Empirical Software Engineering (2022) 27:63
https://doi.org/10.1007/s10664-022-10118-5

Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks

Zishuo Ding[1] · Heng Li[2] · Weiyi Shang[1] · Tse-Hsun (Peter) Chen[1]

Accepted: 10 January 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Word representation plays a key role in natural language processing (NLP). Various representation methods have been developed, among which pre-trained word embeddings (i.e., dense vectors that represent words) have shown to be highly effective in many neural network-based NLP applications, such as named entity recognition (NER) and part-of-speech (POS) tagging. However, the use of pre-trained code embeddings for software engineering (SE) tasks has not been extensively explored. A recent study by Kang et al. (2019) finds that code embeddings may not be readily leveraged for the downstream tasks that the embeddings are not trained for. However, Kang et al. (2019) only evaluate two code embedding approaches on three downstream tasks and both approaches may have not taken full advantage of the context information in the code when training code embeddings. Considering the limitations of the evaluated embedding techniques and downstream tasks in Kang et al. (2019), we would like to revisit the prior study by examining whether the lack of generalizability of pre-trained code embeddings can be addressed by considering both the textual and structural information of the code and using unsupervised learning. Therefore, in this paper, we propose a framework, StrucTexVec, which uses a two-step unsupervised training strategy to incorporate the textual and structural information of the code. Then, we extend prior work (Kang et al. 2019) by evaluating seven code embedding techniques and comparing them with models that do not utilize pre-trained embeddings in six downstream tasks. Our results first confirm the findings from prior work, i.e., pre-trained embeddings may not always have a significant effect on the performance of downstream SE tasks. Nevertheless, we also observe that (1) different embedding techniques can result in diverse performance for some SE tasks; (2) using well pre-trained embeddings usually improve the performance of SE tasks (e.g., all six downstream tasks in our study); and (3) the structural context has a non-negligible impact on improving the quality of code embeddings (e.g., embedding approaches that leverage the structural context achieve the best performance in five out of six downstream tasks among all the evaluated non-contextual embeddings), and thus, future work can consider incorporating such information into the large pre-trained

Communicated by: Dan Hao

✉ Zishuo Ding
zi_ding@encs.concordia.ca

Extended author information available on the last page of the article.

Published online: 17 March 2022

models. Our findings imply the importance and effectiveness of combining both textual and structural context in creating code embeddings. Moreover, one should be very careful with the selection of code embedding techniques for different downstream tasks, as it may be difficult to prescribe a single best-performing solution for all SE tasks.

Keywords Machine learning · Source code representation · Code embeddings · Neural network

1 Introduction

In recent times, distributed representations of words, also called word embeddings, have shown to be highly effective in many neural network models-based natural language processing (NLP) tasks, such as named entity recognition (NER), part-of-speech (POS) tagging (Li et al. 2018a), and sentence classification (Komninos and Manandhar 2016). In NLP, various word embedding techniques have been developed to encode words with different meanings into a low-dimensional vector space. Meanwhile, distributed code/program representations (i.e., code embeddings) have also proven to be useful in assisting in software engineering tasks, such as automatic program repair (Chen and Monperrus 2018; Wang et al. 2018; White et al. 2019), software vulnerability prediction (Harer et al. 2018; Pradel and Sen 2018), method name prediction (Alon et al. 2019; Allamanis et al. 2015), and code clone detection (Büch and Andrzejak 2019).

Researchers have worked on a number of approaches for representing source code into vectors in SE tasks. Among these, some directly apply word embedding techniques to source code to produce representations for code tokens, for example, Theeten et al. (2019) use Word2vec (Mikolov et al. 2013a, b) to generate embeddings for software libraries, while other researchers have proposed task-specific approaches for SE tasks. For example, Alon et al. (2019) propose a path-attention network to learn source code embeddings for the task of method name prediction. These generated source code embeddings that are trained on large source code datasets can later be used for other SE tasks, and thus are also called pre-trained code embeddings.

Although different code embeddings learning techniques have been proposed, the use of the pre-trained code embeddings for different SE downstream tasks has not been extensively explored. A recent study by Kang et al. (2019) evaluates two code embedding approaches (i.e., GloVe (Pennington et al. 2014) and code2vec (Alon et al. 2019)) on three downstream SE tasks, namely code comment generation, code authorship identification, and code clone detection. They find that the pre-trained code embeddings may not be readily leveraged for the downstream tasks that the embeddings are not trained for.

Intuitively, pre-trained code embeddings can bring more knowledge about the semantic and syntactic meanings of code tokens as they are trained on large external datasets. Thus, models using pre-trained code embeddings are expected to perform better than models without that information. On the other hand, both studied embedding techniques only utilize partial information during the embedding training and do not take full advantage of the information from the source code. In particular, GloVe treats the source code as plain text and only considers the unstructured local textual information, and code2vec 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. Hence, we would like to find out whether the poor performance of the pre-trained code embeddings can be addressed by

combining both the textual and structural information in the source code, as well as how the different types of information affect the performance of downstream tasks. In addition, we would like to understand how the advancement of embedding techniques in recent years impacts the findings from the prior research.

Therefore, in this paper, we revisit and extend the assessment of using pre-trained code embeddings across a wider variety of SE tasks, aiming to provide more insights to guide further research that leverages code embeddings. Note that the main goal of the paper is not meant to propose entirely new methods. Instead, the goal and main contribution of the paper is to revisit the findings from prior research in order to understand whether they still hold with the fast progress of related research in recent years and with the consideration of extra information (e.g., method invocation).

Our work extends prior work (Kang et al. 2019) from two aspects. First, in addition to GloVe and code2vec, we evaluated more code embedding techniques. In particular, we propose a two-stage embedding learning approach called StrucTexVec, designed specifically for source code data with the special consideration of learning from both the textual and structural information. In particular, in the first stage, to capture the structural information, we pre-train the embeddings by customizing the dependency-based word embedding approach (Li et al. 2018a). Unlike code2vec, which only considers the AST information within every single method, StrucTexVec also utilizes method call and variable reference information. In the second stage, to incorporate the textual context information, we re-train the embeddings on the tokenized source code. The code embedding learning is framed as an unsupervised learning procedure, as we aim to generalize the learned embeddings to different downstream tasks. Thus, StrucTexVec does not require any manual labeling of the training data. In addition, we also consider Word2vec (Mikolov et al. 2013a, b), fastText (Bojanowski et al. 2017) and the recently released contextual embedding techniques, such as CodeBERT (Feng et al. 2020) and CuBERT (Kanade et al. 2020) for training code embeddings. In total, we evaluate seven code embedding techniques with different configurations.

Second, we also extend prior work (Kang et al. 2019) by considering more downstream tasks. To assess the effectiveness of using pre-trained code embeddings in SE tasks and understand the impact of structural and textual information on creating generalizable code embeddings, we conduct a comprehensive quantitative evaluation in six downstream SE tasks: code comment generation, code authorship identification, code clone detection, source code classification, logging statement prediction and software defect prediction. We apply and compare the seven learned code embeddings in these benchmark tasks. The source code and benchmark tasks are publicly available. The contributions of this paper are as follows:[1]

- We revisit and extend previous work by evaluating more embedding techniques across a wider variety of downstream SE tasks.
- Our findings confirm the challenge of using pre-trained code embeddings in downstream SE tasks (Kang et al. 2019), as using pre-trained code embeddings may not always achieve boosting in performance.
- We observe that using pre-trained embeddings performs better than not using them in all the downstream tasks. However, different embedding techniques can result in diverse performance, and there does not exist an embedding technique that outperforms others in all nor even the majority of the tasks.

[1] We share the trained embeddings together with the downstream tasks at Google Drive.

– We also observe that both the structural and textual information have a non-negligible impact on the quality of pre-trained code embeddings and find that the structural information has a larger impact on the quality of the code embeddings than the textual information. Researchers may consider incorporating the structural information into CodeBERT or CuBERT for further improvement.

Our findings suggest that future research and practice should take careful consideration on the selection of code embedding techniques before training their models for different tasks, as it may be impossible to prescribe a single best-performing solution for all SE tasks.

Paper Organization We present the background in Section 2 and describe our proposed approach, StrucTexVec in Section 3. Section 4 presents the experimental setup. Section 5 presents our experimental results and answers to research questions. Section 6 discusses our lessons learned. Section 7 presents the prior research that is related to this paper. Section 8 presents threats to the validity of our study. Finally, Section 9 concludes this paper.

2 Background

In this section, we discuss the background related to this revisiting study. We first introduce the training context and then talk about the code embedding techniques that are evaluated in this work.

2.1 Training Context

In this part, we introduce two types of training contexts for code embeddings: (1) textual context, which is the plain text of the source code, and (2) structural context, which refers to the abstract syntax trees (ASTs) of the source code.

Textual Context Like natural languages, programming languages are repetitive and predictable (Hindle et al. 2012), and thus researchers (Theeten et al. 2019; Efstathiou and Spinellis 2019; Bojanowski et al. 2017) consider source code as plain text and directly apply existing word embedding techniques to source code. More specifically, to make source code suitable for embeddings training, the source code is usually tokenized into a sequence of tokens (Theeten et al. 2019; Efstathiou and Spinellis 2019; Bojanowski et al. 2017). For example, the following source code[2] in Listing 1, after tokenization, is converted to a sequence of code tokens, shown in Listing 2. The generated token sequence is used as the training corpus and finally fed into embeddings techniques.

Structural Context Another representation of source code is the abstract syntax tree (AST). AST represents source code with a tree structure. Figure 1 is the AST representation of the code snippet in Listing 1, where the leaf nodes of the tree are the tokens from the source code, while the non-leaf nodes are a set of AST node types that provide the syntax structure of the code. Due to its ability of capturing not only the lexical information but also the syntactic structure of source code, AST has proven to be useful in a wide range of software

[2]https://commons.apache.org/proper/commons-io/javadocs/api-2.5/src-html/org/apache/commons/io/filefilter/AndFileFilter.html

```
public boolean accept(final File file) {
      if (this.fileFilters.isEmpty()) {
         return false;
      }
      for (final IOFileFilter fileFilter : fileFilters) {
         if (!fileFilter.accept(file)) {
            return false;
         }
      }
      return true;
   }
```

Listing 1 Code snippet from Apache Commons project

engineering tasks, including code embeddings (Alon et al. 2019; Zhang et al. 2019; Tufano et al. 2018). For example, Alon et al. (2019) takes the AST nodes as input and train a path-attention network for generating code embeddings.

2.2 Embedding Learning Techniques

With the rapid development of deep learning in SE applications, various distributed code representation techniques have been proposed, which can be categorized into two broad categories: (1) non-contextual embeddings (e.g., Word2vec, GloVe), which learn unique fixed representations for tokens in the vocabulary without considering the meanings of tokens in different contexts, and (2) contextual embeddings (e.g., CodeBERT and CuBERT), which are generally obtained from the transformer-based models and the representations of tokens are adjusted based on different contexts. In this section, we first introduce the two categories, and then describe the existing embedding learning techniques (i.e., Word2vec, GloVe, fastText, code2vec, CodeBERT and CuBERT) that are evaluated in this work. Table 1 also summarizes these techniques in more detail.

2.2.1 Non-contextual Embeddings

Non-contextual embeddings map source code tokens into a low-dimensional semantic space, where each code token is assigned with a unique real-valued vector. Non-contextual embeddings act as a static look-up table $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ to map a token in the vocabulary, V to a d-dimensional vector. The embeddings are usually learned from a large corpus, and can be applied to downstream tasks to either initialize the weights of the embedding layer (i.e., the input layer) of deep learning models or be used as feature vectors for traditional machine learning models. In this section, we introduce several state-of-the-art distributed code representation approaches in detail.

Word2vec Word2vec has become popular in software engineering tasks (Zhang et al. 2019) due to its high efficiency. In particular, Word2vec (Mikolov et al. 2013a, b) has two model

```
public boolean accept ( final File file ) { ... file ) ) { return false
    ; } } return true ; }
```

Listing 2 Code tokens after tokenization

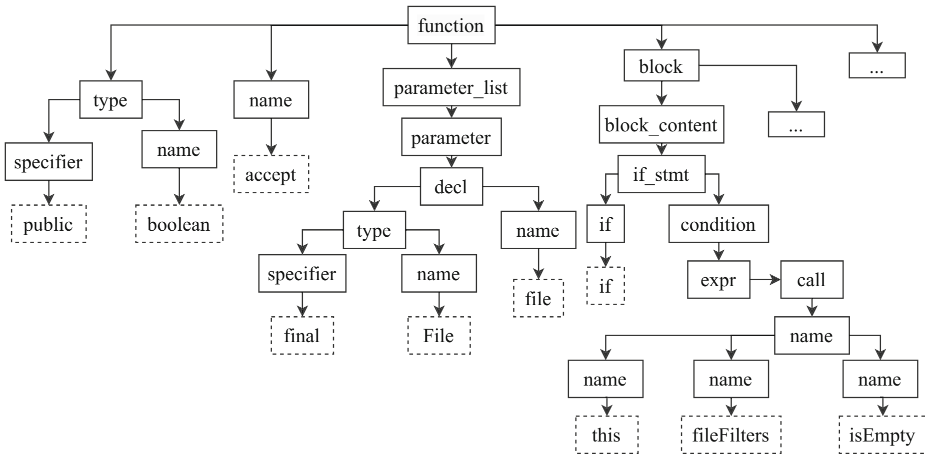

Fig. 1 The AST of the code snippet from Listing 1. The leaf nodes of the tree are the tokens from the source code, while the non-leaf nodes are a set of AST node types that provide the syntax structure of the code

architectures: Continuous Bag-of-Words (CBOW) and skip-gram, both consider the source code as plain text and take the textual context as input for training the embeddings.

An illustration of the CBOW and skip-gram models are shown in Fig. 2, where w_t is one target token from the vocabulary, and C_{w_t} are the context tokens of the target word w_t.

Continuous Bag-of-Words Model. The CBOW model tries to predict the target token by considering its context within the local window. Formally, given a sequence of tokens $\mathcal{D}$, and w_t is the tth token (i.e., target token) in the corpus, the objective of the model is to maximize the following objective function:

$$\mathcal{L} = \sum_{w_t \in \mathcal{D}} \log p \left(w_t | \mathcal{C}_{w_t} \right) \tag{1}$$

where $\mathcal{C}_{w_t}$ are the local context tokens of the target token w_t, and p is the conditional probability of generating the central target token w_t from given context tokens $\mathcal{C}_{w_t}$.

Skip-Gram Model. As Fig. 2b shows, the skip-gram model shares a similar architecture with the CBOW model. Rather than predicting the target token based on the local context, it tries to predict the context tokens based on the target token. Thus, the objective of this model becomes to maximize the function:

$$\mathcal{L} = \sum_{w_t \in \mathcal{D}} \sum_{w_c \in \mathcal{C}_{w_t}} \log p \left(w_c | w_t \right) \tag{2}$$

The conditional probability, $p\left(w_c|w_t\right)$ is defined using the following softmax function:

$$p\left(w_c|w_t\right) = \frac{\exp(\mathbf{V}_{w_t}{}^{\intercal}\mathbf{U}_{w_c})}{\sum_{w \in \mathcal{D}} \exp(\mathbf{V}_{w_t}{}^{\intercal}\mathbf{U}_{w})} \tag{3}$$

where $\mathbf{V}_w$ and $\mathbf{U}_w$ denote the "input" and "output" vectors of token w in the vocabulary $\mathcal{D}$, respectively.

Limitations. As discussed above, Word2vec is one of the shallow window-based methods, which adopts simple neural networks to learn the embeddings based on the tokens within a local context window, and thus, it may fail to take the advantage of some global

Table 1 Summary of different embedding techniques

Technique	Category	Task type	Corpus type	Corpus level	Description
Word2vec	Non-Contextual	Unsupervised	Code as plain text	Token	Word2vec considers source code as plain text and adopts simple neural networks to learn the embeddings based on the tokens within a local context window, and thus, it may fail to take the advantage of (1) some global information, (2) the character level information, and (3) the structural information (e.g., AST).
GloVe					GloVe also considers source code as plain text and learns the embeddings based on the global word-word co-occurrence statistics. However, it only considers the token level information and ignores the character level information as well as the structural information (e.g., AST).
fastText				Character	fastText extends the skip-gram model (cf. Section 2.2.1) and takes into account subword information. However, it still ignores the global and structural information.
code2vec		Supervised	AST	Token	code2vec is a supervised approach which means to guarantee good results, it requires human labeled data for training. Besides, code2vec is trained on the task of method name prediction and thus, the code embeddings produced by code2vec is task-specific and may not generalize well to other tasks downstream (Kang et al., 2019).

Table 1 (continued)

Technique	Category	Task type	Corpus type	Corpus level	Description
CodeBERT CuBERT	Contextual	Unsupervised	Code as plain text & code documentation	Character	Both CodeBERT and CuBERT are transformer-based models and learn the context-sensitive representations of tokens. However, there may still exist limitations for these two models. The first one is that they both treat the source code as plain text and do not consider the structural information explicitly. Another limitation is the high computation cost for training such huge models (both have millions of parameters), making it almost impossible for us to modify and re-train the models by ourselves.

information. For example, considering the preprocessed source code in Listing 2, assuming the window size is 10, and the target token is the method name “accept”, then the first returned value “false” can be captured as one of the context tokens by the window. However, the last returned value “true” is missed as its distance to the target token is beyond the

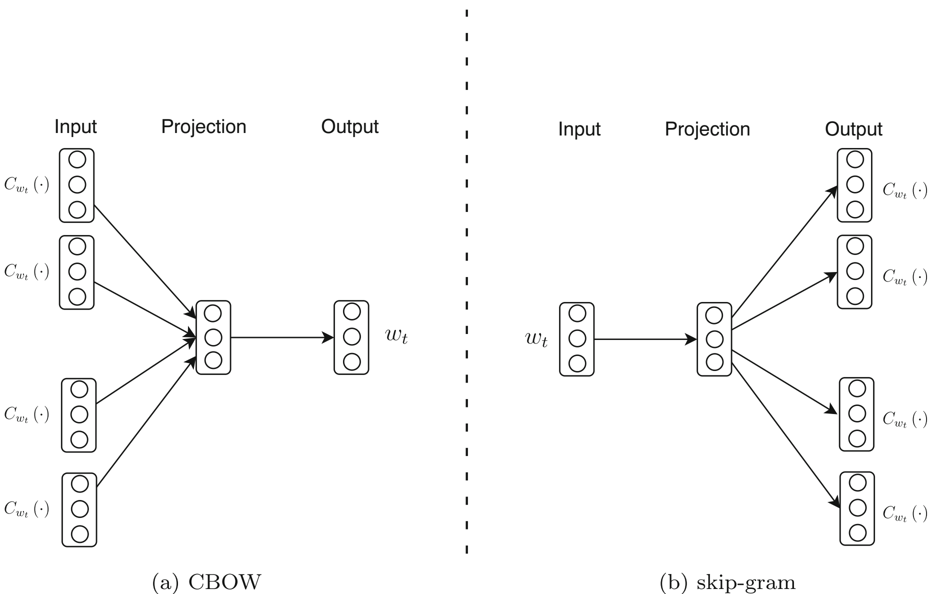

Fig. 2 An illustration of CBOW and skip-gram models architecture

window size, which is not in accord with the programming rules: both should be the context tokens of "accept", as they are the returned values, which should be directly connected to the method.

GloVe To better capture the global statistic information of the training corpus, Pennington et al. (2014) propose GloVe (Global Vectors for Word Representation). GloVe is also an unsupervised embedding learning algorithm and uses token-token co-occurrence statistics to obtain vector representations for source code tokens. Similar to Word2vec, in this model, the source code is treated as plain text, and the textual context is considered as the training context. The goal of GloVe is to minimize the following weighted least squares errors:

$$J = \sum_{i,j=1}^{|\mathcal{D}|} f\left(X_{i,j}\right)\left(w_i^T \tilde{w}_j - \log X_{i,j}\right)^2 \quad (4)$$

where X denotes the matrix of token-token co-occurrence counts, $f\left(X_{i,j}\right)$ is a weighting function, w_i and $\tilde{w}_j$ are word and separate context word vectors, respectively.

Limitations. Models like Word2vec or GloVe learn the embeddings at the token level. Although they can effectively capture the semantic properties of different tokens, they ignore the character level information. Considering the naming conventions for variables/methods in programming languages, for example, the Java camel cases or lowercase with tokens separated by underscores in Python, the insufficient use of the character level information is an non-negligible limitation for the embedding techniques that work at token level. Besides, these techniques also suffer from the out-of-vocabulary (OOV) problem for the tokens that do not appear in the existing training corpus, especially for programming languages, as developers usually combine different tokens together as one.

fastText fastText[3] (Bojanowski et al. 2017) is a recently proposed technique that exploits the internal structure of words (i.e., character level information), and tries to tackle the OOV problem by using character level units. In particular, fastText also considers the source code as plain text, where each token is represented by a set of n-grams appearing in this token. It then learns representations for the character n-grams and represents words as the sum of the character n-gram vectors. fastText extends the skip-gram model (cf. Section 2.2.1) by using a new softmax function which takes into account the subword representation. Similar to skip-gram, the goal of fastTest is also to maximize the function:

$$\mathcal{L} = \sum_{w_t \in \mathcal{D}} \sum_{w_c \in \mathcal{C}_{w_t}} \log p\left(w_c | w_t\right) \quad (5)$$

where $\mathcal{D}$ is the given sequence of tokens, $\mathcal{C}_{w_t}$ is the local context tokens of the target token w_t. The conditional probability, $p\left(w_c | w_t\right)$ is defined using the following softmax function:

$$p\left(w_c \mid w_t\right) = \frac{e^{s(w_t, w_c)}}{\sum_{w \in \mathcal{D}} e^{s(w_t, w)}} \quad (6)$$

$$s(w_t, w_c) = \sum_{g \in \mathcal{G}_{w_t}} \mathbf{z}_g^\top \mathbf{v}_{w_c} \quad (7)$$

where $s\left(w_t, w_c\right)$ is a score function, $\mathcal{G}_{w_t}$ is the set of n-grams appearing in w_t, $\mathbf{z}_g$ and $\mathbf{v}_{w_c}$ are vectors of the n-gram g and w_c, respectively.

[3] https://github.com/facebookresearch/fastText

```
public class Accept{
   ...
   public int a;
   ...
   public boolean accept(final File file) {
      ...
   }
}
```

Listing 3 An example code snippet with public decelerations for class, method and variable

Limitations. Although fastText has the ability to capture the character level information, the three embedding techniques discussed above still consider source code as plain text and take the textual context for embedding training. However, source code is by nature different from plain text, as it also contains structural information, which may be helpful for generating distributed representations of code tokens.

Code2vec Code2vec is a recently released code representation model by Alon et al. (2019) that takes into account the structural context (i.e., the abstract syntax tree representation of the source code). Given the AST representation of a code snippet, code2vec collects the paths between every two AST leaf nodes, and thus it represents the code snippet as a bag of paths. Code2vec then employs a path-attention network with fully connected layers to learn vector representations of the tokens and method names. The embeddings are trained in a supervised process with the objective to minimize the cross-entropy loss of predicting method names,

$$\mathcal{L}(p\|q) = -\sum_{y \in Y} p(y) \log q(y) \tag{8}$$

where, $p(y)$ is the distribution of the ground truth, if y is the true label of the case, then $p(y) = 1$, and 0 otherwise (i.e., binary indicator of whether y is the actual label.), $q(y)$ is the predicted probability.

Limitations. As code2vec is a supervised approach which means to guarantee good results, it requires human labeled data for training. Besides, it is trained on the task of method name prediction, and thus, the code embeddings produced by code2vec are task-specific and may not generalize well to other downstream tasks (Kang et al. 2019).

Limitations of Non-contextual Embeddings. Non-contextual embeddings (e.g., Word2vec and GloVe) have been playing an important role for improving the results of downstream tasks. Despite the powerful ability of representing source code tokens into vectors, these non-contextual embeddings also have their limitations: they are context-independent and assign each token with a single static representation. Therefore, they cannot effectively capture the different nuances of the same token in different contexts. For example, given the following code snippet in Listing 3, non-contextual embeddings assign the keyword "public" with only one unique vector, despite that they appear three times with different functionalities. However, considering the fact that they are modifiers for different levels of source code (i.e., class, attribute, and method), they should have different representations to better capture the properties.

2.2.2 Contextual Embeddings

To address the limitations of non-contextual embeddings, researchers recently proposed several methods to learn the context-dependent code embeddings (also known as PLM, short for

pre-trained language models), such as CodeBERT and CuBERT. Unlike static code embeddings, contextual embeddings are dynamic representations of tokens, that is the same token can have different representations based on the different surrounding context. For example, the three "public" keywords in Listing 3 would be assigned different vectors with respect to their different contexts. Recall that non-contextual embeddings act as a look-up table where each row of the real numbers are the vector representation of the corresponding token. However, for contextual embeddings, they are actually complicated transformer-based models, which take a sequence of code tokens as input and can return a set of fine-tuned embeddings for each token respectively. Formally, given a target token, w_t, together with the whole code snippet where it appears, $w_1, \cdots, w_t, \cdots, w_n$, the adjusted vector for w_t is

$$\left[\mathbf{v}_{w_1}, \cdots, \mathbf{v}_{w_t}, \cdots \mathbf{v}_{w_n}\right] = f\left(w_1, \cdots, w_t, \cdots, w_n\right) \tag{9}$$

where f is the pre-trained model with millions of parameters.

The pre-trained models (contextual embeddings) are usually trained on a general and large corpus and can be specialized to different downstream tasks by adding a task-specific layer and fine-tuning the parameters based on the training dataset of the specified task. In our work, to make the contextual embeddings suitable for our downstream tasks, we extract the embedding layer of the model and save the weights into the Word2vec format and then use them as described in Section 2.2.1.

In this section, we first introduce the BERT architecture as a background and then describe two recent applications of BERT on learning contextual embeddings for source code (i.e., CodeBERT and CuBERT).

BERT The contextual embeddings are popularized by BERT (Bidirectional Encoder Representation from Transformer) (Devlin et al. 2019). As shown in Fig. 3, BERT uses the bidirectional Transformer encoder which can effectively exploit both the left and right contexts of a target token.

In the work of Devlin et al. (2019), the authors present two main models: $\mathbf{BERT_{BASE}}$ which has a total number of 110M parameters and $\mathbf{BER_{LARGE}}$ with 340M parameters.

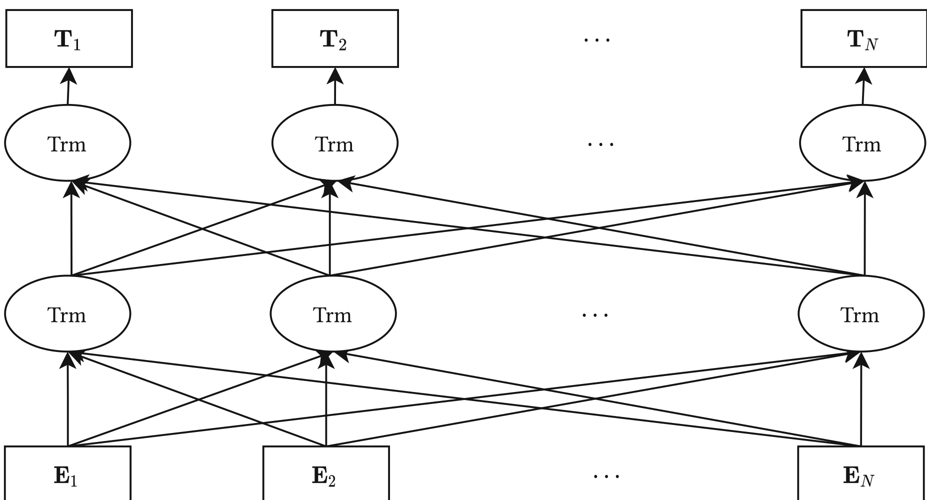

Fig. 3 The overall architecture of BERT. **E** and **T** are the input and output vectors respectively, and Trm is the encoder of Transformer

To effectively learn the model parameters, two objectives are designed for BERT: masked language model (MLM) and next sentence prediction (NSP).

Unlike the most common language model, which uses the previous sequence of tokens to predict the next token (the loss function is shown in (10)):

$$\mathcal{L} = -\sum_{w_t \in \mathcal{D}} \log p\left(w_t \mid w_1, \cdots, w_{t-1}\right) \tag{10}$$

in masked language model, some of the tokens in a sequence are randomly masked and the goal is to predict these masked tokens based on their surrounding unmasked context tokens:

$$\mathcal{L} = -\sum_{w_t \in \mathcal{M}} \log p\left(w_t \mid w_1, \cdots, w_{t-1}, w_{t+1}, \cdots, w_N\right) \tag{11}$$

where $\mathcal{M}$ represents the masked tokens and $w_1, \cdots, w_{t-1}, w_{t+1}, \cdots, w_N$ represent the rest of tokens in the sequence.

BERT also utilizes the next sentence prediction as the second objective to capture relationships between sentences for some sentence-based downstream tasks (e.g., question answering (QA)). In this task, the goal is to predict whether the sentence is the next sentence of the current:

$$\mathcal{L} = -\log p(y \mid \mathbf{s_t}, \mathbf{s_{t+1}}) \tag{12}$$

where $y = 1$ if $\mathbf{s_{t+1}}$ is the next sentence of $\mathbf{s_t}$ and $y = 0$ otherwise.

To apply BERT to downstream tasks, Devlin et al. (2019) also propose to use the two-step training strategy: (1) pre-training on unlabeled data and (2) fine-tuning using labeled data from the downstream tasks.

Since the release of BERT, researchers have made a few changes to the original model and achieved continuous improvements. For example, Liu et al. (2019) find that training the BERT model without the NSP loss can slightly improves downstream task performance. Thus, they propose RoBERTa (short for A Robustly Optimized BERT Pretraining Approach), which improves the performance of $\mathbf{BERT_{BASE}}$ on downstream tasks by re-training the model with larger batches and more data, but without NSP loss.

CodeBERT and CuBERT Given the revolutionized success of BERT for many NLP tasks, it has been widely applied to many other domains. For example, Feng et al. (2020) propose CodeBERT, which shares exactly the same model architecture as $\mathbf{RoBERTa_{BASE}}$. Kanade et al. (2020) propose CuBERT to learn contextual code embeddings using the $\mathbf{BERT_{LARGE}}$ model.

Both CodeBERT and CuBERT use the BERT model architecture and treat the source code as plain text for training. The differences between CodeBERT and CuBERT mainly come from the way of constructing the training corpus. CodeBERT considers the natural language texts (i.e., the description documentation of source code) and source code as two different types of data and constructs the training corpus (i.e., bimodal data and unimodal data) based on these two types of data. However, CuBERT does not separate the natural language texts and source code and mix natural language tokens with source code tokens during training.

Limitations. Based on our understanding, we find that there may still exist limitations for these two models. The first one is that they both treat the source code as plain text and do not consider the structural information explicitly. Another limitation is the high computation cost for training such huge models (both have millions of parameters.), making it almost impossible for us to modify and re-train the models by ourselves. For example, CodeBERT

spends more than ten days to finish the training using 16 interconnected NVIDIA Tesla V100 GPUs.

3 StrucTexVec: Embedding with Structural and Textual Information

To understand the impact of the structural and textual information on the performance of downstream tasks, we propose StrucTexVec, which consists of a context generation phase followed by a two-stage embedding learning phase. Figure 4 outlines the overall framework of StrucTexVec. StrucTexVec preprocesses a collection of source code files to generate the textual and structural context. In the embedding learning phase, the customized dependency-based skip-gram technique (Li et al. 2018a) is used to train the token embeddings based on the structural context. Then, to incorporate the textual information, StrucTexVec re-trains the token embeddings (our focus is the token embeddings, and thus the path embeddings are ignored during the re-training stage) on the tokenized textual context. Below, we elaborate on each of the phases of StrucTexVec.

3.1 Context Generation

In this section, we describe the procedures of generating the textual and structural context from source code files.

3.1.1 Textual Context Generation

Word2vec, GloVe, and fastText all use a window with a fixed length to construct the target word's context (Mikolov et al. 2013a, b; Pennington et al. 2014; Bojanowski et al. 2017) from the training corpus. In software engineering tasks, many existing approaches consider the source code as plain text (i.e., sequences of tokens) and achieve promising results (Allamanis et al. 2014; Allamanis et al. 2016; Hindle et al. 2012; Wang et al. 2016). Similarly, in this work, we also utilize the textual context and treat the source code files as plain text.

As described in Section 2.1, to convert the source code into the trainable textual context, the source code is first tokenized into a sequence of tokens, where all the non-identifiers (e.g., quotation marks) are removed. Meanwhile, following previous studies (Alon et al. 2019; Kang et al. 2019; Chen et al. 2016), the tokenized source code are lowercased.

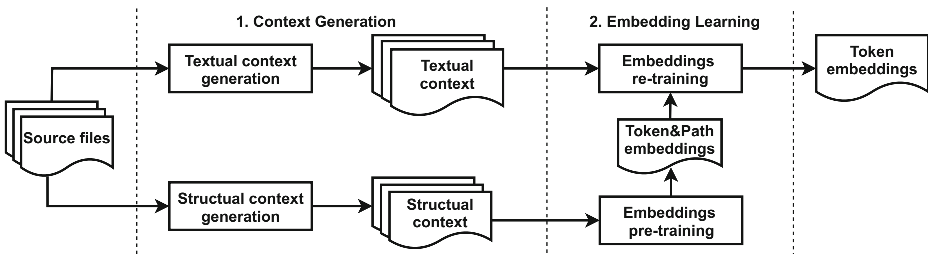

Fig. 4 The overall framework of StrucTexVec

3.1.2 Structural Context Generation

Apart from using the textual context for source code embedding training, some researchers (Alon et al. 2018; Bielik et al. 2016; Raychev et al. 2015) also utilize the structural context for software engineering tasks. Therefore, in our work, we also adopt the structural context. In particular, to enrich the context, our structural context contains three components: (1) AST paths, (2) method calls, and (3) variable references.

In StrucTexVec, we use srcML[4] (Collard et al. 2011) to represent source code as abstract syntax trees. As described in Section 2.1, the leaf nodes are tokens in the source code which are connected by a set of srcML tags that provide the syntax structure of the code.

Based on the XML tree representation provided by srcML, we then extract the structural context into a sequence of path triples. Our work shares an analogous way with that of Alon et al. (2019) to extract within-method triples. However, rather than only considering the information within a single method as in Alon et al. (2019), we enrich the structural context by mining the following three types of context: (1) AST paths, (2) method calls, and (3) variable references as described below.

AST Path Context Given the ASTs of the source code, we perform a structural traversal to extract a collection of path triples. In each triple, $\langle w_1, p, w_2 \rangle$, w_1 and w_2 are two different leaf nodes in one method's AST, p is the shortest path between these two nodes. The leaf nodes are source code tokens and the shortest path describes the syntactic relationship between any two of them. Algorithm 1 presents the details to construct such path triples.

Algorithm 1 TripleGeneration.

Input: AST of a method, M.
Output: A collection of path triples, T.

```
1  begin
2      T ⟵ ∅
       // Extract all the leaf nodes of M
3      leafNodes ⟵ FindLeafNodes(M)
4      foreach w1 ∈ leafNodes do
           // Find the path from root to the leaf node
5          path2w1 ⟵ GetPathFromRootTo(w1)
6          foreach w2 ∈ leafNodes do
7              if w1 ≠ w2 then
8                  path2w2 ⟵ GetPathFromRootTo(w2)
                   /* Return the longest common path prefix of
                      path2w1 and path2w2. */
9                  commPrefix ⟵ GetCommonPrefix(path2w1, path2w2)
10                 remove the subpath commPrefix from path2w1 and path2w2
                   /* Concatenate path2w1 and path2w2 with the last
                      element in commPrefix. */
11                 p ⟵ path2w1 + commPrefix[-1] + path2w2
12                 T ⟵ T + < w1, p, w2 >
13     return T
```

[4] https://www.srcml.org/.

For example, as shown in Fig. 1, given two source code tokens, e.g., "public", and "accept", the AST node sequences in the shortest path is $\langle specifier, type, function, name \rangle$. Considering the traversal directions, the final representation of the path becomes $specifier^{\uparrow}$-$type^{\uparrow}$-$function$-$name^{\downarrow}$, where the $\uparrow$ and $\downarrow$ are traversing directions and no direction means that it is an inflection node of a path. Thus, we get the path triple, ⟨ "public", $specifier^{\uparrow}$-$type^{\uparrow}$-$function$-$name^{\downarrow}$, "accept"⟩. Similarly, we can change the target node and the source node to collect more path triples for embedding learning.

Method Call Context We aim to extract the method calls within one project. We first identify all project-defined methods and the methods that are called by the identified project-defined methods. Then, we start to collect the call chain information between these methods. To accelerate the process of constructing call graphs, we use a heuristic that involves only faster shallow exact method name matching. More specifically, srcML provides a *function* tag that helps us to identify all the project-defined methods and a *call* tag to label the methods that are called by another method.

For example, the previous code snippet (see Listing 1) defines a method "accept", and assume it is called in another method that is defined in this project, "connect", and therefore, we have the triple, ⟨"connect", *call*, "accept"⟩, where the "accept" is the project-defined method name and called by the method "connect".

Variable Reference Context We also extract the variable references. We first identify all class variables and instance variables in one Java file using srcML, and then we collect methods that contain references to the previously identified variables by using the heuristic of exact variable name matching.

For example, in the previous code snippet (see Listing 1), the variable "fileFilters" is declared and initialized as an instance variable and referenced in method "accept", and therefore, we have the triple, ⟨"accept", $reference$, "fileFilters"⟩, where "accept" is the method which contains references to the instance variable "fileFilters".

The output of our context generation phase (i.e., the structural and textual context) are used as the input for the embedding learning phase.

3.2 Embedding Learning

In order to combine both the textual and structural knowledge into code embeddings, StrucTexVec adopts a two-step training strategy: (1) pre-training token and path embeddings using the customized dependency-based model (Li et al. 2018a) and (2) re-training the token embeddings using Word2vec (Mikolov et al. 2013a, b).

3.2.1 Path-Based Model for Embedding Pre-training

As explained in Section 2, the original Word2vec models use a local fixed-size window to construct a word's context, and then the context words are used for embedding training (Mikolov et al. 2013a, b). Different from the original models, Li et al. (2018a) improve Word2vec by integrating the syntactic dependency information between words into the embeddings. Since it is a recently released model and achieves competitive results on different tasks in natural language processing (Li et al. 2018a), in this work, following previous

work (Bojanowski et al. 2017) which extends the skip-gram model of Word2vec, we also customize the improved skip-gram as a path-based skip-gram for training code embeddings, aiming to incorporate the structural context of source code.

Path-Based Skip-Gram Figure 5 is an overview of the customized path-based skip-gram (PSG). In the original work of Li et al. (2018a), a word is modeled by its context of the syntactic dependency information. In our model, words are replaced with the tokens in the source code, and the dependency information is changed to the paths between these tokens in the ASTs of the source code. By doing this, we can apply the model to our extracted structural context. Following Li et al. (2018a), we use negative sampling to improve the computation efficiency. As Fig. 5 shows, in the modified model, the token (i.e., $\mathbf{V}_{\text{``}public\text{''}}$) is the target token, the path (i.e., $\mathbf{V}_{specifier^{\uparrow}\text{-}type^{\uparrow}\text{-}function\text{-}name^{\downarrow}}$) is the token's connecting path, the negative token (i.e., $\mathbf{V}_{\neg\text{``}public\text{''}}$) is selected from the vocabulary and the negative path (i.e., $\mathbf{V}_{\neg specifier^{\uparrow}\text{-}type^{\uparrow}\text{-}function\text{-}name^{\downarrow}}$) is selected from the path sets. All the negative samples are randomly selected based on the frequency of occurrence in the training corpus. We build two vocabularies, the tokens in the training corpus and those in the extracted paths. The vector representations of both vocabularies are updated during the training process.

After that, we concatenate the two negative parts together and form a negative sample for later embeddings learning. We take the following as the objective function of the model:

$$\mathcal{L} = \sum_{w_t \in \mathcal{D}} \prod_{\widetilde{w_t} \in \mathcal{C}_{\mathcal{D}}(w_t)} \prod_{u \in \{w\} \cup NEG^{\widetilde{w_t}}(w_t))} \mathcal{L}(w_t, \widetilde{w_t}, u)), \tag{13}$$

where $NEG^{\widetilde{w_t}}(w_t))$ is defined as the negative sampling set for target token w_t, $\mathcal{C}_{\mathcal{D}}(w_t)$ is the context set for w_t, and $\mathcal{L}(w_t, \widetilde{w_t}, u) = L^{w_t}(u) \cdot \log[\sigma(x_{\widetilde{w_t}}^{\mathsf{T}} \theta^u)] + [1 - L^{w_t}(u)] \cdot \log[1 - \sigma(x_{\widetilde{w_t}}^{\mathsf{T}} \theta^u)]$, where $\sigma(\cdot)$ is the *sigmoid* activation function, θ^u is the parameter vector of NS neuron, and $L^{w_t}(u)$ is an indicator function of which value depends that u is a positive example or negative example.

We would like to note that the path-based model is different from code2vec in the following aspects: (1) our path-based model is an unsupervised approach, which does not require any manual labeling of the training data, aiming to produce more generalizable embeddings; (2) we attempt to include more types of information, such as the method call context and the variable reference context, which are not included in code2vec and Word2vec.

The path-based model produces a vector for each token and each path that captures the structural context of the source code. (i.e., the Token&Path embeddings in Fig. 4).

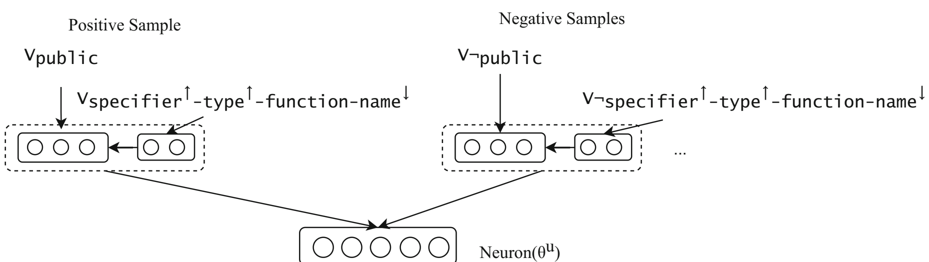

Fig. 5 The overview of the path-based skip-gram model with negative sampling (take "public" as an example)

3.2.2 Word2vec for Embedding Re-training

To further incorporate the textual context of code tokens, we adopt the original skip-gram to re-train the code embeddings produced by the path-based skip-gram. Here, we use the token vectors produced by the pre-training stage to initialize the embeddings instead of using the original random initialization. As introduced in Section 2.2.1 skip-gram takes a sequence of tokens as input, in this work, words are replaced with tokens in the textual context.

The output of our two-stage embedding learning process (i.e., the token embeddings) are used as the input for our downstream tasks.

4 Experimental Setup

In this section, we present details of our dataset used for training the code embeddings. We also introduce six common downstream tasks for quantitative evaluation of the pre-trained embeddings, three of which, i.e., (1) code comment generation, (2) code authorship identification and (3) code clone detection, are used for evaluating pre-trained embeddings in prior research (Kang et al. 2019); while (4) source code classification, (5) logging statement prediction and (6) software defect prediction, are newly added in this paper.

4.1 Dataset for Learning Pre-trained Embeddings

In this work, we use the *Java-Small* dataset[5] to build the pre-trained embeddings for all the non-contextual embedding techniques. This dataset is collected from Java projects hosted on GitHub. Note that for CodeBERT and CuBERT, due to the limitations of the computation resources, we use their released models, instead of training CodeBERT and CuBERT from a scratch. We save their embedding layers into Word2vec/GloVe format to integrate into our evaluation pipelines.

After fetching the files of the training projects, we first perform filtering to remove the irrelevant files and only keep the source code files with the *.java* extension. As illustrated in Fig. 4, we composite two types of contexts from the filtered source code files: (1) structural context, which refers to extracted path triples in the source code and (2) textual context, which is the plain text of Java files.

4.2 Settings for Embedding Learning

At the pre-training stage, the token vectors are randomly initialized and trained using the path-based model. At the re-training stage, the token vectors are initialized by the embeddings produced by the path-based model and further trained using the skip-gram implemented in Gensim[6] (Řehůřek and Sojka 2010).

Moreover, to investigate the impact of pre-trained embeddings on software engineering tasks, we also evaluate the other six existing embedding techniques as described in Section 3.2.2 and compare their performance with the models without pre-trained embeddings. To ensure a fair comparison across all the embedding techniques, we either follow

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

[6] https://radimrehurek.com/gensim/

the parameter settings in previous work (e.g., 128 dimensions) or use the default parameter values when the parameters are not specified in previous work (e.g., training epochs). To avoid bias, we do not try to fine-tune these settings only for our method. The detailed parameter settings are shown in Table 2.

The embeddings of StrucTexVec, GloVe, fastText and Word2vec are all trained in CPUs and code2vec is trained in an NVIDIA GTX 1080 Ti GPU, and it takes less than 30 min to finish the training process of each of the embedding techniques, which is acceptable. For CodeBERT and CuBERT, we do not train them from a scratch and choose to use the already released models, as it requires not only more GPUs but also a long period (several days) to finish the training. For example, CodeBERT spends more than ten days to finish the training using 16 interconnected NVIDIA Tesla V100 GPUs.

4.3 Evaluation Tasks

In this section, we briefly describe the six downstream tasks that are used to evaluate our pre-trained embeddings, as well as the corresponding datasets and evaluation metrics. Five of the six tasks use neural network-based methods and one task (i.e., software defect prediction) uses a traditional machine learning method (i.e., logistic regression). Our focus is the impact of the embeddings on different downstream tasks, i.e., whether the pre-trained code embeddings can improve the model performance or not.

Besides, for comparison between different embedding techniques, all the embeddings are used in the same manner for downstream tasks, that is for each task, only the embeddings are changed, and other parameters are kept the same. For example, for deep learning-based tasks, the code embeddings are used to initialize the embedding layer, and OOV tokens are randomly initialized, which can be later updated based on the training data from different tasks.

Code Comment Generation is a task to automatically generate code comments for a code snippet (McBurney and McMillan 2014; Sridhara et al. 2010; Moreno et al. 2013; Hu et al. 2018), which is helpful in program understanding and maintenance. Code comment generation is considered as a downstream task in previous work (Kang et al. 2019) to evaluate the effectiveness of code embeddings.

In our work, we follow the work of Kang et al. (2019) and evaluate the embeddings based on the approach proposed by Hu et al. (2018). Hu et al. (2018) treat the code comment generation task as a neural machine translation task, where the input is the source code

Table 2 Parameter settings for different embedding techniques

	Non-contextual embeddings					Contextual embeddings	
	Word2vec	GloVe	fastText	code2vec	StrucTexVec	CodeBERT	CuBERT
Vocabulary	109,743	192,363	109,743	507,271	192,362	50,265	50,297
Epoch	5	5	5	20	10 & 5	–	2
Window	5	5	5	5	5	–	–
Negative	5	–	5	–	4 & 5	–	–
Dimension	128	128	128	128	128	768	1024

Note: (1) Dimension: following the settings in prior work (Li et al. 2018a; Alon et al. 2019; Zhang et al. 2019), we set the dimension of the trained token vectors to 128. For CodeBERT and CuBERT, we take the results from their released models. (2) Epoch: StrucTexVec contains a two-step training, in our experiments, 10 epochs for pre-training and 5 epochs for re-training, both are the default values of the released source code

snippet and the output is the code comment. Thus, they adopt an encoder-decoder model. In particular, they use two Long Short-Term Memory (LSTM) layers for both encoder and decoder, and 500 hidden units for each layer. During model training, both the learning rate and dropout rate are set to 0.5. The model is trained for 50 epochs. The training dataset is provided by Hu et al. (2018), which was initially collected from GitHub. The dataset contains over 330,000 <method, comment> pairs for training and 5000 pairs for validation and 5000 for testing.

We follow the work of Hu et al. (2018) and Kang et al. (2019) and use the machine translation evaluation metric BLEU (Papineni et al. 2002) to measure the quality of the generated comments.

Code Authorship Identification is a task of identifying the author of a given code (Abuhamad et al. 2018; Islam et al. 2015). This task has attracted increasing attention in the field of privacy and security, where it can be used to identify the authors of malware and other malicious programs. We follow the work of Kang et al. (2019) and select this task as a downstream task.

In our work, we evaluate the embeddings based on the approach proposed by Kang et al. (2019). Kang et al. (2019) treat the code authorship identification task as a multi-class classification problem, where the input is the code snippet and output is the author. They build a neural network-based model, which contains two LSTM layers and a fully connected layer. During model training, we set the batch size to 64 to guarantee a relatively smaller training loss. The model is trained for 50 epochs. We use the same dataset as that of Kang et al. (2019), which was collected from Google Code Jam. The dataset contains 2250 programs written by 250 authors, among which there are 2000 programs for training and 250 programs for testing, and within each part the classes are balanced.

We follow the previous work (Kang et al. 2019) and use the test accuracy to measure the performance of the models with different code embeddings.

Code Clone Detection is a task of checking whether two code snippets are similar or not. It is a useful task for program maintenance (Mayrand et al. 1996; Dubinsky et al. 2013; Thummalapenta et al. 2010; Barbour et al. 2011; Kamiya et al. 2002; Sajnani et al. 2016; White et al. 2016; Wei and Li 2017). For example, if a bug is identified in one code fragment, all the other duplicate code fragments also need to be checked for the same bug. This task is also identified as a downstream task to evaluate code embeddings in prior work (Kang et al. 2019).

In general, there are four different types of code clones based on the kind of similarity two code fragments have (Roy and Cordy 2007):

- Type-1 clone refers to identical code fragments except for changes in comments, whitespace and layout.
- Type-2 clone refers to identical code fragments except for differences in identifier names or literal values, comments, types, and layouts.
- Type-3 clone refers to code fragments that are syntaxally similar but have statements added, modified, or removed.
- Type-4 clone refers to code fragments that are syntaxally different but with the same functionality.

In our work, we use the approach proposed by Zhang et al. (2019), as it is recently proposed and gives competitive results. Zhang et al. (2019) consider the code clone detection task as a binary classification task, where the input is two code snippets and the output is 1 if they are duplicate and 0 otherwise. They use a bidirectional Recurrent Neural Network

based model, and the model is trained for 15 epochs with a batch size of 128. This task contains two datasets, which are constructed from standard BigCloneBench (BCB) (Svajlenko et al. 2014) and Online Judge system (namely, OJClone).

We follow the work of Zhang et al. (2019) and use F1-measure (F1) as the evaluation metric in this task.

Source Code Classification is a typical classification task that classifies code fragments into corresponding categories. This task is identified as a downstream task as it is widely studied in the literature (Zhang et al. 2019; Gilda 2017; Mou et al. 2016; Kawaguchi et al. 2006; Vásquez et al. 2014) and has various applications (Zhang et al. 2019; Gilda 2017; Mou et al. 2016; Kawaguchi et al. 2006; Vásquez et al. 2014).

In our work, the source code classification task is considered as a multi-class classification problem. We apply a convolutional neural network proposed by Kim (2014) to the source code. We choose to use this model as it is widely used for classification tasks and achieves competitive results, and we want to cover more different types of neural network models. The input to the model is the source code and output is its class label (e.g., functionality). During training, we use the default parameters, that is we set the learning rate to 0.01, batch size to 64 and dropout rate to 0.5. We train the model with 50 epochs. The dataset is collected from the Online Judge system[7] and provided by Mou et al. (2016).

Similar to the task of code authorship identification, we follow the work of Zhang et al. (2019) and use the test accuracy as the evaluation metric.

Logging Statement Prediction is a task of predicting whether there is a need to insert logging statements for a given code snippet. Logging statements play important roles in the daily tasks of developers (Li et al. 2018b; Ding et al. 2022), and this task can provide logging suggestions that are helpful for software developers (Li et al. 2018b).

Li et al. (2018b) consider the logging statement prediction task as a binary classification problem, where the input is the code snippet without the logging statement and output is the decision whether to insert a logging statement or not. In this task, we also use the approach proposed by Kim (2014) as in the task of source code classification. The evaluation dataset is provided by Li et al. (2018b), which contains five subject systems.

In this work, same as prior work (Li et al. 2018b), the balanced accuracy (BA) metric is used to evaluate the performance of the model with different embeddings.

Software Defect Prediction is a task of predicting whether the code snippet contains defects or not. Defect prediction can help avoid future bugs in software releases and improve the quality of software (Wang et al. 2016). This task is selected as a downstream task because it is widely studied in the literature (Wang et al. 2016).

For this task, following prior work by Wang et al. (2016), we leverage the Logistic Regression (LR) classifier where the input features are the average of the embeddings of the code tokens in a file and the output is the 1 if there is a defeat detected or 0 otherwise. The dataset is provided by Wang et al. (2016) and they use F1 score as the evaluation metric. Thus, we follow their work and use the same metric for this task.

5 Experimental Results

In this section, we show the quantitative results on the previously identified downstream tasks, and based on the results, we aim to answer the following research questions:

[7] https://sites.google.com/site/treebasedcnn/

RQ1: How Effective Are Pre-trained Embeddings in Improving the Performance of Downstream SE Tasks?

To evaluate the effectiveness of using pre-trained code embeddings, we compare models using pre-trained embeddings, including the non-contextual embeddings and contextual embeddings, to models that do not use pre-trained code embeddings (i.e., None). **Models using pre-trained embeddings can perform better than models without pre-trained embeddings**. Table 3 shows the evaluation results of utilizing different code embeddings on six downstream tasks. The best results are highlighted in bold. As shown in the table, models using pre-trained embeddings achieve the best results in all the six tasks. For example, by using embeddings produced by code2vec, we obtain a 2.6% absolute increase in accuracy on source code classification task compared to the model without pre-trained embeddings. In addition, we observe that there is a slight improvement in the tasks of code comment generation and logging statement prediction when using the embeddings generated by StrucTexVec than other prior non-contextual pre-trained embeddings that only consider structural or textual information, which implies the effectiveness of combining both the textual and the structural context in creating generalizable code embeddings. We further analyze the datasets of these two tasks and find that when there is a relatively larger training dataset, StrucTexVec performs better. For example, for the task of code comment generation, there are more than 330,000 training samples, and there are more than 20,000 training samples for the task of logging statement prediction. Moreover, we find that in all the evaluated tasks, code embeddings trained in an unsupervised manner do not always outperform embeddings trained on a specific task. The results demonstrate that existing neural networks can benefit from the pre-trained embeddings, which challenges the findings of Kang et al. (2019). The findings highlight the potential of applying well-trained code embeddings to downstream SE tasks.

On the other hand, we find that including more types of training information cannot always guarantee better results, and other factors (e.g., training model, different preprocessing strategies) can also have an impact on the quality of the generated embeddings. For example, both the CodeBERT and CuBERT are trained only based on the textual information, but both of them achieve results comparable with the approaches considering both the textual and structural information (e.g., StrucTexVec). Besides, although StrucTexVec attempts to include more structural information (i.e., AST paths, method call, and variable reference), it cannot always outperform code2vec in the six downstream tasks. To understand why the performance may differ even though we use more types of training information, we do a further analysis between these two techniques. As a result, we summarize that the performance difference can come from the following aspects: (1) The difference between the training objectives. For the training objectives, our method is a task-agnostic embedding approach, which does not need to be trained together with the downstream tasks; code2vec is task-specific and trained together with the downstream task (i.e., method name prediction.). Thus, for the tasks that share similar intrinsic properties with the downstream task that is used to train the embeddings, code2vec would perform better. For example, the task of source code classification, which is to classify code fragments into corresponding categories, is similar to the method name prediction, where a category (i.e., method name) is assigned to a code fragment based on its functionality. (2) The difference between the training models. As we described in Section 3.2, during the embedding learning, we simply concatenate the embeddings of the token and the AST path into one single vector and then use this vector for future training. However, in code2vec, the paths and tokens are treated

Table 3 Evaluation results on the test set of six downstream tasks. The second last row shows the percentage of the best result produced by each approach on 22 datasets and the last row is the weighted averaged percentage of best results on six downstream tasks (i.e., each task's contribution to the percentage is weighted by its number of datasets)

Downstream tasks	Evaluation metrics	Dataset	None	Non-contextual embeddings					Contextual embeddings	
				Word2vec	GloVe	fastText	code2vec	StrucTexVec	CodeBERT	CuBERT
Code comment generation	BLEU	GitHub	14.9	15.4	15.9	14.6	15.3	16.0	**16.7**	16.1
Code authorship identification	Accuracy	Google Code Jam	87.5	80.2	87.5	77.1	85.4	86.5	87.0	**89.1**
Code clone detection	F1	BCB	92.7	**93.8**	**93.8**	**93.8**	93.5	93.6	93.5	93.6
		OJClone	85.1	86.8	81.4	78.0	**89.7**	88.1	85.9	81.6
		Avg.	88.9	90.3	87.6	85.9	**91.6**	90.9	89.7	87.6
Source code classification	Accuracy	OJ dataset	88.5	87.0	89.2	77.7	**91.2**	89.1	79.8	75.8
Logging statement prediction	Balance Accuracy	Airavata	**95.6**	94.3	94.2	93.1	94.8	94.5	**93.8**	93.4
		Camel	76.6	77.8	77.5	76.4	77.4	**79.2**	77.1	75.0
		CloudStack	85.9	86.0	85.5	84.7	86.9	**87.3**	86.0	86.7
		Directory-Server	82.9	84.1	85.6	84.7	84.0	86.6	**88.0**	81.9
		Hadoop	76.7	73.6	71.5	71.7	72.3	71.0	75.4	**77.6**
		Avg.	83.6	83.2	82.8	82.1	83.1	83.7	**84.1**	82.9
Software defect prediction	F1	Ant 1.5–>1.6	28.0	35.5	36.0	32.9	47.6	34.2	36.4	**54.8**
		Ant 1.6–>1.7	33.1	44.9	45.1	39.6	48.4	43.4	51.9	**52.9**
		Camel 1.2–>1.4	23.3	43.3	45.5	43.8	43.2	**46.8**	45.6	44.2
		Camel 1.4–>1.6	26.3	47.0	49.8	46.0	50.0	50.2	**51.2**	50.3
		jEdit 3.2–>4.0	32.7	52.0	56.2	55.9	56.6	59.5	**61.5**	59.4
		jEdit 4.0–>4.1	40.6	60.5	60.1	59.7	57.9	**64.7**	62.4	59.9

Table 3 (continued)

Downstream tasks	Evaluation metrics	Dataset	None	Non-contextual embeddings					Contextual embeddings	
				Word2vec	GloVe	fastText	code2vec	StrucTexVec	CodeBERT	CuBERT
		Log4j 1.0–>1.1	45.5	65.7	67.6	62.7	66.7	62.7	70.4	**72.0**
		Lucene 2.0–>2.2	58.1	62.6	60.6	**65.1**	63.3	63.9	63.8	62.6
		Lucene 2.2–>2.4	60.8	**66.3**	64.7	65.5	60.6	65.2	64.4	63.8
		POI 1.5–>2.5	68.1	64.8	80.1	67.4	81.7	77.8	83.4	**85.1**
		POI 2.5–>3.0	67.5	72.4	72.9	72.6	**74.8**	71.4	72.3	72.4
		Xalan 2.4–>2.5	49.9	41.7	42.9	44.3	**53.6**	47.5	41.5	50.4
		Avg.	44.5	54.7	56.8	54.6	58.7	57.3	58.7	**60.6**
Percentage of best result (%)			4.5	9.1	4.5	9.1	18.2	18.2	18.2	**27.3**
Weighted averaged percentage of best result (%)			3.3	9.7	9.7	9.7	**27.8**	9.4	20.0	20.0

Note: (1) The weighted averaged percentage of best results on six downstream tasks for each technique is calculated as: $\frac{\sum_{t \in tasks} \frac{\#\ of\ best\ performing\ datasets}{\#\ of\ total\ datasets\ in\ t}}{6} * 100\%$. (2) For CodeBERT and CuBERT, we save their embedding layers into Word2vec/GloVe format to integrate to our evaluation pipeline. Although they all have achieved comparable performance in the downstream tasks, CodeBERT and CuBERT are contextual embeddings, which means the way we use them may not unleash their full power

differently at the beginning and then fused together by using an attention layer, which is better at transmitting the information between the paths and tokens than our method. As a result, even if we use more information in our method, due to the limited ability to embed such information into the code embeddings, our method performs worse on some datasets. (3) The difference between the vocabulary sizes. We find that there are 507,271 tokens for code2vec and 192,362 tokens in our generated embeddings, which means more tokens are filtered out for our method during the pre-processing step, and this poses a non-negligible effect on the quality of the generated embeddings.

Besides, for the performance of non-contextual embeddings on the downstream tasks, we find that the vocabulary size of the embeddings has a more significant impact on the traditional machine learning models than on deep learning-based models. For example, Word2vec and fastText have a relatively smaller vocabulary size and perform worse on the task of software defect prediction that uses a traditional machine learning model. On the contrary, code2vec has the largest vocabulary size and performs better on the same task. This can be explained by the fact that a smaller vocabulary size causes more OOV tokens, and thus weaken the representation ability of the code embeddings, especially considering the fact that the code embeddings are directly used as features for the traditional machine learning models. However, for deep learning-based models, the embeddings are only used to initialize the weights of the first embedding layer, and the weights are later adjusted to better fit the training data. As a result, the impact of OOV tokens may be diminished or even erased during the model training and weights updating.

For a specific downstream task, different embedding methods can result in diverse performance, and there does not exist an embedding technique that outperforms others in all nor even majority of the tasks. Table 3 compares the results for applying different embeddings to six downstream tasks. The tasks of code comment generation and code authorship identification illustrate a diverse performance that may be caused by different embeddings. For example, on the code comment generation task, using embeddings trained on fastText can only have a 14.6 of BLEU, compared to 15.9 when using embeddings trained on GloVe. In addition, different evaluation tasks result in different orderings of embedding techniques, raising the question that there may not exist a single optimal vector representation for all SE tasks. For instance, code embeddings suitable for source code classification (e.g., code2vec) even perform no better than random embeddings on code authorship identification. This may come from the fact that different SE tasks might highly differ in their nature and thus require different external information to boost the performance. The findings suggest that one should be very careful with the selection of code embedding techniques before starting the model training in terms of different tasks. In particular, for the non-contextual embeddings that leverage structural information of the code, including StrucTexVec and code2vec, perform better than other non-contextual embeddings techniques and have the best results in five out of six downstream tasks.

Using pre-trained embeddings may not always improve the performance of downstream tasks significantly. We find that by using pre-trained embeddings, although we can obtain an increase in performance on different tasks, the improvement may be limited. For example, in the task of logging statement prediction, we only obtain a maximum increase of 0.5% by utilizing the pre-trained embeddings (the model without pre-trained embeddings compared to that using pre-trained embeddings produced by CodeBERT). In addition, using pre-trained embeddings causes a decrease in accuracy for the task of code authorship identification. This observation is similar to that of prior studies (Li et al. 2018a; Vashishth et al. 2019; Kang et al. 2019). One possible reason is that the neural network-based models themselves are powerful enough and already have good results, thus it is difficult to have a large

improvement (e.g., the great performance in code clone detection task shown in Table 3). The result indicates the limited effect of pre-trained embeddings, as they only work on initializing the embedding layer for neural network-based models. This confirms the findings of Kang et al. (2019). Namely, code embeddings may not be a key role in boosting the performance of deep learning models. Software engineering researchers and practitioners should not only rely on using embeddings to improve their automated techniques.

> In general, using pre-trained code embeddings can improve the performance of downstream SE tasks. However, different embeddings techniques can result in diverse performance. Practitioners and researchers should be careful when selecting the embedding techniques for their specific tasks, as there is no single best solution.

RQ2: How do the Structural and the Local Textual Information Affect the Performance of the Pre-trained Embeddings?

To verify the effectiveness of incorporating different information extracted from source code (i.e., structural and local textual information), we design ablation experiments on these six downstream tasks.

First, to analyze the effect of the structural information, we treat the source code as plain text and only learn from the local textual information. More specifically, we remove the pre-training stage from StrucTexVec, which results in the original Word2vec model. Then, to analyze the performance gain achieved due to the utilization of the local textual context, we train the embeddings only based on the structural information extracted from the source code. In other words, we remove the re-training stage from StrucTexVec, resulting in the path-based skip-gram with negative sampling (i.e., StrucTexVec^{-text} in Table 4). The results of our ablation experiments are shown in Table 4.

The structural information extracted from the source code can improve the performance of the code embeddings. By comparing the results of Word2vec with that of StrucTexVec, in total, we find that StrucTexVec can outperform Word2vec in all downstream tasks. The comparison results demonstrate that incorporating the structural context can help improve the performance of the embeddings. For example, on the code authorship identification task, by pre-training the embeddings using the structural context, StrucTexVec has an accuracy of 86.5% compared to 80.2% without the pre-training phase. This is consistent with the observation of Zhang et al. (2019). We consider that the poor performance of GloVe in the work by Kang et al. (2019) may be related to the exclusion of structural information in the embeddings.

Code embeddings can benefit from the local textual context. According to the results of StrucTexVec^{-text} and StrucTexVec from Table 4, we find that StrucTexVec can achieve better performance than $StrucTexVec^{-text}$ in five out of six downstream tasks. The comparison results indicate that by re-training the embeddings on the textual context, we can achieve an overall better performance. For example, on the code authorship identification task, the embeddings trained only with the structural information have a lower accuracy compared to those use both structural and textual information (i.e., 79.7% vs. 86.5%). This is consistent with the intuition that developers put similar code statements together and thus using a local context window is able to capture some semantic information from the source code.

Table 4 Evaluation results of embeddings trained with and without the structural and local textual contexts. Word2vec is equivalent to the variant of StrucTexVec which removes the structural information from the training process; StrucTexVec^{-text} only utilizes the structural information for embeddings learning

Downstream tasks	Code comment generation	Code authorship identification	Code clone detection			Source code classification	Logging statement prediction					
Datasets	GitHub	Google Code Jam	BCB	OJClone	**Avg.**	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop	**Avg.**
StrucTexVec	**16.0**	**86.5**	93.6	**88.1**	**90.9**	89.1	94.5	**79.2**	**87.3**	86.6	71.0	**83.7**
Word2vec	15.4	80.2	**93.8**	86.8	90.3	87.0	94.3	77.8	86.0	84.1	**73.6**	83.2
StrucTexVec^{-text}	15.7	79.7	93.6	86.9	90.3	**89.7**	**95.3**	76.0	85.2	**90.0**	71.2	83.5

Downstream tasks	Software defect prediction												
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.**
StrucTexVec	34.2	43.4	**46.8**	**50.2**	**59.5**	**64.7**	62.7	63.9	65.2	77.8	71.4	**47.5**	**57.3**
Word2vec	35.5	44.9	43.3	47.0	52.0	60.5	**65.7**	62.6	**66.3**	64.8	**72.4**	41.7	54.7
StrucTexVec^{-text}	**38.1**	**50.2**	46.4	45.2	57.4	58.9	62.9	**66.9**	63.1	**81.2**	71.8	43.6	57.1

However, the benefit from the local textual context is limited for some downstream tasks We also find that the improvement is not always significant. For example, for the tasks of code comment generation, logging statement prediction, and software defect prediction, there is only a 0.2% to 0.3% absolute increase. Moreover, for some tasks and datasets, retraining the code embeddings using local textual context even causes a degradation of the performance. For example, the accuracy of source code classification task decreases from 89.7% to 89.1% when utilizing the local textual information. One possible reason is that the structural information extracted from the source code is rich enough for the downstream tasks and incorporating the local textual context cannot provide much more benefit.

The structural information has a stronger impact on the quality of the code embeddings than that of local textual information By comparing StrucTexVec^{-text} with Word2vec, we can see that StrucTexVec^{-text} outperforms Word2vec in four out six downstream tasks. For example, on software defect prediction task, removing the local textual information from StrucTexVec (i.e., StrucTexVec^{-text}) causes a degradation of 0.2%, while Word2vec only has an accuracy of 54.7%, which is a 2.6% absolute decrease. Our results suggest the promising research direction of utilizing the structural information for SE tasks.

> On one hand, we find that including structural or local textual information into embeddings can improve the performance of downstream SE tasks. On the other hand, the impact of structural information is stronger for some downstream tasks. This finding highlights the importance and effectiveness of incorporating AST-based structural information for SE tasks.

6 Discussion

In this section, we discuss our lessons learned and compare them with the findings of Kang et al. (2019). The summary of the comparison is presented in Table 5.

Models using pre-trained embeddings can perform better than models without pre-trained embeddings. The choice of embedding techniques has a non-negligible impact on the model performance. We evaluate the effect of utilizing pre-trained code embeddings on six downstream tasks. We observe that utilizing the pre-trained code embeddings can improve the performance of existing models which do not use the pre-trained embeddings in all downstream tasks. For example, in the task of code comment generation, using pre-trained code embeddings produced by StrucTexVec results in a BLEU score of 16.0, which is 7.4% relative higher than the model without the pre-trained embeddings. Our findings are different from that of Kang et al. (2019). The different results may be caused by (1) the selection of training corpus: we use the *Java-Small* dataset while Kang et al. (2019) chose the *Java-Large* dataset which may contain noise data that can affect the quality of the generated code embeddings; 2) we evaluated more code embeddings produced by different techniques, among which some embedding techniques cannot benefit the models without pre-trained embeddings. For example, using code embeddings produced by fastText negatively impacts the performance of existing models.

Simpler baselines may not always outperform complex techniques. The model parameters have a non-negligible impact on the performance of the deep learning-based models. In our evaluation, we see that on the task of code authorship identification, all of our reported results outperform the simpler approach that uses simple TF-IDF features reported by Kang et al. (2019) (i.e., an accuracy of 77%). The only difference is that we set

Table 5 Comparison with the findings from previous work (Kang et al. 2019)

Our findings	Findings from Kang et al. (2019)	Discussion
Models using pre-trained embeddings can perform better than models without pre-trained embeddings. The choice of embedding techniques has a non-negligible impact on the model performance.	Code embeddings cannot be used readily to improve the performance of simpler models.	In our evaluation, we find that in all six downstream tasks, the use of code embeddings can improve the performance of the models. The difference of the findings may be caused by (1) the training corpus, in this work, we select the Java-Small dataset 2) we evaluate more embeddings techniques on more downstream tasks.
Simpler baselines may not always outperform complex techniques. The model parameters have a non-negligible impact on the performance of the deep learning-based models.	Simpler baselines run faster and may outperform complex techniques.	In our evaluation, we see that on the task of code authorship identification, the use of code embeddings outperforms the simpler approach that uses simple TF-IDF features reported by Kang et al. (2019). The only difference is that we set the batch size to 64 instead of the default 128. Besides, to verify our findings, we implement another two simpler baselines which use traditional features for the software defeats prediction. As the results show, some of the embeddings still perform better. Meanwhile, our results concur with the findings of Kang et al. (2019) that other considerations (e.g., pre-processing) may have an impact on the performance of deep learning models.
Using pre-trained embeddings may not always improve the performance of downstream tasks significantly.	Code embeddings may not be able to boost the performance of neural network-based models.	Our findings are similar to that of Kang et al. (2019). Although we can obtain an increase in performance of different tasks, the improvement may be limited. One possible reason is that the neural network-based models themselves are powerful enough and already have good results, thus it is difficult to have a large improvement for some downstream tasks.

Table 5 (continued)

Our findings	Findings from Kang et al. (2019)	Discussion
Using pre-trained embeddings may not always improve the performance of downstream tasks significantly.	Code embeddings may not be able to boost the performance of neural network-based models.	Our findings are similar to that of Kang et al. (2019). Although we can obtain an increase in performance of different tasks, the improvement may be limited. One possible reason is that the neural network-based models themselves are powerful enough and already have good results, thus it is difficult to have a large improvement for some downstream tasks.
Composing a meaningful representation from a set of code tokens using pre- trained codeembeddings is a challenging task and further investigation of the composition of code embeddings are needed.	The composition of source code token embeddings requires further investigation.	To further investigate the composition of code embeddings, we do another four experiments for the task of software defect prediction, using different composition strategies. i.e., summation, averaging, TF-IDF weighted averaging and TF-IDF weighted summation. We find that simple averaging still is the best way to represent the source code.
Code embeddings can benefit from considering both the structural and textual information. And the embeddings produced by StrucTexVec and code2vec which leverage structural information of the source code perform better than other non- contextual embeddings that do not.	The poor performance of utilization of code embeddings may indicate that token embeddings learned over source code may not encode enough information usable for different downstream tasks	In our experiment, to check the effect of different information on generating the code embeddings, we use different training context and design ablation experiments. The results indicate the embeddings can encode different information for downstream tasks.

the batch size to 64 instead of the default 128. This finding shows that by tuning the parameters of the complex models, we can outperform the simpler baselines. Also, to verify our findings, we implement another two simpler baselines which use traditional features (i.e., PROMISE and TF-IDF) for the software defect prediction task, and we also observe that the use of pre-trained code embeddings can outperform the simpler baselines. Our results concur with the findings of Kang et al. (2019) that other considerations (e.g., pre-processing) may have an impact on the performance of deep learning models.

Using pre-trained embeddings may not always improve the performance of downstream tasks significantly. Our findings are similar to that of Kang et al. (2019), although we can obtain an increase in the performance of different tasks, the improvement may be limited. For example, in the task of logging statement prediction, the best-performing code embeddings only has a 0.5% absolute improvement compared to the model without the pre-trained code embeddings. One possible reason is that the neural network-based models themselves are powerful enough and already have good results and the code embeddings are only used for initializing the embedding layer of these models, thus it is difficult to have a large improvement for some downstream tasks. The results suggest researchers and developers should be careful when deciding whether or not to use pre-trained embeddings for their specific downstream tasks.

Further investigation of the composition of code embeddings is needed. Pre-trained code embeddings are numerical vectors for individual tokens in the source code and a key challenge is how to represent the whole code snippet with a sequence of code tokens. In practice, multiple methods for code embedding composition (i.e., combining the embeddings of each token in the source code to an embedding of the entire code snippet) are adopted, such as simply adding or averaging the embeddings of all the tokens. Similar to that of Kang et al. (2019), we find that the way of composition of code embeddings can impact the performance of the models. We implement four experiments for the task of software defect prediction with different composition strategies, i.e., summation, averaging, TF-IDF weighted averaging and TF-IDF weighted summation. For example, when the four composition strategies are performed on code2vec for the task of software defect prediction, we get the F1 scores, 52.1, 58.7, 53.3, and 53.7, respectively. The results show that different composition strategies can result in diverse results and simple averaging still is the best way to represent the source code.

Code embeddings can benefit from considering both the structural and textual information. And the embeddings produced by StrucTexVec and code2vec which leverage structural information of the source code perform better than embeddings that do not. In our experiment, to check the effect of different information on generating the code embeddings, we use different training context and design ablation experiments. The results indicate that except for the training techniques, the training context also impacts the quality of the code embeddings. Especially, the embeddings (i.e., StrucTexVec and code2vec) that incorporate the structural context yields better results than other non-contextual models. The results show that the embeddings can encode different information for downstream tasks.

7 Related Work

In this section, we discuss the prior research that proposes and applies code embeddings in software engineering tasks.

Source code embeddings play an important role in many SE tasks (Chen et al. 2016; Chen and Monperrus 2018; Wang et al. 2018; White et al. 2019; Harer et al. 2018; Pradel and Sen 2018; Alon et al. 2019; Allamanis et al. 2015; Büch and Andrzejak 2019), and researchers propose various approaches for learning code embeddings to assist in SE tasks.

Considering Source Code as Plain Text Among the proposed code embeddings techniques, some consider the source code as plain text and directly apply the word embedding techniques to source code. For example, Harer et al. (2018) propose a source-based model for automated software vulnerability detection. In their model, they first tokenize the collected open-source C/C++ programs into sequences of tokens and then apply Word2vec to convert code tokens into vector representations. The learned Word2vec representation of code tokens is finally fed into a TextCNN model (Kim 2014) for classification. Similarly, White et al. (2019) train source code embeddings using Word2vec from the normalized file-level corpora. The trained embeddings are then used for the initialization of the embedding layer of the recursive autoencoder which is a type of neural network that recursively learn the representations of the code snippet. In addition, Efstathiou and Spinellis (2019) adopt fastText (Bojanowski et al. 2017), which utilizes the subword information, to train code embeddings for different programming languages, including Java, Python, PHP, C, C++, and C#. However, they only propose potential applications of the models without evaluation. Theeten et al. (2019) propose to use the skip-gram model of Word2vec (Mikolov et al. 2013a, b) to generate embeddings for library packages of different programming languages, which are later used for retrieving the similar libraries of a given library.

Considering the Structural Information of Source Code Apart from the above mentioned direct use of word embedding techniques to source code, researchers have proposed to learn embeddings for specific software engineering tasks while considering the structure of source code. Zhang et al. (2019) propose to learn source code representations based on the abstract syntax trees. They train the program embeddings for two downstream tasks, i.e., code clone detection and source code classification. Similarly, Büch and Andrzejak (2019) implement an AST-based Recursive Neural Network (RNN) for code clone detection. Alon et al. (2019) propose code2vec, which also relies on the AST representation of the source code. The ASTs are converted into a set of triples which are later fed into an attention-based neural network for the task of method name prediction.

Except for the direct use of ASTs, recently, researchers propose to adopt more sophisticated structural information from source code. For example, Tufano et al. (2018) try to combine different code representations on detecting code clones. In their work, they consider four different training contexts i.e., identifiers, AST, bytecode, and control flow graph (CFG) extracted from the source code fragments. Allamanis et al. (2018) extract the data flows from the source code and then use a Gated Graph Neural Networks to learn the program representation. However, the data flows they extracted are based on the AST representation of the source code. In other words, they explicitly expose part of the AST of a program (i.e., the subtree that contains syntax tokens corresponding to declarations and updates of variables) as the structured input to the embedding learning model. This might be useful for downstream tasks that are sensitive to the data operations of a program, for example, the task used in their work, variable misuse detection. Although by doing this, they can focus on the utilization of the data flows, there is still a chance of missing some important information of the program, at least, one cannot build a program only based on the data flow graph. Instead, in our work, we attempt to produce generalizable code embeddings that can

be used for different downstream tasks, thus, we utilize more types of information, including the AST information within each method, method calls and variable references, which may introduce some noise for code embedding learning as, in our work, the use of method calls and variable references are all based on the simple string match.

To further explore the structural information of the source code for producing generalizable code embeddings, Sui et al. (2020) utilize control-flows and data flows of a program. They extend the structural information by (1) mining long-range data flows across different methods, (2) precisely extracting the data flow information based on the pointer alias information. Compared to our approach, they avoid the noise and mistakes of the simple use of string match for method calls and variable references. Besides, the long-range and precise data flows make the generated code embeddings be able to capture the method or data dependence of the program, which is really useful for the tasks that involves the interaction between different methods, for example, in the task of code summarization: if the target method calls another method in the method body, it would be useful to utilize the long-range data flows to analyze what really happens between these two methods. However, this kind of global information may not bring many benefits for other tasks, such as the task of code authorship identification, as the programs (i.e., input of the task) are isolated from each other and written by different authors, where the method level approach (e.g., code2vec) may perform better.

Our work builds upon the recent advances of code embeddings learning approaches and is closely related to the work of Kang et al. (2019) which trains and evaluates two code embedding techniques, i.e., GloVe and code2vec on three downstream SE tasks. Kang et al. (2019) evaluate GloVe and code2vec on the tasks of code comment generation, code authorship identification and code clone detection. Our work extends theirs and revisits the use of pre-trained code embeddings in downstream tasks. More specifically, we evaluate five more embeddings, i.e., Word2vec, fastText, StrucTexVec, CodeBERT and CuBERT on six downstream SE tasks.

8 Threats to Validity

This section discusses the threats to the validity of our work. We consider three types of threats.

External Validity One major threat of using pre-trained code embeddings in downstream tasks is the computational costs of training the embeddings. In our work, the embeddings of StrucTexVec, GloVe, fastText and Word2vec are all trained in CPUs and code2vec is trained in an NVIDIA GTX 1080 Ti GPU, and it takes less than 30 min to finish the training process of each of the embedding techniques, which is acceptable compared to the computational cost of the running the downstream tasks (e.g., it takes more than 10 h to finish the task of code comment generation). However, for the training of CodeBERT and CuBERT, it not only requires more GPUs but also several days to finish the training. For example, CodeBERT spends more than ten days to finish the training using 16 interconnected NVIDIA Tesla V100 GPUs, which might be challenging for us to train the embeddings in our own machine. The findings in this work are concluded based on the evaluation results of seven code embedding techniques on six downstream tasks, and the code embeddings are trained on Java projects. Thus, we cannot confirm that our findings may generalize to all the SE tasks, programming languages and code embedding techniques. Besides, in this work, we adopt external SE tasks to reflect the impact of different code embedding techniques, and

each task has its specific model to train, and thus the parameters involved during the embeddings evaluation may have an impact on the conclusions. To minimize the influence of these parameters, in our work, we intentionally do not do any other parameter tuning on the model structures (e.g., layers, hidden dimensions) and 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. Another factor that can influence the results and conclusions is the evaluation metrics used in the downstream tasks, as code embedding techniques that perform better under one evaluation metric may perform worse when evaluated using other metrics. In this work, to reduce the selection bias, we try to follow previous works and select the evaluation metrics that are used in the existing papers related to the downstream tasks. In addition, we provide our data and source code for future work to replicate and further improve the evaluation. For the only change of batch size in the task of code authorship identification, we want to explain that we initially set the batch size to 128 and observed that the precision remains at around 50% even on the training dataset, which was far behind the expected performance of deep learning models on the simple classification tasks. Thus, we reduce the batch size, and after that, the training precision improves to more than 90%. Also, this improvement confirms the finding from previous work that using small batch sizes achieves better training stability (Masters and Luschi 2018). Moreover, the datasets used for different downstream tasks are unbalanced which would have an impact on the conclusions if we compare the results on the dataset level. As described in Section 4, all the datasets are provided by previous work and are commonly used benchmark datasets. To avoid such influence, we only focus on the task level comparison. Besides, there is a lack of ways for direct evaluation of the quality of code embeddings. Future studies can develop some datasets or tasks, such as token similarity, that can be directly used for code embeddings evaluation.

Internal Validity One of the threats to the internal validity is related to the conclusion of results in response to RQ2. In order to answer RQ2, we conducted ablation experiments to analyze the impact of structural and textual context. However, the analysis in RQ2 may not indicate the actual impact of structural and textual information, as they may have overlappings. For example, in the AST representation of the source code, the code tokens are also considered during the embedding training. Besides, in our experiments, we only evaluate the embeddings that are trained based on the AST or plain text of the source code. However, there exists other information from the source code that may be more valuable for representing the properties of the source code. For example, Allamanis et al. (2018) exploit the use of data flows in a program and Sui et al. (2020) further extend the use of structured information extracted from the source code and they adopt the interprocedural program dependence for representing the source code into vectors. Another threat comes from the fact that there is a lack of interpretability of the code embeddings. The embeddings are real-valued numbers and hard to be analyzed directly, and thus, our findings cannot explain when and why the embeddings are effective.

Construct Validity The embedding training dataset selection may be biased, as we only select the top-ranked Java projects based on the number of stars they have, and we may still miss some Java projects that are from unpopular fields. Future studies can collect projects across different fields to complement the findings of our study. Another threat is the parameters of the code embedding techniques during embedding learning, such as the embedding dimensions and negative samples. Although, by fine-tuning the parameters, we can make the model better conform to the downstream tasks. However, in our work, we try our best

to follow the literature and intentionally do not fine-tune the parameters to avoid bias from the unfairness among the tasks. We leave it as future work to analyze the impact of different combinations of parameter settings on the quality of the code embeddings.

9 Conclusion

In this paper, we revisit and extend a recent study by Kang et al. (2019) on the assessment of pre-trained code embeddings in SE tasks. Complementing the two evaluated pre-trained embedding techniques in prior work, we propose an unsupervised framework, StrucTexVec, for enhancing the learned code embeddings by incorporating both the textual and structural knowledge into the embedding training process. In total, we evaluate the effectiveness of seven techniques for pre-trained code embeddings on six downstream SE tasks. On one hand, we find that, in general, models using pre-trained embeddings can perform better than the models without pre-trained embeddings, and both the structural information and the textual information have a non-negligible impact on the performance of the downstream SE tasks. On the other hand, our work concurs with prior research that pre-trained embeddings may not always improve the performance of downstream SE tasks significantly, and different embedding techniques can lead to diverse results. Our results suggest the need for researchers and practitioners to carefully consider the choices of embedding techniques when conducting SE tasks. Our findings also shed light on future research for improving embedding techniques to assist in SE tasks.

Declarations

Conflict of Interest The authors declare no competing interests.

References

Abuhamad M, AbuHmed T, Mohaisen A, Nyang D (2018) Large-scale and language-oblivious code authorship identification. In: Lie D, Mannan M, Backes M, Wang X (eds) Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, CCS 2018, Toronto, ON, Canada, October 15–19, 2018. ACM, pp 101–114. https://doi.org/10.1145/3243734.3243738

Allamanis M, Barr ET, Bird C, Sutton CA (2014) Learning natural coding conventions. In: Cheung S, Orso A, Storey MD (eds) Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, (FSE-22), Hong Kong, China, November 16–22, 2014. ACM, pp 281–293. https://doi.org/10.1145/2635868.2635883

Allamanis M, Barr ET, Bird C, Sutton CA (2015) Suggesting accurate method and class names. In: Nitto ED, Harman M, Heymans P (eds) Proceedings of the 2015 10th joint meeting on foundations of software engineering, ESEC/FSE 2015, Bergamo, Italy, August 30–September 4, 2015. ACM, pp 38–49. https://doi.org/10.1145/2786805.2786849

Allamanis M, Peng H, Sutton CA (2016) A convolutional attention network for extreme summarization of source code. In: Balcan M, Weinberger KQ (eds) Proceedings of the 33nd international conference on machine learning, ICML 2016, New York City, NY, USA, June 19–24, 2016, JMLR.org, JMLR Workshop and Conference Proceedings, vol 48, pp 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

Allamanis M, Brockschmidt M, Khademi M (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-

Alon U, Zilberstein M, Levy O, Yahav E (2018) A general path-based representation for predicting program properties. In: Foster JS, Grossman D (eds) Proceedings of the 39th ACM SIGPLAN conference on

programming language design and implementation, PLDI 2018, Philadelphia, PA, USA, June 18–22, 2018. ACM, pp 404–419. https://doi.org/10.1145/3192366.3192412

Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. PACMPL 3(POPL):40:1–40:29. https://doi.org/10.1145/3290353

Barbour L, Khomh F, Zou Y (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, pp 273–282. https://doi.org/10.1109/ICSM.2011.6080794

Bielik P, Raychev V, Vechev MT (2016) PHOG: probabilistic model for code. In: Balcan M, Weinberger KQ (eds) Proceedings of the 33nd international conference on machine learning, ICML 2016, New York City, NY, USA, June 19–24, 2016, JMLR.org, JMLR Workshop and conference proceedings, vol 48, pp 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. Trans Assoc Comput Linguis 5:135–146. https://transacl.org/ojs/index.php/tacl/article/view/999

Büch L, Andrzejak A (2019) Learning-based recursive aggregation of abstract syntax trees for code clone detection. In: Wang X, Lo D, Shihab E (eds) 26th IEEE international conference on software analysis, evolution and reengineering, SANER 2019, Hangzhou, China, February 24–27, 2019. IEEE, pp 95–104. https://doi.org/10.1109/SANER.2019.8668039

Chen Z, Monperrus M (2018) The remarkable role of similarity in redundancy-based program repair. arXiv:1811.05703

Chen T, Thomas SW, Hassan AE (2016) A survey on the use of topic models when mining software repositories. Empir Softw Eng 21(5):1843–1919. https://doi.org/10.1007/s10664-015-9402-8

Collard ML, Decker MJ, Maletic JI (2011) Lightweight transformation and fact extraction with the srcml toolkit. In: 11th IEEE working conference on source code analysis and manipulation, SCAM 2011, Williamsburg, VA, USA, September 25–26, 2011. IEEE Computer Society, pp 173–184. https://doi.org/10.1109/SCAM.2011.19

Devlin J, Chang M, Lee K, Toutanova K (2019) BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein J, Doran C, Solorio T (eds) Proceedings of the 2019 conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2–7, 2019, vol 1 (Long and Short Papers). Association for Computational Linguistics, pp 4171–4186. https://doi.org/10.18653/v1/n19-1423

Ding Z, Li H, Shang W (2022) Logentext: automatically generating logging texts using neural machine translation. In: SANER. IEEE

Dubinsky Y, Rubin J, Berger T, Duszynski S, Becker M, Czarnecki K (2013) An exploratory study of cloning in industrial software product lines. In: Cleve A, Ricca F, Cerioli M (eds) 17th European conference on software maintenance and reengineering, CSMR 2013, Genova, Italy, March 5–8, 2013. IEEE Computer Society, pp 25–34. https://doi.org/10.1109/CSMR.2013.13

Efstathiou V, Spinellis D (2019) Semantic source code models using identifier embeddings. In: Storey MD, Adams B, Haiduc S (eds) Proceedings of the 16th international conference on mining software repositories, MSR 2019, 26–27 May 2019. IEEE/ACM, Montreal, pp 29–33. https://doi.org/10.1109/MSR.2019.00015

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) Codebert: a pre-trained model for programming and natural languages. In: Cohn T, He Y, Liu Y (eds) Findings of the association for computational linguistics: EMNLP 2020, online event, 16–20 november 2020. Association for Computational Linguistics, findings of ACL, vol EMNLP 2020, pp 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Gilda S (2017) Source code classification using neural networks. IEEE, pp 1–6. https://doi.org/10.1109/JCSSE.2017.8025917

Harer JA, Kim LY, Russell RL, Ozdemir O, Kosta LR, Rangamani A, Hamilton LH, Centeno GI, Key JR, Ellingwood PM, McConley MW, Opper JM, Chin SP, Lazovich T (2018) Automated software vulnerability detection with machine learning. arXiv:1803.04497

Hindle A, Barr ET, Su Z, Gabel M, Devanbu PT (2012) On the naturalness of software. In: Glinz M, Murphy GC, Pezzè M (eds) 34th International conference on software engineering, ICSE 2012, June 2–9, 2012. IEEE Computer Society, Zurich, pp 837–847. https://doi.org/10.1109/ICSE.2012.6227135

Hu X, Li G, Xia X, Lo D, Jin Z (2018) Deep code comment generation. In: Khomh F, Roy CK, Siegmund J (eds) Proceedings of the 26th conference on program comprehension, ICPC 2018, Gothenburg, Sweden, May 27–28, 2018. ACM, pp 200–210. https://doi.org/10.1145/3196321.3196334

Islam AC, Harang RE, Liu A, Narayanan A, Voss CR, Yamaguchi F, Greenstadt R (2015) De-anonymizing programmers via code stylometry. In: Jung J, Holz T (eds) 24th USENIX security symposium, USENIX security 15, Washington, D.C., USA, August 12–14, 2015. USENIX Association, pp 255–270. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/caliskan-islam

Kamiya T, Kusumoto S, Inoue K (2002) Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670. https://doi.org/10.1109/TSE.2002.1019480

Kanade A, Maniatis P, Balakrishnan G, Shi K (2020) Learning and evaluating contextual embedding of source code. In: Proceedings of the 37th international conference on machine learning, ICML 2020, 13–18 July 2020, Virtual Event, PMLR, Proceedings of Machine Learning Research, vol 119, pp 5110–5121. http://proceedings.mlr.press/v119/kanade20a.html

Kang HJ, Bissyandé TF, Lo D (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, pp 1–12. https://doi.org/10.1109/ASE.2019.00011

Kawaguchi S, Garg PK, Matsushita M, Inoue K (2006) Mudablue: an automatic categorization system for open source repositories. J Syst Softw 79(7):939–953. https://doi.org/10.1016/j.jss.2005.06.044

Kim Y (2014) Convolutional neural networks for sentence classification. arXiv:14085882

Komninos A, Manandhar S (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, pp 1490–1500

Li C, Li J, Song Y, Lin Z (2018a) Training and evaluating improved dependency-based word embeddings. In: McIlraith SA, Weinberger KQ (eds) 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. AAAI Press, pp 5836–5843. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16429

Li H, Chen TP, Shang W, Hassan AE (2018b) Studying software logging using topic models. Empir Softw Eng 23(5):2655–2694. https://doi.org/10.1007/s10664-018-9595-8

Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019) Roberta: a robustly optimized BERT pretraining approach. arXiv:1907.11692

Masters D, Luschi C (2018) Revisiting small batch training for deep neural networks. arXiv:1804.07612

Mayrand J, Leblanc C, Merlo E (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, p 244. https://doi.org/10.1109/ICSM.1996.565012

McBurney PW, McMillan C (2014) Automatic documentation generation via source code summarization of method context. In: Roy CK, Begel A, Moonen L (eds) 22nd International conference on program comprehension, ICPC 2014, Hyderabad, India, June 2–3, 2014. ACM, pp 279–290. https://doi.org/10.1145/2597008.2597149

Mikolov T, Chen K, Corrado G, Dean J (2013a) Efficient estimation of word representations in vector space. In: Bengio Y, Lecun Y (eds) 1st International conference on learning representations, ICLR 2013, Scottsdale, Arizona, USA, May 2–4, 2013. Workshop Track Proceedings. arXiv:1301.3781

Mikolov T, Sutskever I, Chen K, Corrado G, Dean J (2013b) Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th international conference on neural information processing systems. NIPS'13, vol 2. Curran Associates Inc., pp 3111–3119. http://dl.acm.org/citation.cfm?id=2999792.2999959

Moreno L, Aponte J, Sridhara G, Marcus A, Pollock LL, Vijay-shanker K (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, pp 23–32. https://doi.org/10.1109/ICPC.2013.6613830

Mou L, Li G, Zhang L, Wang T, Jin Z (2016) Convolutional neural networks over tree structures for programming language processing. In: Schuurmans D, Wellman MP (eds) Proceedings of the thirtieth AAAI conference on artificial intelligence, February 12–17, 2016. AAAI Press, Phoenix, pp 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

Papineni K, Roukos S, Ward T, Zhu W (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. ACL, Philadelphia, pp 311–318. https://www.aclweb.org/anthology/P02-1040/

Pennington J, Socher R, Manning CD (2014) Glove: global vectors for word representation. In: Moschitti A, Pang B, Daelemans W (eds) 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. ACL, pp 1532–1543. https://www.aclweb.org/anthology/D14-1162/

Pradel M, Sen K (2018) Deepbugs: a learning approach to name-based bug detection. PACMPL 2(OOPSLA):147:1–147:25. https://doi.org/10.1145/3276517

Raychev V, Vechev MT, Krause A (2015) Predicting program properties from "big code". In: Rajamani SK, Walker D (eds) Proceedings of the 42nd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2015, Mumbai, India, January 15–17, 2015. ACM, pp 111–124. https://doi.org/10.1145/2676726.2677009

Řehůřek R, Sojka P (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, pp 45–50. http://is.muni.cz/publication/884893/en

Roy CK, Cordy JR (2007) A survey on software clone detection research. Queen's School of Computing TR 541(115):64–68. https://doi.org/10.1.1.62.7869

Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV (2016) Sourcerercc: scaling code clone detection to big-code. In: Dillon LK, Visser W, Williams L (eds) Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016. ACM, pp 1157–1168. https://doi.org/10.1145/2884781.2884877

Sridhara G, Hill E, Muppaneni D, Pollock LL, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Pecheur C, Andrews J, Nitto ED (eds) ASE 2010, 25th IEEE/ACM international conference on automated software engineering, Antwerp, Belgium, September 20–24, 2010. ACM, pp 43–52. https://doi.org/10.1145/1858996.1859006

Sui Y, Cheng X, Zhang G, Wang H (2020) Flow2vec: value-flow-based precise code embedding. CoRR 4(OOPSLA):233:1–233:27. https://doi.org/10.1145/3428301

Svajlenko J, Islam JF, Keivanloo I, Roy CK, Mia MM (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, pp 476–480. https://doi.org/10.1109/ICSME.2014.77

Theeten B, Vandeputte F, Cutsem TV (2019) Import2vec learning embeddings for software libraries. In: Storey MD, Adams B, Haiduc S (eds) Proceedings of the 16th international conference on mining software repositories, MSR 2019, 26–27 May 2019. IEEE/ACM, Montreal, pp 18–28. https://doi.org/10.1109/MSR.2019.00014

Thummalapenta S, Cerulo L, Aversano L, Penta MD (2010) An empirical study on the maintenance of source code clones. Empir Softw Eng 15(1):1–34. https://doi.org/10.1007/s10664-009-9108-x

Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D (2018) Deep learning similarities from different representations of source code. In: Zaidman A, Kamei Y, Hill E (eds) Proceedings of the 15th international conference on mining software repositories, MSR 2018, Gothenburg, Sweden, May 28–29, 2018. ACM, pp 542–553. https://doi.org/10.1145/3196398.3196431

Vashishth S, Bhandari M, Yadav P, Rai P, Bhattacharyya C, Talukdar PP (2019) Incorporating syntactic and semantic information in word embeddings using graph convolutional networks. In: Korhonen A, Traum DR, Màrquez L (eds) Proceedings of the 57th conference of the association for computational linguistics, ACL 2019, Florence, Italy, July 28–August 2, 2019, vol 1: Long Papers, Association for Computational Linguistics, pp 3308–3318. https://doi.org/10.18653/v1/p19-1320

Vásquez ML, McMillan C, Poshyvanyk D, Grechanik M (2014) On using machine learning to automatically classify software applications into domain categories. Empir Softw Eng 19(3):582–618. https://doi.org/10.1007/s10664-012-9230-z

Wang S, Liu T, Tan L (2016) Automatically learning semantic features for defect prediction. In: Dillon LK, Visser W, Williams L (eds) Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016. ACM, pp 297–308. https://doi.org/10.1145/2884781.2884804

Wang K, Singh R, Su Z (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

Wei H, Li M (2017) Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Sierra C (ed) Proceedings of the twenty-sixth international joint conference on artificial intelligence, IJCAI 2017, Melbourne, Australia, August 19–25, 2017, ijcai.org, pp 3034–3040. https://doi.org/10.24963/ijcai.2017/423

White M, Tufano M, Vendome C, Poshyvanyk D (2016) Deep learning code fragments for code clone detection. In: Lo D, Apel S, Khurshid S (eds) Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016, Singapore, September 3–7, 2016. ACM, pp 87–98. https://doi.org/10.1145/2970276.2970326

White M, Tufano M, Martinez M, Monperrus M, Poshyvanyk D (2019) Sorting and transforming program repair ingredients via deep learning code similarities. In: Wang X, Lo D, Shihab E (eds) 26th IEEE international conference on software analysis, evolution and reengineering, SANER 2019, Hangzhou, China, February 24–27, 2019. IEEE, pp 479–490. https://doi.org/10.1109/SANER.2019.8668043

Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A novel neural source code representation based on abstract syntax tree. In: Atlee JM, Bultan T, Whittle J (eds) Proceedings of the 41st international conference on software engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019. IEEE/ACM, pp 783–794. https://doi.org/10.1109/ICSE.2019.00086

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Heng Li
heng.li@polymtl.ca

Weiyi Shang
shang@encs.concordia.ca

Tse-Hsun (Peter) Chen
peterc@encs.concordia.ca

1 Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada

2 Department of Computer Engineering and Software Engineering, Polytechnique Montreal, Montreal, QC, Canada