The Journal of Systems & Software 184 (2022) 111108

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Precise Learning of Source Code Contextual Semantics via Hierarchical Dependence Structure and Graph Attention Networks ☆

Zhehao Zhao [a], Bo Yang [b,*], Ge Li [a], Huai Liu [c], Zhi Jin [a,*]

[a] Key Laboratory of High Confidence Software Technologies, Peking University, Beijing 100871, China
[b] School of Information Science and Technology, Beijing Forestry University, Beijing 100083, China
[c] Department of Computing Technologies, Swinburne University of Technology, Hawthorn VIC 3122, Australia

ARTICLE INFO

Article history:
Received 8 March 2021
Received in revised form 6 September 2021
Accepted 25 September 2021
Available online 19 October 2021

Keywords:
Graph neural network
Program analysis
Deep learning
Abstract syntax Tree
Control flow graph

ABSTRACT

Deep learning is being used extensively in a variety of software engineering tasks, e.g., program classification and defect prediction. Although the technique eliminates the required process of feature engineering, the construction of source code model significantly affects the performance on those tasks. Most recent works was mainly focused on complementing AST-based source code models by introducing contextual dependencies extracted from CFG. However, all of them pay little attention to the representation of basic blocks, which are the basis of contextual dependencies.

In this paper, we integrated AST and CFG and proposed a novel source code model embedded with hierarchical dependencies. Based on that, we also designed a neural network that depends on the graph attention mechanism. Specifically, we introduced the syntactic structural of the basic block, i.e., its corresponding AST, in source code model to provide sufficient information and fill the gap. We have evaluated this model on three practical software engineering tasks and compared it with other state-of-the-art methods. The results show that our model can significantly improve the performance. For example, compared to the best performing baseline, our model reduces the scale of parameters by 50% and achieves 4% improvement on accuracy on program classification task.

1. Introduction

Recently, deep learning has been increasingly applied into program analysis tasks, such as program classification (Wang and Su, 2019; Ott et al., 2018; Frantzeskou et al., 2008), software defect prediction (Wang et al., 2016; Tantithamthavorn et al., 2016), and code summarization (Hu et al., 2018; Yao et al., 2019; LeClair et al., 2019). However, the performance on these tasks heavily depends on the choice of source code model, which can be divided into three types: abstract syntax tree- (AST-) based, control flow graph- (CFG-) based and the hybrid model of these two. Moreover, depending on the structure of AST adopted during analysis, AST-based source code model can be further divided to the whole AST (Mou et al., 2016; White et al., 2016; Dam et al., 2019) or partial AST (Zhang et al., 2019; Alon et al., 2019, 2018a). The syntactic structure within AST can illustrate all the information of source code, especially the subtle changes on it. However, the contextual dependencies are implicit in AST and cannot be extracted and learnt effectively. In contrast, the CFG-based source code model (Phan et al., 2018; Tufano et al., 2018) is good at providing contextual dependencies, which can be learnt effectively by graph neural networks. Nevertheless, CFG is uneffective to represent the information of statements located in the basic blocks. Therefore, some researches proposed methodologies to embed the contextual dependencies from CFG into AST (Allamanis et al., 2017; Li et al., 2019; Alon et al., 2018b). Such a design idea of the hybrid method still take AST as the core part of the source code model. It would add the contextual dependencies as additional edges (Allamanis et al., 2017) to AST or as assistant features (Li et al., 2019). However, the basic blocks, which are the basis of contextual dependencies, are paid little attention by the existing methodologies. To mine the contextual dependencies effectively, we argue that the features of basic blocks should be prioritized. Fig. 1 shows our motivational example. These two code segments come from the PROMISE dataset used in our study. The defect in Fig. 1(a) is that returning a *null* value on line 7 will cause a *NullPointerException*, and the corresponding fix is to return a *Field* type array of length 0 here. After analyzing this example, we have the following observations.

Observation 1: This defect depends on the actual execution path. As shown in Fig. 1(a), the defect is triggered only if the condition on line 6 is met. However, if the caller of the *getFields*

☆ Editor: Raffaela Mirandola.

* Corresponding authors.
E-mail addresses: zhaozhehao@pku.edu.cn (Z. Zhao), yangbo@bjfu.edu.cn (B. Yang), lige@pku.edu.cn (G. Li), hliu@swin.edu.au (H. Liu), zhijin@pku.edu.cn (Z. Jin).

https://doi.org/10.1016/j.jss.2021.111108
0164-1212/

```
1  class Document {
2
3      public final Field[] getFields(String name){
4          List result = new ArrayList();
5          ...
6          if(result.size() == 0)
7              return null;
8          ...
9      }
10 }
```

(a) The defect version of *lucene-2.2*

```
1  class Document {
2      private final static Field[] NO_FIELDS = new Field[0];
3      public final Field[] getFields(String name) {
4          List result = new ArrayList();
5          ...
6          if(result.size() == 0)
7              return NO_FIELDS;
8          ...
9      }
10 }
```

(b) The fixed version of *lucene-2.4*

Fig. 1. A motivating example from PROMISE dataset. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

function properly handles caught exceptions, this defect will not be triggered. Thus, a reasonable source code model should reflect the execution path. Furthermore, since a large number of invocations to *getFileds* are outside from the *Document* class, the source code model should not be limited to a certain granularity.

Observation 2: These two source codes differ slightly but with total different semantics. As shown in Fig. 1, the difference of these two source codes is a choice between returning an identifier *NO_FIELDS* or a *null* in line 7. The code in Fig. 1(b) does not cause the exception because that *NO_FIELDS* refers to an *object* (see line 2 of Fig. 1(b)). Thus, the difference of these two source code is actually the difference between *object* and *null*. Moreover, for the deep learning models with some textual features (e.g., Bag of Words), the learning of these two words (*null* and *NO_FIELDS*) is uneven, since *null* occurs more frequently than *NO_FIELDS*, which would raise the difficulty for models to learn the real difference.

According to the **Observation 1**, CFG would intuitively become the first choice of source code model, since CFG can show the potential execution path and can be constructed on any granularity. But there still exists the issue about how to represent the basic blocks within the CFG. In the existing CFG-based works, basic blocks are mainly represented by either line numbers (Li et al., 2019) or Bag of Words (Zhong and Mei, 2019; Wang et al., 2020). However, according to the **Observation 2**, these methods only utilize the textual features, which significantly relies on the frequency of occurrence. Thus they cannot effectively capture the difference shown in Fig. 1 to distinguish *NO_FIELDS* and *null*. We argue that a proper source code model should introduce semantic differences (e.g., the difference between *object* and *null*) into the deep learning models more than the textual distinctions.

Motivated by these observations, we propose a novel source code model. Specifically, to overcome the limitation mentioned in the **Observations 1**, we choose CFG with dataflow (ECFG), which can reflect the actual execution paths, as the backbone of the source code model. To address the **Observations 2**, we use the block-level AST, i.e., each AST subtrees correspond to each ECFG basic blocks. Take the source codes in Fig. 1 to illustrate the benefit of such way: since *NO_FIELDS* represents an *object* while *null* is just a keyword, the syntax rule for them are not same, which brings different AST structures. To sum up, the whole model can be divided into two levels. At the outer level, we use the inter-procedure ECFG to express the dependencies between the basic blocks. At the inner level, we choose AST to express the structure of each basic block.

Our source code model has three advantages. First, benefiting from the ECFG as the main body, the granularity of our source code model can be flexibly adjusted. Second, also benefiting from the ECFG, our source code model can show the potential execution path explicitly, thus the contextual dependencies can be captured effectively by a graph neural network. Third, benefiting from the substructure of AST, our source code model can have a more informative representation of basic blocks, hence the features within each basic blocks can be captured effectively by a tree-based neural network.

Furthermore, we designed a *Multi-Flow Graph Neural Network* (MFGNN) to extract features from our source code model. The calculation of MFGNN can be divided into three steps. At the first step, we obtain features named *local features* through TBCNN (Mou et al., 2016) from the collection of AST-substructures, which are corresponded to the basic blocks in ECFG. At the second step, we extract features named *contextual features* from ECFG, whose basic blocks has been filled with *local features*. Since the ECFG is a directed graph with multi-typed edges where we want to adopt attention mechanism, we did a slightly modification on the original Graph Attention Network (GAT). Specifically, the modified model supports directed graph and multi-typed edges, we name it as *Attention-based Graph Network for Directed Graph* (AGN4D), and apply it in the second step. At the third step, we apply a fusion layer to coalesce these features into hybrid features, which can be used for subsequent tasks.

To be specific, this paper has the following three major contributions:

- We propose a source code model that combines AST and CFG with dataflow (ECFG). The source code model can reflect both contextual dependencies and syntactic structure, which allows neural networks to learn richer program features.
- We design a learning model to obtain contextual semantics from the source code model, namely Multi-Flow Graph Neural Network (MFGNN). MFGNN integrates an attention-based graph learning layer evolves from GAT.
- MFGNN is implemented and evaluated on three typical tasks, namely the program classification, software defect prediction and code clone detection. The results show that MFGNN can extract richer program features than the state-of-the-art methods, and hence greatly improve the performance of these tasks.

The remainder of this paper is organized as follows: Section 2 introduces the background of our work. Section 3 describes the new source code model and MFGNN. We report our experimental studies and results in Sections 4 and 5, respectively. The related work is discussed in Section 6. Finally, we conclude this paper in Section 7.

2. Background

In this section, we would introduce some basic concepts and terms that are used in this paper.

2.1. Program representation

To represent a piece of program, there are several ways: token sequences, AST, CFG (Tufano et al., 2018). Among all of them, AST and CFG are adopted most widely, thus we would introduce both of them in this section.

2.1.1. Abstract Syntax Tree

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language (Mou et al., 2016). Each node on the AST represents a nonterminal symbol in the syntax rules of the programming language. Being a near-source-level program graph structure, AST can represent the syntactic information of programs in a simple way, which makes AST widely used in a variety of software engineering tasks (Mou et al., 2016; Allamanis et al., 2017; Dam et al., 2019; Zhang et al., 2019; Wang et al., 2016; Alon et al., 2019, 2018a).

2.1.2. Control Flow Graph

Control Flow Graph (CFG) is a directed graph in which each node (namely basic block) represents a set of sequentially executed instruction sequences, and the edges represent control flow paths. CFG is mostly used in static analysis and compiler applications, as it can accurately represent the flow inside a program. For example, through graph reachability analysis, CFG can help locate inaccessible code in programs, and find syntax structures such as loops. As a source code model for deep learning, CFG's edges are usually considered to represent the contextual dependencies, which have a significant impact on the performance of software engineering tasks (Li et al., 2019; Fang et al., 2020; Allamanis et al., 2017).

2.2. Graph neural networks

Graph is a generic data structure to effectively abstract objects and their connections (Zhou et al., 2018). It has been widely used across multiple domains, such as social networks (Hamilton et al., 2017), chemical interaction (Fout et al., 2017) and knowledge modeling (Hamaguchi et al., 2017).

Graph Neural Networks (GNNs) are methods used to mine the information within a graph and obtain the embedding vector of the graph under a learning model. GNNs are mostly based on the message-passing mechanism, and consist of two functions: the Message function and the Aggregate function (Zhou et al., 2018). The Message function is used to transform the original vector of nodes to obtain the hidden vector; and the Aggregate function is used to aggregate the transformed vectors of a node's adjacency nodes and obtain an embedding vector of the node.

The Message function is generally represented using a parameter $W \in \mathbb{R}^{F*F'}$. Let $X = \{x_1, x_2, \ldots, x_n\}, x_i \in \mathbb{R}^F$ be the initial features of nodes, and $H = \{h_1, h_2, \ldots, h_n\}, h_i \in \mathbb{R}^{F'}$ be the transformed features of nodes. Then, the Message function can be defined as:

$$h_i = Message(x_i) = Wx_i,$$

where F represents the initial dimension of nodes' features, and F' represents the transformed dimension of nodes' features.

Different GNNs often vary in the Aggregate functions. For example, GCN (Bruna et al., 2014) uses summation as the Aggregate function, which is defined as follows.

$$h'_i = Aggregate(h, \mathcal{N}_i) = \sum_i^{\mathcal{N}_i} h_i,$$

where $\mathcal{N}_i$ is the collection of adjacency nodes of i.

GAT (Veličković et al., 2017) uses the self-attention mechanism as the Aggregate function. GAT first calculates *self-attention* weights for all edges in the graph, as defined below:

$$\alpha_{ij} = \frac{exp(LeakyReLU(a(Wh_i \parallel Wh_j)))}{\Sigma_{k\in\mathcal{N}_i} exp(LeakyReLU(a(Wh_i \parallel Wh_k)))},$$

where $\parallel$ is the concatenation operation and $a : \mathbb{R}^{\mathbb{F}'} \times \mathbb{R}^{\mathbb{F}'} \rightarrow \mathbb{R}$ is the shared attention mechanism.

GAT then linearly combines the transformed features of the neighboring nodes according to the attention weights, which is defined as:

$$h'_i = \sigma(\sum_j^{\mathcal{N}_i} \alpha_{ij} h_j)$$

where σ is a nonlinearity function.

3. Approach

In this section, we would introduce our source code model and the learning model named MFGNN.

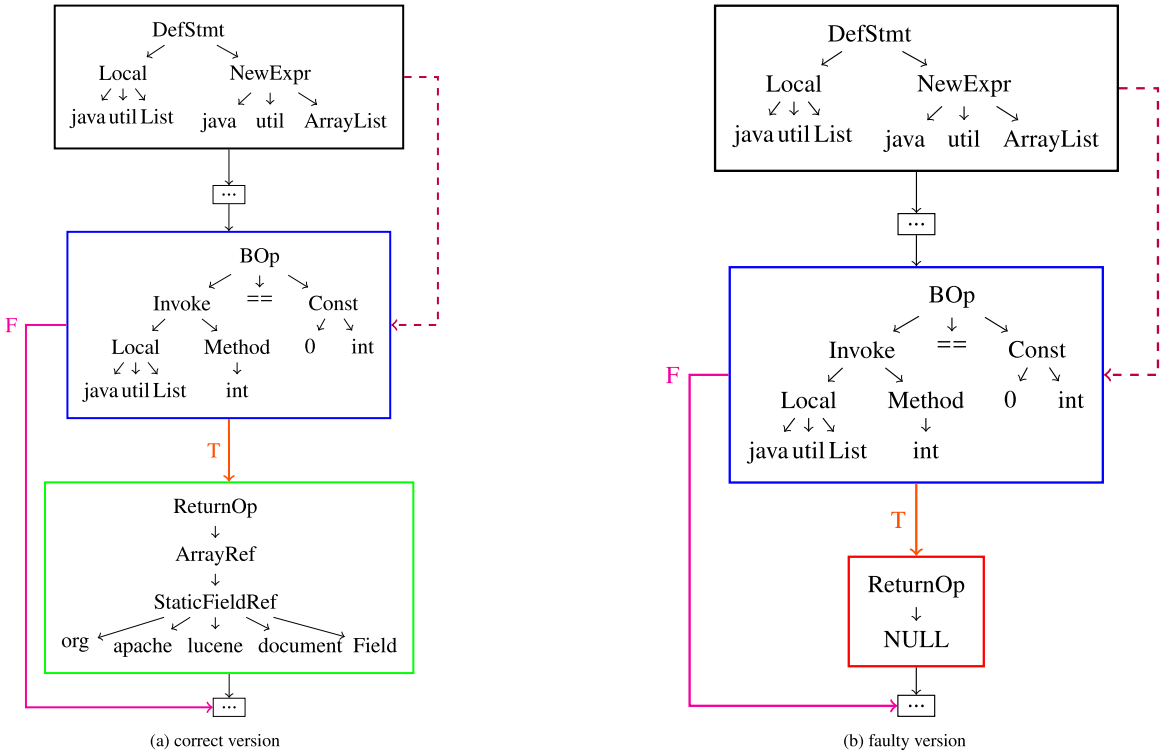
3.1. Constructing graph through combining ECFG and AST

The combination of ECFG and AST can be considered as a type of program dependency graph. The backbone of the graph is an inter-procedural CFG. A CFG $\mathcal{G} = (B, E)$ of the program is a directed graph, where B is a collection of basic blocks and E contains all control flow relationships. We choose three-address code (e.g., LLVM IR for C/C++ and Jimple for Java) as the intermediate representation when generating CFG by analysis frameworks (e.g., Clang for C/C++ and Soot for Java).

From the motivating example (see Section 1), we can conclude that a precise modeling for basic block is essential for the following analysis. Therefore, we choose AST to model basic blocks as its nature of expressing syntactic structures and code information. To further enrich the information in the AST, we introduce the data types of variables to the subtrees of AST, which inherently lacks of such information and corresponds to variable usage (e.g., `DeclRefExpr` node in Clang). Specifically, for the basic data type, we directly add it as a leaf node of the variable node. For user-defined classes, we refer to the method mentioned in Cvitkovic et al. (2018), and separate these classes according to the Camel-Case naming. For type conversion statements, we process both the original type and the target type according to the above method and then add them into the AST as a subtree of the type conversion node. Another aspect we need to consider is the constants. To handle different constants, we disassemble the constant value bitwise (e.g., The constant nodes 456 will be disassembled to three nodes, represent 4, 5, 6,respectively.).

To address the lack of dataflow dependency in both AST and CFG, in addition to the *control flow*, *call flow*, and *exception flow* relationships included in the CFG, we also introduce *data flow* relationship into our graph. Specifically, *dataflow* relationship comes from the intra-procedural dataflow analysis. By traversing the CFG, we have built two collections of variables: *define* stores variables defined in the block and *use* stores variables used in the block. The dataflow relationship between basic blocks is obtained by reaching definition analysis later. Then we divide the edges of *control flow* into four categories according to their functionalities: *sequential execution*, *conditional true branch*, *conditional false branch* and *switch branches*. Different categories of flow relationship are labeled distinctly. Overall, there are seven types of edges in our source code model.

The final graph of the motivating example is shown in Fig. 2. We can observe that the red and green blocks of the two snippets respectively containing different AST, representing the great differences in *local features* between blocks. For the correct version (Fig. 1(b)), the AST of the green block indicates that a static field of that class is returned. For the faulty version (Fig. 1(a)), the AST of the red block indicates that a null value is returned. This slight textual difference, *NO_FIELDS* vs. *null*, can be easily learnt with the help of a tree-based neural network due to the significant difference in the AST. The control flow and the dataflow edges



(a) correct version (b) faulty version

Fig. 2. The comparison of the motivating example using the combination of CFG and AST. Black edges for sequential execute, orange edges for conditional true branch and fuchsia edges for the false branch. The dashed-purple edges are dataflow edges.. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

(i.e., dashed-purple edges) jointly describe the use of variables, and the conditional true edges (i.e., orange edges) indicate the branch and precondition of the faulty block. Combined with the difference in *local features* between green and red blocks, our source code model leads to different *context features*.

3.2. Multi-Flow Graph Neural Network

We design a neural network model to obtain the features of our representation model and name it as Multi-Flow Graph Neural Network (MFGNN). Fig. 3 shows the overall structure of the model. The learning process can be divided into three stages: *local features* embedding stage, *contextual features* embedding stage and fusion stage. In the first stage, the tree-based network is used to learn the *local features* for each block in B. In the second stage, attention-based graph neural network for directed graph (AGN4D) is used to learn *contextual features* in the combined graphs based on *local features* of each block. In the final stage, a fusion layer is used to fuse *local features* and *contextual features*, and the contextual semantics are obtained.

3.2.1. Local features embedding

We use Tree-based Convolutional Neural Network (TBCNN) to obtain the *local features* in each block. The original TBCNN is not suitable for our extended AST because of the additional contents on the leaf nodes of the extended AST. The learning process of the original TBCNN ignores the fact that the deeper the node in the AST, the richer the information. Therefore, we adjust the preset weights of TBCNN to increase the weight of deeper nodes in the convolution window of TBCNN, i.e., the nodes with richer information would have a more significant impact on the training process of the model. The formulas of the weights are shown as follows:

$$\eta_i^t = \frac{d_i - 1}{d_{max} - 1}\eta_i^l = \eta_i^t \frac{p_i - 1}{n - 1}\eta_i^r = \eta_i^t(1 - \eta_i^l), \tag{1}$$

where d_i is the depth of node i in the entire tree, d_{max} is the depth of entire tree, p_i is the position of the node i in subtree, and n is the total number of i's siblings.

The importance of *local features* is two-fold: first, as the features of each block, *local features* is the input for AGN4D (see Section 3.2.2) for learning *contextual features*; second, *local features* play an critical role within the final features, so we pass *local features* to the fusion layer (see Section 3.2.3) directly.

3.2.2. Contextual features embedding

Our source code model can be considered as a directed graph with edge types. Therefore, based on GAT (see Section 2.2), we design a network layer nested in MFGNN, named *Attention-based Graph Neural Network for Directed Graph* (AGN4D), which can handle directed graph and multiple types of edges. With AGN4D, we can extract *contextual features* from the combination graph.

Suppose $\mathcal{G}$ is an instance of the combined graph and $\mathcal{RG}$ is the reverse graph of $\mathcal{G}$. Let $X = \{x_1, x_2, \ldots, x_n\}$ represent *local features* of the blocks in the set B obtained in the previous stage. Let the initial graph embedding $H^0 = \{h_1^0, h_2^0, \ldots, h_n^0\}$ where $H^0 = X$, the graph embedding update process of $\mathcal{G}$ is as follows:

$$\begin{aligned} k_{o,u}^l &= MSG_o^l(h_u^{l-1}) \quad k_{r,u}^l = MSG_r^l(h_u^{l-1}) \\ h_{o,u}^l &= Agg_o^l(k_{o,u}^l, \{k_{o,v}^l | v \in \mathcal{N}_u^{\mathcal{G}}\}) \\ h_{r,u}^l &= Agg_r^l(k_{r,u}^l, \{k_{r,v}^l | v \in \mathcal{N}_u^{\mathcal{RG}}\}) \\ h_u^l &= h_{o,u}^l + h_{r,u}^l + h_u^{l-1} \end{aligned} \tag{2}$$

, where $\mathcal{N}_u^{\mathcal{G}}$ is the collection of successors of block u in original graph $\mathcal{G}$ and $\mathcal{N}_u^{\mathcal{RG}}$ for reverse graph $\mathcal{RG}$. MSG_o^l represents the MSG function of the original graph at layer l and MSG_r^l for the reverse graph. Agg_o^l refers to the Agg function of the original graph at layer l and Agg_r^l for the reverse graph. Note that MSG function and Agg function do not share parameters between different layers.

After obtaining the graph embedding from the two graphs, the graph embedding of previous layer and current layer are

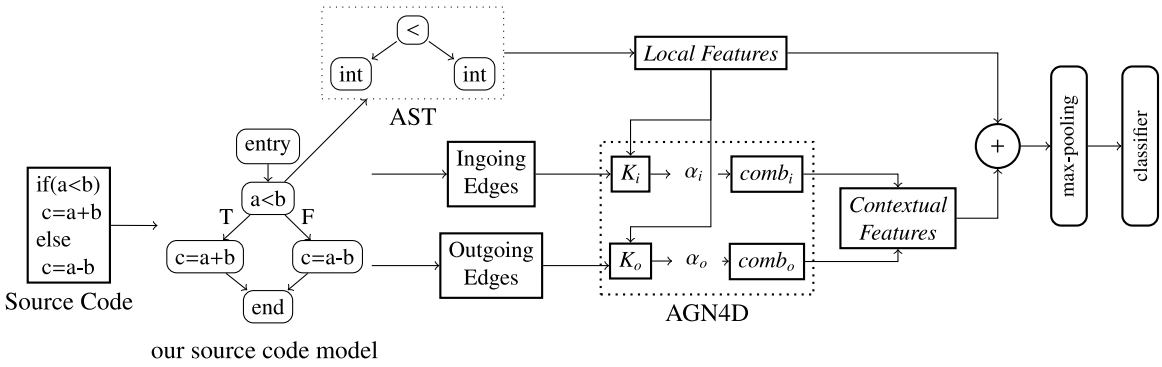


Fig. 3. MFGNN structure.

connected by a *skip-connection* to obtain the final graph representation of this layer.

The *MSG* function needs to transform the graph embedding from the previous layer to obtain the features of for this layer, which is parameterized by a weight matrix W^l_{key} which is defined as follows:

$$k^l_u = MSG^l(h^{l-1}_u) = W^l_{key} h^{l-1}_u. \tag{3}$$

The *Agg* function aggregates features in successor blocks and current block. We add support for multiple types of edges to the *self-attention* mechanism of GAT, defined as follows:

$$\begin{aligned} e^l_v &= P^l_{src} k^l_u + P^{l,f}_{dst} k^l_v \quad \langle u, v, f \rangle \in E \\ \alpha^l_v &= softmax(\{LeakyReLU(e^l_v) | v \in \mathcal{N}_u\}) \\ h'_u &= Agg^l(k^l_u, \{k^l_v | v \in \mathcal{N}_u\}) = \sigma\left(\sum_{v \in \mathcal{N}_u} \alpha^l_v k^l_v\right) \end{aligned} \tag{4}$$

, where f stands for the flow type of the edge from u to v. Attention mechanism is parameterized by P^l_{src} and $P^{l,f}_{dst}$, which indicates the importance of the f-type flow dependency between blocks u and v.

We pass the h'_u of the last layer of AGN4D to the fusion layer as *contextual features*.

3.2.3. Fusion layer

The main functionality of the fusion layer is to fuse *local features* and *contextual features* into the hybrid features of the program. In our design, the fusion layer first adds *local features* and *contextual features*, then gets the fixed size program feature vector through *dynamic pooling*. In practice, we choose max-pooling as a pooling function. Finally, we train a classifier (i.e., Logistic Regression (LR)) for classification tasks.

4. Evaluation

We conducted a series of experiments to evaluate MFGNN with comparison against some existing state-of-art methods. Our experiments run on a 4 T k40c GPUs machine with Xeon E5-2310 32 GB RAM.

4.1. Research questions

To evaluate the effectiveness of our source code model and MFGNN, and compare them with several state-of-the-art methods on some particularly tasks, our experiments were particularly designed to answer the following five research questions:

RQ1 How is the performance of MFGNN in classifying datasets that consists of programs with small textual but large semantic differences?

RQ2 How is the performance of MFGNN in Within-Project Defect Prediction (WPDP) task compared with the state-of-the-art methods?

RQ3 How is the performance of MFGNN in Cross-Project Defect Prediction (CPDP) task compared with the state-of-the-art methods?

RQ4 How is the performance of MFGNN in Functional Code-Clone Detection (CCD) task compared with the state-of-the-art methods?

RQ5 To what extent do different components in MFGNN influence the performance?

4.2. Datasets

For RQ1 and RQ5, we selected two datasets as the objects of our experiments, namely CodeChef and Codeforces. The Codechef dataset is collected by Phan et al. (2018) and composed of solutions, written in C/C++, which are submitted by users for four challenges, namely `SUB`, `MNMX`, `FLOW`, and `SUM`. However, these four challenges are trivial (e.g., `FLOW` only requires an implementation of the GCD algorithm), which cannot evaluate the effectiveness of our tool thoroughly. Thus, we further manually collected a dataset, namely Codeforces, from a public website.[1] Specifically, it consists of solutions submitted by users for five challenges, i.e., `1062C`[2], `721C`[3], `731C`[4], `742C`[5] and `822C`[6]. The challenges involved in the Codeforces dataset covers a variety of algorithms that are more complicated (e.g., disjoint-union sets, Dijkstra and greedy algorithm). Specifically, the detailed description of these challenges are described as follows:

- `1062C`: Given a binary-valued string and a list of intervals., for each interval, the frequencies of each value in the interval is used to calculate a formula. A prefix sum (and product) algorithm is required to solve this challenge.
- `721C`: Given a weighted directed graph, the shortest path is found between two specific nodes. Dijkstra algorithm is required to solve this challenge.
- `731C`: Given an undirected graph, the number of connected components in the graph is counted. A disjoint-union sets is required to solve this challenge.
- `742C`: Given a directed graph, the least common multiplier (LCM) is calculated for the lengths of all the circles in the graph. To solve this challenge correctly, circle finding algorithm and LCM algorithm are required.

1 https://codeforces.com.
2 https://www.codeforces.com/problemset/problem/1062/C.
3 https://www.codeforces.com/problemset/problem/721/C.
4 https://www.codeforces.com/problemset/problem/731/C.
5 https://www.codeforces.com/problemset/problem/742/C.
6 https://www.codeforces.com/problemset/problem/822/C.

Table 1
The statistics of program classification dataset for RQ1 and RQ5.

Index	CodeChef				Codeforces				
Problems	SUB	FLOW	MNMX	SUM	1062C	721C	731C	742C	822C
Instance Count	2313	5487	9693	11666	9136	16084	10170	6971	17379
Avg. Line of Code	30	25	25	36	45	65	55	52	55
Avg. Branches Count	9	8	8	12	12	10	21	15	18
Avg. Operators Count	25	15	15	35	40	40	29	30	39

- 822C: Given a collection of weighted intervals, a subset of the minimum weight sum is found to satisfy some conditions (e.g., no intersect between intervals). A greedy algorithm is required to solve this challenge.

For each program in both datasets, there is a label to indicate the running result of the corresponding program. The meaning of labels is detailed in the following:

- **Accepted (AC)**: The program is able to pass all test cases;
- **Wrong Answer (WA)**: The program can execute normally but output incorrect results;
- **Runtime Error (RE)**: The program cannot execute normally on some test cases, which are generally due to illegal memory access or operation error, e.g., divided by zero;
- **Time Limited Exceeded (TLE)**: The program does not response within the time limits;
- **Memory Limited Exceeded (MLE)**: The consumed resource, i.e., memory, exceed the requirement.

Except for the **AC**, different running results correspond to different defects in source code. For example, the source code with the **TLE** often contains redundant steps or dead loops, while the source code with the **WA** often contains functional errors. Therefore, we argue that a reasonable source code model should reflect these differences and is able to classify them effectively.

Additionally, we conducted a pre-processing on both datasets. First, we removed the source code that are irrelevant to the corresponding challenge. Second, we removed the duplicated ones from datasets. Third, to avoid mislabeling, we generated some test cases according to the requirements of the corresponding challenge. Then, we re-ran the source code and re-labeled them that were mis-labeled. Finally, for each dataset of challenges, we split each of them into training set, validation set and test set in 3:1:1 ratio. Table 1 shows some metrics of the final datasets.

For RQ2 and RQ3, we have selected another well-known public dataset, namely PROMISE. The reason is that it has been widely used for software defect prediction (Wang et al., 2016; Dam et al., 2019; Chen et al., 2020), and it consists of several well-known open-source Java projects. Except for the jedit (Version 3.2), which cannot be compiled properly, the remaining 10 Java projects and their corresponding versions that we selected are identical to a previous work (Wang et al., 2016) for comparison. Finally, 1395 source code files, which cannot be processed successfully by our Soot-based generator, were removed from the dataset. The statistical description of the final dataset for RQ2 and RQ3 is shown in Table 2.

For the remaining research question, i.e., RQ4, we have selected a public dataset, namely OJClone, which has been adopted by several works (Zhang et al., 2019; Fang et al., 2020). It was collected from an online program judgement system for C/C++ source code. Specifically, OJClone contains 15 program tasks, and each of them is composed of 500 source code files submitted by users. For the same task, different users' source codes could pass the test and got **AC** verdict, and thus can be considered as functional code clone. In other words, for each source code pair in the dataset, it will be labeled by either 0 for non-cloned pair or 1 for cloned pair. Similarly to the classifying task, we shuffled and split the dataset into training, validation and testing in 3:1:1 ratio.

Table 2
The statistics of PROMISE dataset, which is specialized for RQ2 and RQ3.

App	Ver	Mean files	Mean defective	Defective rate
lucene	3	247	140	56.7
synapse	3	188	52	27.7
xerces	2	295	54	18.3
xalan	2	665	237	35.6
camel	3	700	165	23.6
log4j	2	70	29	41.4
ant	3	422	95	22.5
jedit	3	311	67	21.5
poi	3	328	219	66.8
ivy	2	253	26	10.3

4.3. Experiment settings

In this section, we present the setup of each RQ's experiment, involving detailed settings about our method, the choices of baseline methods and comparison metrics.

4.3.1. Settings for MFGNN

The input of MFGNN consists of four parts: (1) a collection of AST nodes (represented by one-hot vectors); (2) a collection of AST's substructures; (3) a mapping graph (i.e., mapping the substructure to corresponding basic block); and (4) an ECFG As for the hyper-parameters, the embedding dimension of the AST nodes is set as 50. And the dimension of AGN4D, which is stacked with three layers, was set as 200. MFGNN was optimized by Adamax, and trained for 200 epochs. During the training, we selected the parameters (i.e., weights of MFGNN) that performed best on the validation set, and evaluated them on the test set.

4.3.2. Settings for baselines

For RQ1. To illustrate the effectiveness of MFGNN, we choose three other well-known groups of representative methods for comparison:

SVM-based approaches We chose SVM-based approaches to demonstrate that both datasets, i.e., CodeChef and Codeforces, do consist of source code with small textual but large semantic distinctions. In terms of classifying source code files according to their textual features, the more indistinguishable the source code are, the worse SVM-based methods would perform. To show the textual distinguishability of our dataset, we choose TF–IDF and BoW features as the textual features, and feed them into RBF-kernel SVM.

AST-based approaches To illustrate the advantages of our source code model over AST in program classification, we chose several typical AST-based approaches. Specifically, according to AST granularity, we can divide the AST-based approaches into two categories. One uses the entire AST of source code, like representative methods: TBCNN (Mou et al., 2016) and Tree-LSTM (Niepert et al., 2016). The other one splits AST according to code fragments and is known as ASTNN (Zhang et al., 2019). Moreover, code2vec (Alon et al., 2019) adopts paths in AST to represent the source code and learns the features contained in the paths through a network based on attention mechanisms. Similarly, code2seq (Alon et al., 2018a) uses the same paths as code2vec

but extracts the features by the seq2seq model (Sutskever et al., 2014).

For the settings of AST-based approaches, the AST used in TreeLSTM, TBCNN and ASTNN is generated by Clang, but the AST paths used by code2vec and code2seq are generated by ASTMiner.[7] For code2vec, the embedding dimension is set to 400; For code2seq, the embedding dimension is set to 128 and the decoder dimension is set to 320; The hidden dimension of the other methods is set to 200.

Graph-based approaches Some recent studies focused on representing a program as a graph and adopting a graph-based learning method to extract dependency features from the graph. DGCNN chooses CFG as the source code model and obtains features with GCN (Phan et al., 2018). ContextGraph (CtxG) inserts extra edges (e.g., dataflow edges) into the original AST, and extracts the features with GGNN (Li et al., 2015). For the settings of graph-based approaches, the number of steps of GGNN is set to 3, and the hidden size of all graph-based approaches was set to 200.

For RQ2. We evaluated the performance of MFGNN on Within-Project Defect Prediction (WPDP) task. According to previous studies on defect prediction task (Wang et al., 2016; Dam et al., 2019), we decided to use the same strategy, i.e., training by the earlier version and predicting on the later version. We compared MFGNN with several typical WPDP methods that can be divided into two types according to their adopted source code models. Some of defect prediction technologies used the features of the PROMISE with traditional machine learning methods (Menzies et al., 2007, 2010), including Adaboost, Multi-Layer Perception (MLP) and Random Forest (RF). The others utilize AST-based features, and representative methods (e.g., DBN Wang et al., 2016 and TreeLSTM Dam et al., 2019). Specifically, DBN obtained the semantic features from AST. We classified these features with three classifiers: Naive-Bayes (DBN_{NB}), Logistic Regression (DBN_{LR}) and Decision Tree (DBN_{DT}). As for TreeLSTM, after the AST was parsed by JavaParser,[8] it would take the entire AST as input for prediction. Additionally, we chose another two well-known methods: DTL-DP (Chen et al., 2020) and BugContext (Li et al., 2019). The former one visualized the source code file (or binary file) as an image, and obtained the defect features with AlexNet, while the later one acquired contextual dependencies from CFG and DFG, then introduced them into path-based AST features.

For RQ3. We conducted Cross-Project Defect Prediction (CPDP) experiments to show the performance of MFGNN. Following the previous studies (Wang et al., 2016; Dam et al., 2019), we organized ten groups of experiments, trained models on the *source project* and predicted on the *target project*. For the target project, according to transfer learning methods (Nam et al., 2013), we first randomly selected 30% of the data to fine-tune a LR-based classifier and then predicted the rest 70%. Except for DBN, which was replaced by its CPDP-variant: DBN-CP (Wang et al., 2016), we chose the same set of baseline methods as RQ2. Additionally, we added two transfer learning-based methods, namely TCA+ (Nam et al., 2013) and TNB (Maying et al., 2012), which take the PROMISE feature as same as the machine learning methods.

For RQ4. We conducted Functional Code-Clone Detection (CCD) experiments to demonstrate the distinguishability of semantics obtained by MFGNN. Let the features of the two source code files within a pair that are obtained from MFGNN be v_1 and v_2, respectively. The difference can be defined as $d = |v_1 - v_2|$. Finally, we use a LR-based classifier (i.e., $y = sigmoid(W_o d + b_o)$) to determine whether the code pairs are similar based on the vector d. We compared the performance of MFGNN with several state-of-the-art models that are widely used on CCD task, including RAE+ (Ferrante et al., 1987), Deckard (Jiang et al., 2007), CDLH (Wei and Li, 2017), ASTNN (Zhang et al., 2019), DeepSim (Zhao and Huang, 2018), and FCDetect (Fang et al., 2020).

For RQ5. We carried out some ablation studies. Our approach can be divided into two parts, a source code model based on ECFG and a learning model with the AGN4D layer. Firstly, we explored the impact of different choices in the design of our source code model, which has four options: (1) representing basic blocks with AST (**A**) or BoW features (**B**); (2) including control flow edges (**C**) or not; (3) including dataflow edges (**D**) or not; and (4) embedding the source code model with multi-typed edge (**M**) or with single-typed edge (**S**). We have designed four variants based on the combination of different options.

- **A**ST+**C**FG+**S**ingle: The main body of this model is CFG with no distinction between control flow types, and its basic blocks are represented using ASTs.
- **A**ST+**D**FG+**S**ingle: The main body of this model is DFG, with only one type of flow, and its basic blocks are represented using ASTs.
- **A**ST+**C**FG+**M**ulti: The main body of this model is a CFG that distinguishes between different control flows, and its basic blocks are represented using ASTs.
- **B**oW+**C**FG+**D**FG+**M**ulti: The main body of this model is a CFG that contains the dataflows and distinguishes between different types of flows. Its basic blocks are represented using BoW.

Secondly, we explored the impact of different graph learning methods. We replaced the AGN4D layer with graph convolution network (**GCN**) and gated-graph neural network (**GGNN**), respectively. Additionally, we compared across the different options in AGN4D, i.e., summation and concatenation, to synthesize graph features (see Eq. (2)) on the same source code model.

4.3.3. Metrics

For RQ1 and RQ5, we chose the *accuracy* and *macro-F1* (Liu et al., 2009) to evaluate the prediction result on test sets. Assuming a task has K classes, the *accuracy* is defined as follow:

$$accuracy = \frac{\sum_{i=1}^{K} TP_i}{N}, \tag{5}$$

where TP_i refers to true positive of class i, and N is the total number of samples.

For a binary classification task, the *F1-score* (F1) is defined as follow:

$$F1\text{-}score = \frac{2 * precision * recall}{precision + recall} \tag{6}$$

, where $precision = \frac{TP}{TP+FP}$ and $recall = \frac{TP}{TP+FN}$, TP denotes the true positive, FP represents the false positive, and FN refers to false negative.

A multi-label classification task can be considered as several binary classification tasks on different labels. Based on that, assuming the task has K classes, the *macro-F1* can be defined as follow:

$$Macro\text{-}F1 = \frac{1}{K}\sum_{i=1}^{K} F1\text{-}score_i. \tag{7}$$

For RQ2 and RQ3, in addition to the F1 on the buggy class, we also used the metric *AUC* (Area Under the receiver operating characteristics Curve) (Dam et al., 2019) to evaluate the performance

[7] https://github.com/JetBrains-Research/astminer.

[8] https://javaparser.org.

of defect prediction. Specifically, *AUC* refers to the probability of a classifier ranking a randomly selected positive sample higher than a randomly selected negative sample. Intuitively speaking, a higher value of *AUC* implies a better performance.

For RQ4, following the evaluation metrics of previous works (Zhang et al., 2019; Fang et al., 2020), we choose *precision* (P), *recall* (R) and F1 to measure the performance of the selected models on CCD task.

5. Results

In this section, we show the results of the experiments, and compare the performance of different methods.

5.1. Answer to RQ1

Table 3 illustrates the results related to RQ1, and the best performance are highlighted in bold. In column 2, we list the size of the corresponding model except for the SVM-based approaches, whose size is neglectable. According to these experimental results, we have the following insights:

The dataset does consist of source code with a minimal textual difference. As we can see, SVM-based methods did not play well in our experiments, which is reflected by their corresponding F1 values. This indicates that the source codes with different labels in our dataset cannot be effectively distinguished by textual features. In other words, it proves that the textual differences among the source codes in out dataset are too small to be distinguished effectively.

Compared to AST-based approaches, MFGNN achieves a better performance with fewer parameters. Compared to the best method, i.e., TreeLSTM, among AST-based approaches, MFGNN reduces the model parameters by up to 50%, while achieving 4.0% and 6.8% improvements on accuracy and F1, respectively. Additionally, we can observe that both of code2vec and code2seq did not perform well. This is because both of them model the source code by sampling the path of the AST, which can only capture potential connections between code tokens (Jiang et al., 2019). Program classification task, however, requires the identification of the actual control flow and dataflow information of the program execution, which cannot be achieved by their models. On the contrary, our source code model can reflect the actual execution path of the program with contextual information, which can be better captured by the neural network.

MFGNN achieves a significant performance improvement while adding a limited number of parameters compared with the graph-based approaches. Compared to the best graph-based approach, DGCNN, MFGNN only increases the number of parameters by 4 times, but achieves 5% and 8.1% improvement on accuracy and F1, respectively. Similarly, compared with DGCNN, which has the same scale of parameters as MFGNN, MFGNN achieves 4.8% and 8.4% improvement on accuracy and F1, respectively. This result illustrates that the performance of MFGNN has little correlation with its number of parameters. The main difference between MFGNN and traditional graph-based methods is two-fold. On one hand, the integration of multiple flow information in the source code model clearly expresses the dependency features of the program well. On the other hand, the attention mechanism allows MFGNN to dynamically adjust the weights of different types of flows, resulting in a better mining of the flow features.

5.2. Answer to RQ2

Table 4 shows the performance of different approaches on the within-project defect prediction (WPDP) task, and the best performances are highlighted in bold. Due to the limitations of Soot (e.g., throw exceptions on some data items), our dataset lost a large number of entries in some projects, which resulted in the distribution of the dataset we actually used differs from the previous study (Wang et al., 2016). To ensure the fairness of the comparison, we re-implemented the DBN methods and TreeLSTM mentioned in Dam et al. (2019). We selected multiple groups of parameters randomly, ran all methods multiple times and kept the best result.

Compared with the state-of-art method, namely TreeLSTM, MFGNN achieved 1.6% and 4.0% improvements on F1 and AUC, respectively. Moreover, MFGNN was 5% and 29.6% higher in F1 and AUC, respectively, than DTLDP. Specifically, higher AUC often means that the model has more confidence in the prediction results, and the main difference between MFGNN and these methods is the use of ECFG on the source code model allows MFGNN to capture contextual dependencies.

Compared to the BugContext method, MFGNN improved 7.3% and 18.7% in F1 and AUC, respectively. We think such a significant improvement can be attributed to their structural difference, which can be divided into three-fold. First, ***the representation of basic blocks.*** According to the open-source implementation of the BugContext, it only embeds line numbers into basic blocks, while MFGNN uses AST to represent those basic blocks. Second, ***the process of learning AST features.*** BugContext learns tree features by sampling the paths of the tree, while TBCNN is adopted to learn the features by MFGNN. Third, ***the process of learning graph.*** MFGNN uses AGN4D to capture the dependency features in the graph, while BugContext uses node2vec to learn the information in the PDG. The biggest advantage of AGN4D over node2vec is the introduction of an attention mechanism, which allows different types of dependency features to be fused. In conclusion, the hybrid features obtained by MFGNN could perform better on WPDP task.

5.3. Answer to RQ3

The cross-project defect prediction (CPDP) task mentioned in RQ3 mainly evaluates whether the contextual features learnt by the model can be applied to different projects. To answer this question, we compared our proposed method, MFGNN, with several typical CPDP methods, and the results are shown in Table 5. The best performance among all methods are marked in bold. Depending on the type of input data, we can further divide the performance into two types: the best performance among metric-based methods is marked with underline and among source code model-based methods is marked in lightgray.

Among all methods, MFGNN achieved the highest overall F1 and AUC. Compare to the best metric-based methods, MFGNN outperformed 6.3% and 1.9% in F1 and AUC, respectively. Compare to other source code model-based methods, MFGNN achieved the highest F1 and AUC in most of the tasks. Interestingly, the BugContext does not perform as well as its result on the WPDP task (see Section 5.2). Compared with BugContext, the F1 and AUC of MFGNN were improved by up to 27.6% and 15.8%, respectively. We think the reason of improvement lies behind their difference of using context-dependent information, which could be divided into two-fold. On one side, BugContext uses dependency features to assist AST features, while MFGNN does the opposite. Learning program context-dependent features is critical for CPDP task, thus such a design difference can lead to a discrepancy in performance. On the other side, BugContext extracts features from CFG and

Table 3
Results on program classification task, the numbers in parentheses are the parameter sizes of methods.

Groups	Methods	SUB		MNMX		FLOW		SUM		1062C		721C		731C		742C		822C		Avg	
		Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
SVM	SVM&TF–IDF	34.7	12.9	48.0	16.2	56.6	18.1	38.4	13.9	51.0	13.6	38.7	11.2	42.9	12.0	60.0	15.0	56.2	14.4	47.4	14.1
	SVM&BoW	54.5	41.5	68.6	52.5	80.0	60.8	59.9	52.6	55.7	21.5	42.6	22.6	55.4	33.1	66.9	26.2	56.8	16.6	60.0	36.4
AST	TBCNN (0.5M)	67.2	65.2	74.6	69.2	75.3	66.0	63.8	62.4	63.0	39.9	53.7	47.1	65.6	52.9	66.9	38.8	58.9	48.9	65.4	54.5
	TreeLSTM (4.0M)	66.1	64.1	76.0	69.5	76.8	68.4	66.3	65.9	66.9	47.7	56.0	50.6	69.1	53.1	70.0	41.2	**60.7**	50.2	67.5	56.7
	ASTNN (0.9M)	61.4	58.9	70.3	63.1	74.3	62.4	62.7	62.4	63.6	46.6	49.9	44.3	61.2	50.0	64.6	32.6	55.0	42.3	62.6	51.4
	code2vec (173M)	29.5	24.7	36.6	25.7	29.7	21.1	31.4	24.5	41.2	18.4	28.9	18.5	28.1	18.2	49.2	18.0	30.7	14.5	33.9	20.4
	code2seq (61M)	35.9	16.9	50.4	16.9	51.7	30.0	31.9	21.3	51.4	13.6	36.0	15.1	43.0	13.9	52.3	17.3	56.5	14.5	45.5	17.7
Graph	DGCNN (0.4M)	64.8	64.5	74.6	67.7	**83.8**	70.9	69.1	67.4	64.3	42.8	54.2	49.6	61.4	47.0	70.3	44.1	56.2	44.5	66.5	55.4
	DGCNN (2.4M)	64.4	62.6	74.2	66.5	82.7	**72.0**	69.1	67.9	64.8	42.1	55.3	49.9	61.7	48.5	72.6	43.5	55.9	42.5	66.7	55.1
	CtxG (4.9M)	64.8	62.0	74.0	68.0	74.9	63.9	64.9	64.6	59.1	42.0	51.1	45.3	59.0	47.8	65.0	36.8	56.4	43.3	63.2	52.6
MFGNN (2.1M)		**74.5**	**74.7**	**83.1**	**81.4**	81.8	71.0	**72.9**	**73.5**	**68.0**	**53.2**	**59.5**	**54.5**	**70.0**	**61.0**	**73.8**	**51.6**	59.9	**50.3**	**71.5**	**63.5**

Table 4
The result of WPDP experiment on PROMISE.

Methods			Adaboost		MLP		RF		DBN_{NB}		DBN_{LR}		DBN_{DT}		Tree-LSTM		DTLDP		BugContext		MFGNN	
Project	Tr	T	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC
ant	1.5	1.6	37.8	68.4	32.0	72.5	36.2	70.5	4.3	**81.5**	40.7	80.7	4.3	51.1	29.7	49.7	**45.3**	22.8	31.1	44.4	33.1	72.5
	1.6	1.7	52.2	69.4	51.4	71.6	49.1	74.1	53.2	69.3	51.7	**79.0**	22.8	50.6	44.2	60.8	35.5	50.0	45.1	44.7	**53.7**	75.5
camel	1.2	1.4	40.2	70.3	39.8	68.6	47.2	75.9	12.9	53.1	16.5	40.4	9.3	51.9	53.1	82.7	32.9	34.1	36.2	52.6	**54.3**	**83.6**
	1.4	1.6	40.2	70.9	30.7	68.9	45.9	70.1	13.7	58.4	32.0	58.4	8.0	44.2	55.9	79.7	34.7	44.2	27.8	50.3	**56.8**	**84.0**
ivy	1.4	2.0	14.3	66.9	14.8	67.8	23.1	**69.4**	**47.6**	61.5	27.3	57.9	26.7	57.7	15.9	45.8	21.1	18.5	31.9	44.5	22.9	60.2
jedit	4.0	4.1	57.0	80.7	54.3	80.4	54.5	79.9	41.3	45.6	41.6	50.0	0.0	50.4	62.0	78.8	23.8	35.7	38.5	63.1	**65.0**	**84.4**
lucene	2.0	2.2	58.5	63.7	59.9	63.4	59.4	**65.7**	32.7	65.3	36.6	65.4	35.8	53.3	60.9	59.9	58.9	48.0	43.0	58.4	**64.6**	64.0
	2.2	2.4	64.8	56.6	68.4	57.5	64.8	62.1	25.7	47.3	37.4	**73.3**	14.2	71.6	68.1	59.1	**68.8**	40.3	68.0	60.3	**68.8**	63.4
log4j	1.0	1.1	66.7	78.0	73.3	82.5	75.0	84.2	75.0	88.5	60.5	**90.2**	72.3	64.8	73.3	75.8	24.0	46.9	**75.5**	66.7	73.3	77.0
poi	1.5	2.5	77.3	72.6	78.4	72.1	73.3	74.3	8.5	45.8	8.4	65.4	13.4	40.9	81.6	75.8	81.9	59.5	79.7	62.1	**83.1**	**78.4**
	2.5	3.0	54.6	50.2	68.4	52.2	58.7	55.6	28.0	76.4	27.0	78.6	8.9	**78.7**	73.9	69.5	**77.7**	71.9	65.2	58.3	73.3	69.2
synapse	1.0	1.1	28.9	64.6	15.0	61.1	14.7	57.9	47.9	64.4	43.0	**66.3**	**48.9**	60.5	28.2	43.2	41.0	51.7	18.8	40.1	30.4	61.1
	1.1	1.2	40.3	61.2	44.1	64.4	40.0	66.8	41.5	**69.1**	41.5	50.1	35.9	66.5	50.3	57.8	**54.4**	43.7	42.4	55.0	50.3	65.6
xalan	2.4	2.5	32.9	62.1	21.9	59.7	27.9	59.1	19.1	51.1	30.8	58.2	10.6	55.4	34.5	**63.9**	**50.4**	43.8	17.4	51.9	33.1	58.7
xerces	1.2	1.3	29.6	62.6	24.2	60.3	25.7	57.9	24.1	53.5	32.4	64.0	**33.3**	64.5	29.4	60.7	14.8	29.4	9.4	51.5	30.9	**74.2**
Avg			46.4	66.5	45.1	66.9	46.4	68.2	31.7	62.1	35.2	65.2	23.0	57.5	50.7	64.2	44.3	42.7	42.0	53.6	**52.9**	**71.5**

DFG separately, while MFGNN combines them into the ECFG and extracts features uniformly by AGN4D.

In conclusion, the contextual features obtained by MFGNN are more generalized and are able to result in better performance on the CPDP task.

5.4. Answer to RQ4

Table 6 illustrates the results related to RQ4, and the best results are highlighted in bold. Compared with other methods, MFGNN achieved the highest recall and F1, as well as a relative high precision. Interestingly, we could observe that FCDetect plays well, which apply call graph as the source code model. However, we argue that MFGNN can capture the program context-dependency features more effectively. The main difference between them is the graph learning mechanisms they adopted. Compared to the Graph2Vec adopted by FCDetect, MFGNN uses AGN4D based on the attention mechanism, and thus could adjust the weights of different types of dependency information. Therefore, with the help of more context-dependency features, MFGNN could identify program variants more effectively, leading to higher recall and F1 scores.

Fig. 4 shows the absolute distances of features derived from MFGNN for the data in the test set. We can observe that there is a clear demarcation line between the red and blue dots. This illustrates the features obtained by MFGNN can effectively distinguish source codes under the functional code-clone task. In conclusion, MFGNN can improve the performance of distinguishing between non-cloned and cloned source code pairs.

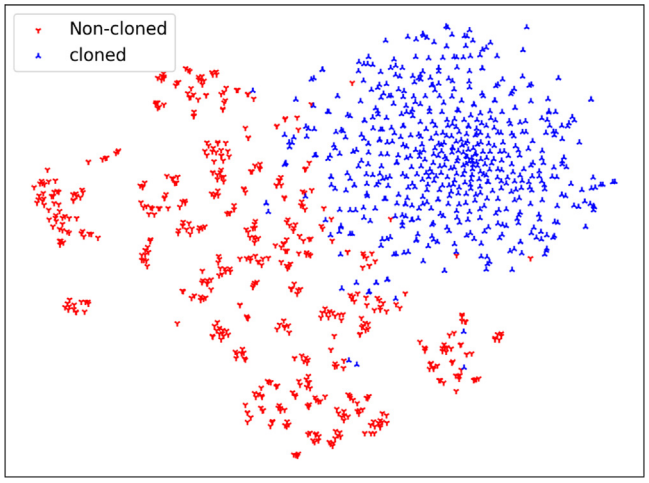


Fig. 4. t-SNE mapping of the absolute distances of the test set's pairs' features. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.5. Answer to RQ5

To answer this question, we have adjusted the default settings in our original methodology and compared their performance on the program classification task. The results are shown in Table 7,

Table 5
The result of CPDP experiment on PROMISE.

Source	Target	Adaboost		MLP		RF		TCA+		TNB		DBN-CP		DTLDP		BugContext		MFGNN	
		F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC
ant-1.6	camel-1.4	23.9	62.9	37.2	63.3	26.8	65.9	28.0	52.9	**40.0**	**67.5**	31.9	60.7	22.8	42.0	22.6	42.5	36.3	65.3
jedit-4.1	camel-1.4	25.7	60.4	26.5	49.4	16.7	59.7	29.0	51.2	32.0	64.2	23.4	61.1	31.3	54.9	11.7	51.6	**39.8**	**67.4**
camel-1.4	ant-1.6	54.3	69.4	32.8	38.1	38.2	70.1	25.0	42.1	**59.0**	**79.0**	56.1	74.3	47.8	18.6	22.2	66.9	50.3	71.4
poi-3.0	ant-1.6	48.8	68.7	**62.7**	**80.3**	55.0	73.8	28.0	46.8	53.0	73.6	48.2	63.3	44.8	26.5	51.0	68.2	56.9	75.2
camel-1.4	jedit-4.1	34.8	57.7	13.2	30.2	37.6	71.1	50.0	63.8	**53.0**	**76.2**	32.3	59.1	38.4	36.7	45.2	70.2	41.5	64.2
log4j-1.1	jedit-4.1	57.7	78.6	56.1	72.1	56.6	78.5	18.0	35.7	**62.0**	**79.4**	48.4	67.4	39.9	62.0	38.0	49.8	57.8	78.8
jedit-4.1	log4j-1.1	26.3	63.4	0.0	13.2	12.9	**84.9**	61.0	61.8	**71.0**	84.3	37.8	61.1	59.6	49.1	31.6	24.2	57.1	67.9
lucene-2.2	log4j-1.1	64.1	74.1	60.4	**92.0**	**70.8**	82.3	52.0	60.9	63.0	79.7	45.2	53.8	46.5	41.7	53.2	62.4	54.5	63.1
lucene-2.2	xalan-2.5	63.6	57.4	**68.5**	60.9	61.7	61.1	58.0	54.8	45.0	53.0	57.2	61.0	37.8	54.4	43.2	48.9	67.4	**66.6**
xerces-1.3	xalan-2.5	38.4	50.7	62.8	59.0	21.8	56.0	59.0	53.9	57.0	53.5	26.8	46.9	**64.9**	40.2	23.6	55.7	63.5	**61.1**
xalan-2.5	lucene-2.2	46.5	54.8	**74.7**	**64.0**	51.6	59.7	64.0	63.1	54.0	57.9	56.4	60.5	74.5	50.7	65.4	54.8	64.3	56.6
log4j-1.1	lucene-2.2	49.3	62.3	37.8	57.9	55.0	60.8	60.0	55.6	54.0	**63.1**	52.7	55.8	**76.4**	56.1	62.9	50.0	70.2	**63.1**
xalan-2.5	xerces-1.3	35.4	55.9	0.0	37.8	39.3	64.7	23.0	39.5	31.0	49.6	32.4	57.5	15.7	43.4	34.4	62.4	**50.0**	**74.3**
ivy-2.0	xerces-1.3	12.5	64.2	0.0	33.8	20.0	52.7	45.0	66.7	37.0	60.3	36.6	59.6	29.4	51.3	32.1	53.2	**47.8**	**71.7**
xerces-1.3	ivy-2.0	34.6	71.1	**39.5**	**79.7**	35.5	70.1	30.0	68.9	34.0	77.2	30.5	57.2	11.3	54.5	25.3	67.8	37.4	79.5
synapse-1.2	ivy-2.0	33.3	74.3	**51.1**	78.7	34.7	74.5	24.0	62.5	38.0	**82.1**	29.6	62.0	22.0	18.2	40.7	71.9	39.0	78.9
ivy-1.4	synapse-1.1	9.4	66.1	20.9	35.6	3.4	63.2	45.0	61.4	**51.0**	**70.0**	9.7	51.9	15.7	54.1	9.4	37.4	42.7	57.8
poi-2.5	synapse-1.1	28.3	48.1	34.9	54.2	46.5	62.9	43.0	62.7	5.0	44.4	**49.0**	63.4	35.2	30.0	37.0	56.4	48.5	**68.6**
ivy-2.0	synapse-1.2	39.7	69.7	34.5	49.7	24.2	68.9	52.0	62.3	57.0	70.7	32.4	53.6	45.7	39.8	17.5	50.2	**62.0**	**73.3**
poi-3.0	synapse-1.2	56.3	69.9	55.8	66.0	53.8	56.2	56.0	67.6	43.0	62.8	49.5	62.3	29.4	34.0	49.8	55.2	**65.7**	**75.1**
synapse-1.2	poi-3.0	57.7	74.1	51.7	59.3	27.2	70.0	72.0	61.6	71.0	75.6	48.5	59.5	73.9	56.5	66.2	56.7	**81.4**	**82.2**
ant-1.6	poi-3.0	47.0	68.5	47.2	53.2	37.8	70.2	38.0	33.9	65.0	79.7	43.5	66.0	33.3	56.4	44.7	41.6	**81.1**	**84.3**
Avg		40.3	64.6	39.5	55.8	37.6	67.2	43.6	55.9	48.9	68.4	39.9	59.9	40.7	44.1	37.6	54.5	**55.2**	**70.3**

Table 6
The results of ccd task on OJClone.

Methods	RAE+	Deckard	CDLH	ASTNN	DeepSim	FCDetect	MFGNN
P	52.5	**99**	47	98.9	70	97	96.7
R	68.3	5	73	92.7	83	95	**96.3**
F1	59.4	10	57	95.5	76	96	**96.5**

in which the default settings are highlighted in bold. We can obtain the following insights:

Sensitivity to the control flow and dataflow differs from challenges. Using only DFG as the source code model (i.e., A+D+S) works better on some challenges, e.g., SUB and SUM. This is because there are much more operators than branches within these source code (see Table 1). In other words, these challenges have simple control flows, but complex computational logic, which is related to data flow heavily. Thus, compare to control flow edges, data flow edges play a more critical role on the test results. However, in general, CFG only (i.e., A+C+S) could perform better than adopting only DFG.

Introducing different types of edges in CFG may lead to poorer performance. Introducing different types of edges plays a positive role on some challenges, including SUB, 721C, 731C, 742C, and 822C. However, on other challenges, MFGNN performs better when the source code model is untyped (e.g., A+C+S). This is because these challenges require fewer branches than the others (see Table 1). The imbalanced distribution of types lead to ineffective optimization of the model on different types. Therefore, the uneven distribution of the number of different edge types prevents MFGNN from effectively fusing the features of different types of flows.

AST is a better choice for node representation in our experiment settings. The results show that using AST as a node representation improved the model's performance significantly. Even when the other settings in the approach were removed (e.g., A+C+S which removed data flow edges and edge types, or A+D+S which removed control flow edges), the approach still performed better than B+C+D+M, which represents node by Bag-of-Words (BoW) model instead of AST. Compared to the model in BoW, i.e., B+C+D+M, our source code model (A+C+D+M) resulted in 7.7% and 9.8% improvement on accuracy and F1, respectively. Because the node representation is the only independent variable here, we can conclude that AST is a better node representation option for our task. Compared to AST, BoW lacks both the lexical order and syntactic structures, which are essential for a proper representation of basic blocks.

AGN4D is the best choice among the three GNNs. To examine the effectiveness of AGN4D, we altered it into two other common GNNs, i.e., GCN and GGNN, respectively, into our approach for a comparison study. Table 7 shows that AGN4D outperformed the other two GNNs, with an average of 2.7% and 5.3% higher accuracy and F1, respectively.

Summation is a better choice than concatenation in contextual feature embedding stage. From the results, the use of summation as a graph feature synthesis method (i.e., the last formula of (2)) delivered better performance. This is because concatenation doubles AGN4D's hidden dimension layer by layer, increasing the number of model parameters and resulting in model overfitting issue.

5.6. Threats to validity

In conducting our experiments, the following factors existed that might affect the validity of the our study.

Table 7
Results of ablation studies.

Different settings	SUB		MNMX		FLOW		SUM		1062C		721C		731C		742C		822C		Avg	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
A+C+S	70.6	70.6	80.9	79.5	82.6	72.2	71.4	72.2	66.8	57.0	59.2	53.5	66.4	56.9	74.6	48.6	60.2	50.8	70.3	62.4
A+D+S	70.8	71.4	80.9	76.7	78.7	69.3	71.5	71.8	66.2	43.5	57.5	53.7	63.8	56.3	72.2	45.0	56.3	49.6	68.7	59.7
A+C+M	70.6	72.2	80.7	77.6	82.1	70.7	70.8	70.8	64.9	48.8	59.7	54.2	66.9	56.6	75.5	52.9	59.4	52.0	70.1	61.8
B+C+D+M	69.7	68.3	75.4	68.8	72.8	63.5	64.6	64.0	57.1	38.8	49.0	44.2	61.6	51.7	68.0	40.3	55.7	43.9	63.8	53.7
A+C+D+M	74.5	74.7	83.1	81.4	81.8	71.0	72.9	73.5	68.0	53.2	59.5	54.5	70.0	61.0	73.8	51.6	59.9	50.3	**71.5**	**63.5**
concatenation	66.9	63.8	79.5	74.6	80.6	69.4	70.6	70.1	65.9	51.0	55.8	50.4	66.8	54.0	69.8	43.4	60.3	47.0	68.5	58.2
summation	74.5	74.7	83.1	81.4	81.8	71.0	72.9	73.5	68.0	53.2	59.5	54.5	70.0	61.0	73.8	51.6	59.9	50.3	**71.5**	**63.5**
GCN	72.3	70.4	80.4	77.2	81.3	71.8	70.7	70.8	64.6	43.5	56.6	52.2	64.3	53.2	69.7	39.9	59.8	48.8	68.9	58.6
GGNN	74.7	73.9	83.2	81.7	82.8	72.1	71.5	71.2	62.7	41.6	53.7	48.1	63.6	50.6	69.5	42.7	56.7	38.0	68.7	57.8
AGN4D	74.5	74.7	83.1	81.4	81.8	71.0	72.9	73.5	68.0	53.2	59.5	54.5	70.0	61.0	73.8	51.6	59.9	50.3	**71.5**	**63.5**

Implementation of baselines. The internal threat to validity is concerned with our implementation. We reproduced TBCNN, ASTNN, CtxG, DBN, TCA+, TreeLSTM, BugContext. Although we have implemented these baseline methods as described in the original studies, we cannot guarantee that these implementations exactly match the original ones.

Applying baselines on our dataset. In carrying out the task, we found that many of the baseline methods were designed specifically for a particular task, for example code2vec's goal was to perform function name generation and CtxG's goal was to perform var-misuse detection. Although we compared these methods as baselines, we cannot guarantee that these we can meet the conditions for these representations of the model to work well.

Missing projects in PROMISE dataset. Our RQ2 and RQ3 experiments are based on the PROMISE dataset, a very early dataset in which some versions of projects recorded are not available on the web. We were only able to conduct experiments using projects that could be found and could not directly use the original experimental data from the DBN (Wang et al., 2016) and TreeLSTM (Dam et al., 2019) studies.

CFG differences in different languages. For C/C++, we use Clang to get the CFG, which converts the program to LLVM IR, a kind of three-address code, and then builds the CFG on top of that. For Java, we use Soot to get the CFG. Soot will first convert the program into Jimple, a kind of SSA, and then build the CFG on top of that. Because of the difference in the intermediate languages used, the final CFG may not be exactly the same for the same statements in both languages.

Conduct experiments on more tasks and more practical datasets. To evaluate the feasibility and effectiveness of MFGNN, we have conducted several tasks (e.g., program classification and defect prediction) on the datasets consisting of source codes from OJ and open source projects. Though the variety of evaluated tasks and the sources of datasets were limited, we argue that MFGNN is robust enough even on large-scale real-world industrial code to perform other types of tasks, which, however, requires follow-up studies in the future.

6. Related works

6.1. Source code representation in deep learning

While performing program analysis with deep learning, the representation model of source code is a fundamental problem, which could be roughly divided into: AST-based and CFG-based. Specifically, as for the AST-based source code model, some studies adopted the AST that is generated from the program directly (White et al., 2016; Mou et al., 2016; Dam et al., 2019) or with some modifications (e.g., inserting additional edges between nodes Allamanis et al., 2017). Moreover, some works (Zhang et al., 2019; Alon et al., 2019, 2018a) just extracted part of the generated AST to conduct the following analysis. For examples, Alon et al. (2019, 2018a) chose the collection of AST's token-to-token path as the source code model, and learned the features by attention-based models. Unlike these models, we chose to split the AST into subtrees based on basic blocks. Though it would slightly broke the integrity of the AST, the explicit contextual dependencies in the CFG could reassemble parts of the AST, making dependencies more salient and easier to learn.

As for the CFG-based source code model, there are two factors that significantly affect the following program analysis with deep learning. One is the way of representing of basic blocks; the other is the role of the graph. To be specific, several works have tried different way to basic blocks in deep learning, e.g., assembly instruction (Phan et al., 2018), Bag-of-Words model (Fang et al., 2020; Wang et al., 2020) and line number (Li et al., 2019). As for the graph, it can be utilized as a leading role (Phan et al., 2018; Fang et al., 2020; Wang et al., 2020) or an auxiliary role (Li et al., 2019) during the analysis. For example, Wang et al. (2020) used graph as a leading role and represents basic blocks with Bag-of-Words model composed of AST's grammatical nodes. Our model similarly adopted graph as a leading role, but represented basic blocks with the corresponding subtree of AST. We retained the structure of AST, which helped us better represent the context-independent grammatical differences than other models.

6.2. Program classification

Program classification, i.e., distinguishing and classifying programs by some features from various aspects, is one of the basic software engineering tasks. For example, as one of the applications, functional code clone detection (Zhang et al., 2019; Fang et al., 2020; Yu et al., 2019) is to determine whether two code snippets implement the same functionality. It is achieved by classifying the functional features of the given program. Except from functional features (Mou et al., 2016; Zhang et al., 2019), language features (Ugurel et al., 2002), defect features (Dam et al., 2019; Wang et al., 2016; Phan et al., 2018) and structure features (Zanoni et al., 2015) are also widely adopted by program classification tasks. In this paper, we decided to apply defect features on classifying program test results. Though Phan et al. (2018) have done this task before, the size of dataset and code complexity were relatively limited compared to ours, which were collected and constructed by crawlers and huge manual efforts.

6.3. Software defect prediction

Software defect prediction is a challenging task that has been researched extensively. Prior to the rise of deep learning, researchers have adopted machine learning to achieve such a goal (Nam et al., 2013; Yang et al., 2015; Walden et al., 2014; Xia et al.,

2016; Breiman, 2001; Briand et al., 2002; Khoshgoftaar and Lanning, 1995; Khoshgoftaar et al., 2000; Xing et al., 2005; Munson and Khoshgoftaar, 1992). However, these techniques require feature engineering that is normally time- and resource-consuming. For example, Xing et al. (2005) proposed a SVM-based defect predicting methods, which depends on both software change metrics and software complexity metrics. Deep learning techniques eliminated the process of feature engineering, and researchers began focusing on improving prediction performance using suitable source code models (Yang et al., 2015; Wang et al., 2016; Chen et al., 2020; Dam et al., 2019). Existing works have pointed out that the source code model needs contextual dependencies (Li et al., 2017) and should be able to distinguish subtle changes (Wang et al., 2016). Both of them were taken into account in our method. Specifically, the contextual dependencies comes from the ECFG; and the subtle changes, i.e., subtle grammatical differences, are represented by the structural differences of the AST. To the best of our knowledge, no other existing source code models have achieved both of these goals.

7. Conclusion

In this paper, we have proposed a new source code model based on ECFG and an attention-based model, namely MFGNN. Our source code model restricts the order in which MFGNN extracts features, and makes it more efficient and effective for MFGNN to obtain program features. Moreover, we have evaluated MFGNN on three practical tasks: program classification, software defect prediction and code clone detection. The results showed that MFGNN significantly outperformed baseline methods. For example, compared with the well-known source code model code2seq (Alon et al., 2018a), the scale of parameters decreased more than 30-fold while the overall accuracy was increased by 26.0%. Our research illustrated that the performance heavily depended on the construction of source code model. Additionally, we highlights a few research directions for future work, e.g., applying our method on more general real-life projects and improving the graph and MFGNN for better performance.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank anonymous reviewers for their thoughtful comments. Thanks to Ningyu He for proofreading the manuscript. This research is supported by the National Key R&D Program of China under Grant No. 2020AAA0109400, the National Natural Science Foundation of China under Grant Nos. 62072007, 61832009, 61620106007, 61502011, the Australian Research Council Discovery Project (Grant No. DP210102447), and "the Fundamental Research Funds for the Central Universities, China" (BLX202003).

References

Allamanis, M., Brockschmidt, M., Khademi, M., 2017. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740.

Alon, U., Levy, O., Yahav, E., 2018a. Code2seq: Generating sequences from structured representations of code. CoRR abs/1808.01400, arXiv:1808.01400, URL http://arxiv.org/abs/1808.01400.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2018b. A general path-based representation for predicting program properties. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2Vec:Learning distributed representations of code. In: Proceedings of the ACM on Programming Languages, Vol. 3. POPL, pp. 1–29. http://dx.doi.org/10.1145/3290353, arXiv:1803.09473.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Briand, L.C., Melo, W.L., Wust, J., 2002. Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Trans. Softw. Eng. 28 (7), 706–720.

Bruna, J., Zaremba, W., Szlam, A., LeCun, Y., 2014. Spectral networks and locally connected networks on graphs. In: 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14–16, 2014, Conference Track Proceedings. URL http://arxiv.org/abs/1312.6203.

Chen, J., Hu, K., Yu, Y., Chen, Z., Xuan, Q., Liu, Y., Filkov, V., 2020. Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction. pp. 578–589.

Cvitkovic, M., Singh, B., Anandkumar, A., 2018. Open vocabulary learning on source code with a graph-structured cache. arXiv:1810.08305, URL http://arxiv.org/abs/1810.08305.

Dam, K.H., Pham, T., Ng, S.W., Tran, T., Grundy, J.C., Ghose, A.K., Kim, T., Kim, C.-J., 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, pp. 46–57.

Fang, C., Liu, Z., Shi, Y., Huang, J., Shi, Q., 2020. Functional code clone detection with syntax and semantics fusion learning. In: ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 516–527. http://dx.doi.org/10.1145/3395363.3397362.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9 (3), 319–349. http://dx.doi.org/10.1145/24039.24041.

Fout, A., Byrd, J., Shariat, B., Ben-Hur, A., 2017. Protein interface prediction using graph convolutional networks. In: Advances in Neural Information Processing Systems, pp. 6530–6539.

Frantzeskou, G., MacDonell, S., Stamatatos, E., Gritzalis, S., 2008. Examining the significance of high-level programming features in source code author classification. J. Syst. Softw. 81 (3), 447–460.

Hamaguchi, T., Oiwa, H., Shimbo, M., Matsumoto, Y., 2017. Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach. arXiv preprint arXiv:1706.05674.

Hamilton, W., Ying, Z., Leskovec, J., 2017. Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems, pp. 1024–1034.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, pp. 200–210.

Jiang, L., Liu, H., Jiang, H., 2019. Machine learning based recommendation of method names: How far are we. In: Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE 2019, IEEE, pp. 602–614. http://dx.doi.org/10.1109/ASE.2019.00062.

Jiang, L., Misherghi, G., Su, Z., Glondu, S., 2007. DECKARD: scalable and accurate tree-based detection of code clones. In: 29th International Conference on Software Engineering. ICSE'07, pp. 96–105. http://dx.doi.org/10.1109/ICSE.2007.30.

Khoshgoftaar, T.M., Lanning, D.L., 1995. A neural network approach for early detection of program modules having high risk in the maintenance phase. J. Syst. Softw. 29 (1), 85–91.

Khoshgoftaar, T.M., Yuan, X., Allen, E.B., 2000. Balancing misclassification rates in classification-tree models of software quality. Empir. Softw. Eng. 5 (4), 313–330.

LeClair, A., Jiang, S., McMillan, C., 2019. A neural model for generating natural language summaries of program subroutines. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 795–806.

Li, J., He, P., Zhu, J., Lyu, M.R., 2017. Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security. QRS, pp. 318–328.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R., 2015. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.

Li, Y., Wang, S., Nguyen, T.N., Nguyen, S.V., 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. Proc. ACM Program. Lang. 3, 1–30.

Liu, Y., Loh, H.T., Sun, A., 2009. Imbalanced text classification: A term weighting approach. Expert Syst. Appl. 36 (1), 690–701.

Maying, LuoGuangchun, Zengxue, ChenAiguo, 2012. Transfer learning for cross-company software defect prediction. Inf. Softw. Technol.

Menzies, T., Greenwald, J., Frank, A., 2007. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. 33, 2–13. http://dx.doi.org/10.1109/TSE.2007.10.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.B., 2010. Defect prediction from static code features: current results, limitations, new approaches. Autom. Softw. Eng. 17 (4), 375–407. http://dx.doi.org/10.1007/s10515-010-0069-5.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Thirtieth AAAI Conference on Artificial Intelligence.

Munson, J.C., Khoshgoftaar, T.M., 1992. The detection of fault-prone programs. IEEE Trans. Softw. Eng. 26 (5), 423–433.

Nam, J., Pan, S.J., Kim, S., 2013. Transfer defect learning. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 382–391.

Niepert, M., Ahmed, M., Kutzkov, K., 2016. Learning convolutional neural networks for graphs. In: International Conference on Machine Learning, pp. 2014–2023.

Ott, J., Atchison, A., Harnack, P., Best, N., Anderson, H., Firmani, C., Linstead, E., 2018. Learning lexical features of programming languages from imagery using convolutional neural networks. In: Proceedings of the 26th Conference on Program Comprehension, pp. 336–339.

Phan, A., Nguyen, L., Nguyen, Y., Bui, L., 2018. DGCNN: A convolutional neural network over large-scale labeled graphs. Neural Netw. 108, http://dx.doi.org/10.1016/j.neunet.2018.09.001.

Sutskever, I., Vinyals, O., Le, Q.V., 2014. Sequence to sequence learning with neural networks. CoRR abs/1409.3215, arXiv:1409.3215, URL http://arxiv.org/abs/1409.3215.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2016. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. 43 (1), 1–18.

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D., 2018. Deep learning similarities from different representations of source code. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories. MSR, pp. 542–553.

Ugurel, S., Krovetz, R., Giles, C.L., 2002. What's the code? automatic classification of source code archives. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 632–638.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., 2017. Graph attention networks. arXiv preprint arXiv:1710.10903.

Walden, J., Stuckman, J., Scandariato, R., 2014. Predicting vulnerable components: Software metrics vs text mining. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering. IEEE, pp. 23–33.

Wang, Y., Gao, F., Wang, L., Wang, K., 2020. Learning semantic program embeddings with graph interval neural network 1 (January). arXiv:2005.09997, URL http://arxiv.org/abs/2005.09997.

Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, IEEE, pp. 297–308.

Wang, K., Su, Z., 2019. Learning blended, precise semantic program embeddings. ArXiv abs/1907.02136.

Wei, H.-H., Li, M., 2017. 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. In: IJCAI'17, AAAI Press, pp. 3034–3040.

White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 87–98.

Xia, X., Lo, D., Wang, X., Yang, X., 2016. Collective personalized change classification with multiobjective search. IEEE Trans. Reliab. 65 (4), 1810–1829.

Xing, F., Guo, P., Lyu, M.R., 2005. A novel method for early software quality prediction based on support vector machine. In: 16th IEEE International Symposium on Software Reliability Engineering. ISSRE'05, IEEE, p. 10.

Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J., 2015. Deep learning for just-in-time defect prediction. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE, pp. 17–26.

Yao, Z., Peddamail, J.R., Sun, H., 2019. CoaCor: code annotation for code retrieval with reinforcement learning. In: The World Wide Web Conference, pp. 2203–2214.

Yu, H., Lam, W., Chen, L., Li, G., Xie, T., Wang, Q., 2019. Neural detection of semantic code clones via tree-based convolution. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, pp. 70–80. http://dx.doi.org/10.1109/ICPC.2019.00021.

Zanoni, M., Fontana, F.A., Stella, F., 2015. On applying machine learning techniques for design pattern detection. J. Syst. Softw. 103, 102–117.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 783–794.

Zhao, G., Huang, J., 2018. 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. In: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, pp. 141–151. http://dx.doi.org/10.1145/3236024.3236068.

Zhong, H., Mei, H., 2019. Learning a graph-based classifier for fault localization.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., Sun, M., 2018. Graph neural networks: A review of methods and applications. arXiv preprint arXiv:1812.08434.