Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

2-2022

Modeling functional similarity in source code with graph-based Siamese networks

Nikita MEHROTRA

Navdha AGARWAL

Piyush GUPTA

Saket ANAND

David LO
Singapore Management University, davidlo@smu.edu.sg

See next page for additional authors

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the Graphics and Human Computer Interfaces Commons, OS and Networks Commons, and the Software Engineering Commons

Citation

MEHROTRA, Nikita; AGARWAL, Navdha; GUPTA, Piyush; ANAND, Saket; LO, David; and PURANDARE, Rahul. Modeling functional similarity in source code with graph-based Siamese networks. (2022). *IEEE Transactions on Software Engineering*. 48, (10), 3771-3789.
Available at: https://ink.library.smu.edu.sg/sis_research/7658

This Journal Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Author

Nikita MEHROTRA, Navdha AGARWAL, Piyush GUPTA, Saket ANAND, David LO, and Rahul PURANDARE

This journal article is available at Institutional Knowledge at Singapore Management University: https://ink.library.smu.edu.sg/sis_research/7658

Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks

Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare

arXiv:2011.11228v2 [cs.SE] 25 Nov 2020

Abstract—Code clones are duplicate code fragments that share (nearly) similar syntax or semantics. Code clone detection plays an important role in software maintenance, code refactoring, and reuse. A substantial amount of research has been conducted in the past to detect clones. A majority of these approaches use lexical and syntactic information to detect clones. However, only a few of them target semantic clones. Recently, motivated by the success of deep learning models in other fields, including natural language processing and computer vision, researchers have attempted to adopt deep learning techniques to detect code clones. These approaches use lexical information (tokens) and(or) syntactic structures like abstract syntax trees (ASTs) to detect code clones. However, they do not make sufficient use of the available structural and semantic information hence, limiting their capabilities. This paper addresses the problem of semantic code clone detection using program dependency graphs and geometric neural networks, leveraging the structured syntactic and semantic information. We have developed a prototype tool HOLMES, based on our novel approach and empirically evaluated it on popular code clone benchmarks. Our results show that HOLMES performs considerably better than the other state-of-the-art tool, TBCCD. We also evaluated HOLMES on unseen projects and performed cross dataset experiments to assess the generalizability of HOLMES. Our results affirm that HOLMES outperforms TBCCD since most of the pairs that HOLMES detected were either undetected or suboptimally reported by TBCCD.

Index Terms—Program representation learning, Semantic code clones, graph-based neural networks, siamese neural networks, program dependency graphs

1 INTRODUCTION

CODE clones are code fragments that are similar according to some definition of similarity [1]. There are two types of similarity defined between code snippets: 1) syntactic (textual) similarity and 2) semantic similarity. Syntactic clones are code pairs that have similar syntactic structure. They share similar (or nearly similar) program text, control flow, data flow, and data-types. Semantic clones are syntactically dissimilar code snippets that share similar functionality [2]. Figure 1 shows an example of semantic clones that sorts an array of natural numbers.

Code clones materialize in a software project when developers reuse the existing code by *copy-paste-modify* operation or when they re-implement an already existing similar functionality [2], [3]. Code clones can lead to increased software maintenance costs [4], [5]. They may complicate software evolution as bug fixes and changes have to be propagated to all the clone locations [1], [6], [7], [8]. However, clones are not always disastrous [9]. They can aid in code search [10], refactoring [11], and bug detection [12].

A substantial amount of research effort has been put in to detect and analyze syntactic clones. These techniques [1], [13], [14] use various handcrafted lexical and syntactic program features to identify similar (clone) pairs. However, in recent years with the growing research efforts into applying deep learning techniques for software engineering problems, researchers have adopted deep learning models to detect software clones [3], [15], [16], [17], [18], [19], [20], [21], [22]. The code clone detection process begins by modeling the functional behaviors of the source code. 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 being used to learn program features. For instance, Yu et al. [15] used tree-based convolutions that exploit structural and lexical information from the ASTs of the code fragments. 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 source code.

A few techniques exploit Program Dependence Graphs (PDGs) for measuring code functional similarity. These techniques construct program dependence graphs for each code snippet and use graph isomorphism to measure code functional similarity. For instance, Krinke [23] used PDG representation of code snippets and modeled clone detection problem as *maximal similar subgraph construction* problem. Gabel et al. [24] compared program dependence graphs of code pairs to detect clones. They reduced the graph isomorphism problem to a tree matching problem by mapping PDG representation to AST. 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 in [24].

Addressing the above issues in this paper, 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 [3].

- *N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, and R. Purandare are with the Department of Computer Science Engineering, IIIT Delhi, India (e-mail: nikitam@iiitd.ac.in, navdha16250@iiitd.ac.in, piyush16066@iiitd.ac.in, anands@iiitd.ac.in, purandare@iiitd.ac.in).*
- *D. Lo is with the School of Information Systems SMU, Singapore (e-mail: davidlo@smu.edu.sg)*

```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: $Sort_1.java$

```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--;
    }
    Formatter formatter = new Formatter();
    System.out.println(formatter.format("Case#"+i+":
    %.6f",count));
}
```

Listing 2: $Sort_2.java$

Fig. 1: A semantic code clone example detected by HOLMES, which was reported as false negative by TBCCD. The code in Listings 1 and 2 sort an array of natural numbers by randomly shuffling the array n times.

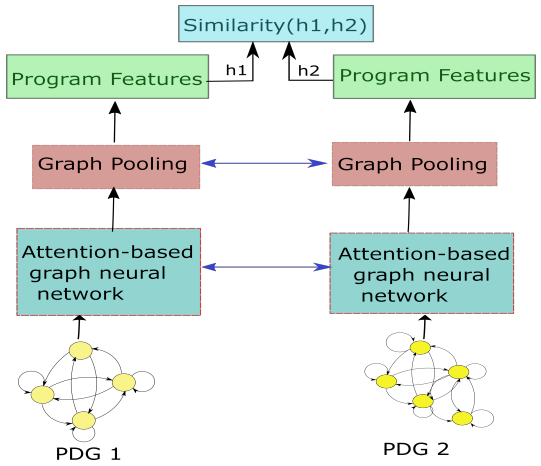


Fig. 2: The proposed Siamese deep neural network consists of two identical sub-networks. The input to the subnetwork is the pair of Java methods represented as PDGs. Each subnetwork incorporates an attention-based graph neural network model to learn PDGs node features, which are then aggregated using 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 denote that the two sub-networks share the same set of weights and parameters.

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 also 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 2 shows the overall architecture of the HOLMES.

We have implemented HOLMES in Java using the Soot optimization framework [25] and Pytorch Geometric [26] deep learning library. We evaluated HOLMES on programming competition datasets and real-world datasets. Our empirical results show that HOLMES outperforms another state-of-the-art tool TBCCD [15] and generalizes better on unseen code pairs.

We make the following contributions in this paper:

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

Paper Organization Section 2 provides an example to motivate our approach. Section 3 presents an overview of basic concepts and code clone terminology. Section 4 describes the code clone detection process and explains the Graph-based Siamese deep neural network used in our approach. Section 5 details the experimental design and evaluation process. Section 6 discusses the results. Section 7 presents

a qualitative analysis of HOLMES. Section 8 discusses the threats to the validity of our proposed approach. Section 9 surveys the related work, and finally, Section 10 concludes our paper with a summary of findings.

2 MOTIVATION

2.1 Motivating Example

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

Listings 1 and 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 1 implements the above functionality by first initializing an array of size T with random numbers. It then checks if the current index element is equal to the index of the next element. The average number of hits 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 is equal to the next index.

Listing 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 average hits required by -1. Syntactically, Listings 1 and 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 [15], [17], [18] used syntactic and(or) lexical information to learn program features. For instance, TBCCD [15] used tree-based convolution over abstract syntax trees (ASTs) to learn program representation. If we look at the ASTs of Listings 1 (for line $8-11$) and 2 (for line $5-8$) shown in Figures 3 and 4, it's 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 1 and 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 program dependency graphs (PDGs) for Listings 1 and 2. The PDGs are shown in Figures 5 and 6. If we look at the Figures 5 and 6, we will 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 the 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*

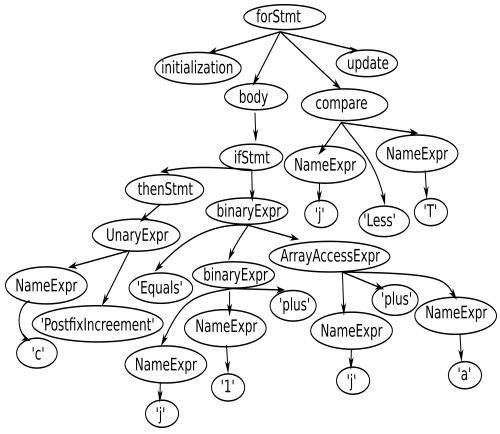


Fig. 3: AST for Listing 1.

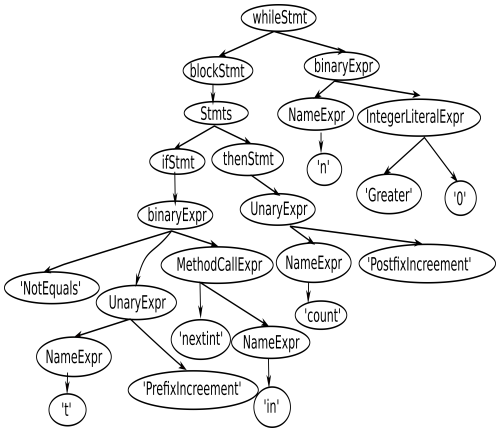


Fig. 4: AST for Listing 2.

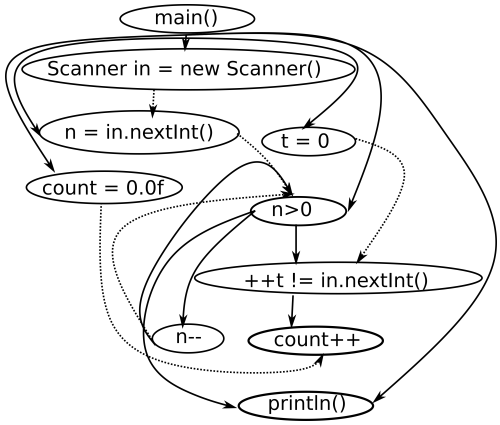


Fig. 5: PDG for Listing 1.

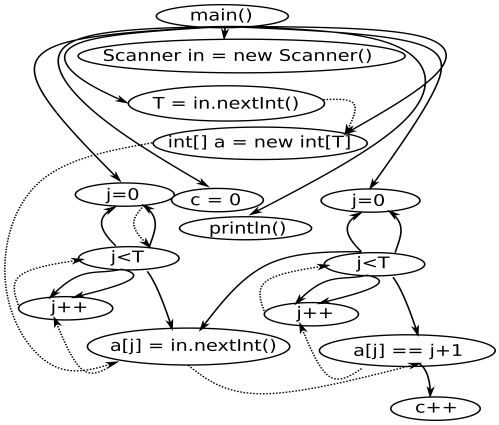


Fig. 6: PDG for Listing 2.

learn to give higher weights to semantically relevant paths.

Source code is a complex web of interacting components such as classes, routines, program statements, etc. Understanding source code amounts to understanding the interactions between different components. Previous studies such as [27] 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 [3], [16], [17] 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.*

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

3 Background

This section gives a brief overview of the basic concepts and defines the terminology used in the paper.

3.1 Program Dependence Graphs

Program Dependence Graph (PDG) is a directed attributed graph that explicitly encodes a program's control, and data dependence information [28]. PDGs approximate program semantics. A node in a PDG represents a program statement such as an assignment statement, a declaration statement, or a method invocation statement, and the edges denote control or data dependence 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 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 program dependence graphs 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 $s_1 \in G$ and $s_2 \in G$ 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 and
- 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 a control flow graph 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 variable v.

Program dependence graphs connect the computationally related parts of the program statements without enforcing the control sequence present in the control flow graphs [28]. Hence, they are not affected by syntactical changes like statement reordering, variable renaming, etc. [28]. These properties make program dependence graphs to be better representation to detect semantic clones. Horwitz [29] and Podgurski and Clarke [30] also showed that program dependence graphs provide a good representation to measure code semantic similarity.

3.2 Concepts in Deep Learning

3.2.1 Artificial neural networks

ANNs [31] 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.

3.2.2 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 [32], convolutional networks [33], graph-based neural networks [34] etc. In this work, we make use of graph-based neural networks.

3.2.3 Graph Neural Networks

DNNs have shown unprecedented performance in many complex tasks such as image processing [35] and neural machine translation [36]. DNN architectures like transformers [37] and convolutional networks [38] have often demonstrated performance at par with humans. The key reason behind the success of DNNs is the model's ability to take input data directly and learn to extract feature representations relevant to a complex downstream task like classification or retrieval.

Despite state-of-the-art results, the above models do not perform well in non-Euclidean domains such as graphs and manifolds. The inherent complexity of the data, variegated structural and topological information hampers the ability to gain true insights about the underlying graphical data

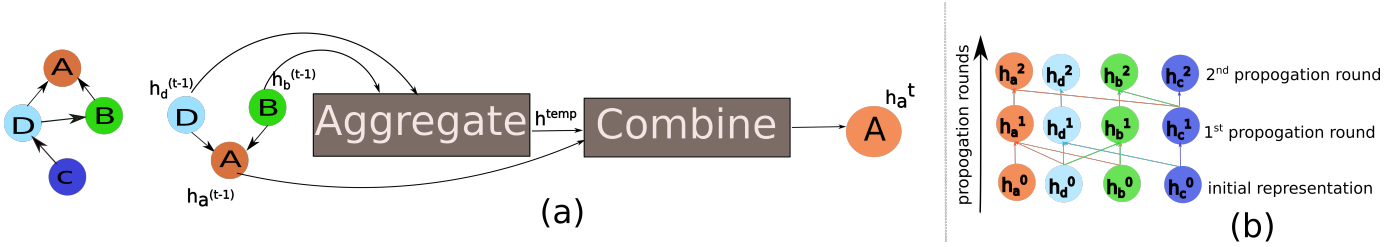


Fig. 7: Visual illustration of the graph neural network framework. (a) Illustration of the t^{th} layer of the graph neural network. The feature vectors h_b^{t-1}, h_d^{t-1} from the neighbouring nodes of A are *aggregated* and *combined* with h_a^{t-1}, the features of node A from the $t-1^{th}$ layer. This constitutes the representation of node A at the t^{th} layer. (b) Illustration showing multiple rounds of propagation in a graph neural network. At the n^{th} 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.

[34], [39], [40]. Nevertheless, one may have to deal with graph-structured data in various fields. For example, in software engineering, programs are modeled as graphs (ASTs, PDGs, etc.) for automatic code summariation [41], identifying vulnerabilities [42], and bug-fixing activities [43].

Dealing with non-Euclidean structured data implies that there are no such properties as the shift-invariance and the vector space structures [44]. Hence, convolutions and filterings are not well defined here. Therefore, spectral-domain [45] and spatial domain [46] techniques have been adopted to learn representation of the graph-structured data.

Our work makes use of the technique from the spatial (vertex) domain. Spatial graph convolutions define convolution operations based on the node's spatial connections and are built on the idea of message passing. The graph convolutional operator learns a function f to generate node v_i's representation by aggregating its own features h_i and neighbor's features h_j. Multiple iterations of graph convolution are performed to explore the depth and breadth of the node's influence. Each iteration uses node representation learned from the previous iteration to get the representation for the current one. For instance, in the first iteration of graph convolution, information flow will be between first-order neighbors; in the second iteration, nodes will receive information from second-order neighbors, i.e., neighbor's neighbor. Thus traversing this way, after multiple iterations, each node's final hidden representation will have information from a further neighborhood. Figure 7 depicts the general framework for spatial graph convolutions.

3.2.4 *Siamese Neural Networks*

Siamese neural network or twin network [47], [48] 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 the input data. They try to learn a mapping function such that the distance measure between the learned latent features in the target space represents the semantic similarity in the input space.

3.3 Representation learning in software engineering

Treating program as data objects and learning syntactically and semantically meaningful representations have drawn a great deal of interest [49], [50], [51].

Following the success of deep neural networks in natural language processing, computer vision, etc., learning tasks on source code data have been considered recently. Program synthesis [52], [53], program repair [54], bug localization [51], [55], and source code summarization [56] are some of the well-explored areas. The idea is to use the knowledge from the existing code repositories to enable a wide array of program analysis and maintenance tasks. The key step is to design a precise and semantically meaningful program representation that neural networks will use in the array of downstream tasks.

Most existing approaches use two kinds of program representations extracted from static and dynamic program analysis techniques. These representations can further be categorized into syntactic and semantic program representations. Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Call Graphs, etc., represent the program's syntactic structure while Program Dependence Graphs(PDGs), execution traces, etc., capture program semantics. These representations help to transform programs in an appropriate form to deep learning models.

3.4 Deep learning for code clone detection

Learning-based techniques automatically (using neural networks) learn a continuous-valued feature vector representing program semantics and syntax to learn similarities between code snippets. This feature vector is then compared directly (using a distance-based classifier) or is passed to a neural network classifier to predict similarity.

For instance, White et al. [3] used a recursive neural network to learn program representation. They represented source code as a stream of identifiers and literals and used it as an input to their deep learning model. Tufano et al. [21] used a similar encoding approach as [3] and encoded four different program representations- identifiers, Abstract Syntax Trees, Control Flow Graphs, and Bytecode. They then used a deep learning model to measure code similarity based on their multiple learned representations. Zhao and Huang [16] used a feed-forward neural network to learn a semantic feature matrix constructed from a program's control flow and data flow graphs. Yu et al. [15] used a tree-based convolutional neural network to detect code clones.

```
1  //orginal code snippet
2  static int gcd(int a, int b)
3  {
4      if (b == 0)
5          return a;
6      return gcd(b, a % b);
7  }
8
9  //type 1 clone
10 static int gcd1(int a, int b) {
11     if (b == 0){
12         return a;
13     }
14     return gcd1(b, a % b);
15 }
16
17 //type 2 clone
18 public static int gcd2(int no1, int no2) {
19  if (no2 == 0) {
20      return 1;
21  }
22  return gcd2(no2, no1 % no2);
23
24
25 //type 3 clone
26 public static int gcd3(int m, int n) {
27     if (0 == n) {
28         return m;
29     } else {
30         return gcd3(n, m % n);
31     }
32 }
33
34 //type 4 clone
35 static int gcd4(int a, int b) {
36     while (b != 0) {
37         int t = b;
38         b = a % b;
39         a = t;
40     }
41     return a;
42 }
```

Listing 3: Different clone types of gcd

These code clone detection approaches have used syntactic and lexical features to measure code similarity. They do not exploit the source code's available structured semantics, even though this information might be useful to measure code functional similarity. Hence to overcome the limitations of existing approaches, we have proposed a novel code clone detection tool HOLMES. HOLMES uses PDGs and graph-based neural networks to learn structured semantics of the source code. Section 4 explains the code clone detection process of HOLMES.

3.5 Terminologies

This paper 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.

Code clones are pairs of similar code snippets existing in a source file or a software system. Researchers have broadly classified clones into four categories stretching from syntactic to semantic similarity [2]:

- **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 w.r.t. to each other.
- **Type-4 clones (semantic similarity)**: Syntactically different code snippets implementing the same functionality.

Listing 3 enumerates different clone types from the BigCloneBench dataset. The original code snippet (starting from line 2) computes the greatest common divisor (gcd) of two numbers. The Type-1 clone (starting from line 10) of the original code snippet is identical except for the formatting variation. The Type-2 clone (starting from line 18) have different identifier names (no1 and no2). Type-3 clone (starting from line 26) of the original code snippet is syntactically similar but differs at the statement level. Finally, the Type-4 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.

4 OUR APPROACH

This section discusses the details of our graph neural network architecture that is used to learn high level program features from program dependency graphs. Figure 8 shows an overview of one branch of the Siamese neural network shown in Figure 2 . The following subsections give details of the main steps of the proposed approach.

4.1 Attention Based Global Context Learning

Our work builds on Graph Attention Networks (GAT) [57], 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 node feature matrix is represented by $X \in \mathbb{R}^{|V|\times d}$ with $x_v \in \mathbb{R}^d$ denoting 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`, `LookupSwitch`, `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

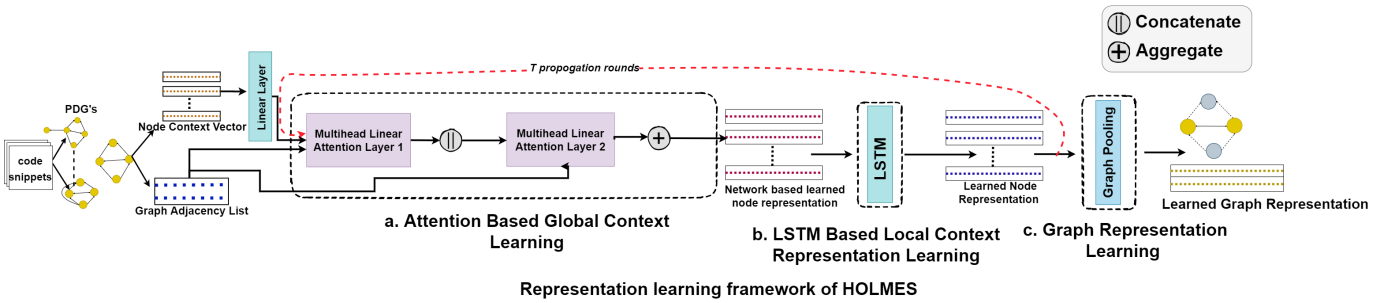


Fig. 8: The architecture of one branch of the Siamese neural network is shown in Figure 2. (a) Our model first parses the given Java methods in the datasets to build PDGs. Node feature matrix and graph adjacency matrix are extracted from the source code. HOLMES then passes this as input to a multi-head 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.

[0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]. In the first place, to obtain initial node vectors, we pass node features through a linear transformation layer:

$$H^0 = X \times W + b \tag{1}$$

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

Next, to obtain node features for a given graph, we learn an adaptive function $\varphi(A, H; \phi)$ parametrized by ϕ similar to GAT [57]. The input to the function is the set of node features $\{h_1^0, h_2^0, \cdots, h_{|V|}^0\}$ obtained from Equation (1). The function φ then outputs the set of new node features $\{h_1^{'}, h_2^{'}, \cdots, h_{|V|}^{'}\}$ as the output of the first attention block. It computes the self-attention on nodes based on the graph structural information where a node v_j attends to its one-hop neighboring node v_i, i.e., if $(v_i, v_j) \in E$. The attention mechanism $a : \mathbb{R}^{d^{'}} \times \mathbb{R}^{d^{'}} \rightarrow \mathbb{R}$ computes attention coefficients

$$e_{ij} = a(Wh_i, Wh_j) \tag{2}$$

Equation (2) denotes the importance of node $j's$ features for node i. The scores are then normalized using the softmax function

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{j \in N(i)} exp(e_{ij})} \tag{3}$$

The attention scores computed in Equation (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.

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

$$h_i^{'} = \|_{k=1}^{8} \sigma \left(\sum_{j \in N(i)} \alpha_{ij}^k W^{'k} h_j^0 \right) \tag{4}$$

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 5 represents the output of attention module 2. Here, we have aggregated the output from different attention nodes.

$$h_i^{''} = \sigma \left(\sum_{k=1}^{6} \sum_{j \in N(i)} \beta_{ij}^k W^{''k} h_j^{'} \right) \tag{5}$$

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 node representation $h_i^{''}$ is input to the LSTM module as $h_i^{''(0)}$ in the first propagation round. As shown in Figure 2, the LSTM module's output, $h_i^{(1)}$, is fed back to the first attention block as an input. The process described above is repeated to update the node representations, with the output of the second attention block $h_i^{''(t-1)}$ as input to the LSTM at the t^{th} propagation round. The T propagation rounds result in learned representations of the individual nodes that 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.

4.2 LSTM Based Local context Learning

The multi-head attention mechanism enables node representations to capture the context from their one-hop neighbors. However, the semantic context within a code fragment typically requires a broader context provided by a node's t-hop neighbors. Inspired by the architecture of the adaptive path layer in GeniePath [58], we use an LSTM layer, which when combined with the multi-head attention previously described, helps learn node representations that are better equipped to capture the semantic information of the code fragment. The input to the LSTM model at the $(t+1)^{th}$ propagation round is $h_i^{''(t)}$, the representation of the i^{th} node.

Algorithm 1: Code Similarity Detection

Input: T rounds of propagation, $train\_Data$
Output: $code\_similarity$

Define *ComputeNodeFeatures*(H^0, A, T):
  $H_F = [H^0]$
  // C^0 is the initial cell state of $LSTM$
  Initialize Array C^0
  for $t \leftarrow 1$ **to** T **do**
    // Attention block 1 (Eqn. (4))
    $H^{'} = Attn_1\left(A, H^{(t-1)}; \phi_1\right)$
    // Attention block 2 (Eqn. (5))
    $H^{''(t-1)} = Attn_2\left(A, H^{'}; \phi_2\right)$
    // t^{th} propagation round (Eqn. (6))
    $H^{(t)}, C^{(t)} = LSTM(H^{''(t-1)}, C^{(t-1)})$
    // Final node features (Eqn. (7))
    $H_F = CONCAT(H^{(t)})$
  return H_F

Define *GraphPooling*(H_{node}):
  $H_G = a(MLP(H_{node})) \odot MLP(H_{node})$
  return H_G

Define *ComputeGFeatures*(X, A_c, A_d, T):
  $H^0 = X \times W + b$
  H_d =ComputeNodeFeatures(H^0, A_d, T)
  H_c =ComputeNodeFeatures(H^0, A_c, T)
  $H_{final} = H_d + H_c$
  G_{final} =GraphPooling(H_{final})
  return G_{final}

Define *Train*($train\_data$, T):
  while $not\_converged$ **do**
    while $data$ *in* $train\_data$ **do**
      $X_1 = data.X_1$
      $X_2 = data.X_2$
      $A_{c1} = data.A_{control1}$
      $A_{d1} = data.A_{data1}$
      $A_{c2} = data.A_{control2}$
      $A_{d2} = data.A_{data2}$
      $Y = data.Y$
      G_1 =ComputeGFeatures(X_1, A_{c1}, A_{d1}, T)
      G_2 =ComputeGFeatures(X_2, A_{c2}, A_{d2}, T)
      $featureRep = CONCAT(G_1, G_2)$
      $featureRep = a(MLP(featureRep))$
      $similarity = a(MLP(featureRep))$
      $loss = LOSS(similarity, Y)$
      $Update\_Optimizer(loss)$

This strategy allows the node representations to capture the context from its t-hop neighbors [58].
We initialize the LSTM cell with random values. The cell state $C_j^{(t)}$ corresponding to the j^{th} node ($j \in V$) is updated in the t^{th} propagation round, effectively aggregating information from t-hop neighbors of node j. The node representation is then accordingly updated as a function of the cell state. The update Equations are presented below.

$$\left.\begin{aligned} i_j &= \sigma(W_i h_j^{''(t-1)}), & f_j &= \sigma(W_f h_j^{''(t-1)}), \\ o_j &= \sigma(W_o h_j^{''(t-1)}), & \widetilde{C}_j &= tanh(W_c h_j^{''(t-1)}), \\ C_j^{(t)} &= f_j \odot C_j^{(t-1)} + i_j \odot \widetilde{C}_j, & h_j^{(t)} &= o_j \odot tanh(C_j^{(t)}) \end{aligned}\right\} \quad (6)$$

Where $\odot$ denotes element-wise multiplication. The input gate of LSTM i_j is being used to extract new messages from the input $h_j^{''(t-1)}$ and are added to memory $C_j^{(t)}$. The gated unit f_j is the forget gate used to filter out unwanted messages from the old memory $C_j^{(t-1)}$. Lastly, the output gate o_j and the updated memory $C_j^{(t)}$ are used for constructing the final node representation $h_j^{(t)}$ at $(t)^{th}$ propagation round for node j.

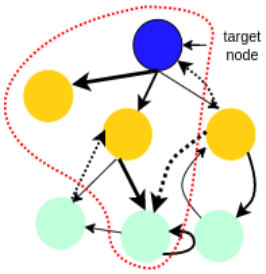


Fig. 9: A synthetic example showing explored receptive path (area covered by the dotted red line) for the target node. The edge thickness denotes the received attention scores while learning features for the target node. Control and data-dependent edges are shown through solid and dotted edges, respectively.

Figure 9 conveys the above exposition through a synthetic example. HOLMES tries to filter and aggregate meaningful features from different two-hop neighbors while learning representation for the target node. The multi-head attention module in each propagation round attends to different neighbors (edge width in Figure 9 denotes the importance of different hop neighbors at each propagation step). The LSTM module filters and aggregates the messages received from different hop neighbors over multiple propagation rounds. The area covered by the red dotted line denotes the relevant neighboring nodes (receptive path) for learning the feature representation of the target node.

4.3 Graph Representation Learning with Jumping Knowledge Networks

To learn the high-level program features from the program dependence graphs, our graph neural network (GNN) model iteratively aggregates the node features from different n^{th} hop neighbors via message passing scheme, described in Sections 4.1 and 4.2.

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 19).

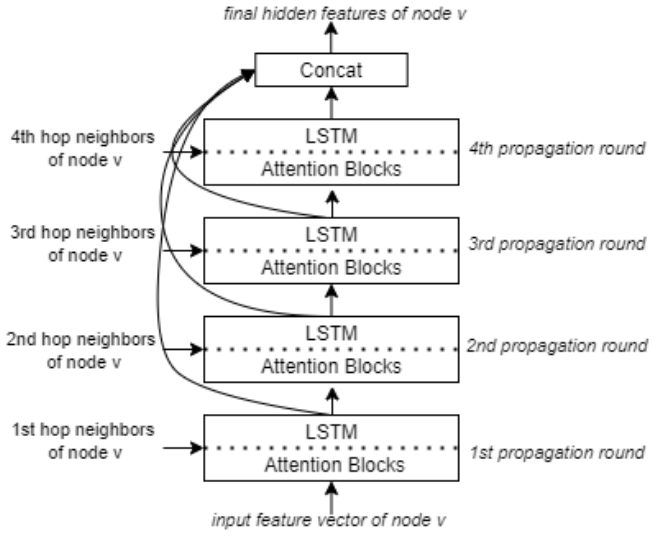


Fig. 10: Illustration of the 4-layer architecture of HOLMES using Jumping Knowledge Networks (JK nets) defined by Xu et al. [59]. At every propagation round t, the feature vector of node v is aggregated with its t^{th} order neighbors. At the last layer (4^{th} propagation round), all the hidden feature vectors from all the propagation rounds are concatenated to constitute the final hidden representation for node v. The concatenation of hidden features from all the propagation rounds ensures that the features learned from n^{th} hop neighbors during different propagation rounds are retained till last and also reflected in the final hidden representation of node v.

Thus, to stabilize the training and learn the diverse local neighborhood for each node, we employed Jumping Knowledge Networks (JK) [59], shown in Figure 10. JK combines (concatenates; denoted by $\|$ operator below) the hidden features ($H^{(t)}$ defined on Line 7, Algorithm 1) learned at each GNN iteration independently:

$$H_F = \left[H^0 \| H^{(1)} \| \cdots \| H^{(T)}\right] \tag{7}$$

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

For obtaining graph level representation from the learned node feature vectors, we have employed a soft attention mechanism proposed by Li et al. [60]:

$$H_G = \left(\sum_{i \in V} \sigma\big(MLP(h_i^{(T)})\big) \odot MLP(h_i^{(T)})\right) \tag{8}$$

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.

4.4 Edge-attributed PDGs

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.

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 matrix 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 $H_{control}$.

$$\left.\begin{aligned} H_{data} &= ComputeNodeFeatures(H_0, A_d, T) \\ H_{control} &= ComputeNodeFeatures(H_0, A_c, T) \\ H_{final} &= H_{data} + H_{control} \end{aligned}\right\} \tag{9}$$

Thereafter, to obtain the final graph representation G_{final} we applied graph pooling defined in Section 4.3 on the learned node feature matrix H_{final}.

$$G_{final} = GraphPooling(H_{final}) \tag{10}$$

4.5 Implementation and Comparative Evaluation

We have used the Soot optimization framework [25] to build program dependence graphs. To compute the control dependence graph, we first build a control flow graph. Then Cytron's method [61] is used to compute control dependence. For computing data dependence graph, reaching definition [62] and upward exposed analysis [63] is used.

We have used the PyTorch geometric [64] deep learning library to implement HOLMES. All LSTMs have a single LSTM layer with 100 hidden units. We have used the LeakyReLU [65] 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 [66]. The Siamese network is trained to minimize binary cross-entropy loss given in Equation 11 using Adam [67] optimizer with a learning rate set to 0.0002 and batch size to 50.

$$BCELoss = -\frac{1}{N}\sum_{i=1}^{N} y_i \times log(p(y_i)) + (1-y_i) \times log(1-p(y_i)) \tag{11}$$

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 5.2.2

For the BigCloneBench (BCB) dataset since it does not provide the input files' dependency information, we used

TABLE 1: Dataset Statistics.

Dataset	Language	Project Files	Clone Pairs	Non-clone Pairs
GCJ	Java	9,436	4,40,897	5,00,000
SeSaMe	Java	11 Java projects	93	n.a.
BCB	Java	9134	6,50,000	6,50,000

TABLE 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

JCoffee [68] 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. We compared HOLMES with the state-of-the-art code clone detection tool TBCCD [15]. Other recent machine-learning-based code clone detection tools namely, CDLH [17] and CDPU [18] do not have their implementation available open-source. DeepSim [16] 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 [14], RtvNN [16] and Sourcerer [69] as TBCCD significantly outperformed these approaches. Therefore, TBCCD became our natural choice for comparative evaluation.

5 EXPERIMENTAL DESIGN

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

RQ1: How effective is HOLMES as compared to other state-of-the-art approaches?

RQ2: How well HOLMES generalizes on unseen projects and data sets?

5.1 Dataset Collection

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

1) Programming Competition Dataset: We followed the recent work [16] and used code submissions from GoogleCodeJam[1] (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 [16] that used 12 different functionalities in their experiments, we collected 9436 solutions from 100 different functionalities from GCJ. Thus, building a large and representative dataset for evaluation. Detailed statistics are reported in Table 1. 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.

a) SeSaMe dataset. SeSaMe [70] dataset consists of semantically similar method pairs mined from 11 open-source Java repositories. The authors applied text similarity measures on Javadoc 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) [71] dataset, released by Svajlenko et al., was developed from IJAdataset-2.0[2]. 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 a BCB covers ten functionalities, including $6M$ clone pairs and $260K$ non-clone pairs. The current release of the BCB dataset has about $8M$ tagged clones pair covering 43 functionalities. Some recent code clone detection tools TBCCD [15], CDLH [17] 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 1 and 2 summarises the data distribution of the BCB dataset.

5.2 Experimental Procedure and Analysis

5.2.1 *RQ1: 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 [15], a state-of-the-art clone detector that

1. https://code.google.com/codejam/past-contests

2. https://sites.google.com/site/asegsecold/projects/seclone

uses AST and tree-based convolutions to measure code similarity. We followed similar experimental settings as used by Yu et al. in TBCCD [15]. To address this RQ, 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 use the same code fragments from the related work [15], [17]. We had 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 did 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): In this variant, we did not differentiate between the control and data-dependent edges to learn the program features.

2) Edge-Attributed HOLMES (EA-HOLMES): Program dependence graphs 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}).

5.2.2 RQ2: How well HOLMES generalizes on unseen projects and data sets?

To evaluate the robustness and generalizability of EU-HOLMES and EA-HOLMES, we evaluated the proposed approaches on unseen projects. In particular, we took EU-HOLMES, EA-HOLMES, and TBCCD trained on the GCJ dataset. We then tested the stability of the above tools on the following datasets:

1) GoogleCodeJam (GCJ*): We used the dataset of functionally similar code snippets (FSCs) proposed by Wagner et al. [72]. This dataset comprises of 32 clone pairs from *GCJ2014*. The authors classified the pairs into full syntactic similarity and partial syntactic similarity. The clone pairs are further classified into five categories – *Algorithms, Data Structures, Input/Output, Libraries, and Object-Oriented Design*. Each category has three different clone pairs classified based on the degree of similarity – *low, medium, and high*.

2) SeSaMe Dataset: This dataset [70] 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.

6 RESULTS

The results of our comprehensive evaluation are summarized in this section.

TABLE 3: Comparative evaluation with TBCCD variants.

Tool	#Params	BCB			GCJ		
		P	R	F1	P	R	F1
TBCCD(-token)	2.1×10^5	0.77	0.73	0.74	0.77	0.80	0.80
TBCCD	1.7×10^5	0.96	.96	0.96	0.79	0.85	0.82
EU-HOLMES	1.7×10^6	0.72	0.97	0.83	0.84	0.92	0.88
EA-HOLMES	6.6×10^6	**0.97**	**0.98**	**0.98**	**0.91**	**0.93**	**0.92**

TABLE 4: F1 value comparison w.r.t various clones types in BigCloneBench dataset.

Tools	T1	T2	ST3	MT3	WT3/T4
TBCCD(-token)	**1.0**	0.90	0.80	0.65	0.60
TBCCD	**1.0**	**1.0**	0.98	0.96	0.96
EU-HOLMES	**1.0**	**1.0**	0.86	0.80	0.80
EA-HOLMES	**1.0**	**1.0**	**0.99**	**0.99**	**0.99**

6.1 RQ1: How effective is HOLMES as 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. These variants of TBCCD are reported in the paper [15]. The variant TBCCD(-token) uses randomly initialized AST node embeddings in place of source 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, special symbols, etc. Table 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 3, we can see 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 further shows the performance of TBCCD(-token), TBCCD, EA-HOLMES, and EU-HOLMES on different code clone types in the 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 such as Moderately Type-3, Weakly Type-3+4, TBCCD(-token) performs poorly compared to the TBCCD variant, we also see an improvement of ~3% in EA-HOLMES as compared to TBCCD. The reason for the improved performance of TBCCD is attributed to the use of

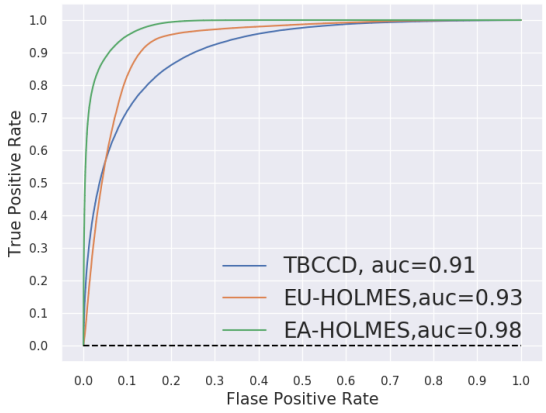


Fig. 11: ROC curve of TBCCD, EU-HOLMES, EA-HOLMES on GCJ dataset. (AUC values are rounded up to 2 decimal places)

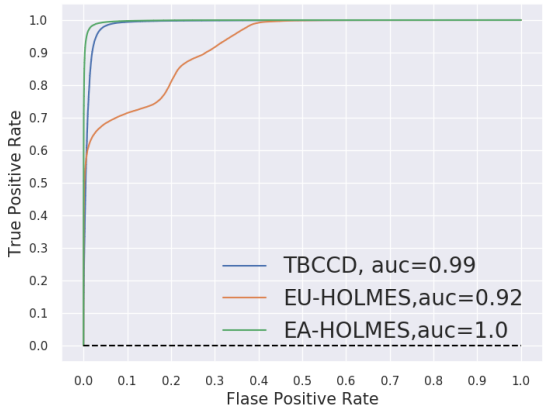


Fig. 12: ROC curve of TBCCD, EU-HOLMES, EA-HOLMES on BCB dataset.(AUC values are rounded up to 2 decimal places)

```
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);
    }
}
//clone pair
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);
}
```

Listing 4: 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.

syntactic similarity existing between the code snippets in the BCB dataset, as shown in Listing 4. This syntactic similarity existing in the form of identifiers, tokens, etc., is exploited by TBCCD while learning for code similarity.

On the other hand, on the GCJ dataset, the performance of both TBCCD's variants, i.e., TBCCD(-token) and TBCCD, drops significantly. We can see an improvement of ~10% in

TABLE 5: Performance on unseen dataset.

Tool	GCJ*			SeSaMe		
	P	R	F1	P	R	F1
TBCCD	1.0	0.63	0.77	1.0	0.48	0.64
EU-HOLMES	1.0	0.65	0.78	1.0	0.52	0.68
EA-HOLMES	1.0	0.87	0.93	1.0	0.85	0.92

the F1-score on the GCJ dataset in EA-HOLMES compared to TBCCD. The performance drop in TBCCD(-token) and TBCCD on the GCJ dataset is attributable to two factors: (1) Both TBCCD's variants use ASTs to learn program features. ASTs represent program syntactic structure, and as shown in Listings 1 and 2, the code clone pairs in GCJ have a significant structural difference. Thus, the ASTs of these code pairs are very different, making it hard for the model to infer similarity; and (2) As opposed to the BCB dataset, where there was some syntactic similarity between the code pairs, the GCJ code clone pairs have substantial differences in structure and algorithm. These differences are not unexpected because the submissions are made by independent programmers implementing the solutions from scratch. Consequently, without modeling semantics, the GCJ dataset's clones are harder to detect compared to BCB.

Also, from Tables 3 and 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.

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 11 and 12 show the ROC curve and corresponding Area Under Curve (AUC) values for the GCJ and BCB dataset. We have plotted the ROC curve of the TBCCD variant only, as it has outperformed the TBCCD(-token) variant on the GCJ and BCB datasets. For all other experiments also, we have considered the TBCCD variant only.

TABLE 6: Detailed analysis of results on unseen GCJ* submissions.

Category	EA-HOLMES						TBCCD					
	Low		Medium		High		Low		Medium		High	
	Full	Part	Full	Part	Full	Part	Full	Part	Full	Part	Full	Part
Data Structure	✓	✓	✓	×	×	✓	×	×	✓	✓	×	×
OO Design	✓	×	✓	✓	✓	✓	✓	×	✓	✓	✓	✓
Algorithm	✓	✓	✓	✓	✓	✓	×	✓	✓	×	×	✓
Library	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	×	×
Input/Output	✓	✓	×	✓	✓	✓	×	✓	✓	✓	✓	×

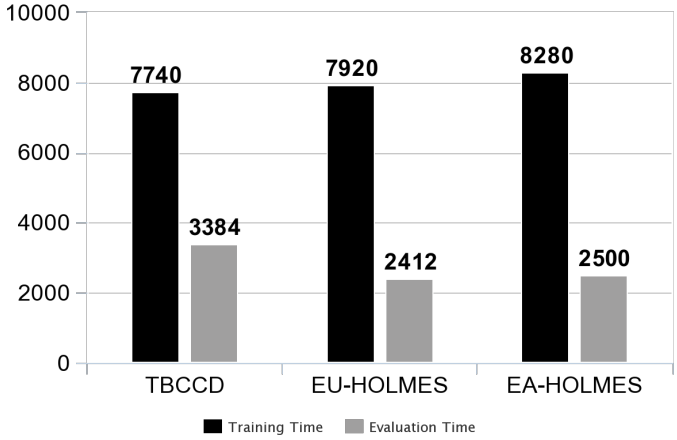


Fig. 13: Time performance analysis on the GCJ dataset.

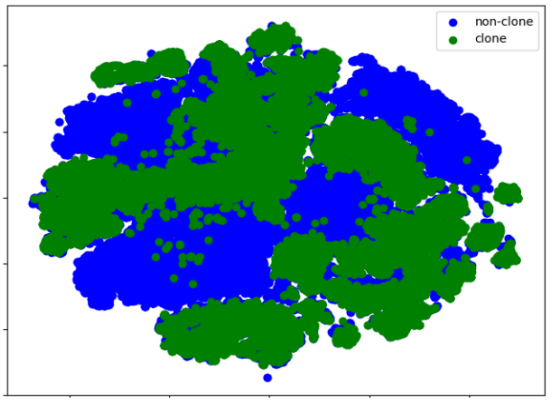


Fig. 14: t-SNE plot of graph embeddings of clone and non-clone pairs of GCJ dataset generated by EA-HOLMES.

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 11 and 12 that EA-HOLMES has the best AUC value on both the datasets.

Time Performance. We also 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 4.5 on the full GCJ 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 AST for 9436 project files was 30 minutes, while PDG took 60 minutes. Figure 13 shows the training and evaluation time analysis of TBCCD, EU-HOLMES, and EA-HOLMES. EA-HOLMES learns separate representation for the control and data dependence graphs. Thus, 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.

6.2 RQ2: How well HOLMES generalizes on unseen projects and data sets?

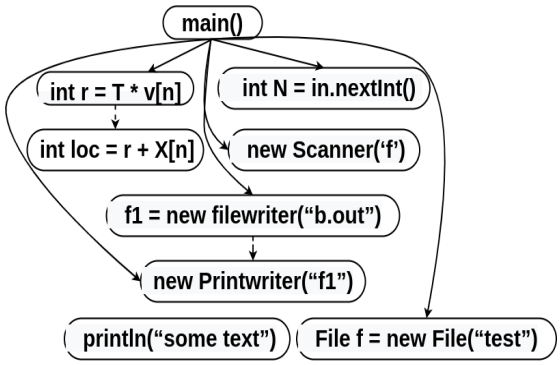
Table 5 shows performance of EU-HOLMES, EA-HOLMES, and TBCCD on unseen datasets. We can see that EA-HOLMES performs considerably better on both datasets as compared to TBCCD.

Table 6 shows the detailed classification result of TBCCD and EA-HOLMES on the GCJ* dataset. In the table, ✓ indicates that the code clone detector detects the pair, while × indicates that the pair goes undetected. From table 6, we can infer that our proposed approach can detect the majority of the pairs correctly, even the pairs with partial syntactic similarity. These results affirm that EA-HOLMES generalizes well on unseen datasets also.

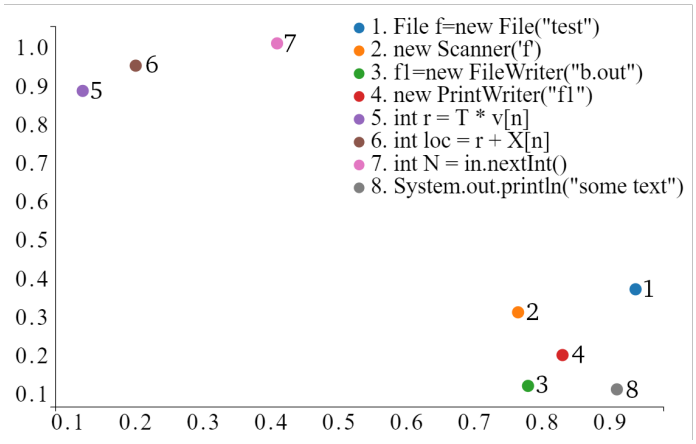
7 DISCUSSION

7.1 Why HOLMES outperforms other state-of-the-art clone detectors

Our approach uses Program Dependence Graphs (PDGs) for representation learning. PDGs represent the program's semantics through data dependence and control dependence edges. Our approach models the relations between the program statements in PDGs using Graph Neural Networks (GNNs). We have used attention-based GNNs and LSTMs 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 are used to capture



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

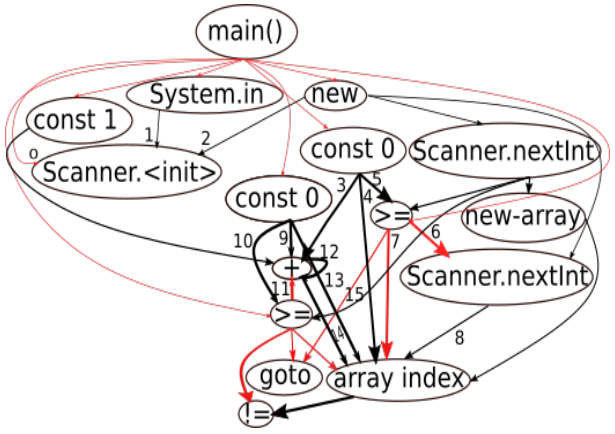


(b) t-SNE plot of input feature vectors generated by EA-HOLMES.

Fig. 15: Qualitative analysis of the features learned by EA-HOLMES.

```
1  package test;
2  import java.util.*;
3  public class Test{
4  public static void main(String[] args)
5  {
6      Scanner input = new Scanner(System.in);
7      int T = input.nextInt();
8      int[] a = new int[T];
9      for (int j = 0; j < T; j ++)
10         a[j] = input.nextInt();
11     int c = 0;
12     for (int j = 0; j < T; j ++)
13         if (a[j] == j+1)
14             c ++;
15 }
```

(a) Java Source Code.



(b) Attention Encoded Program Dependence Graph. Control dependence edges are colored red, whereas data dependent edges are colored black.

Fig. 16: Qualitative assessment of the learned PDG paths.

wider context and long-range dependencies between nodes of the PDG.

Thus representing source code as graphs and modeling them through GNNs and LSTMs helps 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 learned two different representations corresponding to each edge type. This information helps us to differentiate and prioritize between the available semantic and syntactic relations between different program statements.

7.2 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. The importance of attention has also been demonstrated in Figures 16b and 17. Using GGNN, another recurrent graph feature learning layer, can also be problematic as it uses Gated Recurrent Unit (GRU) and Backpropagation Through Time (BPTT), which can be problematic for large graphs and may require large memory.

7.3 Qualitative analysis of the features learned by HOLMES

Figure 14 shows the t-SNE [73] plot of the final hidden layer of HOLMES trained on the GCJ dataset. The figure shows that the learned features can effectively differentiate between the clone and non-clone classes. Since we have only achieved the F1-Score of 92% on GCJ dataset, we can see some overlap between the clone and non-clone classes in the figure.

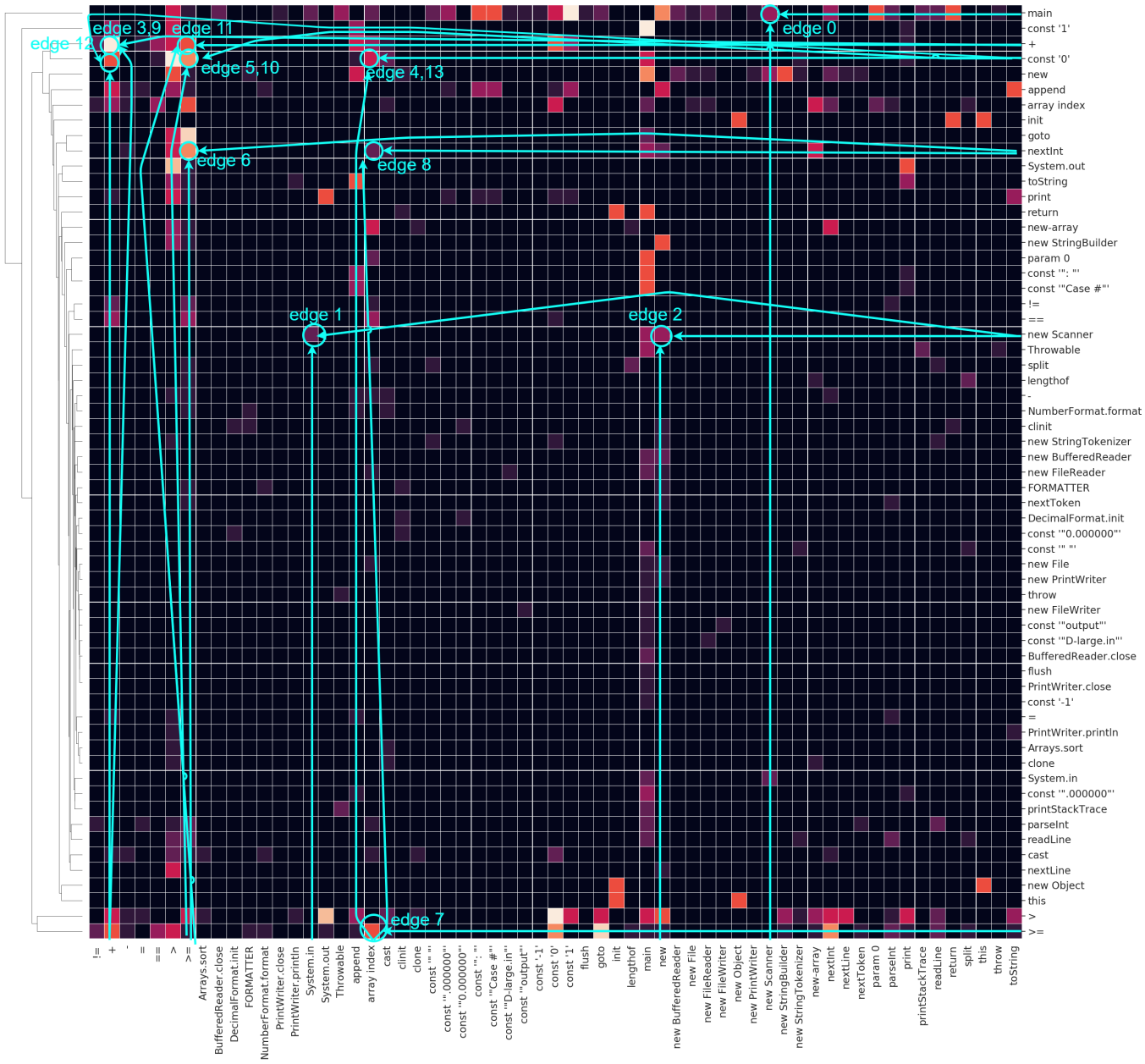


Fig. 17: Cluster map of the attention scores received by code snippets similar to Listing 16a. The annotations in Figure denote the corresponding edges of Figure 16b. [Best viewed in color.]

To get more insights into the learned feature space of HOLMES, we processed and extracted the node features and adjacency matrix from the PDG shown in Figure 15a. We then passed this to the first hidden layer of HOLMES. Figure 15b shows the t-SNE visualization of the generated vector embedding.

It can be seen from the figure 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.

7.4 Qualitative assessment of the learned PDG paths

To gain further insights into our model's working, we plotted the aggregate of multi-head self-attention scores on the PDG paths. Figure 16b shows the plot of attention scores received by the PDG of Listing 16a. Here in Figure 16b, the edge thickness denotes the attention score received by each path. From the Figure, it can be seen that the model assigns higher weights to the semantic paths.

For instance, in Listing 16a, statement 6 initializes the `Scanner` class's object from the `java.util` package. The

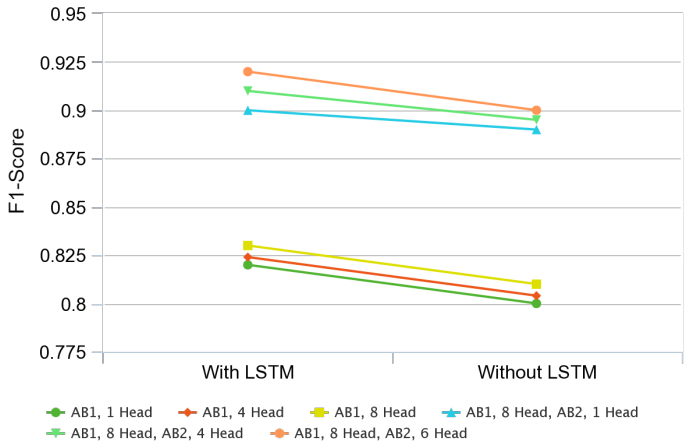


Fig. 18: The effect on the performance of HOLMES after varying the number of attention heads in both the attention blocks and after removing the LSTM layer. AB in the legend stands for Attention Block, for instance, "AB1, 1 Head" corresponds to "Attention Block 1 and 1 attention head".

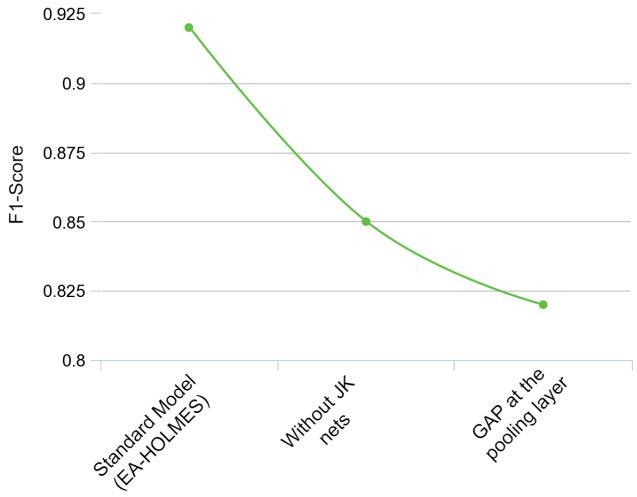


Fig. 19: The effect on the performance of HOLMES after removing Jumping Knowledge (JK) nets and soft attention mechanism from the graph readout layer.

Scanner class is used to read input in a java program. The attention scores received by statement 6 are shown by edges $0-2$ in Figure 16b. In the same code, the attention scores received by statement $9-10$ and $12-14$ are shown through edges $3-8$ and $9-15$, respectively. Statement $9-10$ from Listing 16a initializes an array of size T with random integers, and statement $12-14$ increments variable c based on some condition. From Figure 16b, it can also be seen that statement 6, being the general object creation statement of `java.util.Scanner` class, receives less attention as compared to statements $9-10$ and $12-14$. Even the statement 10 that fills the array with some random runtime integers received less attention, as shown through edge 8. This suggests that HOLMES learns to attend important semantically meaningful paths that contain the actual application logic.

Further, to strengthen the above claims, we randomly selected five Java programs similar to Listing 16a and computed attention scores received by each PDGs of these programs. We then aggregated the attention scores received by these snippets across similar paths. Figure 17 shows the cluster map [74] of the attention scores received by these Java programs.

A cluster map is a clustered version of a heatmap that uses hierarchical clustering to order data by similarity. In Figure 17, we have used cosine similarity across the rows to group similar statements together. The brighter color in the Figure represents higher attention and vice versa. From Figure 17, we can see that the attention received across all five snippets is consistent with the attention scores received in Figure 16b. For instance, the attention scores received on edges $0-2$ (statement initializing `Scanner` class object) is less than the attention scores received on edges $3-8$ (array initialization statement) as shown in Figure 16b and this can also be verified in Figure 17 (the edges are annotated in the Figure). Similarly, the attention scores received on edges $9-13$ in Figure 16b are similar to the attention score received by the five random java snippets' attention score, as shown in Figure 17.

Besides this, the cluster map in Figure 17 also justifies our claims made in Section 7.3. As we can see, relational operators such as $>$, $>=$, !=, and == are clubbed together across rows of Figure 17. File handling methods like the buffered reader, file reader, are also clubbed together. FileWriter is clubbed with PrintWriter and new File statements. Thus we can say that HOLMES learns to attend semantically meaningful paths along with modeling graph topology and node distributions.

7.5 Ablation Study

To understand the contribution of each component of our model, we conducted an ablation study, and the results are shown in Figures 18 and 19. 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 18 shows the results of varying attention heads. From Figure 18, we can see that the HOLMES performance degrades on removing the second attention block. The F1-score also degrades further when we reduce the number of attention heads.

Next, to examine the influence of LSTMS on the model's performance, we removed the LSTM layer from the HOLMES architecture and varied the attention heads of both the attention blocks. The results are shown in the right part of Figure 18. We can see that after removing the LSTM layer from the HOLMES architecture, the performance degrades. This shows the importance of using LSTMS in aggregating the local neighborhood for learning node representation.

We removed the Jumping Knowledge (JK) networks from the HOLMES architecture in the next set of experiments. The results in Figure 19 show that the F1-score reduces drastically after removing JK nets. Thus, it can be said that the JK nets help in the model's stability and improve performance. In the end, we replaced 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 19, we can observe

that HOLMES performance degrades when a GAP layer is employed in place of the soft attention mechanism. This shows the importance of the soft attention mechanism at the graph readout layer.

7.6 Limitations of HOLMES

We have used PDG representation to learn the 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 have used JCoffee [68] to infer the missing dependencies in the BCB dataset. In addition, we have used 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 our model with techniques such as domain adaptation and transfer learning so that it can be applied to other unseen and unlabeled datasets.

8 THREATS TO VALIDITY

8.1 Implementing baselines on our datasets.

We have used the available implementation of TBCCD [15]. There are various options available to tune the hyper-parameters of TBCCD, such as varying batch size, learning rate, etc. Each possible option tuning of TBCCD might have produced different results. To mitigate this, we have selected the default settings (the best parameters for TBCCD) reported by Yu et al. in [15].

8.2 Generalizing results in other programming languages.

In this paper, we have implemented the proposed approach for the Java language. While the PDG generation part is implemented in Java, all other subsequent steps are language agnostic. Attention, graph-based neural networks have been used in different contexts and for other languages as well. Also, the PDG can be generated for code written in other languages; for instance, LLVMs can generate PDGs for C/C++ code snippets. Therefore, HOLMES can potentially be adapted to work for code written in other programming languages. However, since we have not tested this, we can not make a sound claim regarding its efficacy.

8.3 Evaluating HOLMES on open source projects and programming competition datasets.

We conducted experiments on two widely used datasets for code clone detection in this work - GoogleCodeJam and BigCloneBench. We have also tried to use a large and representative dataset for our experiments. Unlike the past work [16], which has used 12 different functionality in their evaluation, we have used 100 different functionalities from GoogleCodeJam. However, HOLMES performance might vary across other projects, as these benchmarks are not representative of all software systems. To mitigate this threat and assess HOLMES generalizability, we have also performed some cross dataset experiments on SeSaMe and a GoogleCodeJam dataset variant.

9 RELATED WORK

This section describes the related work on code clone detection techniques and learning program representations using learning-based techniques.

9.1 Code Clone Detection

9.1.1 Traditional code clone detection approaches.

Most traditional code clone detection techniques target Type 1-3 code clones. These techniques measures code similarity by using program representation such as abstract syntax trees [75], lexical tokens [76], [77], program text [78], [79]. Deckard [75], a popular tree-based code clone detection technique, computes characteristic vector for AST nodes of the given program. It then applies Locality Sensitive Hashing (LSH) to find similar code pairs. SourcererCC [69] is a token-based code clone detection technique that compares token subsequences to identify program similarity.

There are also some graph-based techniques [80], [81] that use program dependence graphs to identify Type-4 clones. PDG-DUP [80] first converts the given program to PDGs and then uses program slicing and subgraph isomorphism to identify clone pairs. DUPLIX [81] also uses program slicing and graph isomorphism to identify similar code pairs. However, these approaches do not scale to large codebases and are very time-consuming, limiting their applications in practical software systems. In addition to these, some techniques also exist that compare program runtime behavior [82] or program memory states [13] to identify code clones.

9.1.2 Learning based code clone detection approaches.

Learning from data to identify code clones has been a great deal of interest from the past. There have been techniques using data mining approaches to learn code similarity [83], [84]. For example, Marcus and Maletic [83] has proposed to use latent semantic indexing to detect semantic clones. The proposed approach examines the source code text (comments and identifiers) and identifies the implementation of similar high-level concepts such as abstract data types. Much recent work uses learning-based techniques to learn code similarity. These approaches try to learn continuous vector-based representations of code fragments. These vectors are then compared using some distance metric (e.g., Euclidean distance) or using neural networks to measure code functional similarity. White et al. [3] used a recursive neural network to learn program representation. They represented source code as a stream of identifier and literals and used it as an input to their deep learning model. Tufano et al. [21] using a similar encoding approach as [3] encoded four different program representations- identifiers, Abstract Syntax Trees, Control Flow Graphs, and Bytecode. They then used a deep learning model to measure code similarity based on their multiple learned representations. Wei et al. [18] uses AST and tree-based LSTM to learn program representation. To incorporate structural information available with the source code Yu et al. [15] uses tree-based convolutions over ASTs to learn program representation. Saini et al. [85] proposes using software metrics and machine learning to detect clones in the twilight zone. Zhao et al. [16] used feature vectors extracted from the data

flow graph of a program to learn program representation using deep neural networks. Mathew et al. [86] proposed a cross-language clone detection by comparing the input and output of the potential clone candidates.Additionally, there also exists techniques to detect clones in binaries [87], [88], [89], [90]. Li et al. [88] proposed a Graph Matching Network(GMN) to address the problem of matching and retrieval of graph structured data. They have proposed a new cross-graph attention-based matching mechanism to compute similarity score for a given pair of graph. The proposed graph matching network model is shown to outperform the graph embedding models on binary function similarity search. Xu et al. [87] proposed a technique to detect cross-platform clones in binaries. The proposed tool Gemini uses Structure2vec [91] neural network model to learn the hidden binary function features from control flow graphs. The learned features are then compared using cosine distance to measure code similarity.

9.2 Representation Learning for Source Code

There has been a significant interest in utilizing deep learning models to learn program embeddings. The goal is to learn precise representations of source code for solving various software engineering tasks. Gupta et al. [92] propose to fix common programming errors using a multilayered sequence to sequence neural networks with attention. The deep learning model is trained to predict the erroneous program locations in a C program and the required correct statements. Allamanis et al. [93] use graph-based neural networks over AST based program representation to learn program embeddings. The learned embeddings are then used to predict the names of variables and varmisue bugs. Wang et al. [94] use program execution traces to learn program embeddings to predict method names. Ben-Nun et al. [95] use Intermediate Representation (IR) of source code with recurrent neural networks to learn program embeddings. Hoang et al. [96] propose a neural network model CC2Vec to learn a representation of source code changes. CC2Vec uses attention to model the hierarchical structure of source code. The learned vectors represent the semantic intent of code change. The authors have evaluated the proposed approach on three downstream tasks: log message generation, bug fixing patch identification, and just-in-time defect prediction. There has also been some work on assessing the quality of learned representations. Kang et al. [97] present an empirical study to assess the generalizability of Code2vec token embeddings. The authors have evaluated the Code2vec token embeddings on three downstream tasks: code comments generation, code authorship identification, and code clone detection. Their results show that the learned representation by the Code2vec model is not generalized and cannot be used readily for the downstream tasks.

10 Conclusion and Future Work

There has been a significant interest in detecting duplicated code fragments due to its 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 paper, 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 recent state-of-the-art clone detection tool TBCCD [15]. 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 explore the feasibility of the proposed approach in cross-language clone detection.

Acknowledgments

This work is supported in part by the Department of Science and Technology (DST) (India), Science and Engineering Research Board (SERB), the Confederation of Indian Industry (CII), Infosys Center for Artificial Intelligence at IIIT-Delhi, and Nucleus Software Exports Ltd.

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] 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.

[4] 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.

[5] 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.

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," in *OSDI*, 2004.

[7] 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.

[8] 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.

[9] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *2006 13th Working Conference on Reverse Engineering*, 2006, pp. 19–28.

[10] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 664–675.

[11] 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. New York, NY, USA: Association for Computing Machinery, 2011, p. 75–76.

[12] 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. New York, NY, USA: Association for Computing Machinery, 2007, p. 55–64.

[13] 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.

[14] 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)*, May 2007, pp. 96–105.

[15] 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.

[16] 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.

[17] 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.

[18] ——, "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.

[19] 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.

[20] 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)*, 2016, pp. 87–98.

[21] 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.

[22] 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.

[23] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 301–309.

[24] 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. New York, NY, USA: Association for Computing Machinery, 2008, p. 321–330.

[25] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," 10 2011.

[26] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[27] D. Binkley, "Source code analysis: A road map," in *IN FUTURE OF SOFTWARE ENGINEERING*, 2007, pp. 104–119.

[28] 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.

[29] 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. New York, NY, USA: Association for Computing Machinery, 1990, p. 234–245.

[30] 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. New York, NY, USA: Association for Computing Machinery, 1989, p. 168–178.

[31] G. E. Hinton, "Connectionist learning procedures," *Artif. Intell.*, vol. 40, no. 1–3, p. 185–234, Sep. 1989.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997.

[33] Y. Bengio and Y. Lecun, "Convolutional networks for images, speech, and time-series," 11 1997.

[34] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," 2016.

[35] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[36] G. Lample, M. Ott, A. Conneau, L. Denoyer, and M. Ranzato, "Phrase-based & neural unsupervised machine translation," *CoRR*, vol. abs/1804.07755, 2018.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[38] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," *CoRR*, vol. abs/1709.01507, 2017.

[39] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, "Graph convolutional networks: a comprehensive review," *Computational Social Networks*, vol. 6, 12 2019.

[40] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019.

[41] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," 2020.

[42] X. Wang, T. Zhang, R. Wu, W. Xin, and C. Hou, "CPGVA: code property graph based vulnerability analysis by deep learning," in *10th International Conference on Advanced Infocomm Technology, ICAIT 2018, Stockholm, Sweden, August 12-15, 2018*. IEEE, 2018, pp. 184–188.

[43] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 25–36.

[44] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *CoRR*, vol. abs/1611.08097, 2016.

[45] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral Networks and Locally Connected Networks on Graphs," *arXiv e-prints*, p. arXiv:1312.6203, Dec. 2013.

[46] A. Micheli, "Neural network for graphs: A contextual constructive approach," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 498–511, 2009.

[47] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, ser. NIPS'93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, p. 737–744.

[48] P. Baldi and Y. Chauvin, "Neural networks for fingerprint recognition," *Neural Comput.*, vol. 5, no. 3, p. 402–418, May 1993.

[49] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *CoRR*, vol. abs/1808.01400, 2018.

[50] 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.

[51] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017.

[52] 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.

[53] 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.

[54] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," *CoRR*, vol. abs/1711.07163, 2017.

[55] 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, Oct. 2019.

[56] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016.

[57] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2017.

[58] Z. Liu, C. Chen, L. Li, J. Zhou, X. Li, and L. Song, "Geniepath: Graph neural networks with adaptive receptive paths," *CoRR*, vol. abs/1802.00910, 2018.

[59] K. Xu, C. Li, Y. Tian, T. Sonobe, K. ichi Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," 2018.

[60] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015.

[61] 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, Oct. 1991.

[62] D. Grunwald and H. Srinivasan, "Data flow equations for explicitly parallel programs," *SIGPLAN Not.*, vol. 28, no. 7, p. 159–168, Jul. 1993.

[63] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, no. 3, p. 137, Mar. 1976.

[64] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[65] 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.

[66] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015.

[67] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[68] P. Gupta, N. Mehrotra, and R. Purandare, "Jcoffee: Using compiler feedback to make partial code snippets compilable," 2020.

[69] 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.

[70] 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. IEEE Press, 2019, p. 529–533.

[71] 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. 2014, pp. 476–480.

[72] 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.

[73] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," 2008.

[74] M. Waskom, O. Botvinnik, D. O'Kane, P. Hobson, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. B. Cole, J. Warmenhoven, J. de Ruiter, C. Pye, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, P. Bachant, M. Martin, K. Meyer, A. Miles, Y. Ram, T. Yarkoni, M. L. Williams, C. Evans, C. Fitzgerald, Brian, C. Fonnesbeck, A. Lee, and A. Qalieh, "mwaskom/seaborn: v0.8.1 (september 2017)," Sep. 2017.

[75] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," 01 2004, pp. 289–302.

[76] 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.

[77] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," 01 2004, pp. 289–302.

[78] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, 1992.

[79] J. Cordy and C. Roy, "The nicad clone detector," 06 2011, pp. 219–220.

[80] 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.

[81] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 301–309.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] G. Mathew, C. Parnin, and K. T. Stolee, "Slacc: Simion-based language agnostic code clones," *ArXiv*, vol. abs/2002.03039, 2020.

[87] 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.

[88] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," *CoRR*, vol. abs/1904.12787, 2019.

[89] 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.

[90] 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.

[91] L. Song, "Structure2vec: Deep learning for security analytics over graphs." Atlanta, GA: USENIX Association, May 2018.

[92] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *AAAI*, 2017.

[93] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2017.

[94] K. Wang, "Learning scalable and precise representation of program semantics," 2019.

[95] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," 2018.

[96] T. Hoang, H. J. Kang, J. Lawall, and D. Lo, "Cc2vec: Distributed representations of code changes," 2020.

[97] 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.

Nikita Mehrotra is a Ph.D. student in Computer Science and Engineering at the Indraprastha Institute of Information Technology Delhi (IIIT-Delhi), India. Her research is supported by the Prime Minister's Research Fellowship with industrial support from Nucleus Software Exports. Her research interests include Deep learning applied software engineering, program understanding, program analysis, software evolution and maintenance.

Navdha Agarwal is an undergrad student pursuing Computer Science and Applied Mathematics at the Indraprastha Institute of Information Technology Delhi (IIIT-Delhi), India. Her research interests include program analysis and software maintenance.

Piyush Gupta is an undergrad student pursuing Computer Science Engineering at the Indraprastha Institute of Information Technology Delhi (IIIT-Delhi), India. His research interests include program analysis and software verification.

Saket Anand is an Assistant Professor at the Indraprastha Institute of Information Technology Delhi (IIIT-Delhi). He received his Ph.D. in Electrical and Computer Engineering from Rutgers University, New Jersey, USA. His research interests include computer vision, machine learning and deep learning. He is a member of the IEEE.

David Lo is an Associate Professor in the School of Information Systems, Singapore Management University (SMU). He received his Ph.D. in Computer Science from the National University of Singapore. His research interests include software analytics, software maintenance, empirical software engineering, and cybersecurity.

Rahul Purandare is an Associate Professor in the department of Computer Science and Engineering at the Indraprastha Institute of Information Technology Delhi (IIIT-Delhi). He received his Ph.D. in Computer Science from the University of Nebraska - Lincoln. His research interests include program analysis, software testing, automatic program repair, code search, and code comprehension.