Journal of King Saud University - Computer and Information Sciences 36 (2024) 101970

HOSTED BY

Contents lists available at ScienceDirect

Journal of King Saud University - Computer and Information Sciences

journal homepage: www.sciencedirect.com

Full length article

FCNN: Simple neural networks for complex code tasks

Xuekai Sun, Tieming Liu [*], Chunling Liu, Weiyu Dong

Information Engineering University, China

ARTICLE INFO

Keywords:
Program analysis
Deep learning
Code representation
Abstract syntax trees
Code classification
Code similarity detection

ABSTRACT

Program analysis using deep learning has become a focus of research, and representing code as model input is a major challenge. While abstract syntax trees (ASTs) have proven effective, using them directly as model input introduces issues of long-term dependency. Approaches based on AST paths objectively alleviate the vanishing gradient problem caused by large-scale syntax trees, but still face limitations. This paper introduces a novel approach to code analysis using fully connected neural networks (FCNN). We propose a new context structure that cleverly preserves the up-down relationships between path nodes. Simultaneously, it employs an encoding method based on node embeddings, mitigating model sparsity. Despite its simplicity, FCNN demonstrates versatility in handling various tasks associated with code analysis. Our work underscores the importance of improvements and innovations in code representation compared to using more advanced and complex deep learning models in the field of code analysis. Evaluation on two common code analysis tasks, namely code classification and code similarity detection, validates the effectiveness of the proposed approach. In code classification, FCNN achieves an F1 score of 0.45, surpassing all comparison baselines. In code similarity detection, FCNN attains an F1 score of 0.91, outperforming RtvNN and CDLH by 35.82% and 10.98%, respectively.

1. Introduction

To analyze source code effectively, the initial stage involves representing it (Chen and Monperrus, 2019; Gao et al., 2019; Li et al., 2017). The chosen method of representation plays a pivotal role in extracting maximum information from the source code and significantly influences subsequent steps, including code pre-processing and algorithm design (Li et al., 2018; Tufano et al., 2018; Wei and Li, 2017; Zhang et al., 2019). For instance, when the source code is represented as text (Roy and Cordy, 2008; Lee and Jeong, 2005) , the pre-processing procedure primarily involves filtering spaces and comments within the code. Algorithms used in text comparison are then applied for code analysis. On the other hand, if the code is represented as an AST (Jiang et al., 2007; White et al., 2016), the pre-processing stage necessitates the involvement of a programming language interpreter. The algorithm in this scenario takes into account the semantic and structural information between the code elements, enabling more effective analysis of code similarity. Hence, the selection and design of an appropriate code representation are of utmost importance in the code analysis process.

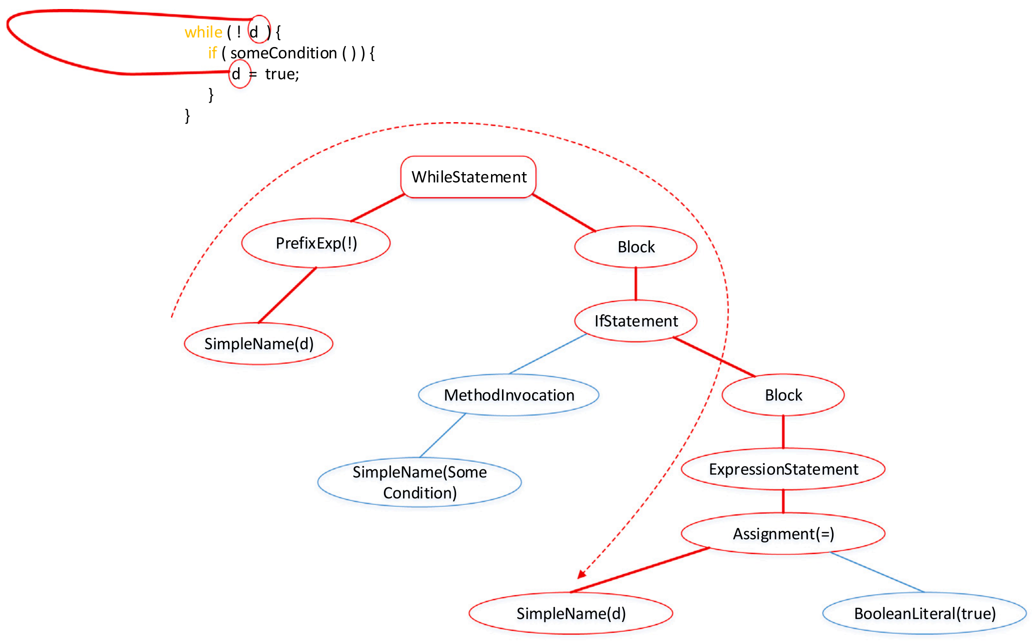
The effectiveness of using ASTs to represent code has been substantiated in various studies (Koschke et al., 2006; Fu et al., 2017; Zeng et al., 2019). In comparison to graph structures like control flow graphs and program dependency graphs (PDG) (Higo et al., 2011; Zou et al., 2020), which demand complete compilable code accessibility, ASTs offer the advantage of easy accessibility and extensibility. Nevertheless, with the expansion of code size, the corresponding syntax tree size also increases. This makes tree-based neural models more susceptible to gradient vanishing. Additionally, the conversion of ASTs to binary trees weakens the model's ability to capture semantic information about the code (Wan et al., 2018; Mou et al., 2016).

Alon et al. (2018b) proposed a path-based representation initially designed to predict program properties. We believe that this path-based modeling approach can effectively address the problems posed by the large size of syntax trees, which motivated us to conduct an in-depth study. We identified some limitations of existing path-based code representations. For instance, the literature (Alon et al., 2018b) addresses AST paths by treating them directly as ternaries, disregarding

* Corresponding author.
E-mail addresses: sunxuekai1226@163.com (X. Sun), 549855301@qq.com (T. Liu), lcl_507@163.com (C. Liu), dongxinbaoer@163.com (W. Dong).
Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2024.101970
Received 17 December 2023; Received in revised form 11 February 2024; Accepted 12 February 2024
Available online 14 February 2024



```
while ( ! d ) {
  if ( someCondition ( ) ) {
    d = true;
  }
}
```

Fig. 1. Example of extracting an AST path.

the up-down relationship between nodes. Code2vec (Alon et al., 2019) builds on this by learning code snippet embedding and generating path identifiers via a hashing algorithm. This further leads to sparsity in the model, causing an increase in training time. Code2seq (Alon et al., 2018a) adopts the concept of Neural Machine Translation (NMT) and uses a bidirectional Long Short-Term Memory (LSTM) network for path embedding. It solves the parameter sparsity problem of code2vec, but the encoder–decoder architecture increases the complexity of the model. Also, the model requires large-scale datasets and performs not well on smaller-scale datasets.

This paper introduces a neural network model grounded in AST paths. The fundamental concept involves parsing the source code into the corresponding AST, employing a path generation algorithm to derive a set of paths, and generating a vector representation of the code through a layer of fully connected neural networks. The model incorporates an innovative path-context structure that adeptly preserves the up-down relationships between nodes.

The main contributions of this paper include the following:

- We propose a code representation based on AST paths, which generates vector by representing the AST of the corresponding code as sets of paths.
- We also design a new path-context structure that contains two sub-paths and a top node as inputs to the model, which preserves the up-down relationships between path nodes.
- In order to be able to perform code similarity detection, we redesign the model by borrowing the idea of siamese network and introduce the attention mechanism.
- We have conducted extensive experimental evaluations on code classification and code similarity detection tasks respectively, including comparisons with other models, hyperparameter analysis, and data ablation experiments.

The rest of the paper is organized as follows. In Section 2, we describe the limitations of existing approaches and elaborate on our motivation for this research. In Section 3, we describe our model in detail. Section 4 describes the applications of our proposed neural model on two program comprehension tasks. Section 5 performs experimental evaluations on source code classification and source code similarity detection respectively. Section 6 presents related work. Finally, Section 7 concludes our work.

2. Background

In this section, we focus on the problems found in existing AST path-based approaches.

Fig. 1 gives a code fragment and its corresponding AST, as well as a sample example of extracting a path from the AST. The AST path corresponding to the first occurrence of variable d to its second occurrence in the source code can be represented as:

$$SimpleName \uparrow PrefixExp \uparrow WhileStatement \downarrow Block$$
$$\downarrow IfStatement \downarrow Block \downarrow$$
$$ExpressionStatement \downarrow Assignment \downarrow SimpleName$$

Here, ↑ and ↓ indicate the direction of movement of the path. We believe it contains structural information in the code and therefore needs to be retained. In contrast, existing AST path-based approaches discard these relationships and retain only the nodes when generating model inputs. We hypothesize that if the model could be given this up-down relationship information, the model's performance might improve.

Inspired by this hypothesis, we started to investigate a method to preserve the up-and-down relationships between nodes for the model. From analyzing and observing a large number of ASTs and the corresponding paths, we find that there is usually a rather special node in the paths, which we call the top node. The reason why a node is special is that it represents a change in the up and down relationships in the path. For example, in the example shown in Fig. 1, the top node is WhileStatement, the direction of travel of the path before this node is ↑, and after the node the direction of movement of the path is ↓. Therefore, we propose a novel path-context structure that contains two sub-paths and a special top node. We will introduce this structure in detail in the next section.

In addition to this, we found encoding problem in the existing approach code2vec. Specifically, code2vec utilizes a hashing algorithm to generate identifiers for paths, and this path encoding method makes the model sparse. Algorithm 1 shows the source code for the hashing algorithm used to generate path identifiers.

Let us assume that there are two paths $NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr$ and $NameExpr \uparrow AssignExpr \downarrow StringLiteralExpr$. It is easy to observe that most of the nodes in these two paths are the same, the only difference being the compositions $Integer$ and $String$ in the last node. Code2vec hashes only each character in a path, then

Algorithm 1: Method for generating path identifier

```
Input: The string of the path sequence, $S$
Output: The identifier of path sequence, $R$
1 $h = 0$;
2 for $c$ in $S$ do
3     $h = (31 * h + ord(c))\&0xFFFFFFFF$
4 end
5 $H = (h + 0x80000000)\&0xFFFFFFFF$;
6 $R = H - 0x80000000$
```

adds them up and the resulting value is entered into the model as the identity of that path. The identifiers of the two paths are "690749252" and "93831848", respectively. These two identifiers are fed directly into the model, which only knows that they are two different paths, but has no way of knowing the details of the nodes in the paths. It is difficult to get the same identifiers unless some paths have exactly the same nodes, so this approach generates a large number of identifiers, resulting in a sparse model.

Finally, code2seq introduces more complex model architecture in order to improve the accuracy of the model. This makes the model less flexible and requires better hardware support and more time for training.

Based on the above, we started this research. Our improvements were mainly focused on the code representation and to train the model we used only fully connected neural networks. We believe that this research is necessary because it solves some of the problems in the existing methods, and the simplicity of our model structure opens up the possibility of future extensions.

3. Our model

In response to the limitations observed in the existing path-based approaches, we present solutions that address these constraints. We first design a new path-context structure that preserves the contextual relationships between nodes for the model. Second, a path-context encoding method is proposed, which generates node embeddings through word2vec pre-training, and then encodes the path-context through these node embeddings, alleviating the model sparsity caused by the hash algorithm in code2vec. Finally, we used only one layer of fully-connected neural network in generating code vectors without adopting a complex model architecture, which provides the possibility of expanding the model in the future.

The workflow of our model is shown in Fig. 2. First, the code fragment C is parsed to construct the AST. Subsequently, the syntax tree is traversed to extract the paths between nodes. These paths are then represented as path-contexts and the path-contexts are encoded through an encoder. Finally, the vector representation is generated by a neural network.

3.1. AST path generation and path-context representation

In this subsection, we elucidate the process of extracting an AST path and representing it as a path-context. To facilitate a clearer understanding, we first introduce several key definitions, including AST, AST path, and path-context.

3.1.0.1 Abstract syntax tree (AST): We define the AST of a code fragment C as a tuple $\langle N, T, R, \delta \rangle$, where N represents a set of nodes; T is a set of terminal nodes; R is a collection of special nodes, with its elements referred to as top nodes; and δ is a function that maps a node to a list of its children.

3.1.0.2 AST path: Let p be an AST path with k nodes, which can be expressed in the form: $n_1 d_1 \cdots n_{r-1} d_{r-1} n_r d_r n_{r+1} d_{r+1} \cdots n_k$, where for $i \in [1 \cdots k] : n_i \in N$ is a node, and $d_i \in \{\uparrow, \downarrow\}$ denotes the movement direction. Specifically, $n_r \in R$ serves as the top node of this path. Each path has only one top node, indicating the point where the direction of the path changes. n_1 and n_k are the start and end nodes of the path, respectively. Both are terminal nodes of the AST, i.e., $n_1, n_k \in T$.

3.1.0.3 Path-context: For the AST path p, its path-context is represented as a triple $\langle p_{pre}, r, p_{post} \rangle$, where p_{pre} and p_{post} denote the two sub-paths of p respectively, and r is the top node of p. In essence, a path-context encapsulates information about two sub-paths and a special node situated between them.

The path-context corresponding to the AST path given in Fig. 1 can be represented as:

$$\Big\langle SimpleName \uparrow PrefixExp, WhileStatement, Block \downarrow$$
$$IfStatement \downarrow Block \downarrow$$
$$ExpressionStatement \downarrow Assignment \downarrow SimpleName \Big\rangle$$

It is noteworthy that we consider the node $WhileStatement$ in this path as a particularly distinctive node. This is attributed to the directional nature of relationships between nodes, where the direction is ↑ before this node and ↓ after it. We use this path-context as input to the model, whereby the inherent up-down relationships between nodes are automatically encapsulated. This path-context structure is one of the innovations of this paper, through which the model is able to capture structural information in the code more accurately.

3.2 Path-context encoding and vector generation

This subsection is dedicated to elucidating the process of encoding path-contexts into input vectors for the model. Additionally, it discusses how neural networks are employed to learn the vector representation of the code.

For a given path-context $\langle p_{pre}, r, p_{post} \rangle$, the encoding process into a context vector is outlined in Fig. 3. Each path is constructed from AST nodes, and the complete set of paths and nodes is obtained by traversing the AST. We utilize word2vec (Mikolov et al., 2013) to learn unsupervised vectors for these nodes. The vectors trained for nodes serve as the initial parameters for the encoders.

A path-context is composed of two sub-paths and a top node, and we represent these components separately. Initially, we define an embedding matrix $E^{nodes} \in \mathbb{R}^{|X| \times d}$ to signify the embedding of each node, where X denotes the set of nodes identified in the training corpus, and d represents the dimension of the node embedding. Concurrently, we maintain an index table of nodes $D^{nodes} \in \mathbb{R}^{|X|}$. When a node embedding needs to be queried, the process involves looking up its index from the index table and subsequently retrieving the corresponding row in E^{nodes} based on the index. For instance, consider the top node of the path highlighted in red in Fig. 1, denoted as $WhileStatement$. Assuming its index is i, its embedding corresponds to row i in E^{nodes}.

For the top node r, we first look up its index in the index table:

$$index(r) = D^{nodes}_r$$

Then the corresponding embedding in E^{nodes} based on this index is obtained:

$$vector(r) = E^{nodes}_{r_{index}}$$

A sub-path is composed of several nodes, and its vector can be expressed as the sum of the embeddings of the nodes that constitute it. For a given sub-path p_{sub}, the first step is to query the indexes of all nodes in the sub-path:

$$index(p_{sub}) = \{ D^{nodes}_n | \forall n \in N \}$$

where N is the set of nodes forming the sub-path p_{sub}. Subsequently, the corresponding embeddings are queried and summed based on this index set:

$$vector(p_{sub}) = \sum_{i \in index(p_{sub})} E^{nodes}_i$$

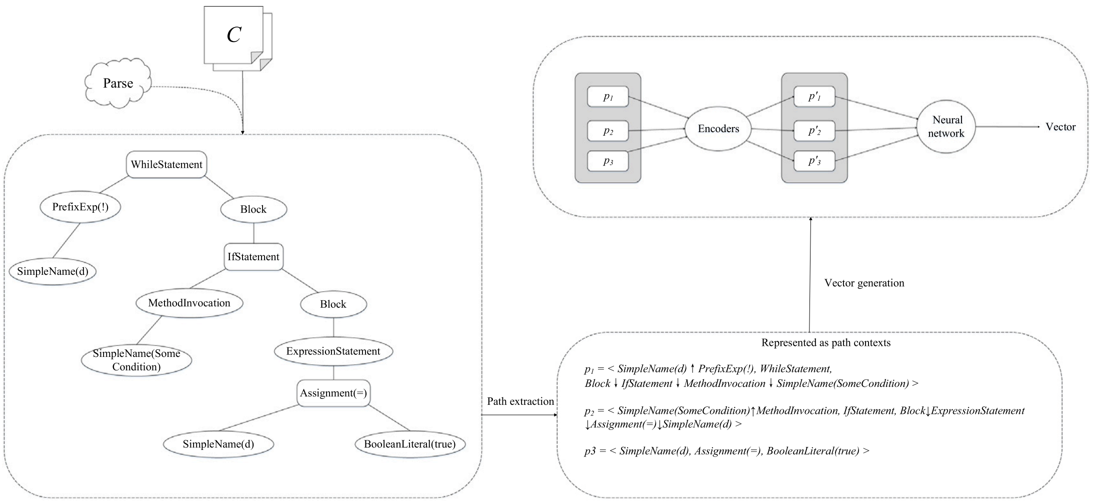

C
Parse
WhileStatement
PrefixExp(!)
Block
SimpleName(d)
IfStatement
MethodInvocation
Block
SimpleName(Some Condition)
ExpressionStatement
Assignment(=)
SimpleName(d)
BooleanLiteral(true)
Path extraction
Represented as path contexts
p_1 = < SimpleName(d) ↑ PrefixExp(!), WhileStatement, Block ↓ IfStatement ↓ MethodInvocation ↓ SimpleName(SomeCondition) >
p_2 = < SimpleName(SomeCondition)↑MethodInvocation, IfStatement, Block↓ExpressionStatement ↓Assignment(=)↓SimpleName(d) >
p3 = < SimpleName(d), Assignment(=), BooleanLiteral(true) >
Vector generation
p_1
p_2
p_3
Encoders
p'_1
p'_2
p'_3
Neural network
Vector

Fig. 2. The overall workflow of our model.

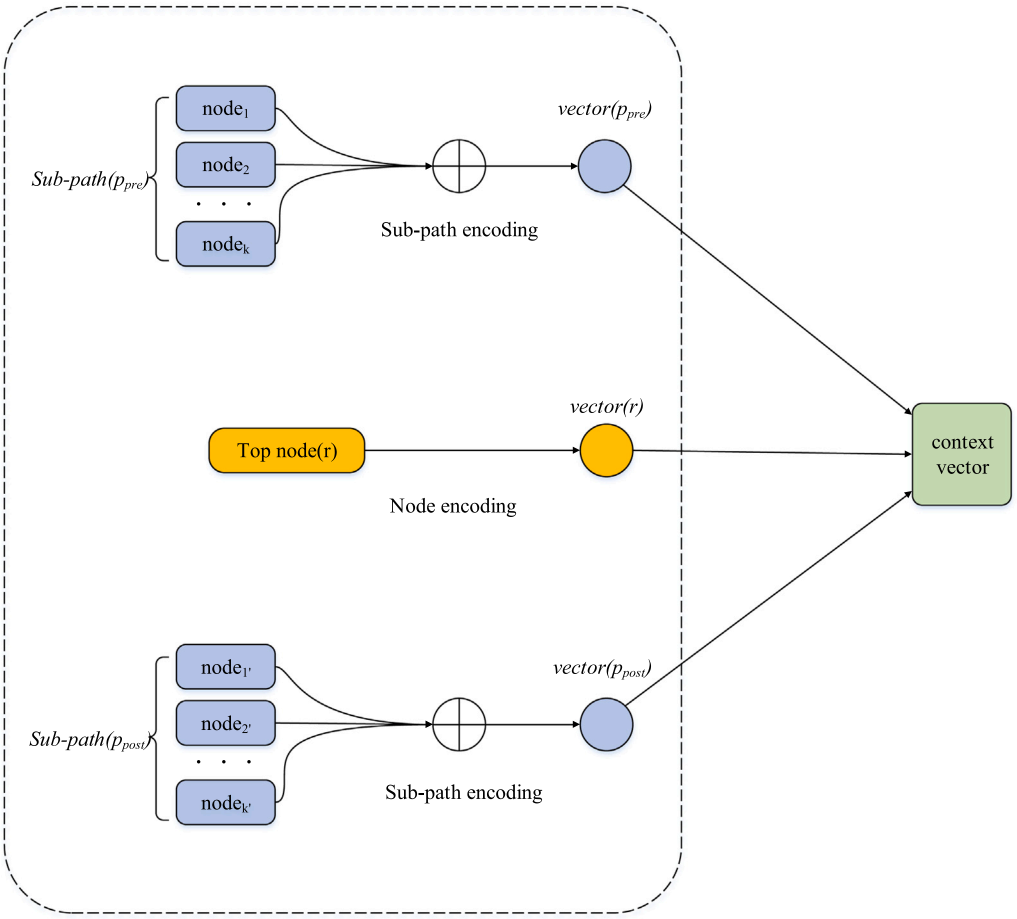


Fig. 3. Encoding path-context.

After obtaining the vectors for the two sub-paths and the top node respectively, they are concatenated sequentially to form a context vector $c \in \mathbb{R}^{3d}$:

$$c = Encoding\left(\left\langle p_{pre}, r, p_{post}\right\rangle\right)$$
$$= \left[vector\left(p_{pre}\right); vector\left(r\right); vector\left(p_{post}\right)\right]$$

Utilizing the aforementioned encoding method, we have successfully addressed the second limitation, namely, the issue of model sparsity. In the same set of data samples, we applied both the hash algorithm of code2vec and our proposed method for path encoding. The results revealed that code2vec generated 93,764 distinct identifiers, whereas our method produced only 249 node embeddings.

The process of utilizing a fully connected neural network to generate code vectors is illustrated in Fig. 4. In this depiction, the fully-connected layer encompasses a learning matrix and a bias term, which are combined with context vectors to produce code vectors.

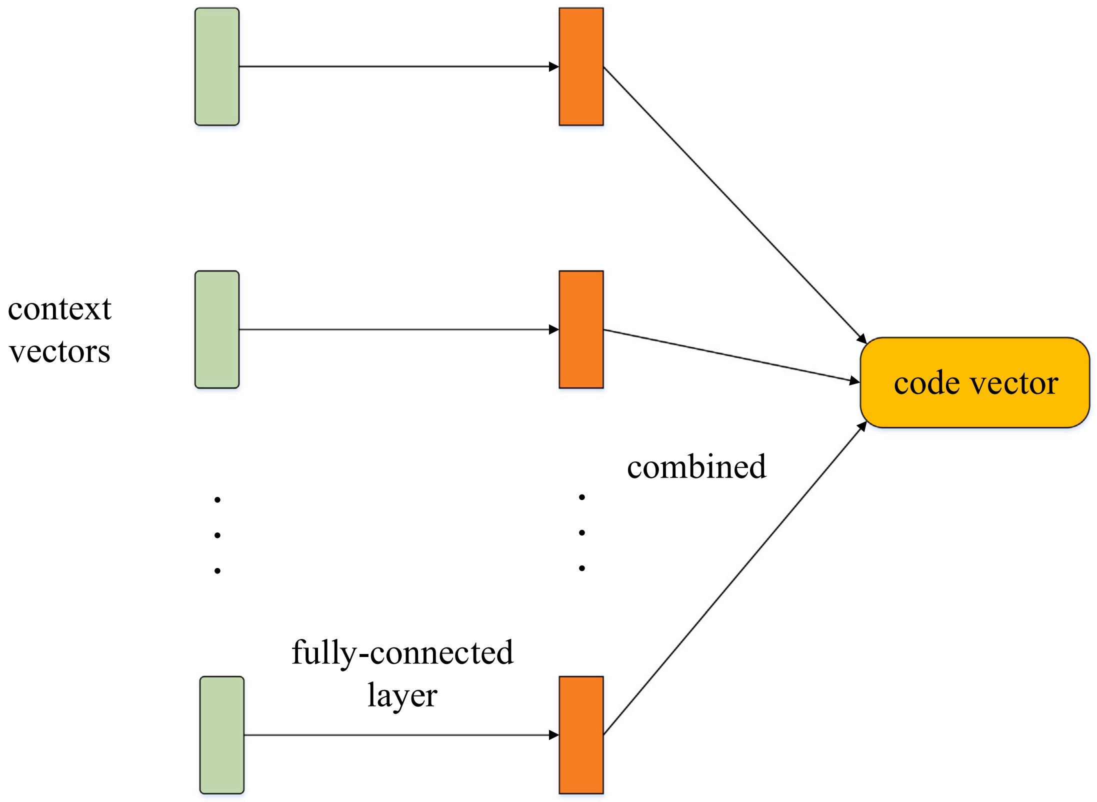


Fig. 4. Generating code vector.

For a given code snippet C, suppose h paths are extracted from its AST, resulting in a set of context vectors $\{c_1, \ldots, c_h\}$ obtained through the encoding of these paths. The output $\widetilde{c_i} \in \mathbb{R}^{3d}$ is acquired after one layer of a fully connected neural network:

$$\widetilde{c_i} = W \cdot c_i + b$$

Here $c_i, i \in [1, h]$ serves as the input for the network, $W \in \mathbb{R}^{3d\times 3d}$ represents the learning matrix, and b is a bias term. Finally, a vector representation of the code is generated by aggregating the outputs of these fully connected layers:

$$v = \sum_{i=1}^{h} \widetilde{c_i}$$

In this equation, $v \in \mathbb{R}^{3d}$ is considered a vector representation of the code fragment.

4 Applications of the proposed model

In this study, we assessed the efficacy of the proposed model by examining its ability to provide suitable source code representations for various program comprehension tasks. Our model is capable of generating a code vector based on the content of the code, and this vector can be utilized in another deep learning pipeline for a variety of tasks, such as code classification, code similarity detection, bug detection, code search, and refactoring suggestions. In this paper, we specifically focused on two prevalent tasks in software engineering, namely source code classification and source code similarity detection, to illustrate the application of the proposed model.

4.1 Source code classification

This classification task aims to categorize code snippets based on their functionality—a task frequently employed in real-world scenarios for program comprehension and maintenance (Kawaguchi et al., 2004; Linares-Vásquez et al., 2014). In this task, the model receives inputs consisting of code snippets and their corresponding names or labels. Let C represent a code snippet, and L denote the corresponding label. The fundamental assumption is that the distribution of labels can be inferred from the syntactic paths of the code snippet C. Consequently, the model endeavors to learn the distribution of labels conditioned on the code: $P(L|C)$.

Given the vector v representing the code snippet C and the set of labels Y in the training corpus, we compute the logits x as $W_l \cdot v + b_l$, where $W_l \in \mathbb{R}^{3d\times|Y|}$ is a weight matrix, and b_l is a bias term. The predictive distribution $q(y)$ of the model is defined as follows:

$$for\ y_i \in Y : q\left(y_i\right) = \frac{exp\left(x_i\right)}{\sum_{y_j\in Y} exp\left(x_j\right)}$$

Here, y_i represents row i of Y, and x_i is a component of $x \in \mathbb{R}^{|Y|}$ with index i.

The cross-entropy loss between the predictive distribution q and the true distribution p is used as the loss function. For distribution p, the true label in the training instance is assigned a value of 1, otherwise, it is assigned 0. The cross-entropy loss for a single instance is expressed as:

$$\mathcal{H}(p, q) = -\sum_{y\in Y} p(y)\, log\, q(y) = -\, log\, q\left(y_{true}\right)$$

Here, y_{true} is the true label in the sample. Since $p(y_{true})$ is 1 and all other terms are 0, the loss is essentially the negative logarithm of $q(y_{true})$. A smaller loss indicates a more accurate prediction, while a larger loss signifies less accurate predictions. Therefore, the optimization objective of the model is to minimize this loss.

The training of our model involves the use of any gradient descent algorithm. In this study, we employ the Adam optimization algorithm (Kingma and Ba, 2014), a widely used and computationally efficient adaptive gradient descent algorithm in the field of deep

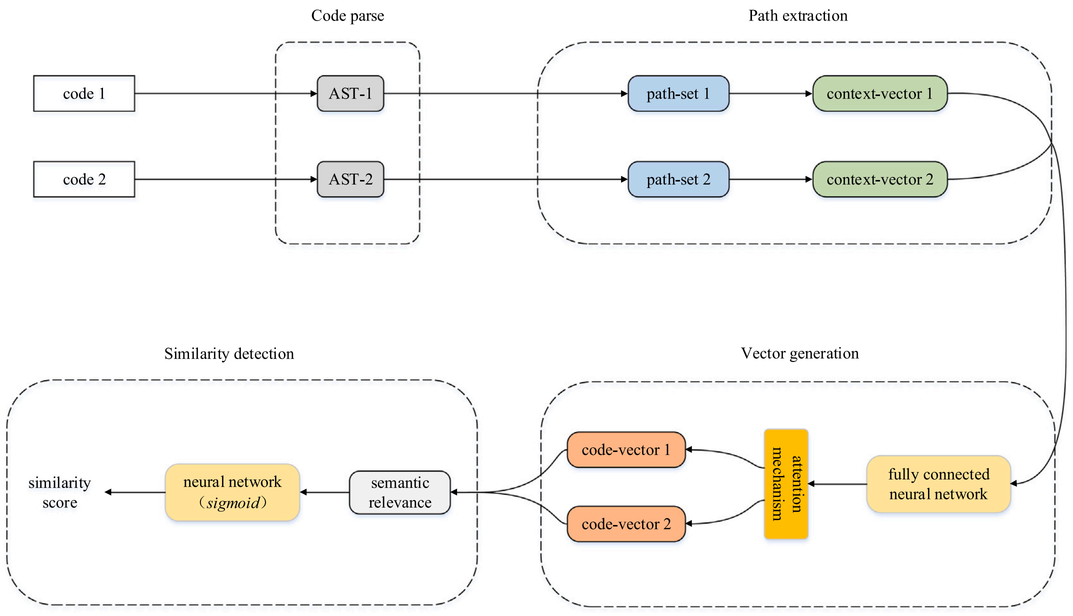


Fig. 5. The workflow of code similarity detection.

learning. Once all the parameters are optimized, the trained model is ready to predict unseen code snippets. For source code classification, predictions can be made by identifying the closest target label using the following formula:

$$prediction = argmax_{\mathcal{L}} P(\mathcal{L}|C) = argmax_{\mathcal{L}} \left\{ q_{v_C}\left(y_{\mathcal{L}}\right) \right\}$$

The predicted distribution of the code fragment C across the entire set of labels can be obtained using the above equation. The label corresponding to the maximum value in this distribution is considered the category to which the model predicts C should belong.

4.2 Source code similarity detection

The goal of code similarity detection is to determine whether a pair of code snippets is similar, and this task has garnered considerable attention in software engineering research (Kamiya et al., 2002; Sajnani et al., 2016; Keller et al., 2021).

The workflow for code similarity detection using the proposed approach is depicted in Fig. 5. In the vector generation part of this workflow, rather than directly aggregating the outputs of the neural network, we introduce an attention mechanism to learn the weight values associated with each path-context. Subsequently, the output is combined with its own weights to generate a vector representation of the code.

The introduction of the attention mechanism is motivated by the belief that each path in a code fragment is not equally important. In practice, there are typically a few crucial parts of a code fragment that determine its main function, and these parts often contain the most critical information. Experienced programmers often assess code similarity by first identifying key parts and then focusing more on those parts.

The attention mechanism aims to learn scalar weights for the elements in a given input, which are then used to perform a weighting calculation to generate a vector representation of the code. An attention vector $\boldsymbol{\alpha} \in \mathbb{R}^{3d}$ is introduced, initially randomly initialized before model training and updated during training. Given a set of neural network output vectors: $\{\widetilde{c_1}, \dots, \widetilde{c_h}\}$, each vector $\widetilde{c_i}$ corresponds to an attention weight value α_i. The weights α_i are calculated as follows:

$$attention\ weight\ \alpha_i = \frac{exp\left(\tilde{c}_i^T \cdot \boldsymbol{\alpha}\right)}{\sum_{j=1}^{n} exp\left(\tilde{c}_j^T \cdot \boldsymbol{\alpha}\right)}$$

The exponentiation in the above equation ensures that the attention weights are positive, and all weights $\{\alpha_1, \dots, \alpha_h\}$ sum to 1, effectively implementing a standard softmax function. A weighted computation of the vectors $\{\widetilde{c_1}, \dots, \widetilde{c_h}\}$ with their corresponding weight values $\{\alpha_1, \dots, \alpha_h\}$ is then performed to obtain a vector $v \in \mathbb{R}^{3d}$:

$$v = \sum_{i=1}^{h} \alpha_i \cdot \widetilde{c_i}$$

Since code similarity detection is not strictly a classification problem, we incorporate ideas from the siamese network structure (Bromley et al., 1993) and make some variations to the model. The Siamese network structure, depicted in Fig. 6, utilizes two neural networks with the same architecture and parameters. A difference layer is constructed to compute the distance between the outputs of the two networks, and finally, a fully connected layer along with a sigmoid function is employed to obtain the similarity score.

When dealing with two code fragments and a constant label y, which indicates whether the two code fragments are similar, our objective is to train a model to learn a function f that maps two codes to a similarity score. For any code pair (c_i, c_j), the similarity score between them, denoted as $s_{ij} = f(c_i, c_j)$, should closely align with the known similarity label y_{ij}.

Assuming the vectors of the two code fragments are v_i and v_j, we represent the semantic relatedness (Tai et al., 2015) between them as $\lambda = |v_i - v_j|$. This λ is then input to a fully connected layer to obtain a scalar. Subsequently, the scalar is mapped to a similarity score $s \in [0, 1]$ between the two code fragments using a sigmoid function:

$$s = sigmoid(W_l \cdot \lambda + b_l)$$

The loss function is defined as the binary cross-entropy:

$$\mathcal{H}(s, y) = \sum(-(y \cdot log(s) + (1 - y) \cdot log(1 - s)))$$

For source code similarity detection, where the similarity score between code pairs is a single value in the range of $[0, 1]$, predictions can be made as follows:

$$prediction = \begin{cases} True, & p > \delta \\ False, & p \le \delta \end{cases}$$

Here, δ is the threshold value. When the similarity is greater than this value, the code pair is predicted to be similar; otherwise, it is predicted to be dissimilar.

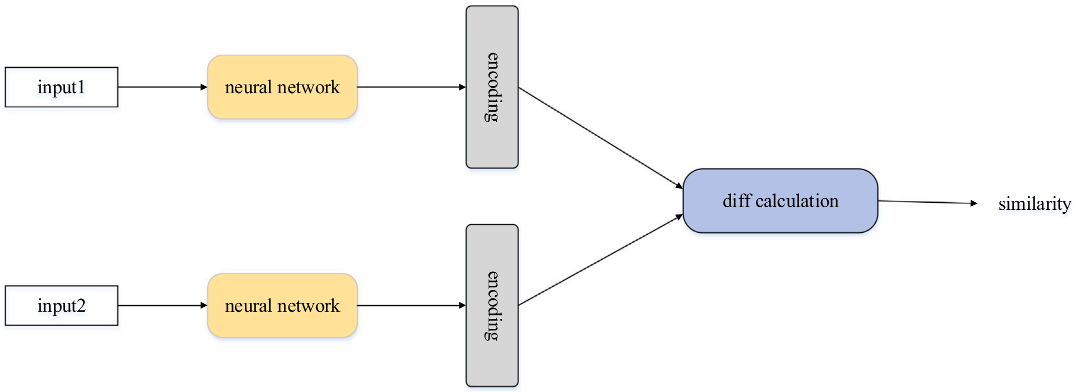


Fig. 6. Siamese network structure.

Table 1
Projects information in the small dataset.

Project name	Description	Number of files
Cassandra	Distributed Database	1786
Elasticsearch	Distributed Search and Analysis Engine	4230
Gradle	Project Automation Building System	5191
Hibernate-orm	Object-Relational Mapping Framework	6437
Intellij-community	Integrated Development Environment	41 809
Liferay-portal	Open Source Portal Framework	15 383
Presto	Distributed SQL Query Engine	2562
Spring-framework	Layered Application Framework	5603
Wildfly	Application Server	6392
Libgdx	Cross-platform Development Framework	1875
Hadoop	Distributed Systems Infrastructure	5269

Table 2
Distribution of samples in the small dataset.

Description	Number of projects	Number of samples
Training	9	665 115
Validation	1	23 505
Test	1	56 165

Table 3
Distribution of samples in the big dataset.

Description	Number of projects	Number of samples
Training	800	3 004 536
Validation	100	410 699
Test	96	411 751

5 Experiments

In this section, we conduct comprehensive experiments on two distinct software engineering tasks – source code classification and source code similarity detection – to assess the performance of the proposed model.

We aimed to address the following research questions:

- How does FCNN perform on two program comprehension tasks compared to other models? (Section 5.3)?
- How do variations in FCNN parameters, such as the number of epochs, embedding dimensions, network depth and threshold, affect the model? (Section 5.4)?
- As the input to FCNN, how does each tuple in the proposed path-context contribute to the model? (Section 5.5)?

5.1 Dataset description

In the code classification experiments, we utilized the dataset introduced in Ref. Allamanis et al. (2016), which we denote as the "Small Dataset" for distinction from other datasets. This dataset comprises samples from 11 different open-source Java projects available on GitHub. To ensure the diversity of the corpus and mitigate the impact of a substantial amount of duplicate code on GitHub (Lopes et al., 2017), the authors selected the top-ranking 11 projects. Table 1 provides details about these projects.

The dataset was originally employed for training and predicting within the same project using 11 different models. In our case, we extend the usage of the same data but without restricting training and prediction to the same project. Specifically, we use 9 projects as a training set, 1 project as a validation set, and 1 project as a test set. The dataset encompasses approximately 700 thousand samples, and the distribution of samples is illustrated in Table 2.

In order to compare FCNN with other models in more depth, and also to test the robustness and stability of the model, we conducted experiments on a much larger dataset. This dataset consists of 996 top-stared JAVA programs, of which we selected 800 for training, 100 for validation, and 96 for testing. The dataset contains about 4 million samples and we call it Big Dataset. The details of this dataset are shown in Table 3.

For the code similarity detection task, we utilize the BigCloneBench (BCB) dataset. BCB was originally introduced by Svajlenko et al. (2014), where the authors organized and annotated approximately 60 thousand code fragments for 10 different functions. This dataset comprises almost 6 million true clone samples and about 260 thousand false clone samples. Based on the similarity scores, the true clone samples are categorized into various types:

- Type-1 (T-1) and Type-2 (T-2): These samples are textually identical after proper normalization, and they both have a similarity score of 1.
- Strongly Type-3 (ST-3): Defined as samples with at least 70% similarity at the statement level, these samples have a similarity score in the range [0.7, 1). They are very similar but contain some statement-level differences compared to T-1 and T-2.
- Moderately Type-3 (MT-3): These samples have a similarity in the range [0.5, 0.7) and contain more statement-level differences. Code detectors may require a certain degree of semantic recognition capability to detect samples of this type without degrading accuracy.
- Weakly Type-3+4 (WT-3/T-4): Samples with a similarity range of [0, 0.5). This type has the largest number of samples in the dataset and is the most challenging to detect.

Table 4 provides a detailed description of the distribution of samples in the BCB dataset.

From Table 4, it is evident that the number of samples of type WT-3/T-4 accounts for more than 98% of all types, indicating that semantically similar code predominates in real software projects. From

Table 4
Distribution of samples in the bcb dataset.

Function description	True clone samples					False clone samples
	T-1	T-2	ST-3	MT-3	WT-3/T-4	
Decompress Zip	0	0	1	1	34	56
SGV Event Handling	0	0	0	0	55	1272
Java Project Creation	0	0	8	0	245	0
Init. SGV	3	7	5	2	259	78
SQL Update and Rollback	122	10	259	97	8828	24
FTP Authenticated Login	9	0	94	191	49 161	4202
Bubble Sort	43	4	239	1752	13 538	5432
Secure Hash(MD5)	632	587	3294	24 923	871 717	4564
Web Download	1544	9	1439	2715	410 611	38 838
File Copying	13 805	3116	5947	24 199	4 725 438	204 108

the BCB dataset, we extracted 20 thousand false clone samples as negative samples. For T-1, T-2, and ST-3, since the number of samples in each of them is less than 20 thousand, we extracted all of their samples as positive samples. On the other hand, for MT-3 and WT-3/T-4, which have a relatively large number of samples, 20 thousand samples were taken from each as positive samples. In the code similarity detection experiments, we divided these samples into 5 groups to assess the performance of the model on each type.

5.2 Experiment settings

The experiments were conducted on a server equipped with an Intel Xeon Gold 5120 CPU. The server has 128 GB of RAM and Ubuntu 18.04 operating system. We used Pytorch 2.0 as the software environment and Python 3.8 as the programming language version. The AST of the code is extracted using the parser provided by GumTree (Falleri et al., 2014), and word2vec (Mikolov et al., 2013) is used to initialize the embedding of the path nodes. The node embedding size is set to 128. The batch sizes on the two tasks are 64 and 32, and the maximum number of epochs is 200 and 5, respectively. We use the optimizer AdaMax (Kingma and Ba, 2014) with a learning rate of 0.002 for training. The threshold for code similarity detection is set to 0.5. The maximum number of paths is set to 200.

5.3 Comparison experiments

We conducted comparison experiments on two tasks respectively.

For the source code classification task, we compared our model against the following reference models: CNN+Attention (Allamanis et al., 2016), which employs a convolutional attention network to predict method names; Paths+CRFs (Alon et al., 2018b), which uses a syntactic path representation based on conditional random fields as a learning algorithm; code2vec (Alon et al., 2019), a neural model that represents code snippets as continuously distributed vectors; code2seq (Alon et al., 2018a), which generates path embeddings using a bidirectional LSTM and introduces an encoder–decoder architecture; Tree-LSTM (Tai et al., 2015), which is capable of processing tree-structured inputs; and Transformer (Vaswani et al., 2017), which is based on an attention mechanism.

For code similarity detection, we compared our model against the following reference models: Deckard (Jiang et al., 2007), which is a typical code similarity detection tool based on ASTs; RtvNN (White et al., 2016), which is a code similarity detection model based on recurrent neural networks; CDLH (Wei and Li, 2017), which is based on ASTs and deep learning; and ASTNN (Allamanis et al., 2014), a code analysis approach based on ASTs, which computes vectors of code by splitting the AST into smaller subtrees.

For the convenience of other researchers aiming to reproduce the experiments, we provide parameter settings for the models used in the two tasks. These settings can be used as a reference and are shown in Table 5 for the source code classification task and Table 6 for the code similarity detection task. It is worth noting that we did not reproduce some of these models, such as Paths+CRFs and CDLH. Instead, we directly cite their experimental results under the same dataset and organize the parameters provided in the related papers into the tables for readers' reference.

Table 5
Parameters of the models in the code classification.

Model name	Parameters

CNN+Attention	Conv. Dropout:0.5; Copy Dropout:0.4; Embedding Dimension:128;
	Conv. Kernel Size:8; Conv. Sliding Window Size:24,29,10;
	Copy Kernel Size:32,16; Copy Sliding Window Size:18,19,2

Paths+CRFs	Max Path Length:6; Max Path Width:4; Path Preservation:0.8
Code2vec	Epoch:20; Batch Size:1024; Max Path Number:200; Top-k:10; Embedding Size:128; Dropout:0.25

Code2seq	Epoch:3000; Train Batch Size:512; Test Batch Size:256; Max Path
	Length:9; Max Path Number:200; Patience:10;Embedding Size:128;
	RNN Size: 128 × 2; Decoder Size:320; Embedding Dropout:0.25;
	RNN Dropout:0.5

Tree-LSTM	Batch Size:25; Learning Rate:0.05; LSTM Dimensions:168; Strength for L2 Regularization:0.0001; Dropout:0.5
Transformer	Encoder–Decoder Layers:6; Output Dimension:512; Number of Linear Transformations:8; Key and Value Dimensions:64

Table 6
Parameters of the models in the code similarity detection.

Model name	Parameters
Deckard	Minimal Depth:2; Minimal Number of Nodes:3; Minimal Number of Tokens:50; Size of the Sliding Window:2; Threshold:0.9
RtvNN	Hidden Layer Size:400; Epoch:25; Learning Rate:0.003; Gradient Clipping:(−5,5); λ for L2 regularization:0.005
CDLH	Code length:32; Embedding Size:100; Encode Dimension:128; Hidden Dimension:100
ASTNN	Epoch:5; Batch Size:64; Hidden Layer Number:1; Threshold:0.5; Learning Rate:0.002

Table 7
Results of the models in the code classification.

Model name	Small dataset			Big dataset		
	P	R	F1	P	R	F1
CNN+Attention	0.50	0.25	0.33	0.61	0.27	0.37
Paths+CRFs	0.08	0.06	0.07	0.33	0.20	0.25
Code2vec	0.19	0.17	0.18	0.38	0.28	0.32
Code2seq	0.51	0.37	0.43	0.61	0.47	0.53
Tree-LSTM	0.40	0.32	0.35	0.53	0.42	0.47
Transformer	0.38	0.27	0.31	0.50	0.35	0.41
FCNN	0.51	0.40	0.45	0.57	0.45	0.50

Table 7 presents the evaluation results of the model in the code classification task, including precision, recall, and F1 scores. From the table, it is evident that our model outperforms CNN+Attention, achieving approximately 36% and 35% higher F1 scores on the two datasets, respectively. Compared to Paths+CRFs, another model based on AST paths, FCNN exhibits absolute advantages on the small database and

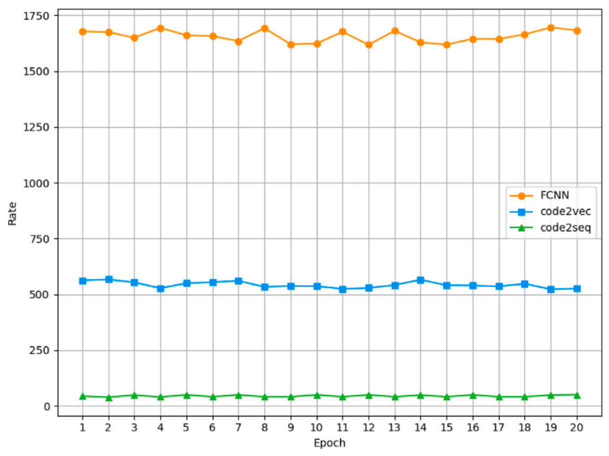


Fig. 7. Speed of the models for code classification.

Table 8
Results of the models in the code similarity detection.

Model name	P	R	F1
Deckard	0.93	0.02	0.04
RtvNN	0.76	0.59	0.67
CDLH	0.92	0.74	0.82
ASTNN	0.99	0.88	0.94
FCNN	0.97	0.86	0.91

Table 9
F1 values of the models on different types.

Model name	T-1	T-2	ST-3	MT-3	WT-3/T-4
Deckard	0.73	0.71	0.54	0.21	0.03
RtvNN	1.00	0.92	0.76	0.70	0.66
CDLH	1.00	1.00	0.94	0.88	0.81
ASTNN	1.00	1.00	0.97	0.96	0.94
FCNN	1.00	1.00	0.95	0.93	0.91

achieves a 100% higher F1 score on the large dataset. Simultaneously, it is observed that among all models, Paths+CRF performs the worst, attributed to the significant sparsity in its path and CRF models. This sparsity also leads to increased training time and memory consumption.

Despite using AST paths, Code2vec's performance is not ideal compared to FCNN. Our model achieves F1 scores of 0.45 and 0.50 on the two datasets, while Code2vec only attains 0.18 and 0.32. As mentioned earlier, Code2vec employs a simple hashing algorithm to generate path identifiers, increasing the sparsity of the model and hindering its ability to capture code information. Code2seq performs better than FCNN on the large dataset but lags behind on the small dataset. This is mainly because code2seq adopts a more complex model architecture, utilizing bidirectional LSTM to process paths and introducing an encoder–decoder framework. Therefore, with the increase in dataset size, the advantage of code2seq becomes apparent. However, Code2seq fails to address the issue of lost relationships between path nodes, and its more complex mechanisms and architecture also lead to increased sample data processing time.

Tree-LSTM captures local attribute relationships in the code, achieving F1 scores of 0.35 and 0.41 on the two datasets, surpassing most models but falling short of FCNN and Code2seq. Transformer performs poorly on the small dataset, with an F1 score of only 0.31. However, as the dataset size increases, its performance improves rapidly, reaching an F1 score of 0.41 on the large dataset. As a large-scale language model, Transformer typically requires datasets with a substantial number of samples for effective training.

Simultaneously, we observe that on the Small Dataset, FCNN achieves a precision of 0.51 and a recall of 0.40. On the Big Dataset, the precision improves to 0.57, the recall increases to 0.45, and the composite metric F1 reaches 0.50. FCNN demonstrates robust performance on a larger and potentially more complex dataset, showcasing its effectiveness across datasets of varying sizes. This suggests that the model holds practical application potential in real-world scenarios.

The processing speed results for code2vec, code2seq, and FCNN for the initial 20 epochs provide valuable insights into the efficiency of these models, as depicted in Fig. 7. It can be observed that FCNN demonstrates a significant advantage in training speed, processing over 1600 samples per second. The simplicity of FCNN's structure, without intricate mechanisms, contributes to its high processing speed.

Code2vec processes approximately 540 samples per second. Despite the relatively simple model of code2vec, the discussed sparsity issue results in longer training times. Code2seq exhibits the slowest processing speed, handling only about 44 samples per second. Although code2seq addresses the sparsity issue with bi-directional LSTM, the complexity introduced by LSTM and the encoder–decoder architecture leads to the slowest training speed among the three models.

It is noteworthy that code2vec, in its paper, claims prediction speeds that are 10 000 times and 100 times faster than CNN+Attention and Paths+CRFs, respectively, while FCNN's training speed is approximately three times faster than code2vec. Therefore, FCNN exhibits an absolute speed advantage compared to other models.

Table 8 shows the experimental comparison results of the models for code similarity detection. Table 9 further shows their distribution of F1 values on different types of samples.

From Table 8, it can be seen that Deckard has a low recall although it has a high precision. Combined with Table 9, Deckard only has an F1 score of 0.03 on samples of type WT-3/T-4, which indicates that the model is not effective in extracting the semantic information of the code. The number of samples of type WT-3/T-4 in the dataset accounts for more than 98% of all types, so Deckard can hardly detect code similarity at the semantic level, and it has the worst overall performance among all the models involved in the comparison.

From the experimental results RtvNN performs better than Deckard, although the precision is reduced compared to Deckard, the recall rate is greatly improved. From Table 9, the performance of RtvNN on all types is improved compared to Deckard. Thus it can be seen that deep learning based code detection approaches have started to show advantages. However, since RtvNN belongs to the early attempts of applying deep learning to code analysis, there are still quite a few gaps compared with later code detection models.

CDLH employs an LSTM based on AST, which has a better performance compared to RtvNN because it has a mechanism to memorize historical information. In particular, it can achieve completely accurate recognition on Type-1 and Type-2 samples, and also performs well on the other three types of samples, with F1s of 0.94, 0.88, and 0.81, respectively. This indicates that the deep learning-based model can effectively extract relevant syntactic features from the AST and can achieve good results in code similarity detection at the semantic level.

As one of the state-of-the-art code representations, ASTNN performs best among all the models involved in the comparison. As can be seen in Table 8, the F1 score of ASTNN has reached 0.94, which is a 16% improvement compared to CDLH. The core idea of ASTNN is to split the AST into smaller subtrees, which makes it possible to extract more precise functional information from the code, while also mitigating the effects of vanishing model gradients. As can be seen from Table 9, ASTNN obtains F1 scores of more than 0.90 on all five different types. Particularly, for WT-3/T-4, which accounts for more than 98% of the total, it can still have an F1 score of 0.94, which indicates that the approach can effectively recognize the semantic similarity of the codes, and meets the requirement of being a high-precision code similarity detection tool.

In the results presented in Table 8, FCNN achieves an F1 score of 0.91 on the BCB. This marks an improvement of approximately

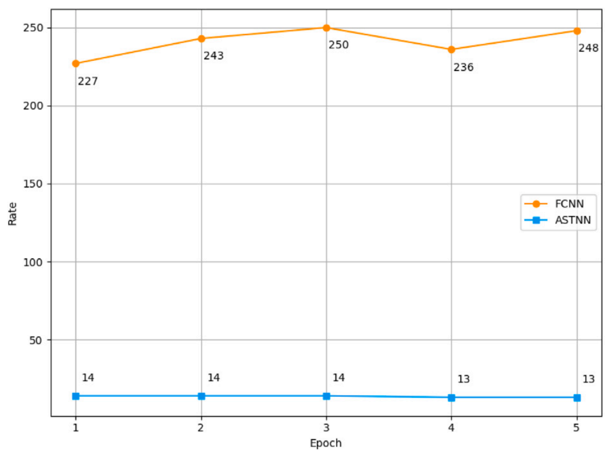


Fig. 8. Speed of FCNN and ASTNN for code similarity detection.

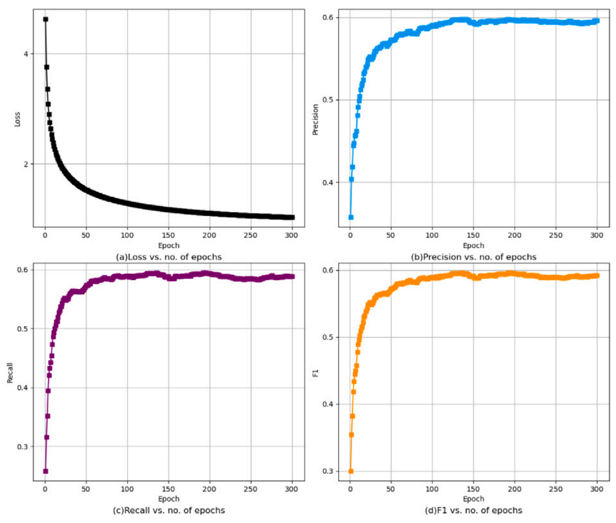


Fig. 9. Distribution of results with epoch in code classification.

36% and 11% compared to RtvNN and CDLH, respectively. Table 9 demonstrates that our model, similar to ASTNN, attains F1 scores exceeding 0.90 across all types. Overall, FCNN exhibits superior performance among all the models compared, except ASTNN. Although FCNN's F1 score is marginally lower than that of ASTNN, our model displays higher responsiveness in processing samples. Fig. 8 illustrates the sample processing speeds of FCNN and ASTNN in code similarity detection. As shown in the figure, FCNN processes over 220 samples per second, while ASTNN processes only 13–14 samples per second. The model efficiency of FCNN is at least 17 times greater than that of ASTNN. Therefore, the advantage of FCNN over ASTNN is its ability to train models quickly, which shows great potential for scalability.

5.4 Hyperparameter analysis

In order to further evaluate the impact of parameters on the model, we conducted more extensive experiments, which include epochs, embedding dimensions, network depth and threshold. Since these experiments do not involve comparisons with other models, in order to improve the efficiency of the experiments, we took 10,856 samples from the original dataset when performing the code classification. Among these, 8706 samples constituted the training set, while the remaining 2150 samples formed the test set. The dataset used for code similarity detection remains unchanged. The experiments in this section exclusively involve the adjustment of parameter values that are currently under evaluation, while keeping all other parameter settings of the model constant.

We first research the effect of epoch on model performance. Since our model can accomplish code similarity detection in a very short epoch, we focus on code classification. The model was trained for 300 epochs, and the loss, precision, recall, and F1 were tested and recorded after every each epoch. The evolution of the model's performance across epochs is illustrated in Fig. 9.

After the initial training epoch, the model exhibited a high loss of 4.63, indicating significant training errors in the initial stages. At this point, the precision, recall, and F1 score were 0.36, 0.26, and 0.30, respectively. Following 20 epochs of training, the loss reduced to 1.96, signifying a gradual reduction in errors during the learning process. Concurrently, the model achieved notable improvements in precision, recall, and F1 score, each reaching 0.54. This suggests successful learning of relevant code information, leading to enhanced performance.

After approximately 120 epochs, the loss further decreased to 1.22, and the F1 score reached 0.59, with corresponding precision and recall values of 0.60 and 0.59. This indicates substantial improvements in both loss and performance metrics. Subsequently, the model's metrics no longer exhibited significant improvement with increasing epochs. Observing Fig. 9, it is evident that the model experienced notable enhancement in the initial 20 epochs, and performance improvement slowed down after approximately 50 epochs, ceasing to improve significantly around epoch 135. This observation suggests that an appropriate number of epochs should not exceed 200. To achieve optimal performance, it is recommended to search for and save the model within the range of 120 to 135 epochs.

Table 10
Performance of fcnn on two tasks with different embedding dimensions.

Dimension	Task1			Task2		
	P	R	F1	P	R	F1
32	0.52	0.53	0.53	0.89	0.80	0.84
64	0.56	0.57	0.56	0.94	0.84	0.89
128	0.60	0.59	0.59	0.97	0.86	0.91
256	0.59	0.59	0.59	0.97	0.86	0.91
512	0.60	0.58	0.59	0.97	0.86	0.91

Subsequently, we evaluated the impact of different embedding dimensions on the model. In these experiments, the embedding dimensions were set to 32, 64, 128, 256, and 512, while keeping other settings constant. The results are detailed in Table 10, where Task 1 and Task 2 represent code classification and code similarity detection, respectively.

When the node embedding dimension was set to 32, the model's performance on code classification exhibited precision of 0.52, recall of 0.53, and an F1 value of 0.53. In code similarity detection, the corresponding results were 0.89, 0.80, and 0.84. As the node embedding dimension increased, the model's performance gradually improved in both tasks. With a dimension of 64, the model achieved 0.56, 0.57, and 0.56 in code classification, and 0.94, 0.84, and 0.89 in code similarity detection. Compared to the dimension of 32, the F1 values of the model in both tasks increased by 5.66% and 5.95%, respectively. With a dimension of 128, the performance further improved, reaching 0.60, 0.59, 0.59 (code classification), and 0.97, 0.86, 0.91 (code similarity detection).

For dimensions 256 and 512, the model's performance remained relatively stable, with no significant changes observed in both tasks. From the experiments, it is evident that when the node embedding dimension is low, the embedding vector cannot carry sufficient code information, leading to decreased model performance. As the dimension increases, the model's performance also improves. However, when the dimension exceeds 128, it is no longer a limiting factor for the model's

Table 11
Performance of fcnn on two tasks with different network depth.

Depth	Task1			Task2		
	P	R	F1	P	R	F1
1	0.60	0.59	0.59	0.97	0.86	0.91
2	0.59	0.59	0.59	0.97	0.86	0.91
3	0.59	0.59	0.59	0.97	0.86	0.91

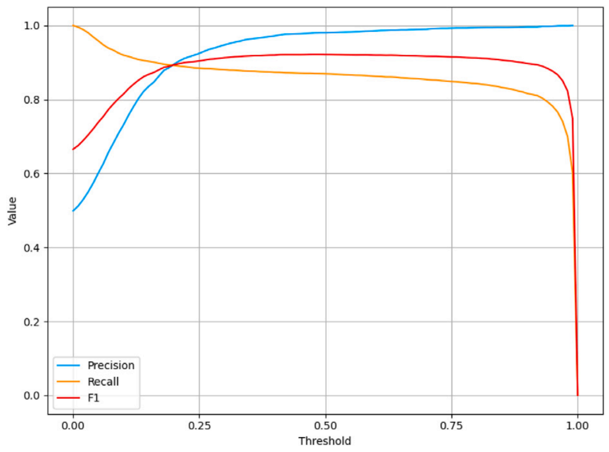


Fig. 10. Performance of FCNN in different thresholds.

performance. Moreover, increasing the dimension will inevitably increase the model's computational cost. Therefore, considering both the performance and computational efficiency of the model, an appropriate embedding dimension should be set to 128.

The FCNN employs a single-layer fully connected neural network to generate code vectors. To investigate the impact of network depth on model performance, we experimented with varying the number of network layers (1, 2, and 3) for both code classification and code similarity detection tasks. The results are summarized in Table 11. From the table, it is observed that when the network has only one layer, the model achieves a performance of precision 0.60, recall 0.59, and F1 score 0.59 for code classification, and results of 0.97, 0.86, and 0.91 for code similarity detection. However, when the network has 2 or 3 layers, the model's performance in both tasks does not show improvement, yielding results of 0.59, 0.59, 0.59 (code classification), and 0.97, 0.86, 0.91 (code similarity detection).

The analysis indicates that in this experiment, increasing the network depth did not significantly enhance the model's performance. The reason may lie in the fact that our model employs only fully-connected neural networks, which are linear combinations by nature, and multiple fully-connected layers will essentially only perform linear transformations in the absence of a nonlinear activation function, failing to capture more complex nonlinear patterns. Based on the experimental findings, it can be inferred that, for FCNN, simply increasing the network depth does not improve model performance significantly, and therefore, a single-layer fully connected neural network is more suitable.

To analyze the impact of different similarity thresholds on FCNN, we conducted code similarity detection experiments by varying the threshold value. Since the range of similarity is [0,1], the threshold value should also vary within this range.

Fig. 10 illustrates how the model's precision, recall, and F1 score change with varying thresholds. When the threshold is set to 0, the precision is 0.50, recall is 1.00, and F1 score is 0.67. According to the definition of the threshold, when the predicted similarity is greater than the threshold, the model considers two code segments as similar. Therefore, when the threshold is 0, the model classifies all samples as similar (all are classified as positive samples).

Table 12
Our model while hiding input components.

Description	Input	Task1			Task2		
		P	R	F1	P	R	F1
Only-path	$\langle p_{pre}, -, p_{post}\rangle$	0.57	0.56	0.56	0.94	0.81	0.87
No-paths	$\langle -, r, -\rangle$	0.35	0.27	0.31	0.81	0.68	0.74
Pre-path	$\langle p_{pre}, -, -\rangle$	0.48	0.47	0.47	0.87	0.75	0.81
Post-path	$\langle -, -, p_{post}\rangle$	0.51	0.50	0.51	0.90	0.79	0.84
Full	$\langle p_{pre}, r, p_{post}\rangle$	0.60	0.59	0.59	0.97	0.86	0.91

Precision represents the proportion of true positive samples among those classified as positive, and since our samples have a balanced distribution of positive and negative samples, the precision is close to 0.5. Recall represents the proportion of true positive samples among all actual positive samples, resulting in a recall of 1. As the threshold gradually increases, precision increases, recall decreases, and the F1 score increases, indicating an overall improvement in the model's performance with increasing thresholds. At a threshold of 0.25, precision and recall are 0.92 and 0.88, respectively, yielding an F1 score of 0.90. At a threshold of 0.50, the F1 score reaches its highest point at approximately 0.91, with precision rising to around 0.97 and recall at 0.86. Beyond this point, the F1 score no longer increases.

When the threshold reaches 0.75, the F1 score slightly decreases, but precision has already reached 0.99, while recall decreases to 0.84. Beyond a threshold of 0.75, the comprehensive evaluation metric, F1 score, gradually declines. When the threshold exceeds 0.95, recall rapidly decreases, leading to a swift reduction in the F1 score. In summary, within the range [0.2, 0.9], the changes in precision, recall, and F1 score are relatively smooth, reaching the highest F1 score in this range. This smooth region occupies a substantial portion of the [0, 1] range, suggesting that minor variations in the threshold do not significantly impact the model's performance. It is observed that F1 generally reaches its peak around a threshold of 0.5, indicating that the optimal threshold is 0.5.

5.5 Data ablation

In order to assess the contribution of each component of the path-context presented in this paper, we conducted an ablation study. We varied our model by "hiding" one or more input positions in code classification and code clone detection experiments, respectively, and recorded the variations in performance.

The "full" representation is denoted as $\langle p_{pre}, r, p_{post}\rangle$. The following experiments were conducted:

- "only-path" - using only two sub-paths as model input, without the top node connecting them. Each path-context is represented as: $\langle p_{pre}, -, p_{post}\rangle$.
- "no-paths" - using only the top node: $\langle -, r, -\rangle$, without two sub-paths.
- "pre-path" - using only the pre-path as input: $\langle p_{pre}, -, -\rangle$.
- "post-path" - using only the post-path as input: $\langle -, -, p_{post}\rangle$.

Table 12 presents the performance of the proposed model when hiding different input components. The results indicate that the "full" input achieves the best performance, suggesting that all input components contribute accordingly. In comparison, the "only-path" input performs slightly worse, implying that incorporating the top node as one of the input components positively impacts the model's performance. The "no-path" input, which entirely abandons sub-paths and relies only on the top node, yields the poorest results. This emphasizes that discarding syntactic paths has a more detrimental effect on the model than excluding the critical node.

When comparing the "pre-path" and "post-path", it is observed that the latter exhibits superior performance than the former, suggesting that p_{post} contributes more to the model. A more in-depth analysis of the samples reveals that the number of nodes comprising p_{post} is generally greater than those constituting the p_{pre}. This discrepancy arises from the fact that nodes in the pre-path often correspond to relevant parameter or condition information in the source code, while the post-path typically aligns with code blocks.

For instance, in the example illustrated in Fig. 1, the pre-path can be represented as $SimpleName \uparrow PrefixExp$, corresponding to the condition for executing the $while$ loop. In contrast, the post-path is represented as $Block \downarrow IfStatement \downarrow Block \downarrow ExpressionStatement \downarrow Assignment \downarrow SimpleName$, corresponding to the content of the code block within the $while$ loop. Obviously, code blocks usually contain more information than parameters or conditions. Thus, we observe that inputs containing more information have a greater impact on the model, an observation that is consistent with common sense. These findings help us understand the prediction process and key decision factors of FCNN more clearly, and also improve the interpretability of FCNN to some extent.

6 Related work

Effectively representing source code is a crucial challenge in advancing code analysis.

In paper (Hindle et al., 2016), the authors argue that programming languages share similarities with natural languages in that both are composed of words and exhibit contextual, semantic, and logical relationships. First, the authors draw inspiration from natural language research and introduce NLP methods into the field of programming language research, making the assumption that most software is equally repetitive and predictable. Then, using the modeling approach of statistical language models, the widely adopted n-gram model is used to successfully prove the regularity of the code, further affirming the similarity between programming languages and natural languages. Finally the authors also develop a simple Java code completion engine that demonstrates the potential for practical application of statistical methods in the field of software engineering.

Mou et al. (2016) proposed that programs contain rich, explicit and complex structural information and traditional NLP models may not be suitable for programs. To overcome this challenge, they introduced a novel Tree-Based Convolutional Neural Network (TBCNN), in which the convolutional kernel is specifically designed to capture structural information in the AST of a program. TBCNN outperforms the baseline approaches at the time, including several neural models commonly used in NLP. The main contribution of TBCNN is to introduce a novel and efficient neural network structure to the field of programming language processing, which opens up new possibilities for the application of deep learning in program analysis. However, the method converts ASTs into continuous full binary trees, which alters the original semantics of the code and makes the long-term dependency problem worse.

In order to solve the above problem, ASTNN (Zhang et al., 2019) proposes an AST-based code representation method. The method decomposes a large AST into a series of smaller subtrees, then encodes these subtrees into vectors, and finally utilizes an RNN model to capture the semantic information of the code and generate a vector representation of the code. The contribution of ASTNN is that it innovatively proposes a segmentation method for ASTs, and experimentally proves the validity of this method. Facing the problem of gradient vanishing that may be caused by large-scale ASTs, the ASTNN provides useful ideas to solve this difficulty.

Paper (Alon et al., 2018b) introduces a code representation method based on AST paths. The main idea is to utilize paths in the AST to represent programs, enabling learning models to leverage the structural nature of the code rather than treating it as a flat sequence of tokens. In experimental evaluations, this method demonstrates significant performance advantages in tasks such as predicting program properties. The contribution of this research is to provide an innovative path-based code representation that proposes a novel research direction for code analysis. Since path extraction is very laborious, in order to allow researchers to focus on more important research tasks, Vladimir et al. (2019) proposed PathMiner, an open-source software dedicated to generating AST paths. The contribution of PathMiner is that it simplifies the path mining process by making it more efficient, flexible, and easily extensible to a variety of downstream code analysis tasks. It provides researchers with greater flexibility and convenience, allowing them to focus more on the research of code analysis techniques themselves.

Code2vec (Alon et al., 2019) is a neural network model for learning code representations, mapping code snippets to fixed-length code vectors capable of predicting the semantic properties of code fragments. Specifically, the model decomposes code into a collection of paths in its AST, then employs a neural network to learn embeddings for each path and aggregates these path embeddings into a vector representation of the code. Code2vec demonstrates excellent performance in predicting method names in software analysis tasks. However, as mentioned earlier, it discards the relationships between path nodes, and its use of a hash algorithm to generate code path identifiers leads to model sparsity.

Code2seq (Alon et al., 2018a) is also a code representation model based on AST paths. Unlike code2vec, it employs an encoder–decoder architecture for learning code representations and introduces LSTM to encode AST paths. Compared to code2vec, code2seq performs better in tasks such as predicting method names. However, code2seq similarly does not improve the context structure of paths, and thus the model cannot capture the relationships between path nodes. Therefore, we introduce a novel path context structure designed to effectively preserve the up-dowm relationships between path nodes. And we propose a node embedding-based path encoding method to alleviate model sparsity. Compared to code2vec, our model exhibits superior performance. In comparison to code2seq, our model has a simpler architecture and less processing time, making it more suitable for scenarios with high time efficiency requirements.

In addition to the above approaches, code analysis methods based on the transformer architecture have recently attracted a lot of attention. Devlin et al. (2018) introduced BERT, a revolutionary pre-trained model. BERT achieved outstanding performance in natural language processing tasks by incorporating innovations such as bidirectional context modeling, the transformer architecture, unsupervised pre-training, and masked language modeling. Feng et al. (2020) pioneered the integration of BERT into source code representation, introducing the CodeBERT pre-trained model. Unlike NLP pre-trained models, CodeBERT leverages both multimodal data consisting of code and corresponding comments and unimodal code data, achieving remarkable results.

CodeT5 (Wang et al., 2021) is a unified pre-trained encoder–decoder transformer model that better leverages the code semantics conveyed by developer-assigned identifiers. In contrast to most current methods relying on encoder-only (or decoder-only) pre-training, this model adopts a unified framework supporting both code understanding and generation tasks, allowing for multi-task learning. Salza et al. (2022) extended BERT to cross-language code search, pre-training it on multiple languages like Python, Java, PHP, and Go, and fine-tuning the code search model on languages like Ruby not seen during pre-training. The results show that cross-language code search performs better than training from scratch in a single language, validating the effectiveness of transfer learning in code search.

Several excellent code analysis methods have also emerged recently. SCDetector (Wu et al., 2020) is a method employed for detecting software functional clones. Specifically, it begins with the extraction of a control flow graph through static analysis, treating it as a social network. The application of social network centrality analysis is then used to unveil the centrality of each basic block. Subsequently, the centrality is assigned to each token within a basic block, and the centralities of identical tokens across different basic blocks are aggregated. Through this methodology, the graph is transformed into specific tokens with graph details, specifically centrality, referred to as semantic tokens. Ultimately, these semantic tokens are fed into a Siamese architecture neural network for training a code clone detector. SCDetector innovatively treats the graph as a social network, applying social network centrality analysis to reveal the centrality of each basic block, thereby integrating the scalability of token-based methods with the accuracy of graph-based methods.

FCCA (Hua et al., 2020) is a deep learning-based method designed for detecting functional code clones with a focus on preserving multiple code features. The innovation of FCCA is that it combines multiple features consisting of tokens, ASTs and CFGs, with the use of an attention mechanism to merge them into a code representation. The approach was implemented and evaluated on 275,777 real-world code clone pairs written in Java, and the experimental results demonstrate that FCCA outperforms various code analysis methods, including TBCNN.

Amain Wu et al. (2022) is an extensible tree-based semantic code clone detector that creatively employs a Markov chain model for code clone detection. Specifically, Amain transforms the original complex tree into simple Markov chains and measures the distance between all states in these chains. Subsequently, all distance values are fed into a machine learning classifier to train the code clone detector. Evaluation on widely used datasets such as Google Code Jam and BigCloneBench demonstrates that Amain outperforms advanced methods including TBCNN, ASTNN, and SCDetector in the realm of code clone detection. TreeCen (Hu et al., 2022) is another method for code clone detection that utilizes social network centrality. In contrast to SCDetector, TreeCen constructs a tree graph based on ASTs rather than extracting the control flow graph. Through centrality analysis on the tree graph, TreeCen converts the intricate syntax tree into a 72-dimensional vector containing AST structural information, which is then used to train the detector. The authors have substantiated through experiments that this approach outperforms SCDetector and ASTNN in code clone detection.

7 Conclusion

In this paper, we present a neural model designed for learning code representations. Our approach involves traversing the AST of a code snippet to extract a set of paths, subsequently learning vectors for each path. The proposed model is applied to two prevalent program understanding tasks: source code classification and source code clone detection. The experimental results demonstrate the effectiveness of our model, showcasing its good performance in both source code classification and code clone detection. In code classification, FCNN outperforms all baseline models. Similarly, in code similarity detection, FCNN surpasses all models except ASTNN. Compared to ASTNN, FCNN excels in its ability to rapidly train model, with 17 times the training speed of ASTNN, demonstrating its great potential for scalability.

FCNN exhibits potential for integration with established code analysis methodologies. An illustrative example is its collaboration with code2vec, a method included in our comparative analysis. Such integration has the potential to mitigate the limitations of code2vec, thereby augmenting its efficacy. Furthermore, the amalgamation of our proposed method with sophisticated machine learning models holds the prospect of significantly enhancing algorithmic performance. The integration of this model into existing development tools can furnish software developers with expedited insights into code functionality, consequently contributing to heightened development efficiency. This underscores the versatility and applicability of FCNN in the realm of code analysis and software development.

While our model has shown promising results, we acknowledge two key challenges: (1) the necessity for regular training on updated datasets to accommodate new nodes; and (2) the model's dependence on tokens, which may be susceptible to misguidance if variable names lack informativeness or are obfuscated. To address these challenges, a potential solution involves training the model on a diverse dataset that includes both well-informative and intentionally obscured variable names. This approach aims to reduce the model's reliance on variable names and represents one of the future directions for our work.

Furthermore, due to the AST-based nature of our approach, it inherently extends to various programming languages. We plan to conduct further assessments of the proposed model on datasets encompassing multiple programming languages in future research, aiming to explore its generality. This strategic approach is intended to provide a comprehensive understanding of the model's performance across diverse coding contexts and linguistic environments. Through these forthcoming investigations, we anticipate gaining deeper insights into the versatility and applicability of our method, ensuring its broad effectiveness in practical applications. We look forward to contributing valuable insights and effective solutions to both the academic and practical domains through this extended exploration.

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.

References

Allamanis, Miltiadis, Barr, Earl T., Bird, Christian, Sutton, Charles, 2014. Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 281–293.

Allamanis, Miltiadis, Peng, Hao, Sutton, Charles, 2016. A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning. PMLR, pp. 2091–2100.

Alon, Uri, Brody, Shaked, Levy, Omer, Yahav, Eran, 2018a. Code2seq: Generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400.

Alon, Uri, Zilberstein, Meital, Levy, Omer, Yahav, Eran, 2018b. A general path-based representation for predicting program properties. ACM SIGPLAN Not. 53 (4), 404–419.

Alon, Uri, Zilberstein, Meital, Levy, Omer, Yahav, Eran, 2019. Code2vec: Learning distributed representations of code. Proc. ACM Programm. Lang. 3 (POPL), 1–29.

Bromley, Jane, Guyon, Isabelle, LeCun, Yann, Säckinger, Eduard, Shah, Roopak, 1993. Signature verification using a "siamese" time delay neural network. In: Advances in Neural Information Processing Systems, vol. 6.

Chen, Zimin, Monperrus, Martin, 2019. A literature study of embeddings on source code. arXiv preprint arXiv:1904.03061.

Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, Toutanova, Kristina, 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Falleri, Jean-Rémy, Morandat, Floréal, Blanc, Xavier, Martinez, Matias, Monperrus, Martin, 2014. Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 313–324.

Feng, Zhangyin, Guo, Daya, Tang, Duyu, Duan, Nan, Feng, Xiaocheng, Gong, Ming, Shou, Linjun, Qin, Bing, Liu, Ting, Jiang, Daxin, et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Fu, Deqiang, Xu, Yanyan, Yu, Haoran, Yang, Boyang, et al., 2017. WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection. Sci. Program. 2017.

Gao, Yi, Wang, Zan, Liu, Shuang, Yang, Lin, Sang, Wei, Cai, Yuanfang, 2019. TECCD: A tree embedding approach for code clone detection. In: 2019 IEEE International Conference on Software Maintenance and Evolution. ICSME, ieee, pp. 145–156.

Higo, Yoshiki, Yasushi, Ueda, Nishino, Minoru, Kusumoto, Shinji, 2011. Incremental code clone detection: A PDG-based approach. In: 2011 18th Working Conference on Reverse Engineering. IEEE, pp. 3–12.

Hindle, Abram, Barr, Earl T., Gabel, Mark, Su, Zhendong, Devanbu, Premkumar, 2016. On the naturalness of software. Commun. ACM 59 (5), 122–131.

Hu, Yutao, Zou, Deqing, Peng, Junru, Wu, Yueming, Shan, Junjie, Jin, Hai, 2022. TreeCen: Building tree graph for scalable semantic code clone detection. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–12.

Hua, Wei, Sui, Yulei, Wan, Yao, Liu, Guangzhong, Xu, Guandong, 2020. Fcca: Hybrid code representation for functional clone detection using attention networks. IEEE Trans. Reliab. 70 (1), 304–318.

Jiang, Lingxiao, Misherghi, Ghassan, Su, Zhendong, Glondu, Stephane, 2007. Deckard: Scalable and accurate tree-based detection of code clones. In: 29th International Conference on Software Engineering. ICSE'07, IEEE, pp. 96–105.

Kamiya, Toshihiro, Kusumoto, Shinji, Inoue, Katsuro, 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Eng. 28 (7), 654–670.

Kawaguchi, Shinji, Garg, Pankaj K, Matsushita, Makoto, Inoue, Katsuro, 2004. Mudablue: An automatic categorization system for open source repositories. In: 11th Asia-Pacific Software Engineering Conference. IEEE, pp. 184–193.

Keller, Patrick, Kaboré, Abdoul Kader, Plein, Laura, Klein, Jacques, Le Traon, Yves, Bissyandé, Tegawendé F, 2021. What you see is what it means! semantic representation learning of code based on visualization and transfer learning. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (2), 1–34.

Kingma, Diederik P., Ba, Jimmy, 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Koschke, Rainer, Falke, Raimar, Frenzel, Pierre, 2006. Clone detection using abstract syntax suffix trees. In: 2006 13th Working Conference on Reverse Engineering. IEEE, pp. 253–262.

Lee, Seunghak, Jeong, Iryoung, 2005. SDD: High performance code clone detection system for large scale source code. In: Companion To the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 140–141.

Li, Liuqing, Feng, He, Zhuang, Wenjie, Meng, Na, Ryder, Barbara, 2017. Cclearner: A deep learning-based clone detection approach. In: 2017 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 249–260.

Li, Zhen, Zou, Deqing, Xu, Shouhuai, Ou, Xinyu, Jin, Hai, Wang, Sujuan, Deng, Zhijun, Zhong, Yuyi, 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

Linares-Vásquez, Mario, McMillan, Collin, Poshyvanyk, Denys, Grechanik, Mark, 2014. On using machine learning to automatically classify software applications into domain categories. Empir. Softw. Eng. 19, 582–618.

Lopes, Cristina V., Maj, Petr, Martins, Pedro, Saini, Vaibhav, Yang, Di, Zitny, Jakub, Sajnani, Hitesh, Vitek, Jan, 2017. Déjàyu: A map of code duplicates on GitHub. Proc. ACM Programm. Lang. 1 (OOPSLA), 1–28.

Mikolov, Tomas, Sutskever, Ilya, Chen, Kai, Corrado, Greg S., Dean, Jeff, 2013. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, vol. 26.

Mou, Lili, Li, Ge, Zhang, Lu, Wang, Tao, Jin, Zhi, 2016. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 30.

Roy, Chanchal K., Cordy, James R., 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: 2008 16th IEEE International Conference on Program Comprehension. IEEE, pp. 172–181.

Sajnani, Hitesh, Saini, Vaibhav, Svajlenko, Jeffrey, Roy, Chanchal K., Lopes, Cristina V., 2016. Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1157–1168.

Salza, Pasquale, Schwizer, Christoph, Gu, Jian, Gall, Harald C., 2022. On the effectiveness of transfer learning for code search. IEEE Trans. Softw. Eng..

Svajlenko, Jeffrey, Islam, Judith F., Keivanloo, Iman, Roy, Chanchal K., Mia, Mohammad Mamun, 2014. Towards a big data curated benchmark of inter-project code clones. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 476–480.

Tai, Kai Sheng, Socher, Richard, Manning, Christopher D., 2015. Improved semantic representations from tree-structured long short-term memory networks. arXiv preprint arXiv:1503.00075.

Tufano, Michele, Watson, Cody, Bavota, Gabriele, Di Penta, Massimiliano, White, Martin, Poshyvanyk, Denys, 2018. Deep learning similarities from different representations of source code. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 542–553.

Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N., Kaiser, Łukasz, Polosukhin, Illia, 2017. Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30.

Vladimir, Kovalenko, Bogomolov, Egor, Bryksin, Timofey, Bacchelli, Alberto, 2019. Pathminer: A library for mining of path-based representations of code. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 13–17.

Wan, Yao, Zhao, Zhou, Yang, Min, Xu, Guandong, Ying, Haochao, Wu, Jian, Yu, Philip S., 2018. Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 397–407.

Wang, Yue, Wang, Weishi, Joty, Shafiq, Hoi, Steven C.H., 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.

Wei, Huihui, Li, Ming, 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: IJCAI. pp. 3034–3040.

White, Martin, Tufano, Michele, Vendome, Christopher, Poshyvanyk, Denys, 2016. Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 87–98.

Wu, Yueming, Feng, Siyue, Zou, Deqing, Jin, Hai, 2022. Detecting semantic code clones by building AST-based Markov chains model. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–13.

Wu, Yueming, Zou, Deqing, Dou, Shihan, Yang, Siru, Yang, Wei, Cheng, Feng, Liang, Hong, Jin, Hai, 2020. SCDetector: Software functional clone detection based on semantic tokens analysis. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 821–833.

Zeng, Jie, Ben, Kerong, Li, Xiaowei, Zhang, Xian, 2019. Fast code clone detection based on weighted recursive autoencoders. IEEE Access 7, 125062–125078.

Zhang, Jian, Wang, Xu, Zhang, Hongyu, Sun, Hailong, Wang, Kaixuan, Liu, Xudong, 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.

Zou, Yue, Ban, Bihuan, Xue, Yinxing, Xu, Yun, 2020. CCGraph: A PDG-based code clone detector with approximate graph matching. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 931–942.