Towards Learning Generalizable Code Embeddings using Task-agnostic Graph Convolutional Networks

ZISHUO DING, Concordia University, Canada
HENG LI, Polytechnique Montréal, Canada
WEIYI SHANG, Concordia University, Canada
TSE-HSUN (PETER) CHEN, Concordia University, Canada

Code embeddings have seen increasing applications in software engineering (SE) research and practice recently. Despite the advances in embedding techniques applied in SE research, one of the main challenges is their generalizability. A recent study finds that code embeddings may not be readily leveraged for the downstream tasks that the embeddings are not particularly trained for. Therefore, in this paper, we propose *GraphCodeVec*, which represents the source code as graphs and leverages the Graph Convolutional Networks to learn a more generalizable code embeddings in a task-agnostic manner. The edges in the graph representation are automatically constructed from the paths in the abstract syntax trees, and the nodes from the tokens in the source code. To evaluate the effectiveness of *GraphCodeVec*, we consider three downstream benchmark tasks (i.e., code comment generation, code authorship identification, and code clones detection) that are used in a prior benchmarking of code embeddings and add three new downstream tasks (i.e., source code classification, logging statements prediction, and software defect prediction), resulting in a total of six downstream tasks that are considered in our evaluation. For each downstream task, we apply the embeddings learned by *GraphCodeVec* and the embeddings learned from four baseline approaches and compare their respective performance. We find that *GraphCodeVec* outperforms all the baselines in five out of the six downstream tasks and its performance is relatively stable across different tasks and datasets. In addition, we perform ablation experiments to understand the impacts of the training context (i.e., the graph context extracted from the abstract syntax trees) and the training model (i.e., the Graph Convolutional Networks) on the effectiveness of the generated embeddings. The results show that both the graph context and the Graph Convolutional Networks can benefit *GraphCodeVec* in producing high-quality embeddings for the downstream tasks, while the improvement by Graph Convolutional Networks is more robust across different downstream tasks and datasets. Our findings suggest that future research and practice may consider using graph-based deep learning methods to capture the structural information of the source code for SE tasks.

CCS Concepts: • **Computing methodologies → Machine learning**; • **Software and its engineering**;

Additional Key Words and Phrases: Machine learning, Source code representation, Code embeddings, Neural network

1 INTRODUCTION

Over the last few years, both researchers and practitioners have witnessed the success of applying deep learning techniques to natural language processing (NLP) tasks [?]. The advances of these neural network methods have led to breakthroughs in addressing a variety of NLP based research problems, including machine translation,

Authors' addresses: Zishuo Ding, zi_ding@encs.concordia.ca, Concordia University, Department of Computer Science and Software Engineering, Montreal, QC, Canada; Heng Li, heng.li@polymtl.ca, Polytechnique Montréal, Department of Computer Engineering and Software Engineering, Montreal, QC, Canada; Weiyi Shang, shang@encs.concordia.ca, Concordia University, Department of Computer Science and Software Engineering, Montreal, QC, Canada; Tse-Hsun (Peter) Chen, peterc@encs.concordia.ca, Concordia University, Department of Computer Science and Software Engineering, Montreal, QC, Canada.

1049-331X/2022/6-ART $15.00
https://doi.org/10.1145/3542944

document classification, etc. For example, in the task of document classification, **?**] directly apply convolutional neural networks (CNN) to the text data and obtain better results compared to the traditional support vector machine (SVM) method. As one of the key aspects in NLP, distributed vector representation of words, *a.k.a.*, word embeddings, has attracted much attention. Word embeddings project words into a low-dimensional semantic space, where each word is represented by a vector of real numbers. Studies [40, 43] show that the use of pre-trained word embeddings can improve the performance of downstream tasks (e.g., sentence classification [40]). In addition to the wide application of word embeddings in NLP, prior software engineering (SE) research also illustrates the effectiveness of distributed code representation (i.e., code embeddings) in assisting in software engineering tasks, such as automatic program repair [16, 86, 90], software vulnerability prediction [25, 70], method name prediction [2, 6], and code clones detection [11].

Despite recent advances in code embeddings, one of the main challenges of applying such embeddings in research and practice is their generalizability to downstream tasks that the embeddings were not particularly trained for. Recently, Kang et al. [34] evaluate two pre-trained code embeddings generated by GloVe [69] and code2vec [6], by applying these two pre-trained embeddings to three downstream SE tasks, including code comment generation, code authorship identification, and code clones detection. However, the results show that code embeddings may not be readily leveraged in the models of the downstream tasks for which they have not been trained. In other words, pre-trained code embeddings may not generalize to different downstream tasks.

On the other hand, both studied embedding techniques in the prior work [34] have their limitations. In particular, GloVe [69] treats the source code as plain text and only considers the unstructured local textual information which may miss the useful syntax information from the source code. Code2vec [6] parses each method in the source code to an abstract syntax tree (AST) and focuses on the utilization of the structural information extracted from such ASTs. However, the token vectors are learned using a supervised approach, where the training objective is method name prediction instead of a task-agnostic purpose. Therefore, in this work, **we aim to find out whether the lack of generalizability of these code embeddings can be alleviated by learning task-agnostic embeddings from both the syntax and semantic information of the source code in a task-agnostic manner**.

Meanwhile, the recently proposed graph-based deep learning methods [47] have been successfully employed in several SE tasks such as variable name prediction [3] and variable misuse prediction [3]. However, such graph-based methods have not been used for learning source code embeddings. Therefore, in this paper, we adopt the Graph Convolutional Networks (GCN) [17, 38] to learn code embeddings due to its ability for handling structural information in graphs. We first construct graph representations from the abstract syntax trees (ASTs) of the source code, then leverage the GCN model to train the code embeddings from the context information provided by the graph representations. Unlike previous work [3, 6, 95] which learns code representations for specific tasks, this work learns task-agnostic code embeddings, aiming to effectively apply the learned embeddings to different downstream SE tasks.

To quantitatively assess the quality of our learned code embeddings in SE tasks, we use and extend the existing benchmark tasks published by Kang et al. [34]. Specially, we add three new downstream tasks to the existing ones, resulting in a total of six downstream tasks: code comment generation, code authorship identification, code clones detection, source code classification, logging statements prediction, and software defect prediction. We apply our learned code embeddings in these benchmark tasks and compare it with four baseline approaches. Specifically, we organize the discussion of our results along with the following three research questions (RQs).

RQ1 How effective is *GraphCodeVec* compared with other baseline embedding techniques in representing the source code? We compare *GraphCodeVec* with other four state-of-the-art baseline embedding techniques in the six downstream tasks. We observe that *GraphCodeVec* outperforms the baseline approaches in five out of the six downstream tasks.

RQ2 How does the structural context information of the source code impact the effectiveness of the embeddings generated by *GraphCodeVec*? We perform an ablation experiment to understand the impact of the context information extracted from the structures of the source code (i.e., the ASTs) on *GraphCodeVec*. We find that although overall, such structural context information can benefit *GraphCodeVec* in producing code embeddings for the downstream tasks, there may be cases where the structural information may not provide additional benefit.

RQ3 How does the GCN model impact the effectiveness of the embeddings generated by *GraphCodeVec*? We perform another ablation experiment to understand the impact of the used model (GCN) for training the code embeddings. We find that using the GCN model performs better than using a shallow neural network as used in Word2vec.

The main contributions of this work include:

- We propose a source code embeddings approach, *GraphCodeVec*, which represents the source code as graphs and utilizes the Graph Convolutional Networks (GCN) to learn task-agnostic code token representations.
- We extend an existing benchmark to a total of six downstream SE tasks for evaluating code embeddings.
- We conduct comprehensive experiments on the benchmark downstream tasks, which demonstrates that *GraphCodeVec* performs comparable or better than the existing approaches on all the studied downstream tasks.
- We perform ablation experiments to understand the impact of the important modeling decisions (i.e., training context and training model) on our approach and demonstrate that both the structural context information and the GCN model benefit our approach in producing more generalizable code embeddings.
- We share our trained embeddings and downstream tasks with the research community. [1].

Paper organization. We present the background and survey prior research that is related to our work in Section 2. In Section 3, we describe our proposed approach. Section 4 presents our experimental setup. Section 5 discusses the experimental results of evaluating *GraphCodeVec* along three research questions. In Section 6, we further discuss the impact of different parameter settings and different data sampling strategies on the performance of code embeddings. Section 7 discusses the threats to the validity of our study. Finally, Section 8 concludes this paper.

2 BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Word embeddings in NLP. ~~Recently, word embeddings have become one of the most powerful techniques in natural language processing.~~ Word embeddings are a way to represent words of a vocabulary into a space with real-valued numbers. A meaningful word embedding projects each word into a low-denominational space (i.e., vector), where words with similar semantics are located closer to each other (i.e., vectors with shorter distances).

Considering the importance of word embeddings in NLP, in this subsection, we present the recent development of influential word embeddings models.

The field of word embeddings has witnessed a fast growth since the release of Word2vec [56, 57]. **Word2vec** uses a simple two-layer neural architecture to learn distributed word representations. Word2vec contains two different but related models: Continuous Bag-Of-Words (CBOW) and Skip-gram. The CBOW model tries to predict the target word by considering its surrounding words within the context window. The goal of the model

[1]The embeddings and downstream tasks are available at Google Drive.

is to minimize the following loss function:

$$L = -\frac{1}{N} \sum_{t=1}^{N} \sum_{-c \leq j \leq c, j \neq 0} \log p \left(w_t | w_{t+j} \right) \tag{1}$$

where w_t is the target word, c is the context window size, and $p \left(w_t | w_{t+j} \right)$ is the conditional probability of generating the central target word w_t from given context word w_{t+j}. Different from CBOW which utilizes the context words to predict the target one, Skip-gram model tries to predict the surrounding context words given the target word. The goal of the model is to minimize the following loss function:

$$L = -\frac{1}{N} \sum_{t=1}^{N} \sum_{-c \leq j \leq c, j \neq 0} \log p \left(w_{t+j} | w_t \right) \tag{2}$$

where w_t is the target word, c is the context window size, and $p \left(w_{t+j} | w_t \right)$ is the conditional probability of generating the context word w_{t+j} from the given central target word w_t.

GloVe [69] is a popular unsupervised embedding learning algorithm that is based on the words co-occurrence statistics. To obtain the vector representation for each word in the vocabulary, Pennington et al. [69] adopt the following loss function to train word embeddings,

$$J = \sum_{i,j=1}^{N} f \left(X_{i,j} \right) \left(w_i^T \tilde{w}_j - \log X_{i,j} \right)^2 \tag{3}$$

where X denotes the word-word co-occurrence matrix, $f(\cdot)$ is a weighting function, w_i and $\tilde{w}_j$ are the corresponding word vectors, respectively.

fastText [10] is another recent prominent embedding technique proposed by Facebook's AI Research lab. Compared to previous mentioned embedding techniques which ignore the internal structure of a word (i.e., character level information), fastText extends Skip-gram model and exploits subword information to construct word embeddings. To include the internal information of each word, fastText represents each word as a bag of character n-grams (i.e., each subword is represented by a n-gram) and learns the vector for each n-gram. Finally, each word is represented by the sum of the vector representations of its subword n-grams.

Due to the ability to capture the semantics interpretable for machines, word embeddings play an important role for many downstream NLP tasks. For example, Li et al. [43] and Vashishth et al. [84] adopt the trained word embeddings to initialize the embedding layer of neural networks based models for the task of named entity recognition (NER) which is to identify and classify the entity mentions into predefined categories, such as persons, locations, etc. Meanwhile, **?**] computes a linear combination of word embedding of each word in the text, which is then fed as the features into a logistic regression model for the task of sentiment classification to determine whether a document is positive or negative.

2.1.2 Code embeddings. Similar to word embeddings, code embeddings are a way to represent each source code token into a space with real-valued numbers. Prior research proposes various approaches for learning distributed code representations (i.e., code embeddings). In this section, we present a background of existing code embedding techniques. Based on the training context, the existing code embedding techniques can be classified into two categories: (1) textual context-based and (2) structural context-based methods.

Textual context-based embeddings. Similar to natural languages, programming languages are usually repetitive and predictable [26]. Thus, prior research [10, 20, 81] considers source code as plain text and directly applies existing word embedding techniques to source code. In this section, we review three of the most popular textual context-based works for code embeddings, i.e., Word2vec [56, 57], GloVe [69] and fastText [10].

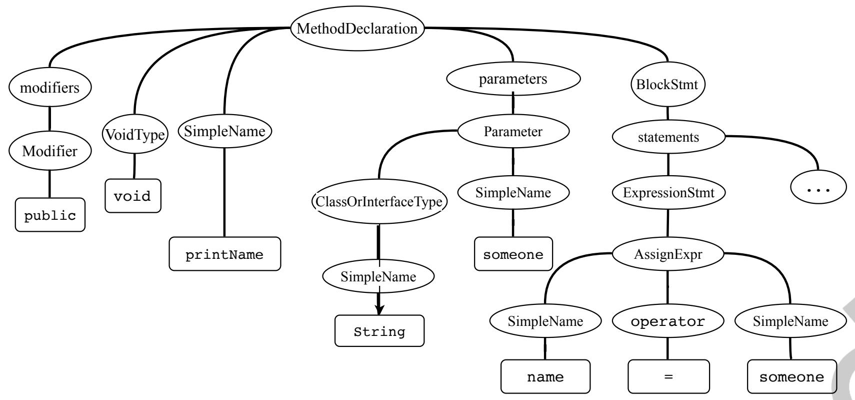


Fig. 1. Tree representation of the code snippet generated by JavaParser. For simplicity, only part of the tree are displayed.

As described in Section 2.1.1, Word2vec [56, 57], GloVe [69] and fastText [10] are all unsupervised embedding learning algorithms and can be easily adopted for source code embeddings training. In these models, the source code is treated as plain text and only the local textual information is considered.

Word2vec uses a local window with a fixed length and considers the tokens in the window that surround the target token as its context. The context tokens are treated equally or processed based on the distance with the central target token. However, this is not in accord with the programming rules. For example, consider the following class deceleration in Java.

```
public class Embeddings {
   public static int dims;
   ...
   public static float empty;
   ... }
```

Assuming the window size is five, and the target token is “Embeddings”, the token “dims” is one of the context tokens captured by the window, but the token “empty” is missed due to its long distance to the target token. However, both should have the same importance for “Embeddings”, as they are the variables declared in the same scope. Moreover, the first two “public” keywords are also within the window, and they are treated equally for “Embeddings”; but intuitively, they should be processed differently, as the first is an access modifier for the class while the other is for its attribute. fastText considers the subword information, but it still adopts a similar strategy to construct the context and faces the same problems. Although GloVe adopts the global co-occurrence statistics, it does not consider the structural dependencies among the tokens.

Structural context-based embeddings. Source code contains explicit structural information (e.g., classes, methods, branches), which may not be fully represented by a sequence of tokens [61, 65, 95]. Thus, researchers have proposed approaches that consider the structural information in the source code [6, 11, 29, 95]. In this part, we first introduce the abstract syntax tree (AST) and then describe code2vec [6], a baseline approach in our experiments, which produces the embeddings based on ASTs.

The **abstract syntax tree** represents the programs with the syntax information using a tree. As illustrated in Figure 1, the leaf nodes are the tokens of the program, and others are AST node types. Considering its power in preserving all levels of information of the source code, including the text as well as the syntactic structure, ASTs have been applied into a variety of software tasks, such as log levels suggestion [?] and code clones detection [95], etc.

Code2vec is a code representation model recently proposed by Alon et al. [6]. Like many other AST-based models, code2vec is also trained and evaluated on a single task, namely, method name prediction. In particular, in this model, the authors first extract all the methods from the selected code repository. Then the methods are transformed into a collection of ASTs. Next, triplets are constructed from the trees, where the first and last elements are the terminal nodes of the AST, and the middle element is the path connecting them. Once having the training corpus (i.e., the triplets), a path-attention network [6] is used to learn token and method names as well as the path vectors. Code2vec uses the cross-entropy loss to train the model, and the learned embeddings are task-specific.

2.2 Related work

Source code embeddings is an essential part of many SE tasks [2, 6, 11, 15, 16, 25, 70, 86, 90]. Due to the advancement of neural networks, researchers propose various approaches for learning code embeddings to assist in SE tasks. In this section, we report related works for each category of code embedding presented in Section 2.1.2.

Textual context-based code embeddings. Prior work extracts the local textual information from the source code and then applies embeddings techniques on the extracted textual information. For example, Efstathiou and Spinellis [20] treat the source code as plain text and use fastText [10] to train the embeddings for different languages. Harer et al. [25] convert code tokens into a vectorial representation using the Word2vec algorithm. They collect open-source C/C++ programs and apply the lexer on the source code. The trained embeddings are used to initialize the feature embedding layer of the TextCNN model [36], which is later used for vulnerability detection. Chen and Monperrus [16] train Doc2vec [41] on a corpus of Java files. Source code components from each java file are extracted and tokenized. The tokenized source code components are used to train a Doc2vec model for automated program repair. Similarly, White et al. [90] adopt Word2vec to transform the file-level corpus for each program revision into streams of embeddings. Intuitively, using local textual context is reasonable as developers always code the related statements together. However, during the embeddings training, neither using a too-large local window nor a too-small window is desired. A too-large local window size may include redundant or unrelated tokens (i.e., noise tokens) in, while a too-small local window size may lose the important context tokens. In addition, considering the code snippet as plain text results in the omitting of the structural information in the source code that may be important for some downstream tasks.

AST-based code embeddings. To leverage the structural information of source code, some researchers propose AST-based representation approaches. An AST represents the source code with a tree structure, which has been proven to be useful in a wide range of software engineering fields. Zhang et al. [95] propose an AST-based neural network for source code representation. In their work, ASTs are split into a sequence of small statement trees, which are later encoded into vectors. Alon et al. [6] propose code2vec and parse ASTs to a collection of triples, where the first and last elements are leaf nodes in the tree representation, and the middle element is the path connecting these two nodes. Then, they feed the triples to an attention model for learning vector representations for arbitrarily-sized snippets of code. Büch and Andrzejak [11] implement an AST-based Recursive Neural Network (RNN) for code clones detection. Recently, Allamanis et al. [3] adopt graph-based deep learning methods [47] for variable name prediction task and variable misuse prediction task. However, all of them train the embeddings on a specific task and thus require well-labelled data and may suffer the generalizability problem. Tufano et al. [83] first train four separate code embeddings based on different training contexts (i.e., identifiers, AST, bytecode, & CFG) and then use these four embeddings in detecting similar code fragments. Our work is different from these works from both the training context extraction and the embedding learning aspects.

Downstream SE tasks using code embeddings. Similar to the usage of word embeddings for downstream NLP tasks as described in Section 2.1.1, the trained code embeddings can also be integrated for downstream SE tasks

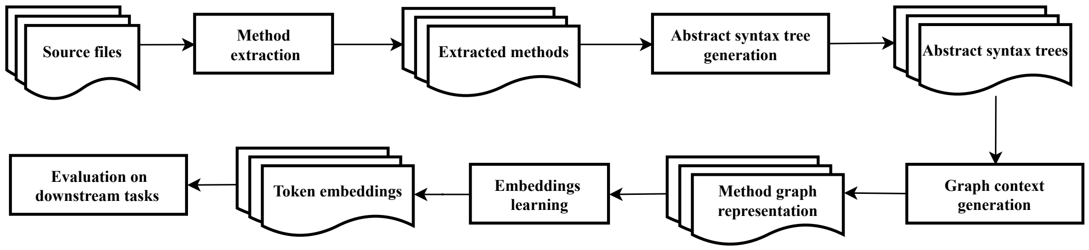


Fig. 2. The overall framework of *GraphCodeVec*. Note: we apply the same token embeddings trained from a general dataset on all downstream tasks.

in the same way. That is, these embeddings can support both deep learning and traditional machine learning SE tasks [34]. Some researchers [34, 83] use the embeddings of tokens in programs as a feature vector and then feed these features into traditional machine learning methods. For example, Tufano et al. [83] first learn the code embeddings for each program fragment and then they adopt the ensemble learning (i.e., random forest) for detecting similarities of different code fragments. In addition, code embeddings can also be used as initialization of embeddings layers of the neural network based models for downstream SE tasks. For example, Zhang et al. [95] adopt the embeddings generated by Word2vec to initialize the embedding layer's parameters in their neural network based model for the tasks of clone detection and code classification..

The limitations of the existing work. On one hand, Word2vec and GloVe only utilize the textual context without incorporating the structural context explicitly to learn code embeddings. On the other hand, most of the existing structural context-based models learn code embeddings in a supervised way, which heavily rely on the availability of well-annotated training data which is usually not available. Moreover, the embeddings are often trained and evaluated on the same task, raising the concern that the learned embeddings may not generalize well to other tasks.

Considering the limitations of existing works that leverage textual or AST-based code embeddings, we propose *GraphCodeVec*, which improves the code embeddings by representing source code as graphs and training the embeddings in an task-agnostic manner, aiming to learn task-agnostic code token representations.

3 APPROACH

Prior work [34] finds that pre-trained code embeddings may not be readily leveraged for the downstream tasks that the embeddings are not trained for. However, considering the limitations of the existing code embedding techniques, we propose *GraphCodeVec*, which consists of a training context preparation phase followed by an embedding learning phase. Figure 2 outlines the overall framework of *GraphCodeVec*. *GraphCodeVec* first extracts methods from a collection of source code files (i.e., Java classes), which are later transformed into AST representations. Based on these AST representations of methods, a context graph is then constructed for each extracted method. In the embedding learning phase, the GCN embedding approach [84] is used to train the token embeddings based on the graph context. Below, we describe the training context preparation and embedding learning phases in detail.

3.1 Training context preparation

In this section, we describe the procedures of how to represent the source code using a graph. Formally, given a code snippet $\mathcal{D} = (w_1, w_2, \ldots, w_n)$, where w_n is the nth token in the code, the goal of this step is to generate its graph representation, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes (i.e., tokens in the source code), $\mathcal{E} = \left\{e_{u,v} | u, v \in \mathcal{V}\right\}$ refers to the edges in the graph ($e_{u,v}$ represents the edge connecting nodes u and v).

3.1.1 AST generation. Apart from using the local window to construct the context, many NLP tasks adopt Syntactic Dependency Parse (SDP) to composite the context [40, 42, 43, 71, 92]. Meanwhile, previous studies [5, 9, 72] demonstrate that software engineering tasks can greatly benefit from leveraging the syntax information of programming languages. Hence, in this section, we follow a similar approach with that of Alon et al. [5] to extract the AST representations of source code.

In *GraphCodeVec*, source code is first transformed into ASTs using JavaParser[2], which provides the functionality of converting source code into tree representations. The structural syntax information of each method is preserved in an AST tree. For example, given the following code snippet, JavaParser produces the tree representation shown in Figure 1.

```
public void printName (String someone){
    name = someone;
    System.out.println(name);
}
```

As Figure 1 shows, the leaf nodes are tokens in the source code which are connected by a set of JavaParser AST node types that provide the syntax structure of the code.

Based on the AST, we then extract the nodes and edges from the AST and represent the source code using a graph. Our work shares a similar way with Alon et al. [5, 6].

3.1.2 Graph context construction. Once we have the AST representation of each method of the source code, we start to construct the graph context. We first traverse the extracted ASTs (see Section 3.1.1) to collect all the leaf nodes for each method (i.e., code tokens in the source code). The collected leaf nodes are the nodes in the constructed graph. We adopt the depth-first search algorithm implemented in "TreeVisitor" [3] for the traversal. To construct a graph representation of the method, we also need to identify the AST node types connecting these leaf nodes. The identified AST node types are the edges in the constructed graph. Given any two different leaf nodes, w_1 and w_2, the edge, $e_{1,2}$ is the shortest path between these two nodes in the method's AST. We also keep the path traversing direction to preserve as much information as possible. As a result, we can collect two different type paths for each pair of leaf nodes. The reason why we preserve the path direction is that different paths represent different syntactic relationships between these nodes. For example, in our above example, "name = someone;", for the token "name", "someone" is the source expression (i.e., assigner) and for the token "someone", "name" is the target variable (i.e., assignee). In other words, the dependency relationship from "name" to "someone" is different from the dependency relationship from "someone" to "name". Moreover, the direction of the dependency relationship is not only considered in SE tasks(e.g., [6]) but also in NLP tasks (e.g., [43]). By doing such a directed structural traversal, we construct the graph representation of the source code, where nodes represent the code tokens in the source code while the edges represent the AST node types connecting two nodes. In the constructed graph, there are N nodes and $N * (N - 1)$ directed edges[4] describing the syntactic relationship between any two nodes, and N is the number of leaf nodes in the AST (i.e., code tokens in the source code).

Figure 3 illustrates a simple example of how to construct a graph from an AST. Basically, we start from one leaf node and keep traversing until finding the shortest path that connects to another leaf node. The detailed procedure is as follows:

(1) Given an abstract syntax tree of a method, e.g., "printName", we first collect all the leaf nodes.
(2) We then choose two of the leaf nodes as the target and source nodes (e.g., "String" and "someone"), respectively.

[2]https://javaparser.org/.
[3]https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.6.0/com/github/javaparser/ast/visitor/TreeVisitor.html.
[4]We further filter the edges by a length threshold, explained later in this section.

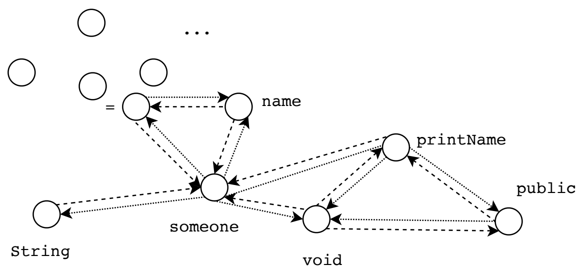


Fig. 3. Graph representation of the code snippet based on the AST. For simplicity, only part of the graph are displayed.

(3) Next, we extract the paths from the root node, `MethodDeclaration`, to the target and source nodes respectively (i.e., ⟨`MethodDeclaration, parameters, Parameter,ClassOrInterfaceType, SimpleName`⟩ and ⟨`MethodDeclaration, parameters, Parameter, SimpleName`⟩). The longest common prefix of these two paths is ⟨`MethodDeclaration, parameters, Parameter`⟩.

(4) We then remove the longest common prefix from the two paths, resulting in two sub-paths, ⟨`ClassOrInterfaceType, SimpleName`⟩ and ⟨`SimpleName`⟩. We keep the last element of the common prefix (i.e., `Parameter`).

(5) We preserve the path direction from the target node to the source node and connect the path elements with −. Specifically, we reverse the sub-path connecting the target node and assign the up direction (represented as ↑). For the sub-path connecting the source node, we remain the same order and assign the down direction (represented as ↓). For example, after this step, the two paths become `SimpleName`$^{\uparrow}$-`ClassOrInterfaceType`$^{\uparrow}$ and `SimpleName`$^{\downarrow}$.

(6) We then concatenate the two sub-paths with the preserved last element of the common prefix (i.e., `Parameter`), `SimpleName`$^{\uparrow}$-`ClassOrInterfaceType`$^{\uparrow}$-`Parameter-SimpleName`$^{\downarrow}$. Finally, we have two nodes, "String" and "someone" and the edge connecting them, i.e., `SimpleName`$^{\uparrow}$-`ClassOrInterfaceType`$^{\uparrow}$-`Parameter-SimpleName`$^{\downarrow}$, where the ↑ and ↓ are the traversing directions and no direction means an inflection node of a traversing path.

(7) We repeat steps (2) through (6) for each pair of source and target nodes, until we collect all the nodes and edges in the AST.

However, the number of edges is approximately the square of the number of tokens (i.e., leaf nodes). To reduce the size of the training data, we follow previous work [6] and limit the number of edges by a maximum length: if the length of an edge (i.e., the number of AST node types in the shorted path) exceeds the threshold, the edge will be ignored. In our work, we follow the work of code2vec [6], and set the threshold to eight as we find that two tokens connected by a longer edge usually do not have a direct structural relationship. Note that a relatively longer edge can preserve a more complete relationship between the leaf nodes, in other words, with a larger threshold, in the constructed graph, the target node can have edges to more other nodes and thus, generating more training context. Meanwhile, if the threshold is too large, more indirect relationships with the target node would be included, which may introduce more noise to the training corpus, leading to poor quality of the generated code embeddings. And if the threshold is too small, although the target token would have a more direct relationship with other nodes, the number of connected nodes would be small and lead to insufficient training data. Thus, the threshold should be tuned for specific tasks or training context.

The output of our training context preparation phase (i.e., the graph context of code tokens) are used as the input for our embedding learning.

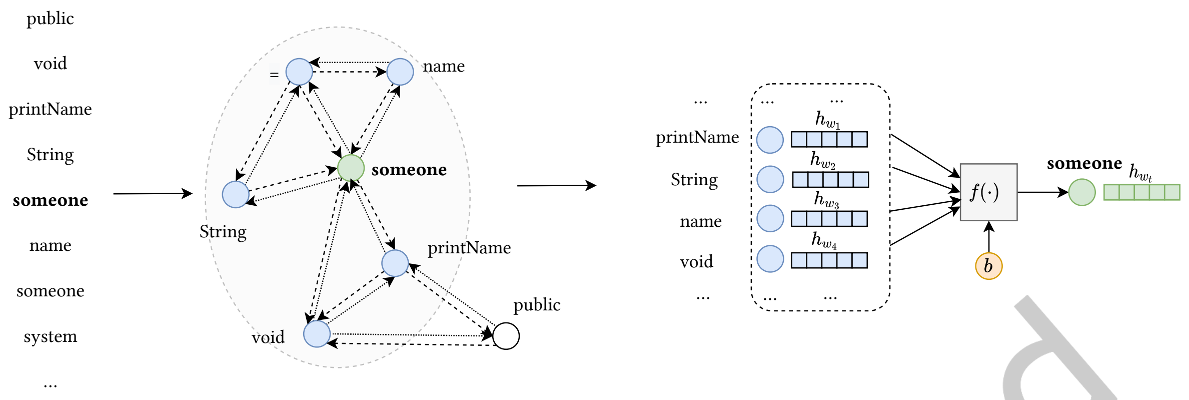


Fig. 4. An overview of our embedding learning phase: assume the target code token is "someone", the nodes in blue are the relevant context tokens which are fed into a one-layer Graph ConvolutionalNetwork (GCN) for learning the distributed representations of the target token. h_{w_i}, h_{w_t} are the hidden represetations of context token and target token, respectively.

3.2 Embedding learning

This section provides a detailed description of our approach to learning distributed token representations in a task-agnostic manner. More specially, in this work, we adopt the Graph Convolutional Networks (GCN) [84] to train the token embeddings based on the graph context generated in Section 3.1. The reason why we choose GCN is that it can not only preserve both the semantic information (i.e., leaf nodes in ASTs), but also the structural information (i.e., the connecting paths in ASTs) of the source code [3].

Figure 4 illustrates our embedding learning phase. Assuming the target token is "someone", the relevant context tokens (e.g., "name", "String", "printName", "void") are fed into the GCN model for predicting the target token, "someone". **Formally, given a graph representing the source code snippet, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the goal is to learn a d-dimensional embedding for each token in $\mathcal{V}$.**

Similar to the Continuous Bag-Of-Words (CBOW) model [56, 57], which tries to predict the target token using its surrounding tokens within a local window, our approach utilizes the directly connected nodes (i.e., its neighbors), C_{w_t} to predict the given target node w_t.

Hidden representation for each node. The hidden representation (hidden state) of each node is the output of a convolutional layer in GCN. As Figure 4 shows, the hidden representation of the target token $h_{w_t} \in \mathbb{R}^d$ is updated based on its neighbors in the graph context. More specially, the representation for the target node w_t at the $(l+1)$th layer in GCN is computed by:

$$h_{w_t}^{l+1} = f\left(\sum_{w_c \in C_{w_t}} \left(W_{e_{w_c,w_t}}^{l} h_{w_c}^{l} + b_{e_{w_c,w_t}}^{l}\right)\right) \tag{4}$$

where $W_{e_{w_c,w_t}}^{l}$ and $b_{e_{w_c,w_t}}^{l}$ are a trainable weight matrix and a bias, and $h_{w_c}^{l}$ is the hidden representation for context node w_c at the lth layer.

Edge-wise gating mechanism. As described in Section 3.1, to reduce the number of edges in the graph, we do filtering using a threshold of edge length. In addition, there may exist different relationships among the leaf nodes: some are weak and meaningless, while others may be more meaningful. For example, we see in Figure 4 that even though the target token "someone" is directly connected with the token "void", their relationship is not meaningful. In comparison, the relationship between "name" and "someone" is stronger. Therefore, we should assign different weights to different context nodes when calculating the hidden representation for the target node.

To address this issue, we adopt the edge-wise gating mechanism [52]. For each target node w_t, the weight score with its context token w_c is calculated as follows:

$$g^l_{e_{w_c,w_t}} = \sigma\left(W'^k_{e_{w_c,w_t}} h^k_{w_c} + b'^k_{e_{w_c,w_t}}\right) \tag{5}$$

where $W'^k_{e_{w_c,w_t}}$ and $b'^k_{e_{w_c,w_t}}$ are trainable parameters and $\sigma(\cdot)$ is the sigmoid function. Thus, the hidden representation of the target nodes is formulated as:

$$h^{l+1}_{w_t} = f\left(\sum_{w_c \in C_{w_t}} g^l_{e_{w_c,w_t}} \times \left(W^l_{e_{w_c,w_t}} h^l_{w_c} + b^l_{e_{w_c,w_t}}\right)\right) \tag{6}$$

Training objective. Given a graph representation of the source code, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and the target node, w_t (the tth node), the objective of the model is to maximize the following objective function:

$$\mathcal{L} = \sum_{w_t \in \mathcal{V}} \log P\left(w_t | C_{w_t}\right) \tag{7}$$

where C_{w_t} is the context nodes (i.e., neighbors in the graph) of the target nodes w_t, $P\left(w_t|C_{w_t}\right)$ is the conditional probability of observing the target node w_t given the context nodes, C_{w_t}. $P\left(w_t|C_{w_t}\right)$ is defined using the following *softmax* function:

$$P\left(w_t|C_{w_t}\right) = \frac{\exp(\mathbf{v}_{w_t}{}^\top h_{w_t})}{\sum_{w \in \mathcal{V}} \exp(\mathbf{v}_w{}^\top h_{w_t})} \tag{8}$$

where $\mathbf{v}_w$ and h_w denote the target embedding and hidden representation of the node w, respectively.

Optimization. One issue in *GraphCodeVec* is the high cost of computation in the *softmax* function (i.e., Equation 8) because it involves the iteration through every node over $\mathcal{V}$. To address this issue, different optimization strategies can be applied, such as hierarchical softmax and negative sampling [57]. Hierarchical softmax [**? ?**] uses a binary tree to represent the tokens in the vocabulary, $\mathcal{V}$, where each leaf node of the tree is a token. The probability of traversing from the root to the leaf node (i.e., target token) along the unique path is used to estimate the conditional probability. By doing such an approximation, the complexity of calculating the probability of each word goes down from $O(|\mathcal{V}|)$ to around $\log_2(|\mathcal{V}|)$ [**? ?**]. While negative sampling is more straightforward [**? ?**]. The idea of negative sampling is to update a small sample of the token vectors rather than all of them, such that the computing cost of the *softmax* function can be reduced. In this work, following previous work [69, 84**?**], we adopt the negative sampling, as it tends to give better results than hierarchical softmax [24, 69].

The output of our embedding learning phase (i.e., the token embeddings) are used as the input for our downstream tasks for evaluation.

4 EXPERIMENTAL SETUP

In this section, we present details of our embedding training settings and describe the six downstream tasks used in our quantitative evaluation. Three of the SE tasks, i.e, (1) code comment generation, (2) code authorship identification, and (3) code clones detection, are used for the evaluation of code embeddings in prior research [34]; while the other three, i.e., (4) source code classification, (5) logging statements prediction and (6) code defects prediction, are newly added in our extended benchmark. We select these tasks either due to the fact that they are chosen for evaluating code embeddings in previous work [34], or they are of great importance for SE community and commonly studied in the literature.

4.1 Dataset preparation

In our experiments, the dataset used for embedding learning comes from the *Java-small* dataset[5], which is provided by Alon et al. [6] and originally based on the dataset of Allamanis et al. [4]. This dataset is collected from publicly available open-source GitHub repositories.

Following the previous approach for pre-processing the source code [6, 15, 34], we convert the tokens into lower cases and remove all the non-identifiers (e.g., quotation marks). Meanwhile, we follow the common practice [56, 57, 69, 84] and ignore all tokens with a total frequency of less than five as there is not enough data to do any meaningful training on those rare tokens [10, 73, 89]. While constructing the graph representation, due to the limitation of the memory, we only keep the top-100 most frequent edge types (i.e., edges with the identical path representation) and others are replaced with a unique identifier (i.e., -1). As during embedding learning, we need to batch the training context with different edge types into the GCN model, and in the GCN model, we create an adjacency matrix for each edge type, that means if there are a large number of edge types, the model requires more memory to keep these matrices and would run out of memory and cannot be moved to GPU for embedding training. Besides, as recommended by Vashishth et al. [84], we also limit the size of each graph to a maximum of 100 unique nodes and 800 edges; that is, if the size of the graph exceeds the threshold, the graph will be removed from the training set. After preprocessing, we collect 637,108 training methods (there are 665,115 methods before the preprocessing), each of which is represented by a graph for subsequent embedding learning. In this work, considering the fact that code2vec can only be trained on method level corpus (c.f., Section 2), to have a fair comparison with these baselines, we only construct the method level graph context. However, as ASTs can represent the source code with different levels (e.g., method level, statement level, class level, etc.), our method can also be applied to other types of training data.

The datasets used in the downstream tasks may have different vocabulary from the training dataset, *a.k.a*, the out-of-vocabulary (OOV) problem. To handle the OOV tokens, we choose to randomly initialize the vector representation of tokens that only appear in downstream tasks to minimize the impact of these unseen tokens (i.e., to make tasks with OOV vocabulary predictable). By doing this, we can make sure that all tokens in downstream tasks have vector representations, therefore it is always predictable (but may lead to poor performance as the vectors representing these OOV tokens are not learned from their context).

4.2 Training details

While training the model, we follow the settings in prior work [6, 34, 95] and set the dimension of token vectors to 128. To prevent overfitting and avoid performance degradation, we set the number of GCN layers to 1, as GCN tends to suffer performance degradation with increased depth (i.e., number of layers) [**? ? ? ? ?**]. The training batch size is set to 64 by default. Considering 1) the small number of weights of our model (i.e., one layer and 128 input dimension), 2) the relatively large size of the training data (i.e., more than half million graphs), and 3) the remarkable learning ability of GCNs from the graph data, we train our embeddings for one epoch and the training loss is small enough. This is consistent with the finding of Mikolov et al. [56], that is for word embeddings, training a model on a relatively large dataset using one epoch gives comparable or better results than more epochs on the same dataset. As it is indicated in a prior work by Mikolov et al. [57], the number of negative samples in the range of two to five is useful for large training datasets and five to 20 for small training data. Hence, in this work, to balance the efficiency and accuracy, we set the number to five. The training of our embeddings are conducted in a machine with an NVIDIA GTX 1080Ti GPU and 32GB memory. We summarize the thresholds and hyperparameters used in our experiment in Table 1.

We evaluate the quality of the trained embeddings on six downstream tasks. For the downstream tasks that use neural network-based models, the embeddings are used to initialize the embedding layer of neural networks, as

[5]https://s3.amazonaws.com/code2vec/data/java-small_data.tar.gz

Table 1. Hyperparamters and thresholds used during the two stages of *GraphCodeVec* for generating the code embeddings.

Stage	Name	Default value	Description
Training context generation (c.f., Sec. 3.1 and RQ2)	Edge length	8	Edge length (c.f., Sec. 3.1) is the number of AST nodes connecting two leaf nodes (i.e., code tokens). It would influence the quality and quantity of the training context. A smaller value would result in a more tight connection but less connected nodes to the target token, leading to not enough training context. On the contrary, a larger value may include more unrelated token pairs and introduce noise to the training context. In our work, we follow the work of code2vec [6] and set it to eight to make a fair comparison.
	Unique node	100	Unique node (c.f., Sec. 4.1) refers to the number of unique tokens (c.f., Sec. 3.1.2) within each method. This parameter would influence the size of each constructed graph for training. The values should be tuned based on the GPU memory size, as we need to batch the graphs into the GPU for training, if the graphs are too large, the model requires more memory to keep these data and would throw "out of memory" error. Vashishth et al. [84] suggest the number of unique node should be set no larger than 100. And in our settings, only about 4% of the graphs are filtered which not only has a small effect on the quantity of the training context but also can avoid the memory error.
	Edge	800	Edge (c.f., Sec. 4.1) refers to the total number of extrated AST node types (c.f., Sec. 3.1.2) within each method. It has a similar effect with the parameter Unique node on the size of contracted graph. Also, the values should be tuned based on the GPU memory size, And in our work, we set this parameter to 800, as we find that only about a small portion (i.e., 4%) of the graphs are filtered out.
	Window (c.f., RQ2)	5	Window (c.f., RQ2) is the maximum distance between the current and its neighboring word within a method. It is similar to edge length, which also has an impact on the constructed training context. A larger window size would be able to capture more broad context, but with the possibility of introducing noise as the context tokens might not be tightly related to the target token. On the contrary, a smaller window size may contain more focused information about the target word but may not be able to capture sufficient context. Setting the context window size to five is commonly done in the literature [10, 42, 56, 57, 73].
Embedding learning (c.f., Sec. 3.2)	Layer	1	Layer (c.f., Sec. 4.2) refers to number of layers in GCN. It controls the depth of a GCN and directly influences the quality of the model. Previous work [? ? ? ? ?] shows that GCN tends to suffer performance degradation with increasing depth (i.e., number of layers). In our work, we follow the work of [84] and use the default value.
	Dim.	128	Dim. (c.f., Sec. 4.2) is the dimensionality (i.e., vector size) of each token. It has a non-negligible impact on the quality of the embeddings. A small vector size cannot preserve the properties of the tokens of high dimensional spaces, leading to the degradation of quality of learned embeddings. However, a too large size requires more computing resources and training time, and may suffer the sparsity problem if the training data is not enough. In our work, we follow the settings in prior work [6, 34, 95].
	Neg.	5	Neg. (c.f., Sec. 3.2) refers to the number of negative samples used when updating the wights of the model. A larger value means more samples to calculate and thus more training time needed. This parameter should be adjusted based on the size of training context. As suggested by Mikolov et al. [56, 57], 5-20 samples works well for smaller datasets, and 2-5 words for large datasets.
	Batch size	64	Batch size (c.f., Sec. 4.2) defines the number of training samples presented in a single batch. A larger size can speed up the training process but requires more GPU memory [?] while using small batch sizes achieves better training stability [53]. In this work, we use the default 64.
	Dropout rate	0	Dropout rate is the probability of dropping a unit out. Dropout is a regularization technique for avoiding the model overfitting. As larger models (more layers or more units) tend to more easily overfit the training data [? ?] and considering the small size of our model, we don't use this strategy, instead, we reduce the training epochs to avoid overfitting.
	Epoch	1	Epoch (c.f., Sec. 4.2) is the number of iterations through the entire training dataset. This factor affects the performance of the embeddings directly. Increasing the number of epochs may overfit the model and a small number of epochs may lead to a not fully trained model. In our work, considering the size of the training dataset and the number of weights of our model (i.e., one layer, 128 input dimensions) [56, 57], we train our model for one epoch, as the training loss is small enough.

changing the embeddings of the embedding layer would affect the way the model is learnt and thus the models with different code embeddings would have different performance. For the downstream tasks that use traditional machine learning models, the embeddings are used as feature vectors (i.e., each dimension of the embeddings is treated as a feature). For example, we have a code snippet “String name = someone”, and each token (“string”, “name”, and “someone”; “=” is removed) within the vocabulary has its corresponding vector representation, such as [0.1, 0.2, 0.3, ...], [0.1, 0.1, 0.1, ...] and [0.2, 0.2, 0.3, ...], these vectors can be summed up (or other operations) as a feature vector (each feature is one dimension of the embedding), which later can be used for traditional machine learning models.

4.3 Baselines

To evaluate the effectiveness of our trained embeddings, we compare *GraphCodeVec* with the following existing embedding models (i.e., the baselines):

- **Word2vec**[6] is a popular unsupervised word embedding method proposed by Mikolov et al. [56] . We use the implementation in Gensim[7] [73].
- **GloVe** is an unsupervised algorithm using token-token co-occurrence statistics, proposed by Pennington et al. [69].
- **fastText** is proposed by Facebook's AI Research lab [10]. It is an unsupervised algorithm, which utilizes the subword information to enrich the word vectors. In their approach, each word is represented as a bag of character n-grams, and the word is represented as the sum of these character n-grams representations. We use the implementation in Gensim[8] [73].
- **code2vec**[9] is a recently proposed supervised model for source code representation. Prior work [34] evaluates code2vec on three downstream SE tasks. This model is proposed by Alon et al. [6] and utilizes the AST information to learn code embeddings.

We train these embeddings on the same dataset that is used for training our *GraphCodeVec* embeddings (i.e., the *Java-small* dataset). To make a fair comparison, we do the same preprocessing as in Section 4.1, that is converting the tokens into lower cases and removing all the non-identifiers, as well as ignoring all tokens with a total frequency lower than five.

4.4 Downstream tasks for evaluation

In this section, we briefly describe the six downstream tasks, including the approaches, the corresponding datasets used for the evaluation, and the evaluation criteria.

To control the quality of the embeddings evaluation experiments, we enrich the work of Kang et al. [34] by adding three new tasks and adopting different modeling methods for the six tasks, including deep learning approaches and traditional machine learning methods. Specifically, for the first five tasks, including (1) code comment generation, (2) code authorship identification, (3) code clones detection and (4) source code classification, and (5) logging statements prediction, we use neural network-based approaches; while for the task of (6) software defect prediction, we follow the approaches used in their original work and adopt traditional machine learning methods (i.e., logistic regression, LR in short).

We intentionally select both the deep learning and the traditional machine learning approaches to ensure the code embeddings are adequately evaluated across different tasks (i.e., six downstream tasks) and modeling approaches (i.e., traditional machine learning and deep learning). However, we specifically select LR for the only task of software defect prediction due to the fact that most of the downstream tasks that rely on code embeddings use deep learning models, thus we only select one task and put more focus on the impact on deep learning models. We run the experiments with 10-fold cross validation to mitigate the effects of the random separation of the training and test sets, and report the average scores of the results of the 10-fold cross validation. For the models

[6]There are two variants in the implementation of Word2vec (i.e., Skip-gram and CBOW) and two different optimization strategies (i.e., negative sampling and hierarchical softmax). Following previous work [84], we here select the CBOW with negative sampling as a representation for comparison.
[7]https://radimrehurek.com/gensim/
[8]https://radimrehurek.com/gensim/
[9]https://github.com/tech-srl/code2vec

selection for downstream SE tasks, we follow the rules that 1) are used in previous work [34, 65, 87], and 2) are commonly used and have the state-of-the-art or competitive results [37, 95]. To further ensure a fair comparison with baselines, we either follow the parameter settings in previous work or use the default parameters and avoid only fine-tuning these settings only for our method.

In the evaluation, our focus is the effectiveness of different embeddings instead of the approaches for the specific tasks themselves. Thus, we do not aim to reach the SOTA for a specific task. Moreover, for each downstream task, we try to use the same experimental settings that are reported in the literature, hence only examining the impact of different embedding techniques on the downstream tasks.

4.4.1 Code comment generation. Given a code snippet, which can be either a method or a class, the task is to automatically generate the corresponding code comments [29, 55, 59, 78], in order to assist in program understanding and maintenance.

Approach. Following Kang et al. [34], we use the Sequence-to-Sequence (Seq2Seq) approach proposed by Hu et al. [29] to generate the comments. Hu et al. [29] consider the comment generation task as a neural machine translation task. A Recurrent Neural Network-based Seq2Seq model is applied to generate comments based on the context of the source code.

We train the model using OpenNMT[10] [39] and keep the hyperparameters the same with literature [29]. We set the number of layers to 2 and use Long Short-Term Memory (LSTM) [27] as both the encoder and the decoder. Each LSTM has 500 hidden states, the learning rate is set to 0.5, and the dropout ratio is 0.5. The model is trained for 50 epochs , and we select the model that has the best results on the validation set as the final model. Both the encoder and decoder contain an embedding layer, which can be initialized by different embeddings.

Dataset. The evaluation dataset is provided by prior work[11] [29], which was initially collected from GitHub. We preprocess the dataset by converting all the tokens into lower cases and remove all the non-identifiers (e.g., quotation marks). After preprocessing, the dataset contains 470,485 <Java method, comment> pairs for training, 58,810 pairs for validation and 58,810 pairs for testing.

Evaluation. We evaluate the quality of the generated code comments using two machine translation evaluation metrics i.e., BLEU [67] and ROUGE[12] [**?**] as they are widely used in the task of code comment generation [29, 34**?** **?**]. BLEU is calculated as follows:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{9}$$

$$BP = \begin{cases} 1 & if\ c > r \\ e^{(1-r/c)} & if\ c \le r \end{cases} \tag{10}$$

where p_n is the modified n-gram precision (i.e., the maximum number of n-grams co-occurring in the automatically generated code comment and the reference comment divided by the the total number of n-grams in the generated comment), w_n are positive weights that can be configured, BP is a brevity penalty, c is the length of the generated comment and r is the length of the reference comment. In our evaluation, we choose $N = 4$ and uniform weights $w_n = 1/N$, same as prior work [29]. ROUGE is calculated as follows:

$$\text{ROUGE-n} = \frac{\sum_{gram_n \in Ref} Count_{match}(gram_n)}{\sum_{gram_n \in Ref} Count(gram_n)} \tag{11}$$

where n is the length of the n-gram ($gram_n$), and $Count_{match}(gram_n)$ is the number of n-grams co-occurring in the automatically generated code comment and the reference code comment, Ref. Specifically, following previous

[10]https://opennmt.net/
[11]https://github.com/xing-hu/DeepCom
[12]https://github.com/pltrdy/rouge

work [**? ?**], we calculate ROUGE-L which measures the longest matching sequence of tokens using LCS (Longest Common Subsequence). The higher the BLEU and ROUGE scores, the better the model.

4.4.2 Code authorship identification. Given a code snippet, the task is to identify its author based on the programmer's distinctive stylometric features [1, 30]. The task has many applications in the privacy and security filed, such as identifying programmers of malware and other malicious programs. Following Kang et al. [34], we also evaluate the embeddings on this task.

Approach. Kang et al. [34] treat code authorship identification as a classification problem. Following Kang et al. [34], we use an LSTM neural network, which contains two hidden LSTM layers followed by a fully-connected layer. The learning rate is set to 0.005, and the model is trained for 50 epochs. We select the model from the last epoch as the final model. This neural network contains an embedding layer, and we initialize it with different code embeddings. We follow the use of token embeddings in Kang et al. [34].

Dataset. The evaluation dataset is provided by prior research [34], which was initially collected from Google Code Jam. The dataset contains 2,250 programs (5,548 methods) in total from 250 authors. Each author has the same number of programs. Similar to the previous task [6, 15, 34], we preprocess the dataset by converting the tokens into lower cases and removing all the non-identifiers (e.g., quotation marks).

Evaluation. Following the existing work [1, 30], we use the test accuracy as the evaluation metric. It calculates the percentage of correct classifications for the test set:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} \tag{12}$$

4.4.3 Code clones detection. Given two code fragments, the task of code clones detection aims to check whether they are duplicate or not. It is widely studied in the literature and useful for program maintenance and avoiding bugs caused by source code reuse in software systems [7, 19, 33, 54, 76, 82, 88, 91]. This task is identified as a downstream task to evaluate token embeddings in prior work [34].

Approach. For this task, we use the approach proposed by Zhang et al. [95], which considers code clones detection as a binary classification problem. The approach splits the entire AST into a collection of statement trees and then encodes the statement trees to vectors while retaining the lexical and syntax information. A bidirectional Recurrent Neural Network-based model is used to produce the representation of the code fragment. We select this approach since it uses a neural network-based approach and contains an embedding layer that can be initialized by pre-trained code embeddings. In addition, the model is recently proposed and gives competitive results.

Following the settings in the work of Zhang et al. [95], we set the hidden dimension of the encoder and bidirectional GRU to 100. The learning rate is set to 0.002, and the model is trained for 15 epochs.

Dataset. There are two public dataset benchmarks used for code clones detection [95] . The first dataset is constructed from the standard BigCloneBench (BCB) [80]. The dataset contains nearly 6 million true clone pairs and 260 thousand false clone pairs parsed from BCB. The second dataset is collected from the Online Judge system (namely, OJClone) which was initially provided by Mou et al. [62].

Evaluation. Following prior work [95], the commonly used classification evaluation metric F1-measure (F1) is used to measure the performance of the models with different embeddings. It is given as follow:

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{13}$$

where $Precision = \frac{TP}{TP+FP}$, and $Recall = \frac{TP}{TP+FN}$, TP refers to the number of true positives, FP is the number of false positives, and FN is the number of false negatives.

4.4.4 Source code classification. Given a collection of code fragments, this task is to classify them into corresponding categories based on their functionalities. We choose this task as it is commonly studied in the literature [35, 62, 85, 95] and has various applications. For example, in order to help other developers on Github find and contribute to projects, owners are encouraged to assign related topics to projects[13]. With the help of code classification techniques, the topics can be automatically attached to the projects.
Approach. Source code classification is a multi-class classification problem. We use the approach proposed by Kim [37] as it is a widely used classification model and achieves competitive results [95]. The model is trained for 50 epochs and we select the model that has best results on the training set as the final model. The learning rate is set to 0.01 and the batch size is 64. The kernels sizes for convolution are set to 3, 4, 5 and the number of output channels for the convolutional layer is set to 100.
Dataset. The dataset is collected from the Online Judge system[14] and provided by Mou et al. [62]. The dataset contains 104 classes categories, and each has 500 code fragments. The code in the dataset is converted into lower cases and all the non-identifiers are removed.
Evaluation. Following prior work [95], we use the test accuracy metric, which is the same as the one used in the code authorship identification task (c.f., Section 4.4.2).

4.4.5 Logging statements prediction. Given a code snippet, this task is to predict whether there is a need to insert logging statements for collecting valuable runtime information. Logging statements play a crucial role in tracking important runtime information of software systems. Developers rely heavily on such information to monitor system behaviors and debug system failures [**?**]. However, adding unnecessary logging statements can significantly increase system overhead [18, 94] and hide the truly useful information [21]. Therefore, providing logging suggestions on whether to log is helpful for software developers.
Approach. Logging statements prediction is a binary classification problem. Similar to source code classification, we use the approach proposed by Kim [37] as it is recently proposed and widely used for classification tasks. We adopt the same experimental settings as we do in the task of source code classification (c.f., Section 4.4.4).
Dataset. The dataset contains five open source Java systems [15]: Hadoop, Directory-Server, CloudStack, Camel and Airavata and is provided by **?**].
Evaluation. Following the work of **?**], we use the balanced accuracy (BA) metric to evaluate the performance of the model with different embeddings.

BA averages the percentage of correctly identified logged and unlogged methods and is widely used to evaluate the performance of models on imbalanced data. BA is calculated as follows:

$$BA = \frac{1}{2} \times \frac{TP}{TP + FN} + \frac{1}{2} \times \frac{TN}{FP + TN} \tag{14}$$

where TP refers to the number of true positives, FP is the number of false positives, and FN is the number of false negatives. A higher value of BA indicates a better model.

4.4.6 Software defect prediction. The task of software defect prediction is to predict whether the given code snippet contains defects. Various techniques have been proposed to detect defects [87]. We select this task as a downstream task since it can effectively help developers find bugs in source code and prioritize their testing efforts [87].
Approach. software defect prediction is a binary classification problem. To extend the generalizability of our evaluation of different embeddings, unlike the previous five downstream tasks where the pre-trained embeddings are used to initialize the embedding layer of neural networks, we choose to use a logistic regression (LR) classifier

[13]https://help.github.com/en/github/administering-a-repository/classifying-your-repository-with-topics
[14]https://sites.google.com/site/treebasedcnn/
[15]The original dataset contains six projects but the repository of Qpid-Java is unavailable at the time of this work.

to learn the likelihood of defects from the code snippets, as it is used in original work [87] and thus we can compare our results with that of the original work [87] to check whether we build an up-to-standard model. To represent the code snippets, we follow a similar way in NLP tasks [13]. Specifically, we average the embedding vectors of the code tokens in a source code file:

$$\vec{v_C} = \frac{1}{|C|}\sum_{c\in C}\vec{v_c} \tag{15}$$

where C is a set of code tokens in a source code snippet and $\vec{v_c} \in V$ is the embedding vector of token c, V is the learned embeddings, $\vec{v_C}$ is the final vector representation of the code snippet, which are then fed into an LR classifier as features.

For our LR classifier, the implementation is based on scikit-learn [68], and the threshold for binary classification is set to 0.5, which is the default value of scikit-learn. As the defect data are often imbalanced, we perform a re-sampling technique (i.e., SMOTE) to balance the training data.

Dataset. The dataset is provided by Wang et al. [87], which contains eight open source Java systems. Following Wang et al. [87], we use two consecutive versions of each project to generate the training and testing dataset: the source code of an older version is considered as the training data, and that of a newer version is used to generate the testing data.

Evaluation. Following the work of Wang et al. [87], we use the F1 score to evaluate the performance of the model with different embeddings.

5 EXPERIMENTAL RESULTS

In this section, we discuss our experimental results of evaluation our proposed approach, *GraphCodeVec*, organized along three research questions (RQs). For each RQ, we explain the motivation and the approach before discussing the corresponding results.

RQ1: How effective is *GraphCodeVec* compared with other baseline embedding techniques in representing the source code?

Motivation

Prior research [6, 11, 16, 20, 25, 29, 83, 95] proposes different distributed code representations (i.e., code embeddings) approaches to assist in software engineering tasks (e.g., method name prediction and software vulnerability prediction). However, a recent study by Kang et al. [34] finds that code embeddings may not be readily leveraged to enhance existing models for the downstream tasks which they have not been trained for. Therefore, in this research question, we would like to explore whether our task-agnostic GCN-based approach (i.e., *GraphCodeVec*) can produce a more generalizable token embeddings for a variety of SE tasks compared with other baselines.

Approach

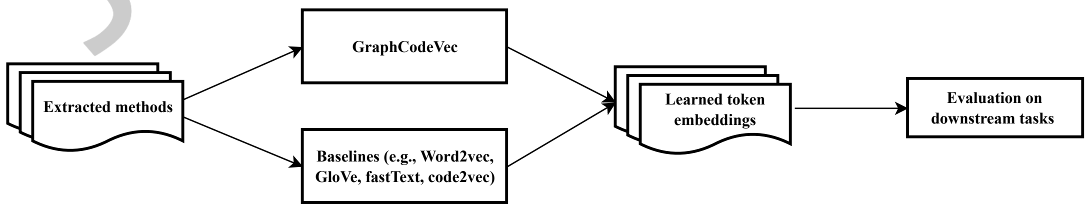

GraphCodeVec
Extracted methods
Learned token embeddings
Evaluation on downstream tasks
Baselines (e.g., Word2vec, GloVe, fastText, code2vec)

Fig. 5. The overall design of the approach for RQ1. In this experiment, the same prepocessed dataset is used by *GraphCodeVec* and baselines.

To answer our first research question, we need to train the token embeddings produced by different embedding techniques (i.e., *GraphCodeVec* and baselines, c.f., Section 4.3). As shown in Figure 5, during code embeddings training, the same preprocessed training dataset (i.e., the *Java-small* dataset, c.f., Section 4.1) is used by different embedding techniques.

Once finished the code embeddings training, we then need to evaluate these pre-trained embeddings. However, as there is no direct evaluation methodology for evaluating the quality of code embeddings, we thus follow previous work [34] and use six downstream SE tasks to evaluate the quality of code embeddings. Each of the tasks has its respective dataset for model training and evaluation, and the only varying factor in each evaluation task is the code embeddings (produced by *GraphCodeVec* and baselines) used for code token representation (i.e., for each task, only the embeddings are changed and other parameters are kept the same), and thus we can conclude the performance changes are caused by the code embeddings. Note that the change of code embeddings would also impact the weights learned for each model, which is discussed in Section 7. The detailed description of the downstream SE tasks and the corresponding evaluation metrics are presented in Section 4.4.

Results

Overall, *GraphCodeVec* **performs comparable or better than all baseline approaches on all downstream tasks.** The experimental results are provided in Table 2 with the best results for each task and dataset highlighted in bold. In particular, *GraphCodeVec* achieves the best results in five out of the six tasks, including the tasks of code authorship identification, code clones detection, source code classification, logging statements prediction, and software defect prediction. To better illustrate the results, we specifically compare with GloVe, as did in Kang et al. [34], since it was one of the most important work aiming for generating task-agnostic embeddings at the time of our research. Then, we conduct a statistical analysis using a Wilcoxon signed-rank test to compare the performance of *GraphCodeVec* and the performance of GloVe. We use a p-value that is below 0.05 to indicate that the performance difference is statistically significant. For the differences that are statistically significant, we further compute the Cliff's delta effect size. The reason why we use the Wilcoxon signed-rank test and Cliff's delta is that they both do not assume a normal distribution of the compared data. As shown in Table 2, *GraphCodeVec* performs better than Glove in 16 out of 23 cases, and 68.8% of the improvements are statistically significant with a magnitude of "large". We obtain a 5.0% relative increase in accuracy on source code classification task compared to the representative baseline (i.e., GloVe). Moreover, for the evaluation on the task of software defect prediction, which uses a traditional machine learning approach (i.e., Logistic Regression), our embeddings reach the best results on more than half of the datasets. For the Log4j dataset, we obtain around 10.1% absolute increase (24.1% relative increase) in the F1 score compared to that of GloVe. The results demonstrate that the learned embeddings from *GraphCodeVec* can better represent the source code and generalize to various downstream tasks. Besides, we find that on the task of code authorship identification, by using the code embeddings generated by GloVe, we achieve an accuracy of 79.3% and outperform the simpler approach in the work of Kang et al. [34] which uses the TF-IDF features. This finding is different from that of Kang et al. [34]. The difference may be caused by the different preprocessing steps on the training corpus and parameters for GloVe training. This finding suggests that researchers and developers should be careful with the parameters selection and corpus reprocessing. To further investigate the influence of these factors, we have conducted more than 20 new experiments with different experimental settings, the results are discussed in Section 6 and Section 7.

However, we observe that for some downstream tasks (e.g., source code classification and code authorship identification), different embedding techniques can result in diverse performance. In particular, for the source code classification task, using the embeddings trained by fastText can only have a 76.7% of test accuracy, compared to a 89.2% test accuracy when using the embeddings trained by GloVe. This finding suggests practitioners should be careful with the selection of code embedding techniques for different downstream tasks, as they may produce diverse results. On the other hand, we also observe that leveraging different embeddings may not always impact the performance of downstream tasks significantly. This observation is similar to that of prior studies [34, 43, 84].

Table 2. Evaluation results of using *GraphCodeVec* and baselines on the test sets in the six downstream tasks.

Downstream Tasks	Evaluation Metrics	Dataset	*GraphCodeVec*	Baselines			
				Word2vec	GloVe	fastText	code2vec
Code comment generation	BLEU	GitHub	20.7(-4.7%)^{*L}	21.1	**21.7**	19.9	21.0
	ROUGE	GitHub	36.1(-2.4%)^{*L}	36.9	**37.0**	36.0	36.3
Code authorship identification	Accuracy	Google Code Jam	**80.2(+1.1%)**	78.9	79.3	76.6	79.4
Code clones detection	F1	BCB	93.4(+0%)	93.4	93.4	93.4	93.4
		OJClone	**93.8(+8.7%)**^{*L}	88.4	86.3	84.6	93.4
		Avg	**93.6(+4.2%)**	90.9	89.8	89.0	93.4
Source code classification	Accuracy	OJ dataset	**93.7(+5.0%)**^{*L}	85.5	89.2	76.7	91.4
Logging statements prediction	BA	Airavata	**95.7(+0.7%)**	95.3	95.1	95.1	95.0
		Camel	**81.4(+0.4%)**	80.9	81.1	79.8	80.5
		CloudStack	86.3(-0.8%)	86.5	**87.0**	86.7	86.1
		Directory-Server	**89.1(+2.5%)**	87.9	86.9	88.6	87.6
		Hadoop	75.6(+0.7%)	**75.7**	75.0	74.4	73.9
		Avg	**85.6(+0.7%)**	85.3	85.0	84.9	84.6
Software defect prediction	F1	Ant 1.5 -> 1.6	42.7(+23.5%)^{*L}	35.9	34.6	36.0	**47.5**
		Ant 1.6 -> 1.7	**50.5(+13.0%)**^{*L}	43.9	44.7	44.2	46.8
		Camel 1.2 -> 1.4	**44.6(+5.3%)**^{*L}	41.9	42.3	41.8	42.7
		Camel 1.4 -> 1.6	46.7(-4.6%)^{*L}	45.3	49.0	45.8	**49.6**
		jEdit 3.2 -> 4.0	57.0(-2.3%)	53.4	58.3	53.6	**57.9**
		jEdit 4.0 -> 4.1	58.0(-3.3%)^{*L}	**61.0**	60.0	60.7	58.5
		Log4j 1.0 -> 1.1	**72.5(+9.1%)**^{*L}	64.0	66.5	63.1	68.5
		Lucene 2.0 -> 2.2	**67.0(+9.3%)**^{*L}	63.1	61.3	63.2	63.2
		Lucene 2.2 -> 2.4	65.2(+2.4%)^{*L}	**65.4**	63.7	65.3	62.4
		POI 1.5 -> 2.5	**84.6(+4.0%)**^{*L}	65.7	81.4	65.1	82.1
		POI 2.5 -> 3.0	**74.9(+2.6%)**^{*L}	72.5	73.0	72.2	74.0
		Xalan 2.4 -> 2.5	**52.5(+24.1%)**^{*L}	42.5	42.3	42.4	51.2
		Avg	**59.7(+5.8%)**	54.6	56.4	54.5	58.7

Note: The best results for each task and dataset are highlighted in bold. The numbers in the brackets indicate the relative change of *GraphCodeVec* to GloVe. The * means that the difference is statistically significant. The superscript L represents large effect size.

We find that by using different embeddings, although we can obtain different performances on different tasks, the difference is limited in some cases. For example, the different embedding techniques result in the same F1 score of 93.4% on the BCB dataset for code clones detection. One possible explanation is that the approaches used in the SE tasks are already powerful enough and there is enough dataset for learning a good model. Thus the impact of using different embedding techniques may be negligible.

Compared to other embedding techniques, *GraphCodeVec* **produces more stable results across all the downstream tasks and datasets.** Figure 6 shows the comparison of performance results produced by *GraphCodeVec* and baselines. In this figure, to show the difference to the best performance of each task, the results are scaled to the range of 0-100%, which is the ratio of the current method's performance to the best

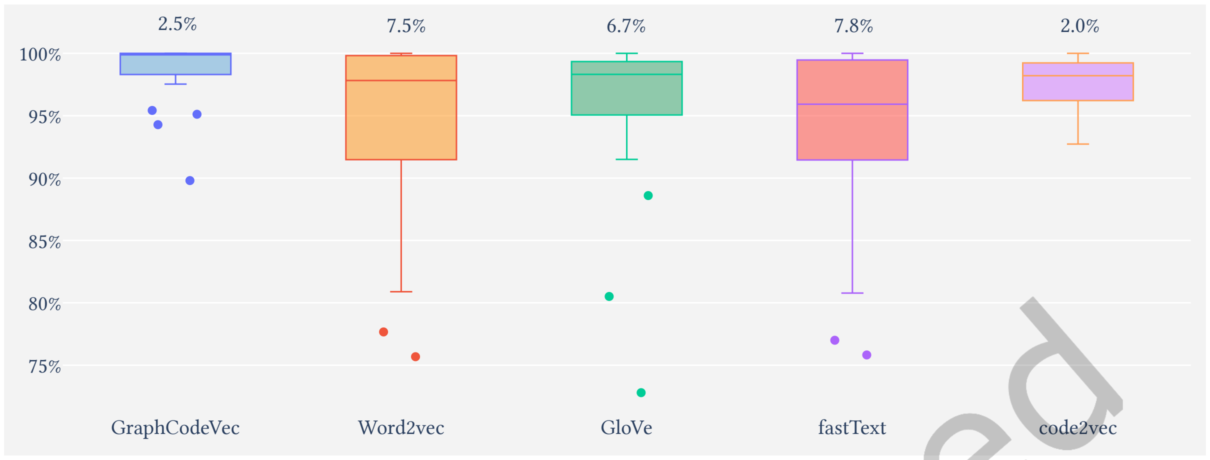


Fig. 6. Comparison of the results of *GraphCodeVec* and baselines. The horizontal axis represents all the evaluated methods; the vertical axis is the scaled performance of different methods, which is calculated as the ratio of the current method's performance to the best performance of one task. The numbers on top of each box are the corresponding coefficient of variance.

performance of one task, and in each boxplot, we consider all the measures for all the datasets (i.e., there are 23 data points in each boxplot). We did not rank all the results and check the overall ranking of each technique, because different downstream tasks use different measures with different ranges. Thus, for each downstream task, we normalize the performance of each technique against the best performance across all techniques (i.e., we use scaled performance). The scaled performance has consistent ranges across different downstream tasks thus allowing better comparison and visualization of the performance of different techniques. We also calculate the coefficient of variation (CV) for all the embedding techniques to quantify the variances. The results show that *GraphCodeVec* has a relatively lower variance among all the tasks. For example, the biggest relative difference appears in the task of software defect prediction on Ant dataset, which is 42.7% compared to the best result, 47.5%. Meanwhile, code2vec also has a stable performance on SE tasks, but its median is lower than that of *GraphCodeVec*. On the contrary, some embedding techniques lead to unstable results. For example, the fastText embedding technique achieves the best results on BCB dataset but the worst result (i.e., 84.6% compared to the best result, 93.4%) on OJClone dataset for the code clones detection task. Future work that depends on embedding techniques should consider a stable technique such as *GraphCodeVec*, otherwise the performance may be compromised.

Discussion

In the above paragraphs, we have quantitatively demonstrated the superiority of *GraphCodeVec* on the six downstream tasks, thus in this part, we would like to discuss the limitations of *GraphCodeVec*, as well as provide a qualitative analysis of the learned embeddings to complement our quantitative evaluation on downstream tasks.

Strengths and limitations

As shown in Table 2, although, overall, *GraphCodeVec* performs the best compared to all baseline approaches on five out of six downstream tasks, there is still a non-negligible gap between *GraphCodeVec* and GloVe on the task of code comment generation. By comparing the natural property of these tasks, we find that our *GraphCodeVec* works better on the classification tasks, such as code authorship identification, code clones detection and source code classification, etc., but not on the text generation task (i.e., code comment generation). For the classification tasks, the output is pre-defined labels and the embeddings only work in the first embedding layer which converts the source code tokens to real number vectors. However, for the task of code comment generation, we use an

Table 3. The agreement of the results between *GraphCodeVec* and baselines on the task of code clones detection.

	Word2vec	GloVe	fastText	code2vec
OJClone	0.82	0.77	0.75	0.89
BCB-Type-1	1.00	1.00	1.00	1.00
BCB-Type-2	1.00	1.00	1.00	1.00
BCB-Type-3	0.99	0.99	0.99	1.00
BCB-Type-4	0.99	0.99	0.99	0.99
BCB-Type-5	1.00	1.00	0.99	0.99

Note: We use Cohen's kappa to measure the agreement between the results generated by our method and that of the other four baselines.

encoder-decoder architecture where in the encoder part, similar to classification tasks, the embeddings are utilized to transform source code tokens into vectors, while in the decoder part, the same code embeddings (instead of word embeddings trained on comments or texts) are also used to convert the comment tokens (i.e., extracted from code comments) into vectors. As the code tokens and comment tokens are naturally different and thus, using only one code embeddings for both source code and code comments would confound the model, in other words, one good code embedding may not perform well on texts. Thus, we conclude that the poor performance may be caused by the fact that *GraphCodeVec* is able to capture the properties of the source code, but the learned knowledge is too specific for source code and thus cannot be transferred to natural language tokens. In future work, to improve the performance of *GraphCodeVec* on such text generation tasks, we can enhance the model by jointly learning the code and word embeddings based on the code and text information (e.g., documents and comments).

Moreover, we also observe that for some tasks or datasets, *GraphCodeVec* does not bring significant benefits. For example, *GraphCodeVec* has the same results on the BCB dataset with the other embedding techniques for code clones detection[16] but best performance (i.e., 8.7% improvement) on the OJClone dataset. Besides, similar results are also observed on the Camel dataset for logging statements prediction, where *GraphCodeVec* has a small improvement (i.e. 0.4%) compared to other embedding techniques but a relatively larger improvement (i.e., 2.5%) on the Directory-Server dataset. One explanation for this phenomenon is that larger training datasets may produce more powerful models and mitigate the differences between different embedding techniques. To obtain such fully trained models, one possible way is to collect enough training dataset. Thus, we check the sizes of datasets, and we find that the size of the BCB dataset is almost twice larger than that of OJClone dataset and the size of Camel dataset is more than five times larger than that of Directory-Server. The findings highlight that *GraphCodeVec* can work better for downstream tasks which have small training datasets. In other words, if the model cannot learn enough knowledge from the training dataset, we can use the embeddings generated by *GraphCodeVec*, as it can bring more external knowledge to the trained model, which is another ultimate goal of the pre-trained embeddings (i.e., learning useful knowledge from external datasets to improve the performance of downstream tasks).

Qualitative analysis of the learned embeddings

To further understand the trained embeddings, following prior work [6, 81], we discuss the characteristic of the trained embeddings from a qualitative perspective. We manually inspect code embeddings on one qualitative task, i.e., token similarity, as it is usually considered as the most straightforward feature to evaluate token representations [6, 56, 57, 81].

We select the target tokens and query their most similar tokens and then explore them intuitively. However, we should be aware that there is no explicit guideline for selecting the representative tokens, thus qualitative analysis might be subjective. In this work, we try our best to avoid the bias and select the subject tokens based

[16] We further check the results, and as Table 3 shows, all the code embeddings almost produce identical (clones) results on the BCB dataset.

on the following three criteria: (1) tokens should be well-known in the vocabulary - to ensure that evaluators are familiar with the characteristics of the tokens, (2) some of the tokens should provide different functionalities - to ensure that their embeddings have a low similarity and thus are located far from each other in the semantic space, and (3) some of the tokens should share similar functionalities - to ensure that their embeddings have a high semantic similarity.

Following prior work [6, 43, 81], we manually chose nine tokens from the vocabulary with different frequencies. All the selected tokens are either Java reserved words (e.g. "println" and "finally") or frequently used methods (e.g. "sort") and some of them share similar functionalities (e.g., "sort" vs. "comparator") and others provide different functionalities (e.g., "while" vs. "sort"). For each chosen token, we retrieve its 40 most similar tokens (using cosine similarity) according to different embeddings.

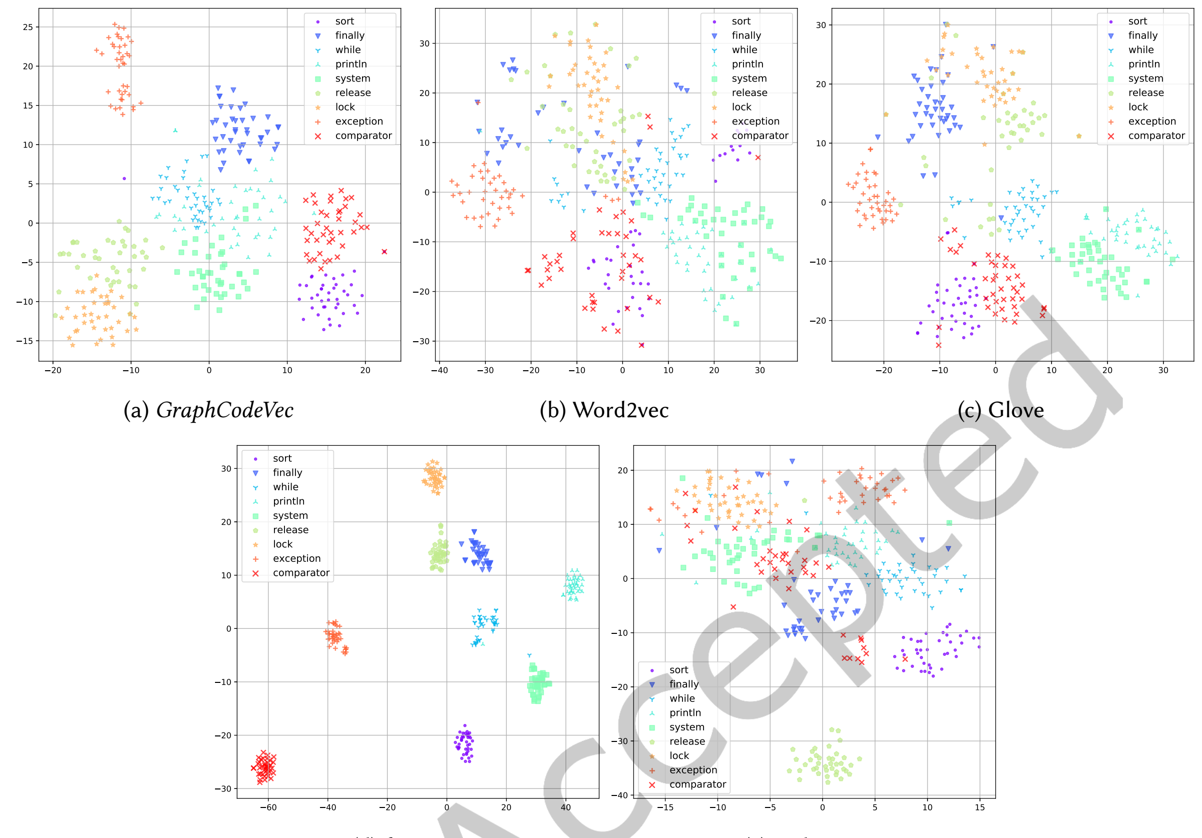
Visualization. In order to visualize high-dimension (i.e., 128 dimensions) embeddings, we compress them down to a low dimensional space (i.e., two dimensions) using t-SNE [51]. The idea of t-SNE is to reduce dimensions while trying to preserve the information of the original data points, namely, keeping similar tokens close on the plane while maximizing the distance between dissimilar tokens. We plot the target tokens and their most similar tokens. A good code embedding should project similar (e.g., similar functionalities) code tokens into the space with a shorter distance and project the unrelated code tokens far from each other.

As shown in Figure 7, for the visualization of our embeddings (i.e., *GraphCodeVec*), we see that several clusters are plotted closely, such as the clusters of "sort" and "comparator", which is consistent with the fact that they are frequently used together when performing sorting actions. Meanwhile, we do co-occurrence statistics (i.e., count the number of times that every two tokens are used together) of the listed keywords on the training corpus and find that for the token "comparator", "sort" is the one that occurs together more times than any other listed keywords, which conforms to our interpretation. Besides, for fastText, each target token's cluster is clearly separated with that of other target tokens. However, fastText cannot project similar tokens with a relatively shorter distance. For example, the cluster of "sort" is plotted closer to that of "system" or "while", instead of "comparator". For the comparison between *GraphCodeVec* and Glove, if we focus on the inter-relationships between these clusters, they both project "sort" and "comparator", "release" and "lock" as well as "system" and "println" closer in the space; however, if we focus on the intra-relationships within each cluster, GloVe projects the token "release" scattered across different clusters, while on the contrary each cluster of *GraphCodeVec* is more compact.

This finding confirms that *GraphCodeVec* can project syntactically similar tokens to the vector space with a relatively short distance. Although the visualization cannot provide us with direct measurement of the quality of the embeddings, it still helps us gain insights into the characteristics of the resulting embeddings.

Meanwhile, we also try to manually inspect the top-10 nearest neighbors of the given token using cosine similarity. For example, given the target token, "while", we retrieve its top-10 most similar tokens and examine whether the token, "for" appears in the list or not. The results show that the "for" token only appears in the top-10 nearest neighbors of "while" when retrieved using the embeddings generated by Word2vec. This observation shows that the trained embeddings may return some results that are different from the prior knowledge of developers or researchers and hard to interpret [34]. This finding also suggests the necessity of exploring the characteristics of the learned embeddings from different perspectives and shows that Word2vec may perform better than other embeddings when used for retrieving the similar tokens.

However, the above findings may not violate the first visualization part of the qualitative analysis. For the visualization using the t-SNE, we retrieve 40 most similar tokens for the given target token and plot the clustering figures for each token, which is different from retrieving top-k nearest neighbors and checking whether the expected tokens are in the list or not.



(a) *GraphCodeVec* (b) Word2vec (c) Glove

(d) fastText (e) code2vec

Fig. 7. Visualization of the target tokens and their 40 most similar tokens. The horizontal and vertical axes show the two dimensions that are reduced from the original 128 dimensions using the t-SNE.

Summary

Our evaluation results show that *GraphCodeVec* achieves the best results than all the baselines in five out of six downstream tasks. Besides, *GraphCodeVec* has the most stable results on all downstream tasks. Future research and practice that rely on code embeddings should be careful with the selection of code embedding techniques for specific downstream tasks, as they may produce diverse results.

RQ2: How does the structural context information of the source code impact the effectiveness of the embeddings generated by *GraphCodeVec*?

Motivation

In RQ1, our results show that our GCN-based approach *GraphCodeVec* has the most stable performance and outperforms the baseline approaches. On one hand, prior studies [6, 11, 29, 83, 95] show that incorporating the structural information (e.g., AST structure of source code) of a particular source code of interest may provide promising results in some software engineering (SE) tasks that rely on neural network-based techniques and code is structured by its nature (e.g, class, method and block) and thus the code embeddings may benefit from the structural representation. On the other hand, there are some studies that treat the source code as plain text and achieve satisfactory results [16, 20, 25]. Therefore, in this research question, we aim to understand how the structural information (i.e., the graph context extracted from the ASTs) affects the performance of *GraphCodeVec*.

Approach

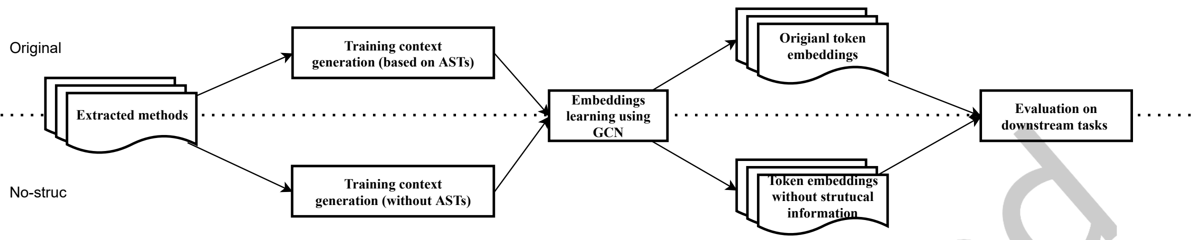

Original
No-struc
Extracted methods
Training context generation (based on ASTs)
Training context generation (without ASTs)
Embeddings learning using GCN
Origianl token embeddings
Token embeddings without strutucal information
Evaluation on downstream tasks

Fig. 8. The overall design of the approach for RQ2. In this figure, original refers to our *GraphCodeVec*. No-struc refers to the method which does not utilize the ASTs while keeping other settings the same as *GraphCodeVec*.

In RQ1, the training context for generating code embeddings by *GraphCodeVec* is constructed based on the graph representation of source code, which preserves the structural information of source code. Thus, in this section, to analyze the impact of our graph context on generating the embeddings, we design an ablation experiment on these six downstream tasks. In this experiment, as shown in Figure 8, the embedding training technique is the same (i.e., GCN), with the only difference of the training context. We generate the training context from the extracted methods without the structural information (unlike our *GraphCodeVec*, which generates the training context based on ASTs) and then feed it into the GCN model to obtain the code embeddings. We then compare the performance of the embeddings trained with and without the AST structure using the same training technique (i.e., GCN). As the training context is the only changing factor, thus we state that the performance changes are caused by the different training context. That is if the embeddings with the AST information performs better, then we can conclude that our embeddings can benefit from the utilizing the ASTs. More specifically, we treat the source code as plain text and do not consider the AST relationship among the tokens. Below we discuss the details of how we extract the training context and incorporate it in GCN.

First, the source code is transformed into plain text, of which all the tokens are lowercased, and the non-identifiers (including punctuations such as ";" and operators such as "=") are removed. As the training model (i.e., GCN) requires graph-format data as input, to make source code suitable for training, we then adopt a local window to convert the plain text into graphs. Given a target token, all the surrounding tokens located in this window are connected to the target token in the graph by an edge. For example, given the code snippet in Section 3.1.1, assuming the target token is "public", then "void", "printname", "string", "someone", and "name" are the neighboring nodes in the generated training context. We construct a graph context for the target token "public " in the format shown in Figure 9:

```
public void printname string someone name someone system out println name
```

More specifically, the target token "public" is the central node of this graph and connects to all the other five nodes, among which there is no edge between each other. We then feed the generated context to the embedding learning phase. Finally, the learned embeddings are evaluated on the six downstream tasks. In our experiment, we set the window size to five on each side surrounding the target token, which is by default used in previous work [10, 42, 56, 57, 73]. Note that a larger window size would be able to capture more broad context, but with the possibility of introducing noise as the context tokens might not be tightly related to the target token. On the

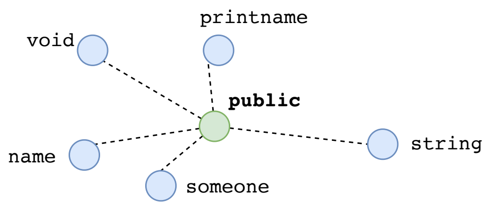


Fig. 9. An overview of the constructed graph context based on the plain text.

Table 4. Evaluation results of *GraphCodeVec* with and without utilizing the graph context extracted from the ASTs.

Dowsnstream Tasks	Code comment generation		Code authorship identification	Code clones detection			Source code classification	Logging statements prediction					
Datasets	GitHub		Google Code Jam	BCB	OJClone	**Avg.**	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop	**Ang.**
Metrics	BLEU	ROUGE	Accuracy	F1			Accuracy	BA					
Original	**20.7**	**36.1**	**80.2**	**93.4**	**93.8**	**93.6**	**93.7**	**95.7**	**81.4**	**86.3**	**89.1**	**75.6**	**85.6**
No-struc	**20.7**	36.0	80.0	**93.4**	93.5	93.5	93.6	95.3	80.6	86.0	87.7	74.7	84.8

Dowsnstream Tasks	Software defect predicttion												
Datasets	Ant 1.5->1.6	Ant 1.6->1.7	Camel 1.2->1.4	Camel 1.4->1.6	jEdit 3.2->4.0	jEdit 4.0->4.1	Log4j 1.0->1.1	Lucene 2.0->2.2	Lucene 2.2->2.4	POI 1.5->2.5	POI 2.5->3.0	Xalan 2.4->2.5	**Avg.**
Metrics	F1												
Original	42.7	**50.5**	**44.6**	46.7	57.0	58.0	**72.5**	**67.0**	65.2	**84.6**	**74.9**	**52.5**	**59.7**
No-struc	**43.9**	48.3	43.5	**48.3**	**57.6**	**58.9**	66.7	59.8	**65.9**	82.7	74.6	51.9	58.5

contrary, a smaller window size may contain more focused information about the target token but may not be able to capture sufficient context.

Results

Overall, the graph context extracted from the ASTs can improve the performance of the code embeddings generated by *GraphCodeVec***; however,** *GraphCodeVec* **may not always significantly benefit from the utilization of the graph context.** Table 4 shows the results of comparing the performance of the original *GraphCodeVec* and the one that does not use the structural information. In the table, Original refers to our *GraphCodeVec*. No-struc is the variant of *GraphCodeVec*, which only utilizes the plain text of the source code instead of the graph context extracted from the ASTs while keeping other settings the same as *GraphCodeVec*. In total, as Table 4 shows, we find that our original *GraphCodeVec* outperforms No-struc (i.e., the variant of *GraphCodeVec* that does not consider the graph context extracted from the ASTs) in all six downstream tasks. The comparison results demonstrate that even though we train the embeddings using the same model, utilizing the graph context extracted from the ASTs can help improve the performance of the embeddings. For example, on the logging statements prediction task, by training the embeddings using the graph context, *GraphCodeVec* has a overall balanced accuracy of 85.6% compared to 84.8% without the graph context.

On the other hand, for some tasks, the improvement is limited, and incorporating the graph context extracted from the ASTs may cause performance degradation on some datasets. For example, on the task of code authorship identification, the overall improvement is only 0.2% and 0.1% for the task if source code classification. In addition, in almost half of the datasets of the software defect prediction task, utilizing the graph context degrades the performance of *GraphCodeVec*. The result indicates the limited effect of incorporating the graph context in some cases. One possible reason is that some tasks may not be sensitive to the structural information of the source code, thus using a structured code representation may not improve the performance significantly.

We further conducted two more experiments with different window sizes (i.e., two and eight) and the result (shown in Table 6) shows that paying more attention to closer neighbors (a smaller window size) would bring more benefits. As when we reduce the window size to two, we observe statistically significant improvement in five out of seven (i.e., seven cases have significant performance changes among which five cases have improvement) cases (71.4%), and when we increase the window size to eight, we observe statistically significant improvement in four out of eight cases (50%).

Summary

Although overall, the structural information extracted from the ASTs can benefit *GraphCodeVec* in producing code embeddings for the downstream SE tasks, there may be cases where the structural information may not provide additional benefit.

RQ3: How does the GCN model impact the effectiveness of the embeddings generated by *GraphCodeVec*?

Motivation

Prior work [43] proposes a novel word embedding approach for NLP tasks that adopts a shallow, two-layer neural network instead of Graph Convolutional Networks to incorporate the syntactic information between words and achieves promising results. Their results raise our concern about whether a simple two-layer neural network is powerful enough to model the syntactic information within the corpus. Therefore, in this research question, we want to study how the GCN model affects the performance of *GraphCodeVec* for generating the code embeddings for the downstream tasks.

Approach

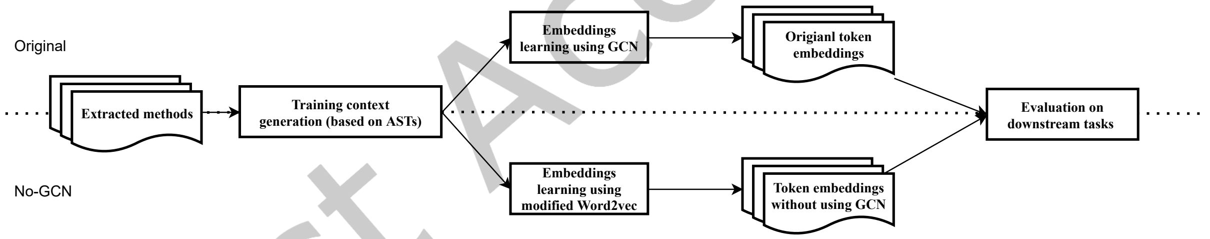

Original
No-GCN
Extracted methods
Training context generation (based on ASTs)
Embeddings learning using GCN
Origianl token embeddings
Embeddings learning using modified Word2vec
Token embeddings without using GCN
Evaluation on downstream tasks

Fig. 10. The overall design of the approach for RQ3. In this figure, original refers to our *GraphCodeVec*. No-GCN refers to the method which does not utilize the GCN for embedding learning.

In RQ2, we analyze the impact of structural context information on the effectiveness of the embeddings generated by GCN. We train two different code embeddings using different training contexts (i.e., with and without AST information) but the same training embedding technique (i.e., GCN). In this section, to analyze the impact of the GCN model on generating the embeddings, similar to RQ2, we design an ablation experiment on these six downstream tasks. In this experiment, as shown in Figure 10, we adopt two different training techniques with the same training context, which both consider the structural information for code embedding learning. Specifically, we implement another method, namely, No-GCN [43] for comparison. No-GCN uses a similar approach to extract the graph context from the ASTs but adopts a shallow, two-layer neural network to train embeddings. No-GCN was originally proposed by Li et al. [43] for learning word embeddings by incorporating the dependency information between words in a sentence. Li et al. [43] modify the original Word2vec model and integrates the syntactic dependency information between words into the embeddings. In this work, we customize No-GCN by replacing the syntactic dependency with the AST paths extracted from the source code.

Table 5. Evaluation results of utilizing different models to train the code embeddings from the graph context.

Dowsnstream Tasks	Code comment generation		Code authorship identification	Code clones detection			Source code classification	Logging statements prediction					
Datasets	GitHub		Google Code Jam	BCB	OJClone	**Avg.**	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop	**Avg.**
Metrics	BLEU	ROUGE	Accuracy	F1			Accuracy	BA					
Original	20.7	36.1	**80.2**	**93.4**	**93.8**	**93.6**	**93.7**	95.7	**81.4**	**86.3**	**89.1**	**75.6**	**85.6**
No-GCN	**21.4**	**36.7**	79.8	**93.4**	91.0	92.2	90.1	**95.9**	80.4	**86.3**	87.4	74.7	84.9

Dowsnstream Tasks	Software defect predicttion												
Datasets	Ant 1.5->1.6	Ant 1.6->1.7	Camel 1.2->1.4	Camel 1.4->1.6	jEdit 3.2->4.0	jEdit 4.0->4.1	Log4j 1.0->1.1	Lucene 2.0->2.2	Lucene 2.2->2.4	POI 1.5->2.5	POI 2.5->3.0	Xalan 2.4->2.5	**Avg.**
Metrics	F1												
Original	42.7	**50.5**	**44.6**	46.7	57.0	**58.0**	**72.5**	**67.0**	**65.2**	**84.6**	**74.9**	**52.5**	**59.7**
No-GCN	**43.3**	49.8	44.1	**49.6**	**58.6**	57.3	69.1	63.0	62.0	77.2	71.8	48.4	57.8

No-GCN uses a similar way for extracting the training context from the source code. It first transforms the source code into ASTs, then traverses the trees to collect triples, where the first and last elements are the leaf nodes of an AST and the second element is the AST path connecting the other two elements. For example, given a target token, “public” in Figure 1, it starts from “public” and keep traversing the tree until it reaches another leaf node (e.g., “void”), and the traversing path is recorded. By doing this, it can collect a set of triples that can be used for training the code embeddings. Similar to our work, the number of triples is also limited by the length of an AST path.

Different from the GCN used in this paper, No-GCN modifies the original Word2vec model to include the AST paths instead of only considering the tokens (more details can be found in the work [43]).

Results

The comparison results with No-GCN show the advantage of using GCN for modeling the graph context. Our experimental results for comparing *GraphCodeVec* with No-GCN on the six SE tasks are presented in Table 5. As Table 5 shows, we find that overall our *GraphCodeVec* has the best results in five out of six downstream tasks. For example, on the source code classification task, No-GCN achieves a test accuracy of 90.1%, while *GraphCodeVec* reaches 93.7%. The comparison results show that GCN are more suitable for representing the source code as graphs and capturing the syntactic structure of the source code when generating code embeddings. However, similar to the results in RQ1, *GraphCodeVec* also does not reach the best results on the task of code comment generation. This may be due to the fact that *GraphCodeVec* is good at capturing the properties of source code, while the task of code comment is for generating the natural language texts, and thus our approach cannot perform well. Besides, we find that the improvement for the task of code authorship identification is limited, with only 0.4% absolute increase. This observation further confirms our findings in RQ2 that some tasks may be not sensitive to the structural information of the source code.

Compared to the impact of the graph context in RQ2, we find that the GCN model has a more stable influence on the performance of *GraphCodeVec*. On the one hand, in RQ2, replacing the graph context with plain text causes a relatively smaller performance decrease on the downstream SE tasks compared to changing the training model in RQ3. For example, there is a 0.1% degradation of the test accuracy on the source code classification task after changing the training context in RQ2, compared to 3.6% degradation after changing the training model in RQ3. On the other hand, in RQ2, we do not observe improvement by using the graph context on almost half of the datasets of the software defect prediction task; while in RQ3, we observe improvement by using the GCN model on nine datasets. Our results suggest the promising research direction of using graph-based deep learning methods for SE tasks.

Summary

Instead of using a vanilla neural network, the use of Graph Convolutional Networks can robustly benefit the performance of *GraphCodeVec* for training code embeddings for the downstream SE tasks.

Table 6. Evaluation results of code embeddings generated by *GraphCodeVec* with different thresholds and model hyperparameters.

Stage	Name	Value	Downstream Tasks																						
			Code comment generation		Code authorship identification	Code clones detection		Source code classification	Logging statements prediction					Software defect predicttion											
			GitHub		Google Code Jam	BCB	OJClone	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop	Ant 1.5->1.6	Ant 1.6->1.7	Camel 1.2->1.4	Camel 1.4->1.6	jEdit 3.2->4.0	jEdit 4.0->4.1	Log4j 1.0->1.1	Lucene 2.0->2.2	Lucene 2.2->2.4	POI 1.5->2.5	POI 2.5->3.0	Xalan 2.4->2.5
			BLEU	ROUGE	Accuracy	F1		Accuracy	BA					F1											
		Default	20.7	36.1	80.2	93.4	93.8	93.7	95.7	81.4	86.3	89.1	75.6	42.7	50.5	44.6	46.7	57	58	72.5	67	65.2	84.6	74.9	52.5
Training context generation	Edge length	6	20.8 (+0.5%)	36.2 (+0.3%)	80 (-0.2%)	93.4 (+0.0%)	94.1 (+0.3%)	93.3 (-0.4%)	95.8 (+0.1%)	81.8 (+0.5%)	87.3 (+1.2%)	88.1 (-1.1%)	74.4 (-1.6%)	**48.9L (+14.5%)**	**52.2L (+3.4%)**	**43.3L (-2.9%)**	47.6 (+1.9%)	**59.4L (+4.2%)**	**55.2L (-4.8%)**	**67.6L (-6.8%)**	**64.3L (-4.0%)**	65.7 (+0.8%)	**82.5L (-2.5%)**	**72.9L (-2.7%)**	**49.7L (-5.3%)**
		10	20.8 (+0.5%)	36.2 (+0.3%)	80 (-0.2%)	93.4 (+0.0%)	93.1 (-0.7%)	93.4 (-0.3%)	94 (-1.8%)	79.2 (-2.7%)	86.4 (+0.1%)	**87.7S (-1.6%)**	74.9 (-0.9%)	**48.2L (+12.9%)**	50.4 (-0.2%)	**41.0L (-8.1%)**	46.9 (+0.4%)	56.2 (-1.4%)	**55.4L (-4.5%)**	**66.5L (-8.3%)**	**60.8L (-9.3%)**	**62.4L (-4.3%)**	**81.9L (-3.2%)**	**72.2L (-3.6%)**	**50.6L (-3.6%)**
	Unique node	50	20.7 (+0.0%)	36.2 (+0.3%)	80.1 (-0.1%)	93.5 (+0.1%)	94.6 (+0.9%)	93.9 (+0.2%)	95.6 (-0.1%)	81.1 (-0.4%)	86.1 (-0.2%)	**86.7L (-2.7%)**	74.7 (-1.2%)	42.5 (-0.5%)	51.1 (+1.2%)	44.4 (-0.4%)	46.5 (-0.4%)	57 (+0.0%)	58.3 (+0.5%)	70.9 (-2.2%)	66.7 (-0.4%)	65.2 (+0.0%)	83.9 (-0.8%)	74.5 (-0.5%)	53.1 (+1.1%)
		80	20.7 (+0.0%)	36.1 (+0.0%)	80 (-0.2%)	93.4 (+0.0%)	94.5 (+0.7%)	94 (+0.3%)	96.4 (+0.7%)	80.3 (-1.4%)	86.8 (+0.6%)	87.4 (-1.9%)	**73.4M (-2.9%)**	43.6 (+2.1%)	50.7 (+0.4%)	44.3 (-0.7%)	46.5 (-0.4%)	56.9 (-0.2%)	58.1 (+0.2%)	72.3 (-0.3%)	66.7 (-0.4%)	64.1 (-1.7%)	84.5 (-0.1%)	75.1 (+0.3%)	52.9 (+0.8%)
	Edge	400	20.7 (+0.0%)	36.2 (+0.3%)	80 (-0.2%)	**93.3N (-0.1%)**	93.8 (+0.0%)	93.4 (-0.3%)	95.3 (-0.4%)	80.8 (-0.7%)	86.9 (+0.7%)	**86.7M (-2.7%)**	75 (-0.8%)	43.9 (+2.8%)	50.3 (-0.4%)	44 (-1.3%)	46.5 (-0.4%)	57.2 (+0.4%)	58.4 (+0.7%)	72.4 (-0.1%)	**66.5L (-0.7%)**	64.7 (-0.8%)	84.3 (-0.4%)	74.7 (-0.3%)	52.6 (+0.2%)
		600	**20.8S (+0.5%)**	36.2 (+0.3%)	80 (-0.2%)	93.4 (+0.0%)	93.7 (-0.1%)	93.6 (-0.1%)	95.3 (-0.4%)	79.5 (-2.3%)	87 (+0.8%)	87.8 (-1.5%)	75.4 (-0.3%)	43.4 (+1.6%)	50.8 (+0.6%)	44 (-1.3%)	**45.9L (-1.7%)**	57.1 (+0.2%)	58.3 (+0.5%)	72.1 (-0.6%)	66.5 (-0.7%)	65 (-0.3%)	84.2 (-0.5%)	75.2 (+0.4%)	53.1 (+1.1%)
	Window (c.f., RQ2)	2	20.7 (+0.0%)	36.1 (+0.3%)	80 (+0.0%)	93.5 (+0.1%)	92.2 (-1.4%)	93 (-0.6%)	95.3 (+0.0%)	79.8 (-1.0%)	87 (+1.2%)	87.5 (-0.2%)	75.1 (+0.5%)	43.2 (-1.6%)	**54.6L (+13.0%)**	**42.2L (-3.0%)**	48.8 (+1.0%)	**61.1L (+6.1%)**	59 (+0.2%)	**62.5L (-6.3%)**	**61.3L (+2.5%)**	**63.0L (-4.4%)**	**83.9L (+1.5%)**	74.3 (-0.4%)	52.3 (+0.8%)
		5	20.7	36	80	93.4	93.5	93.6	95.3	80.6	86	87.7	74.7	43.9	48.3	43.5	48.3	57.6	58.9	66.7	59.8	65.9	82.7	74.6	51.9
		8	20.6 (-0.5%)	36 (+0.0%)	80.3 (+0.4%)	93.5 (+0.1%)	**92.1L (-1.5%)**	93.9 (+0.3%)	95.1 (-0.2%)	79.9 (-0.9%)	**87.5L (+1.7%)**	87 (-0.8%)	74.2 (-0.7%)	42.7 (-2.7%)	**55.1L (+14.1%)**	**42.3L (-2.8%)**	48.5 (+0.4%)	**61.2L (+6.3%)**	58.6 (-0.5%)	**62.4L (-6.4%)**	**61.2L (+2.3%)**	**63.4L (-3.8%)**	82.9 (+0.2%)	74.1 (-0.7%)	52.7 (+1.5%)
Embedding learning	Layer	3	**20.5M (-1.0%)**	**35.8L (-0.8%)**	80.2 (+0.0%)	93.4 (+0.0%)	93.5 (-0.3%)	94 (+0.3%)	94.9 (-0.8%)	82 (+0.7%)	86.7 (+0.5%)	88.2 (-1.0%)	75.6 (+0.0%)	**46.2L (+8.2%)**	**47.5L (-5.9%)**	**40.1L (-10.1%)**	**45.1L (-3.4%)**	57.1 (+0.2%)	**56.9L (-1.9%)**	**64.4L (-11.2%)**	**63.6L (-5.1%)**	**67.8L (+4.0%)**	**81.3L (-3.9%)**	**73.2L (-2.3%)**	**53.7L (+2.3%)**
		5	**20.0L (-3.4%)**	**35.4L (-1.9%)**	80.1 (-0.1%)	93.4 (+0.0%)	**90.4L (-3.6%)**	93.2 (-0.5%)	94.8 (-0.9%)	82.1 (+0.9%)	87.2 (+1.0%)	**86.7L (-2.7%)**	74.5 (-1.5%)	**41.1L (-3.7%)**	**41.5L (-17.8%)**	**34.5L (-22.6%)**	**44.5L (-4.7%)**	**47.7L (-16.3%)**	**47.2L (-18.6%)**	**45.8L (-36.8%)**	0	**57.3L (-12.1%)**	**76.8L (-9.2%)**	**72.1L (-3.7%)**	**55.3L (+5.3%)**
	Dim.	50	20.6 (-0.5%)	36 (-0.3%)	**75.0L (-6.5%)**	93.4 (+0.0%)	**91.0L (-3.0%)**	93.3 (-0.4%)	94.6 (-1.1%)	80 (-1.7%)	85.1 (-1.4%)	87.9 (-1.3%)	75.1 (-0.7%)	**48.7L (+14.1%)**	**48.1L (-4.8%)**	**39.6L (-11.2%)**	46.6 (-0.2%)	**59.2L (+3.9%)**	57.3 (-1.2%)	72 (-0.7%)	**61.8L (-7.8%)**	**61.4L (-5.8%)**	**82.7L (-2.2%)**	74 (-1.2%)	52.8 (+0.6%)
		300	20.7 (+0.0%)	36.1 (+0.0%)	81.7 (+1.9%)	93.4 (+0.0%)	**95.6L (+1.9%)**	92.5 (-1.3%)	95.6 (-0.1%)	80.3 (-1.4%)	87.4 (+1.3%)	**87.5M (-1.8%)**	75.3 (-0.4%)	42.3 (-0.9%)	**52.4L (+3.8%)**	44.1 (-1.1%)	**48.2L (+3.2%)**	**60.8L (+6.7%)**	57.8 (-0.3%)	**68.4L (-5.7%)**	**62.9L (-6.1%)**	64.9 (-0.5%)	**83.8L (-0.9%)**	**72.1L (-3.7%)**	**50.6L (-3.6%)**
	Neg.	2	20.7 (+0.0%)	36.1 (+0.0%)	79.6 (-0.7%)	93.4 (+0.0%)	93.4 (-0.4%)	93.5 (-0.2%)	95.9 (+0.2%)	80.7 (-0.9%)	86.3 (+0.0%)	88.4 (-0.8%)	74.6 (-1.3%)	**50.8L (+19.0%)**	**48.7L (-3.6%)**	**40.1L (-10.1%)**	**48.3L (+3.4%)**	**59.3L (+4.0%)**	58.2 (+0.3%)	**65.4L (-9.8%)**	**60.9L (-9.1%)**	**63.0L (-3.4%)**	**83.0L (-1.9%)**	**73.3L (-2.1%)**	**49.3L (-6.1%)**
		10	**20.8S (+0.5%)**	36.2 (+0.3%)	80.7 (+0.6%)	93.4 (+0.0%)	93.6 (-0.2%)	93.1 (-0.6%)	95.6 (-0.1%)	80.2 (-1.5%)	86.4 (+0.1%)	88.3 (-0.9%)	76 (+0.5%)	**47.0L (+10.1%)**	**49.3L (-2.4%)**	44.4 (-0.4%)	**47.8L (+2.4%)**	**60.2L (+5.6%)**	**59.4L (+2.4%)**	**67.9L (-6.3%)**	**63.2L (-5.7%)**	**62.2L (-4.6%)**	**81.3L (-3.9%)**	**72.8L (-2.8%)**	**49.4L (-5.9%)**
	Batch size	32	**20.9S (+1.0%)**	**36.3M (+0.6%)**	80.3 (+0.1%)	93.4 (+0.0%)	**94.8S (+1.1%)**	**91.8L (-2.0%)**	95.5 (-0.2%)	80.9 (-0.6%)	87 (+0.8%)	87.7 (-1.6%)	73.7 (-2.5%)	**44.4L (+4.0%)**	**51.4L (+1.8%)**	**42.3L (-5.2%)**	**49.4L (+5.8%)**	**60.0L (+5.3%)**	**56.6L (-2.4%)**	**66.0L (-9.0%)**	67.1 (+0.1%)	**66.9L (+2.6%)**	**83.3L (-1.5%)**	**73.6L (-1.7%)**	**49.9L (-5.0%)**
		128	**20.6S (-0.5%)**	36 (-0.3%)	80 (-0.2%)	93.3 (-0.1%)	93.8 (+0.0%)	93.8 (+0.1%)	95.7 (+0.0%)	80.3 (-1.4%)	87.2 (+1.0%)	87.1 (-2.2%)	74.9 (-0.9%)	**44.5L (+4.2%)**	**52.0L (+3.0%)**	**40.3L (-9.6%)**	**44.6L (-4.5%)**	57.1 (+0.2%)	58.4 (+0.7%)	71.2 (-1.8%)	**60.9L (-9.1%)**	**63.2L (-3.1%)**	**83.5L (-1.3%)**	74.8 (-0.1%)	**53.5L (+1.9%)**
	Dropout rate	0.2	20.8 (+0.5%)	36.1 (+0.0%)	80.2 (+0.0%)	93.4 (+0.0%)	93.8 (+0.0%)	92.7 (-1.1%)	96.6 (+0.9%)	81.2 (-0.2%)	87.2 (+1.0%)	87.3 (-2.0%)	74.2 (-1.9%)	41.8 (-2.1%)	**46.9L (-7.1%)**	44 (-1.3%)	47.6 (+1.9%)	**59.8L (+4.9%)**	59.1 (+1.9%)	**66.7L (-8.0%)**	**62.6L (-6.6%)**	65.2 (+0.0%)	**80.0L (-5.4%)**	**73.3L (-2.1%)**	**53.9L (+2.7%)**
		0.5	**20.8S (+0.5%)**	**36.2S (+0.3%)**	80.1 (-0.1%)	93.4 (+0.0%)	93.7 (-0.1%)	93.8 (+0.1%)	94.4 (-1.4%)	80 (-1.7%)	86.7 (+0.5%)	87.4 (-1.9%)	75.3 (-0.4%)	**45.6L (+6.8%)**	51.1 (+1.2%)	**42.6L (-4.5%)**	**45.1L (-3.4%)**	**58.8L (+3.2%)**	57.3 (-1.2%)	**66.6L (-8.1%)**	**61.6L (-8.1%)**	64.7 (-0.8%)	**81.2L (-4.0%)**	**72.6L (-3.1%)**	**51.4L (-2.1%)**
	Epoch	5	**21.1L (+1.9%)**	**36.4L (+0.8%)**	80.3 (+0.1%)	93.4 (+0.0%)	94.3 (+0.5%)	**90.8L (-3.1%)**	95.3 (-0.4%)	81.2 (-0.2%)	86.9 (+0.7%)	87.7 (-1.6%)	74.5 (-1.5%)	42 (-1.6%)	**52.4L (+3.8%)**	**46.1L (+3.4%)**	**48.0L (+2.8%)**	**58.8L (+3.2%)**	**62.3L (+7.4%)**	**68.9L (-5.0%)**	**62.9L (-6.1%)**	**62.7L (-3.8%)**	83.7 (-1.1%)	74.5 (-0.5%)	**50.6L (-3.6%)**
		10	**21.2L (+2.4%)**	**36.6L (+1.4%)**	80.4 (+0.2%)	**93.3M (-0.1%)**	94.2 (+0.4%)	**90.9L (-3.0%)**	95.2 (-0.5%)	79.8 (-2.0%)	86.8 (+0.6%)	88.4 (-0.8%)	75.8 (+0.3%)	42 (-1.6%)	**51.4M (+1.8%)**	**47.2L (+5.8%)**	47 (+0.6%)	57.9 (+1.6%)	**62.6L (+7.9%)**	**69.9L (-3.6%)**	**63.0L (-6.0%)**	**62.4L (-4.3%)**	**81.9L (-3.2%)**	**72.3L (-3.5%)**	**47.6L (-9.3%)**

Note: The results that are significantly different from that of the default settings are highlighted in bold. The numbers in the brackets indicate the relative change to the default settings of *GraphCodeVec*. The letters S, M, L, and N represent small, medium, large and negligible effect sizes, respectively.

6 DISCUSSION

In Section 5, we have conducted several experiments and shown that our task-agnostic *GraphCodeVec* can effectively be applied to different downstream tasks. In this section, we would like to have a discussion about the impact of different model parameters and the results of repeating our experiments with different data sampling using a 10-fold cross-validation.

Impact of modeling parameters. *GraphCodeVec* contains two stages (i.e., training context generation and embedding learning) for learning the code embeddings where some thresholds and model hyperparameters are involved for generating the training corpus as well as defining the GCN structure. In this work, we either simply follow previous work or use the default settings of the model and do not try to fine-tune the parameters for fitting into different tasks. In this part, to examine whether our generated code embeddings can be further improved and assess the impact of the hyperparameters on the quality of the generated code embeddings, we conduct more than 20 new experiments with different parameter settings and the corresponding results are listed in Table 6. We also conduct a Wilcoxon signed-rank statistical test to check whether there is a significant performance change between the performance of the model using the new configured parameters and that of the model using the default parameters results are significant, for significant changes, we further conduct Cliff's delta statistic to check the effect sizes. The significant changes are marked in bold as shown in Table 6.

The performance of *GraphCodeVec* on some tasks can be further improved by fine-tuning the model parameters. For example, if we set the dimensionality of the embeddings to 300, we observe a significant performance increase of the F1 score (i.e., from 93.8 to 95.6) on the OJClone dataset for the task of code clones detection. Meanwhile, changing the parameters can also decrease the performance of *GraphCodeVec*. In our experiment, using a smaller dimensionality (i.e., 50) of the embeddings leads to performance degradation almost on all the tasks and datasets. The reason may be that a small dimensionality of the embeddings cannot preserve the properties of the tokens of high dimensional spaces, leading to the degradation of the quality of learned embeddings. On the contrary, using a relatively larger dimensionality can preserve more information and improve the quality of the generated embeddings. Another possible reason may be underfitting of the models used in downstream tasks, as a smaller input dimension means a simpler model and fewer weights to be learned during model training, and thus the model cannot capture the relationship between the input and output variables accurately, generating a high error rate on the testing data. To examine the impact of the number of training epochs, we provide another two experiments with more training epochs (i.e., five and ten epochs). During the embedding learning phase, the training loss reduces from 1.79 at the beginning of the first training epoch to 0.73 at the end of the first epoch, which further drops to 0.47 and 0.48 at the end of the fifth and the tenth epochs, respectively. Further, we also evaluated the quality of the newly generated embeddings on the downstream tasks. Overall, as shown in Table 6, when the training epoch increases to five, we observe seven significant improvements on the evaluation of the downstream tasks. On the other hand, we also observe that there are significant degradations on some (i.e. five) of the datasets from source code classification and software defect prediction tasks. The results indicate that the training epochs have different effects (either positive or negative depending on the downstream tasks) on the quality of the generated embeddings, and developers can fine-tune these epochs specially for their task. In our experiments, as we stated in Section 4.4, we avoid only fine-tuning these settings only for our method aiming for a better performance.

A model with more layers (i.e., deeper model) may not guarantee a better performance, especially for GCN models. In our supplement experiments, we increase the depth (i.e., layers) of *GraphCodeVec* from one to three and five, we find a continuous performance degradation for almost all the tasks, the model even returns a F1 score of zero for the task of software defect prediction on the Lucene project. This finding is consistent with previous works [**? ? ? ? ?**], that is with an increased depth (i.e., number of layers), GCN tends to easily overfit the training data and suffer a continuous performance degradation. Except for reducing the number of layers (we set the layer to one to avoid overfitting and performance degradation, c.f., Section 4.2) of the model, researchers [**?**] also propose to use dropout to prevent overfitting. Dropout is a regularization technique which randomly drops out the units along with their connections of neural networks. To examine the impact of using dropout on the quality of our generated code embeddings, we have experimented with two different dropout rates (i.e., 0.2 and 0.5). Overall, as Table 6 shows, we do not have obvious performance improvements when using different dropout ratios. This can be explained by the fact that our model (e.g., one layer, 128 input dimension, and trained for only one epoch) does not suffer the overfitting problem, and thus using dropout cannot further improve the performance of our embeddings on downstream tasks. However, the results in our experiments do not mean that dropout is useless, instead, it indicates that our model structure may not suffer from the overfitting issue. Moreover, previous work [24**? ?**] and experiments[17,18] also show that using dropout may not always improve the performance of neural networks, which further confirms our findings. For example, **?**] and **?**] observe that adding dropout may reduce the performance of the model. Besides, in the original work of dropout [**?**], the authors also explored the effect of changing data set size when dropout is used, and the results show that when the size of data sets is very small (e.g., 100, 500 samples) or very large (e.g., 50K samples), dropout may not give any improvements. These results suggest that developers and researchers should be careful when applying the dropout to the neural networks. Meanwhile, previous work [24**?**] provides suggestions on how and when to use dropout to avoid overfitting. For example, it is expected that dropping the neurons in the model would reduce the effective capacity of a model, thus **?**] suggest that increase the size of the model when using dropout and they suggest to set the number of units to n/p, where p is the dropout rate and n is the number of optimal units for a model without dropout. Besides, Goodfellow et al. [24] also suggests that when there is a large amount of training data, the benefit of using dropout may be outweighed by the computational cost of using dropout and larger models. Thus, considering our simple model architecture and the large size of the training dataset (i.e., over 60K samples), it is reasonable that using dropout does not significantly improve the quality of our generated embeddings. To better illustrate the ability of dropout in preventing model overfitting, future work can try to add more layers with more training epochs.

Traditional machine learning model (e.g., logistic regression used in the task of software defect prediction) is more sensitive to the changes of code embeddings. As shown in Table 6, almost any changes of the parameters of *GraphCodeVec* could lead to significant changes of the performance (either improvement or deterioration) of the software defect prediction task. This may be explained by the fact that the code embeddings are directly used as features for the traditional machine learning models, thus any changes of embeddings could be immediately propagated to the final output of these models. However, for deep learning-based models, the embeddings are only used to initialize the first embedding layer of which the value would be later adjusted to better fit the training data, as a result, the impact of utilizing different embeddings may be diminished or even erased during the model training and weights updating.

The thresholds used during the training context generation stage have a relatively more minor impact on the code embeddings generated by *GraphCodeVec* than that of the parameters involved during the embedding learning stage. For example, as we lower the thresholds of unique node (i.e., 50 and 80), there is only one significant performance change among all 23 tasks or datasets. This finding shows that *GraphCodeVec* is different from fastText, which is sensitive to the preprocessing of the corpus [**?**]. To complement the experiments, we have done another experiment for fastText where we only perform a lowercase preprocessing on the tokens (i.e., without the removal of non-identifiers and low-frequency tokens). The results are shown in Table 7. In our experiment, we find that among all the six tasks, the performances of four tasks (i.e., code comment generation,

[17]https://github.com/mvshashank08/article-dropout

[18]https://github.com/harrisonjansma/Research-Computer-Vision/blob/master/08-12-18%20Batch%20Norm%20vs%20Dropout/08-12-18%20Batch%20Norm%20vs%20Dropout.ipynb

Table 7. Evaluation results of fastText with different preprocessing strategies.

Dowsnstream Tasks	Code comment generation		Code authorship identification	Code clones detection			Source code classification	Logging statements prediction					
Datasets	GitHub		Google Code Jam	BCB	OJClone	Avg.	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop	Ang.
Metrics	BLEU	ROUGE	Accuracy	F1			Accuracy	BA					
Original	**19.9**	**36.0**	**76.6**	**93.4**	**84.6**	**89.0**	**76.7**	95.1	79.8	86.7	88.6	**74.4**	84.9
Lowercase	19.3	35.3	70.5	**93.4**	75.2	84.3	60.2	**96.2**	**80.2**	**86.8**	88.5	74.1	**85.2**

Dowsnstream Tasks	Software defect predicttion												
Datasets	Ant 1.5->1.6	Ant 1.6->1.7	Camel 1.2->1.4	Camel 1.4->1.6	jEdit 3.2->4.0	jEdit 4.0->4.1	Log4j 1.0->1.1	Lucene 2.0->2.2	Lucene 2.2->2.4	POI 1.5->2.5	POI 2.5->3.0	Xalan 2.4->2.5	**Avg.**
Metrics	F1												
Original	**36.0**	**44.2**	41.8	45.8	**53.6**	60.7	**63.1**	63.2	**65.3**	65.1	72.2	42.4	**54.5**
Lowercase	29.3	41.7	**44.5**	**46.0**	53.3	**61.5**	58.7	**65.0**	62.3	**69.9**	**72.5**	**47.1**	54.3

Note: Original and Lowercase are two different preprocessing strategies, where Original contains three steps: 1) remove non-identifiers, 2) filter out the rare tokens, and 3) lowercase all tokens; while Lowercase means that we only perform a lowercase preprocessing on the tokens (i.e., without removal of non-identifiers and low-frequency tokens).

code authorship identification, code clones detection, source code classification) have relatively large changes. The results confirm that fastText is sensitive to the preprocessing of training context. For example, on the task of source code classification, there is a 16.5% absolute decrease of fastText with different preprocessing strategies. Besides, by checking the significant performance changes caused by different *GraphCodeVec* settings, we find that changing parameters involved during the embedding learning stage has a higher possibility of causing significant performance changes of the different tasks and datasets. More specifically, modifying the threshold of unique node and the edge only causes one or two significant performance changes, even on the software defect prediction task, which is more sensitive to the change of code embeddings.

Impact of different data sampling. Different separations between training and test sets have a non-negligible effect on the performance of the models. Figure 11 shows the results of different embedding techniques on all datasets using 10-fold cross-validation. We can observe the apparent variances on almost all the tasks or datasets. For example, on the task of code authorship identification, all the results of the evaluated code embeddings have large variances, and the differences between the lowest and highest scores even exceed 10%. Although the variances on the tasks of code comment generation and source code classification seem to be small, both have obvious outliers, and the range of the Y-axis is larger. This finding further indicates the necessity of running multiple times with different separations between the training and test datasets to mitigate the effects on the data separation. Otherwise, the reported conclusions may be misleading, as the rankings of performance of different embedding techniques may differ.

Finally, we want to highlight that, on the one hand, fine-tuning parameters of *GraphCodeVec* for the different downstream tasks can usually result in improved performance. On the other hand, different parameters can have diverse impacts on the final performance, and if we do not know which settings to choose, starting from the suggestions from previous work is always not a bad choice.

7 THREATS TO VALIDITY

This section discusses the threats to the validity of our work.

External validity. One major threat of using GCN for training embeddings is the computational costs. In our work, the embeddings are trained in an NVIDIA GTX 1080Ti GPU, and it takes around 18 minutes to finish the training process, which is acceptable. In fact, the major computational costs are caused by the downstream tasks. For example, it takes around 10 hours to finish the evaluation on the comment generation task. Considering the large amount of time and computing resources needed for executing the downstream tasks, our quantitative evaluation is conducted on six SE tasks. However, we train our embeddings in a task-agnostic manner using an independent dataset from the datasets used in the downstream tasks. Although our study only focuses on six tasks, the scale of our study is comparable to prior research on embeddings evaluation [34]. Meanwhile, there exist other tasks that adopt the pre-trained embeddings, and we cannot confirm that our embeddings might be generalizable to all the tasks. For example, for the tasks that rely on both natural language texts and source code, such as traceability link recovery [**? ?**] and user review classification [**?**], we think that our embeddings may not perform very well on these two tasks as our code embeddings are only trained on source code and cannot capture the properties of natural language texts, which is confirmed by the task of code comment generation. However, we believe it would be a very promising research direction to jointly learn the code and text embeddings, and in that way the embeddings can be applied to such tasks which involve both texts and source code. Another threat is that some of our models used in downstream tasks may not give state-of-the-art results. For example, we use logistic regression in the task of software defect prediction, which is simple and a bit out of date, especially in the era of deep learning. However, our goal is to show the performance changes of different code embeddings. Although this model is simple, it is able to reflect the representation ability of different code embeddings. Nevertheless, we admit that our choices of the models in the studied downstream tasks pose a threat to the generalizability of our findings. Thus, using the downstream task of software defect prediction as an example, we experimented with other models, including Random Forest (RF), Naive Bayesian (NB), and Support Vector Machines (SVM). We observe that our general findings remain the same, and our proposed embedding approach achieves the best results for all the models except NB. We speculate that it may be because NB is not best suited for the task as it holds a strong assumption on the independence of the features which are difficult to satisfy in the resulting embeddings. In fact, the performance of NB is among the worst of all the considered models. On the other hand, we encourage future work to validate our findings on more downstream tasks and models. Moreover, there is a lack of qualitative tasks for quality evaluation, and all the downstream tasks are external tasks, which means we cannot do the evaluation directly. To minimize the threat and explore the internal characteristic of embeddings, we also provide a qualitative evaluation. While the qualitative evaluation may include subjective bias in terms of selection of example tokens and interpretation of their projection in the semantic space, that may be introduced by the different backgrounds of researchers. However, we have already provided the trained embeddings, and readers can explore the properties among the tokens of their own interests. Future studies can apply *GraphCodeVec* to other tasks, such as method name prediction, and develop some qualitative evaluation datasets, such as token similarity or token analogy test sets. For the comparison of the results, we report the final score of each evaluation metric. However, small variations (e.g., when one instance is classified in a different direction) may change the results. Our goal is to understand the performance changes between different code embeddings among different tasks. Although a small number of misclassifications may cause significant changes in the final scores, especially for small datasets, even in such cases, the improvement or decrease of the performance can still reflect the effect of the different embeddings. Besides, we do a 10-fold cross-validation to reduce the impact of such cases.

Internal validity. As described in Section 3.1, we attempt to represent the source code into graphs, where the nodes are tokens in the source code, and edges are AST paths. There could exist other strategies for representing the source code as a graph. Besides, we rely on the surface forms of the tokens to build the connected graph within a method, as a result, changing the name of a variable would lead to the change of the constructed graph. In particular, using meaningless identifiers (e.g., "v") may negatively impact the quality of the resulting embeddings and their effectiveness in the downstream tasks. However, we have applied our approach on a variety of real-world software projects. The results demonstrate the effectiveness of our approach when applied to ordinary code written by different developers. Meanwhile, using the same identifiers in different surrounding code contexts would also impact the performance of our approach. For example, the keyword "public", can be used as modifiers for different levels of source code (i.e., class, attribute, and method), and ideally, they should have different representations to better capture the properties. However, in our approach, the token 'public' has been assigned only one unique vector representation, which is non-optimal. Another threat to validity is that there is a possibility that the temporal dependencies among code tokens (i.e., the sequence of the source code tokens) may not be captured by the ASTs. However, prior work [6, 11, 29, 83, 95**?**] finds that the structural information performs better for some SE tasks. On the other hand, our way of constructing the training context still can capture such information, if the distance between the sequence of code tokens is within the threshold (i.e., a pre-defined value of the maximum number of AST nodes connecting two leaf nodes, c.f., Table 1). To better illustrate it, given the following code sequence, "public static void main", the temporal dependencies would be "public -> static-> void-> main", if we convert the code sequence to an AST, only the structure information of the code sequence is preserved and the sequence information is lost. While, if the distances between these tokens in the source code are within a threshold, by using our method to traverse the AST, we are able to construct the following triples, "public -> static", "static -> void", and "void -> main" (different colors represent different AST paths). And thus we can construct a graph which captures the temporal dependencies, "public -> static -> void -> main". Besides, in RQ2, we also treat the source code as a sequence of plain text for embeddings generation, which also confirms the findings that, overall, the structural information extracted from the ASTs can benefit code embeddings generation for the downstream SE tasks. Another threat is that in RQ2, we examine how the training context (with or without AST) impacts the resulting embeddings and thus on the performance of the downstream tasks. Although we only vary the input of the embedding training from the process point of view, the change in the resulting embeddings can impact the training process of the subsequent downstream tasks. More specifically, the changes of code embeddings would lead to different weight values between layers and neurons depending on the embedding during the training process. Thus, the training context is not the sole varying factor of the analysis and it confounds with other factors such as the training of the downstream tasks. Future work may further investigate the impact of the individual steps (e.g., embedding training) while isolating other steps (e.g., downstream stream task training). In Section 4.1, we remove the rare tokens during the preprocessing stage, which may also cause the removal of important tokens, leading to an overfitting model. While, on the contrary, rare tokens mean that there is not enough data for training. As we described in Section 3, if the tokens appear only once or twice, the vector (i.e., embeddings) of that token can only be updated limited times (depending on the epoch) which would result in a poor embedding of the code embeddings [10, 73, 89], thus we follow common practice [10, 73, 89] and remove these rare tokens. In RQ1, we provide a qualitative analysis of different embeddings, while the manual inspection may include subjective bias introduced by the individual participants. Future work can consider different graph representations and perform manual analysis to verify our findings.

Construct validity. As described in Section 4, we select six different tasks and corresponding models to evaluate the generalizability of the code embeddings. Thus, one of the threats is the quality of the models used in the downstream tasks. In our work, most of the models have a comparable or better performance compared to the work in the literature [34, 87, 95**?**]. Although, in our experiment, the model used in the task of code comment generation performs not as well as the original work [29, 34] (i.e., with a 5.7% performance degradation). This may be caused by the different parameters used for the inference stage and the data separation. Previous work [29, 34] only mentions the parameters for the model training but don't provide the parameters for inference, and unlike what we do in RQ1, they only randomly split the data into training, validation and test sets without a 10-fold cross validation which also has a non-negligible impact on the results. However, we can still observe the performance changes of the model caused by different code embeddings. Another threat is that the training data used for our embeddings is the *Java-small* dataset. There may exist other datasets that can be used for embedding training. And in order to make a fair comparison with baselines, we only extract the training context based on the methods which may lead to the inadequate use of the class or project level information from the source code. However, as ASTs can represent the source code with different levels (e.g., method level, statement level, class level, etc.), our method can also be applied to other types of training data. Besides, the edges (i.e., AST paths) in the graph representations are extracted based on the JavaParser tool. JavaParser is a mature tool and has been widely used in various software engineering research. Nevertheless, the quality of the data generated by JavaParser may impact the results of our study. *GraphCodeVec* requires several hyper-parameters for the training process, such as the dimensions, the number of GCN layers, and the number of training epochs, which may impact the resulting code embeddings. To minimize the bias caused by the hyper-parameter configurations, we follow the practices from prior studies [6, 34, 95] to configure the hyper-parameters. Performing further fine-tuning on these hyper-parameters may further improve the results of *GraphCodeVec*. In our experiments, we randomly initialize the OOV tokens with real numbers, which may affect the performance of downstream tasks. However, to minimize such influence, we conduct a 10-fold cross-validation for all experiments.

8 CONCLUSIONS

In this paper, we introduce a graph convolutional network based approach, *GraphCodeVec*, which represents source code as graphs and learns code token embeddings from the context information provided by the graphs. *GraphCodeVec* trains code token embeddings in an unsupervised way, aiming to improve the generalizability of the learned embeddings. We evaluate *GraphCodeVec* on an extended benchmark containing six downstream SE tasks. The experiment results show that *GraphCodeVec* performs comparable or better than all existing code embedding techniques on all SE tasks. Our approach and our pre-trained embeddings can be leveraged by software engineering researchers and practitioners in their downstream tasks that rely on or can be improved by code embeddings. Our work also sheds light on future work that explores different approaches of constructing graph representations of source code and utilizing graph-based deep learning methods to leverage the graph representations.

REFERENCES

[1] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. 2018. Large-Scale and Language-Oblivious Code Authorship Identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 101–114. https://doi.org/10.1145/3243734.3243738

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 38–49. https://doi.org/10.1145/2786805.2786849

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJOFETxR-

[4] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 404–419. https://doi.org/10.1145/3192366.3192412

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *PACMPL* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[7] Liliane Barbour, Foutse Khomh, and Ying Zou. 2011. Late propagation in software clones. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 273–282. https://doi.org/10.1109/ICSM.2011.6080794

[97]]Bengio2012 Yoshua Bengio. [n. d.]. Practical Recommendations for Gradient-Based Training of Deep Architectures. In *Neural Networks: Tricks of the Trade - Second Edition*, Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller (Eds.). Lecture Notes in Computer Science, Vol. 7700. Springer, 437–478. https://doi.org/10.1007/978-3-642-35289-8_26

[9] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings,*

Vol. 48), Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

[10] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146. https://transacl.org/ojs/index.php/tacl/article/view/999

[11] Lutz Büch and Artur Andrzejak. 2019. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 95–104. https://doi.org/10.1109/SANER.2019.8668039

[97]]Cai2019 Shaofeng Cai, Yao Shu, Gang Chen, Beng Chin Ooi, Wei Wang, and Meihui Zhang. [n. d.]. Effective and Efficient Dropout for Deep Convolutional Neural Networks. ([n. d.]). arXiv:1904.03392 [cs.LG]

[13] Yixin Cao, Lifu Huang, Heng Ji, Xu Chen, and Juanzi Li. 2017. Bridge Text and Knowledge by Learning Multi-Prototype Entity Mention Embedding. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1623–1633. https://doi.org/10.18653/v1/P17-1149

[97]]Chen2020 Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. [n. d.]. Measuring and Relieving the Over-Smoothing Problem for Graph Neural Networks from the Topological View. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020* (2020). AAAI Press, 3438–3445. https://aaai.org/ojs/index.php/AAAI/article/view/5747

[15] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. 2016. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering* 21, 5 (2016), 1843–1919. https://doi.org/10.1007/s10664-015-9402-8

[16] Zimin Chen and Martin Monperrus. 2018. The Remarkable Role of Similarity in Redundancy-based Program Repair. *CoRR* abs/1811.05703 (2018). arXiv:1811.05703 http://arxiv.org/abs/1811.05703

[17] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 3837–3845. http://papers.nips.cc/paper/6081-convolutional-neural-networks-on-graphs-with-fast-localized-spectral-filtering

[18] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet?. In *Proceedings of the 42st International Conference on Software Engineering, ICSE 2020, Seoul, South Korea, July 6-11, 2020* (2020-07-11).

[19] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, Anthony Cleve, Filippo Ricca, and Maura Cerioli (Eds.). IEEE Computer Society, 25–34. https://doi.org/10.1109/CSMR.2013.13

[20] Vasiliki Efstathiou and Diomidis Spinellis. 2019. Semantic source code models using identifier embeddings. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 29–33. https://doi.org/10.1109/MSR.2019.00015

[21] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. ACM, New York, NY, USA, 24–33.

[97]]Garbin2020 Christian Garbin, Xingquan Zhu, and Oge Marques. [n. d.]. Dropout vs. batch normalization: an empirical study of their impact to deep learning. 79, 19-20 ([n. d.]), 12777–12815. https://doi.org/10.1007/s11042-019-08453-9

[97]]Goldberg2017 Yoav Goldberg. [n. d.]. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00762ED1V01Y201703HLT037

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[25] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Marc W. McConley, Jeffrey M. Opper, Sang Peter Chin, and Tomo Lazovich. 2018. Automated software vulnerability detection with machine learning. *CoRR* abs/1803.04497 (2018). arXiv:1803.04497 http://arxiv.org/abs/1803.04497

[26] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[27] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[97]]Hu2020 Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. [n. d.]. Deep code comment generation with hybrid lexical and syntactical information. 25, 3 ([n. d.]), 2179–2217. https://doi.org/10.1007/s10664-019-09730-9

[29] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 200–210. https://doi.org/10.1145/3196321.3196334

[30] Aylin Caliskan Islam, Richard E. Harang, Andrew Liu, Arvind Narayanan, Clare R. Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing Programmers via Code Stylometry. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 255–270. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/caliskan-islam

[97]]Johnson2015 Rie Johnson and Tong Zhang. [n. d.]. Effective Use of Word Order for Text Categorization with Convolutional Neural Networks. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015* (2015), Rada Mihalcea, Joyce Yue Chai, and Anoop Sarkar (Eds.). The Association for Computational Linguistics, 103–112. https://doi.org/10.3115/v1/n15-1011

[97]]Kallis2021 Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. [n. d.]. Predicting issue types on GitHub. 205 ([n. d.]), 102598. https://doi.org/10.1016/j.scico.2020.102598

[33] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Eng.* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[34] Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. 2019. Assessing the Generalizability of Code2vec Token Embeddings. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1–12. https://doi.org/10.1109/ASE.2019.00011

[35] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. 2006. MUDABlue: An automatic categorization system for Open Source repositories. *J. Syst. Softw.* 79, 7 (2006), 939–953. https://doi.org/10.1016/j.jss.2005.06.044

[36] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[37] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1746–1751. https://doi.org/10.3115/v1/d14-1181

[38] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[39] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. 2017. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, System Demonstrations*, Mohit Bansal and Heng Ji (Eds.). Association for Computational Linguistics, 67–72. https://doi.org/10.18653/v1/P17-4012

[40] Alexandros Komninos and Suresh Manandhar. 2016. Dependency based embeddings for sentence classification tasks. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*. 1490–1500.

[41] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014 (JMLR Workshop and Conference Proceedings, Vol. 32)*. JMLR.org, 1188–1196. http://proceedings.mlr.press/v32/le14.html

[42] Omer Levy and Yoav Goldberg. 2014. Dependency-Based Word Embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, Volume 2: Short Papers*. The Association for Computer Linguistics, 302–308. https://doi.org/10.3115/v1/p14-2050

[43] Chen Li, Jianxin Li, Yangqiu Song, and Ziwei Lin. 2018. Training and Evaluating Improved Dependency-Based Word Embeddings. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 5836–5843. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16429

[44])]Li2019 Guohao Li, Matthias Müller, Ali K. Thabet, and Bernard Ghanem. [n. d.]. DeepGCNs: Can GCNs Go As Deep As CNNs?. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019* (2019). IEEE, 9266–9275. https://doi.org/10.1109/ICCV.2019.00936

[63])]Li2018a Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. [n. d.]. Studying software logging using topic models. 23, 5 ([n. d.]), 2655–2694. https://doi.org/10.1007/s10664-018-9595-8

[64])]Li2018b Qimai Li, Zhichao Han, and Xiao-Ming Wu. [n. d.]. Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018* (2018), Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 3538–3545. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16098

[47] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.05493

[48])]Li2021 Zhenhao Li, Heng Li, Tse-Hsun Peter Chen, and Weiyi Shang. [n. d.]. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021* (2021). IEEE, 1461–1472. https://doi.org/10.1109/ICSE43902.2021.00131

[97]]Lin2004 Chin-Yew Lin. [n. d.]. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out* (Barcelona, Spain, 2004-07). Association for Computational Linguistics, 74–81. https://aclanthology.org/W04-1013

[97]]Liu2020 Zhiyuan Liu, Yankai Lin, and Maosong Sun. [n. d.]. *Representation Learning for Natural Language Processing.* Springer. https://doi.org/10.1007/978-981-15-5573-2

[51] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, Nov (2008), 2579–2605.

[52] Diego Marcheggiani and Ivan Titov. 2017. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, Martha Palmer, Rebecca Hwa, and Sebastian Riedel (Eds.). Association for Computational Linguistics, 1506–1515. https://doi.org/10.18653/v1/d17-1159

[53] Dominic Masters and Carlo Luschi. 2018. Revisiting Small Batch Training for Deep Neural Networks. abs/1804.07612 (2018). arXiv:1804.07612 http://arxiv.org/abs/1804.07612

[54] Jean Mayrand, Claude Leblanc, and Ettore Merlo. 1996. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*. IEEE Computer Society, 244. https://doi.org/10.1109/ICSM.1996.565012

[55] Paul W. McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, Chanchal K. Roy, Andrew Begel, and Leon Moonen (Eds.). ACM, 279–290. https://doi.org/10.1145/2597008.2597149

[56] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781

[57] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., USA, 3111–3119. http://dl.acm.org/citation.cfm?id=2999792.2999959

[97]]Mnih2008 Andriy Mnih and Geoffrey E. Hinton. [n. d.]. A Scalable Hierarchical Distributed Language Model. In *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008* (2008), Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou (Eds.). Curran Associates, Inc., 1081–1088. https://proceedings.neurips.cc/paper/2008/hash/1e056d2b0ebd5c878c550da6ac5d3724-Abstract.html

[59] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 23–32. https://doi.org/10.1109/ICPC.2013.6613830

[97]]Morin2005 Frederic Morin and Yoshua Bengio. [n. d.]. Hierarchical Probabilistic Neural Network Language Model. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, AISTATS 2005, Bridgetown, Barbados, January 6-8, 2005* (2005), Robert G. Cowell and Zoubin Ghahramani (Eds.). Society for Artificial Intelligence and Statistics. http://www.gatsby.ucl.ac.uk/aistats/fullpapers/208.pdf

[61] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

[62] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

[63])]Oliveto2010 Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. [n. d.]. On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery. In *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010* (2010). IEEE Computer Society, 68–71. https://doi.org/10.1109/ICPC.2010.20

[64])]Oliveto2020 Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. [n. d.]. On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery: A Ten-Year Retrospective. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020* (2020). ACM, 1. https://doi.org/10.1145/3387904.3394491

[65] John F. Pane, Chotirat (Ann) Ratanamahatana, and Brad A. Myers. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum. Comput. Stud.* 54, 2 (2001), 237–264. https://doi.org/10.1006/ijhc.2000.0410

[97]]Panichella2015 Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. [n. d.]. How can i improve my app? Classifying user reviews for software maintenance and evolution. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015* (2015), Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 281–290. https://doi.org/10.1109/ICSM.2015.7332474

[67] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318. https://www.aclweb.org/anthology/P02-1040/

[68] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[69] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1532–1543. https://www.aclweb.org/anthology/D14-1162/

[70] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[71] Likun Qiu, Yue Zhang, and Yanan Lu. 2015. Syntactic dependencies and distributed word representations for analogy detection and mining. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 2441–2450.

[72] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 111–124. https://doi.org/10.1145/2676726.2677009

[73] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.

[97]]Rong2014 Xin Rong. [n. d.]. word2vec Parameter Learning Explained. ([n. d.]). arXiv:1411.2738 [cs.CL]

[97]]Rong2020 Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. [n. d.]. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (2020). OpenReview.net. https://openreview.net/forum?id=Hkx1qkrKPr

[76] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 1157–1168. https://doi.org/10.1145/2884781.2884877

[97]]Schnabel2015 Tobias Schnabel, Igor Labutov, David M. Mimno, and Thorsten Joachims. [n. d.]. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015* (2015), Lluís Màrquez, Chris Callison-Burch, Jian Su, Daniele Pighin, and Yuval Marton (Eds.). The Association for Computational Linguistics, 298–307. https://doi.org/10.18653/v1/d15-1036

[78] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 43–52. https://doi.org/10.1145/1858996.1859006

[97]]Srivastava2014 Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. [n. d.]. Dropout: a simple way to prevent neural networks from overfitting. 15, 1 ([n. d.]), 1929–1958. http://dl.acm.org/citation.cfm?id=2670313

[80] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 476–480. https://doi.org/10.1109/ICSME.2014.77

[81] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. 2019. Import2vec learning embeddings for software libraries. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 18–28. https://doi.org/10.1109/MSR.2019.00014

[82] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. 2010. An empirical study on the maintenance of source code clones. *Empirical Software Engineering* 15, 1 (2010), 1–34. https://doi.org/10.1007/s10664-009-9108-x

[83] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 542–553. https://doi.org/10.1145/3196398.3196431

[84] Shikhar Vashishth, Manik Bhandari, Prateek Yadav, Piyush Rai, Chiranjib Bhattacharyya, and Partha P. Talukdar. 2019. Incorporating Syntactic and Semantic Information in Word Embeddings using Graph Convolutional Networks. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, Anna Korhonen,

David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 3308–3318. https://doi.org/10.18653/v1/p19-1320

[85] Mario Linares Vásquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. 2014. On using machine learning to automatically classify software applications into domain categories. *Empirical Software Engineering* 19, 3 (2014), 582–618. https://doi.org/10.1007/s10664-012-9230-z

[86] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embeddings for Program Repair. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJuWrGW0Z

[87] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 297–308. https://doi.org/10.1145/2884781.2884804

[88] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 3034–3040. https://doi.org/10.24963/ijcai.2017/423

[89] Laura Wendlandt, Jonathan K. Kummerfeld, and Rada Mihalcea. 2018. Factors Influencing the Surprising Instability of Word Embeddings. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, Marilyn A. Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, 2092–2102. https://doi.org/10.18653/v1/n18-1190

[90] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 479–490. https://doi.org/10.1109/SANER.2019.8668043

[91] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 87–98. https://doi.org/10.1145/2970276.2970326

[92] Yichun Yin, Furu Wei, Li Dong, Kaimeng Xu, Ming Zhang, and Ming Zhou. 2016. Unsupervised Word and Dependency Path Embeddings for Aspect Term Extraction. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, Subbarao Kambhampati (Ed.). IJCAI/AAAI Press, 2979–2985. http://www.ijcai.org/Abstract/16/423

[97]]Yu2020 Xiaohan Yu, Quzhe Huang, Zheng Wang, Yansong Feng, and Dongyan Zhao. [n. d.]. Towards Context-Aware Code Comment Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020* (2020) *(Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 3938–3947. https://doi.org/10.18653/v1/2020.findings-emnlp.350

[94] Lei Zeng, Yang Xiao, and Hui Chen. 2015. Linux auditing: Overhead and adaptation. In *2015 IEEE International Conference on Communications, ICC 2015, London, United Kingdom, June 8-12, 2015*. IEEE, 7168–7173. https://doi.org/10.1109/ICC.2015.7249470

[95] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 783–794. https://doi.org/10.1109/ICSE.2019.00086

[97]]Zhang2020 Xiaoqing Zhang, Yu Zhou, Tingting Han, and Taolue Chen. [n. d.]. Training Deep Code Comment Generation Models via Data Augmentation. In *Internetware'20: 12th Asia-Pacific Symposium on Internetware, Singapore, November 1-3, 2020* (2020). ACM, 185–188. https://doi.org/10.1145/3457913.3457937

[97]]Zhou2021 Kuangqi Zhou, Yanfei Dong, Kaixin Wang, Wee Sun Lee, Bryan Hooi, Huan Xu, and Jiashi Feng. [n. d.]. Understanding and Resolving Performance Degradation in Deep Graph Convolutional Networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (Virtual Event, Queensland, Australia, 2021-10-26) *(CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 2728–2737. https://doi.org/10.1145/3459637.3482488

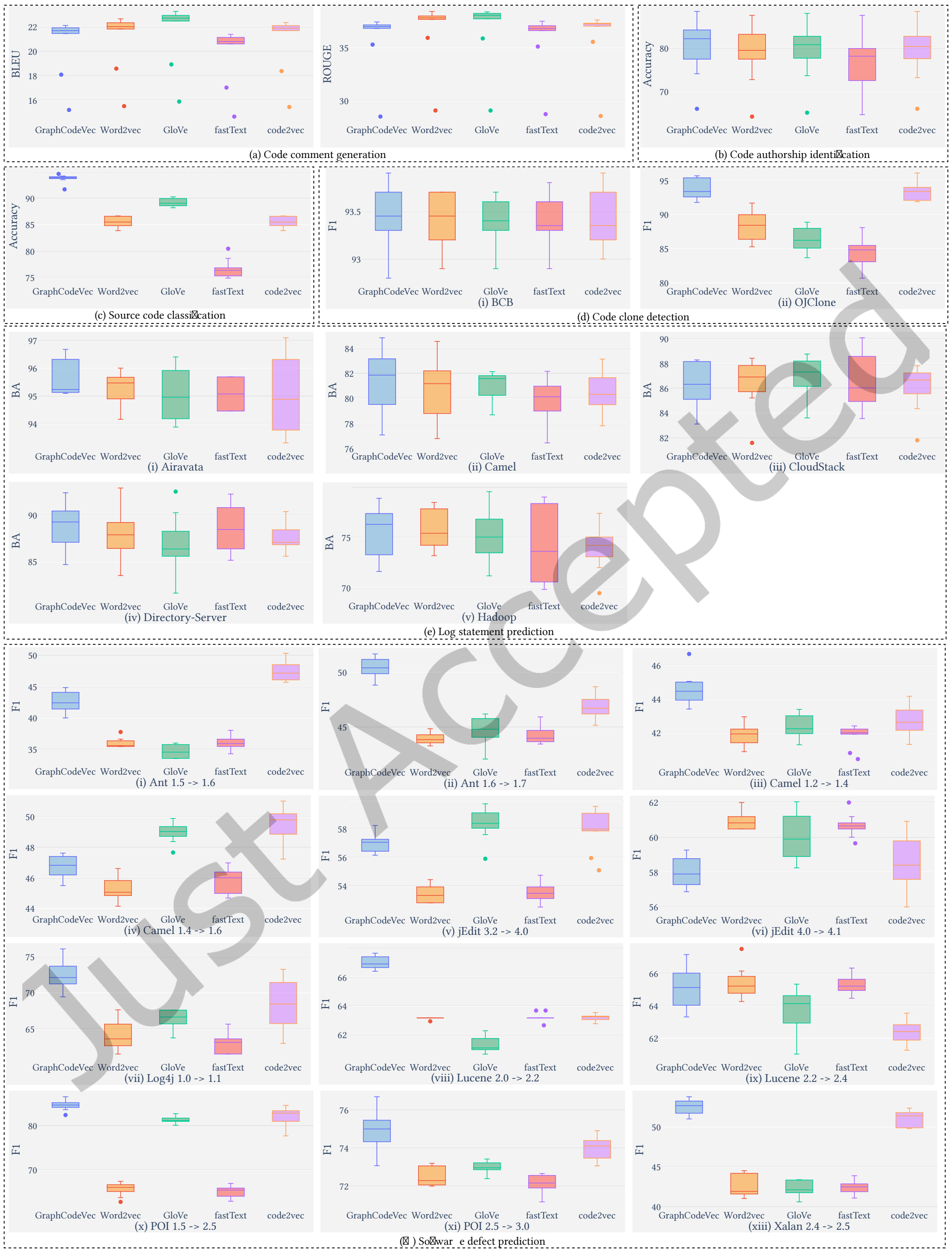


Fig. 11. The results of different embedding techniques on all datasets using 10-fold cross-validation.