Assessing the Generalizability of code2vec Token Embeddings

Hong Jin Kang
Singapore Management University
hjkang.2018@phdis.smu.edu.sg

Tegawendé F. Bissyandé
University of Luxembourg, Luxembourg
tegawende.bissyande@uni.lu

David Lo
Singapore Management University
davidlo@smu.edu.sg

***Abstract*—Many Natural Language Processing (NLP) tasks, such as sentiment analysis or syntactic parsing, have benefited from the development of word embedding models. In particular, regardless of the training algorithms, the learned embeddings have often been shown to be generalizable to different NLP tasks. In contrast, despite recent momentum on word embeddings for source code, the literature lacks evidence of their generalizability beyond the example task they have been trained for.**

In this experience paper, we identify 3 potential downstream tasks, namely code comments generation, code authorship identification, and code clones detection, that source code token embedding models can be applied to. We empirically assess a recently proposed code token embedding model, namely *code2vec*'s token embeddings. Code2vec was trained on the task of predicting method names, and while there is potential for using the vectors it learns on other tasks, it has not been explored in literature. Therefore, we fill this gap by focusing on its generalizability for the tasks we have identified. Eventually, we show that source code token embeddings cannot be readily leveraged for the downstream tasks. Our experiments even show that our attempts to use them do not result in any improvements over less sophisticated methods. We call for more research into effective and general use of code embeddings.

***Index Terms*—Code Embeddings, Distributed Representations, Big Code**

I. Introduction

In recent years, there have been many works [1], [2] proposing new techniques to construct representations of text. The representation of a word by "embedding" it onto a vector space is known as "word embeddings". Embedding techniques, such as word2vec [1], output vectors of real numbers which are successfully leveraged in a variety of Natural Language Processing (NLP) tasks. Indeed, embedding models attempt to represent words that are semantically similar as vectors which are close in the vector space.

While the NLP literature provides several methods to train word embeddings, a number of applications have successfully demonstrated that most word embedding models generalize to various NLP tasks, beyond the ones the embeddings have been initially trained on. For example, word embeddings, which are often trained over predicting the next word in a sentence given its previous words, have been used for many downstream tasks such as sentiment analysis, word sense disambiguation, named entity recognition, and part-of-speech tagging [1], [3]–[6].

Besides embeddings for natural language words, various embedding models have been proposed for object representations in other domains, including graphs, knowledge bases, or even source code. For example, in recent years, inspired by the advances in word embeddings, many researchers have used embedding models for software engineering tasks. On the one hand, the *naturalness hypothesis* [7], which suggests that, like natural language, software is also likely to be statistically repetitive and predictable, has encouraged the use of natural language processing techniques on source code. The increase of publicly-available source code, such as open-source projects data on Github, facilitates research on big code [8] analysis. In this context, several code embeddings approaches [9]–[11] have been proposed where source code tokens are treated similarly to text.

Apart from the direct application of word embedding techniques to source code, other researchers have proposed specialized methods for source code. For example, there have been attempts to include more structural information of the program. Among recent advances, *code2vec* [12], which stands as a state-of-the-art, traverses paths in the Abstract Syntax Tree (AST) to train embeddings for predicting method names. The authors' study shows that accounting for such structural information improves the performance in method name predictions.

Many research directions in the software engineering domain have proposed new methods of training code embeddings. However, they are often only evaluated on the single task that they were trained for [13]–[19]. In contrast to the plentiful evidence that word embeddings are useful in downstream tasks, there is little evidence to suggest that the code embeddings can be useful in a variety of software engineering tasks. Given the promise of code embeddings, our work is motivated by this need to fill the gap in confirming the generalizability of code embedding models. Treating code2vec as representative of code embeddings, our study investigates whether it can be successfully used in a variety of software engineering tasks beyond predicting method names.

In this paper, we assess the token embeddings proposed by code2vec in terms of providing appropriate representations of source code in several different software engineering downstream tasks, i.e. tasks in which the learned code embeddings may be helpful on. These tasks are not directly related to its original training task of predicting method names. Concretely, we propose to use source code token embeddings to enhance existing models for tasks of code comment generation, code authorship identification and code clones detection. Then, we

evaluate the performance of these models against baseline models [20]–[22] originally designed for the considered tasks. We make a replication package of our work available.[1] This paper makes the following contributions:

- We investigate the generalizability of tokens embeddings learned by code2vec for downstream software engineering tasks. To the best of our knowledge, this is the first effort in this direction in the literature.
- We experimentally demonstrate that code2vec's token embeddings may not be generalizable: they do not always contribute to a significant increase in the performance of models for each of the software engineering tasks.
- We provide a comprehensive discussion on the generalizability of code embeddings, and motivate the need for further research on novel embedding models for source code that can generalize to various downstream tasks.

The remainder of this paper is organized as follows. In Section II, we provide details for the context of this study. In Section III, we introduce the three software engineering downstream tasks that we have identified for evaluating code embeddings, evaluate code2vec on the 3 tasks and discuss the results. In Section IV, we discuss the threats to validity of our work. In Section V, we present related work. Finally, in Section VI, we conclude the paper with a call for further research.

II. Preliminaries

Our study aims to answer a single research question: *Are embeddings of source code tokens generalizable for use in tasks that they are not trained for?*

The main components of our study are as follows:

- **Code embedding model.** Structural information from ASTs has been widely leveraged for building models in the literature [23]–[27]. In particular, it has been used to train representations of source code for predicting method names, a common task in software engineering literature [8], [28]–[30]. As *code2vec* [12] is located in the intersection of these two trends in code representation research, we select it as a representative state-of-the-art among embedding models. Code2vec embeds entire code snippets into a single vector during training. However, our work focuses on the token embeddings that result from its training due to its similar granularity to word tokens used in word embeddings and its broader vocabulary, which we believe allows it to be more generalizable.
- **Word embedding model.** Aside from code2vec being specialized for source code tokens, we also consider generic NLP word embeddings to build a comparison baseline. To that end, we consider *GloVe* [2] as a representative word embedding technique, which we train on a similar dataset as code2vec by considering source code as natural language text.
- **Downstream tasks.** We identify three downstream tasks of code comment generation, code authorship identification and code clones detection. For each task, we select an existing approach that uses source code tokens as part of its input. We select models from recent work or use well-known models. For code comment generation and code authorship identification, we use recent work by Hu et al. [20] and Abuhamad et al. [21] respectively. For code clone detection, we use the well-documented baseline model, SourcererCC [22]. We try to augment these models using embeddings of source code tokens. Finally, we compare the performance of the models using embeddings against simpler models as a baseline. In the same vein as the suggestions provided by Fu and Menzies [31] in their comparative study of machines learning approaches in software engineering, the first baseline we use is the simpler basic model without the use of code embeddings. Secondly, we compare our code2vec-augmented model against a model that is augmented in the same way, but this time using the GloVe vectors that are trained over source code tokens.

[1] https://github.com/code2vec-critique/generalizability

A. Code2Vec

The code2vec deep representation learning technique was initially proven effective through a demonstration of training code embeddings for the following prediction task: a code snippet was given as input, and a tag was predicted as output. Code2vec was mainly used for method name prediction where each input is a method body and the method name is used as the associated output tag. Structural information from the AST, notably paths between AST terminals, is extracted and leveraged during training: each code snippet is represented by a bag of path contexts. An attention mechanism is used to learn the importance of each path context to the output tag. A single path context comprises a tuple of the 2 terminal nodes, and the path between them. From the path contexts, a neural-network model is trained to predict the code snippet's method name. The following example of a path context for the expression, $x = 7$, is given by Alon et al. [12]:

$$\langle x, (\ \text{NameExpr}\ \uparrow\ \text{AssignExpr}\ \downarrow\ \text{IntegerLiteralExpr}\ ), 7 \rangle$$

Code2vec produces 2 sets of vectors. From its output layer, a set of vectors for method names can be exported. From its input layer, a set of vectors for token/identifiers can be exported. Our work focuses on the token vectors due to its more precise granularity, which we believe will make it more generalizable and applicable to other tasks.

For our study, we use code2vec token vectors exported from the trainable model downloaded from the code2vec repository. These vectors are 128 dimensions wide, with a vocabulary size of 1 300 852 words.

B. GloVe

To evaluate code2vec, we need a baseline set of vectors that is simpler and easier to train. To this end, we treat source

code as text and we use the GloVe algorithm [2], that is used to learn word embeddings, to train code embeddings. Due to the naturalness hypothesis, embedding models designed for natural language may be effective when directly applied to source code. Unlike many word embedding algorithms, GloVe is an unsupervised algorithm using token-token co-occurrence statistics.

We trained GloVe vectors that are 128 dimensions, to have the same dimensionality as the code2vec token embeddings. We adjusted the parameters of GLoVe to have a similar number of tokens in its vocabulary as code2vec. Thus, we set the minimum number of occurrences to be 70. Other parameters of GloVe are set to the default values; for example, we trained GloVe for 15 iterations, with a window size of 15.

The dataset used to trained GloVe is the *Java-Large* dataset provided by the authors of code2vec.[2] It includes the 9500 most-starred Github Java projects, consisting of about 16 million methods, and amounts to 37GB of data. Eventually, the GloVe vectors are 128 dimensions wide, with a vocabulary size of 1 335 292 words. The original dataset used to trained code2vec is only available in a processed form, and is not appropriate for training GloVe embeddings. However, the Java-Large dataset is comparable to the original dataset in terms of size and its source. As such, we do not expect this difference to be a threat to validity.

III. Evaluation on Downstream Tasks

We consider in this study three downstream software engineering tasks targeting different properties of source code that may be encoded in embeddings: (1) code comment generation, (2) code authorship identification and (3) code clones detection. We now briefly describe the objective of each task, the dataset that we use as well as the evaluation procedure.

To evaluate code embeddings, we try to augment existing models with the code embeddings and observe if the resulting model leads to an improved performance. For each task, we select a model from recent papers. Whenever it is not obvious how to augment an existing model with embeddings, we propose a new approach to incorporate embeddings into it. We focus on using techniques that are token-based to make it simple to augment the model with code embeddings. The techniques we use in these tasks also vary. Two out of three tasks use a neural-network-based approach, while the last task utilizes vector space calculations to compare the similarity of two vectors. In this study, we do not focus on the overall effectiveness of the techniques. Instead, we evaluate if the use of code embeddings can improve the performance of these techniques. For each task, we select datasets and use experiment settings similar to what was reported in literature.

A. Code Comment Generation

Our first task is code comment generation. As we focus our work at the granularity of methods, this task involves the automatic generation of method-level comment from the body of a method [11], [32], [33]. The generated method comment should summarize the functionality provided by the method in the form of a descriptive, high-level, natural language sentence. The task has implications for software maintenance and program comprehension. Techniques developed for this task can produce a wide range of benefits for developers, including helping in software reuse, re-documentation and concept location [33]. Several recent works have used neural networks to synthesize natural language from source code [11], [20].

Approach: For this task, there have been several techniques using a deep learning approach. We use the latest approach proposed by Hu et al. [20]. In their approach, they treated the problem as a machine translation task. Their approach incorporates and retains structural information from the AST when preprocessing the data from code snippets representing method bodies into sequences of tokens representing the AST nodes. A Recurrent Neural Network-based Seq2Seq language model is used to translate these sequences to natural language code comments. We selected this approach since it uses a neural network approach and uses an embedding layer where our code embeddings can be used. In addition, their model gave state-of-the-art results. Thus, we follow the approach described by Hu et al.

Similar to the preprocessing done by Hu et al., we took only the first sentence of the Javadoc method comment, as this first sentence is usually the description of the functionality provided by a method based on Javadoc convention. Like Hu et al., we filtered out simple cases from the dataset. We omitted a pair of code snippet and comment if the comment is empty or just contain a single word, Additionally, getters, setters, constructors, and test methods are omitted.

We show a sample input for this task, taken from Table 5 in the work by Hu et al. in Listing 1. For this example, the ground truth output will be the first (in this case, the only) sentence of the Javadoc comment, summarizing the method.

Listing 1. Example of a code snippet

```
/**
 * Convert Bitmap to byte array.
 */
public static byte[] bitmapToByte(Bitmap b){
  ByteArrayOutputStream o = new
      ByteArrayOutputStream();
  b.compress(Bitmap.CompressFormat.PNG,100,o
      );
  return o.toByteArray();
}
```

The metric BLEU [34] is used to measure the quality of generated comments. This is commonly used to evaluate the performance of machine translation of natural language, and measures how closely the translation is to a human translation. BLEU takes the generated translation and reference translations as input and outputs a percentage value between 0 and 100, with scores closer to 100 indicating higher quality. It is as computed as follows.

[2]https://s3.amazonaws.com/code2seq/datasets/java-large.tar.gz

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} \frac{1}{N} \log\left(p_n\right)\right)$$
$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

p_n is the ratio of length n subsequences that are both in the candidate and reference translation. N is the maximum number of N-grams. BP is the brevity penalty. c is the length of the candidate translation generated while r is the length of the reference translation.

The BLEU metric has been shown to correlate well with human perception of the quality of translations [35]. In software engineering literature, this has been used for the evaluation of several tasks [15], [36]–[38], including comment generation. There are a few variants of BLEU, depending on the value of N. In this task, we use BLEU-4 (i.e. N=4) since it was used by Hu et al. to evaluate the quality of the generated comment. BLEU-4 is a measure of precision of 4-grams.

To accurately summarize a method using high-level natural language, a technique will need to infer semantic properties of the source code. Thus, this task may be sensitive to the ability of code embeddings to encode semantic information.

Preprocessing: We used the dataset that was collected by Hu et al., which involves 9714 Java projects from Github. Then, we followed the same procedure as Hu et al. to convert the AST of each method body into a sequence of tokens. A high-level overview of the procedure to preprocess the dataset is given in Figure 1.

The Eclipse JDT parser was used to construct the AST, and we traversed the tree following the Structure-Based Traversal (SBT) algorithm described by Hu et al. This preserves information about the tree structure of the AST in the sequence, and allows the reconstruction of the AST tree from the sequence of tokens. Tokens in the sequence consist of the AST node type and the value of the node (either a literal or identifier name). For example, for a local variable `x`, its representation in the sequence will be `SimpleName_x`. For a method invocation of a method `toLowerCase`, this will be represented as follows:

```
( MethodInvocation
  ( SimpleName_toLowerCase )
    SimpleName_toLowerCase
) MethodInvocation
```

In machine translation for natural language, the vocabulary is often restricted to the most common words. Words out of the vocabulary are marked as `<UNK>`. Similarly, we limited the vocabulary of both code tokens and comments to the 30000 most common tokens. For code tokens outside of the vocabulary, we convert them to to the AST node type. Concretely, this means that rare identifier names will not be represented in the dataset. For example, a variable `veryRareAndLongName` will be converted into the token `SimpleName`, while a common variable name such as `x` will be converted as `SimpleName_x`. The rationale for doing this was given by Hu et al.: this representation of out-of-vocabulary tokens mitigate the problem caused by the fact that the vocabulary of source code tokens is much greater than natural language. Rather than losing all information when we remove rare words from the dataset, some structural information is retained.

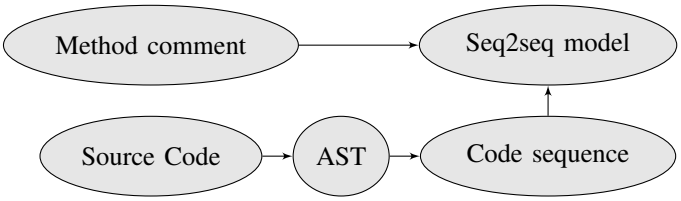

Fig. 1. Preprocessing data for comment generation

We preprocessed the vocabulary in the embeddings similarly, by prefixing AST node types to each word. Next, for any word found in the training data that is not contained in the code embeddings, we expanded the vocabulary of the embeddings to include it and initialize it with a random vector. We found that there are less than 200 such tokens in the dataset. This indicates that the step of normalizing rare identifiers into their AST node types is effective in minimizing the number of out-of-vocabulary tokens seen during training of the machine translator.

For the example about `bitmapToByte` in Listing 1, the following sequence will be generated for the statement `return o.toByteArray();`. We do not show the sequence for the entire method due to space constraints.

```
( ReturnStatement
  ( MethodInvocation
    ( SimpleName_o ) SimpleName_o
    ( SimpleName_toByteArray )
    SimpleName_toByteArray
  ) MethodInvocation
) ReturnStatement
```

Training: Next, we trained a Recurrent Neural Network-based Seq2Seq language model using OpenNMT [39]. The model consists of an encoder-decoder network. On both encoder and decoder, we use a 2 Long Short-Term Memory (LSTM) [40] layers with 500 hidden units in each layer. We set the learning rate to 0.5, the dropout to 0.5, and we train it for 50 epochs. In total, there are over 330,000 methods in the training data, and we limit the validation and test data to 5000 methods.

The model consists of an embedding layer. When the code2vec and GloVe embeddings are not used, randomly initialized vectors are used instead, similar to the work in Hu et al. There are 2 settings that we use for the experiments on this task. As the tokens from code2vec are all lower-cased, we lower-cased the identifiers from the AST trees we extracted and also created a version of the GloVe vectors with its tokens lowercased. For a more comprehensive evaluation, we trained a new set of code2vec embeddings on the Java-Large dataset described above. In this set of embeddings, the words are not lowercased.

TABLE I
QUALITY OF COMMENTS GENERATED, WITH SBT PREPROCESSING

Preprocessing	Embedding model	BLEU-4
Lowercased	GloVe	27.4
Lowercased	code2vec	29.9
Lowercased	No pretrained embeddings	28.1
Non-lowercased	GloVe	28.1
Non-lowercased	code2vec	29.3
Non-lowercased	**No pretrained embeddings**	**33.5**

TABLE II
QUALITY OF COMMENTS GENERATED, WITHOUT SBT PREPROCESSING

Preprocessing	Embedding model	BLEU-4
Lowercased	GloVe	29.7
Lowercased	code2vec	29.3
Lowercased	**No pretrained embeddings**	**31.3**
Non-lowercased	GloVe	22.0
Non-lowercased	code2vec	31.0
Non-lowercased	No pretrained embeddings	26.7

Results: We report results in Table I. As the SBT traversal adds structural information to the model, we wish to investigate if not doing so will affect the performance of the model. Thus, we present the results of running a seq2seq model without performing proprocessing with the SBT traversal. In this case, without the handling of rare tokens in the source code, there are over 300,000 tokens that are unrepresented in the code2vec vectors. The results of not using the AST node information in comment generation are given in Table II

Findings: Based on the results, it appears that the use of pretrained embeddings do not improve the sequence-to-sequence model. The best performing configuration does not use either code2vec or GloVe. However, code2vec token vectors outperforms GloVe vectors in each setting, indicating that structural information may be valuable during the training of code embeddings. These results suggest that this approach of generating code comment cannot utilize any semantic information encoded in either the GloVe or code2vec vectors to boost performance.

B. Code Authorship Identification

Our second task is to identify the authors of short programs. While identifying authors of natural language documents have been studied extensively [41], there are fewer works for doing so on source code.

This task has many implications for privacy and security concerns. For example, techniques on this task may be used to violate the privacy of programmers. They may be used to de-anonymize programmers who wish to hide their identities, such as the creator of Bitcoin. Other uses of such techniques include copyright infringement or plagiarism detection. The process of identifying the author of a code fragment may also be useful for identifying the authors of malware and other malicious programs.

To identify authors successfully, approaches must be able to distinguish between the coding styles of programmers in their code. Techniques used in this task leverage features that express the programming style of programmers, such as layout and lexical features [42]. Previous works have found that the use of machine learning using TF-IDF features covering unigrams, bigrams and trigrams can be used to identify programmers at high accuracy [21].

Dataset: As we did not manage to find any of the datasets used in prior work, we follow the same procedure described by Abuhamad et al. [21] to build a similar dataset. This dataset is collected from program submissions to the Google Code Jam.[3] The Google Code Jam is a programming competition organized by Google over several years. Participants may choose from several programming languages and have to solve a small number of problems within a short time period. For any problem, a participant may make multiple submissions. Naturally, in this setting, a single program only has one author.

We obtain 9 programs written in Java from 250 authors participating in Google Code Jam and train a model over the dataset. In total, there are 2250 programs in our dataset. On average, there are 106 lines in each Java program although the number of lines varies from 1 to over 70000 lines. Existing works on code authorship [21], [42] evaluate their approaches using accuracy. The dataset is constructed such that each author has the same number of programs in it. Thus, as a classification task, the classes are balanced and accuracy is a sufficient evaluation metric.

While the previous task of code comment generation evaluates techniques for capturing semantic properties of code, we select this task for evaluation as it requires techniques to encode features related to syntactic style. The ability of code embeddings to improve basic models on this task may indicate that it is able to distinguish between syntactic styles of different authors.

Approach: Inspired by the work of Abuhamad et al. [21], we propose a similar neural network. As baseline, we compare our approach against a model using TF-IDF features, which was shown to be effective by Abuhamad et al. [21]. This network is comprised of 2 hidden fully-connected layers with 1024 nodes. We use dropout for regularization and set it to 0.6. We use the top 1000 TF-IDF features, determined by feature selection using the ANOVA F-value between each feature and the authors. While the work of Abuhamad et al. [21] used a random forest classifier based on the intermediate outputs of their neural network, we were not able to replicate good results, without our own modifications, on a neural network based on the architecture they described. As such, we experimented with a neural network with some modifications that allowed it to produce comparably accurate predictions. As our goal was to evaluate the code embeddings and not to have a state-of-the-art system, we did not use the second step

[3]https://code.google.com/codejam/past-contests

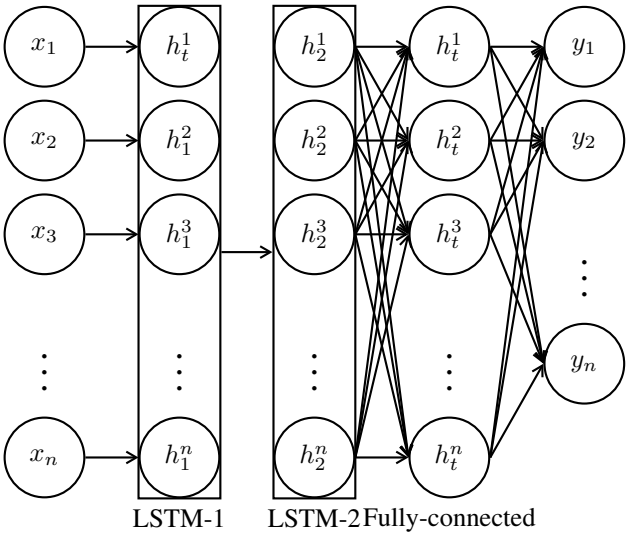

Fig. 2. Neural Network for code authorship identification

of passing the neural network results through a random forest classifier once we achieved good performance on our model.

While our objective is to evaluate code embeddings, we were not able to successfully replace the TF-IDF features with them. We attempted to use the average of the code vectors in the program and replace each TF-IDF feature with one dimension of the averaged code vectors. This will result in 128 features (each corresponding to one dimension of the code vectors). However, we find that this results in poor performance. Further tuning of the model's hyperparameters did not help to improve its performance.

As there are 1000 TF-IDF features and only 128 dimensions in our code vectors, the number of input features decreased from 1000 to 128 features. Hence, it may be possible that the poor performance can be attributed to the decrease in the number of features. We tried to train a new set of code2vec vectors of 1000 dimensions. However, this did not improve the performance of the model. Thus, to further evaluate the potential of code embeddings on this task, we used another neural network using LSTM layers. An LSTM neural network provides the advantage of allowing variable length input, hence we can input the entire code snippet into our model. This neural network comprises of 2 hidden LSTM layers followed by a fully-connected layer. We illustrate the neural network in Figure 2.

We limited our study to programs written in Java, the same language that code2vec and our GloVe vectors were trained on. For the LSTM neural network, we use both pretrained embeddings as well as randomly initialized embeddings. We present the accuracy as a number between 0 to 100, with 100 indicating a perfect accuracy. For each model, we train to 50 epochs.

Results: From Table III, we see that initializing the LSTM neural network using both code2vec and GloVe embeddings underperforms a randomly initialized embedding layer. Furthermore, the use of the LSTM neural network using code embeddings underperformed a fully-connected neural network that used TF-IDF features. When using the code embeddings with the neural network with only fully-connected layers, we get accuracies of near 0.

TABLE III
ACCURACY FOR IDENTIFICATION OF CODE AUTHORSHIP

Setting	Accuracy
LSTM, code2vec	39
LSTM, GloVe	50
LSTM, randomly initialized	69
Fully connected layers, TF-IDF	**77**

Findings: Comparing code2vec vectors and GloVe vectors, the GloVe vectors obtained a higher accuracy than the code2vec vectors. This implies that GloVe embeddings may encode syntactic relationships better than the code2vec token embeddings. However, both GloVe and code2vec token embeddings were outperformed by a randomly initialized set of embeddings. This suggests that code2vec token embeddings do not generalize to the task of code authorship identification. Finally, the model with the TF-IDF features is the best performing model in our experiments. The poor performance of using code2vec token embeddings suggests that they are unable to distinguish between the syntactic differences of code authors as well as TF-IDF features.

C. Detecting Code Clones

Finally, our last task is to detect code clones. Code clones detection is the task of determining if a pair of code fragments are similar to each other. This task has received much attention in the literature. Detecting code clones has numerous implications for software development and maintenance. For example, code clones can potentially increase the cost of maintenance, complicating the design of software and make it difficult to introduce minor changes in the long run [43], [44]. Code clones are also likely to cause bugs to be propagated through copy-paste behaviour in a software system [45], [46].

Dataset: We use 2 datasets for this task. Firstly, we use the standard BigCloneBench [47]–[49], which is a benchmark of known clones in the IJaDataset [50]. The IJaDataset is a large repository of over 25000 open-source Java projects, with over 3 million source files. In BigCloneBench, a code fragment is a single method and there are over 8 million validated code clones in the dataset. As only a subset of code fragment pairs in the entire IJaDataset is validated, the BigCloneBench benchmark reports only the estimated recall of a model, but not its precision. As such, we use a second dataset, OJClone [27], [51]. OJClone is a dataset of 104 programming problems with the student submissions to each problem. Each programming problem has 500 submissions. For detecting code clones, two submissions to the same problem are considered as code clones. In total, there are 52000 code fragments in this dataset. Between each pair of code fragments, we can determine

TABLE IV
RECALL ON BIGCLONEBENCH

Setting	Type-1	Type-2	Strong Type-3	Moderately Type-3	Weak Type-3, Type-4
code2vec	0.99	0.81	**0.50**	**0.28**	**0.16**
GloVe	0.92	0.81	**0.50**	**0.28**	**0.16**
SourcererCC	**0.98**	**0.93**	0.43	0.01	0.00

whether a pair is code clones based on whether or not they are submitted to the same programming problem. As such, we can compute both recall and precision on this dataset.

Researchers have classified code clones into 4 types, based on the level of syntactic similarity. Pairs of Type-1 clones are syntactically the same, while pairs of Type-4 clones have no syntactic similarity but implements the same functionality. While BigCloneBench classifies each code clone from Type-1 to Type-4, there is no classification of the code clones into their types in the OJClone dataset although previous work have considered them to be Type-3 and above [51].

For Type-1 and Type-2 clones, approaches that only use syntactic information is sufficient for achieving high precision and recall, but for Type-3 and Type-4 clones which are syntactically different, approaches need to consider the semantics of the program fragment. There has been documented difficulties in detecting Type-3 and Type-4 clones at high precision and recall [52].

Success at detecting code clones of Type-1 and Type-2 indicates that syntactic information is encoded within an approach, while successfully detecting Type-3 and Type-4 code clones may indicate that semantic information can be encoded. Thus, this task measures both the ability of code embeddings to capture syntactic and semantic information at the same time.

Approach: Technique-wise, we used SourcererCC [22], a token-based model that gives near state-of-the-art performance for Type-1 to Type-3 clones, as basis for our work. SourcererCC compares the tokens contained in pairs of code fragments. To do so efficiently, SourcererCC implements a sophisticated algorithm using several properties and heuristics they identified to reduce the number of comparisons to perform. However, the main criterion to determine if two code fragments are code clones is based on the number of tokens that are present in both code fragments. Given 2 code fragments B_x and B_y, to determine if they are code clones, it counts the number of tokens that overlap in both code fragments, computing its overlap similarity, and compare it against a configurable threshold, θ, to identify if the fragments are code clones. The measure of overlap, $O(B_x, B_y)$, is computed as follows:

$$O\left(B_x, B_y\right) = |B_x \cap B_y|$$

To adapt SourcererCC to use vectors of source code tokens, we changed the criteria to determine if a pair of code fragments are code clones. Instead of a count-based measure of the overlap between tokens, our criteria is based on the cosine similarity of the average of the token vectors of the 2 code fragments. Prior work has shown that averaging vectors for both natural language and source code can be used to represent larger fragments of tokens [6], [53], [54].

Thus, in our adaptation of SourcererCC, for two code fragments, B_x and B_y, to be considered as code clones, they must share at least one token, and the Cosine Similarity between them should exceed a threshold, θ (in this work, we use a default value of $\theta = 0.8$). This requirement of sharing at least one token is made for scalability reasons. Without this additional criteria, all pairs of code fragments need to be compared, which will be prohibitively large for our experiments. For tokens in the code fragments that not in the embeddings' vocabulary, we use the zero vector. The Cosine Similarity of two code fragments is computed based on averaging all the vectors of tokens contained in B_x and B_y. After averaging the token vectors, we have X and Y, the average of token vectors in the two code snippets, that are vectors in an n-dimensional space. Let X_i refer to the i^{th} dimensional value of the vector X. Then, the Cosine Similarity is computed as follows:

$$\frac{\sum_{i=1}^{n} X_i \times Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2}\sqrt{\sum_{i=1}^{n} Y_i^2}}$$

Results: As described earlier, we evaluated our adaptations to SourcererCC on two datasets, BigCloneBench and OJClone. In our experiments on BigCloneBench, we only consider clones that are greater than 6 lines and 50 tokens. This is the standard configuration for measuring recall [52]. As described earlier, BigCloneBench does not provide a process for evaluating precision. Table IV shows the recall of our approach on each clone type. For ease of interpretation, we provide the definition of the classification of the clone types that are widely used in literature:

- Type-1: Identical code fragments differing by whitespace, comments
- Type-2: Identical code fragments differing by identifier names or literal values
- Type-3: Code fragments that have statements added, modified, or removed
- Type-4: Code fragments that semantically perform the same computation with little syntactic similarity.

Type-3 clones can be split up further based on the level of syntactic similarity. Strong Type-3 clones refer to Type-3 clones that are syntactically similar, while weak Type-3 clones

TABLE V
RECALL AND PRECISION ON THE OJCLONE DATASET

Setting	Precision	Recall	F1
code2vec	0.03	0.45	**0.06**
GloVe	0.03	**0.67**	**0.06**
SourcererCC	**0.87**	0.01	0.01
Random	0.01	0.02	0.01

TABLE VI
COUNTS OF TOKENS IN THE CODE FRAGMENTS THAT ARE OUT OF THE EMBEDDINGS' VOCABULARY

Vectors	Tokens found	OOV tokens
Code2vec	15,172,900,000	250,200,000
GloVe	14,861,900,000	561,200,000

are syntactically dissimilar.

The results of evaluating SourcererCC on BigCloneBench suggests that the use of code embeddings improve the recall of the SourcererCC on the less syntactically-similar clone types (Type-3 and Type-4), but may cause the recall on Type-1 and Type-2 clones to drop. However, it is unclear what its effects on precision are.

In order to evaluate our approach's precision, we used a second dataset, OJClone. We used all 104 programming contest questions in the dataset and our results are shown in Table V. As about 1% of pairs of code snippet in this dataset are clone pairs, we introduce another baseline where we randomly accept a pair of code fragments as code clones 1% of the time.

The clones in the OJClone dataset has been previously considered to be of Type-3 and above [51]. The increase in recall that we see is consistent with what we observe in the BigCloneBench benchmark on Type-3 and Type-4 clones. On the other hand, the results suggest that our approach causes the precision of SourcererCC to decline. Due to the poor recall of SourcererCC, the overall F1 of the approach using code embeddings are slightly higher than the F1 when using SourcererCC alone.

One hypothesis is that the poor precision of code embeddings is caused by a large number of tokens encountered in the task's dataset that are out-of-vocabulary (OOV) in the embeddings. We count the number of times our approach tried to retrieve a code vector, and the number of times it failed to retrieve a token. The counts are presented in Table VI. We see that over 95% of the time, our approach can successfully retrieve the vectors of the tokens in the program fragments. Therefore, out-of-vocabulary tokens are not the cause of poor performance of our model.

Findings: While it appears that the approaches using embeddings improves the overall F1 score, the improvement is small. It is unconvincing that the approaches using code embeddings have encoded semantic qualities of code fragments necessary to detect code clones. Overall, the recall of all approaches are low and the embeddings-enhanced approaches suffer a drastic loss of precision (0.87 to less than 0.1). Therefore, we do not conclude that embeddings have successfully improved SourcererCC's ability to detect clones.

From a practical standpoint, our augmentation to SourcererCC also resulted in a large decline in speed, as we cannot use the heuristics employed by SourcererCC to reduce the number of clone pairs to compare. Scalability of detecting code clones is a key concern for real-world usage, and we note that our technique is only proposed to evaluate the code embeddings and our technique may not be appropriate for real-world usage due to its lack of scalability.

D. Lessons Learned

From the 3 tasks above, we see that code embeddings cannot be used readily to improve simpler models. In fact, in 2 out of 3 tasks, the use of embeddings lowers the performance of the models they are added to. In the case of the code authorship task, simpler approaches such as TF-IDF outperforms models augmented with embeddings of source code tokens. The only task where the code vectors do not cause performance to deteriorate is the task of detecting code clones. Even in this task, it appears that the use of embeddings exchanges SourcererCC's high precision for a higher recall while maintaining a similar F1 score.

Our findings support the observations by Fu and Menzies [31] that simpler baselines run faster and may outperform complex techniques and they should be used as baselines. We see that on the task of code authorship identification, the use of code embeddings under-perform a simpler approach that uses simple TF-IDF features. Likewise, the use of code embeddings did not improve performance of SourcererCC. In short, having a continuous representation of code tokens may not necessarily perform better than simple baselines that treat code tokens simply as symbols.

Code embeddings may not be a silver bullet to boost the performance of deep learning models; other considerations may have more impact. For example, we found that the pre-processing on the data has large consequences on the model's performance. One hurdle to the effective use of neural networks in the Software Engineering domain is the problem of out-of-vocabulary tokens. Hellendoorn and Devanbu have raised this issue before [55], and in their work, suggested that deep learning techniques struggle with the large vocabulary of source code. They demonstrated their point with a non-deep learning model that can update and expand its vocabulary, outperforming a deep learning model with a fixed vocabulary. Our experiments validate the importance of pre-processing on the code comment generation task, in which the technique of converting rare tokens into the AST node type proposed by Hu et al. result in improved performance. On the other hand, on both the code clone detection and code authorship tasks, we achieve poor performance although out-of-vocabulary tokens did not appear to be an issue on those tasks. In the code authorship identification task, the baseline features of just 1000 TF-IDF features were sufficient for good performance.

The composition of source code token embeddings requires further investigation. In the domain of natural language, it has been suggested that the composition of tokens by summation, averaging or concatenation may be useful for representing larger fragments of tokens [6], [53], [54]. However, in the code clones detection task, the composition of source code tokens by these operators does not appear to represent a body of tokens meaningfully enough to effectively detect code clones. We believe that this result should motivate research into other operators or methods of composing code embeddings.

We find the lack of interpretability of code embeddings to be a source of difficulty in this work. SourcererCC's criteria of measuring the overlap of code tokens is simple and easy to interpret, on the other hand, our approach averaging code vectors result in a representation that is hard to debug. In additional, there are practical ramifications of our approach. The original SourcererCC, with its heuristics based on its count-based criteria, runs in a significantly shorter time and achieves a comparable F-measure.

Finally, due to the lack of success we face trying to improve models with code embeddings, **it may indicate that token embeddings learned over source code may not encode a significant amount of either semantic or syntactic information usable in different downstream tasks**. We are unable to use either the code2vec or GloVe embeddings effectively in the 3 downstream tasks we identified, which may suggest a lack of generalizability of token embeddings.

We believe our findings should motivate more work in the area of finding good representations of code tokens, and that future work on code representation should aim to address the difficulties of using code embeddings on downstream tasks.

IV. Threats to Validity

A. Threats to Internal Validity

Threats to internal validity concern factors that may influence our results. Our evaluation of code embeddings relies on enhancing other techniques with the embeddings. One limitation is that the techniques we have picked may not be suitable for using a vector representation of source code. Another limitation is that our integration of code2vec into these techniques may be too naive. We have tried to mitigate these limitations by selecting techniques that already use code tokens in its input, and by using a variety of different techniques in our evaluation. Both neural network-based approaches and techniques to use vector space calculations are used, which will explore both the potential of code embeddings to be used to initialize a neural network model and also its ability to encode semantic and syntactic qualities through vector space calculations.

Our experiments do not provide any insight about the code embeddings' lack of generalizability that we observed. They suggest that code embeddings may not generalize beyond the task it was trained on, but we were unable to find conclusive reasons explaining why the code embeddings did not help existing techniques. We leave further experiments and detailed analysis that may provide these reasons for future work.

B. Threats to External Validity

Threats to external validity concern the generalizability of our findings. While we have experimented on 3 downstream tasks, there are other tasks that may benefit from the use of pretrained token embeddings. It may also be possible that the 3 downstream tasks we have selected are not the best tasks for applying code embeddings. For example, we only considered downstream tasks where the input data is homogeneous. We did not explore the effectiveness of code embeddings in downstream tasks involving heterogenous inputs, such as duplicate bug report detection [56] or duplicate StackOverflow post detection [57]. The inputs to models may contain multiple types of data (e.g. code snippets, stack traces, and text). To use code embeddings, they will have to be used together with other types of embeddings (e.g. word embeddings for natural language text) and it will be interesting to observe if code embeddings is helpful in these tasks. However, existing literature does not suggest what common software engineering tasks can benefit from token embeddings. To the best of our knowledge, this is the first work that applies any model of code embeddings to multiple downstream tasks. Indeed, the lack of obvious software engineering tasks to apply code embeddings on is motivation of our work. Moreover, in our work, we try to cover a diversity of tasks that are different from one another. For example, only code comment generation is closely related to Natural Language Processing. Each task is likely to measure different qualities that the code embeddings can encode. For example, the code authorship tasks will require techniques to distinguish between syntax preferences of different authors while detecting Type-3 and Type-4 code clones will require techniques that detect the same semantic functionality. In addition, the tasks in this work involve both generative and classification tasks.

Our experiments also may not imply anything about embeddings of other granularity of source code. Embeddings have been trained over execution traces [53] or sequences of API method invocations [58]. These embeddings may be generalizable to other downstream tasks that do not use token-based approaches. We note that evaluating these embeddings are out of the scope of our work, and we do not say anything about their generalizability. In this work, we focus only on embeddings of source code tokens and code2vec is selected to be representative of these embedding techniques. Comparison with other code embedding techniques is beyond the scope of this paper and it is worth investigating them in future to confirm or refute the findings of this work.

V. Related work

In this section, we discuss prior work on evaluation of word embeddings done in the NLP domain and embeddings of source code. Due to page limitations, the survey here is by no means complete.

A. Evaluation of word embeddings

In Natural Language Processing, there has been identical work on evaluating word embeddings. Evaluation of word embeddings can be categorized into intrinsic and extrinsic evaluation [4], [59]. Intrinsic evaluation involves the use of word analogy or similarity tasks, while extrinsic evaluation refers to the evaluation of embeddings when used on downstream tasks. Research has also found that intrinsic evaluations of word embeddings do not correlate with extrinsic performance [60]. We believe that these insights and conclusions are applicable to code embeddings as well, thus our work performs an extrinsic evaluation of code embeddings.

B. Embeddings for source code

Granularity of embeddings: Other than code2vec, there have been many proposed embeddings for code. A survey of existing code embeddings are presented by Chen and Monperrus [61], where works on embeddings are into categories depending on the granularity of source code that is embedded: source code tokens, functions, sequences or sets of method calls, and binary code are the granularities of source code that have been considered.

For examples of models that embed granularities of program elements other than source code tokens, consider the work by Xu et al. [62] and Theetan et al. [18] Xu et al. trained embeddings of binary instructions and Theetan et al. trained embeddings of library imports. While in this work we investigated only token-based code embeddings, future work should investigate and evaluate other granularities of code embeddings on downstream tasks.

Token-based embeddings: In our work, we focused on embedding source code tokens, which we consider to be the most specific granularity. Researchers have trained and used embeddings in a diverse set of tasks.

There are several examples of models that trained embeddings from source code tokens into a vector space [13], [14], [16], [63]. Azcona et al. [13] proposed user2code2vec, which are embeddings trained for profiling students. These embeddings are used to predict if a student's submissions are correct. White et al. [14] trained embeddings for automatic program repair. They used embeddings to compute the similarity between identifiers for use in their repair technique. They used a recurrent neural network language model to learn embeddings and used them to transform program repair "ingredients" by replacing identifiers based on identifier similarity. For code search, Gu et al. [16] trained a joint-embedding space representing both code snippets and method documentation. They represented code snippets by their method names, API sequences in the method bodies, and the tokens in the method bodies. They are jointly trained such that the method and its documentation are embedded near to each other in the vector space. Hellendoorn et al. [63] used a deep neural network for type inference for dynamically-typed languages. Their model first embeds tokens into a vector space then learn type vectors in order to annotate the types of variables.

Evaluation of embeddings: While there are many embeddings proposed on a large variety of tasks, many research works, including the works discussed above, did not evaluate their embeddings on downstream tasks or did so only on tasks that are closely related to the training task [13]–[19]. We found only few examples of works that evaluate their work on downstream tasks.

Similar to code2vec, Alon et al. [29] trained embeddings of AST paths. They evaluated it on predicting names of variable and methods, and predicting types of local variables. They compared it against a baseline trained using word2vec [1]. Defreez et al. [64] proposed `func2vec`, which maps synonymous functions to vectors grouped together and a downstream task of mining error-handling specifications in the Linux kernel. Their miner successfully detected 2 violations of the specifications they mined.

Henkel et al. [53] proposed to embed traces of symbolic execution into a vector space. They evaluated their code embeddings on a downstream task of predicting error codes. Their results indicate that their embeddings may be useful for finding bugs or suggesting repairs. In their work, they proposed a benchmark for the code analogy task. The code analogy task is an intrinsic evaluation on code embeddings, where the embeddings are evaluated on their ability to express relationships between analogous words in the vector space, such as the analogy that "mutex_lock is to mutex_unlock as spin_lock is to spin_unlock". However, work on NLP [60] has suggested that performance on the word analogy task does not imply good performance on downstream tasks.

In these works, while the code embeddings are shown to be useful, they are often not compared against simpler baselines and the use of code embeddings is evaluated only in at most one other task. One exception is the work by Ben-Nun et al. [65], where the trained embeddings are evaluated on 3 downstream tasks. They trained statement embeddings over a graph constructed from both the data and control-flow graph. They evaluated their embeddings on 3 downstream tasks of classifying algorithms, a prediction task to predict if a program will run faster on a CPU or GPU, and another prediction task of the amount of work done on each GPU thread while running a given program. However, unlike our evaluation of code2vec, the downstream tasks they use include uncommon software engineering tasks and are similar to one another.

VI. Conclusion and Future Work

To conclude, our experiments on source code embeddings suggest that they do not generalize readily to other tasks. We performed experiments using code embeddings on three downstream tasks: code comment generation, code authorship identification and code clones detection. In each task, the code embeddings do not result in models with improved overall performance. Furthermore, in two of the tasks, they are outperformed by simpler models.

As a consequence of our work, we call for the community to evaluate embedding models more carefully. Similar to the work already done for NLP, we propose that the usefulness of

embeddings are more appropriately evaluated using a variety of downstream tasks. While it may be interesting to have distributed representations of program elements, it is far more important that embeddings can help in downstream tasks. Users of code embeddings should be careful in their choice of code embeddings, keeping in mind that not all code embeddings will necessarily be helpful for their targeted downstream task.

For future work, a more comprehensive evaluation of existing source code token embeddings can be done on the three tasks we identified in this work. Deeper analysis of the differences between embeddings may lead to deeper insights into how to train and use token embeddings. Beyond token embeddings, an evaluation of distributed representations of other granularities, e.g. function embeddings, in downstream tasks is a natural next step for future work.

We end with a call for further research beyond the introduction of new models of training code embeddings, but to describe how the embeddings can be used for a variety of downstream tasks and to demonstrate that they can be useful beyond the single task they were trained on. We believe that the software engineering community should not view the training of embeddings as an end to itself, but instead, as a means to achieve better performance in other tasks.

REFERENCES

[1] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[2] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[3] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[4] T. Schnabel, I. Labutov, D. Mimno, and T. Joachims, "Evaluation methods for unsupervised word embeddings," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 298–307.

[5] J. Turian, L. Ratinov, and Y. Bengio, "Word representations: a simple and general method for semi-supervised learning," in *Proceedings of the 48th annual meeting of the association for computational linguistics*. Association for Computational Linguistics, 2010, pp. 384–394.

[6] I. Iacobacci, M. T. Pilehvar, and R. Navigli, "Embeddings for word sense disambiguation: An evaluation study," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 897–907.

[7] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[8] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 111–124.

[9] V. Efstathiou and D. Spinellis, "Semantic source code models using identifier embeddings," in *16th International Conference on Mining Software Repositories (MSR 2019)*, 2019.

[10] M. Pradel and K. Sen, "Deepbugs: a learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 147, 2018.

[11] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 2073–2083.

[12] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[13] D. Azcona, P. Arora, I.-H. Hsiao, and A. Smeaton, "user2code2vec: Embeddings for profiling students based on distributional representations of source code," in *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*. ACM, 2019, pp. 86–95.

[14] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 479–490.

[15] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[16] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[17] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, "Learning program embeddings to propagate feedback on student code," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*. JMLR. org, 2015, pp. 1093–1102.

[18] B. Theeten, F. Vandeputte, and T. Van Cutsem, "Import2vec-learning embeddings for software libraries," *16th International Conference on Mining Software Repositories (MSR 2019)*, 2019.

[19] C. S. Corley, K. Damevski, and N. A. Kraft, "Exploring the use of deep learning for feature location," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 556–560.

[20] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 200–210.

[21] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 101–114.

[22] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 1157–1168.

[23] M. Allamanis, M. Brockschmidt, and M. Khademi, "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*, 2018.

[24] J. Devlin, J. Uesato, R. Singh, and P. Kohli, "Semantic code repair using neuro-symbolic transformation networks," *arXiv preprint arXiv:1710.11054*, 2017.

[25] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 95–104.

[26] N. D. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[27] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[28] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[29] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *ACM SIGPLAN Notices*, vol. 53, no. 4. ACM, 2018, pp. 404–419.

[30] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to sport and refactor inconsistent method names," in *41st ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[31] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 49–60.

[32] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.

[33] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 223–226.

[34] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[35] D. Coughlin, "Correlating automated and human assessments of machine translation quality," in *Proceedings of MT summit IX.*, 2003, pp. 63–70.

[36] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 709–712.

[37] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 373–384.

[38] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 135–146.

[39] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "Opennmt: Open-source toolkit for neural machine translation," in *Proc. ACL*, 2017. [Online]. Available: https://doi.org/10.18653/v1/P17-4012

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[41] E. Stamatatos, "A survey of modern authorship attribution methods," *Journal of the American Society for information Science and Technology*, vol. 60, no. 3, pp. 538–556, 2009.

[42] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 255–270.

[43] J. Mayrand, C. Leblanc, and E. Merlo, "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*, 1996, p. 244.

[44] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 25–34.

[45] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.

[46] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 273–282.

[47] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 131–140.

[48] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[49] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 596–600.

[50] "Ambient software engineering group seclone project," https://sites.google.com/site/asegsecold/projects/seclone, accessed: 2019-05-13.

[51] H.-H. Wei and M. Li, "Positive and unlabeled learning for detecting software functional clones with adversarial training," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, 2018, pp. 2840–2846.

[52] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 354–365.

[53] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 163–174.

[54] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196.

[55] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 763–773.

[56] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 45–54.

[57] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Mining duplicate questions in stack overflow," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 402–412.

[58] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deepam: migrate apis with multi-modal sequence to sequence learning," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 3675–3681.

[59] M. Faruqui, Y. Tsvetkov, P. Rastogi, and C. Dyer, "Problems with evaluation of word embeddings using word similarity tasks," in *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, 2016, pp. 30–35.

[60] B. Chiu, A. Korhonen, and S. Pyysalo, "Intrinsic evaluation of word vectors fails to predict extrinsic performance," in *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, 2016, pp. 1–6.

[61] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," *arXiv preprint arXiv:1904.03061*, 2019.

[62] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 363–376.

[63] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 152–162.

[64] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to specification mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018.

[65] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Advances in Neural Information Processing Systems*, 2018, pp. 3585–3597.