*, ser. Advances in Computer Vision and Pattern Recognition, G. Csurka, Ed. Springer, 2017, pp. 153–171. [Online]. Available: https://doi.org/10.1007/978-3-319-58347-1_8

[206] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell, “Adversarial discriminative domain adaptation,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 2962–2971. [Online]. Available: https://doi.org/10.1109/CVPR.2017.316

[207] K. You, M. Long, Z. Cao, J. Wang, and M. I. Jordan, “Universal domain adaptation,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2720–2729.

[208] H. Tang and K. Jia, “Discriminative adversarial domain adaptation,” in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 5940–5947. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/6054