GraphCode2Vec: Generic Code Embedding via Lexical and Program Dependence Analyses

Wei Ma*
University of Luxembourg
Luxembourg
wei.ma@uni.lu

Mengjie Zhao*
LMU Munich
Germany
mzhao@cis.lmu.de

Ezekiel Soremekun
University of Luxembourg
Luxembourg
ezekiel.soremekun@uni.lu

Qiang Hu
University of Luxembourg
Luxembourg
qiang.hu@uni.lu

Jie M. Zhang
University College London
United Kingdom
jie.zhang@ucl.ac.uk

Mike Papadakis
University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

Maxime Cordy
University of Luxembourg
Luxembourg
maxime.cordy@uni.lu

Xiaofei Xie
Singapore Management University
Singapore
xfxie@smu.edu.sg

Yves Le Traon
University of Luxembourg
Luxembourg
yves.letraon@uni.lu

ABSTRACT

Code embedding is a keystone in the application of machine learning on several Software Engineering (SE) tasks. To effectively support a plethora of SE tasks, the embedding needs to capture program syntax and semantics in a way that is *generic*. To this end, we propose the *first self-supervised pre-training* approach (called GraphCode2Vec) which produces task-agnostic embedding of lexical and program dependence features. GraphCode2Vec achieves this via a synergistic combination of *code analysis* and *Graph Neural Networks*. GraphCode2Vec is *generic*, it *allows pre-training*, and it is *applicable to several SE downstream tasks*. We evaluate the effectiveness of GraphCode2Vec on four (4) tasks (method name prediction, solution classification, mutation testing and overfitted patch classification), and compare it with four (4) similarly *generic* code embedding baselines (Code2Seq, Code2Vec, CodeBERT, GraphCodeBERT) and seven (7) *task-specific*, learning-based methods. In particular, GraphCode2Vec is more effective than both generic and task-specific learning-based baselines. It is also complementary and comparable to GraphCodeBERT (a larger and more complex model). We also demonstrate through a probing and ablation study that GraphCode2Vec learns lexical and program dependence features and that self-supervised pre-training improves effectiveness.

KEYWORDS

code embedding, code representation, code analysis

*Both authors contributed equally.

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05.
https://doi.org/10.1145/3524842.3528456

ACM Reference Format:
Wei Ma*, Mengjie Zhao*, Ezekiel Soremekun, Qiang Hu, Jie M. Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. 2022. GraphCode2Vec: Generic Code Embedding via Lexical and Program Dependence Analyses. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524842.3528456

1 INTRODUCTION

Applying machine learning to address software engineering (SE) problems often requires a vector representation of the program code, especially for deep learning systems. A naïve representation, used in many SE applications, is one-hot encoding that represents every feature with a dedicated binary variable (a vector including binary values) [55]. However, this type of embedding is usually a high-dimensional sparse vector because the size of vocabulary is very large in practice, which results in the notorious *curse of dimensionality* problem [4]. Besides, one-hot encoding has *out-of-vocabulary* (OOV) problem, which decreases model generalization capability such that it cannot handle new type of data [59].

To deal with these issues, researchers use dense and reasonably concise vectors to encode program features for specific SE tasks, since they generalise better [30, 64, 66, 74]. More recently, researchers apply natural language processing (NLP) techniques to learn the universal code embedding vector for general SE tasks [1–3, 5–7, 10, 17, 23, 26, 33, 49, 51, 62, 65]. The resulting code embedding represents a mapping from the "program space" to the "latent space" that captures the different code-used semantics, i.e., the semantic similarities between program snippets. The aim is that similar programs should have similar representations in the latent space.

State-of-the-art code embedding approaches focus either on *syntactic features* (*i.e.*, tokens/AST), or on *semantic features* (*i.e.*, program dependencies) ignoring the importance of combining both features together. For example, Code2Vec [3] and CodeBERT [17]) focus on syntactic features, while PROGRAML [10] and NCC [5]) focus on program semantics. There are few studies using both program semantics and syntax, e.g., GraphCodeBERT [23]. However,

Figure 1: Motivating example showing (a) an original method (`LowerBound`), and two behaviorally equivalent clones of the original method, namely (b) a renamed method (`findLowerBound`), and (c) a refactored method (`getLowerBound`).

```java
public static int lowerBound(int[] array,
            int length, int value) {
      int low = 0;
      int high = length;
      while (low < high) {
         final int mid = (low + high) / 2;
         if (value <= array[mid]) {
            high = mid;
         } else {
            low = mid + 1;
         }
      }
      return low;
}
```

(a) Original Method

```java
public static int findLowerBound(int[] inputs,
                        int size, int v) {
      int bounder = 0;
      int l = size;
      int mindex = 0;
      while (bounder < l) {
         mindex = (bounder + l) / 2;
         if (v <= inputs[mindex]) {
            l = mindex;
         } else {
            bounder = mindex + 1;
         }
      }
      return bounder;
}
```

(b) Renamed Method

```java
public static int getLowerBound(int v,
            int size, int[] inputs) {
      int h = size;
      int mindex = 0;
      int check = 0;
      while (check < h) {
         mindex = (check + h) / 2;
         if (v > inputs[mindex]) {
            check = mindex + 1;
         } else {
            h = mindex;
         }
      }
      return check;
}
```

(c) Refactored Method

these approaches are not *precise*, they do not obtain or embed the entire program dependence graph. Instead, they estimate program dependence via string matching (instead of static program analysis), then augment AST trees with sequential data flow edges.

To address these challenges, we propose the *first approach (called* GraphCode2Vec*) to synergistically capture syntactic and semantic program features with Graph Neural Network (GNN) via self-supervised pretraining.* The *key idea* of our approach is to use *static program analysis* and *graph neural networks* to effectively represent programs in the latent space. This is achieved by combining lexical and program dependence analysis embeddings. During lexical embedding, GraphCode2Vec embeds the syntactic features in the latent space via *tokenization*. In addition, it performs dependence embedding to capture program semantics via static program analysis, it derives the program dependence graph (PDG) and represent it in the latent space using Graph Neural Networks (GNN). It then concatenates both lexical embedding and dependence embedding in the program's vector space. This allows GraphCode2Vec to be effective and applicable on several downstream tasks.

To demonstrate the importance of semantic embedding, we compare the similarity of three pairs of programs using our approach, in comparison to a syntax-only embedding approach – CodeBERT, and GraphCodeBERT, which embeds both syntax and semantic, albeit without program dependence analysis. Consider the example of three program clones in Figure 1. This example includes three behaviorally or semantically equivalent programs, that have low syntactic similarity (i.e., different tokens), but with similar semantic features, i.e., program dependence graphs (PDGs). To measure the similarity distance in the latent space, in addition to the example code clones (Figure 1), we randomly select 10 other different code methods (from GitHub) without any change to establish a baseline for comparing all approaches. To this end, we compute the average cosine similarity distance for all 91 program pairs ($\frac{14\times13}{2}$) for reference to show that all approaches report similar scores for all randomly selected 91 pairs (Table 1).[1] For all three approaches, the similarity between the "original program" and a direct copy of the program with only method name renaming to "searchLowerBound", is well captured with an almost perfect cosine similarity score for all approaches (1 or 0.99). Likewise, the cosine similarity of the original program and the "renamed" program (`findLowerBound`) is mostly well captured by all approaches, since they all embed program syntax, albeit with lower cosine similarity scores for CodeBERT (0.61) and GraphCodeBERT (0.70), in comparison to our approach (0.99).

Table 1: Cosine Similarity of three behaviorally/semantically similar program pairs from our motivating example, using GraphCodeBERT, CodeBERT and GraphCode2Vec

Program Pairs	Graph-CodeBERT	CodeBERT	GraphCode2Vec
`searchLowerBound` & `lowerBound`	1	0.99	1
`findLowerBound` & `lowerBound`	0.70	0.61	0.99
`getLowerBound` & `lowerBound`	0.70	0.51	0.99
Average of 91 pairs	-0.05	-0.06	-0.03

Meanwhile, CodeBERT fails to capture the semantic similarity between the "original program" and the "refactored program" (`getLowerBound`), even though they are behaviorally similar and share similar program dependence. This is evidenced by the low cosine similarity score (0.51), because it does not account for semantic information in its embedding, especially the similar program dependence graph shared by both programs. Lastly, GraphCodeBERT performs slightly better than CodeBERT (0.70 vs. 0.51), but lower than our approach (0.99). This is due to lack of actual static program analysis in the embedding of GraphCodeBERT, since it only applies a heuristic (string matching) to estimate program dependence, it is *imprecise*. This example demonstrates the importance and necessity of embedding precise dependence information.

A key ingredient of GraphCode2Vec is *self-supervised pretraining*. Even though task-specific learning based approaches (e.g., CNNSentence [45]) learn the vector representation of code without pre-training, they are non-generic and less effective. Applying their learned vector representation to other (SE) tasks requires re-tuning model parameters, and the lack of pretraining reflects in their performance. As an example, our evaluation (in RQ1 section 5) showed that our self-supervised pretraining approach improves effectiveness when compared to 7 task-specific approaches (i.e., without pretraining) addressing two (SE) tasks (solution classification and patch classification). To further demonstrate the importance of

[1] The purpose of computing the average cosine similarity of all 91 code pairs is to establish a meaningful reference for comparing embeddings and to serve as a sanity check. We expect the mean of the cosine similarity of a set of randomly selected pairs of code clones and non-clones to lie around zero for all approaches (range -1 to 1).

self-supervised pretraining, we compare the effectiveness of GraphCode2Vec with and without pretraining using two downstream tasks. Overall, we demonstrate that our self-supervised pretraining improves effectiveness by 28% (*see* RQ3).

To evaluate GraphCode2Vec, we compare it to four generic code embedding approaches, and seven (7) task-specific learning-based applications. We also investigate the stability and learning ability of our approach through sensitivity, ablation and probing analyses. Overall, we make the following *contributions*:

Task-specific learning-based applications. We introduce the automatic application of GraphCode2Vec to solve specific downstream SE tasks, without extensive human intervention to adapt model architecture. In comparison to the state-of-the-art task-specific learning-based approaches (*e.g.*, ODS [72]), our approach does not require any effort to tune the hyper-parameters to be applicable to a downstream task (Section 3). Our evaluation on two downstream tasks, solution classification and patch classification, showed that GraphCode2Vec outperforms the state-of-the-art task-specific learning-based applications: For all tasks it outperforms all task-specific applications (RQ1 in Section 5).

Generic Code embedding. We propose a novel and generic code embedding learning approach (*i.e.,* GraphCode2Vec) that captures the lexical, control flow and data flow features of programs through a novel combination of *tokenization*, *static code analysis* and *graph neural networks* (GNNs). To the best of our knowledge, GraphCode2Vec is the first code embedding approach to precisely capture syntactic and semantic program features with GNNs via self-supervised pretraining. We demonstrate that *GraphCode2Vec is effective* (RQ2 in Section 5): *It outperforms all syntax-only generic code embedding baselines.* We provide our pre-trained models and generic embedding for public use and scrutiny.[2]

Further Analyses. We extensively evaluate the *stability* and *interpretability* of our approach by conducting *sensitivity*, *probing* and *ablation* analyses. We also investigate the impact of configuration choices (i.e., pre-training strategies and GNN architectures) on the effectiveness of our approach on downstream tasks. Our evaluation results show that GraphCode2Vec *effectively learns lexical and program dependence features*, it is *stable* and insensitive to the choice of GNN architecture or pre-training strategy (RQ3 in Section 5).[3]

2 BACKGROUND

2.1 Generic code embedding

We discuss methods that learn general-purpose code representations to support several downstream tasks. These approaches are not designed for a specific task. There are three major types of generic code embedding approaches, namely *syntax-based*, *semantic-based* and *combined semantic and syntactic* approaches (*see* Table 2).

Syntax-based Generic Approaches: These approaches encode program snippets, either by dividing the program into *strings*, lexicalizing them into *tokens* or parsing the program into a *parse tree or abstract syntax tree (AST)*. Syntax-only generic embedding approaches include Code2Vec [3], Code2Seq [2], CodeBERT [17], C-BERT [7], InferCode[6], CC2Vec [26], AST-based NN [73] and ProgHeteroGraph [65] (*see* Table 2). Notably, these approaches use neural models for representing code (snippets), *e.g.*, via code vector (*e.g.*, Code2Vec [3]), machine translation (*e.g.*, Code2Seq [2]) or transformers (*e.g.*, CodeBERT [17]). Code2Vec [3] is an AST-based code representation learning model that represents code snippets as single fixed-length code vector. It decomposes a program into a collection of paths using an AST and learns the atomic representation of each path while simultaneously learning how to aggregate the set of paths. Code2Seq [2] is an alternative code embedding approach that uses Sequence-to-sequence (seq2seq) models, adopted from neural machine translation (NMT), to encode code snippets. CodeBERT [17] is a bimodal pre-trained model for programming language (PL) and natural language (NL) tasks, which uses transformer-based neural architecture to encode code snippets. Besides, CodeBERT [17], C-BERT [7] and Cu-BERT [33] are BERT-inspired approaches, these methods adopt similar methodologies to learn code representations as BERT [12].

GraphCode2Vec is similar to the aforementioned generic code embedding methods, it is also a general-purpose code embedding approach that captures syntax by lexicalizing the program into tokens (*see* Table 2). However, all of the aforementioned generic approaches are syntax-based, none of these approaches account for program semantics (i.e., data and control flow). Unlike these approaches, GraphCode2Vec additionally captures program semantics via static analysis. In this paper, we compare our approach (GraphCode2Vec) to the three (3) most popular and recent syntax-based generic code embedding approaches, namely Code2Vec [3], Code2Seq [2] and CodeBERT [17] (*see* section 5).

Semantic-based Generic Approaches: This refers to code embedding methods that capture *only* semantic information such as control and data flow dependencies in the program. *Semantic-only generic approaches* include NCC [5] and PROGRAML [10]. On one hand, NCC [5] extracts the contextual flow graph of a program by building an LLVM intermediate representation (IR) of the program. It then applies word2vec [43] to learn code representations. On the other hand, PROGRAML [10] is a language-independent, portable representation of whole-program semantics for deep learning, which is designed for data flow analysis in compiler optimization. It adopts message passing neural networks (MPNN) [22] to learn LLVM IR representations. In contrast to these approaches, GraphCode2Vec captures both semantics and syntax.

Combined Semantic and Syntactic -based Approaches: There are generic approaches that capture both syntactic and semantic features such as IR2Vec [5], OSCAR [49], ProgramGraph [1], ProjectCodeNet [51] and GraphCodeBERT [23]. IR2Vec [5] and OSCAR [49] use LLVM IR representation of a program to capture program semantics. Meanwhile, ProgramGraph [1] uses GNN to learn syntactic and semantic representations of code from ASTs augmented with data and control edges. ProgHeteroGraph leverages abstract syntax description language (ASDL) grammar to learn code representations via heterogeneous graphs [65]. Finally, GraphCodeBERT [23] is built upon CodeBERT [17], but in addition to capturing syntactic features it also accounts for semantics by employing data flow information in the pre-training stage.

[2] https://github.com/graphcode2vec/graphcode2vec

[3] In the rest of this work, we interchangeably use the terms “lexical” and ”syntactic” interchangeably, as well as “(program) dependence” and “semantic”. Such that the terms “lexical embedding” and “syntactic embedding” refer to the embedding of program syntax, and the terms “dependency embedding” and “semantic embedding” refer to the embedding of program dependence information.

Table 2: Details of the state-of-the-art Code Embedding approaches. "Semantic" or "Sem" means program dependence, and "Syntactic" or "Syntax" refers to strings, tokens, parse tree or AST-tree. Symbol "✓" means the approach supports a feature, and "×" means it does not support the feature.

Type		Approaches	Syntactic	Semantic	Granularity	
					Method	Class
Task-specific	*Syntax*	CNNSentence [45]	✓	×	×	✓
		OneCNNLayer [50]	✓	×	×	✓
		SequentialCNN [21]	✓	×	×	✓
	Both	SimFeatures [63]	✓	✓	×	✓
		Prophet [41]	✓	✓	×	✓
		PatchSim [69]	✓	✓	×	✓
		ODS [72]	✓	✓	×	✓
Generic	*Syntax-only*	CodeBERT [17]	✓	×	✓	×
		Code2Vec [3]	✓	×	✓	×
		Code2Seq [2]	✓	×	✓	×
		C-BERT [7]	✓	×	✓	✓
		InferCode [6]	✓	×	✓	✓
		CC2Vec [26]	✓	×	✓	✓
		AST-based NN [73]	✓	×	×	✓
		ProgHeteroGraph [65]	✓	×	✓	×
	Sem.	NCC [5]	×	✓	✓	✓
		PROGRAML [10]	×	✓	✓	✓
	Both	IR2Vec [5]	✓	✓	✓	✓
		OSCAR [49]	✓	✓	✓	✓
		ProgramGraph [1]	✓	✓	✓	✓
		ProjectCodeNet [51]	✓	✓	×	✓
		GraphCodeBERT [23]	✓	✓	✓	×
		GraphCode2Vec	✓	✓	✓	✓

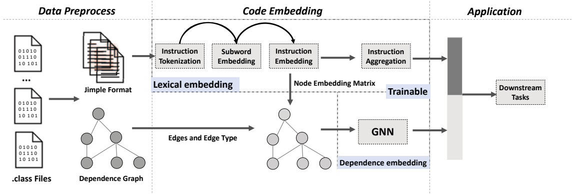
Similar to these approaches, our approach (GraphCode2Vec) learns both syntactic and semantic features. In this work, we compare GraphCode2Vec to GraphCodeBERT because it is the most recent state-of-the-art and closely related approach to ours, since it captures both syntax and semantics (*see* RQ2 section 5).

2.2 Task-specific learning-based applications

Task-specific learning-based approaches are typically designed for a single SE task, but GraphCode2Vec is amenable to several tasks and learns a generic code representation beyond the specific task-at-hand, via model pre-training. Researchers have proposed specialised learning-based techniques to tackle specific (SE) downstream tasks, e.g.. *patch classification* [41, 72] and *solution classification* [21, 45, 50]. In our experiments, we consider specialised learning approaches for both tasks. This is because these tasks have several software engineering applications, especially during software maintenance and evolution [41, 45, 72]. Table 2 highlights details of our task-specific learning methods.

Solution classification: Let us describe the state-of-the-art learning-based approaches for solution classification. Most of these approaches are syntax-based and adopt convolution neural networks (CNNs) to classify programming tasks. SequentialCNN [21] applies a CNN to predict the language/tasks from code snippets using lexicalized tokens represented as a matrix of word embeddings. CNNSentence [45] is similar to SequentialCNN since it also uses CNNs, except that it classifies source code without relying on keywords, *e.g.*, variable and function names. It instead considers the structural features of the program in terms of tokens that characterize the process of arithmetic processing, loop processing, and conditional branch processing. Finally, OneCNNLayer [50] also uses CNN for solution classification. It firstly pre-processes the program to remove unwanted entities (*e.g.*, comments, spaces, tabs and new lines), then tokenizes the program to generate the code embedding using word2vec. The resulting embedding includes the token connections and their underlying meaning in the vector space.

Figure 2: Overview of GraphCode2Vec



Patch Classification: These are techniques designed to determine the correctness of patches (i.e., identify correct, wrong or over-fitting patches). These learning-based techniques can be static (e.g., ODS [72]), dynamic (e.g., Prophet [41]), heuristic-based (e.g., PatchSim [69]) or hybrid (e.g., SimFeatures [63]). Table 2 provides details of these approaches. Notably, they all capture both syntactic information (e.g. via AST) and program dependence information (e.g., via execution paths or control flow information). For instance, PatchSim [69] is a *heuristic approach* that leverages the behavioral similarity of test case executions to determine patch correctness by leveraging the complete path spectrum of test executions. Meanwhile, Wang et al. [63] proposed (SimFeatures –) a *hybrid strategy that identifies correct patches by integrating static code features with dynamic features or (test) heuristics*. SimFeatures combines a learned static code model with dynamic or heuristic-based information (such as the dependency similarity between a buggy program and a patch) using majority voting. More recently, Ye et al. [72] proposed a supervised learning approach (called ODS) that employs static code features of patched and buggy programs to determine patch correctness, specifically to classify over-fitting patches. It uses supervised learning on extracted static code at the AST level to learn a probabilistic model for determining patch correctness. ODS also tracks program dependencies by tracking control flow statements. For this task, we compare GraphCode2Vec to ODS, PatchSim, Prophet and SimFeatures (*see* Section 5).

In this work, we compare GraphCode2Vec to the aforementioned seven (7) learning-based methods for solution classification and patch classification (*see* Section 5).

3 APPROACH

3.1 Overview

Figure 2 illustrates the steps and components of our approach. First, GraphCode2Vec takes as input a Java program (i.e. a set of class files) that is converted to a Jimple intermediate representation (IR) [15]. Jimple is typed, based on a three-address code and provides 15 different operations; hence, it is easier to analyse and optimize than Java bytecode (with over 200 operations). Secondly, GraphCode2Vec employs Soot [60] to obtain the program dependence graph (PDG) by feeding the class files as input. From the resulting

Jimple representation and PDG, GraphCode2Vec learns two program embeddings, namely a lexical embedding and a dependence embedding. These two embeddings are ultimately concatenated to form the final code embedding.

To achieve *lexical embedding*, our approach first tokenizes the Jimple instructions obtained from our pre-processing step into subwords. Next, given the sub-words, our approach learns sub-word embedding using word2vec [42]. Then, it learns the instruction embedding by representing every Jimple instruction as a sequence of subwords embeddings using a bi-directional LSTM (BiLSTM, Section 3.2). The forward and backward hidden states of this BiLSTM allows to build the instruction embeddings. GraphCode2Vec employs a BiLSTM since it learns context better: BiLSTM can learn both past and future information while LSTM only learns past information. Finally, it aggregates multiple instruction embeddings using element-wise addition, in order to obtain the overall lexical program embedding.

To learn the *dependence embedding*, GraphCode2Vec applies a Graph Neural Network (GNN) [54] to embed Jimple instructions and their dependencies. Each node in the graph corresponds to a Jimple instruction and contains the (dependence) embedding of this instruction. Node attributes are from lexical embeddings. The edges of the graph represent the dependencies between instructions. Our approach considers the following program dependencies: data flow, control flow and method call graphs. GraphCode2Vec uses intra-procedural analysis [18] to extract data-flow and control-flow dependencies by invoking Soot [60]. Then, it builds method call graphs via class hierarchy analysis [11].

The training of GNNs is an iterative process where, at each iteration, the embedding of each node n is updated based on the embedding of the neighboring nodes (i.e., nodes connected to n) and the type of n's edges [70, 77]. The *message passing function* determines how to combine the embedding of the neighbors – also based on the edge types – and how to update the embedding n based on its current embedding and the combined neighbors' embedding. The dependence embedding of an instruction is the embedding of the corresponding node at the end of the training process.

Finally, after obtaining lexical embedding and dependence embedding, our approach concatenates both embeddings to obtain the overall program representation.

3.2 Lexical embedding

Step 1 - Jimple code tokenization: The first crucial step of GraphCode2Vec is to properly tokenize Jimple code into meaningful "tokens", to learn the vector representations. The traditional way to tokenize code is to split it on whitespaces. However, this manner is inappropriate for two reasons. First, whitespace-based tokenization often results in long tokens such as long method names (e.g., "getFunctionalInterfaceMethodSignature"). Long sequences often have a low frequency in a given corpus, which subsequently leads to an embedding of inferior quality. Second, whitespace-based tokenization is not able to process new words that do not occur in the training data – these out-of-vocabulary words are typically replaced by a dedicated "unknown" token. This is an obvious disadvantage for our approach, whose goal is to support practitioners to analyze diverse programs – which may then include words that did not occur in the programs used to learn the embedding.

To address this challenge, we tokenize the Jimple code into *subwords* [37, 56, 67], which are units shorter than words, e.g., morphemes. Subwords have been widely adopted in representation learning systems for texts [13, 25, 52, 76] as they solve the problem of overly long tokens and out-of-vocabulary words. New code programs can be smoothly handled using short tokens representation, by limiting the amount of long, but different tokens. Subwords get rid of the almost-infinite character combinations that are common in many program codes. For example, this is the reason why BERT uses wordpiece subwords [67], and XLNet [71] and T5 [52] use sentence-piece subwords. Similarly, GraphCode2Vec uses sentence-piece subwords. When using subwords, the long token "getFunctionalInterfaceMethodSignature" is split into "get", "Functional", "Interface", "Method" and "Signature". It is worth noting that most of the subwords are in fact words, e.g., "get" [31]. In this step, punctuation (e.g., semi-colon ";") is treated as a common character.

Step 2 - Subword embedding with word2vec: Given a subword-tokenized Jimple code corpus C with vocabulary size $|C|$, our approach learns a subword embedding matrix $\mathbf{E} \in \mathbb{R}^{|C| \times d}$ where d is a hyperparameter referring to the embedding dimension (d is usually set to 100). It uses the popular Skip-gram with negative sampling (SGNS) method in word2vec [42] to produce **E**. And **E** is utilized as the subword embedding matrix [42].

Step 3 - Instruction embedding: After forming the subword embeddings, GraphCode2Vec represents every Jimple instruction as a sequence of subword embeddings $(\mathbf{w}_0, \mathbf{w}_1, ..., \mathbf{w}_n)$, by using a bidirectional LSTM (BiLSTM). The role of BiLSTM is to learn the embedding of the instruction from the subword sequence of the instruction. Let $\overrightarrow{\mathbf{h}_t}$ and $\overleftarrow{\mathbf{h}_t}$ be the forward hidden state and backward hidden state of LSTM after feeding the final subword. Then, it forms the instruction embedding by concatenating $\overrightarrow{\mathbf{h}_t}$ and $\overleftarrow{\mathbf{h}_t}$, denoted as $\mathbf{x} = (\overrightarrow{\mathbf{h}_t}, \overleftarrow{\mathbf{h}_t})$.

Step 4 - Instruction embedding aggregation: The last step in the process of forming lexical embedding is the aggregation of the instruction embeddings in order to form the overall program lexical embedding. The reason why we aggregate instruction-level embedding as opposed to learning an embedding for the whole program is that LSTMs work with sequences of limited length and thus, truncate the instructions into small sequences (not exceeding the maximal length). After tokenization, a program can have many subwords and if one directly consider all subwords in the program, one needs to cut these subwords into the limited sequence length for LSTM and result in information loss. GraphCode2Vec uses element-wise addition as the instruction aggregation function. This operation allows for the aggregation of multiple instruction embeddings while keeping a limited vector length.

3.3 Dependence embedding

Step 1 - Building method graphs: A method graph is a tuple $G = (V, E, \mathbf{X}, \mathbf{K})$, where V is the set of nodes (i.e. Jimple instructions), E is the set of edges (dependence relations between the instructions), **X** is the node embedding matrix (which contains the embedding of the instructions) and **K** is the edge attribute matrix (which encodes the dependencies that exist between instructions). For each node n there is a column vector $\mathbf{x_n}$ in $\mathbf{X}$ such that $\mathbf{x_n} = (\overrightarrow{\mathbf{h}}_t, \overleftarrow{\mathbf{h}}_t)$ (instruction embedding).

To define E and $\mathbf{K}$, our approach extracts data-flow and control-flow dependencies by invoking Soot [18, 60]. Then, GraphCode2Vec introduces an edge between two nodes if and only if the two corresponding instructions share some dependence.

Step 2 - Building program graphs: A program graph consists of a pair $\mathcal{P} = (\mathcal{G}, \mathcal{R})$ where $\mathcal{G} = \{G_0, G_1, ..., G_m\}$ is a set of method graphs and where $\mathcal{R} \subseteq \mathcal{G}^2$ is the call relation between the methods, that is, $(G_i, G_j) \in \mathcal{R}$ if and only if the method that G_i represents calls the method that G_j represents. To represent this relation in the GNN, GraphCode2Vec introduces an entry node and an exit node for each method and edges linking those nodes with caller instructions.

Step 3 - Message passing function: The exact definition of the message passing function depends on the used GNN architecture. We choose the widely-used GNN architectures with linear complexity [68] that has been successfully applied in various application domains. GraphCode2Vec employs four GNN architectures, namely Graph Convolutional Network (GCN; Kipf and Welling [35]), GraphSAGE [24], Graph Attention Network (GAT; Veličković et al. [61]), Graph Isomorphism Network (GIN; Xu et al. [70]).

Step 4 - Learning the dependence embedding: The dependence embedding of each instruction is obtained by running the message passing function on all nodes for a pre-defined number of iterations, i.e., the number of GNN layers. Once these instruction embeddings have been produced, GraphCode2Vec aggregates them using the global attention pool operation [39] in order to produce the program-level dependence embedding. Attention mechanism can make program-level dependence embedding consider more important nodes (instructions).

The dependence embeddings that GNN produces depend on the learnable parameters of (a) the message passing function and (b) bidirectional LSTM from Section 3.2. These parameters can be automatically set to optimize the effectiveness of GraphCode2Vec either directly on the downstream task or on some pre-training objectives, as described hereafter.

In the end, to obtain the program embedding vector, our approach uses *feature fusion* as the concatenation operator that combines both lexical embedding and dependency embedding. Specifically, GraphCode2Vec combines the tensors of both features as one tensor via feature fusion to reduce information loss. Concatenation has been shown to be an effective method to fuse features without information loss when using DNN [20, 28, 38, 46], e.g., DenseNet [57] and U-Net [58]. Although the dependence embedding inherently encodes the lexical embedding, the importance of lexical inherently fades away as the semantic representation is learnt. Our ablation study (see RQ3 in Section 5) later reveals the benefits of concatenating an explicit lexical embedding with the dependence embedding.

3.4 Pre-training

Self-supervised learning has been applied with success for pre-training deep learning models [16, 40, 53]. It allows a model to learn how to perform tasks without human supervision [44, 78] by learning a universal embedding that can be fine-tuned to solve multiple downstream tasks. In this work, we employed three (3) self-supervised learning strategies to pre-train the BiLSTM and GNN in GraphCode2Vec, namely *node classification*, *context prediction* [27], and *variational graph encoding* (VGAE) [36]. Node (or instruction) classification trains the model to infer the type of Jimple instruction, given its embedding. Context prediction requires the model to predict a masked node representation, given its surrounding context. Variational graph encoding (VGAE) learns to encode and decode the code dependence graph structure. Note that these pre-training procedures *do not* require any human-labeled datasets. The model learns from the raw datasets without any human supervision.

4 EXPERIMENTAL SETUP

Research Questions: Our research questions (RQs) are designed to evaluate the *effectiveness* of GraphCode2Vec. In particular, we compare the *effectiveness* of GraphCode2Vec to the state-of-the-art in *task-specific* and *generic* code embedding methods (*see* RQ1 and RQ2). This is to demonstrate the utility of GraphCode2Vec in solving downstream tasks, in comparison to specialised learning-based approaches tailored towards solving specific SE tasks (RQ1) and other general-purpose code embedding approaches (RQ1). We also examine if GraphCode2Vec effectively embeds lexical and program dependence features in the latent space, and how this impacts its effectiveness on downstream tasks (*see* RQ3). The first goal of RQ3 is to demonstrate the validity of our approach, i.e., analyse that it indeed embeds lexical and dependence features as intended via *probing analysis*. In addition, we analyse the contribution of lexical embedding and dependence embedding to its effectiveness on downstream tasks by conducting an *ablation study*. We also investigate the *sensitivity* of our approach to the choices in GraphCode2Vec's framework, e.g., *model pre-training (strategy)* and *GNN configuration*. These experiments allow to evaluate the influence of these choices on the effectiveness of GraphCode2Vec.

Specifically, we ask the following research questions (RQs):

RQ1 Task-specific learning-based applications: Is our approach (GraphCode2Vec) effective in comparison to the state-of-the-art *task-specific* learning-based applications? What is the benefit of capturing semantic features in our code embedding?

RQ2 Generic Code embedding: How effective is our approach (GraphCode2Vec), in comparison to the state-of-the-art syntax-only generic code embedding approaches? What is the impact of capturing both syntactic and semantic features (i.e., program dependencies) in code embedding? How does GraphCode2Vec compare to GraphCodeBERT, a larger and more complex model?

RQ3 Further Analyses: What is the impact of model pre-training on the effectiveness of GraphCode2Vec? Does our approach effectively capture lexical and program dependence features? What is the contribution of lexical embedding or dependence embedding to the effectiveness of our approach on downstream tasks? Is our approach *sensitive* to the choice of GNN?

Baselines: We compare the effectiveness of GraphCode2Vec to several state-of-the-art code embedding approaches (aka *generic baselines*), and specialised or *task-specific learning-based applications*. On one hand, *generic baselines* refers to code embedding approaches that are designed to be general-purpose, i.e., they provide a code embedding that is amenable to address several downstream tasks. On the other hand, *task-specific* baselines refers to learning-based approaches that address a specific downstream SE task, e.g., patch classification. Table 2 provides details about these baselines for solution classification and patch classification. Specifically, we evaluated GraphCode2Vec in comparison to four (4) generic code embedding approaches, namely Code2Seq [2], Code2Vec [3], CodeBERT [17] and GraphCodeBERT [23] (*see* RQ2 in section 5). We have selected these generic baselines because they have been evaluated against several well-known state-of-the-art code embedding methods and demonstrated considerable improvement over them. Besides, these approaches are recent, popularly used and have been applied on many downstream (SE) tasks.

For task-specific learning-based approaches, we consider solution classification, and patch classification. These are popular SE downstream tasks that have been studied using learning-based approaches. We utilised three (3) specialised learning-based baseline for the solution classification task, namely CNNSentence [45], OneCNNLayer [50] and SequentialCNN [21]. We also used all four patch classifiers (Prophet [41], PatchSim [69], SimFeatures [63] and ODS [72]). These task-specific baselines have been selected because they have been shown to outperform other proposed learning-based approaches for these tasks. For instance, SequentialCNN [21] has been evaluated against five other learning-based approaches and demonstrated to be more effective. ODS [72] has also been shown to be more effective and efficient than the three other patch classifiers.

Subject Programs: In our experiments, we employed eight (8) subject programs written in Java. Table 3 provides details about each of our subject programs and their experimental usage. Notably, we employ four (4) publicly available programs for the downstream tasks, namely Defects4J [32], Java-Small [3], and Java250 [51]. Since we need the bytecode representation of each program, we utilise only programs that can be compiled. These datasets were employed for our comparative evaluation (*see* RQ1 and RQ2). We chose these datasets because they are popular and have been employed in the evaluation of our downstream tasks in previous studies [2, 51, 72, 74]. Besides, we employed Java-Small and Java250 in our ablation study where we evaluate the contribution of lexical and dependence embedding to the effectiveness of GraphCode2Vec (RQ3). We chose these two datasets for this task because they correspond to tasks that require lexical and semantic information to be effectively addressed. To further analyze GraphCode2Vec (*see* RQ3), we employed the Concurrency dataset [14, 19] and collected two (2) subject programs (named LeetCode-10 and M-LeetCode) from LeetCode[4]. We use these programs to investigate the difference between capturing lexical and dependence information. In particular, the Concurrency dataset contains different concurrent code types, which have similar syntactic/lexical features but different structure information. We mutated LeetCode-10 to create M-LeetCode dataset. Our mutation preserves lexical features, but modifies semantic or program dependence features such that LeetCode-10 and M-LeetCode have the same lexical features, but different semantics. For example, a simple dependence mutant involves switching outer and inner loops. We utilize LeetCode-10, M-LeetCode and Concurrency for the probing analysis of our approach (GraphCode2Vec).

Table 3: Details of Subject Programs

Subject Program	#Progs.	Tasks/Analyses
`Java-Small`	11	Method Name Prediction and Ablation Studies
`Java250`	75000	Solution Classification and Ablation Studies
`Defects4J`	15 & 5	Mutant Prediction and Patch Classification
`LeetCode-10`	100	Probing Analysis
`M-LeetCode`	100	Probing Analysis
`Concurrency`	46	Probing Analysis
`Jimple-Graph`	1976	Model Pre-training

Downstream Tasks: In our evaluation, we considered four (4) major software engineering tasks, namely, *mutant prediction*, *patch classification*, *method name prediction*, and *solution classification*. These are popular downstream SE tasks that have been investigated in the community for decades. For these four tasks, we evaluated GraphCode2Vec in comparison to four *generic baselines*, namely Code2Seq [2], Code2Vec [3], CodeBERT [17] and GraphCodeBERT [23]. Table 3 provides details on the subject programs employed for each downstream tasks. In the following, we provide further details about the experimental setup for each task evaluated in this paper.

Method Name Prediction: This refers to the task of predicting the method name of a function in a program, given a set of method names and the body of the function as inputs [6]. This task is useful for automatic code completion during programming. In our experiment, all four generic baselines were evaluated for this task. We evaluated this task using the Java-Small dataset, since it was designed for this task in previous studies [3] (*see* Table 3).

Solution Classification: This refers to the classification of source code into a predefined number of classes, e.g., based on the task it solves [50], or programming languages [21]. This is useful to assist or assess programming tasks and manage code warehouse. We evaluated all four generic baselines on this task, as well as three specialised learning-based approaches for this task, namely CNNSentence [45], OneCNNLayer [50], SequentialCNN [21] (Table 2). We evaluated this task using the Java250 dataset, which was designed for this task in previous studies [51] (*see* Table 3).

Patch Classification: For this task, the aim is to identify the correctness of patches, i.e., if a patch is (in)correct, wrong or overfitting [69, 72]. In our experiment, we compare the performance of GraphCode2Vec to the four generic baselines, as well as the current state-of-the-art learning-based approach for patch classification, i.e, ODS [72]. We employed the Defects4J [32] dataset (*see* Table 3) which has also been used by previous studies for this task [69, 72]. The goal of this task is to identify over-fitting APR patches. We used five (5) programs and 890 APR patches[5] containing 643 over-fitting patches and 247 correct patches.

Mutant Prediction: The goal of this task is to predict different types of mutants employed during mutation testing. Mutation testing is an important SE task that is typically deployed to determine the adequacy of a test suite to expose injected faults in a program [47].

[4] https://leetcode.com/

[5] We exempted 12 patches out of the 902 patched programs used by ODS, since they deleted complete functions, and there is no code representation for deleted functions.

In this work, we predict if a mutant is *killable* or *live* [8]. To this end, we employ the Defects4J [32] dataset (*see* Table 3) which has been popularly employed for several SE tasks, including mutation testing [48]. We curated a mutant prediction dataset containing 15 Java programs, and 16,216 mutants.

Pre-training Setup: For model pre-training, we curated the `Jimple-Graph` dataset from the Maven repository[6], it contains 1,976 Java libraries with about 3.5 millions methods in total. We randomly sample around 10% data for the pre-traning purpose. These Java libraries are from 42 application domains, this ensures a reasonable program diversity, these domains include math and image processing libraries. For the BiLSTM component (Section 3.2), we use one layer with hidden dimension size 150. We pre-train sub-tokens using the `Jimple` text for each program, the sub-token embedding dimension is set to 100 (*see* Section 3). We fine-tune the downstream tasks using the obtained pre-trained weights after one epoch. All GNNs use five (5) layers with dropout ratio 0.2. We use Adam [34] optimizer with 0.001 learning rate. In our experiment, we evaluated all three (3) pre-training strategies (Section 3.4).

Metrics and Measures: For all tasks, we report F1-score, precision and recall. We discuss most of our results using F1-score since it is the harmonic mean of precision and recall. Besides, it is a better measurement metric than accuracy, especially when the dataset is imbalanced (e.g., Java-Small). Hence, we do not report the accuracy for imbalanced datasets, e.g., mutant data is imbalanced with about 30% live mutants and 70% killable mutants. We provide the code details in the Github repository[7].

Probing Analysis: The goal of our probing analysis is to ensure that lexical and dependence features are indeed learned by GraphCode2Vec's code embedding. Probing is a widely used technique to examine an embedding for desired properties [9, 53, 75]. To this end, we trained diagnostic classifiers to probe GraphCode2Vec's code embedding for our desired properties (i.e., lexical and/or program dependence features). Concretely, we train a simple classifier with one MLP layer fed with the learned code embedding (e.g. lexical) to examine if our code embedding encodes the desired property. To achieve this, we curated a dedicated dataset for training and evaluating our probing classifiers. Specifically, we employ three probing datasets, namely LeetCode-10, M-LeetCode and Concurrency (Table 3). We have employed these datasets because they require lexical or dependence embedding to address their corresponding tasks.

Probing Task Design: We design four probing tasks. The first three (Task-1, Task-2 and Task-3) use LeetCode-10 and M-LeetCode, and the last one (Task-4) uses Concurrency. *Task-1* classifies what problem the solution code solves on LeetCode-10. LeetCode-10 shares lexical token similarities within one problem group, and some solutions from the different problem groups may have the same semantic structure, e.g., using one for-loop. Therefore, we hypothesize that the lexical embedding is more informative than the semantic embedding for Task-1. *Task-2* mixes LeetCode-10 and M-LeetCode, and then judges which dataset the input code is from (binary classification). LeetCode-10 and M-LeetCode share lots of similar lexical tokens but the code semantic structures are different. Hence, the semantic embedding should be more informative than the code lexical syntactic embedding. *Task-3* also mixes the two datasets but uses all the 20 labels instead of a binary classification. Task-3 integrates Task-1 and Task-2, requiring both lexical and semantic information. *Task-4* is a concurrency bug classification task. The code with same label can have the high lexical similarity but the code semantic structure should be different.

GraphCode2Vec's Configuration: We employ three (3) pre-training strategies, namely node classification, context prediction and VGAE. Our approach supports four (4) GNN architectures for dependence embedding (*see* Section 3), namely GCN [35], GraphSAGE [24], GAT [61] and GIN [70]. In total, we have 12 possible configurations. However, the *default configuration is context prediction for pre-training and dependence embedding with GAT architecture.* In our experiments, we evaluate the effect of each configuration on the effectiveness of our approach (*see* Section 5). For method name prediction, we used the default data splitting from the public Java-Small dataset. For patch classification task, we used the 10-fold cross validation described in ODS [72]. For all other experiments, we keep 10-20% of the dataset as test data.

Implementation Details and Platform: GraphCode2Vec was implemented in about 4.8 KLOC of Python code, using the Pytorch ML framework. Our data processing and evaluation code is about 3 KLOC of Java code. We use Soot [60] to extract the program dependence graph (PDG). We reuse the code from the public repository of each baseline in our experiments.[8] However, we adapt each baseline to our downstream tasks, e.g., by replacing the classifier but using the same performance metrics. All experiments were conducted on a Tesla V100 GPU server, with 40 CPUs (2.20 GHz) and 256G of main memory. The implementation of GraphCode2Vec is available online[9].

5 EXPERIMENTAL RESULTS

RQ1 Task-specific learning-based applications: This experiment examines how GraphCode2Vec compares to seven (7) state-of-the-art task-specific learning-based techniques for *solution classification* and *patch classification*. We selected these two tasks for this experiment due to their popularity, availability of ML-based baselines and their application to vital SE tasks, e.g., automated program repair, patch validation, code evolution, and software warehousing. We evaluated against three solution classifiers, namely CNNSentence [45], OneCNNLayer [50], SequentialCNN [21]. We also compare GraphCode2Vec to four patch classifiers – Prophet [41], PatchSim [69], SimFeatures [63] and ODS [72].

Our evaluation results show that *GraphCode2Vec outperforms the state-of-the-art task-specific learning based approaches for the tested tasks, i.e., patch classification, and solution classification.* Table 5 highlights the effectiveness of GraphCode2Vec in comparison to learning-based approaches for patch classification and solution classification, respectively. In particular, GraphCode2Vec outperforms all seven task-specific baselines in our evaluation. GraphCode2Vec outperforms all three baselines for solution classification, it is almost twice as effective as SequentialCNN and OneCNNLayer, and 40% more effective than the best baseline – CNNSentence (*see*

[6] https://mvnrepository.com/
[7] https://github.com/graphcode2vec/graphcode2vec
[8] https://github.com/tech-srl/code2vec,https://github.com/tech-srl/code2seq, https://github.com/microsoft/CodeBERT, https://github.com/hukuda222/code2seq
[9] https://github.com/graphcode2vec/graphcode2vec

Table 4: Effectiveness of GraphCode2Vec vs. Syntax-only Generic Code Embedding approaches. The best results are in bold text, the results for the best-performing baseline are in *italics*. We report the improvement in effectiveness between GraphCode2Vec and the best-performing baseline in "% Improvement", improvements above five percent (>5%) are in bold text.

Generic Code	Method Name Prediction			Solution Classification			Mutant Prediction			Patch Classification		
Embedding	F1	Preci	Recall	F1	Preci	Recall	F1	Preci	Recall	F1	Preci	Recall
Code2Seq	*0.4920*	*0.5963*	0.4187	0.7542	0.7678	0.7536	0.5911	0.6423	0.5881	0.8901	0.8355	*0.9541*
Code2Vec	0.3309	0.3779	0.2943	0.8034	0.8081	0.8028	0.6398	0.6632	0.6320	0.8787	0.8806	0.8782
CodeBERT	0.3963	0.3295	*0.4969*	*0.8783*	*0.8747*	*0.8878*	*0.7106*	*0.7305*	*0.6995*	*0.9275*	*0.9099*	0.9473
GraphCode2Vec	**0.5807**	**0.6150**	**0.5502**	**0.9746**	**0.9753**	**0.9746**	**0.7542**	**0.7569**	**0.7524**	**0.9359**	**0.9145**	**0.9602**
% Improvement	**18.03%**	3.14%	**10.73%**	**10.96%**	**11.50%**	**9.78%**	**6.14%**	3.61%	**7.56%**	0.91%	0.51%	0.64%

Table 5: Effectiveness of GraphCode2Vec (aka "Graph.") vs. Task-Specific learning-based approaches for two SE tasks. The best results are in bold text, the results for the second best-performing approach are in *italics*. The improvement in effectiveness between GraphCode2Vec and the best-performing baseline is reported in "Graph. *(% Improv.)*".

	Solution Classification				Patch Classification				
	CNN Sen.	One CNN.	Seq.-CNN	Graph. *(% Improv.)*	SimFeatures	Prophet	Patch-Sim	ODS	Graph. *(% Improv.)*
F1-Score	*0.690*	0.540	0.470	**0.970 (40.6%)**	0.881	0.892	0.881	*0.900*	**0.915 (1.7%)**
Recall	*0.690*	0.540	0.470	**0.970 (40.6%)**	0.895	0.891	0.389	*0.950*	**0.960 (2.1%)**
Precision	*0.700*	0.550	0.480	**0.970 (38.6%)**	0.870	0.889	0.830	*0.924*	**0.936 (1.3%)**

Table 5). In addition, GraphCode2Vec outperforms all four state of the art patch classifiers, i.e., ODS [72], Prophet [41]), PatchSim [69] and SimFeatures [63]. It is at least twice as effective as PatchSim (in terms of recall) and slightly (up to 2%) more effective than the best baseline, i.e., ODS (*see* Table 5). This result demonstrates the utility of our approach in addressing both downstream tasks. Furthermore, it highlights the effectiveness of generic code embedding in comparison to specialised learning-based approaches. This superior performance can be attributed to the fact that GraphCode2Vec is generic, and it employs self-supervised model pre-training.

GraphCode2Vec is up to two times (2x) more effective than the seven (7) state-of-the-art task-specific approaches, for both tasks.

RQ2 Generic Code embedding: In this experiment, we demonstrate how GraphCode2Vec compares to the state-of-the-art generic code embedding approaches. We thus, compare the effectiveness of GraphCode2Vec with three (3) syntax-only generic baselines, namely CodeBERT, Code2Seq and Code2Vec. Additionally, we compare the effectiveness of our approach to a a larger and more complex state-of-the-art generic approach that captures both syntax and semantics, specifically, GraphCodeBERT. We used four (4) downstream SE tasks – method name prediction, solution classification, mutant prediction and patch classification.

Syntax-only Generic Embedding: In our evaluation, we found that *our approach (GraphCode2Vec) outperforms all syntax-based generic baselines for all tasks.* Table 4 highlights the effectiveness of GraphCode2Vec in comparison to the baselines (i.e., Code2seq, Code2Vec and CodeBERT). As an example, consider method name prediction, GraphCode2Vec is twice as effective as some baselines, e.g., Code2Vec. For all (four) tasks, GraphCode2Vec clearly outperforms all baselines across all metrics. It is up to 12% and 18% more effective than the best baselines, CodeBERT and Code2Seq, respectively. We observed CodeBERT is the best baseline on three tasks. We attribute the performance of CodeBERT on these tasks to its much higher complexity (i.e., huge number of trainable parameters, more than 124M) and the size of the pre-training dataset (8.5M) [29]. Overall, our results demonstrate that including semantic program features improves the performance of code representation across these downstream tasks. Thus, emphasizing the importance of semantic features in addressing SE tasks, especially the need to capture program dependencies in code representation.

For all (four) tasks, GraphCode2Vec is (up to 18%) more effective than (the best) syntax-only baselines.

Complementarity with GraphCodeBERT: We also observe that despite the lower complexity of our approach (GraphCode2Vec), *it is comparable and complementary to GraphCodeBERT* across tested tasks. GraphCodeBERT captures both syntactic and semantic program features but, it is significantly larger and complex than GraphCode2Vec. Table 6 highlights the complexity and effectiveness of GraphCodeBERT in comparison to GraphCode2Vec. For instance, GraphCodeBERT has at least 50 times (50x) as many trainable parameters as GraphCode2Vec (124 million versus 2.8 million parameters), and seven times (7x) as much pre-training data (2.3M versus 314K methods). Despite the difference in size and complexity, GraphCodeBERT has a comparable performance to GraphCode2Vec. Specifically, GraphCode2Vec outperforms GraphCodeBERT on two tasks (method name prediction and patch classification) and it is comparable on the other two tasks (solution classification, and mutant prediction). Notably, GraphCodeBERT has a negligible improvement over GraphCode2Vec for these two tasks (about 1%). These results demonstrate that although simpler and trained on 7 times less data, GraphCode2Vec is complementary to GraphCodeBERT. This disparity in size and complexity implies that precise program dependence information is important. Nevertheless, our results show that both GraphCode2Vec and GraphCodeBERT are more effective than syntax-only approaches, e.g., CodeBERT (*cf.* Table 5 and Table 6).

GraphCode2Vec is complementary to GraphCodeBERT despite being simpler and trained on seven times (7x) less data. It is more effective on two tasks, and comparable on the other two tasks.

RQ3 Further Analyses: The goal of this research question is to examine the impact of *model pre-training* on improving GraphCode2Vec's effectiveness on downstream tasks. We also investigate if GraphCode2Vec effectively captures lexical and/or semantic

Table 6: Effectiveness of GraphCode2Vec vs. GraphCodeBERT. Lower complexity, the best results and higher improvements (above five percent (>5%)) are in bold text.) are in bold text.

Generic Code	Model	Pretrain	Method Name Prediction			Solution Classification			Mutant Prediction			Patch Classification		
Embedding	Size	Data	F1	Preci	Recall	F1	Preci	Recall	F1	Preci	Recall	F1	Preci	Recall
GraphCodeBERT	124M	2.3M	0.5761	**0.7261**	0.4775	**0.9850**	**0.9868**	**0.9843**	**0.7649**	**0.768**	**0.7623**	0.9317	0.9108	0.9557
GraphCode2Vec	**2.8M**	**314K**	**0.5807**	0.6150	**0.5502**	0.9746	0.9753	0.9746	*0.7542*	*0.7569*	*0.7524*	**0.9359**	**0.9145**	**0.9602**
% Improvement	**50X**	**7X**	**7.99%**	-15.30%	**15.23%**	-1.07%	-1.17%	-0.18%	-1.40%	-1.45%	-1.30%	0.45%	0.41%	0.47%

Table 7: Probing Analysis results showing the accuracy for all pre-training strategies and GNN configurations. Best results for each sub-category are in bold, and the better results between syntactic (lexical) embedding and semantic embedding is in *italics.* "syn+sem" refers to GraphCode2Vec's models capturing both syntactic and semantic features.

Pre-training	Captured	**Task-1** (syntax-only)				**Task-2** (semantic-only)				**Task-3** (syntax and semantic)				**Task-4** (semantic-only)			
Strategy	Feature	GCN	GIN	GSAGE	GAT	GCN	GIN	GSAGE	GAT	GCN	GIN	GSAGE	GAT	GCN	GIN	GSAGE	GAT
Context	**semantic**	0.822	0.674	0.842	0.886	***0.684***	*0.614*	***0.704***	*0.741*	0.513	0.381	*0.543*	***0.612***	***0.654***	***0.666***	***0.657***	***0.594***
	syntactic	***0.934***	***0.938***	*0.942*	*0.928*	0.615	0.602	0.617	0.602	*0.529*	*0.527*	*0.528*	0.527	0.580	0.525	0.524	0.449
	syn+sem	0.918	0.928	**0.95**	**0.942**	0.641	**0.641**	0.688	**0.797**	**0.559**	**0.546**	**0.587**	0.6	0.605	0.592	0.608	0.592
Node	**semantic**	0.758	0.820	0.802	0.840	***0.651***	***0.667***	***0.741***	***0.686***	0.426	***0.514***	***0.625***	***0.563***	***0.647***	***0.664***	***0.659***	***0.670***
	syntactic	***0.904***	*0.884*	***0.876***	***0.916***	0.584	0.587	0.606	0.593	*0.516*	0.504	0.490	0.513	0.484	0.476	0.420	0.550
	syn+sem	0.872	**0.9**	**0.876**	0.902	0.624	0.618	0.691	0.67	**0.522**	0.508	0.572	0.545	0.519	0.522	0.451	0.57
VGAE	**semantic**	0.856	0.812	0.868	0.866	***0.594***	***0.653***	0.583	***0.617***	0.403	*0.532*	0.407	0.477	***0.673***	***0.680***	***0.674***	***0.656***
	syntactic	*0.916*	***0.932***	***0.928***	***0.950***	0.591	0.572	***0.594***	0.599	*0.485*	0.494	*0.492*	***0.495***	0.523	0.617	0.584	0.591
	syn+sem	**0.92**	0.926	**0.928**	0.938	0.59	0.63	0.591	0.596	**0.498**	**0.548**	**0.508**	0.492	0.627	0.658	0.531	0.586
Best Config.		**Syntactic = 8/12**				**Semantic = 9/12**				**Syntactic + Semantic = 7/12**				**Semantic = 12/12**			

Table 8: Effectiveness (F1-Score) of GraphCode2Vec on all GNN configurations and Pre-training Strategies, for all downstream tasks. For each subcategory, the best results for each category are in bold text.

	Method Name Prediction					Solution Classification				
GNN	No Pre-	Pre-training Strategies				No Pre-	Pre-training Strategies			
	training	Context	Node	VGAE	Average	training	Context	Node	VGAE	Average
GCN	0.4494	0.5018	0.4859	0.5337	0.4930	0.9679	0.9710	0.9710	0.9751	0.9712
GIN	0.4347	0.4684	0.4037	0.5266	0.4584	0.9645	0.9711	0.9700	0.9710	0.9692
GraphSage	0.3998	0.5006	0.4531	0.5412	0.4736	0.9675	0.9712	0.9721	0.9727	0.9709
GAT	0.4246	0.5807	0.6194	0.5890	0.5534	0.9647	0.9746	0.9703	0.9735	0.9708
Average	0.4271	0.5129	0.4905	**0.5476**		0.9662	0.9720	0.9718	**0.9731**	
Variance	0.0003	0.0017	**0.0064**	0.0006		2.2e-6	**2.2e-6**	7.1e-7	2.3e-6	
SD	0.0180	**0.0413**	0.0800	0.0244		**0.0015**	**0.0015**	0.0008	**0.0015**	

Table 9: Ablation Study results showing the F1-Score of GraphCode2Vec. Best results are bold.

Pre-training	Captured	Method Name Prediction				Solution Classification			
Strategy	Feature	GCN	GIN	GSAGE	GAT	GCN	GIN	GSAGE	GAT
Context	**semantic**	**0.5454**	**0.4674**	**0.5038**	**0.6082**	**0.9698**	**0.9649**	**0.9682**	**0.9740**
	syntactic	0.4575	0.4500	0.4644	0.4381	0.9614	0.9560	0.9588	0.9610
Node	**semantic**	**0.4843**	**0.4136**	**0.4404**	**0.5888**	**0.9738**	**0.9711**	**0.9696**	**0.9704**
	syntactic	0.3800	0.3845	0.3660	0.3560	0.9563	0.9562	0.9572	0.9595
VGAE	**semantic**	**0.5988**	**0.4786**	0.3675	**0.5464**	**0.9725**	**0.9663**	**0.9671**	**0.9711**
	syntactic	0.3922	0.4053	**0.3936**	0.4058	0.9711	0.9659	0.9626	0.9705
Best config.		**Semantic = 11/12**				**Semantic = 12/12**			

program feature(s). We employ *probing analysis* to analyze if pre-trained GraphCode2Vec models learn the lexical and semantic features required for feature-specific tasks, i.e, that require capturing either or both features to be well-addressed. For instance, Task-4 is the concurrency classification task requiring semantic features. In addition, we conduct an *ablation study* to investigate how the syntactic and semantic information captured by GraphCode2Vec influence its effectiveness on downstream tasks. Finally, we evaluate the *sensitivity of our approach* to the selected *GNN*.

Model Pre-training: We examine if the three pre-training strategies improve the effectiveness of GraphCode2Vec on downstream tasks, using two downstream tasks and all three pre-training strategies (node, context and VGAE) (*see* Table 8).

We found that *model pre-training improves the effectiveness of GraphCode2Vec across all tasks.* Pre-training improves its effectiveness by up to 28%, on average. For instance, consider model pre-training with VGAE strategy for method name prediction (*see* Table 8). This result implies that model pre-training improves the effectiveness of GraphCode2Vec on downstream SE tasks.

> *Model pre-training improves the effectiveness of GraphCode2Vec (by up to 28%, on average) across all tasks.*

Probing Analysis: Let us examine if our pre-trained code embedding indeed encodes the desired lexical and semantic program features. To achieve this, we use the lexical embedding and semantic embedding from GraphCode2Vec's pre-training as inputs for probing. In this probing analysis, only the classifier is trainable and GraphCode2Vec is frozen and non-trainable. We use one

MLP-layer classifier to evaluate these models on four tasks, Task-1 requires only lexical/syntactic information. However, Task-2 and Task-4 require only semantic information (program dependence). Finally, Task-3 subsumes tasks one and two, such that it requires both syntactic and semantic information.

Our evaluation results show that *GraphCode2Vec's pre-trained code embedding mostly captures the desired lexical and semantic program features for all tested tasks, regardless of the pre-training strategy or GNN configuration.* Table 7 highlights the effectiveness of each frozen pre-trained model for each task, configuration and pre-training strategy. Notably, the frozen pre-trained model performed best for the desired embedding for each task in three-quarters (36/48=75%) of all tested configurations. As an example, for tasks requiring semantic information (Task-2 and Task-4), our pre-trained model encoding only semantic information performed best for 88% of all configurations (21/24 cases). This result demonstrates that GraphCode2Vec effectively encodes either or both syntactic and semantic features, this is evidenced by the effectiveness of models encoding desired feature(s) for feature-specific tasks.

GraphCode2Vec effectively encodes the syntactic and/or semantic features, feature-specific models performed best in 75% of cases.

Ablation Study: We investigate the impact of syntactic/lexical embedding and semantic/dependence embedding on addressing downstream tasks. Using method name prediction and solution classification, we examine how removing lexical embedding or dependence embedding during the fine-tuning of GraphCode2Vec's pre-trained model impacts the effectiveness of the approach.

Our results show that *GraphCode2Vec's dependence embedding is important to effectively address our downstream SE tasks.* Table 9 presents the ablation study results. In particular, results show that models fine-tuned with only semantic information outperformed those fine-tuned with syntactic features in almost all (23/24 = 96% of) cases. This result demonstrates the effectiveness of dependence embedding in addressing downstream SE tasks.

Results show that dependence/semantic embedding is vital to the effectiveness of GraphCode2Vec on downstream SE tasks.

GNN Sensitivity: This experiment evaluates the sensitivity of our approach to the choice of GNN. Table 8 provides details of the GNN sensitivity analysis, tasks and GNN configurations. To evaluate this, we compute the *variance* and *standard deviation (SD)* of the effectiveness of GraphCode2Vec when employing different GNNs.

Our evaluation results show that *GraphCode2Vec is stable, it is not highly sensitive to the choice of GNN.* Table 8 shows the details of the SD and variance of our approach for each GNN configuration. Across all tasks, the variance and SD of the GraphCode2Vec is mostly low, it is maximum 0.0064 and 0.0413, respectively.

GraphCode2Vec is stable across GNN configurations, the variance and SD of its effectiveness are very low for all configurations.

6 THREATS TO VALIDITY

External Validity: This refers to the generalizability of our approach and results, especially beyond our data sets, tasks and models. For instance, there is a threat that GraphCode2Vec does not generalize to other (SE) tasks and other Java programs. To mitigate this threat, we have evaluated GraphCode2Vec using mature Java programs with varying sizes and complexity (*see* Table 3), as well as downstream tasks with varying complexities and requirements.
Internal Validity: This threat refers to the correctness of our implementation, if we have correctly represented lexical and semantic features in our code embedding. We mitigate this threat by evaluating the validity of our implementation with probing analysis and ablation studies (*see* Section 5). We have also compared GraphCode2Vec to 7 baselines using four (4) major downstream tasks. In addition, we have conducted further analysis to test our implementation using different pre-training strategies and GNN configurations. We also provide our implementation, (pre-trained) models and experimental data for scrutiny, replication and reuse.
Construct Validity: This is the threat posed by our design/implementation choices and their implications on our findings. Notably, our choice of intermediate code representation (i.e., Jimple) instead of source code implies that our approach lacks natural language text (such as code comments) in the (pre-)training dataset. Indeed, GraphCode2Vec would not capture this information as it is. However, it is possible to extend GraphCode2Vec to also capture natural language text. This can be achieved by performing lexical and program dependence analysis at the source code level.

7 CONCLUSION

In this paper, we have proposed GraphCode2Vec, a novel and generic code embedding approach that captures both syntactic and semantic program features. We have evaluated it in comparison to the state-of-the-art generic code embedding approaches, as well as specialised, task-specific learning based applications. Using seven (7) baselines and four (4) major downstream SE tasks, we show that GraphCode2Vec is stable and effectively applicable to several downstream SE tasks, e.g., patch classification and solution classification. Moreover, we show that it indeed captures both lexical and dependency features, and we demonstrate the importance of generically embedding both features to solve downstream SE tasks.

In the future, we plan to address certain limitations of GraphCode2Vec. GraphCode2Vec does not capture dynamic information (e.g., program execution) and textual source code details (e.g., comments), both of which are important for some SE tasks. To address the lack of dynamic code embedding, we plan to investigate how GraphCode2Vec can effectively capture run-time information (such as program traces or code coverage) so it is applicable to SE tasks that are dependent on dynamic information (e.g., testing, debugging and program repair). Additionally, we plan to extend GraphCode2Vec to account for textual source code information such that it is amenable to tasks requiring such features (e.g., program search and code completion).

To encourage replication and reuse, we provide our prototypical implementation of GraphCode2Vec and our experimental data:

https://github.com/graphcode2vec/graphcode2vec

ACKNOWLEDGMENTS

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C17/IS/11686509/CODEMATES.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. arXiv:1808.01400 [cs.LG]

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[4] Richard E Bellman. 2015. *Adaptive control processes*. Princeton university press.

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *arXiv preprint arXiv:1806.07336* (2018).

[6] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1186–1197.

[7] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641* (2020).

[8] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting fault revealing mutants. *Empir. Softw. Eng.* 25, 1 (2020), 434–487. https://doi.org/10.1007/s10664-019-09778-7

[9] Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. 2018. What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Melbourne, Australia, 2126–2136. https://doi.org/10.18653/v1/P18-1198

[10] Chris Cummins, Hugh Leather, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, and Michael O'Boyle. 2020. Deep Data Flow Analysis. *arXiv preprint arXiv:2012.01470* (2020).

[11] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 77–101.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[14] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.

[15] Arni Einarsson and Janus Dam Nielsen. 2008. A survivor's guide to Java program analysis with soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark* 17 (2008).

[16] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. 2010. Why does unsupervised pre-training help deep learning?. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 201–208.

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

[18] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[19] Abel Garcia and Cosimo Laneve. 2017. JaDA–the Java deadlock analyser. *Behavioural Types: from Theories to Tools* (2017), 169–192.

[20] Sahar Ghannay, Benoit Favre, Yannick Esteve, and Nathalie Camelin. 2016. Word embedding evaluation and combination. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. 300–305.

[21] Shlok Gilda. 2017. Source code classification using Neural Networks. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 1–6.

[22] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. *CoRR* abs/1704.01212 (2017). arXiv:1704.01212 http://arxiv.org/abs/1704.01212

[23] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[24] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[25] Benjamin Heinzerling and Michael Strube. 2018. BPEmb: Tokenization-free Pre-trained Subword Embeddings in 275 Languages. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. European Language Resources Association (ELRA), Miyazaki, Japan. https://aclanthology.org/L18-1473

[26] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 518–529. https://doi.org/10.1145/3377811.3380361

[27] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. 2019. Strategies for pre-training graph neural networks. *arXiv preprint arXiv:1905.12265* (2019).

[28] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[29] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

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

[31] Dan Jurafsky and James H. Martin. 2021. *Speech & language processing*.

[32] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[33] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*. PMLR, 5110–5121.

[34] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[35] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[36] Thomas N Kipf and Max Welling. 2016. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308* (2016).

[37] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Brussels, Belgium, 66–71. https://doi.org/10.18653/v1/D18-2012

[38] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. 2016. Fractalnet: Ultra-deep neural networks without residuals. *arXiv preprint arXiv:1605.07648* (2016).

[39] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[40] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[41] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

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

[43] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[44] T Nathan Mundhenk, Daniel Ho, and Barry Y Chen. 2018. Improvements to context based self-supervised learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 9339–9348.

[45] Hiroki Ohashi and Yutaka Watanobe. 2019. Convolutional neural network for classification of source codes. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 194–200.

[46] Oyebade K Oyedotun and Djamila Aouada. 2020. Why do Deep Neural Networks with Skip Connections and Concatenated Hidden Representations Work?. In *International Conference on Neural Information Processing*. Springer, 380–392.

[47] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[48] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and

Mark Harman (Eds.). ACM, 537–548. https://doi.org/10.1145/3180155.3180183

[49] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could Neural Networks understand Programs? *arXiv preprint arXiv:2105.04297* (2021).

[50] Ádám Pintér and Sándor Szénási. 2018. Classification of source code solutions based on the solved programming tasks. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 000277–000282.

[51] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *arXiv preprint arXiv:2105.12655* (2021).

[52] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[53] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A Primer in BERTology: What We Know About How BERT Works. *Transactions of the Association for Computational Linguistics* 8 (2020), 842–866. https://doi.org/10.1162/tacl_a_00349

[54] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[55] Cedric Seger. 2018. An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing.

[56] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

[57] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[58] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[59] Siddhaling Urolagin, KV Prema, and NV Subba Reddy. 2011. Generalization capability of artificial neural network incorporated with pruning method. In *International Conference on Advanced Computing, Networking and Security*. Springer, 171–178.

[60] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[61] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[62] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. 2020. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–27.

[63] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 968–980.

[64] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[65] Wenhan Wang, Kechi Zhang, Ge Li, and Zhi Jin. 2020. Learning to Represent Programs with Heterogeneous Graphs. *arXiv preprint arXiv:2012.04188* (2020).

[66] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98.

[67] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[68] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[69] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering*. 789–799.

[70] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[71] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf

[72] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering* (2021).

[73] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[74] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 141–151. https://doi.org/10.1145/3236024.3236068

[75] Mengjie Zhao, Philipp Dufter, Yadollah Yaghoobzadeh, and Hinrich Schütze. 2020. Quantifying the Contextualization of Word Representations with Semantic Class Probing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1219–1234. https://doi.org/10.18653/v1/2020.findings-emnlp.109

[76] Mengjie Zhao and Hinrich Schütze. 2019. A Multilingual BPE Embedding Space for Universal Sentiment Lexicon Induction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3506–3517. https://doi.org/10.18653/v1/P19-1341

[77] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.

[78] Andrew Zisserman. 2018. Self-Supervised Learning. https://project.inria.fr/paiss/files/2018/07/zisserman-self-supervised.pdf.