The Journal of Systems & Software 180 (2021) 111011

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A hybrid code representation learning approach for predicting method names☆

Fengyi Zhang, Bihuan Chen *, Rongfan Li, Xin Peng

School of Computer Science, Fudan University, China
Shanghai Key Laboratory of Data Science, Fudan University, China

ARTICLE INFO

Article history:
Received 18 November 2020
Received in revised form 14 April 2021
Accepted 20 May 2021
Available online 27 May 2021

Keywords:
Code representation learning
Method name prediction
Deep learning

ABSTRACT

Program semantic properties such as class names, method names, and variable names and types play an important role in software development and maintenance. Method names are of particular importance because they provide the cornerstone of abstraction for developers to communicate with each other for various purposes (e.g., code review and program comprehension). Existing method name prediction approaches often represent code as lexical tokens or syntactical AST (abstract syntax tree) paths, making them difficult to learn code semantics and hindering their effectiveness in predicting method names. Initial attempts have been made to represent code as execution traces to capture code semantics, but suffer scalability in collecting execution traces.

In this paper, we propose a hybrid code representation learning approach, named Meth2Seq, to encode a method as a sequence of distributed vectors. Meth2Seq represents a method as (1) a bag of paths on the program dependence graph, (2) a sequence of typed intermediate representation statements and (3) a sentence of natural language comment, to scalably capture code semantics. The learned sequence of vectors of a method is fed to a decoder model to predict method names. Our evaluation with a dataset of 280.5K methods in 67 Java projects has demonstrated that Meth2Seq outperforms the two state-of-the-art code representation learning approaches in F1-score by 92.6% and 36.6%, while also outperforming two state-of-the-art method name prediction approaches in F1-score by 85.6% and 178.1%.

1. Introduction

Representation learning (Bengio et al., 2013) transfers raw data into low dimensional vectors (i.e., *embeddings*) that preserve the properties of raw data and can be fed into downstream deep learning models. It allows the model to learn how to extract the features and how to use them in specific tasks. For example, word embedding (Mikolov et al., 2013b,a; Pennington et al., 2014) and paragraph embedding (Le and Mikolov, 2014) are the two most widely adopted representation learning approaches in natural language processing (NLP), and significantly enable the recent success and advance in NLP tasks. Basically, they learn continuous fixed-length vector representations of words, sentences, paragraphs and documents from a text corpus so that the vectors of semantically similar words, sentences, paragraphs and documents are close to each other in the vector space.

With the availability of a massive collection of code corpus from open-source repository hosting platforms (e.g., GitHub), leveraging machine learning models for various programming language and software engineering tasks has recently become a trending topic of much interest (Yahav, 2015; Bielik et al., 2015; Ernst, 2017; Allamanis et al., 2018b). One of the programming comprehension tasks is to predict program semantic properties such as class names, method names, variable names and variable types. This task is very important because such semantic properties or labels are common and important during software development and maintenance. On one hand, developers must select meaningful semantic names for every class, method, variable and parameter they declare in the program. On the other hand, developers often rely on names to understand behaviors of program or API elements during code review or maintenance. However, predicting program semantic properties is a non-trivial task; and poor semantic labels hinder code comprehension and maintenance (Arnaoudova et al., 2016; Lawrie et al., 2006b; Liblit et al., 2006; Takang et al., 1996; Binkley et al., 2013; Butler et al., 2011), or even lead to program bugs (Butler et al., 2009; Abebe et al., 2012; Butler et al., 2010).

☆ Editor: Earl Barr.

* Corresponding author at: School of Computer Science, Fudan University, China.
E-mail addresses: 18110240048@fudan.edu.cn (F. Zhang), bhchen@fudan.edu.cn (B. Chen), 18212010062@fudan.edu.cn (R. Li), pengxin@fudan.edu.cn (X. Peng).

https://doi.org/10.1016/j.jss.2021.111011
0164-1212/

Existing machine learning-based approaches for predicting program semantic properties (Allamanis and Sutton, 2013; Allamanis et al., 2015a, 2016; Liu et al., 2019; Cvitkovic et al., 2019; Allamanis et al., 2018; Alon et al., 2018, 2019b,a; Wang and Su, 2019; Bavishi et al., 2018; Hellendoorn et al., 2018; Malik et al., 2019) can be categorized into three groups according to how they represent the code, i.e., token-based (Allamanis and Sutton, 2013; Allamanis et al., 2015a, 2016; Liu et al., 2019; Bavishi et al., 2018; Hellendoorn et al., 2018; Malik et al., 2019), abstract syntax tree (AST)-based (Cvitkovic et al., 2019; Allamanis et al., 2018; Alon et al., 2018, 2019b,a), and execution-based (Wang and Su, 2019). Token-based approaches represent the code as a sequence of tokens at the lexical level, and apply *n*-gram models or deep neural network models to predict program properties. AST-based approaches represent the code as a sequence of AST nodes or paths at the syntax level. However, these two types of approaches characterize the code syntax, but it is difficult for them to learn code semantics. To better learn code semantics, Wang and Su (2019) proposed an execution-based method to learn code representations from symbolic and concrete execution traces at the semantics level. While achieving promising accuracy in predicting program properties, this method may suffer scalability issues in collecting execution traces. In summary, *the key challenge for learning-based program semantic property prediction is how to represent code in a semantically enhanced way such that code semantics can be learned through code representation learning.*

To address this challenge, we propose Meth2Seq, a hybrid code representation learning approach, to encode a method as a sequence of distributed vectors. Meth2Seq is designed based on the following insights: (i) program dependence graph (i.e., control/data flows) captures semantic structures of a method; (ii) intermediate representation (IR) expresses a method as a sequence of statements of limited types, and each type implicitly carries certain operational semantics; (iii) natural language comment summarizes the high-level semantics of a method; and (iv) program dependence graph, intermediate representation, and natural language comment capture different aspects of the method semantics. Therefore, we represent a method as (1) a bag of paths on the program dependence graph, (2) a sequence of typed IR statements based on the context free grammar that describes the operational semantics of each IR statement type, and (3) a sentence of natural language comment. Then, we apply a neural network on these representations to generate a sequence of vectors. The learned sequence of vectors is fed to a decoder model to predict method names. We focus on the task of method name prediction in this work because (i) method names are of particular importance as they provide the cornerstone of abstraction for developers to communicate with each other for various purposes (e.g., code review and program comprehension) (Høst and Østvold, 2009) and (ii) method name prediction is the difficult task in program semantic property prediction (Alon et al., 2019b).

To demonstrate the effectiveness of Meth2Seq, we compared it against two state-of-the-art code representation learning approaches (i.e., Code2Vec Alon et al., 2019b and Code2Seq Alon et al., 2019a) as well as two state-of-the-art method name prediction approaches (i.e., LiuName[1] (Liu et al., 2019) and HeMa (Jiang et al., 2019)) with a dataset of 280.5K methods in 67 Java projects. Our experimental results have demonstrated that Meth2Seq can significantly outperform Code2Vec and Code2Seq in F1-score respectively by 92.6% and 36.6%, and also significantly outperform LiuName and HeMa in F1-score respectively by 85.6% and 178.1%. Besides, F1-score respectively decreases by 26.2%, 33.5% and 17.7% after we exclude program dependence graph, intermediate representation and comment from our hybrid code representation. Thus, each of the three code representations contributes to the achieved effectiveness of Meth2Seq.

In summary, this paper makes the following contributions.

- We proposed a hybrid code representation learning approach, named Meth2Seq, to embed methods based on a mixture of three code representations, i.e., a bag of paths on the program dependence graph, a sequence of typed IR statements, and a sentence of natural language comment.
- We evaluated Meth2Seq on the task of method name prediction with a dataset of 280.5K methods in 67 Java projects, and it significantly outperformed state-of-the-art code representation learning approaches and method name prediction approaches in F1-score.

2. Preliminaries on encoder–decoder model

As program property prediction is usually formulated as a machine translation problem, we briefly introduce the encoder–decoder model that our approach is built upon.

Encoder–Decoder Model. Encoder–decoder model has shown promising performance in various fields, especially in machine translation field (Cho et al., 2014; Sutskever et al., 2014). Sequence-to-sequence model (Cho et al., 2014) has become the state-of-art method on a wide range of translation tasks. Given a sentence $x = (x_1, x_2, \ldots, x_{T_x})$ of the source language, it generates a sentence $y = (y_1, y_2, \ldots, y_{T_y})$ of the target language. To achieve this goal, the encoder computes a hidden state h_t at each time step t as $f(h_{t-1}, x_t)$; i.e., f is a non-linear function that maps x_t to a hidden state h_t by integrating the previous hidden state h_{t-1}. The last hidden state h_{T_x} can be used as a context vector c to the decoder. The decoder predicts the word y_t at time step t as $g(h_t, y_{t-1}, c)$; i.e., g is a non-linear function that predicts the output depending on the context vector c from the encoder, the previous output y_{t-1}, and the hidden state of the decoder h_t. During training, the model parameters are learned to maximize the conditional log-likelihood (Cho et al., 2014).

Attention. While sequence-to-sequence model has gained huge successes, it suffers poor performance on translating long sentences because it is very likely that the model "forgets" the long term dependency information in previous words if the sentence grows. Then, attention mechanism (Luong et al., 2015) is proposed to address this problem. It makes the model only pay attention to relevant words in source sentence. Specifically, at each time stamp t in decoding, a context vector c_t is computed through weighing over each hidden state in the encoder. Thus, the current word y_t is predicted by $g(h_t, y_{t-1}, c_t)$.

Transformer. The transformer model uses only attention as the model's encoder and decoder. The insight behind this model is: if attention performs so good when it is placed between encoder and decoder, it should be used inside encoder and decoder too. As shown in the left part of Fig. 1, the encoder takes each word's embedding as input, and generates as output two attention vectors for each word. The attention vectors characterize the relationship between the corresponding word and all input words. The encoder has four layers. The self-attention layer computes two attention vectors for each input word, and the feed-forward layer applies a non-linear mapping. Each of the two layers are followed by an add and norm layer which applies a residual connection and normalization to speed up training and prevent over-fitting. The decoder shares the similar structure to the encoder. However, the self-attention layer in the decoder also takes as input the encoder's output. The masked self-attention operates differently from the encoder's self-attention. It takes as input only the previous words of the predicted word, and masks the words after the

[1] For the ease of reference, we named their approach as LiuName.

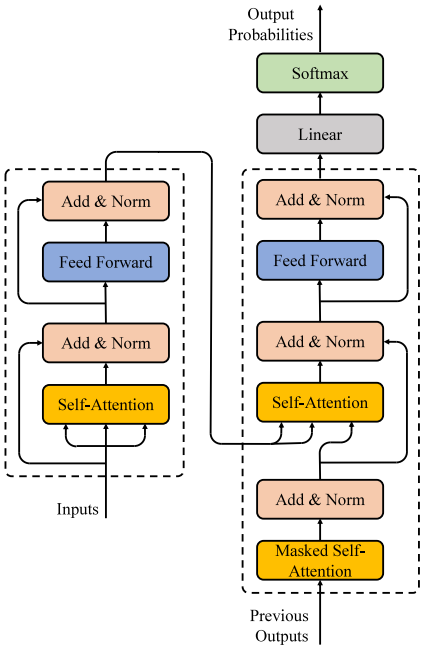

Fig. 1. The architecture of the transformer model.

predicted word in order to prevent the model to "see" the word before it actually predicts it. At last, the decoder's output is put in a linear and softmax layer to compute each word's probability at the current time stamp.

3. Methodology

We first present our approach overview. Then, we explain how we represent a method, learn the code representation, and use the representation for predicting method names.

3.1. Approach overview

At a high level, our goal is to represent methods in such a way that captures semantic information, enables learning across a massive collection of methods, and can be used to predict method names. To this end, our main idea is to use the code semantic structures in program dependence graph (PDG), the implicit operational semantics in intermediate representation (IR), and the explicit summarized semantics in natural language comment. In particular, control and data flows in PDG respectively characterize the execution order of statements and the data definition and usage among statements. Statements in IR have a limited number of types, and each type of IR statement is an operation that carries simple operational semantics. Compared with AST, both PDG and IR provide a higher abstraction of code. As a higher abstraction requires capturing more semantic information of code (Zhao and Huang, 2018), PDG and IR are more powerful in capturing code semantics than AST. Besides, method comment offers a good source of semantics summarized by developers. Hence, these three code representations capture code semantics from different perspectives, whose integration helps to better characterize code semantics.

Based on these insights, as shown in Fig. 2, we propose Meth2Seq to encode a method body as a sequence of distributed vectors. It represents a method body as a bag of PDG paths (Section 3.2), a sequence of typed IR statements (Section 3.3), and a sentence of method comment (Section 3.4). Then, it first applies positional encoding and a fully connected layer to initialize each PDG path, each typed IR statement and the method comment as a vector, and uses positional encoding and Transformer encoder to generate a sequence of vectors for each method (Section 3.5). The learned sequence of vectors can be fed to a Transformer decoder to predict method names (Section 3.6).

It is worth mentioning that our approach is general and can be extended to support other programming languages and intermediate representations, we focus on Java programs in this paper and use Jimple (Vallee-Rai and Hendren, 1998) as the IR.

3.2. Representing method as a bag of PDG paths

As shown in the top part of Fig. 2, our method representation with PDG has two steps, i.e., PDG construction and PDG path representation. We will elaborate each step below.

PDG Construction. Given the method corpus in projects, we construct a PDG for each method. The PDG of each method is denoted as a 5-tuple $\mathcal{G} = \langle \mathcal{N}, \mathcal{C}, \mathcal{D}, n_e, \iota \rangle$, where $\mathcal{N}$ denotes a set of nodes, and each node $n \in \mathcal{N}$ denotes a (Jimple) statement; $\mathcal{C} : \mathcal{N} \times \mathcal{N}$ denotes a set of edges, and each edge $c = n_1 \mapsto n_2 \in \mathcal{C}$ denotes a control flow between n_1 and n_2; $\mathcal{D} : \mathcal{N} \times \mathcal{N}$ denotes a set of edges, and each edge $d = n_1 \hookrightarrow n_2 \in \mathcal{D}$ denotes a data flow between n_1 and n_2; n_e denotes the exit node; and ι denotes a function that maps each node $n \in \mathcal{N}$ to a (Jimple) statement $\iota(n)$. In particular, We use $edge(n) \subseteq \mathcal{C} \cup \mathcal{D}$ to denote the out edges from n. We use Soot (Vallée-Rai et al., 1999) to compute the control flows and data flows for each method at the Jimple statement level, and then construct $\mathcal{G}$.

Example 3.1. Fig. 3 shows an example of a Java method; and Fig. 4 presents its PDG generated by Soot with the option *use-original-names* enabled. The black arrows denote control flows and the red arrows denote data flows. It has eight Jimple statements, $n_1, n_2, \ldots, n_8$; and n_8 is the exit statement. n_1 obtains the

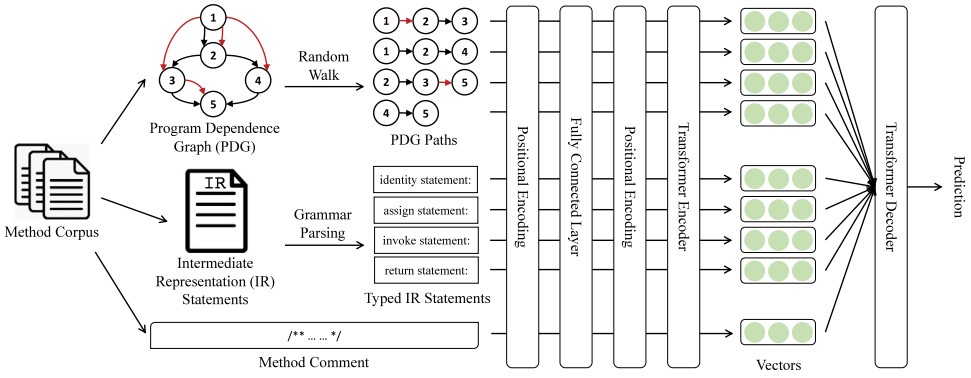

Fig. 2. Overall of our approach.

```
1  protected String randomAvailableHostname() {
2      List<InetAddress> available = info.nodeList();
3      Collections.shuffle(available);
4      return available.get(0).getHostName();
5  }
```

Fig. 3. An example of a method.

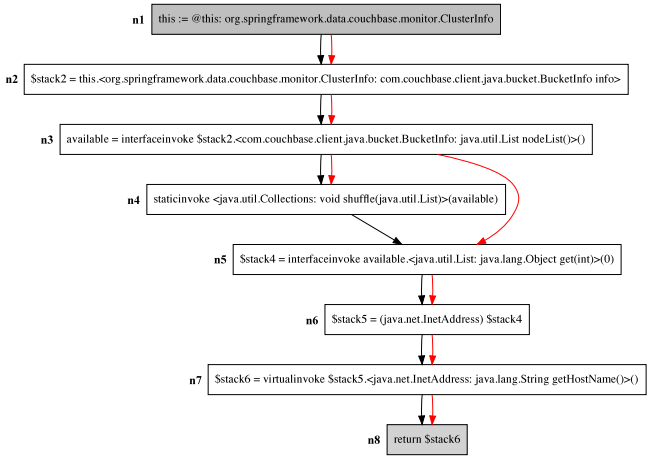

Fig. 4. The PDG of the method in Fig. 3.

`this` variable of the residing class of the method in Fig. 3. n_2 uses `this` to get the field `info` and assigns it to `stack2`. n_3 invokes `nodeList` on `stack2`, and assigns its return value to `available`. n_4 invokes a static method `shuffle` that takes `available` as the argument. n_5 invokes `get` on `available`, and assigns the return value to `stack4`. n_6 casts the type of `stack4`, and assigns it to `stack5`. n_7 invokes `getHostName` on `stack5`, and assigns the return value to `stack6`. Finally, n_8 returns `stack6`.

PDG Path Representation. A PDG path in $\mathcal{G}$ of a length at most k is denoted as a sequence $p = n_1, e_1, \ldots, n_{l-1}, e_{l-1}, n_l$, where $2 \leq l \leq k$; for each $i \in [1, l]$, $n_i \in \mathcal{N}$ is a Jimple statement; and for each $i \in [1, l-1]$, $e_i \in \mathcal{C} \cup \mathcal{D}$ is a control or data flow between n_i and n_{i+1}. We use $flow(e_i) \in \{control, data\}$ to denote the flow type of e_i, use $n_i \mapsto n_{i+1}$ to denote n_i, e_i, n_{i+1} if $flow(e_i) = control$, and use $n_i \hookrightarrow n_{i+1}$ to denote n_i, e_i, n_{i+1} if $flow(e_i) = data$. We use a bag of PDG paths to represent a method body as it has several advantages. First, PDG paths can capture diverse local semantic structure information. Second, PDG paths transform the non-sequential graph structure into sequences, and thus reduce the efficiency of representation learning. Third, PDG paths are automatically generated and are applicable to other programming languages.

To automatically generate a PDG path from $\mathcal{G}$, we use a random walk strategy. It first chooses a node from $\mathcal{N}$ uniformly at random as the current node n, and then works in iterations until the maximum length k is reached or the exit node n_e is reached. In each iteration, it selects an edge from n's out edges $edge(n)$ uniformly at random, and uses the edge's target node as the current node to continue the iteration. To generate a bag of PDG paths $\mathcal{P}$, we apply the previous procedure for multiple times. As PDGs for different methods have different size, we limit the number of generated PDG paths and the maximum length of a PDG path with respect to the size of $\mathcal{N}$. For each method, we generate $b \times |\mathcal{N}|$ PDG paths of length at most $\frac{|\mathcal{N}|}{a} + 1$. Hence, the larger the PDG, the more PDG paths are generated to capture the semantic structure more completely, and the more local semantic structure information is captured in a PDG path.

Example 3.2. Given the PDG in Fig. 4, if we set a to 2, then a PDG path has the maximum length of 5. If our generation starts at n_2, a possible PDG path is $p_1 = n_2 \mapsto n_3 \hookrightarrow n_5 \mapsto n_6 \mapsto n_7$, and if starts at n_5, a possible PDG path is $p_2 = n_5 \hookrightarrow n_6 \mapsto n_7 \hookrightarrow n_8$.

We regard each PDG path $p \in \mathcal{P}$ as a sentence in two steps. First, for $p = n_1, e_1, \ldots, n_{l-1}, e_{l-1}, n_l$, we represent each node n_i ($i \in [1, l]$) as its corresponding Jimple statement $\iota(n_i)$, and each edge e_i ($i \in [1, l-1]$) as the flow type $flow(e_i)$; i.e., p is represented as $\iota(n_1)flow(e_1)\ldots\iota(n_{l-1})flow(e_{l-1})\iota(n_l)$. Then, we use tokenizing, splitting according to camel notation, removing non-alphabet characters and locals (e.g., `stack2`), and transforming to lower cases to pre-process the previous representation. Finally, we have $|\mathcal{P}|$ sentences for each method.

Example 3.3. n_7 in Fig. 4 is represented as a sub-sentence of "virtualinvoke java net inet address java lang string get host name", the data flow edge between n_3 and n_5 is represented as a word of "data", while the control flow edge between n_4 and n_5 is represented as a word of "control".

3.3. Representing method as a sequence of typed IR statements

As shown in the middle part of Fig. 2, our method representation with IR is composed of two steps, i.e., IR generation and typed IR representation. We will elaborate each step below.

IR Generation. As we use Jimple as the IR, we run Soot (Vallée-Rai et al., 1999) to generate the Jimple representation, i.e., a sequence of Jimple statements, for each method. Jimple is a typed and compact 3-address code representation for Java bytecode (Vallee-Rai and Hendren, 1998). Statements are restricted to the least number of operands (two in most cases, but possibly more for some method invocation statements), and the operands must either be constants or local variables; and local variables must be explicitly declared and typed.

Example 3.4. The Jimple representation of the method in Fig. 3 is actually already showed in Fig. 4, i.e., the eight statements in the eight nodes. Type information is contained in statements; e.g., in n_2, `stack2` is of type `BucketInfo`, and in n_3, `available` is of type `List`.

Typed IR Representation. The underlying operational semantics of Jimple statements are valuable, but are not explicitly but are implicitly reflected and thus are hard to be learned. For example, it is difficult to learn the operational semantics of the Jimple statement in n_6 although it is obvious for human developers to known that it is a cast operation. Fortunately, in Jimple, there are a total of 15 types of Jimple statements with the operational semantics well-defined in its context free grammar (Vallee-Rai and Hendren, 1998). A context free grammar contains a set of production rules. Each production rule is composed of a left-hand side and right-hand side. The left-hand side is a non-terminal symbol, and the right-hand side is a sequence of non-terminal and terminal symbols. Based on the context free grammar, each Jimple statement is derived via a set of production rules by recursively rewriting each non-terminal symbol in the left-hand side of a production rule until there is no non-terminal symbol. In that sense, Jimple statements consist of terminal symbols that lose the type information of non-terminal symbols where the terminal symbols come. For example, n_6 contains a cast expression but we lose this semantic information after the derivation.

Based on this observation, we try to add the non-terminal (or type) information for each Jimple statement according to the context free grammar of Jimple, in order to explicitly express the underlying operational semantics of each statement. Particularly, for each statement, we first add the non-terminal

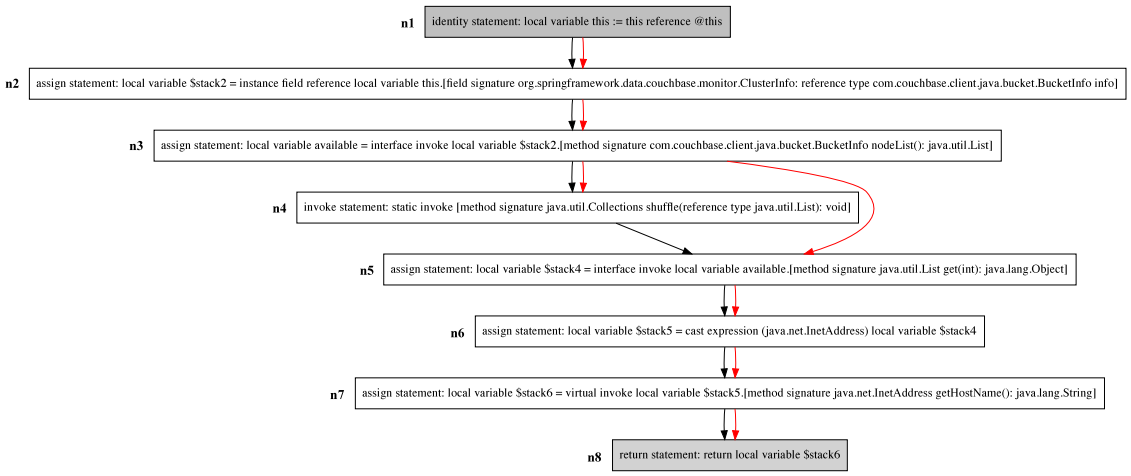

Fig. 5. Typed IR statements of the method in Fig. 3.

symbol that represents the type of the statement (e.g., assign statement); then we parse each Jimple statement according to the context free grammar and add for each terminal symbol the non-terminal symbol where it comes. Based on these two rules, we automatically transform each Jimple statement into typed Jimple statements which have more naturalness (Hindle et al., 2012) than original Jimple statements and hence can ease the burden of deep learning in extracting semantics.

Example 3.5. Fig. 5 presents the generated typed Jimple statements based on the original Jimple statements in Fig. 4. n_6 is of type `JAssignStmt`, which could assign a *rvalue* to a local, or an *immediate* (a local or a constant) to a static field, to an instance field or to an array reference according to the context free grammar. n_6 is an assign statement with its *rvalue* being a cast expression. Compared with the original n_6 in Fig. 4, it has the type information of the statement (i.e., assign statement), the type information of `stack2` (i.e., local variable), the type information of the enclosed expression (i.e., cast expression), and the type information of `stack4` (i.e., local variable). Obviously, the semantics expressed in n_6 in Fig. 5 is implied by n_6 in Fig. 4, and can hard to be learned from n_6 in Fig. 4.

We regard each typed Jimple statement as a sentence, and use the same pre-processing steps as in method representation with PDG (see Section 3.2). Finally, we have $|\mathcal{N}|$ sentences for each method.

3.4. Representing method as a sentence of comment

As shown in the bottom part of Fig. 2, our method representation with comment is simply to use the first sentence of the method comment because the first sentence is often a summary of a method according to the JavaDoc guidance.[2] We use common strategies (i.e., tokenizing, removing stop words, and stemming) to pre-process the extracted sentence. Finally, we have one sentence of comment for each method. Notice that for method without comments, we leave the comment empty with the intuition that as long as the comment is available, we should use any available semantic information from it.

[2] https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

3.5. Learning code representation in a hybrid way

After we represent each method as $|\mathcal{P}|$ sentences for PDG paths, $|\mathcal{N}|$ sentences for typed IR statements, and one sentence for method comment, we define the input vocabulary as the words in these $|\mathcal{P}| + |\mathcal{N}| + 1$ sentences for each method in corpus, and randomly initialize the embedding of each word which will be later learned simultaneously with the neural network during training. For each of the $|\mathcal{P}| + |\mathcal{N}| + 1$ sentences for a method, we first use positional encoding to encode the position information of the words in each PDG path, each typed IR statement, and the method comment. Then, we use a fully connected layer to combine the embedding of each word in this sentence into a vector. Here we also try a position-wise feed forward layer and a one-dimensional convolution kernel. It turns out that position-wise feed forward layer and fully connected layer have significant advantages over convolution kernel, while position-wise feed forward layer is only slightly better than fully connected layer. Considering the model complexity, we finally choose fully connected layer as the fusion layer. Then, the $|\mathcal{P}| + |\mathcal{N}| + 1$ vectors are sequentially fed into positional encoding to encode the position information (especially useful for the vectors for typed IR statements) and then the Transformer encoder (see Section 2) to generate a sequence of $|\mathcal{P}|+|\mathcal{N}|+1$ attention vectors that represent a method in a hybrid way.

3.6. Predicting method names

Using the sequence of attention vectors generated by Meth2Seq, we use a Transformer decoder (see Section 2) to achieve the task of method name prediction. To construct the output vocabulary, we extract method names and use them as the output vocabulary.

To train the network, we use cross-entropy loss between the predicted probability distribution and the true probability distribution. To use the trained network for method name prediction, we get PDG paths, typed IR statements and comment for the method under prediction, and then put them to Meth2Seq to predict one token at a time until "EOS" (denoting end of sentence) is outputted. Finally, the token list (except for the last "EOS" token) predicted is used as the method name.

Example 3.6. For the method in Fig. 3, Meth2Seq accurately predicts the token list as "random", "available", "host", "name" and "EOS". This shows that Meth2Seq captures the semantics underlying the shuffle operation on node list.

4. Evaluation

We have implemented METH2SEQ in 11.8K lines of Python and Java code, using Soot for static analysis and PyTorch for deep learning. We have released the code of METH2SEQ at our website[3] with the dataset used in our evaluation.

4.1. Evaluation setup

We compared our approach with state-of-the-art code representation learning approaches and method name prediction approaches, and analyzed the contribution of code representations in METH2SEQ, using 67 GitHub Java projects. Our evaluation is designed to answer the following research questions.

- **RQ1:** What is the effectiveness of METH2SEQ in predicting method names, compared with state-of-the-art approaches?
- **RQ2:** What is the contribution of the three code representations in METH2SEQ to the achieved effectiveness?
- **RQ3:** What is the sensitivity of METH2SEQ's effectiveness to the number and length of PDG paths?

Dataset. As our approach needs to apply static analysis at the Jimple representation, we need to ensure that selected projects can be successfully compiled. Due to this restriction, the existing datasets used in the literature (Allamanis et al., 2016; Alon et al., 2018, 2019b,a) cannot be used. We attempted to manually compile GitHub Java projects, but found that most compilation problems were caused by library dependency. Hence, we decided to leverage Travis CI to automatically build the projects. Finally, we created a dataset of 67 highly-starred Java projects, and applied Soot to generate the PDG and IR of each method. Following previous works (e.g., Allamanis et al., 2016 and Liu et al., 2019), we removed main methods, constructor methods, empty methods, test methods and example methods, and created a corpus of 280.5K methods. We split our method dataset into training, validation and testing dataset by 7:1:2.

State-of-the-Art Approaches. We selected CODE2VEC (Alon et al., 2019b) and CODE2SEQ (Alon et al., 2019a) as the state-of-the-art code representation learning approaches because they have been empirically demonstrated to outperform the existing code representation learning approaches (e.g., Alon et al., 2018) on the task of method name prediction, and they also outperform previous method name prediction approaches (Allamanis and Sutton, 2013; Allamanis et al., 2015a, 2016). Notice that both CODE2VEC and CODE2SEQ are AST-path based approaches. We did not compared METH2SEQ with the recent work DYPRO (Wang and Su, 2019) as it is not open-sourced and it requires dynamic execution traces which are not easy to obtain. Furthermore, we also selected two recent method name prediction approaches, i.e., LIUNAME (Liu et al., 2019) and HEMA (Jiang et al., 2019). Specifically, LIUNAME also uses a deep learning-based approach but represents code as tokens, while HEMA relies on heuristic rules to predict method names and demonstrates better effectiveness than CODE2VEC.

Evaluation Metrics. Following prior works (Allamanis et al., 2016; Alon et al., 2018, 2019b,a), we used four metrics to measure the effectiveness of method name prediction over case-insensitive sub-tokens. The first metric is *exact match accuracy*, i.e., the percent of method names predicted exactly; e.g., given a method name `totalCount`, a prediction of `total_count` is considered as exactly matched. The other three metrics are *precision*, *recall* and *F1-score*. For example, given a method name `totalCount`, a prediction of `total` has full precision but low recall, while a prediction of `totalUserCount` has full recall but low precision.

Training Configuration. We set all the layers inside the encoder and decoder identical in shape with 256 hidden units, and the 256 hidden units are equally divided into 8 heads in the self-attention layer. Our model is trained via Adafactor optimizer with $beta_1 = 0.9$, $beta_2 = 0.98$ and $epsilon = 10^{-9}$. In the training process, the learning rate starts from an initial value of 0 and varies along each training step. It increases linearly in the first warmup=4,000 steps and decreases proportionally with the step number thereafter. To prevent over-fitting, we add a dropout in each self-attention head with a probability of 50%. For CODE2VEC, CODE2SEQ, LIUNAME and HEMA, we directly use the model parameter settings described in their original work. We set the training epochs to 100 and the training batch size to 128 for all the approaches. Notice that the training has converged (i.e., reach the best performance) within 100 epochs for all the approaches. All models are trained on a machine with a Nvidia Titan X GPU and a 2.4GHZ CPU with 16 cores and 128G memory.

Table 1
Quantitative comparative results in four metrics.

Approach	Exact	Precision	Recall	F1
CODE2VEC	0.296	0.407	0.399	0.403
CODE2SEQ	0.329	0.626	0.520	0.568
LIUNAME	0.227	0.457	0.385	0.418
HEMA	0.200	0.443	0.204	0.279
METH2SEQ	**0.582**	**0.778**	**0.778**	**0.776**

4.2. Quantitative study (RQ1)

Table 1 presents the results of CODE2VEC, CODE2SEQ, LIUNAME, HEMA and METH2SEQ with respect to the four metrics. Overall, METH2SEQ significantly outperformed CODE2VEC, CODE2SEQ, LIUNAME and HEMA in all the four metrics, which demonstrates that leveraging PDG paths, typed IR statements and method comment to encode method code can better capture code semantics as code has well-defined semantics that would be difficult to capture by only considering source code tokens or syntactical AST paths. CODE2SEQ achieved better performance than CODE2VEC as it uses LSTMs to encode AST paths node-by-node rather than monolithically. LIUNAME had similar performance to CODE2VEC but worse performance than CODE2SEQ because it only treats code as tokens. Besides, HEMA, a rule-based approach, achieved the worst performance.

Numerically, METH2SEQ outperformed CODE2SEQ in exact match accuracy by 76.9%. Predicting the exact same name to the original method name is challenging because method names are often project-specific, and developers may use semantically equivalent or slightly misleading method names. METH2SEQ significantly outperformed CODE2SEQ in precision by 24.3% and recall by 49.6%, and hence had an improvement of 36.6% in F1-score. Moreover, METH2SEQ had a more balanced result for precision and recall which are both important for method name prediction. Compared to CODE2VEC, METH2SEQ had 96.6% and 92.6% improvement in exact match accuracy and F1-score, respectively. These results indicate that METH2SEQ can capture code semantics much better than the existing state-of-the-art code representation learning approaches. Besides, METH2SEQ improved exact match accuracy and F1-score over LIUNAME by 156.4% and 85.6%, and over HEMA by 191.0% and 178.1%, showing the superior advantage over token-based or rule-based approaches.

As prior works (Fu and Menzies, 2017; Hellendoorn and Devanbu, 2017; Liu et al., 2018; Menzies et al., 2018) suggest that it is a good practice to look into the data when leveraging deep

[3] https://meth2seq.github.io/meth2seq/

Table 2
Get/Set methods in all and exact match predictions.

Approach	All Prediction (%)	Exact Prediction (%)
Code2Vec	30.9	44.5
Code2Seq	39.3	56.1
LiuName	35.9	39.6
HeMa	32.7	38.1
Meth2Seq	34.6	37.4

Table 3
Code representation contribution analysis.

Approach	Exact	Precision	Recall	F1
Meth2Seq	**0.582**	**0.778**	**0.778**	**0.776**
Meth2Seq w/o PDG	0.475	0.571	0.585	0.573
Meth2Seq w/o IR	0.431	0.513	0.530	0.516
Meth2Seq w/o Comment	0.534	0.638	0.648	0.639
Meth2Seq w/o Type in IR	0.462	0.569	0.565	0.561
Meth2Seq with only PDG	0.414	0.472	0.509	0.489
Meth2Seq with only IR	0.460	0.548	0.541	0.544

learning on programming language and software engineering tasks. Hence, we manually analyzed all exact match predictions in the five approaches, and found that methods starting with "get" or "set" accounted for a high percentage. This is also observed by Jiang et al. (2019), and motivates one of their rules to generate get/set methods. This result motivated us to systematically analyzed the percentage of get/set methods in the testing data, in all predictions, and in all exact match predictions. It turns out that there are around 30% of get/set methods in the testing data. Table 2 reports the percentage of get/set methods in prediction results. We can observe that the five approaches had a similar percentage of get/set methods in all their predictions. However, in the exact match predictions, 44.5% and 56.1% were get/set methods in Code2Vec and Code2Seq, and 39.6%, 38.1% and 37.4% were get/set methods in LiuName, HeMa and Meth2Seq, which were much lower than Code2Vec and Code2Seq. These results indicate that Meth2Seq is more diverse in predicting method names, and it is also worthwhile to combine AST-path approaches.

> Meth2Seq significantly outperformed two state-of-the-art code representation learning approaches, Code2Vec and Code2Seq, in exact match accuracy and F1-score respectively by 96.6% and 92.6% and by 76.9% and 36.6%. Meth2Seq also significantly outperformed two state-of-the-art method name prediction approaches, LiuName and HeMa, in exact match accuracy and F1-score respectively by 156.4% and 85.6% and by 191.0% and 178.1%.

4.3. Ablation study (RQ2)

We conducted an ablation study to understand the contribution of various settings in code representations in Meth2Seq to the achieved effectiveness, and thus ran six experiments.

Removing PDG Path Representation. We removed the PDG path representation, and only represented methods as a sequence of typed IR statements and method comment. The result is listed in the third row of Table 3. We can observe that after removing PDG paths from our model, the performance of Meth2Seq decreased in exact match accuracy by about 18% and in precision, recall and F1-score by more than 24%. It shows that PDG paths significantly contribute to the semantics characterization.

Removing IR Representation. We removed the typed IR representation, and only represented methods as a bag of PDG paths and method comment. The result is shown in the fourth row of Table 3. It turns out that after removing IR statements from our model, Meth2Seq had a larger performance degradation than in the previous setting, i.e., around 25% decrease in exact match accuracy and more than 31% decrease in precision, recall and F1-score. It indicates that typed IR statements have more contribution to capture code semantics than PDG paths.

Removing Comment Representation. We removed the available method comments, and only represented methods as a bag of PDG paths and a sequence of typed IR statements. The result is shown in the fifth row of Table 3. After removing comments, Meth2Seq had a slight performance degradation (i.e., 8%) in exact match accuracy but a more than 16% decrease in precision, recall and F1-score. This indicates that comments do contribute to the semantics understanding in Meth2Seq but in a less significant way than PDG paths and typed IR statements.

Removing Type in IR Representation. We directly used the original Jimple statements rather than type IR statements for method IR representation. The result is shown in the sixth row of Table 3. After removing the type information in IR statements, the performance of Meth2Seq decreased in exact match accuracy by about 20% and in precision, recall and F1-score by more than 26%. It indicates that type information encoded by context free grammar does contribute to better reflect code semantics, and significantly improves the contribution of IR statements.

Only Keeping PDG Path Representation. We only represented methods as a bag of PDG paths. The result is shown in the seventh row of Table 3. We can see that the performance of Meth2Seq decreased in exact match accuracy by about 29% and in precision, recall and F1-score by more than 34%. It indicates that PDG path alone cannot effectively capture code semantics.

Only Keeping IR Representation. We only represented methods as a sequence of typed IR statements. The result is shown in the last row of Table 3. It can be observed that Meth2Seq had a smaller performance degradation than in the previous setting, i.e., around 21% decrease in exact match accuracy and more than 29% decrease in precision, recall and F1-score.

> PDG paths, typed IR statements and method comment all contribute to the semantics understanding in Meth2Seq; and typed IR statements make the most contribution.

4.4. Sensitivity analysis (RQ3)

Our method representation with PDG paths relies on two parameters: the maximum length of PDG paths (i.e., $\frac{|\mathcal{N}|}{a}+1$) and the number of PDG paths (i.e., $b \times |\mathcal{N}|$). In Sections 4.2 and 4.3, we reported the results when a and b was set to 4 and 8. We analyzed Meth2Seq's sensitivity to them.

Sensitivity to the Maximum Length of PDG Paths. We set a to 1, 2, 4 and 8 and fixed b to 8, and ran the experiments. The result is presented in Table 4. When a was smaller than 4, the performance of Meth2Seq degraded in all the four metrics. As a decreases, the maximum length of a PDG path increases. However, a long PDG path might fail to capture local semantics but cause noises and pollute relevant information. On the other hand, when a was larger than 4, the performance of Meth2Seq also gradually decreased in all the four metrics. As a increases, the maximum length of a PDG path decreases. A short path may capture incomplete or fragmented semantics, and thus causes the performance degradation. Based on these results, we suggest to set a to 4.

Table 4
Sensitivity analysis of the maximum path length.

Path Length $= \frac{\lvert\mathcal{N}\rvert}{a} + 1$	Exact	Precision	Recall	F1
a = 1	0.545	0.736	0.740	0.738
a = 2	0.562	0.755	0.753	0.754
a = 4	**0.582**	**0.778**	**0.778**	**0.776**
a = 8	0.569	0.762	0.759	0.760

Table 5
Sensitivity analysis of the number of paths.

Path Number $= b \times \lvert\mathcal{N}$	Exact	Precision	Recall	F1
b = 2	0.538	0.734	0.732	0.733
b = 4	0.554	0.715	0.743	0.729
b = 6	0.565	0.746	0.753	0.750
b = 8	**0.582**	**0.778**	**0.778**	**0.776**

Sensitivity to the Number of PDG Paths. We set b to 2, 4, 6 and 8 and fixed a to 4, and ran the experiments. The result is reported in Table 5. We can see that, as b increased, the performance of Meth2Seq also gradually increased in almost all the four metrics. This indicates that a larger number of PDG paths can capture code semantics more completely.

> A relatively short or long PDG path can degrade the performance of Meth2Seq. A large number of PDG paths can help to improve the performance of Meth2Seq.

4.5. Qualitative study

Apart from the quantitative study in Section 4.2, we report some interesting cases to illustrate Meth2Seq's capability.

For the method in Fig. 3, Meth2Seq accurately predicted its name. Code2Vec predicted "get" and "hostname", Code2Seq predicted "get" and "host", LiuName predicted "get", "host" and "name", and HeMa predicted "to" and "string", all failing to precisely capture and reflect the semantics.

Fig. 6 shows a method where Meth2Seq achieved an exact match, while Code2Vec predicted "convert", Code2Seq predicted "get", "message" and "converter", LiuName predicted "init", "default" and "converters", and HeMa failed to predict as there was no matched heuristic rule. The code at Line 5–9 gets message converters of a specific type. Meth2Seq successfully captured this code semantics, and predicted "for" and "type" as the last two tokens for this method name.

Fig. 7 shows a method where Meth2Seq correctly predicted the first token of the method name. Meth2Seq predicted "convert", Code2Vec predicted "decode", Code2Seq predicted "get", "request" and "content", LiuName predicted "copy", "to", "byte" and "array", and HeMa predicted "to", "byte" and "array". Code2Seq fails to capture the underlying action, Meth2Seq and Code2Vec fail to capture the action target, and LiuName HeMa directly use tokens in the code.

Fig. 8 shows an example that Meth2Seq predicted the incorrect name. Meth2Seq predicted "add". However, "add" is semantically equivalent to "put" in this case. Predicting exact names is difficult as developers may use semantically equivalent names. In that sense, the results in Section 4.2 are an underapproximation of Meth2Seq's capability to predict meaningful method names. Code2Vec predicted "to" and "json", Code2Seq predicted "add" and "payload", LiuName predicted "parse", and HeMa predicted "get". All are not accurate.

Fig. 9 shows an example that two methods have different implementations but have similar functionality and thus the same method name. Specifically, Line 3–4 and Line 10–12 in the first and second method prepares a query, while Line 6 and Line 14 in the first and second method returns a list of all query results. Meth2Seq correctly predicted "find" and "all" for both methods. Code2Vec predicted "execute" for the first method, and "sort" for the second. Code2Seq predicted "find" and "pageable" for the first

```
1  public MessageConverter getMessageConverterForType(MimeType mimeType) {
2    List<MessageConverter> converters = new ArrayList<>();
3    for (MessageConverter converter : this.converters) {
4      if (converter instanceof AbstractMessageConverter) {
5        for (MimeType type : ((AbstractMessageConverter) converter)
6            .getSupportedMimeTypes()) {
7          if (type.includes(mimeType)) {
8            converters.add(converter);
9          }
10       }
11     }
12     else {
13       if (this.log.isDebugEnabled()) {
14         this.log.debug("Ommitted " + converter + " of type "
15             + converter.getClass().toString() + " for '"
16             + mimeType.toString()
17             + "' as it is not an AbstractMessageConverter");
18       }
19     }
20   }
21   if (CollectionUtils.isEmpty(converters)) {
22     throw new ConversionException(
23         "No message converter is registered for " + mimeType.toString());
24   }
25   if (converters.size() > 1) {
26     return new CompositeMessageConverter(converters);
27   }
28   else {
29     return converters.get(0);
30   }
31 }
```

Fig. 6. An example of exact match prediction.

```
1  private byte[] convertContent(Object content) {
2    if (content instanceof String) {
3      return ((String) content).getBytes();
4    }
5    else if (content instanceof byte[]) {
6      return (byte[]) content;
7    }
8    else if (content instanceof File) {
9      return copyToByteArray((File) content);
10   }
11   else if (content instanceof InputStream) {
12     return copyToByteArray((InputStream) content);
13   }
14   else if (content == null) {
15     return new byte[0];
16   }
17   else {
18     throw new IllegalStateException(
19         "Unsupported request content: " + content.getClass().getName());
20   }
21 }
```

Fig. 7. An example of first-token match prediction.

```
1  public final CouchbaseList put(final Object value) {
2      verifyValueType(value);
3      payload.add(value);
4      return this;
5    }
```

Fig. 8. An example of incorrect prediction.

```
1  public Page<T> findAll(Predicate predicate, Pageable pageable) {
2    Assert.notNull(pageable, "Pageable must not be null!");
3    final JPQLQuery<?> countQuery = createCountQuery(predicate);
4    JPQLQuery<T> query = querydsl.applyPagination(pageable, createQuery(
           predicate).select(path));
5
6    return PageableExecutionUtils.getPage(query.fetch(), pageable, countQuery::
           fetchCount);
7  }
8
9  public <S extends T> Page<S> findAll(Example<S> example, Pageable pageable) {
10   ExampleSpecification<S> spec = new ExampleSpecification<>(example,
           escapeCharacter);
11   Class<S> probeType = example.getProbeType();
12   TypedQuery<S> query = getQuery(new ExampleSpecification<>(example,
           escapeCharacter), probeType, pageable);
13
14   return isUnpaged(pageable) ? new PageImpl<>(query.getResultList()) :
           readPage(query, probeType, pageable, spec);
15 }
```

Fig. 9. An example of semantically similar methods.

method, and predicted correctly for the second. LiuName predicted "find" and "query" for the first method, and "get", "result" and "list" for the second. HeMa predicted "get" and "pageable" for the first method, and failed to predict for the second due to no matched rule.

Fig. 10 gives an example that two methods have similar code but different functionality and thus different method names. Line 3–11 in the first method and Line 23–31 in the second method are exactly the same, preparing a query statement by an id. The key difference is their return statement at Line 12 and Line 32: the former returns one of the fetched results, and the latter returns all fetched results. Meth2Seq captured this difference, and correctly predicted the method names. Code2Vec also had a correct prediction, Code2Seq respectively predicted "get", "all" and "id", and "get", "all" and "mapper" for the two methods, LiuName predicted "get" and "maps", and "get", "mapped" and "object" for them, while HeMa predicted "get", "topology" and "conf", and "supports" for them.

Besides, we report a method, as shown in Fig. 11, where Meth2Seq failed to predict the correct name. This method verifies whether the type of a value is a certain and supported type. Meth2Seq predicted "is" and "value", failing to understand the code semantics. In fact, the method call at Line 7 reflects the semantics of this method. However, Meth2Seq failed to capture it. One potential reason is that our code representation only considers intra-procedural data/control flows but does not consider inter-procedural data/control flows. Therefore, one potential improvement is to leverage call graph information in code representation learning. Notice that Code2Vec predicted "get", "class" and "type", Code2Seq predicted "get" and "value", LiuName predicted "equals", and HeMa predicted "value" and "of". All are not accurate.

```
1  public Mono<T> findById(ID id) {
2    Assert.notNull(id, "Id must not be null!");
3    List<String> columns = this.accessStrategy.getAllColumns(this.entity.
           getJavaType());
4    String idColumnName = getIdColumnName();
5
6    StatementMapper mapper = this.accessStrategy.getStatementMapper().forType(
           this.entity.getJavaType());
7    StatementMapper.SelectSpec selectSpec = mapper.createSelect(this.entity.
           getTableName()) //
8        .withProjection(columns) //
9        .withCriteria(Criteria.where(idColumnName).is(id));
10
11   PreparedOperation<?> operation = mapper.getMappedObject(selectSpec);
12   return this.databaseClient.execute(operation).as(this.entity.getJavaType())
           .fetch().one();
13 }
14
15 public Flux<T> findAllById(Publisher<ID> idPublisher) {
16   Assert.notNull(idPublisher, "The Id Publisher must not be null!");
17   return Flux.from(idPublisher).buffer().filter(ids -> !ids.isEmpty()).
           concatMap(ids -> {
18
19     if (ids.isEmpty()) {
20       return Flux.empty();
21     }
22
23     List<String> columns = this.accessStrategy.getAllColumns(this.entity.
             getJavaType());
24     String idColumnName = getIdColumnName();
25
26     StatementMapper mapper = this.accessStrategy.getStatementMapper().forType
             (this.entity.getJavaType());
27     StatementMapper.SelectSpec selectSpec = mapper.createSelect(this.entity.
             getTableName()) //
28         .withProjection(columns) //
29         .withCriteria(Criteria.where(idColumnName).in(ids));
30
31     PreparedOperation<?> operation = mapper.getMappedObject(selectSpec);
32     return this.databaseClient.execute(operation).as(this.entity.getJavaType
             ()).fetch().all();
33   });
34 }
```

Fig. 10. An example of semantically different methods.

```
1  private void verifyValueType(final Object value) {
2    if (value == null) {
3      return;
4    }
5
6    final Class<?> clazz = value.getClass();
7    if (simpleTypeHolder.isSimpleType(clazz)) {
8      return;
9    }
10
11   throw new IllegalArgumentException("Attribute of type "
12     + clazz.getCanonicalName() + "can not be stored and must be converted.");
13 }
```

Fig. 11. An example of failed prediction.

4.6. Discussion

We discuss the threats to our evaluation and the potential applications of our approach.

Threats. One threat to our evaluation is the size of the dataset, when compared with the dataset used in AST path-based approaches (Allamanis et al., 2016; Alon et al., 2018, 2019b,a). As our approach relies on heavyweight static analysis to generate PDG and Jimple representations, we selected 67 projects that could be successfully built by Travis CI as our dataset. We believe that the current promising results are convincing enough to indicate the capability of static analysis in capturing code semantics for predicting method names. In fact, our dataset is actually much larger than the deep learning-based works that use control flow graphs (e.g., 28K methods for code search Wan et al., 2019, and 9 projects for bug detection Li et al., 2019b). We argue that it is the step we must take to apply program analysis on "big code" for better program comprehension. We have released our dataset at our website. To the best of our knowledge, this is the first dataset of program methods with PDGs and Jimple representations. We are also continuously enlarging this dataset,

and investigating the possibility of using automatic repository building techniques (Hassan et al., 2017) to ease the enlarging.

Another threat to our evaluation is that we only evaluated the effectiveness of Meth2Seq on the task of method name prediction. In fact, we deliberately picked this task because method name prediction is difficult in the sense that it needs to have an accurate and summarized semantic understanding of the potentially large method body. The same strategy is also used by prior work (Alon et al., 2019b), where they also claimed that "succeeding in [method name prediction] would suggest that the code vector has indeed accurately captured the functionality and semantic role of the method". We are applying Meth2Seq to other program property prediction tasks such as variable name prediction to empirically demonstrate the generality of our approach over different tasks.

Applications. We design Meth2Seq for the task of method name prediction. We believe that the overall idea of Meth2Seq that uses a hybrid code representation is general and can be used in more applications, e.g., method comment generation, semantic clone detection, and code search. Moreover, we are applying this idea to "big code" driven code change comprehension; i.e., we generate the PDG for the methods before and after changes, and use differencing algorithms to extract PDG changes as the representation of code changes.

Furthermore, we are exploring other orthogonal possibilities to improve Meth2Seq. First, we are investigating other possibilities to learn from PDGs. Graph embedding techniques (Cai et al., 2018) can be a potential way to directly learn from graphs. However, the learning efficiency might be reduced. Second, we are investigating visualization techniques to explain which PDG paths, typed IR statements or method comment contribute to the prediction. Third, we currently use random walk, inspired by DeepWalk (Perozzi et al., 2014), to generate representative PDG paths. We ran our approach for five times, and found no significant difference. We plan to investigate how to systematically generate PDG paths and whether such systematic approach can achieve better performance. Last but not the least, we believe that each code representation at the lexical, syntactical and semantic level has its merit in understanding code semantics. Hence, we plan to combine lexical tokens and AST paths into our approach to investigate whether it can further improve the performance.

5. Related work

We review and discuss the most closely related work in predicting program semantic properties and deep learning for programming language and software engineering.

5.1. Predicting program semantic properties

Advances have been recently made to predict program semantic properties (e.g., method names, variable names, and variable types) from a massive collection of programs (Yahav, 2015; Bielik et al., 2015; Ernst, 2017; Allamanis et al., 2018b).

Allamanis and Sutton (2013) learned a *n*-gram model to predict method names, variable names and variable types. Allamanis et al. (2015a) learned a log-bilinear neural context model to suggest method names and class names. While their neural context model can capture long-distance context that is difficult to be captured by a *n*-gram model, it contains a set of hard-coded features that hinder the generality. To improve the generality, Allamanis et al. (2016) proposed a convolutional attention neural network to predict method names. Different from previous approaches that represent code as tokens at the lexical level, Alon et al. (2018) proposed to represent code as AST paths at the syntax level, and achieved a higher accuracy in predicting method names, variable names and variable types. Then, Alon et al. (2019b,a) improved their previous work (Alon et al., 2018) by adopting an attention mechanism to learn the weights over AST paths and using LSTMs to represent AST paths node-by-node, and achieved a higher accuracy in predicting method names. Bui et al. (2021) adapted the idea of self-supervised training and developed a pre-training task on code ASTs to learn code representation. Their output embeddings can be used for multiple downstream tasks. They achieved a comparable result with Code2Seq on the task of method name prediction. Wang and Su (2019) learned code representations from symbolic and concrete execution traces at the semantics level. While achieving higher accuracy in predicting method names, their approach might suffer scalability in collecting execution traces. Our approach also represents code at the semantic level but uses the semantics in program dependence graph and intermediate representation which are easier to obtain. Recently, Sui et al. (2020) proposed a new code embedding approach that precisely preserves interprocedural program dependence (a.k.a value-flows) and embeds control-flows and alias-aware data-flows of a program in a low-dimensional vector space, and applied it to method name prediction and code classification. Differently, instead of only relying on control and data flows, we further consider IR and comment to achieve a hybrid code semantic representation learning.

Allamanis et al. (2014) leveraged *n*-gram model to suggest identifier names. They first retrieved a set of candidate names that have occurred in the same context in other code snippets, and then measured the naturalness by a learned *n*-gram model to rank candidate names. Raychev et al. (2015) modeled relations among variables and program elements as a dependency network, and predicted variable names with conditional random fields. Vasilescu et al. (2017) recovered variable names through statistical machine translation, while Tran et al. (2019) leveraged information retrieval to recover variable names. Recently, deep learning techniques are applied to predict variable names by representing code as tokens (Bavishi et al., 2018; Liu et al., 2019) or graphs (i.e., ASTs augmented with semantic relationships between AST nodes) (Cvitkovic et al., 2019; Allamanis et al., 2018). Most of them rely on lexical or syntactic knowledge of code, and hence it is difficult for them to learn code semantics, hidering their effectiveness in predicting variable names. Note that naming variables often relies on relatively local context, while naming methods is more difficult as it often requires non-local information from method bodies (Allamanis et al., 2015a). This is also why we select method naming as the application scenario of our approach. Besides, a number of advances have been made to detect naming inconsistencies (Lee et al., 2021; Arnaoudova et al., 2013; Høst and Østvold, 2009; Binkley et al., 2011; Kim and Kim, 2016; Deissenboeck and Pizka, 2006; Lawrie et al., 2006a). Our approach can be applied to detect naming inconsistencies and suggest method names for identified inconsistencies.

Phan et al. (2018) adopted statistical machine translation to learn fully qualified names for API elements. Hellendoorn et al. (2018) and Malik et al. (2019) applied a sequence-to-sequence model to infer types, respectively using code tokens and natural language information in code. It is also difficult for these approaches to learn code semantics, while our approach uses program dependence graph and intermediate representation to better capture the semantics of a method.

After reviewing these approaches, we can observe a trend that code is first represented at the lexical level and then at the syntax level. Following this trend, we represent code at the semantic level by a hybrid combination of program dependence graph, intermediate representation and natural language comments. It is also interesting to further investigate the effectiveness of a hybrid combination across these representation levels (i.e., lexical, syntax and semantic levels).

It is worth mentioning that Yefet et al. (2020) introduced a novel approach for attacking trained neural models of code using adversarial examples, i.e., forcing a given trained model to make an incorrect prediction. To defend a model from suck attacks, they also proposed several defense strategies to drop the success rate of attacker and thus improve the model robustness. It is interesting to investigate whether our model can be attacked by their approach.

5.2. Deep learning for PL and SE

Apart from program semantic property prediction, an increased interest has been attracted in learning a probabilistic model of code from a corpus for various programming language and software engineering tasks (Yahav, 2015; Bielik et al., 2015; Ernst, 2017; Allamanis et al., 2018b). The previous successful attempts to model code range from using *n*-gram models (Hsiao et al., 2014; Campbell et al., 2014; Nguyen et al., 2013a; Karaivanov et al., 2014; Nguyen et al., 2013b; Tu et al., 2014; Hindle et al., 2012; Oda et al., 2015; Franks et al., 2015) to graph-based object usage model (Nguyen et al., 2009, 2018b, 2012; Nguyen and Nguyen, 2015; Nguyen et al., 2016b) to probabilistic grammars (Allamanis and Sutton, 2014; Bielik et al., 2016; Allamanis et al., 2018a; Raychev et al., 2016). Recent advances in deep learning techniques have triggered a new wave of applying them to address various programming language and software engineering tasks, e.g., defect prediction (Wang et al., 2016; Li et al., 2017b; Yang et al., 2015; Hoang et al., 2019), bug localization (Lam et al., 2015; Huo et al., 2016; Lam et al., 2017; Xiao et al., 2019; Huo et al., 2019; Seidel et al., 2017; Li et al., 2019), bug detection (Murali et al., 2017; Li et al., 2018b; Pradel and Sen, 2018; Habib and Pradel, 2019; Zhou et al., 2019; Li et al., 2019b; Li et al., 2021), bug fixing (Tufano et al., 2018a; Wang et al., 2018), syntax error fixing (Pu et al., 2016; Gupta et al., 2017; Bhatia et al., 2018; Mesbah et al., 2019; Gupta et al., 2019), code clone detection (White et al., 2016; Li et al., 2017a; Büch and Andrzejak, 2019; Zhao and Huang, 2018; Tufano et al., 2018b; Zhang et al., 2019), code migration (Nguyen et al., 2016a, 2017; Gu et al., 2017; Nguyen et al., 2018a), code generation (Sun et al., 2019; Quirk et al., 2015; Yin and Neubig, 2017; Rabinovich et al., 2017; Desai et al., 2016; Gvero and Kuncak, 2015; Chen et al., 2018; Sun et al., 2020), code completion (Brockschmidt et al., 2018; Maddison and Tarlow, 2014; Li et al., 2018a; White et al., 2015; Raychev et al., 2014), code search (Allamanis et al., 2015b; Gu et al., 2018; Wan et al., 2019), code change learning (Tufano et al., 2019), API usage search (Gu et al., 2016; Van Nguyen et al., 2016; Li et al., 2020), API analogy mining (Henkel et al., 2018; Chen et al., 2021), API retrieval (Nguyen et al., 2018c), code classification (Peng et al., 2015; Mou et al., 2016; Bui et al., 2018; Ben-Nun et al., 2018; DQ et al., 2019; Zhang et al., 2019; Wang, 2019), function synonym detection (DeFreez et al., 2018), performance prediction (Ben-Nun et al., 2018), traceability linking (Ye et al., 2016; Guo et al., 2017; Xie et al., 2019), specification mining (Le and Lo, 2018), comment generation (Movshovitz-Attias and Cohen, 2013; Iyer et al., 2016; Wan et al., 2018; Hu et al., 2020, 2018a; Guerrouj et al., 2015; LeClair et al., 2019; Hu et al., 2018b), and commit message generation (Cortés-Coy et al., 2014; Loyola et al., 2017; Jiang et al., 2017). They exploit the availability of a code corpus and the naturalness of and regularities in code (Hindle et al., 2012).

Among these deep learning-based approaches, except for Zhao and Huang (2018), Tufano et al. (2018b), DeFreez et al. (2018), Ben-Nun et al. (2018), Henkel et al. (2018), Wang et al. (2018), Wang (2019), Zhou et al. (2019), Li et al. (2019b), Wan et al. (2019) and Li et al. (2021), they represent code at the level of tokens, AST nodes, AST subtrees, AST paths or grammar rules, characterizing the code syntax but often having difficulty in learn code semantics. To better learn code semantics, Tufano et al. (2018b) investigated the effectiveness of different code representations (i.e., tokens, AST nodes, control flow graph, and bytecode) in detecting clones, and found that these code representations are complementary to each other. Inspired by this work, we use two code representations (i.e., program dependence graph and IR statements) to encode a method and use a neural network to combine the code representations for the problem of method name prediction. DeFreez et al. (2018) represented a method as paths in its inter-procedure control flow graph, and learned a method embedding where vectors for method synonyms are close. Ben-Nun et al. (2018) represented code as contextual flow graphs. Zhao and Huang (2018) represented code as a semantic matrix that encodes control flow and data flow information, and developed a neural network model to measure code similarity based on semantic matrices. We also use control and data flows, but further combine them with IR to represent a method. Zhou et al. (2019), Li et al. (2019b) and Wan et al. (2019) attempt to combine AST, control flow and data flow, while we intentionally only combine different semantic code representations (i.e., PDG, IR and comment) to reflect their advantage over syntax representations. Similarly, Li et al. (2021) used PDG to extract semantic representations of vulnerabilities and code tokens to extract syntax representations of vulnerabilities for the task of vulnerability detection. Differently, we combine different semantic code representations for the task of method name prediction. It is interesting to further consider AST representation in our approach. Henkel et al. (2018) represented code as abstracted symbolic execution traces for learning word embedding. Wang et al. (2018), Wang (2019) proposed a dynamic program embedding that is learned from dynamic concrete execution traces. While execution traces can capture deep and precise program semantics, these approaches may suffer scalability in collecting symbolic or concrete execution traces.

6. Conclusions

We have proposed Meth2Seq to encode a method as a sequence of distributed vectors based on a hybrid combination of code representations, i.e., a bag of PDG paths, a sequence of typed IR statements, and method comment. The learned sequence of vectors of a method is fed to a decoder model to predict method names. Our evaluation results with a dataset of 280.5K methods in 67 Java projects have indicated that Meth2Seq outperforms the two state-of-the-art code representation learning approaches in F1-score by 92.6% and 36.6%, while outperforming two state-of-the-art method name prediction approaches in F1-score by 85.6% and 178.1%. Our evaluation has also demonstrated the significant contribution of PDG paths, typed IR statements and method comment to the effectiveness of Meth2Seq. In future, we plan to explore potentials of learning from combined code representations.

CRediT authorship contribution statement

Fengyi Zhang: Methodology, Software, Writing - original draft. **Bihuan Chen:** Conceptualization, Supervision, Writing - review & editing. **Rongfan Li:** Data curation, Investigation, Validation. **Xin Peng:** Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 61802067).

References

Abebe, S.L., Arnaoudova, V., Tonella, P., Antoniol, G., Gueheneuc, Y.-G., 2012. Can lexicon bad smells improve fault prediction? In: Proceedings of the 19th Working Conference on Reverse Engineering. pp. 235–244.

Allamanis, M., Barr, E.T., Bird, C., Devanbu, P., Marron, M., Sutton, C., 2018a. Mining semantic loop idioms. IEEE Trans. Softw. Eng. 44 (7), 651–668.

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

Allamanis, M., Barr, E.T., Bird, C., Sutton, C., 2015. Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 38–49.

Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C., 2018b. A survey of machine learning for big code and naturalness. ACM Comput. Surv. 51 (4), 81.

Allamanis, M., Brockschmidt, M., Khademi, M., 2018. Learning to represent programs with graphs. In: Proceedings of the 6th International Conference on Learning Representations.

Allamanis, M., Peng, H., Sutton, C., 2016. A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33rd International Conference on Machine Learning. pp. 2091–2100.

Allamanis, M., Sutton, C., 2013. Mining source code repositories at massive scale using language modeling. In: Proceedings of the 10th Working Conference on Mining Software Repositories. pp. 207–216.

Allamanis, M., Sutton, C., 2014. Mining idioms from source code. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 472–483.

Allamanis, M., Tarlow, D., Gordon, A., Wei, Y., 2015. Bimodal modelling of source code and natural language. In: Proceedings of the 32nd International Conference on Machine Learning. pp. 2123–2132.

Alon, U., Brody, S., Levy, O., Yahav, E., 2019. code2seq: Generating sequences from structured representations of code. In: Proceedings of the 7th International Conference on Learning Representations.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2018. A general path-based representation for predicting program properties. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 404–419.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. code2vec: Learning distributed representations of code. In: Proc. ACM Program. Lang. p. 40.

Arnaoudova, V., Di Penta, M., Antoniol, G., 2016. Linguistic antipatterns: What they are and how developers perceive them. Empir. Softw. Eng. 21 (1), 104–158.

Arnaoudova, V., Di Penta, M., Antoniol, G., Gueheneuc, Y.-G., 2013. A new family of software anti-patterns: Linguistic anti-patterns. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering. pp. 187–196.

Bavishi, R., Pradel, M., Sen, K., 2018. Context2Name: A deep learning-based approach to infer natural variable names from usage contexts. arXiv abs/:1809.05193.

Ben-Nun, T., Jakobovits, A.S., Hoefler, T., 2018. Neural code comprehension: a learnable representation of code semantics. In: Proceedings of the 32nd Conference on Neural Information Processing Systems. pp. 3585–3597.

Bengio, Y., Courville, A., Vincent, P., 2013. Representation learning: A review and new perspectives. IEEE Trans. Pattern Anal. Mach. Intell. 35 (8), 1798–1828.

Bhatia, S., Kohli, P., Singh, R., 2018. Neuro-symbolic program corrector for introductory programming assignments. In: Proceedings of the IEEE/ACM 40th International Conference on Software Engineering. pp. 60–70.

Bielik, P., Raychev, V., Vechev, M., 2015. Programming with" big code": Lessons, techniques and applications. In: Proceedings of the 1st Summit on Advances in Programming Languages. pp. 1–10.

Bielik, P., Raychev, V., Vechev, M., 2016. PHOG: probabilistic model for code. In: Proceedings of the 33rd International Conference on Machine Learning. pp. 2933–2942.

Binkley, D., Davis, M., Lawrie, D., Maletic, J.I., Morrell, C., Sharif, B., 2013. The impact of identifier style on effort and comprehension. Empir. Softw. Eng. 18 (2), 219–276.

Binkley, D., Hearn, M., Lawrie, D., 2011. Improving identifier informativeness using part of speech information. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 203–206.

Brockschmidt, M., Allamanis, M., Gaunt, A.L., Polozov, O., 2018. Generative code modeling with graphs. In: Proceedings of the 7th International Conference on Learning Representations.

Büch, L., Andrzejak, A., 2019. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. pp. 95–104.

Bui, N.D., Jiang, L., Yu, Y., 2018. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In: Proceedings of the Workshops At the 32nd AAAI Conference on Artificial Intelligence. pp. 758–761.

Bui, N.D.Q., Yu, Y., Jiang, L., 2021. InferCode: Self-supervised learning of code representations by predicting subtrees. In: Proceedings of the 43rd International Conference on Software Engineering.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2009. Relating identifier naming flaws and code quality: An empirical study. In: Proceedings of the 16th Working Conference on Reverse Engineering. pp. 31–35.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2010. Exploring the influence of identifier names on code quality: An empirical study. In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering. pp. 156–165.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2011. Mining java class naming conventions. In: 2011 27th IEEE International Conference on Software Maintenance. ICSM. pp. 93–102.

Cai, H., Zheng, V.W., Chang, K.C.-C., 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. IEEE Trans. Knowl. Data Eng. 30 (9), 1616–1637.

Campbell, J.C., Hindle, A., Amaral, J.N., 2014. Syntax errors just aren't natural: improving error reporting with language models. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 252–261.

Chen, C., Su, T., Meng, G., Xing, Z., Liu, Y., 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In: Proceedings of the 40th International Conference on Software Engineering. pp. 665–676.

Chen, C., Xing, Z., Liu, Y., Ong, K.L.X., 2021. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. IEEE Trans. Softw. Eng. 47 (3), 432–447.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. pp. 1724–1734.

Cortés-Coy, L.F., Linares-Vásquez, M., Aponte, J., Poshyvanyk, D., 2014. On automatically generating commit messages via summarization of source code changes. In: Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation. pp. 275–284.

Cvitkovic, M., Singh, B., Anandkumar, A., 2019. Open vocabulary learning on source code with a graph-structured cache. In: Proceedings of the 36th International Conference on Machine Learning. pp. 1475–1485.

DeFreez, D., Thakur, A.V., Rubio-González, C., 2018. Path-based function embedding and its application to error-handling specification mining. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 423–433.

Deissenboeck, F., Pizka, M., 2006. Concise and consistent naming. Softw. Qual. J. 14 (3), 261–282.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., Roy, S., et al., 2016. Program synthesis using natural language. In: Proceedings of the 38th International Conference on Software Engineering. pp. 345–356.

DQ, B.N., Yu, Y., Jiang, L., 2019. Bilateral dependency neural networks for cross-language algorithm classification. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. pp. 422–433.

Ernst, M.D., 2017. Natural language is a programming language: Applying natural language processing to software development. In: Proceedings of the 2nd Summit on Advances in Programming Languages. pp. 1–14.

Franks, C., Tu, Z., Devanbu, P., Hellendoorn, V., 2015. Cacheca: A cache language model based code suggestion tool. In: Proceedings of the 37th International Conference on Software Engineering, Vol. 2. pp. 705–708.

Fu, W., Menzies, T., 2017. Easy over hard: A case study on deep learning. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 49–60.

Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: Proceedings of the 40th International Conference on Software Engineering. pp. 933–944.

Gu, X., Zhang, H., Zhang, D., Kim, S., 2016. Deep API learning. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 631–642.

Gu, X., Zhang, H., Zhang, D., Kim, S., 2017. DeepAM: Migrate APIs with multi-modal sequence to sequence learning. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence. pp. 3675–3681.

Guerrouj, L., Bourque, D., Rigby, P.C., 2015. Leveraging informal documentation to summarize classes and methods in context. In: Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering. pp. 639–642.

Guo, J., Cheng, J., Cleland-Huang, J., 2017. Semantically enhanced software traceability using deep learning techniques. In: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering. pp. 3–14.

Gupta, R., Kanade, A., Shevade, S., 2019. Deep reinforcement learning for syntactic error repair in student programs. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence. pp. 930–937.

Gupta, R., Pal, S., Kanade, A., Shevade, S., 2017. Deepfix: Fixing common c language errors by deep learning. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence. pp. 1345–1351.

Gvero, T., Kuncak, V., 2015. Synthesizing Java expressions from free-form queries. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 416–432.

Habib, A., Pradel, M., 2019. Neural bug finding: A study of opportunities and challenges. arXiv abs/1906.00307.

Hassan, F., Mostafa, S., Lam, E.S., Wang, X., 2017. Automatic building of java projects in software repositories: A study on feasibility and challenges. In: ESEM. pp. 38–47.

Hellendoorn, V.J., Bird, C., Barr, E.T., Allamanis, M., 2018. 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. pp. 152–162.

Hellendoorn, V.J., Devanbu, P., 2017. Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 763–773.

Henkel, J., Lahiri, S.K., Liblit, B., Reps, T., 2018. 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. pp. 163–174.

Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P., 2012. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering. pp. 837–847.

Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N., 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In: Proceedings of the 16th International Conference on Mining Software Repositories. pp. 34–45.

Høst, E.W., Østvold, B.M., 2009. Debugging method names. In: Proceedings of the European Conference on Object-Oriented Programming. pp. 294–317.

Hsiao, C.-H., Cafarella, M., Narayanasamy, S., 2014. Using web corpus statistics for program analysis. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 49–65.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension. pp. 200–210.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2020. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. 25 (3), 2179–2217.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z., 2018. Summarizing source code with transferred API knowledge. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence. pp. 2269–2275.

Huo, X., Li, M., Zhou, Z.-H., 2016. Learning unified features from natural and programming languages for locating buggy source code. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence. pp. 1606–1612.

Huo, X., Thung, F., Li, M., Lo, D., Shi, S.-T., 2019. Deep transfer bug localization. IEEE Trans. Softw. Eng. (in press).

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2016. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics. pp. 2073–2083.

Jiang, S., Armaly, A., McMillan, C., 2017. Automatically generating commit messages from diffs using neural machine translation. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. pp. 135–146.

Jiang, L., Liu, H., Jiang, H., 2019. Machine learning based recommendation of method names: How far are we. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. pp. 602–614.

Karaivanov, S., Raychev, V., Vechev, M., 2014. Phrase-based statistical translation of programming languages. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. pp. 173–184.

Kim, S., Kim, D., 2016. Automatic identifier inconsistency detection using code dictionary. Empir. Softw. Eng. 21 (2), 565–604.

Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2015. Combining deep learning with information retrieval to localize buggy files for bug reports. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. pp. 476–481.

Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2017. Bug localization with combination of deep learning and information retrieval. In: Proceedings of the IEEE/ACM 25th International Conference on Program Comprehension. pp. 218–229.

Lawrie, D., Feild, H., Binkley, D., 2006. Syntactic identifier conciseness and consistency. In: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation. pp. 139–148.

Lawrie, D., Morrell, C., Feild, H., Binkley, D., 2006. What's in a name? A study of identifiers. In: Proceedings of the 14th IEEE International Conference on Program Comprehension. pp. 3–12.

Le, T.-D.B., Lo, D., 2018. Deep specification mining. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 106–117.

Le, Q., Mikolov, T., 2014. Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on Machine Learning. pp. 1188–1196.

LeClair, A., Jiang, S., McMillan, C., 2019. A neural model for generating natural language summaries of program subroutines. In: Proceedings of the 41st International Conference on Software Engineering. pp. 795–806.

Lee, S., Wu, R., Cheung, S.-C., Kang, S., 2021. Automatic detection and update suggestion for outdated api names in documentation. IEEE Trans. Softw. Eng. 47 (4), 653–675.

Li, L., Feng, H., Zhuang, W., Meng, N., Ryder, B., 2017. Cclearner: A deep learning-based clone detection approach. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution. pp. 249–260.

Li, J., He, P., Zhu, J., Lyu, M.R., 2017. Software defect prediction via convolutional neural network. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security. pp. 318–328.

Li, X., Jiang, H., Kamei, Y., Chen, X., 2020. Bridging semantic gaps between natural languages and APIs with word embedding. IEEE Trans. Softw. Eng. 46 (10), 1081–1097.

Li, X., Li, W., Zhang, Y., Zhang, L., 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 169–180.

Li, J., Wang, Y., Lyu, M.R., King, I., 2018. Code completion with neural attention and pointer networks. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence. pp. 4159–4165.

Li, Y., Wang, S., Nguyen, T.N., Van Nguyen, S., 2019b. Improving bug detection via context-based code representation learning and attention-based neural networks. Proc. ACM Program. Lang. 3 (OOPSLA), 162:1–162:30.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021. SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium.

Liblit, B., Begel, A., Sweetser, E., 2006. Cognitive perspectives on the role of naming in computer programs. In: Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group. p. 11.

Liu, K., Kim, D., Bissyandé, T.F., Kim, T., Kim, K., Koyuncu, A., Kim, S., Traon, Y.L., 2019. Learning to spot and refactor inconsistent method names. In: Proceedings of the 41st International Conference on Software Engineering. pp. 1–12.

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

Loyola, P., Marrese-Taylor, E., Matsuo, Y., 2017. A neural architecture for generating natural language descriptions from source code changes. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. pp. 287–292.

Luong, T., Pham, H., Manning, C.D., 2015. Effective approaches to attention-based neural machine translation. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. pp. 1412–1421.

Maddison, C., Tarlow, D., 2014. Structured generative models of natural source code. In: Proceedings of the 31st International Conference on Machine Learning. pp. 649–657.

Malik, R.S., Patra, J., Pradel, M., 2019. NL2Type: inferring JavaScript function types from natural language information. In: Proceedings of the 41st International Conference on Software Engineering. pp. 304–315.

Menzies, T., Majumder, S., Balaji, N., Brey, K., Fu, W., 2018. 500+ Times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In: Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories. pp. 554–563.

Mesbah, A., Rice, A., Johnston, E., Glorioso, N., Aftandilian, E., 2019. DeepDelta: Learning to repair compilation errors. In: Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013a. Efficient estimation of word representations in vector space. CoRR abs/1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J., 2013b. Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th International Conference on Neural Information Processing Systems. pp. 3111–3119.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence. pp. 1287–1293.

Movshovitz-Attias, D., Cohen, W.W., 2013. Natural language models for predicting programming comments. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics. pp. 35–40.

Murali, V., Chaudhuri, S., Jermaine, C., 2017. Bayesian specification learning for finding API usage errors. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering. pp. 151–162.

Nguyen, A.T., Nguyen, T.N., 2015. Graph-based statistical language model for code. In: Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering. pp. 858–868.

Nguyen, A.T., Nguyen, T.T., Nguyen, T.N., 2013. Lexical statistical machine translation for language migration. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 651–654.

Nguyen, T.D., Nguyen, A.T., Nguyen, T.N., 2016. Mapping API elements for code migration with vector representations. In: Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion. pp. 756–758.

Nguyen, T.T., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2013. A statistical semantic language model for source code. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 532–542.

Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J., Nguyen, T.N., 2012. Graph-based pattern-oriented, context-sensitive source code completion. In: Proceedings of the 34th International Conference on Software Engineering. pp. 69–79.

Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N., 2009. Graph-based mining of multiple object usage patterns. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 383–392.

Nguyen, T.D., Nguyen, A.T., Phan, H.D., Nguyen, T.N., 2017. Exploring API embedding for API usages and applications. In: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering. pp. 438–449.

Nguyen, A.T., Nguyen, T.D., Phan, H.D., Nguyen, T.N., 2018. A deep neural network language model with contexts for source code. In: Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. pp. 323–334.

Nguyen, T.T., Pham, H.V., Vu, P.M., Nguyen, T.T., 2016. Learning API usages from bytecode: a statistical approach. In: Proceedings of the 38th International Conference on Software Engineering. pp. 416–427.

Nguyen, A., Rigby, P., Nguyen, T., Palani, D., Karanfil, M., Nguyen, T., 2018. Statistical translation of English texts to API code templates. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution. pp. 194–205.

Nguyen, T., Tran, N., Phan, H., Nguyen, T., Truong, L., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2018. Complementing global and local contexts in representing API descriptions to improve API retrieval tasks. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 551–562.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., Nakamura, S., 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. pp. 574–584.

Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., Jin, Z., 2015. Building program vector representations for deep learning. In: Proceedings of the International Conference on Knowledge Science, Engineering and Management. pp. 547–553.

Pennington, J., Socher, R., Manning, C., 2014. Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. pp. 1532–1543.

Perozzi, B., Al-Rfou, R., Skiena, S., 2014. Deepwalk: Online learning of social representations. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 701–710.

Phan, H., Nguyen, H., Tran, N., Truong, L., Nguyen, A., Nguyen, T., 2018. Statistical learning of api fully qualified names in code snippets of online forums. In: Proceedings of the IEEE/ACM 40th International Conference on Software Engineering. pp. 632–642.

Pradel, M., Sen, K., 2018. DeepBugs: A learning approach to name-based bug detection. Proc. ACM Program. Lang. 2 (OOPSLA), 147:1–147:25.

Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R., 2016. sk_p: a neural program corrector for MOOCs. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity. pp. 39–40.

Quirk, C., Mooney, R., Galley, M., 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing. pp. 878–888.

Rabinovich, M., Stern, M., Klein, D., 2017. Abstract syntax networks for code generation and semantic parsing. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. pp. 1139–1149.

Raychev, V., Bielik, P., Vechev, M., Krause, A., 2016. Learning programs from noisy data. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 761–774.

Raychev, V., Vechev, M., Krause, A., 2015. Predicting program properties from big code. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 111–124.

Raychev, V., Vechev, M., Yahav, E., 2014. Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 419–428.

Seidel, E.L., Sibghat, H., Chaudhuri, K., Weimer, W., Jhala, R., 2017. Learning to blame: Localizing novice type errors with data-driven diagnosis. Proc. ACM Program. Lang. 1 (OOPSLA), 60:1–60:27.

Sui, Y., Cheng, X., Zhang, G., Wang, H., 2020. Flow2vec: Value-flow-based precise code embedding. Proc. ACM Program. Lang. 4 (OOPSLA).

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., Zhang, L., 2019. A grammar-based structural cnn decoder for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 7055–7062.

Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., Zhang, L., 2020. TreeGen: A tree-based transformer architecture for code generation. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence. pp. 8984–8991.

Sutskever, I., Vinyals, O., Le, Q.V., 2014. Sequence to sequence learning with neural networks. In: Proceedings of the 27th Annual Conference on Neural Information Processing Systems. pp. 3104–3112.

Takang, A.A., Grubb, P.A., Macredie, R.D., 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. J. Program. Lang. 4 (3), 143–167.

Tran, H., Tran, N., Nguyen, S., Nguyen, H., Nguyen, T.N., 2019. Recovering variable names for minified code with usage contexts. In: Proceedings of the 41st International Conference on Software Engineering. pp. 1165–1175.

Tu, Z., Su, Z., Devanbu, P., 2014. On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 269–280.

Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., Poshyvanyk, D., 2019. On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering. pp. 25–36.

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D., 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 832–837.

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D., 2018. Deep learning similarities from different representations of source code. In: Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories. pp. 542–553.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 1999. Soot: A Java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. p. 13.

Vallee-Rai, R., Hendren, L.J., 1998. Jimple: Simplifying Java bytecode for analyses and transformations. Technical Report, McGill University.

Van Nguyen, T., Nguyen, A.T., Nguyen, T.N., 2016. Characterizing API elements in software documentation with vector representation. In: Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion. pp. 749–751.

Vasilescu, B., Casalnuovo, C., Devanbu, P., 2017. Recovering clear, natural identifiers from obfuscated JS names. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering. pp. 683–693.

Wan, Y., Shu, J., Sui, Y., Xu, G., Zhao, Z., Wu, J., Yu, P., 2019. Multi-modal attention network learning for semantic source code retrieval. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. pp. 13–25.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S., 2018. Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 397–407.

Wang, K., 2019. Learning scalable and precise representation of program semantics. arXiv abs/:1905.05251.

Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: Proceedings of the IEEE/ACM 38th International Conference on Software Engineering. pp. 297–308.

Wang, K., Singh, R., Su, Z., 2018. Dynamic neural program embeddings for program repair. In: Proceedings of the 6th International Conference on Learning Representations.

Wang, K., Su, Z., 2019. Learning blended, precise semantic program embeddings. CoRR abs/1907.02136.

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

White, M., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., 2015. Toward deep learning software repositories. In: Proceedings of the 12th Working Conference on Mining Software Repositories. pp. 334–345.

Xiao, Y., Keung, J., Bennin, K.E., Mi, Q., 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. Inf. Softw. Technol. 105, 17–29.

Xie, R., Chen, L., Ye, W., Li, Z., Hu, T., Du, D., Zhang, S., 2019. DeepLink: A code knowledge graph based deep learning approach for issue-commit link recovery. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. pp. 434–444.

Yahav, E., 2015. Programming with "Big Code". In: Proceedings of the Asian Symposium on Programming Languages and Systems. pp. 3–8.

Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J., 2015. Deep learning for just-in-time defect prediction. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security. pp. 17–26.

Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C., 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering. pp. 404–415.

Yefet, N., Alon, U., Yahav, E., 2020. Adversarial examples for models of code. Proc. ACM Program. Lang. 4 (OOPSLA).

Yin, P., Neubig, G., 2017. A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. pp. 440–450.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: Proceedings of the 41st International Conference on Software Engineering. pp. 783–794.

Zhao, G., Huang, J., 2018. Deepsim: deep learning code functional similarity. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 141–151.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proceedings of the 32nd Annual Conference on Neural Information Processing Systems. pp. 10197–10207.

Fengyi Zhang is a Ph.D. student at the School of Computer Science in Fudan University, China. His research interests include software engineering and big data analysis.

Bihuan Chen received his Ph.D. degree in Computer Science from Fudan University in 2014. He was a postdoctoral research fellow in Nanyang Technological University from 2014 to 2017. He is an associate professor in Fudan University, China. His research interests lie in software engineering, focusing on big code analysis, program analysis, software testing, and software security.

Rongfan Li is a master student at the School of Computer Science in Fudan University, China. His research interests include software engineering and big code analysis.

Xin Peng received his Ph.D. degree in Computer Science from Fudan University in 2006. He is a full professor at the School of Computer Science in Fudan University, China. His current research interests include big code analysis, intelligent software development, software maintenance and evolution, and mobile computing.